

User and Reference Manual: All Parts

Release 3.2.1

13 July 2006

Internal Release 3.2.1-I-21

Contents

I	General Introduction	1
1	General Introduction	3
1.1	License Issues	3
1.2	Third Party Software	5
1.3	Advanced	6
1.4	Namespace CGAL	6
1.5	Inclusion Order of Header Files	7
1.6	Compile-time Flags to Control Inlining	7
1.7	Checks	7
II	Kernels	11
2	2D and 3D Kernel	13
2.1	Introduction	13
2.2	Kernel Representations	14
2.3	Kernel Geometry	17
2.4	Predicates and Constructions	18
2.5	Extensible Kernel	21
2.6	Kernel Related Tools	27
	Reference Manual	31
2.7	Concepts	35
2.8	Kernel Classes and Operations	41

2.9	Predefined Kernels	57
2.10	Kernel Objects	60
2.11	Constants and Enumerations	123
2.12	Global Functions	135
2.13	Kernel Function Object Concepts	207
2.14	Tag Classes	426
3	dD Kernel	427
3.1	Introduction	427
3.2	Kernel Representations	428
3.3	Kernel Geometry	431
3.4	Predicates and Constructions	432
	Reference Manual	435
3.5	Linear Algebra Concepts and Classes	437
3.6	Kernel Objects	447
3.7	Global Kernel Functions	477
3.8	Kernel Concept	499
4	2D Circular Kernel	533
4.1	Introduction	533
4.2	Software Design	533
4.3	Examples	534
4.4	Design and Implementation History	538
	Reference Manual	541
4.5	Geometric Concepts	541
4.6	Algebraic Concepts	542
4.7	Geometric Kernels and Classes	542
4.8	Algebraic Kernel and Classes	543
4.9	Traits Classes for CGAL Arrangements	543
4.10	Alphabetical List of Reference Pages	543

III	Convex Hull Algorithms	607
5	2D Convex Hulls and Extreme Points	609
5.1	Introduction	609
5.2	Convex Hull	610
5.3	Example using Graham-Andrew's Algorithm	610
5.4	Extreme Points and Hull Subsequences	611
5.5	Traits Classes	611
5.6	Convexity Checking	612
	Reference Manual	613
5.7	Classified Reference Pages	613
5.8	Alphabetical List of Reference Pages	615
6	3D Convex Hulls	655
6.1	Introduction	655
6.2	Static Convex Hull Construction	655
6.3	Incremental Convex Hull Construction	657
6.4	Dynamic Convex Hull Construction	659
	Reference Manual	661
6.5	Classified Reference Pages	661
6.6	Alphabetical List of Reference Pages	663
7	dD Convex Hulls and Delaunay Triangulations	681
7.1	Introduction	681
7.2	dD Convex Hull	681
7.3	Delaunay Triangulation	682
	Reference Manual	683
7.4	Classified Reference Pages	683
7.5	Alphabetical List of Reference Pages	684

IV Polygons and Polyhedra	707
8 2D Polygons	709
8.1 Introduction	709
8.2 Example	709
Reference Manual	713
8.3 Classified Reference Pages	713
8.4 Alphabetical List of Reference Pages	714
9 2D Polygon Partitioning	735
9.1 Introduction	735
9.2 Monotone Partitioning	735
9.3 Convex Partitioning	736
Reference Manual	737
9.4 Classified Reference Pages	737
9.5 Alphabetical List of Reference Pages	738
10 3D Polyhedral Surfaces	781
10.1 Introduction	781
10.2 Definition	782
10.3 Example Programs	782
10.4 File I/O	789
10.5 Extending Vertices, Halfedges, and Facets	790
10.6 Advanced Example Programs	792
Reference Manual	797
10.7 Classified Reference Pages	797
10.8 Alphabetical List of Reference Pages	798
11 Halfedge Data Structures	835
11.1 Introduction	835
11.2 Software Design	836

11.3	Example Programs	837
	Reference Manual	845
11.4	Classified Reference Pages	845
11.5	Alphabetical List of Reference Pages	846
V	Polygon and Polyhedron Operations	893
12	2D Regularized Boolean Set-Operations	895
12.1	Introduction	895
12.2	Boolean Set-Operations on Linear Polygons	896
12.3	Boolean Set-Operations on General Polygons	906
	Reference Manual	915
12.4	Classified Reference Pages	915
12.5	Alphabetical List of Reference Pages	916
13	2D Boolean Operations on Nef Polygons	953
13.1	Introduction	953
13.2	Construction and Composition	954
13.3	Exploration	955
13.4	Traits Classes	956
13.5	Implementation	956
	Reference Manual	959
13.6	Classified Reference Pages	959
13.7	Alphabetical List of Reference Pages	959
14	2D Boolean Operations on Nef Polygons Embedded on the Sphere	981
14.1	Introduction	981
14.2	Restricted Spherical Geometry	982
14.3	Example Programs	983
	Reference Manual	989
14.4	Classified Reference Pages	989

14.5	Alphabetical List of Reference Pages	989
15	3D Boolean Operations on Nef Polyhedra	1011
15.1	Introduction	1011
15.2	Definition	1012
15.3	Infimaximal Box	1015
15.4	Regularized Set Operations	1015
15.5	Example Programs	1016
15.6	File I/O	1021
15.7	Further Example Programs	1022
15.8	Visualization	1026
	Reference Manual	1031
15.9	Classified Reference Pages	1031
15.10	Alphabetical List of Reference Pages	1032
16	2D Straight Skeleton and Polygon Offsetting	1057
16.1	Definitions	1057
16.2	Representation	1063
16.3	API	1066
16.4	Straight Skeletons, Medial Axis and Voronoi Diagrams	1071
16.5	Usages of the Straight Skeletons	1072
16.6	Straight Skeleton of a General Figure in the Plane	1072
	Reference Manual	1073
16.7	Classified Reference Pages	1073
16.8	Alphabetical List of Reference Pages	1073
VI	Arrangements	1101
17	2D Arrangements	1103
17.1	Introduction	1104
17.2	The Main Arrangement Class	1105

17.3	Issuing Queries on an Arrangement	1121
17.4	Free Functions in the Arrangement Package	1129
17.5	Traits Classes	1137
17.6	The Notification Mechanism	1163
17.7	Extending the DCEL	1167
17.8	Overlaying Arrangements	1172
17.9	Storing the Curve History	1177
17.10	Input/Output Functions	1183
17.11	Adapting to BOOST Graphs	1189
17.12	How To Speed Up Your Computation	1194
	Reference Manual	1197
17.13	Classified Reference Pages	1197
17.14	Alphabetical List of Reference Pages	1199
18	2D Intersection of Curves	1329
18.1	Introduction	1329
	Reference Manual	1333
18.2	Alphabetical List of Reference Pages	1333
19	2D Snap Rounding	1337
19.1	Introduction	1337
19.2	What is Snap Rounding/Iterated Snap Rounding	1338
19.3	Terms and Software Design	1338
19.4	Four Line Segment Example	1338
	Reference Manual	1341
19.5	Alphabetical List of Reference Pages	1341
VII	Triangulations and Delaunay Triangulations	1349
20	2D Triangulations	1351
20.1	Definitions	1352

20.2	Representation	1353
20.3	Software Design	1354
20.4	Basic Triangulations	1356
20.5	Delaunay Triangulations	1359
20.6	Regular Triangulations	1361
20.7	Constrained Triangulations	1364
20.8	Constrained Delaunay Triangulations	1365
20.9	Constrained Triangulations Plus	1367
20.10	The Triangulation Hierarchy	1368
20.11	Flexibility: Using Customized Vertices and Faces	1371
20.12	Design and Implementation History	1374
	Reference Manual	1375
20.13	Classified Reference Pages	1375
20.14	Alphabetical List of Reference Pages	1376
21	2D Triangulation Data Structure	1455
21.1	Definition	1455
21.2	The Concept of Triangulation Data Structure	1456
21.3	The Default Triangulation Data Structure	1457
	Reference Manual	1461
21.4	Classified Reference Pages	1461
21.5	Alphabetical List of Reference Pages	1462
22	3D Triangulations	1485
22.1	Representation	1486
22.2	Delaunay Triangulation	1487
22.3	Regular Triangulation	1488
22.4	Triangulation Hierarchy	1488
22.5	Software Design	1489
22.6	Examples	1493

22.7	Design and Implementation History	1500
	Reference Manual	1501
22.8	Classified Reference Pages	1501
22.9	Alphabetical List of Reference Pages	1502
23	3D Triangulation Data Structure	1561
23.1	Representation	1561
23.2	Software Design	1564
23.3	Examples	1567
23.4	Design and Implementation History	1570
	Reference Manual	1571
23.5	Classified Reference Pages	1571
23.6	Alphabetical List of Reference Pages	1572
24	2D Alpha Shapes	1599
24.1	Definitions	1600
24.2	Functionality	1600
24.3	Concepts and Models	1601
24.4	Examples	1601
	Reference Manual	1603
24.5	Classified Reference Pages	1603
24.6	Alphabetical List of Reference Pages	1604
25	3D Alpha Shapes	1619
25.1	Definitions	1621
25.2	Functionality	1622
25.3	Concepts and Models	1622
25.4	Examples	1623
	Reference Manual	1627
25.5	Classified Reference Pages	1628
25.6	Alphabetical List of Reference Pages	1628

VIII	Voronoi Diagrams	1645
26	2D Segment Delaunay Graphs	1647
26.1	Definitions	1647
26.2	Software Design	1649
26.3	The Geometric Traits	1652
26.4	The Segment Delaunay Graph Hierarchy	1653
26.5	Examples	1654
	Reference Manual	1659
26.6	Classified Reference Pages	1659
26.7	Alphabetical List of Reference Pages	1660
27	2D Apollonius Graphs (Delaunay Graphs of Disks)	1699
27.1	Definitions	1699
27.2	Software Design	1701
27.3	The Geometric Traits	1702
27.4	The Apollonius Graph Hierarchy	1705
27.5	Examples	1705
	Reference Manual	1711
27.6	Classified Reference Pages	1711
27.7	Alphabetical List of Reference Pages	1712
28	2D Voronoi Diagram Adaptor	1739
28.1	Introduction	1739
28.2	Software Design	1741
28.3	The Adaptation Traits	1742
28.4	The Adaptation Policy	1743
28.5	Examples	1745
	Reference Manual	1749
28.6	Classified Reference Pages	1749
28.7	Alphabetical List of Reference Pages	1750

IX	Meshing	1785
29	2D Conforming Triangulations and Meshes	1787
29.1	Conforming Triangulations	1787
29.2	Meshes	1791
	Reference Manual	1799
29.3	Classified Reference Pages	1799
29.4	Alphabetical List of Reference Pages	1799
30	3D Surface Mesher	1821
30.1	Introduction	1822
30.2	The Surface Mesher Interface	1822
30.3	Examples	1823
30.4	Meshing Criteria, Guarantees and Variations	1826
30.5	Design and Implementation History	1827
	Reference Manual	1829
30.6	Classified Reference Pages	1829
30.7	Alphabetical List of Reference Pages	1830
31	3D Surface Subdivision Methods	1867
31.1	Introduction	1867
31.2	Subdivision Method	1868
31.3	A Quick Example: Catmull-Clark Subdivision	1869
31.4	Catmull-Clark Subdivision	1870
31.5	Refinement Host	1873
31.6	Geometry Policy	1874
31.7	The Four Subdivision Methods	1876
31.8	Other Subdivision Methods	1877
	Reference Manual	1881
31.9	Classified Reference Pages	1881
31.10	Alphabetical List of Reference Pages	1881

32 Planar Parameterization of Triangulated Surface Meshes	1895
32.1 Introduction	1896
32.2 Basics	1897
32.3 Surface Parameterization Methods	1902
32.4 Sparse Linear Algebra	1908
32.5 Cutting a Mesh	1909
32.6 Output	1914
32.7 Complexity and Guarantees	1920
32.8 Software Design	1922
32.9 Extending the Package and Reusing Code	1926
Reference Manual	1929
32.10 Classified Reference Pages	1929
32.11 Alphabetical List of Reference Pages	1932
 X Search Structures	 2005
 33 2D Search Structures	 2007
33.1 Introduction	2007
33.2 Examples	2008
Reference Manual	2011
33.3 Classified Reference Pages	2011
33.4 Alphabetical List of Reference Pages	2011
 34 Interval Skip List	 2025
34.1 Definition	2025
34.2 Example Programs	2025
Reference Manual	2029
34.3 Classified Reference Pages	2029
34.4 Alphabetical List of Reference Pages	2029

35 dD Spatial Searching	2037
35.1 Introduction	2037
35.2 Splitting Rules	2039
35.3 Example Programs	2039
35.4 Software Design	2049
Reference Manual	2051
35.5 Classified Reference Pages	2051
35.6 Alphabetical List of Reference Pages	2052
 36 dD Range and Segment Trees	 2109
36.1 Introduction	2109
36.2 Definitions	2109
36.3 Software Design	2110
36.4 Creating an Arbitrary Multilayer Tree	2113
36.5 Range Trees	2113
36.6 Segment Trees	2115
Reference Manual	2119
36.7 Classified Reference Pages	2119
36.8 Alphabetical List of Reference Pages	2120
 37 Intersecting Sequences of dD Iso-oriented Boxes	 2147
37.1 Introduction	2147
37.2 Definition	2148
37.3 Software Design	2148
37.4 Minimal Example for Intersecting Boxes	2149
37.5 Example for Finding Intersecting 3D Triangles	2150
37.6 Example for Using Pointers to Boxes	2152
37.7 Example Using the <i>topology</i> and the <i>cutoff</i> Parameters	2152
37.8 Runtime Performance	2154
37.9 Example Using a Custom Box Implementation	2155

37.10	Example for Point Proximity Search with a Custom Traits Class	2157
37.11	Design and Implementation History	2158
	Reference Manual	2159
37.12	Classified Reference Pages	2159
37.13	Alphabetical List of Reference Pages	2160
XI	Geometric Optimization	2181
38	Geometric Optimization	2183
38.1	Bounding and Inscribed Volumes	2183
38.2	Optimal Distances	2186
38.3	Monotone and Sorted Matrix Search	2187
	Reference Manual	2189
38.4	Classified References Pages	2189
38.5	Alphabetical List of Reference Pages	2191
39	Principal Component Analysis	2335
39.1	Examples	2336
	Reference Manual	2341
39.2	Classified Reference Pages	2341
39.3	Alphabetical List of Reference Pages	2341
XII	Interpolation	2349
40	Interpolation	2351
40.1	Natural Neighbor Coordinates	2352
40.2	Surface Natural Neighbor Coordinates and Surface Neighbors	2355
40.3	Interpolation Methods	2357
	Reference Manual	2363
40.4	Classified Reference Pages	2363
40.5	Alphabetical List of Reference Pages	2364

41 2D Placement of Streamlines	2393
41.1 Definitions	2394
41.2 Fundamental Notions	2394
41.3 Farthest Point Seeding Strategy	2395
41.4 Implementation	2396
41.5 Examples	2396
Reference Manual	2399
41.6 Classified Reference Pages	2399
41.7 Alphabetical List of Reference Pages	2400
 XIII Kinetic Data Structures	 2411
42 Kinetic Data Structures	2413
42.1 An Overview of Kinetic Data Structures and Sweep Algorithms	2413
42.2 An Overview of the Kinetic Framework	2415
42.3 Using Kinetic Data Structures	2416
Reference Manual	2423
42.4 Classified Reference Pages	2423
42.5 Alphabetical List of Reference Pages	2424
 43 Kinetic Framework	 2461
43.1 Architecture	2461
43.2 Algebraic Kernel	2465
43.3 Examples	2466
Reference Manual	2473
43.4 Classified Reference Pages	2473
43.5 Alphabetical List of Reference Pages	2474
 XIV Support Library	 2511
 44 Number Type Support	 2513

44.1	Required Functionality of Number Types	2513
44.2	Utility Routines	2514
44.3	Built-in Number Types	2514
44.4	Number Types Provided by CGAL	2514
44.5	Number Type Provided by CORE	2515
44.6	Number Types Provided by GMP	2515
44.7	Number Types Provided by LEDA	2515
44.8	User-supplied Number Types	2516
	Reference Manual	2517
44.9	Classified Reference Pages	2517
44.10	Alphabetical List of Reference Pages	2519
45	STL Extensions for CGAL	2595
45.1	Doubly-Connected List Managing Items in Place	2595
45.2	Compact Container	2595
45.3	Multiset with Extended Functionality	2596
	Reference Manual	2597
45.4	Classified Reference Pages	2597
45.5	Alphabetical List of Reference Pages	2599
46	Handles and Circulators	2703
46.1	Handles	2703
46.2	Circulators	2703
	Reference Manual	2709
46.3	Classified Reference Pages	2709
46.4	Alphabetical List of Reference Pages	2710
47	Geometric Object Generators	2733
47.1	Example Generating Degenerate Point Sets	2734
47.2	Example Generating Grid Points	2735
47.3	Examples Generating Segments	2736

Reference Manual	2741
47.4 Classified Reference Pages	2741
47.5 Alphabetical List of Reference Pages	2742
48 Timers, Hash Map, Union-find, Modifiers	2771
48.1 Timers	2771
48.2 Memory Size	2771
48.3 Unique Hash Map	2771
48.4 Union-find	2771
48.5 Protected Access to Internal Representations	2772
Reference Manual	2773
48.6 Classified Reference Pages	2773
48.7 Alphabetical List of Reference Pages	2773
49 IO Streams	2785
49.1 Output Operator	2786
49.2 Input Operator	2786
49.3 Stream Support	2787
Reference Manual	2789
49.4 Classified Reference Pages	2789
49.5 Alphabetical List of Reference Pages	2790
50 IO Streams Colors	2807
50.1 Colors	2807
Reference Manual	2809
50.2 Colors	2809
51 Geomview	2811
51.1 Definition	2811
51.2 Implementation	2811
51.3 Example	2812

Reference Manual	2815
51.4 Alphabetical List of Reference Pages	2815
52 Qt_widget	2823
52.1 Introduction	2823
52.2 Qt_widget	2824
52.3 Layers	2829
52.4 The Standard Toolbar	2834
52.5 The Help Window	2837
52.6 Some Predefined Icons	2837
52.7 What Shall I Use?	2838
Reference Manual	2839
52.8 Classified Reference Pages	2839
52.9 Alphabetical List of Reference Pages	2839
Index	2878

Part I

General Introduction

Chapter 1

General Introduction

CGAL, the *Computational Geometry Algorithms Library*, is written in C++ and consists of several parts.

The first part is about the kernels, which consist of constant-size non-modifiable geometric primitive objects and operations on these objects. The objects are represented both as stand-alone classes that are parameterized by a representation class, which specifies the underlying number types used for calculations and as members of the kernel classes, which allows for more flexibility and adaptability of the kernel. CGAL has several kernels, for 2D and 3D, for arbitrary dimensional objects, and for 2D curved objects.

The following parts present a collection of basic geometric data structures and algorithms, which are parameterized by traits classes that define the interface between the data structure or algorithm and the primitives they use. In many cases, the kernel classes provided in CGAL can be used as traits classes for these data structures and algorithms. The collection of basic geometric algorithms and data structures currently includes polygons, half-edge data structures, polyhedral surfaces, arrangements of curves, triangulations in 2D and 3D, surface mesh generators, subdivision and parametrisation of surface meshes, Voronoi diagrams of points, disks and segments, Boolean operations on polygons and polyhedra, convex hulls, alpha shapes, optimisation algorithms, dynamic point sets for geometric queries, range and segment trees, and kinetic data structures.

The last part of the library consists of non-geometric support facilities, such as support for number types, STL extensions for CGAL, handles, circulators, protected access to internal representations, geometric object generators, timers, I/O stream operators and other stream support including PostScript, colors, windows, and visualization tools Geomview and a Qt widget for 2D CGAL objects.

Additional documents accompanying the CGAL distribution are the ‘Installation Guide’ and ‘The Use of STL and STL Extensions in CGAL’, which gives a manual style introduction to STL constructs such as iterators and containers, as well an extension, called circulator, used in many places in CGAL. We also recommend the standard text book by Austern [Aus98] for the STL and its notion of *concepts* and *models*.

Other resources for CGAL are the tutorials at <http://www.cgal.org/Tutorials/> and the user support page at www.cgal.org.

1.1 License Issues

CGAL is Open Source software, and consists of different parts covered by different licenses. In this section we explain the essence of the different licenses, as well as the rationale why we have chosen them.

The fact that CGAL is Open Source software does not mean that users are free to do whatever they want with the software. Using the software means to accept the license, which has the status of a contract between the user and the owner of the CGAL software. A more detailed description of the license terms is available in the CGAL software tarball.

1.1.1 QPL

The QPL is an Open Source license that obliges you to distribute your software based on QPLed CGAL data structures. The rationale behind this is that we can claim access to your software. The license further obliges you to put your software under an Open Source license as well. The rationale behind is that we can distribute your software, even if this is not your intention. Finally, the QPL requires that, if you modify CGAL, you distribute the modifications in the form of patches and you distribute the sources of your changes as well.

The exact license terms as well as an annotated version of the license can be found at the Trolltech web site: <http://www.trolltech.com/products/qt/licenses/licensing/qpl> and <http://www.trolltech.com/products/qt/licenses/licensing/qpl-annotated>

1.1.2 LGPL

The LGPL is an Open Source license that obliges you to distribute modifications you make on CGAL software accessible to the users. There is no obligation to make the source code of software you build on top of LGPLed CGAL data structures available.

Currently the linear kernel, the support library, the halfedge data structure, the kinetic data structures, and the mesh subdivision framework are distributed under the LGPL. The rationale behind is that we want to promote them as defacto standards.

The exact license terms can be found at the Free Software Foundation web site: <http://www.gnu.org/copyleft/lesser.html>.

1.1.3 Commercial Licenses

Users who cannot comply to the Open Source license terms can buy individual data structures under various commercial licenses from GeometryFactory: <http://www.geometryfactory.com>.

1.1.4 License Compatibility

The General Public License (GPL) has a viral effect which makes it incompatible with the QPL. For more information, please refer to the paragraph about the QPL on the licenses web page of the Free Software Foundation (FSF): http://www.fsf.org/licenses/licenses/index_html. It is therefore not possible to build a program including GPL code and some QPL parts of CGAL. In this case, if you are the copyright owner of the GPL code, you can amend the license by adding an exception allowing the use of CGAL with it (see again the FSF web page).

1.2 Third Party Software

In this section we list the software

1.2.1 Standard Template Library

CGAL heavily uses the STL, and in particular adopted many of its design ideas. The STL comes with the compiler, but it is possible to use the compiler together with an alternative STL implementation.

1.2.2 Boost

Boost is a collection of libraries. CGAL needs some of them, that is it is mandatory. If Boost is not already on your system, e.g., on Windows, you can download it from <http://www.boost.org>.

1.2.3 GMP

A library for multi precision integers and rational numbers. CGAL offers adapters for these number types. The usage of the GMP library is optional. If it is not already on your system, e.g., on Windows, you can download it from www.swox.com/gmp or from the Download section of <http://www.cgal.org>.

1.2.4 MPFR

A library for multi precision floating point numbers. The usage of the MPFR library is optional, and you must install it when you use GMP. You can download MPFR from <http://www.mpfr.org> or from the Download section of <http://www.cgal.org>.

1.2.5 Leda

A library of efficient data structures and algorithms. CGAL offers adapters to the LEDA number types. The usage is optional. It is only available commercially from <http://www.algorithmic-solutions.com>.

1.2.6 Taucs

A library of sparse linear solvers. It can be used by the *Surface_mesh_parameterization* package in order to speed up the algorithm. You can download the official release from <http://www.tau.ac.il/~stoledo/taucs/> or download a version customized by CGAL team from the Download section of <http://www.cgal.org>.

1.2.7 OpenNL

OpenNL (Open Numerical Library) is a library to easily construct and solve sparse linear systems. It is the default solver of the *Surface_mesh_parameterization* package.

The author is Bruno Lévy. OpenNL main page is <http://www.loria.fr/~levy/software/>.

CGAL includes a version of OpenNL in C++, made especially for CGAL by Bruno Lévy.

1.2.8 zlib

A data compression library. It is used in the examples of the *Surface_mesher* package. If it is not already on your system, e.g., on Windows, you can download it from <http://www.gzip.org/zlib>.

1.2.9 Qt

A GUI library. The usage of Qt is optional, but note that it is used for all 2D demos.

As Qt is the layer underneath KDE, Qt is installed on many Linux systems. Otherwise you can download it from <http://www.trolltech.com>.

1.2.10 Coin

An implementation of Open Inventor. It is used in the demo of the *Kinetic_data_structures* package. You can download it from <http://www.sim.no>.

1.3 Advanced

In this manual you will encounter sections marked as follows.

— *advanced* —

Some functionality is considered more advanced. Such functionality is described in sections such as this one that are bounded by horizontal brackets.

— *advanced* —

1.4 Namespace CGAL

All names introduced by CGAL, especially those documented in these manuals, are in a namespace called *CGAL*, which is in global scope. A user can either qualify names from CGAL by adding *CGAL::*, e.g., *CGAL::Point_2* < *CGAL::Homogeneous* < *int* > >, make a single name from CGAL visible in a scope via a *using* statement, e.g., *using CGAL::Cartesian*;, and then use this name unqualified in this scope, or even make all names from namespace *CGAL* visible in a scope with *using namespace CGAL*;. The latter, however, is likely to give raise to name conflicts and is therefore not recommended.

1.5 Inclusion Order of Header Files

Not all compilers fully support standard header names. CGAL provides workarounds for these problems in *CGAL/basic.h*. Consequently, as a golden rule, you should always include *CGAL/basic.h* first in your programs (or *CGAL/Cartesian.h*, or *CGAL/Homogeneous.h*, since they include *CGAL/basic.h* first).

1.6 Compile-time Flags to Control Inlining

Making functions inlined can, at times, improve the efficiency of your code. However this is not always the case and it can differ for a single function depending on the application in which it is used. Thus CGAL defines a set of compile-time macros that can be used to control whether certain functions are designated as inlined functions or not. The following table lists the macros and their default values, which are set in one of the CGAL include files.

macro name	default
<i>CGAL_KERNEL_INLINE</i>	inline
<i>CGAL_KERNEL_MEDIUM_INLINE</i>	
<i>CGAL_KERNEL_LARGE_INLINE</i>	
<i>CGAL_MEDIUM_INLINE</i>	inline
<i>CGAL_LARGE_INLINE</i>	
<i>CGAL_HUGE_INLINE</i>	

If you wish to change the value of one or more of these macros, you can simply give it a new value when compiling. For example, to make functions that use the macro *CGAL_KERNEL_MEDIUM_INLINE* inline functions, you should set the value of this macro to `inline` instead of the default blank.

1.7 Checks

Much of the CGAL code contains checks. For example, all checks used in the kernel code are prefixed by *CGAL_KERNEL*. Other packages have their own prefixes, as documented in the corresponding chapters. Some are there to check if the kernel behaves correctly, others are there to check if the user calls kernel routines in an acceptable manner.

There are four types of checks. The first three are errors and lead to a halt of the program if they fail. The last only leads to a warning.

Preconditions check if the caller of a routine has called it in a proper fashion. If such a check fails it is the responsibility of the caller (usually the user of the library).

Postconditions check if a routine does what it promises to do. If such a check fails it is the fault of this routine, so of the library.

Assertions are other checks that do not fit in the above two categories.

Warnings are checks for which it is not so severe if they fail.

By default, all of these checks are performed. It is however possible to turn them off through the use of compile time switches. For example, for the checks in the kernel code, these switches are the following: *CGAL_KERNEL_NO_PRECONDITIONS*, *CGAL_KERNEL_NO_POSTCONDITIONS*, *CGAL_KERNEL_*

NO_ASSERTIONS and *CGAL_KERNEL_NO_WARNINGS*. So, in order to compile the file `foo.C` with the post-condition checks off, you can do:

```
CC -DCGAL_KERNEL_NO_POSTCONDITIONS foo.C
```

This is also preferably done by modifying your makefile by adding *-DCGAL_KERNEL_NO_POSTCONDITIONS* to the *CXXFLAGS* variable.

The name *KERNEL* in the macro name can be replaced by a package specific name in order to control assertions done in a given package. This name is given in the documentation of the corresponding package, in case it exists.

Note that global macros can also be used to control the behavior over the whole CGAL library:

- *CGAL_NO_PRECONDITIONS*,
- *CGAL_NO_POSTCONDITIONS*,
- *CGAL_NO_ASSERTIONS* and
- *CGAL_NO_WARNINGS*.

Moreover, the standard macro *NDEBUG*, which controls the behavior of the *std::assert* function, also affects all checks in CGAL. This way, adding *-DNDEBUG* to your compilation flags removes absolutely all checks, including standard ones using *std::assert*. This is the default recommended setup for performing timing benchmarks for example.

Not all checks are on by default. All four types of checks can be marked as expensive or exactness checks (or both). These checks need to be turned on explicitly by supplying one or both of the compile time switches *CGAL_KERNEL_CHECK_EXPENSIVE* and *CGAL_KERNEL_CHECK_EXACTNESS*.

Expensive checks are, as the word says, checks that take a considerable time to compute. Considerable is an imprecise phrase. Checks that add less than 10 percent to the execution time of the routine they are in are not expensive. Checks that can double the execution time are. Somewhere in between lies the border line. Checks that increase the asymptotic running time of an algorithm are always considered expensive. Exactness checks are checks that rely on exact arithmetic. For example, if the intersection of two lines is computed, the postcondition of this routine may state that the intersection point lies on both lines. However, if the computation is done with doubles as number type, this may not be the case, due to round off errors. So, exactness checks should only be turned on if the computation is done with some exact number type.

1.7.1 Altering the Failure Behaviour

As stated above, if a postcondition, precondition or assertion is violated, the program will abort (stop and produce a core dump). This behaviour can be changed by means of the following function.

```
#include <CGAL/assertions.h>
```

```
Failure_behaviour set_error_behaviour( Failure_behaviour eb)
```

The parameter should have one of the following values.

```
enum Failure_behaviour { ABORT, EXIT, EXIT_WITH_SUCCESS, CONTINUE};
```

The first value is the default. If the *EXIT* value is set, the program will stop and return a value indicating failure, but not dump the core. The last value tells the checks to go on after diagnosing the error.

————— *advanced* —————

If the *EXIT_WITH_SUCCESS* value is set, the program will stop and return a value corresponding to successful execution and not dump the core.

————— *advanced* —————

The value that is returned by *set_error_behaviour* is the value that was in use before.

For warnings there is a separate routine, which works in the same way. The only difference is that for warnings the default value is *CONTINUE*.

Failure_behaviour set_warning_behaviour(Failure_behaviour eb)

1.7.2 Control at a Finer Granularity

The compile time flags as described up to now all operate on the whole library. Sometimes you may want to have a finer control. CGAL offers the possibility to turn checks on and off with a bit finer granularity, namely the module in which the routines are defined. The name of the module is to be appended directly after the CGAL prefix. So, the flag *CGAL_KERNEL_NO_ASSERTIONS* switches off assertions in the kernel only, the flag *CGAL_CH_CHECK_EXPENSIVE* turns on expensive checks in the convex hull module. The name of a particular module is documented with that module.

————— *advanced* —————

1.7.3 Customising how Errors are Reported

Normally, error messages are written to the standard error output. It is possible to do something different with them. To that end you can register your own handler. This function should be declared as follows.

```
void          my_failure_function(
                const char *type,
                const char *expression,
                const char *file,
                int line,
                const char *explanation)
```

Your failure function will be called with the following parameters. *type* is a string that contains one of the words precondition, postcondition, assertion or warning. The parameter *expression* contains the expression that was violated. *file* and *line* contain the place where the check was made. The *explanation* parameter contains an explanation of what was checked. It can be *NULL*, in which case the *expression* is thought to be descriptive enough.

There are several things that you can do with your own handler. You can display a diagnostic message in a different way, for instance in a pop up window or to a log file (or a combination). You can also implement a different policy on what to do after an error. For instance, you can throw an exception or ask the user in a dialogue whether to abort or to continue. If you do this, it is best to set the error behaviour to *CONTINUE*, so that it does not interfere with your policy.

You can register two handlers, one for warnings and one for errors. Of course, you can use the same function for both if you want. When you set a handler, the previous handler is returned, so you can restore it if you want.

```

#include <CGAL/assertions.h>
Failure_function      set_error_handler( Failure_function handler)
Failure_function      set_warning_handler( Failure_function handler)

```

Example

```

#include <CGAL/assertions.h>

void my_failure_handler(
    const char *type,
    const char *expr,
    const char* file,
    int line,
    const char* msg)
{
    /* report the error in some way. */
}

void foo()
{
    CGAL::Failure_function prev;
    prev = CGAL::set_error_handler(my_failure_handler);
    /* call some routines. */
    CGAL::set_error_handler(prev);
}

```

└────────── *advanced* ─────────┘

Part II

Kernels

Chapter 2

2D and 3D Kernel

Hervé Brönnimann, Andreas Fabri, Geert-Jan Giezeman, Susan Hert, Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Stefan Schirra

2.1 Introduction

CGAL, the *Computational Geometry Algorithms Library*, is written in C++ and consists of three major parts. The first part is the kernel, which consists of constant-size non-modifiable geometric primitive objects and operations on these objects. The objects are represented both as stand-alone classes that are parameterized by a representation class, which specifies the underlying number types used for calculations and as members of the kernel classes, which allows for more flexibility and adaptability of the kernel. The second part is a collection of basic geometric data structures and algorithms, which are parameterized by traits classes that define the interface between the data structure or algorithm and the primitives they use. In many cases, the kernel classes provided in CGAL can be used as traits classes for these data structures and algorithms. The third part of the library consists of non-geometric support facilities, such as circulators, random sources, I/O support for debugging and for interfacing CGAL to various visualization tools.

This part of the reference manual covers the kernel. The kernel contains objects of constant size, such as point, vector, direction, line, ray, segment, triangle, iso-oriented rectangle and tetrahedron. With each type comes a set of functions which can be applied to an object of this type. You will typically find access functions (e.g. to the coordinates of a point), tests of the position of a point relative to the object, a function returning the bounding box, the length, or the area of an object, and so on. The CGAL kernel further contains basic operations such as affine transformations, detection and computation of intersections, and distance computations.

2.1.1 Robustness

The correctness proof of nearly all geometric algorithms presented in theory papers assumes exact computation with real numbers. This leads to a fundamental problem with the implementation of geometric algorithms. Naively, often the exact real arithmetic is replaced by inexact floating-point arithmetic in the implementation. This often leads to acceptable results for many input data. However, even for the implementation of the simplest geometric algorithms this simplification occasionally does not work. Rounding errors introduced by an inaccurate arithmetic may lead to inconsistent decisions, causing unexpected failures for some correct input data. There are many approaches to this problem, one of them is to compute exactly (compute so accurate that all decisions made by the algorithm are exact) which is possible in many cases but more expensive than standard floating-point arithmetic. C. M. Hoffmann [[Hof89a](#), [Hof89b](#)] illustrates some of the problems arising in the implementation of geometric algorithms and discusses some approaches to solve them. A more recent overview

is given in [Sch00]. The exact computation paradigm is discussed by Yap and Dubé [YD95] and Yap [Yap97].

In CGAL you can choose the underlying number types and arithmetic. You can use different types of arithmetic simultaneously and the choice can be easily changed, e.g. for testing. So you can choose between implementations with fast but occasionally inexact arithmetic and implementations guaranteeing exact computation and exact results. Of course you have to pay for the exactness in terms of execution time and storage space. See the section on number types in the Support Library for more details on number types and their capabilities and performance.

2.2 Kernel Representations

Our object of study is the d -dimensional affine Euclidean space. Here we are mainly concerned with cases $d = 2$ and $d = 3$. Objects in that space are sets of points. A common way to represent the points is the use of Cartesian coordinates, which assumes a reference frame (an origin and d orthogonal axes). In that framework, a point is represented by a d -tuple $(c_0, c_1, \dots, c_{d-1})$, and so are vectors in the underlying linear space. Each point is represented uniquely by such Cartesian coordinates. Another way to represent points is by homogeneous coordinates. In that framework, a point is represented by a $(d + 1)$ -tuple (h_0, h_1, \dots, h_d) . Via the formulae $c_i = h_i/h_d$, the corresponding point with Cartesian coordinates $(c_0, c_1, \dots, c_{d-1})$ can be computed. Note that homogeneous coordinates are not unique. For $\lambda \neq 0$, the tuples (h_0, h_1, \dots, h_d) and $(\lambda \cdot h_0, \lambda \cdot h_1, \dots, \lambda \cdot h_d)$ represent the same point. For a point with Cartesian coordinates $(c_0, c_1, \dots, c_{d-1})$ a possible homogeneous representation is $(c_0, c_1, \dots, c_{d-1}, 1)$. Homogeneous coordinates in fact allow to represent objects in a more general space, the projective space \mathbb{P}_d . In CGAL, we do not compute in projective geometry. Rather, we use homogeneous coordinates to avoid division operations, since the additional coordinate can serve as a common denominator.

2.2.1 Genericity through Parameterization

Almost all the kernel objects (and the corresponding functions) are templates with a parameter that allows the user to choose the representation of the kernel objects. A type that is used as an argument for this parameter must fulfill certain requirements on syntax and semantics. The list of requirements defines an abstract kernel concept. For all kernel objects types, the types `CGAL::Type<Kernel>` and `Kernel::Type` are identical.

CGAL offers four families of concrete models for the concept Kernel, two based on the Cartesian representation of points and two based on the homogeneous representation of points. The interface of the kernel objects is designed such that it works well with both Cartesian and homogeneous representation. For example, points in 2D have a constructor with three arguments as well (the three homogeneous coordinates of the point). The common interfaces parameterized with a kernel class allow one to develop code independent of the chosen representation. We said “families” of models, because both families are parameterized too. A user can choose the number type used to represent the coordinates.

For reasons that will become evident later, a kernel class provides two typenames for number types, namely `Kernel::FT` and `Kernel::RT`.¹ The type `Kernel::FT` must fulfill the requirements on what is called a *FieldNumberType* in CGAL. This roughly means that `Kernel::FT` is a type for which operations $+$, $-$, $*$ and $/$ are defined with semantics (approximately) corresponding to those of a field in a mathematical sense. Note that, strictly speaking, the built-in type `int` does not fulfill the requirements on a field type, since *ints* correspond to elements of a ring rather than a field, especially operation $/$ is not the inverse of $*$. The requirements on the type `Kernel::RT` are weaker. This type must fulfill the requirements on what is called a *RingNumberType* in CGAL. This roughly means that `Kernel::RT` is a type for which operations $+$, $-$, $*$ are defined with semantics (approximately) corresponding to those of a ring in a mathematical sense.

¹The double colon `::` is the C++ scope operator.

2.2.2 Cartesian Kernels

With *Cartesian*<*FieldNumberType*> you can choose a Cartesian representation of coordinates. When you choose Cartesian representation you have to declare at the same time the type of the coordinates. A number type used with the *Cartesian* representation class should be a *FieldNumberType* as described above. As mentioned above, the built-in type *int* is not a *FieldNumberType*. However, for some computations with Cartesian representation, no division operation is needed, i.e., a *RingNumberType* is sufficient in this case. With *Cartesian*<*FieldNumberType*>, both *Cartesian*<*FieldNumberType*>::*FT* and *Cartesian*<*FieldNumberType*>::*RT* are mapped to *FieldNumberType*.

Cartesian<*FieldNumberType*> uses reference counting internally to save copying costs. CGAL also provides *Simple_cartesian*<*FieldNumberType*>, a kernel that uses Cartesian representation but no reference counting. Debugging is easier with *Simple_cartesian*<*FieldNumberType*>, since the coordinates are stored within the class and hence direct access to the coordinates is possible. Depending on the algorithm, it can also be slightly more or less efficient than *Cartesian*<*FieldNumberType*>. Again, in *Simple_cartesian*<*FieldNumberType*> both *Simple_cartesian*<*FieldNumberType*>::*FT* and *Simple_cartesian*<*FieldNumberType*>::*RT* are mapped to *FieldNumberType*.

2.2.3 Homogeneous Kernels

Homogeneous coordinates permit to avoid division operations in numerical computations, since the additional coordinate can serve as a common denominator. Avoiding divisions can be useful for exact geometric computation. With *Homogeneous*<*RingNumberType*> you can choose a homogeneous representation for the coordinates of the kernel objects. As for the Cartesian representation, one has to declare the type used to store the coordinates. Since the homogeneous representation does not use divisions, the number type associated with a homogeneous representation class must be a model for the weaker concept *RingNumberType* only. However, some operations provided by this kernel involve divisions, for example computing squared distances or Cartesian coordinates. To keep the requirements on the number type parameter of *Homogeneous* low, the number type *Quotient*<*RingNumberType*> is used for operations that require divisions. This number type can be viewed as an adaptor which turns a *RingNumberType* into a *FieldNumberType*. It maintains numbers as quotients, i.e., a numerator and a denominator. With *Homogeneous*<*RingNumberType*>, *Homogeneous*<*RingNumberType*>::*FT* is equal to *Quotient*<*RingNumberType*>, while *Homogeneous*<*RingNumberType*>::*RT* is equal to *RingNumberType*.

Homogeneous<*RingNumberType*> uses reference counting internally to save copying costs. CGAL also provides *Simple_homogeneous*<*RingNumberType*>, a kernel that uses homogeneous representation but no reference counting. Debugging is easier with *Simple_homogeneous*<*RingNumberType*>, since the coordinates are stored within the class and hence direct access to the coordinates is possible. Depending on the algorithm, it can also be slightly more or less efficient than *Homogeneous*<*RingNumberType*>. Again, in *Simple_homogeneous*<*RingNumberType*> the type *Simple_homogeneous*<*RingNumberType*>::*FT* is equal to *Quotient*<*RingNumberType*> while *Simple_homogeneous*<*RingNumberType*>::*RT* is equal to *RingNumberType*.

2.2.4 Naming conventions

The use of kernel classes not only avoids problems, it also makes all CGAL classes very uniform. They **always** consist of:

1. The *capitalized base name* of the geometric object, such as *Point*, *Segment*, or *Triangle*.
2. An *underscore* followed by the *dimension* of the object, for example *_2*, *_3*, or *_d*.

3. A *kernel class* as parameter, which itself is parameterized with a number type, such as *Cartesian<double>* or *Homogeneous<leda_integer>*.

2.2.5 Kernel as a Traits Class

Algorithms and data structures in the basic library of CGAL are parameterized by a traits class that subsumes the objects on which the algorithm or data structure operates as well as the operations to do so. For most of the algorithms and data structures in the basic library you can use a kernel as a traits class. For some algorithms you even do not have to specify the kernel; it is detected automatically using the types of the geometric objects passed to the algorithm. In some other cases, the algorithms or data structures needs more than is provided by the kernel concept. In these cases, a kernel can not be used as a traits class.

2.2.6 Choosing a Kernel and Predefined Kernels

If you start with integral Cartesian coordinates, many geometric computations will involve integral numerical values only. Especially, this is true for geometric computations that evaluate only predicates, which are tantamount to determinant computations. Examples are triangulation of point sets and convex hull computation. In this case, the Cartesian representation is probably the first choice, even with a ring type. You might use limited precision integer types like *int* or *long*, use *double* to present your integers (they have more bits in their mantissa than an *int* and overflow nicely), or an arbitrary precision integer type like the wrapper *Gmpz* for the GMP integers, *leda_integer*, or *MP_Float*. Note, that unless you use an arbitrary precision ring type, incorrect results might arise due to overflow.

If new points are to be constructed, for example the intersection point of two lines, computation of Cartesian coordinates usually involves divisions. Hence, one needs to use a *FieldNumberType* with Cartesian representation, or alternatively, switch to homogeneous representation. The type *double* is a – though imprecise – model for *FieldNumberType*. You can also put any *RingNumberType* into the *Quotient* adaptor to get a field type which then can be put into *Cartesian*. But using homogeneous representation on the *RingNumberType* is usually the better option. Other valid *FieldNumberTypes* are *leda_rational* and *leda_real*.

If it is crucial for you that the computation is reliable, the right choice is probably a number type that guarantees exact computation. The *Filtered_kernel* provides a way to apply filtering techniques [BBP01] to achieve a kernel with exact and efficient predicates. Still other people will prefer the built-in type *double*, because they need speed and can live with approximate results, or even algorithms that, from time to time, crash or compute incorrect results due to accumulated rounding errors.

Predefined kernels. For the user's convenience, CGAL provides 3 typedefs to generally useful kernels.

- They are all Cartesian kernels.
- They all support constructions of points from *double* Cartesian coordinates.
- All these 3 kernels provide exact geometric predicates.
- They handle geometric constructions differently:
 - *Exact_predicates_exact_constructions_kernel*: provides exact geometric constructions, in addition to exact geometric predicates.

- *Exact_predicates_exact_constructions_kernel_with_sqrt*: same as *Exact_predicates_exact_constructions_kernel*, but the number type it provides (*Exact_predicates_exact_constructions_kernel_with_sqrt::FT*) supports the square root operation exactly².
- *Exact_predicates_inexact_constructions_kernel*: provides exact geometric predicates, but geometric constructions may be inexact due to roundoff errors. It is however enough for most CGAL algorithms, and faster than both *Exact_predicates_exact_constructions_kernel* and *Exact_predicates_exact_constructions_kernel_with_sqrt*.

2.3 Kernel Geometry

2.3.1 Points and Vectors

In CGAL, we strictly distinguish between points, vectors and directions. A *point* is a point in the Euclidean space \mathbb{E}^d , a *vector* is the difference of two points p_2, p_1 and denotes the direction and the distance from p_1 to p_2 in the vector space \mathbb{R}^d , and a *direction* is a vector where we forget about its length. They are different mathematical concepts. For example, they behave different under affine transformations and an addition of two points is meaningless in affine geometry. By putting them in different classes we not only get cleaner code, but also type checking by the compiler which avoids ambiguous expressions. Hence, it pays twice to make this distinction.

CGAL defines a symbolic constant *ORIGIN* of type *Origin* which denotes the point at the origin. This constant is used in the conversion between points and vectors. Subtracting it from a point p results in the locus vector of p .

```
Point_2< Cartesian<double> > p(1.0, 1.0), q;
Vector_2< Cartesian<double> > v;
v = p - ORIGIN;
q = ORIGIN + v;
assert( p == q );
```

In order to obtain the point corresponding to a vector v you simply have to add v to *ORIGIN*. If you want to determine the point q in the middle between two points p_1 and p_2 , you can write³

```
q = p_1 + (p_2 - p_1) / 2.0;
```

Note that these constructions do not involve any performance overhead for the conversion with the currently available representation classes.

2.3.2 Kernel Objects

Besides points (*Point_2<Kernel>*, *Point_3<Kernel>*, *Point_d<Kernel>*), vectors (*Vector_2<Kernel>*, *Vector_3<Kernel>*), and directions (*Direction_2<Kernel>*, *Direction_3<Kernel>*), CGAL provides lines, rays, segments, planes, triangles, tetrahedra, iso-rectangles, iso-cuboids, circles and spheres.

Lines (*Line_2<Kernel>*, *Line_3<Kernel>*) in CGAL are oriented. In two-dimensional space, they induce a partition of the plane into a positive side and a negative side. Any two points on a line induce an orientation of this

²Currently it requires having either LEDA or CORE installed

³you might call *midpoint(p_1,p_2)* instead

line. A ray (*Ray_2<Kernel>*, *Ray_3<Kernel>*) is semi-infinite interval on a line, and this line is oriented from the finite endpoint of this interval towards any other point in this interval. A segment (*Segment_2<Kernel>*, *Segment_3<Kernel>*) is a bounded interval on a directed line, and the endpoints are ordered so that they induce the same direction as that of the line.

Planes are affine subspaces of dimension two in \mathbb{E}^3 , passing through three points, or a point and a line, ray, or segment. CGAL provides a correspondence between any plane in the ambient space \mathbb{E}^3 and the embedding of \mathbb{E}^2 in that space. Just like lines, planes are oriented and partition space into a positive side and a negative side. In CGAL, there are no special classes for halfspaces. Halfspaces in 2D and 3D are supposed to be represented by oriented lines and planes, respectively.

Concerning polygons and polyhedra, the kernel provides triangles, iso-oriented rectangles, iso-oriented cuboids and tetrahedra. More complex polygons⁴ and polyhedra or polyhedral surfaces can be obtained from the basic library (*Polygon_2*, *Polyhedron_3*), so they are not part of the kernel. As with any Jordan curves, triangles, iso-oriented rectangles and circles separate the plane into two regions, one bounded and one unbounded.

2.3.3 Orientation and Relative Position

Geometric objects in CGAL have member functions that test the position of a point relative to the object. Full dimensional objects and their boundaries are represented by the same type, e.g. halfspaces and hyperplanes are not distinguished, neither are balls and spheres and discs and circles. Such objects split the ambient space into two full-dimensional parts, a bounded part and an unbounded part (e.g. circles), or two unbounded parts (e.g. hyperplanes). By default these objects are oriented, i.e., one of the resulting parts is called the positive side, the other one is called the negative side. Both of these may be unbounded.

For these objects there is a function *oriented_side()* that determines whether a test point is on the positive side, the negative side, or on the oriented boundary. These function returns a value of type *Oriented_side*.

Those objects that split the space in a bounded and an unbounded part, have a member function *bounded_side()* with return type *Bounded_side*.

If an object is lower dimensional, e.g. a triangle in three-dimensional space or a segment in two-dimensional space, there is only a test whether a point belongs to the object or not. This member function, which takes a point as an argument and returns a boolean value, is called *has_on()*

2.4 Predicates and Constructions

2.4.1 Predicates

Predicates are at the heart of a geometry kernel. They are basic units for the composition of geometric algorithms and encapsulate decisions. Hence their correctness is crucial for the control flow and hence for the correctness of an implementation of a geometric algorithm. CGAL uses the term predicate in a generalized sense. Not only components returning a Boolean value are called predicates but also components returning an enumeration type like a *Comparison_result* or an *Orientation*. We say components, because predicates are implemented both as functions and function objects (provided by a kernel class).

CGAL provides predicates for the orientation of point sets (*orientation*, *leftturn*, *rightturn*, *collinear*, *coplanar*), for comparing points according to some given order, especially for comparing Cartesian coordinates

⁴Any sequence of points can be seen as a (not necessary simple) polygon or polyline. This view is used frequently in the basic library as well.

(e.g. *lexicographically_xy_smaller*), in-circle and in-sphere tests, and predicates to compare distances.

2.4.2 Constructions

Functions and function objects that generate objects that are neither of type *bool* nor enum types are called constructions. Constructions involve computation of new numerical values and may be imprecise due to rounding errors unless a kernel with an exact number type is used.

Affine transformations (*Aff_transformation_2<Kernel>*, *Aff_transformation_3<Kernel>*) allow to generate new object instances under arbitrary affine transformations. These transformations include translations, rotations (in 2D only) and scaling. Most of the geometric objects in a kernel have a member function *transform(Aff_transformation t)* which applies the transformation to the object instance.

CGAL also provides a set of functions that detect or compute the intersection between objects of the 2D kernel, and many objects in the 3D kernel, and functions to calculate their squared distance. Moreover, some member functions of kernel objects are constructions.

So there are routines that compute the square of the Euclidean distance, but no routines that compute the distance itself. Why? First of all, the two values can be derived from each other quite easily (by taking the square root or taking the square). So, supplying only the one and not the other is only a minor inconvenience for the user. Second, often either value can be used. This is for example the case when (squared) distances are compared. Third, the library wants to stimulate the use of the squared distance instead of the distance. The squared distance can be computed in more cases and the computation is cheaper. We do this by not providing the perhaps more natural routine. The problem of a distance routine is that it needs the *sqrt* operation. This has two drawbacks:

- The *sqrt* operation can be costly. Even if it is not very costly for a specific number type and platform, avoiding it is always cheaper.
- There are number types on which no *sqrt* operation is defined, especially integer types and rationals.

2.4.3 Polymorphic Return Values

Some functions can return different types of objects. A typical C++ solution to this problem is to derive all possible return types from a common base class, to return a pointer to this class and to perform a dynamic cast on this pointer. The class *Object* provides an abstraction. An object *obj* of the class *Object* can represent an arbitrary class. The only operations it provides is to make copies and assignments, so that you can put them in lists or arrays. Note that *Object* is NOT a common base class for the elementary classes. Therefore, there is no automatic conversion from these classes to *Object*. Rather this is done with the global function *make_object()*. This encapsulation mechanism requires the use of *assign* or *object_cast* to access the encapsulated class.

Example

In the following example, the object class is used as return value for the intersection computation, as there are possibly different return values.

```
{
    typedef Cartesian<double> K;
    typedef K::Point_2      Point_2;
    typedef K::Segment_2    Segment_2;
```

```

Point_2 point;
Segment_2 segment, segment_1, segment_2;

std::cin >> segment_1 >> segment_2;

Object obj = intersection(segment_1, segment_2);

if (assign(point, obj)) {
    /* do something with point */
} else if ((assign(segment, obj)) {
    /* do something with segment*/
}

/* there was no intersection */
}

```

A more efficient way is to use *object_cast* :

```

{
    typedef Cartesian<double> K;
    typedef K::Point_2      Point_2;
    typedef K::Segment_2     Segment_2;

    Segment_2 segment_1, segment_2;

    std::cin >> segment_1 >> segment_2;

    Object obj = intersection(segment_1, segment_2);

    if (const Point_2 *point = object_cast<Point_2>(&obj)) {
        /* do something with *point */
    } else if (const Segment_2 *segment = object_cast<Segment_2>(&obj)) {
        /* do something with *segment*/
    }

    /* there was no intersection */
}

```

The intersection routine itself looks roughly as follows:

```

template < class Kernel >
Object intersection(Segment_2<Kernel> s1, Segment_2<Kernel> s2)
{

    if (/* intersection in a point */ ) {

        Point_2<Kernel> p = ... ;
        return make_object(p);
    }
}

```

```

    } else if (/* intersection in a segment */ ) {

        Segment_2<Kernel> s = ... ;
        return make_object(s);
    }
    return Object();
}

```

2.4.4 Constructive Predicates

For testing where a point p lies with respect to a plane defined by three points q , r and s , one may be tempted to construct the plane $\text{Plane_3}<\text{Kernel}>(q,r,s)$ and use the method $\text{oriented_side}(p)$. This may pay off if many tests with respect to the plane are made. Nevertheless, unless the number type is exact, the constructed plane is only approximated, and round-off errors may lead $\text{oriented_side}(p)$ to return an orientation which is different from the orientation of p , q , r , and s .

In CGAL, we provide predicates in which such geometric decisions are made directly with a reference to the input points p , q , r , s , without an intermediary object like a plane. For the above test, the recommended way to get the result is to use $\text{orientation}(p,q,r,s)$. For exact number types, the situation is different. If several tests are to be made with the same plane, it pays off to construct the plane and to use $\text{oriented_side}(p)$.

2.5 Extensible Kernel

This manual section describe how users can plug user defined geometric classes in existing CGAL kernels. This is best illustrated by an example.

2.5.1 Introduction

CGAL defines the concept of a geometry kernel. Such a kernel provides types, construction objects and generalized predicates. Most implementations of Computational Geometry algorithms and data structures in the basic library of CGAL were done in a way that classes or functions can be parametrized with a geometric traits class.

In most cases this geometric traits class must be a model of the CGAL geometry kernel concept (but there are some exceptions).

2.5.2 An Extensive Example

Assume you have the following point class, where the coordinates are stored in an array of *doubles*, where we have another data member *color*, which shows up in the constructor.

```

class MyPointC2 {

private:
    double vec[2];
    int col;

```

```

public:

    MyPointC2()
        : col(0)
    {
        *vec = 0;
        *(vec+1) = 0;
    }

    MyPointC2(const double x, const double y, int c)
        : col(c)
    {
        *vec = x;
        *(vec+1) = y;
    }

    const double& x() const { return *vec; }

    const double& y() const { return *(vec+1); }

    double & x() { return *vec; }

    double& y() { return *(vec+1); }

    int color() const { return col; }

    int& color() { return col; }

    bool operator==(const MyPointC2 &p) const
    {
        return ( *vec == *(p.vec) )  && ( *(vec+1) == *(p.vec + 1) && ( col == p.col) );
    }

    bool operator!=(const MyPointC2 &p) const
    {
        return !(*this == p);
    }

};

```

As said earlier the class is pretty minimalistic, for example it has no *bbox()* method. One might assume that a basic library algorithm which computes a bounding box (e.g. to compute the bounding box of a polygon), will not compile. Luckily it will, because it does not use of member functions of geometric objects, but it makes use of the functor *Kernel::Construct_bbox_2*.

To make the right thing happen with *MyPointC2* we have to provide the following functor.

```

template <class ConstructBbox_2>
class MyConstruct_bbox_2 : public ConstructBbox_2 {
public:

```

```

CGAL::Bbox_2 operator()(const typename MyPointC2& p) const {
    return CGAL::Bbox_2(p.x(), p.y(), p.x(), p.y());
}
};

```

Things are similar for random access to the Cartesian coordinates of a point. As the coordinates are stored in an array of *doubles* we can use *double** as random access iterator.

```

class MyConstruct_coord_iterator {
public:
    const double* operator()(const MyPointC2& p)
    {
        return &p.x();
    }

    const double* operator()(const MyPointC2& p, int)
    {
        const double* pyptr = &p.y();
        pyptr++;
        return pyptr;
    }
};

```

The last functor we have to provide is the one which constructs points. That is you are not forced to add the constructor with the *Origin* as parameter to your class, nor the constructor with homogeneous coordinates. The functor is a kind of glue layer between the CGAL algorithms and your class.

```

template <typename K>
class MyConstruct_point_2
{
    typedef typename K::RT      RT;
    typedef typename K::Point_2 Point_2;
public:
    typedef Point_2      result_type;
    typedef CGAL::Arity_tag< 1 >  Arity;

    Point_2
    operator()(CGAL::Origin o) const
    { return Point_2(0,0, 0); }

    Point_2
    operator()(const RT& x, const RT& y) const
    { return Point_2(x, y, 0); }

    // We need this one, as such a functor is in the Filtered_kernel
    Point_2
    operator()(const RT& x, const RT& y, const RT& w) const
    {
        if(w != 1){
            return Point_2(x/w, y/w, 0);
        } else {

```

```

return Point_2(x,y, 0);
    }
}
};

```

Now we are ready to put the puzzle together. We won't explain it in detail, but you see that there are *typedefs* to the new point class and the functors. All the other types are inherited.

```

#ifndef MYKERNEL_H
#define MYKERNEL_H

#include <CGAL/Cartesian.h>
#include "MyPointC2.h"
#include "MySegmentC2.h"

// K_ is the new kernel, and K_Base is the old kernel
template < typename K_, typename K_Base >
class MyCartesian_base
: public K_Base::template Base<K_>::Type
{
    typedef typename K_Base::template Base<K_>::Type    OldK;
public:
    typedef K_                                            Kernel;
    typedef MyPointC2                                    Point_2;
    typedef MySegmentC2<Kernel>                          Segment_2;
    typedef MyConstruct_point_2<Kernel, OldK>             Construct_point_2;
    typedef const double*                                Cartesian_const_iterator_2;
    typedef MyConstruct_coord_iterator                   Construct_cartesian_const_iterator_2;
    typedef MyConstruct_bbox_2<typename OldK::Construct_bbox_2> Construct_bbox_2;

    Construct_point_2
    construct_point_2_object() const
    { return Construct_point_2(); }

    template < typename Kernel2 >
    struct Base { typedef MyCartesian_base<Kernel2, K_Base>  Type; };
};

template < typename FT_ >
struct MyKernel
: public CGAL::Type_equality_wrapper<
    MyCartesian_base<MyKernel<FT_>, CGAL::Cartesian<FT_> >,
    MyKernel<FT_> >
{};

#endif // MYKERNEL_H

```

Finally, we give an example how this new kernel can be used. Predicates and constructions work with the new point, they can be used to construct segments and triangles with, and data structures from the Basic Library, as the Delaunay triangulation work with them.

The kernel itself can be made robust by plugging it in the *Filtered_kernel*.

```
// file: examples/Kernel_23/MyKernel.C

#include <CGAL/basic.h>
#include <CGAL/Filtered_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/squared_distance_2.h>
#include <cassert>
#include "MyKernel.h"

typedef MyKernel<double> MK;
typedef CGAL::Filtered_kernel_adaptor<MK> K;
typedef CGAL::Delaunay_triangulation_2<K> Delaunay_triangulation_2;

typedef K::Point_2 Point;
typedef K::Segment_2 Segment;
typedef K::Ray_2 Ray;
typedef K::Line_2 Line;
typedef K::Triangle_2 Triangle;
typedef K::Iso_rectangle_2 Iso_rectangle;

const int RED= 1;
const int BLACK=2;

int main()
{
    Point a(0,0), b(1,0, BLACK), c(1,1), d(0,1);
    a.color()=RED;
    b.color()=BLACK;
    d.color()=RED;

    Delaunay_triangulation_2 dt;
    dt.insert(a);

    K::Orientation_2 orientation;
    orientation(a,b,c);

    Point p(1,2), q;
    p.color() = RED;
    q.color() = BLACK;
    std::cout << p << std::endl;

    K::Compute_squared_distance_2 squared_distance;

    std::cout << "squared_distance(a, b) == "
                << squared_distance(a, b) << std::endl;

    Segment s1(p,q), s2(a, c);

    K::Construct_midpoint_2 construct_midpoint_2;
```

```

Point mp = construct_midpoint_2(p,q);

assert(s1.source().color() == RED);

K::Intersect_2 intersection;

CGAL::Object o = intersection(s1, s2);

K::Construct_cartesian_const_iterator_2 construct_it;
K::Cartesian_const_iterator_2 cit = construct_it(a);
assert(*cit == a.x());

cit = construct_it(a,0);

cit--;
assert(*cit == a.y());

Line l1(a,b), l2(p, q);

intersection(l1, l2);

intersection(s1, l1);

Ray r1(d,b), r2(d,c);
intersection(r1, r2);

intersection(r1, l1);

squared_distance(r1, r2);
squared_distance(r1, l2);
squared_distance(r1, s2);

Triangle t1(a,b,c), t2(a,c,d);
intersection(t1, t2);
intersection(t1, l1);

intersection(t1, s1);

intersection(t1, r1);

Iso_rectangle i1(a,c), i2(d,p);
intersection(i1, i2);
intersection(i1, s1);

intersection(i1, r1);
intersection(i1, l1);

t1.orientation();

std::cout << s1.source() << std::endl;

std::cout << t1.bbox() << std::endl;

```

```

std::cout << "done" << std::endl;
return 0;
}

```

2.5.3 Limitations

The point class must have member functions $x()$ and $y()$ (and $z()$ for the 3d point). We will probably introduce function objects that take care of coordinate access.

As we enforce type equality between *MyKernel::Point_2* and *Point_2<MyKernel>*, the constructor with the color as third argument is not available.

2.6 Kernel Related Tools

2.6.1 Introduction

The following manual sections describe various tools that might be useful for various kinds of users of the CGAL kernel. The kernel concept archetype describes a minimal model for the CGAL kernel that can be used for testing CGAL kernel compatibility of geometrical algorithm implementations. It can be useful for all people developing CGAL-style code that uses the CGAL kernel.

2.6.2 Kernel Concept Archetype

Introduction

CGAL defines the concept of a geometry kernel. Such a kernel provides types, construction objects and generalized predicates. Most implementations of CG algorithms and data structures in the basic library of CGAL were done in a way that classes or functions can be parametrized with a geometric traits class.

In most cases this geometric traits class must be a model of the CGAL geometry kernel concept (but there are some exceptions).

The CGAL distribution comes with a number of models (or geometry kernels), for instance the Cartesian kernel (*CGAL::Cartesian*) or the homogeneous kernel (*CGAL::Homogeneous*), that can be used with the packages of the basic library.

But does it mean that packages of the basic library are fully compatible with the CGAL kernel concept if they can be used with these CGAL kernel models? Not necessarily, because such a package might also use member functions or global functions/operators, that are implemented for CGAL kernel types but not for other classes or kernels.

That's why it is important to verify whether the documented requirements of a package are really covered by the implementation. Manual verification is error prone, so there should be something better available in a generic library for this application.

That's why the CGAL kernel concept archetype *CGAL::Kernel_archetype* was developed. It provides all functionality required by the CGAL kernel concept, but nothing more, so it can be seen as a minimal implementation

of a model for the CGAL kernel concept. It can be used for testing successful compilation of packages of the basic library with a minimal model. Deprecated kernel functionality is not supported. All geometrical types (like the 2d/3d point or segment types) of *CGAL::Kernel_archetype* have copy constructors, default constructors and an assignment operator, and nothing else. Comparison operators are by default not supported, but can be switched on by the flag *CGAL_CONCEPT_ARCHETYPE_ALLOW_COMPARISONS*.

The geometrical types of the concept archetype encapsulate no data members, so runtime checks with the archetype are not very useful (*CGAL::Kernel_archetype* is only meant for compilation checks with a minimal model in the testsuites of CGAL packages).

The header file for the concept archetype is *CGAL/Kernel_archetype.h*.

The package supports the two- and three-dimensional part of the CGAL kernel concept. The d-dimensional part is not supported.

Restricting the Interface

Normally packages of the Basic Library or Extension packages use only a small subset of the functionality offered by models of the CGAL kernel concept. In these cases testing with a model that offers only this (used and) documented subset makes sense. *CGAL::Kernel_archetype* normally offers the full functionality (all types, functors and constructions of a CGAL kernel model), but it is possible to restrict the interface.

If you want to do this, you have to define the macro *CGAL_CA_LIMITED_INTERFACE* (before the inclusion of *CGAL/Kernel_archetype.h*) for switching on the interface limitation. Now you have to tell the kernel archetype what types have to be provided by it. For every type you have to define a macro. The name of the macro is *CGAL_CA_NAME_OF_KERNEL_TYPE*, where *NAME_OF_KERNEL_TYPE* is the name of the kernel type (written in capitals) that has to be provided by the kernel archetype for a specific package. Lets have a look at a small example. The kernel archetype has to provide in some test suite a limited interface. The interface has to offer type definitions for *Point_3* and *Plane_3* and the 3d orientation functor type definition *Orientation_3*:

```
// limit interface of the Kernel_archetype
#define CGAL_CA_LIMITED_INTERFACE
#define CGAL_CA_POINT_3
#define CGAL_CA_PLANE_3
#define CGAL_CA_ORIENTATION_3

#include <CGAL/Kernel_archetype.h>
```

Now other kernel functionality is removed from the interface of *CGAL::Kernel_archetype*, so access to these other kernel types will result in a compile-time error. Another option is to use an own archetype class that encapsulates only the needed type definitions and the corresponding member functions. See the following code snippet for a simple example.

```
#include <CGAL/Kernel_archetype.h>

// build an own archetype class ...

// get needed types from the kernel archetype ...
typedef CGAL::Kernel_archetype      KA;
typedef KA::Point_3                  KA_Point_3;
typedef KA::Plane_3                  KA_Plane_3;
typedef KA::Construct_opposite_plane_3 KA_Construct_opposite_plane_3;
```

```
// reuse the types from the kernel archetype in the own archetype class
struct My_archetype {
    typedef KA_Point_3          Point_3;
    typedef KA_Plane_3         Plane_3;
    typedef KA_Construct_opposite_plane_3 Construct_opposite_plane_3;

    Construct_opposite_plane_3
    construct_opposite_plane_3_object()
    { return Construct_opposite_plane_3(); }
};
```

Example Program

The following example shows a program for checking the 2d convex hull algorithm of CGAL with the archetype. You can see the usage of the *CGAL::Kernel_archetype* that replaces a CGAL kernel that is normally used.

test_convex_hull_2.C :

```
#include <CGAL/basic.h>
#include <CGAL/convex_hull_2.h>
#include <CGAL/Kernel_archetype.h>
#include <list>

typedef CGAL::Kernel_archetype K;
typedef K::Point_2 Point_2;

int main()
{
    std::list<Point_2> input;

    Point_2 act;
    input.push_back(act);

    std::list<Point_2> output;

    K traits;

    CGAL::convex_hull_2(input.begin(), input.end(),
                        std::back_inserter(output), traits);
    return 0;
}
```


2D and 3D Kernel Reference Manual

Hervé Brönnimann, Andreas Fabri, Geert-Jan Giezeman, Susan Hert, Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Stefan Schirra

The following pages give a complete overview on the functionality provided in the kernel.

Concepts

Kernel	page 35
EuclideanRingNumberType	page 2528
FieldNumberType	page 2530
RingNumberType	page 2576

Kernel Classes and Operations

<i>CGAL::Cartesian<FieldNumberType></i>	page 41
<i>CGAL::Cartesian_converter<K1, K2, NTConverter></i>	page 42
<i>CGAL::cartesian_to_homogeneous</i>	page 43
<i>CGAL::Filtered_kernel<CK, EK, FK, C2E, C2F></i>	page ??
<i>CGAL::Filtered_predicate<EP, FP, C2E, C2F></i>	page 46
<i>CGAL::Homogeneous<RingNumberType></i>	page 48
<i>CGAL::Homogeneous_converter<K1, K2, RTConverter, FTConverter></i>	page 49
<i>CGAL::homogeneous_to_cartesian</i>	page 50
<i>CGAL::homogeneous_to_quotient_cartesian</i>	page 51
<i>CGAL::quotient_cartesian_to_homogeneous</i>	page 194
<i>CGAL::Simple_cartesian<FieldNumberType></i>	page 55
<i>CGAL::Simple_homogeneous<RingNumberType></i>	page 56

Predefined Kernels

<i>CGAL::Exact_predicates_exact_constructions_kernel</i>	page 57
<i>CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt</i>	page 58
<i>CGAL::Exact_predicates_inexact_constructions_kernel</i>	page 59

Classes for 2D Geometry

<i>CGAL::Aff_transformation_2<Kernel></i>	page 60
<i>CGAL::Bbox_2</i>	page 64
<i>CGAL::Circle_2<Kernel></i>	page 65
<i>CGAL::Direction_2<Kernel></i>	page 67
<i>CGAL::Iso_rectangle_2<Kernel></i>	page 69
<i>CGAL::Line_2<Kernel></i>	page 72
<i>CGAL::Point_2<Kernel></i>	page 75
<i>CGAL::Ray_2<Kernel></i>	page 79
<i>CGAL::Segment_2<Kernel></i>	page 81
<i>CGAL::Triangle_2<Kernel></i>	page 83
<i>CGAL::Vector_2<Kernel></i>	page 85

Classes for 3D Geometry

<i>CGAL::Aff_transformation_3<Kernel></i>	page 89
<i>CGAL::Bbox_3</i>	page 88
<i>CGAL::Direction_3<Kernel></i>	page 92
<i>CGAL::Iso_cuboid_3<Kernel></i>	page 94
<i>CGAL::Line_3<Kernel></i>	page 97
<i>CGAL::Point_3<Kernel></i>	page 102
<i>CGAL::Plane_3<Kernel></i>	page 99
<i>CGAL::Ray_3<Kernel></i>	page 105
<i>CGAL::Segment_3<Kernel></i>	page 107
<i>CGAL::Sphere_3<Kernel></i>	page 109
<i>CGAL::Tetrahedron_3<Kernel></i>	page 112
<i>CGAL::Triangle_3<Kernel></i>	page 114
<i>CGAL::Vector_3<Kernel></i>	page 116

Classes for dD Geometry

<i>CGAL::Point_d<Kernel></i>	page 447
------------------------------------------	----------

Polymorphic Return Type Class

<i>CGAL::Object</i>	page 120
---------------------------	----------

Constants and Enumerations

<i>CGAL::Angle</i>	page 123
<i>CGAL::Bounded_side</i>	page 123
<i>CGAL::CLOCKWISE</i>	page 126

<i>CGAL::Comparison_result</i>	page 124
<i>CGAL::COUNTERCLOCKWISE</i>	page 126
<i>CGAL::COLLINEAR</i>	page 127
<i>CGAL::COPLANAR</i>	page 128
<i>CGAL::DEGENERATE</i>	page 128
<i>CGAL::LEFT_TURN</i>	page 127
<i>CGAL::RIGHT_TURN</i>	page 127
<i>CGAL::NULL_VECTOR</i>	page 129
<i>CGAL::Null_vector</i>	page 129
<i>CGAL::Orientation</i>	page 125
<i>CGAL::Oriented_side</i>	page 125
<i>CGAL::Origin</i>	page 130
<i>CGAL::ORIGIN</i>	page 130
<i>CGAL::Identity_transformation</i>	page 132
<i>CGAL::Reflection</i>	page 132
<i>CGAL::Sign</i>	page 124
<i>CGAL::Rotation</i>	page 133
<i>CGAL::Scaling</i>	page 133
<i>CGAL::Translation</i>	page 134

Distance and Intersection Operations

<i>CGAL::do_intersect</i>	page 943
<i>CGAL::intersection</i>	page 945
<i>CGAL::squared_distance</i>	page 498

Predicates and Constructions

<i>CGAL::angle</i>	page 135
<i>CGAL::are_ordered_along_line</i>	page 136
<i>CGAL::are_strictly_ordered_along_line</i>	page 137
<i>CGAL::area</i>	page 138
<i>CGAL::bisector</i>	page 139
<i>CGAL::centroid</i>	page 2345
<i>CGAL::circumcenter</i>	page 141
<i>CGAL::compare_distance_to_point</i>	page 145
<i>CGAL::compare_signed_distance_to_line</i>	page 146
<i>CGAL::compare_signed_distance_to_plane</i>	page 147
<i>CGAL::collinear</i>	page 142
<i>CGAL::collinear_are_ordered_along_line</i>	page 143
<i>CGAL::collinear_are_strictly_ordered_along_line</i>	page 144
<i>CGAL::compare_slopes</i>	page 148
<i>CGAL::compare_x</i>	page 149
<i>CGAL::compare_x_at_y</i>	page 153
<i>CGAL::compare_xy</i>	page 151
<i>CGAL::compare_xyz</i>	page 152
<i>CGAL::compare_y</i>	page 155
<i>CGAL::compare_y_at_x</i>	page 157
<i>CGAL::compare_yx</i>	page 159

<i>CGAL::compare_z</i>	page 160
<i>CGAL::coplanar</i>	page 161
<i>CGAL::coplanar_orientation</i>	page 162
<i>CGAL::coplanar_side_of_bounded_circle</i>	page 163
<i>CGAL::cross_product</i>	page 164
<i>CGAL::do_overlap</i>	page 166
<i>CGAL::has_larger_distance_to_point</i>	page 168
<i>CGAL::has_larger_signed_distance_to_line</i>	page 169
<i>CGAL::has_larger_signed_distance_to_plane</i>	page 170
<i>CGAL::has_smaller_distance_to_point</i>	page 171
<i>CGAL::has_smaller_signed_distance_to_line</i>	page 172
<i>CGAL::has_smaller_signed_distance_to_plane</i>	page 173
<i>CGAL::left_turn</i>	page 177
<i>CGAL::lexicographically_xyz_smaller</i>	page 178
<i>CGAL::lexicographically_xyz_smaller_or_equal</i>	page 179
<i>CGAL::lexicographically_xy_larger</i>	page 180
<i>CGAL::lexicographically_xy_larger_or_equal</i>	page 181
<i>CGAL::lexicographically_xy_smaller</i>	page 182
<i>CGAL::lexicographically_xy_smaller_or_equal</i>	page 183
<i>CGAL::midpoint</i>	page 493
<i>CGAL::operator+</i>	page 187
<i>CGAL::operator-</i>	page 188
<i>CGAL::operator*</i>	page 189
<i>CGAL::opposite</i>	page 190
<i>CGAL::orientation</i>	page 494
<i>CGAL::parallel</i>	page 193
<i>CGAL::rational_rotation_approximation</i>	page 195
<i>CGAL::right_turn</i>	page 196
<i>CGAL::side_of_bounded_circle</i>	page 197
<i>CGAL::side_of_bounded_sphere</i>	page 496
<i>CGAL::side_of_oriented_circle</i>	page 199
<i>CGAL::side_of_oriented_sphere</i>	page 497
<i>CGAL::squared_radius</i>	page 202
<i>CGAL::volume</i>	page 203
<i>CGAL::x_equal</i>	page 204
<i>CGAL::y_equal</i>	page 205
<i>CGAL::z_equal</i>	page 206

Tag Classes

<i>CGAL::Tag_false</i>	page 426
<i>CGAL::Tag_true</i>	page 426

2.7 Concepts

Kernel

The concept of a *kernel* is defined by a set of requirements on the provision of certain types and access member functions to create objects of these types. The types are function object classes to be used within the algorithms and data structures in the basic library of CGAL. This allows you to use any model of a kernel as a traits class in the CGAL algorithms and data structures, unless they require types beyond those provided by a kernel.

A kernel provides types, construction objects, and generalized predicates. The former replace constructors of the kernel classes and constructive procedures in the kernel. There are also function objects replacing operators, especially for equality testing.

Has Models

<i>CGAL::Cartesian</i> < <i>FieldNumberType</i> >	page 41
<i>CGAL::Homogeneous</i> < <i>RingNumberType</i> >	page 48
<i>CGAL::Simple_cartesian</i> < <i>FieldNumberType</i> >	page 55
<i>CGAL::Simple_homogeneous</i> < <i>RingNumberType</i> >	page 56

Types

<i>Kernel::FT</i>	a model of <i>FieldNumberType</i>
<i>Kernel::RT</i>	a model of <i>RingNumberType</i>

Two-dimensional Kernel

Coordinate Access

<i>Kernel::Cartesian_const_iterator_2</i>	a model of <i>Kernel::CartesianConstIterator_2</i>
-------------------------------------------	----------------------------------------------------

Geometric Objects

<i>Kernel::Point_2</i>	a model of <i>Kernel::Point_2</i>
<i>Kernel::Vector_2</i>	a model of <i>Kernel::Vector_2</i>
<i>Kernel::Direction_2</i>	a model of <i>Kernel::Direction_2</i>
<i>Kernel::Line_2</i>	a model of <i>Kernel::Line_2</i>
<i>Kernel::Ray_2</i>	a model of <i>Kernel::Ray_2</i>
<i>Kernel::Segment_2</i>	a model of <i>Kernel::Segment_2</i>
<i>Kernel::Triangle_2</i>	a model of <i>Kernel::Triangle_2</i>
<i>Kernel::Iso_rectangle_2</i>	a model of <i>Kernel::IsoRectangle_2</i>
<i>Kernel::Circle_2</i>	a model of <i>Kernel::Circle_2</i>
<i>Kernel::Object_2</i>	a model of <i>Kernel::Object_2</i>

Constructions

<i>Kernel:: Construct_point_2</i>	a model of <i>Kernel::ConstructPoint_2</i>
<i>Kernel:: Construct_vector_2</i>	a model of <i>Kernel::ConstructVector_2</i>
<i>Kernel:: Construct_direction_2</i>	a model of <i>Kernel::ConstructDirection_2</i>
<i>Kernel:: Construct_segment_2</i>	a model of <i>Kernel::ConstructSegment_2</i>
<i>Kernel:: Construct_line_2</i>	a model of <i>Kernel::ConstructLine_2</i>
<i>Kernel:: Construct_ray_2</i>	a model of <i>Kernel::ConstructRay_2</i>
<i>Kernel:: Construct_circle_2</i>	a model of <i>Kernel::ConstructCircle_2</i>
<i>Kernel:: Construct_triangle_2</i>	a model of <i>Kernel::ConstructTriangle_2</i>
<i>Kernel:: Construct_iso_rectangle_2</i>	a model of <i>Kernel::ConstructIsoRectangle_2</i>
<i>Kernel:: Construct_object_2</i>	a model of <i>Kernel::ConstructObject_2</i>
<i>Kernel:: Construct_scaled_vector_2</i>	a model of <i>Kernel::ConstructScaledVector_2</i>
<i>Kernel:: Construct_translated_point_2</i>	a model of <i>Kernel::ConstructTranslatedPoint_2</i>
<i>Kernel:: Construct_point_on_2</i>	a model of <i>Kernel::ConstructPointOn_2</i>
<i>Kernel:: Construct_projected_point_2</i>	a model of <i>Kernel::ConstructProjectedPoint_2</i>
<i>Kernel:: Construct_projected_xy_point_2</i>	a model of <i>Kernel::ConstructProjectedXYPoint_2</i>
<i>Kernel:: Construct_cartesian_const_iterator_2</i>	a model of <i>Kernel::ConstructCartesianConstIterator_2</i>
<i>Kernel:: Construct_vertex_2</i>	a model of <i>Kernel::ConstructVertex_2</i>
<i>Kernel:: Construct_bbox_2</i>	a model of <i>Kernel::ConstructBbox_2</i>
<i>Kernel:: Construct_perpendicular_vector_2</i>	a model of <i>Kernel::ConstructPerpendicularVector_2</i>
<i>Kernel:: Construct_perpendicular_direction_2</i>	a model of <i>Kernel::ConstructPerpendicularDirection_2</i>
<i>Kernel:: Construct_perpendicular_line_2</i>	a model of <i>Kernel::ConstructPerpendicularLine_2</i>
<i>Kernel:: Construct_max_vertex_2</i>	a model of <i>Kernel::ConstructMaxVertex_2</i>
<i>Kernel:: Construct_midpoint_2</i>	a model of <i>Kernel::ConstructMidpoint_2</i>
<i>Kernel:: Construct_min_vertex_2</i>	a model of <i>Kernel::ConstructMinVertex_2</i>
<i>Kernel:: Construct_center_2</i>	a model of <i>Kernel::ConstructCenter_2</i>
<i>Kernel:: Construct_centroid_2</i>	a model of <i>Kernel::ConstructCentroid_2</i>
<i>Kernel:: Construct_circumcenter_2</i>	a model of <i>Kernel::ConstructCircumcenter_2</i>
<i>Kernel:: Construct_bisector_2</i>	a model of <i>Kernel::ConstructBisector_2</i>
<i>Kernel:: Construct_opposite_direction_2</i>	a model of <i>Kernel::ConstructOppositeDirection_2</i>
<i>Kernel:: Construct_opposite_segment_2</i>	a model of <i>Kernel::ConstructOppositeSegment_2</i>
<i>Kernel:: Construct_opposite_ray_2</i>	a model of <i>Kernel::ConstructOppositeRay_2</i>
<i>Kernel:: Construct_opposite_line_2</i>	a model of <i>Kernel::ConstructOppositeLine_2</i>
<i>Kernel:: Construct_opposite_triangle_2</i>	a model of <i>Kernel::ConstructOppositeTriangle_2</i>
<i>Kernel:: Construct_opposite_circle_2</i>	a model of <i>Kernel::ConstructOppositeCircle_2</i>
<i>Kernel:: Construct_opposite_vector_2</i>	a model of <i>Kernel::ConstructOppositeVector_2</i>

If the result type is not determined, there is no *Construct_* prefix:

<i>Kernel:: Intersect_2</i>	a model of <i>Kernel::Intersect_2</i>
<i>Kernel:: Assign_2</i>	a model of <i>Kernel::Assign_2</i>

If the result type is a number type, the prefix is *Compute_*:

<i>Kernel:: Compute_squared_distance_2</i>	a model of <i>Kernel::ComputeSquaredDistance_2</i>
<i>Kernel:: Compute_squared_length_2</i>	a model of <i>Kernel::ComputeSquaredLength_2</i>
<i>Kernel:: Compute_squared_radius_2</i>	a model of <i>Kernel::ComputeSquaredRadius_2</i>
<i>Kernel:: Compute_area_2</i>	a model of <i>Kernel::ComputeArea_2</i>

Generalized Predicates

<i>Kernel:: Angle_2</i>	a model of <i>Kernel::Angle_2</i>
-------------------------	-----------------------------------

<i>Kernel:: Equal_2</i>	a model of <i>Kernel::Equal_2</i>
<i>Kernel:: Equal_x_2</i>	a model of <i>Kernel::EqualX_2</i>
<i>Kernel:: Equal_y_2</i>	a model of <i>Kernel::EqualY_2</i>
<i>Kernel:: Less_x_2</i>	a model of <i>Kernel::LessX_2</i>
<i>Kernel:: Less_y_2</i>	a model of <i>Kernel::LessY_2</i>
<i>Kernel:: Less_xy_2</i>	a model of <i>Kernel::LessXY_2</i>
<i>Kernel:: Less_yx_2</i>	a model of <i>Kernel::LessYX_2</i>
<i>Kernel:: Compare_x_2</i>	a model of <i>Kernel::CompareX_2</i>
<i>Kernel:: Compare_x_at_y_2</i>	a model of <i>Kernel::CompareXAtY_2</i>
<i>Kernel:: Compare_y_2</i>	a model of <i>Kernel::CompareY_2</i>
<i>Kernel:: Compare_xy_2</i>	a model of <i>Kernel::CompareXY_2</i>
<i>Kernel:: Compare_y_at_x_2</i>	a model of <i>Kernel::CompareYAtX_2</i>
<i>Kernel:: Compare_distance_2</i>	a model of <i>Kernel::CompareDistance_2</i>
<i>Kernel:: Compare_angle_with_x_axis_2</i>	a model of <i>Kernel::CompareAngleWithXAxis_2</i>
<i>Kernel:: Compare_slope_2</i>	a model of <i>Kernel::CompareSlope_2</i>
<i>Kernel:: Less_distance_to_point_2</i>	a model of <i>Kernel::LessDistanceToPoint_2</i>
<i>Kernel:: Less_signed_distance_to_line_2</i>	a model of <i>Kernel::LessSignedDistanceToLine_2</i>
<i>Kernel:: Less_rotate_ccw_2</i>	a model of <i>Kernel::LessRotateCCW_2</i>
<i>Kernel:: Left_turn_2</i>	a model of <i>Kernel::LeftTurn_2</i>
<i>Kernel:: Collinear_2</i>	a model of <i>Kernel::Collinear_2</i>
<i>Kernel:: Orientation_2</i>	a model of <i>Kernel::Orientation_2</i>
<i>Kernel:: Side_of_oriented_circle_2</i>	a model of <i>Kernel::SideOfOrientedCircle_2</i>
<i>Kernel:: Side_of_bounded_circle_2</i>	a model of <i>Kernel::SideOfBoundedCircle_2</i>
<i>Kernel:: Is_horizontal_2</i>	a model of <i>Kernel::IsHorizontal_2</i>
<i>Kernel:: Is_vertical_2</i>	a model of <i>Kernel::IsVertical_2</i>
<i>Kernel:: Is_degenerate_2</i>	a model of <i>Kernel::IsDegenerate_2</i>
<i>Kernel:: Has_on_2</i>	a model of <i>Kernel::HasOn_2</i>
<i>Kernel:: Collinear_has_on_2</i>	a model of <i>Kernel::CollinearHasOn_2</i>
<i>Kernel:: Has_on_bounded_side_2</i>	a model of <i>Kernel::HasOnBoundedSide_2</i>
<i>Kernel:: Has_on_unbounded_side_2</i>	a model of <i>Kernel::HasOnUnboundedSide_2</i>
<i>Kernel:: Has_on_boundary_2</i>	a model of <i>Kernel::HasOnBoundary_2</i>
<i>Kernel:: Has_on_positive_side_2</i>	a model of <i>Kernel::HasOnPositiveSide_2</i>
<i>Kernel:: Has_on_negative_side_2</i>	a model of <i>Kernel::HasOnNegativeSide_2</i>
<i>Kernel:: Oriented_side_2</i>	a model of <i>Kernel::OrientedSide_2</i>
<i>Kernel:: Bounded_side_2</i>	a model of <i>Kernel::BoundedSide_2</i>
<i>Kernel:: Are_parallel_2</i>	a model of <i>Kernel::AreParallel_2</i>
<i>Kernel:: Are_ordered_along_line_2</i>	a model of <i>Kernel::AreOrderedAlongLine_2</i>
<i>Kernel:: Are_strictly_ordered_along_line_2</i>	a model of <i>Kernel::AreStrictlyOrderedAlongLine_2</i>
<i>Kernel:: Collinear_are_ordered_along_line_2</i>	a model of <i>Kernel::CollinearAreOrderedAlongLine_2</i>
<i>Kernel:: Collinear_are_strictly_ordered_along_line_2</i>	a model of <i>Kernel::CollinearAreStrictlyOrderedAlongLine_2</i>
<i>Kernel:: Counterclockwise_in_between_2</i>	a model of <i>Kernel::CounterclockwiseInBetween_2</i>
<i>Kernel:: Do_intersect_2</i>	a model of <i>Kernel::DoIntersect_2</i>

Three-dimensional Kernel

Coordinate Access

<i>Kernel:: Cartesian_const_iterator_3</i>	a model of <i>Kernel::CartesianConstIterator_3</i>
--------------------------------------------	----------------------------------------------------

Geometric Objects

<i>Kernel:: Point_3</i>	a model of <i>Kernel::Point_3</i>
<i>Kernel:: Vector_3</i>	a model of <i>Kernel::Vector_3</i>
<i>Kernel:: Direction_3</i>	a model of <i>Kernel::Direction_3</i>
<i>Kernel:: Iso_cuboid_3</i>	a model of <i>Kernel::IsoCuboid_3</i>
<i>Kernel:: Line_3</i>	a model of <i>Kernel::Line_3</i>
<i>Kernel:: Ray_3</i>	a model of <i>Kernel::Ray_3</i>
<i>Kernel:: Sphere_3</i>	a model of <i>Kernel::Sphere_3</i>
<i>Kernel:: Segment_3</i>	a model of <i>Kernel::Segment_3</i>
<i>Kernel:: Plane_3</i>	a model of <i>Kernel::Plane_3</i>
<i>Kernel:: Triangle_3</i>	a model of <i>Kernel::Triangle_3</i>
<i>Kernel:: Tetrahedron_3</i>	a model of <i>Kernel::Tetrahedron_3</i>
<i>Kernel:: Object_3</i>	a model of <i>Kernel::Object_3</i>

Constructions

<i>Kernel:: Construct_point_3</i>	a model of <i>Kernel::ConstructPoint_3</i>
<i>Kernel:: Construct_vector_3</i>	a model of <i>Kernel::ConstructVector_3</i>
<i>Kernel:: Construct_direction_3</i>	a model of <i>Kernel::ConstructDirection_3</i>
<i>Kernel:: Construct_plane_3</i>	a model of <i>Kernel::ConstructPlane_3</i>
<i>Kernel:: Construct_iso_cuboid_3</i>	a model of <i>Kernel::ConstructIsoCuboid_3</i>
<i>Kernel:: Construct_line_3</i>	a model of <i>Kernel::ConstructLine_3</i>
<i>Kernel:: Construct_ray_3</i>	a model of <i>Kernel::ConstructRay_3</i>
<i>Kernel:: Construct_sphere_3</i>	a model of <i>Kernel::ConstructSphere_3</i>
<i>Kernel:: Construct_segment_3</i>	a model of <i>Kernel::ConstructSegment_3</i>
<i>Kernel:: Construct_triangle_3</i>	a model of <i>Kernel::ConstructTriangle_3</i>
<i>Kernel:: Construct_tetrahedron_3</i>	a model of <i>Kernel::ConstructTetrahedron_3</i>
<i>Kernel:: Construct_object_3</i>	a model of <i>Kernel::ConstructObject_3</i>
<i>Kernel:: Construct_scaled_vector_3</i>	a model of <i>Kernel::ConstructScaledVector_3</i>
<i>Kernel:: Construct_translated_point_3</i>	a model of <i>Kernel::ConstructTranslatedPoint_3</i>
<i>Kernel:: Construct_point_on_3</i>	a model of <i>Kernel::ConstructPointOn_3</i>
<i>Kernel:: Construct_projected_point_3</i>	a model of <i>Kernel::ConstructProjectedPoint_3</i>
<i>Kernel:: Construct_lifted_point_3</i>	a model of <i>Kernel::ConstructLiftedPoint_3</i>
<i>Kernel:: Construct_cartesian_const_iterator_3</i>	a model of <i>Kernel::ConstructCartesianConstIterator_3</i>
<i>Kernel:: Construct_vertex_3</i>	a model of <i>Kernel::ConstructVertex_3</i>
<i>Kernel:: Construct_bbox_3</i>	a model of <i>Kernel::ConstructBbox_3</i>
<i>Kernel:: Construct_supporting_plane_3</i>	a model of <i>Kernel::ConstructSupportingPlane_3</i>
<i>Kernel:: Construct_orthogonal_vector_3</i>	a model of <i>Kernel::ConstructOrthogonalVector_3</i>
<i>Kernel:: Construct_base_vector_3</i>	a model of <i>Kernel::ConstructBaseVector_3</i>
<i>Kernel:: Construct_perpendicular_plane_3</i>	a model of <i>Kernel::ConstructPerpendicularPlane_3</i>
<i>Kernel:: Construct_perpendicular_line_3</i>	a model of <i>Kernel::ConstructPerpendicularLine_3</i>
<i>Kernel:: Construct_midpoint_3</i>	a model of <i>Kernel::ConstructMidpoint_3</i>
<i>Kernel:: Construct_center_3</i>	a model of <i>Kernel::ConstructCenter_3</i>
<i>Kernel:: Construct_centroid_3</i>	a model of <i>Kernel::ConstructCentroid_3</i>
<i>Kernel:: Construct_circumcenter_3</i>	a model of <i>Kernel::ConstructCircumcenter_3</i>
<i>Kernel:: Construct_bisector_3</i>	a model of <i>Kernel::ConstructBisector_3</i>
<i>Kernel:: Construct_cross_product_vector_3</i>	a model of <i>Kernel::ConstructCrossProductVector_3</i>
<i>Kernel:: Construct_opposite_direction_3</i>	a model of <i>Kernel::ConstructOppositeDirection_3</i>
<i>Kernel:: Construct_opposite_segment_3</i>	a model of <i>Kernel::ConstructOppositeSegment_3</i>
<i>Kernel:: Construct_opposite_ray_3</i>	a model of <i>Kernel::ConstructOppositeRay_3</i>
<i>Kernel:: Construct_opposite_line_3</i>	a model of <i>Kernel::ConstructOppositeLine_3</i>
<i>Kernel:: Construct_opposite_plane_3</i>	a model of <i>Kernel::ConstructOppositePlane_3</i>

<i>Kernel:: Construct_opposite_sphere_3</i>	a model of <i>Kernel::ConstructOppositeSphere_3</i>
<i>Kernel:: Construct_opposite_vector_3</i>	a model of <i>Kernel::ConstructOppositeVector_3</i>

If the result type is not determined, there is no *Construct_* prefix:

<i>Kernel:: Intersect_3</i>	a model of <i>Kernel::Intersect_3</i>
<i>Kernel:: Assign_3</i>	a model of <i>Kernel::Assign_3</i>

If the result type is a number type, the prefix is *Compute_*:

<i>Kernel:: Compute_area_3</i>	a model of <i>Kernel::ComputeArea_3</i>
<i>Kernel:: Compute_squared_area_3</i>	a model of <i>Kernel::ComputeSquaredArea_3</i>
<i>Kernel:: Compute_squared_distance_3</i>	a model of <i>Kernel::ComputeSquaredDistance_3</i>
<i>Kernel:: Compute_squared_length_3</i>	a model of <i>Kernel::ComputeSquaredLength_3</i>
<i>Kernel:: Compute_squared_radius_3</i>	a model of <i>Kernel::ComputeSquaredRadius_3</i>
<i>Kernel:: Compute_volume_3</i>	a model of <i>Kernel::ComputeVolume_3</i>

Generalized Predicates

<i>Kernel:: Angle_3</i>	a model of <i>Kernel::Angle_3</i>
<i>Kernel:: Equal_3</i>	a model of <i>Kernel::Equal_3</i>
<i>Kernel:: Equal_x_3</i>	a model of <i>Kernel::EqualX_3</i>
<i>Kernel:: Equal_y_3</i>	a model of <i>Kernel::EqualY_3</i>
<i>Kernel:: Equal_z_3</i>	a model of <i>Kernel::EqualZ_3</i>
<i>Kernel:: Equal_xy_3</i>	a model of <i>Kernel::EqualXY_3</i>
<i>Kernel:: Less_x_3</i>	a model of <i>Kernel::LessX_3</i>
<i>Kernel:: Less_y_3</i>	a model of <i>Kernel::LessY_3</i>
<i>Kernel:: Less_z_3</i>	a model of <i>Kernel::LessZ_3</i>
<i>Kernel:: Less_xy_3</i>	a model of <i>Kernel::LessXY_3</i>
<i>Kernel:: Less_xyz_3</i>	a model of <i>Kernel::LessXYZ_3</i>
<i>Kernel:: Compare_x_3</i>	a model of <i>Kernel::CompareX_3</i>
<i>Kernel:: Compare_y_3</i>	a model of <i>Kernel::CompareY_3</i>
<i>Kernel:: Compare_z_3</i>	a model of <i>Kernel::CompareZ_3</i>
<i>Kernel:: Compare_xy_3</i>	a model of <i>Kernel::CompareXY_3</i>
<i>Kernel:: Compare_xyz_3</i>	a model of <i>Kernel::CompareXYZ_3</i>
<i>Kernel:: Less_signed_distance_to_plane_3</i>	a model of <i>Kernel::LessSignedDistanceToPlane_3</i>
<i>Kernel:: Less_distance_to_point_3</i>	a model of <i>Kernel::LessDistanceToPoint_3</i>
<i>Kernel:: Compare_distance_3</i>	a model of <i>Kernel::CompareDistance_3</i>
<i>Kernel:: Collinear_3</i>	a model of <i>Kernel::Collinear_3</i>
<i>Kernel:: Coplanar_3</i>	a model of <i>Kernel::Coplanar_3</i>
<i>Kernel:: Orientation_3</i>	a model of <i>Kernel::Orientation_3</i>
<i>Kernel:: Coplanar_orientation_3</i>	a model of <i>Kernel::CoplanarOrientation_3</i>
<i>Kernel:: Coplanar_side_of_bounded_circle_3</i>	a model of <i>Kernel::CoplanarSideOfBoundedCircle_3</i>
<i>Kernel:: Side_of_oriented_sphere_3</i>	a model of <i>Kernel::SideOfOrientedSphere_3</i>
<i>Kernel:: Side_of_bounded_sphere_3</i>	a model of <i>Kernel::SideOfBoundedSphere_3</i>
<i>Kernel:: Is_degenerate_3</i>	a model of <i>Kernel::IsDegenerate_3</i>
<i>Kernel:: Has_on_3</i>	a model of <i>Kernel::HasOn_3</i>
<i>Kernel:: Has_on_bounded_side_3</i>	a model of <i>Kernel::HasOnBoundedSide_3</i>
<i>Kernel:: Has_on_unbounded_side_3</i>	a model of <i>Kernel::HasOnUnboundedSide_3</i>
<i>Kernel:: Has_on_boundary_3</i>	a model of <i>Kernel::HasOnBoundary_3</i>
<i>Kernel:: Has_on_positive_side_3</i>	a model of <i>Kernel::HasOnPositiveSide_3</i>
<i>Kernel:: Has_on_negative_side_3</i>	a model of <i>Kernel::HasOnNegativeSide_3</i>
<i>Kernel:: Oriented_side_3</i>	a model of <i>Kernel::OrientedSide_3</i>

<i>Kernel:: Bounded_side_3</i>	a model of <i>Kernel::BoundedSide_3</i>
<i>Kernel:: Are_parallel_3</i>	a model of <i>Kernel::AreParallel_3</i>
<i>Kernel:: Are_ordered_along_line_3</i>	a model of <i>Kernel::AreOrderedAlongLine_3</i>
<i>Kernel:: Are_strictly_ordered_along_line_3</i>	a model of <i>Kernel::AreStrictlyOrderedAlongLine_3</i>
<i>Kernel:: Collinear_are_ordered_along_line_3</i>	a model of <i>Kernel::CollinearAreOrderedAlongLine_3</i>
<i>Kernel:: Collinear_are_strictly_ordered_along_line_3</i>	a model of <i>Kernel::CollinearAreStrictlyOrderedAlongLine_3</i>
<i>Kernel:: Do_intersect_3</i>	a model of <i>Kernel::DoIntersect_3</i>

d-dimensional Kernel

Coordinate Access

<i>Kernel:: Cartesian_const_iterator_d</i>	a model of <i>Kernel::CartesianConstIterator_d</i>
--------------------------------------------	----------------------------------------------------

Geometric Objects

<i>Kernel:: Point_d</i>	a model of <i>Kernel::Point_d</i>
-------------------------	-----------------------------------

Constructions

<i>Kernel:: Construct_Point_d</i>	a model of <i>Kernel::ConstructPoint_d</i>
-----------------------------------	--------------------------------------------

Operations

For each of the function objects above, there must exist a member function that requires no arguments and returns an instance of that function object. The name of the member function is the uncapitalized name of the type returned with the suffix *_object* appended. For example, for the function object *Kernel::Construct_vector_2* the following member function must exist:

<i>Kernel::Construct_vector_2</i>	<i>kernel.construct_vector_2_object()</i>
-----------------------------------	-------------------------------------------

See Also

Kernel_d

2.8 Kernel Classes and Operations

CGAL::Cartesian<FieldNumberType>

```
#include <CGAL/Cartesian.h>
```

Definition

A model for *Kernel* that uses Cartesian coordinates to represent the geometric objects. In order for *Cartesian*<*FieldNumberType*> to model Euclidean geometry in E^2 and/or E^3 , for some mathematical field E (e.g., the rationals \mathbb{Q} or the reals \mathbb{R}), the template parameter *FieldNumberType* must model the mathematical field E . That is, the field operations on this number type must compute the mathematically correct results. If the number type provided as a model for *FieldNumberType* is only an approximation of a field (such as the built-in type *double*), then the geometry provided by the kernel is only an approximation of Euclidean geometry.

Is Model for the Concepts

Kernel page [35](#)

Types

```
typedef FieldNumberType    FT;
typedef FieldNumberType    RT;
```

Implementation

All geometric objects in *Cartesian*<*FieldNumberType*> are reference counted.

See Also

CGAL::Simple_cartesian<*FieldNumberType*> page [55](#)
CGAL::Homogeneous<*RingNumberType*> page [48](#)
CGAL::Simple_homogeneous<*RingNumberType*> page [56](#)

CGAL::Cartesian_converter<K1, K2, NTConverter>

Definition

Cartesian_converter<*K1*, *K2*, *NTConverter*> converts objects from the kernel traits *K1* to the kernel traits *K2* using *Converter* to do the conversion. Those traits must be of the form *Cartesian*<*FT1*> and *Cartesian*<*FT2*> (or the equivalent with *Simple_cartesian*). It then provides the following operators to convert objects from *K1* to *K2*.

The third template parameter *NTConverter* is a function object that must provide *K2::FT operator()(K1::FT n)* that converts *n* to an *K2::FT* which has the same value.

The default value of this parameter is *CGAL::NT_converter*<*K1::FT*, *K2::FT*>.

```
#include <CGAL/Cartesian_converter.h>
```

Creation

```
Cartesian_converter<K1, K2, NTConverter> conv;
```

Default constructor.

Operations

```
K2::Point_2                      conv.operator()( K1::Point_2 p)
```

returns a *K2::Point_2* which coordinates are those of *p*, converted by *NTConverter*.

Similar operators are defined for the other kernel traits types *Point_3*, *Vector_2*...

See Also

CGAL::Cartesian<*FieldNumberType*> page [41](#)
CGAL::Simple_cartesian<*FieldNumberType*> page [55](#)

CGAL::cartesian_to_homogeneous

```
#include <CGAL/cartesian_homogeneous_conversion.h>
```

```
Point_2< Homogeneous<RT> > cartesian_to_homogeneous( Point_2< Cartesian<RT> > cp)
```

converts 2d point *cp* with Cartesian representation into a 2d point with homogeneous representation with the same number type.

```
Point_3< Homogeneous<RT> > cartesian_to_homogeneous( Point_3< Cartesian<RT> > cp)
```

converts 3d point *cp* with Cartesian representation into a 3d point with homogeneous representation with the same number type.

See Also

<i>CGAL::Cartesian<FieldNumberType></i>	page 41
<i>CGAL::Cartesian_converter<K1, K2, NTConverter></i>	page 42
<i>CGAL::Homogeneous<RingNumberType></i>	page 48
<i>CGAL::Homogeneous_converter<K1, K2, RTConverter, FTConverter></i>	page 49
<i>CGAL::homogeneous_to_cartesian</i>	page 50
<i>CGAL::homogeneous_to_quotient_cartesian</i>	page 51
<i>CGAL::quotient_cartesian_to_homogeneous</i>	page 194
<i>CGAL::Simple_cartesian<FieldNumberType></i>	page 55
<i>CGAL::Simple_homogeneous<RingNumberType></i>	page 56

CGAL::Filtered_kernel<CK>

Definition

Filtered_kernel<CK> is a kernel that uses the filtering technique [BBP01] to achieve a kernel with exact and efficient predicates. The geometric constructions are exactly those of the kernel *CK*, which means that they are not necessarily exact.

Is Model for the Concepts

Kernel

```
#include <CGAL/Filtered_kernel.h>
```

Example

The following example shows how to produce a kernel whose geometric objects and constructions are those of *Simple_cartesian<double>* but the predicates are exact.

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Filtered_kernel.h>

typedef CGAL::Simple_cartesian<double> CK;
typedef CGAL::Filtered_kernel<CK> K;
```

Implementation

The implementation uses *CGAL::Filtered_predicate<EP, FP, C2E, C2F>* over each predicate of the kernel traits interface. Additionally, faster static filters are used for a few selected critical predicates. The static filters can be disabled by compiling with *-DCGAL_NO_STATIC_FILTERS*.

CGAL::Filtered_kernel_adaptor<CK>

Definition

Filtered_kernel_adaptor<CK> is a kernel that uses the filtering technique [BBP01] to obtain a kernel with exact and efficient predicate functors. The geometric constructions are exactly those of the kernel CK, which means that they are not necessarily exact.

In contrast to *Filtered_kernel*, the global functions are those of CK.

Is Model for the Concepts

Kernel

```
#include <CGAL/Filtered_kernel.h>
```

Example

The following example shows how to produce a kernel whose geometric objects and constructions are those of *Simple_cartesian*<double>. The predicate functors of the kernel are exact, the global functions are not.

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Filtered_kernel.h>

typedef CGAL::Simple_cartesian<double> CK;
typedef CGAL::Filtered_kernel_adaptor<CK> K;

typedef K::Point_2 p(0,0), q(1,1), r(1,5);

CGAL::orientation(p,q,r);           // not exact

typedef K::Orientation_2 orientation;
orientation(p,q,r);                 // exact
```

CGAL::Filtered_predicate<EP, FP, C2E, C2F>

Definition

Filtered_predicate<EP, FP, C2E, C2F> is an adaptor for predicate function objects that allows one to produce efficient and exact predicates. It is used to build *CGAL::Filtered_kernel*<CK, EK, FK, C2E, C2F> and can be used for other predicates too.

EP is the exact but supposedly slow predicate that is able to evaluate the predicate correctly. It will be called only when the filtering predicate, FP, cannot compute the correct result. This failure of FP must be done by throwing an exception.

To convert the geometric objects that are the arguments of the predicate, we use the function objects C2E and C2F, which must be of the form *Cartesian_converter* or *Homogeneous_converter*.

```
#include <CGAL/Filtered_predicate.h>
```

Types

```
typedef FP::result_type    result_type;
```

The return type of the function operators. It must also be the same type as *EP::result_type*.

Creation

```
Filtered_predicate<EP, FP, C2E, C2F> fo;
```

Default constructor.

Operations

```
template <class A1>
result_type    fo.operator()( A1 a1)
```

The unary function operator for unary predicates.

```
template <class A1, class A2>
result_type    fo.operator()( A1 a1, A2 a2)
```

The binary function operator for binary predicates.

Similar function operators are defined for up to 7 arguments.

Example

The following example defines an efficient and exact version of the orientation predicate over three points using the Cartesian representation with double coordinates and without reference counting (*Simple_cartesian*<*double*>::*Point_2*). Of course, the orientation predicate can already be found in the kernel, but you can follow this example to filter your own predicates. It uses the fast but inexact predicate based on interval arithmetic for filtering and the slow but exact predicate based on multi-precision floats when the filtering predicate fails.

```

#include <CGAL/Simple_cartesian.h>
#include <CGAL/Filtered_predicate.h>
#include <CGAL/MP_Float.h>
#include <CGAL/Cartesian_converter.h>

typedef CGAL::Simple_cartesian<double> K;
typedef CGAL::Simple_cartesian<CGAL::Interval_nt_advanced> FK;
typedef CGAL::Simple_cartesian<CGAL::MP_Float> EK;
typedef CGAL::Cartesian_converter<K, EK> C2E;
typedef CGAL::Cartesian_converter<K, FK> C2F;

// Define my predicate, parameterized by a kernel.
template < typename K >
struct My_orientation_2
{
    typedef typename K::RT      RT;
    typedef typename K::Point_2 Point_2;

    CGAL::Orientation
    operator()(const Point_2 &p, const Point_2 &q, const Point_2 &r) const
    {
        RT prx = p.x() - r.x();
        RT pry = p.y() - r.y();
        RT qrx = q.x() - r.x();
        RT qry = q.y() - r.y();
        return static_cast<CGAL::Orientation> ( CGAL::sign( prx*qry - qrx*pry ) );
    }
};

typedef CGAL::Filtered_predicate<My_orientation_2<EK>,
                                My_orientation_2<FK>, C2E, C2F> Orientation_2;

int main()
{
    K::Point_2 p(1,2), q(2,3), r(3,4);
    Orientation_2 orientation;
    orientation(p, q, r);
    return 0;
}

```

CGAL::Homogeneous<RingNumberType>

```
#include <CGAL/Homogeneous.h>
```

Definition

A model for a *Kernel* using homogeneous coordinates to represent the geometric objects. In order for *Homogeneous<RingNumberType>* to model Euclidean geometry in E^2 and/or E^3 , for some mathematical ring E (e.g., the integers \mathbb{Z} or the rationals \mathbb{Q}), the template parameter *RingNumberType* must model the mathematical ring E . That is, the ring operations on this number type must compute the mathematically correct results. If the number type provided as a model for *RingNumberType* is only an approximation of a ring (such as the built-in type *double*), then the geometry provided by the kernel is only an approximation of Euclidean geometry.

Is Model for the Concepts

Kernel page [35](#)

Types

```
typedef Quotient<RingNumberType>    FT;
typedef RingNumberType               RT;
```

Implementation

This model of a kernel uses reference counting.

See Also

CGAL::Cartesian<FieldNumberType> page [41](#)
CGAL::Simple_cartesian<FieldNumberType> page [55](#)
CGAL::Simple_homogeneous<RingNumberType> page [56](#)

CGAL::Homogeneous_converter<K1, K2, RTConverter, FTConverter>

Definition

Homogeneous_converter<*K1*, *K2*, *RTConverter*, *FTConverter*> converts objects from the kernel traits *K1* to the kernel traits *K2*. Those traits must be of the form *Homogeneous*<*RT1*> and *Homogeneous*<*RT2*> (or the equivalent with *Simple_homogeneous*). It then provides the following operators to convert objects from *K1* to *K2*.

The third template parameter *RT_Converter* is a function object that must provide *K2::RT operator()(const K1::RT &n)*; that converts *n* to an *K2::RT* that has the same value.

The default value of this parameter is *CGAL::NT_converter*<*K1::RT*, *K2::RT*>, which uses the conversion operator from *K1::RT* to *K2::RT*.

Similarly, the fourth template parameter must provide *K2::FT operator()(const K1::FT &n)*; that converts *n* to an *K2::FT* that has the same value. Its default value is *CGAL::NT_converter*<*K1::FT*, *K2::FT*>.

#include <CGAL/Homogeneous_converter.h>

Creation

Homogeneous_converter<*K1*, *K2*, *RTConverter*, *FTConverter*> *conv*;

Default constructor.

Operations

K2::Point_2 *conv.operator()(K1::Point_2 p)*

returns a *K2::Point_2* which coordinates are those of *p*, converted by *RTConverter*.

Similar operators are defined for the other kernel traits geometric types *Point_3*, *Vector_2*...

See Also

CGAL::Homogeneous<*RingNumberType*>page 48
CGAL::Simple_homogeneous<*RingNumberType*>page 56

CGAL::homogeneous_to_cartesian

```
#include <CGAL/cartesian_homogeneous_conversion.h>
```

```
Point_2< Cartesian<FT> >      homogeneous_to_cartesian( Point_2< Homogeneous<FT> > hp)
```

converts 2d point *hp* with homogeneous representation into a 2d point with Cartesian representation with the same number type.

```
Point_3< Cartesian<FT> >      homogeneous_to_cartesian( Point_3< Homogeneous<FT> > hp)
```

converts 3d point *hp* with homogeneous representation into a 3d point with Cartesian representation with the same number type.

See Also

See Also

<i>CGAL::Cartesian<FieldNumberType></i>	page 41
<i>CGAL::Cartesian_converter<K1, K2, NTConverter></i>	page 42
<i>CGAL::cartesian_to_homogeneous</i>	page 43
<i>CGAL::Homogeneous<RingNumberType></i>	page 48
<i>CGAL::Homogeneous_converter<K1, K2, RTConverter, FTConverter></i>	page 49
<i>CGAL::homogeneous_to_quotient_cartesian</i>	page 51
<i>CGAL::quotient_cartesian_to_homogeneous</i>	page 194
<i>CGAL::Simple_cartesian<FieldNumberType></i>	page 55
<i>CGAL::Simple_homogeneous<RingNumberType></i>	page 56

CGAL::homogeneous_to_quotient_cartesian

```
#include <CGAL/cartesian_homogeneous_conversion.h>
```

```
Point_2< Cartesian<Quotient<RT> > >
```

```
homogeneous_to_quotient_cartesian( Point_2<Homogeneous<RT> > hp)
```

converts the 2d point *hp* with homogeneous representation with number type *RT* into a 2d point with Cartesian representation with number type *Quotient<RT>*.

```
Point_3< Cartesian<Quotient<RT> > >
```

```
homogeneous_to_quotient_cartesian( Point_3<Homogeneous<RT> > hp)
```

converts the 3d point *hp* with homogeneous representation with number type *RT* into a 3d point with Cartesian representation with number type *Quotient<RT>*.

See Also

<i>CGAL::Cartesian<FieldNumberType></i>	page 41
<i>CGAL::Cartesian_converter<K1, K2, NTConverter></i>	page 42
<i>CGAL::cartesian_to_homogeneous</i>	page 43
<i>CGAL::Homogeneous<RingNumberType></i>	page 48
<i>CGAL::Homogeneous_converter<K1, K2, RTConverter, FTConverter></i>	page 49
<i>CGAL::homogeneous_to_cartesian</i>	page 50
<i>CGAL::quotient_cartesian_to_homogeneous</i>	page 194
<i>CGAL::Simple_cartesian<FieldNumberType></i>	page 55
<i>CGAL::Simple_homogeneous<RingNumberType></i>	page 56

CGAL::Kernel_archetype

Definition

CGAL::Kernel_archetype is a concept archetype (minimal model) for the CGAL kernel concept. It provides all functionality required by the CGAL kernel concept, but nothing more. It can be used for testing successful compilation of packages of the basic library with a minimal model. Deprecated kernel functionality is not supported. All geometrical types (like the 2d/3d point or segment types) of *CGAL::Kernel_archetype* have copy constructors, default constructors and an assignment operator, and nothing else. Comparison operators are by default not supported, but can be switched on by defining the macro *CGAL_CONCEPT_ARCHETYPE_ALLOW_COMPARISONS*.

The geometrical types of the concept archetype encapsulate no data members, so runtime checks with the archetype are not very useful (*CGAL::Kernel_archetype* is only meant for compilation checks with a minimal model in the testsuites of CGAL packages).

The package supports the two- and three-dimensional part of the CGAL kernel concept. The d-dimensional part is not supported.

CGAL::Kernel_archetype normally offers the full functionality (all types, functors and constructions of a CGAL kernel model), but it is possible to restrict the interface. This can be useful for testing packages that require only a very small subset of the functionality offered by CGAL kernel models. If you want to do this, you have to define the macro *CGAL_CA_LIMITED_INTERFACE* (before the inclusion of *CGAL/Kernel_archetype.h*) to switch on the interface limitation. Now you have to tell the kernel archetype the types it has to provide for the specific package.

For every type you have to define a macro. The name of the macro is `CGAL_CA_NAME_OF_KERNEL_TYPE`, where `NAME_OF_KERNEL_TYPE` is the name of the kernel type (written in capitals) that has to be provided by the kernel archetype for a specific package. If, for example, a package only needs type definitions for `Point_2` and `Orientation_2`, you would define `CGAL_CA_LIMITED_INTERFACE`, `CGAL_CA_POINT_2` and `CGAL_CA_ORIENTATION_2`.

Is Model for the Concepts

Kernel page 35

```
#include <CGAL/Kernel_archetype.h>
```

Creation

<i>Kernel_archetype</i> <i>ka</i> ;	Default constructor.
-------------------------------------	----------------------

Types

We provide all type definitions that must be provided by a CGAL kernel model. See the CGAL kernel concept manual pages for details. Deprecated functionality is not supported. You can restrict the interface by defining macros (see the definition for details).

Operations

For each of the function objects of the kernel archetype, there is a member function that requires no arguments and returns an instance of that function object. The name of the member function is the uncapitalized name of the type returned with the suffix *_object* appended.

See Also

CGAL::Cartesian<FieldNumberType> page [41](#)
CGAL::Homogeneous<RingNumberType> page [48](#)
CGAL::Simple_cartesian<FieldNumberType> page [55](#)

CGAL::Kernel_traits<T>

Definition

The class *Kernel_traits*<*T*> provides access to the kernel model to which the argument type *T* belongs. (Provided *T* belongs to some kernel model.) The default implementation assumes there is a local type *T::Kernel* referring to the kernel model of *T*. If this type does not exist, a specialization of *Kernel_traits*<*T*> can be used to provide the desired information.

This class is, for example, useful in the following context. Assume you want to write a generic function that accepts two points *p* and *q* as argument and constructs the line segment between *p* and *q*. In order to specify the return type of this function, you need to know what is the segment type corresponding to the Point type representing *p* and *q*. Using *Kernel_traits*<*T*>, this can be done as follows.

```
template < class Point >
typename Kernel_traits<Point>::Kernel::Segment
construct_segment(Point p, Point q)
{ ... }
```

Types

<i>typedef T::R</i>	<i>Kernel</i> ;	If <i>T</i> is a type <i>K::Point_2</i> of some kernel model <i>K</i> , then <i>Kernel</i> is equal to <i>K</i> .
---------------------	-----------------	-------------------------------------------------------------------------------------------------------------------

CGAL::Simple_cartesian<FieldNumberType>

```
#include <CGAL/Simple_cartesian.h>
```

Definition

A model for a *Kernel* using Cartesian coordinates to represent the geometric objects. In order for *Simple_cartesian*<FieldNumberType> to model Euclidean geometry in E^2 and/or E^3 , for some mathematical field E (e.g., the rationals \mathbb{Q} or the reals \mathbb{R}), the template parameter FieldNumberType must model the mathematical field E . That is, the field operations on this number type must compute the mathematically correct results. If the number type provided as a model for FieldNumberType is only an approximation of a field (such as the built-in type *double*), then the geometry provided by the kernel is only an approximation of Euclidean geometry.

Is Model for the Concepts

Kernel page [35](#)

Types

```
typedef FieldNumberType
```

```
FT;
```

```
typedef FieldNumberType
```

```
RT;
```

Implementation

In contrast to *Cartesian*, no reference counting is used internally. This eases debugging, but may slow down algorithms that copy objects intensively.

See Also

CGAL::Cartesian<FieldNumberType> page [41](#)

CGAL::Homogeneous<RingNumberType> page [48](#)

CGAL::Simple_homogeneous<RingNumberType> page [56](#)

CGAL::Simple_homogeneous<RingNumberType>

```
#include <CGAL/Simple_homogeneous.h>
```

Definition

A model for a *Kernel* using homogeneous coordinates to represent the geometric objects. In order for *Simple_homogeneous<RingNumberType>* to model Euclidean geometry in E^2 and/or E^3 , for some mathematical ring E (e.g., the integers \mathbb{Z} or the rationals \mathbb{Q}), the template parameter *RingNumberType* must model the mathematical ring E . That is, the ring operations on this number type must compute the mathematically correct results. If the number type provided as a model for *RingNumberType* is only an approximation of a ring (such as the built-in type *double*), then the geometry provided by the kernel is only an approximation of Euclidean geometry.

Is Model for the Concepts

Kernel page [35](#)

Types

```
typedef Quotient<RingNumberType>   FT;
typedef RingNumberType              RT;
```

Implementation

In contrast to *Homogeneous*, no reference counting is used internally. This eases debugging, but may slow down algorithms that copy objects intensively, or slightly speed up others.

See Also

CGAL::Cartesian<FieldNumberType> page [41](#)
CGAL::Homogeneous<RingNumberType> page [48](#)
CGAL::Simple_cartesian<FieldNumberType> page [55](#)

2.9 Predefined Kernels

CGAL::Exact_predicates_exact_constructions_kernel

Definition

A typedef to a kernel which has the following properties:

- It uses Cartesian representation.
- It supports constructions of points from `double` Cartesian coordinates.
- It provides both exact geometric predicates and exact geometric constructions.

```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
```

Is Model for the Concepts

Kernel

See Also

CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt
CGAL::Exact_predicates_inexact_constructions_kernel
CGAL::Cartesian

CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt

Definition

A typedef to a kernel which has the following properties:

- It uses Cartesian representation.
- It supports constructions of points from `double` Cartesian coordinates.
- It provides both exact geometric predicates and exact geometric constructions.
- Its *FT* nested type supports the square root operation *sqrt()*.

Note that it requires CORE or LEDA installed.

```
#include <CGAL/Exact_predicates_exact_constructions_kernel_with_sqrt.h>
```

Is Model for the Concepts

Kernel

See Also

CGAL::Exact_predicates_exact_constructions_kernel
CGAL::Exact_predicates_inexact_constructions_kernel
CGAL::Cartesian

CGAL::Exact_predicates_inexact_constructions_kernel

Definition

A typedef to a kernel which has the following properties:

- It uses Cartesian representation.
- It supports constructions of points from `double` Cartesian coordinates.
- It provides exact geometric predicates, but inexact geometric constructions.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
```

Is Model for the Concepts

Kernel

See Also

CGAL::Exact_predicates_exact_constructions_kernel
CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt
CGAL::Cartesian

2.10 Kernel Objects

2.10.1 Two-dimensional Objects

CGAL::Aff_transformation_2<Kernel>

Definition

The class *Aff_transformation_2<Kernel>* represents two-dimensional affine transformations. The general form of an affine transformation is based on a homogeneous representation of points. Thereby all transformations can be realized by matrix multiplications.

Multiplying the transformation matrix by a scalar does not change the represented transformation. Therefore, any transformation represented by a matrix with rational entries can be represented by a transformation matrix with integer entries as well. (Multiply the matrix with the common denominator of the rational entries.) Hence, it is sufficient to use the number type *Kernel::RT* to represent the entries of the transformation matrix.

CGAL offers several specialized affine transformations. Different constructors are provided to create them. They are parameterized with a symbolic name to denote the transformation type, followed by additional parameters. The symbolic name tags solve ambiguities in the function overloading and they make the code more readable, i.e., what type of transformation is created.

Since two-dimensional points have three homogeneous coordinates, we have a 3×3 matrix $(m_{ij})_{i,j=0\dots 2}$.

If the homogeneous representations are normalized (the homogenizing coordinate is 1), then the upper left 2×2 matrix realizes linear transformations. In the matrix form of a translation, the translation vector $(v_0, v_1, 1)$ appears in the last column of the matrix. The entries m_{20} and m_{21} are always zero and therefore do not appear in the constructors.

Creation

```
Aff_transformation_2<Kernel> t( Identity_transformation);
```

introduces an identity transformation.

```
Aff_transformation_2<Kernel> t( const Translation, Vector_2<Kernel> v);
```

introduces a translation by a vector v .

```
Aff_transformation_2<Kernel> t( const Rotation,  
                                Direction_2<Kernel> d,  
                                Kernel::RT num,
```

Kernel::RT den = RT(1)

approximates the rotation over the angle indicated by direction d , such that the differences between the sines and cosines of the rotation given by d and the approximating rotation are at most num/den each.

Precondition: $num/den > 0$.

Aff_transformation_2<Kernel> t(const Rotation,
Kernel::RT sine_rho,
Kernel::RT cosine_rho,
Kernel::RT hw = RT(1))

introduces a rotation by the angle ρ .

Precondition: $sine_rho^2 + cosine_rho^2 == hw^2$.

Aff_transformation_2<Kernel> t(const Scaling, Kernel::RT s, Kernel::RT hw = RT(1));

introduces a scaling by a scale factor s/hw .

Aff_transformation_2<Kernel> t(Kernel::RT m00,
Kernel::RT m01,
Kernel::RT m02,
Kernel::RT m10,
Kernel::RT m11,
Kernel::RT m12,
Kernel::RT hw = RT(1))

introduces a general affine transformation in the $3 \times$

3 matrix form $\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ 0 & 0 & hw \end{pmatrix}$. The sub-matrix

$\frac{1}{hw} \begin{pmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \end{pmatrix}$ contains the scaling and rotation information,

the vector $\frac{1}{hw} \begin{pmatrix} m_{02} \\ m_{12} \end{pmatrix}$ contains the translational part of the transformation.

Aff_transformation_2<Kernel> t(Kernel::RT m00,
Kernel::RT m01,
Kernel::RT m10,
Kernel::RT m11,
Kernel::RT hw = RT(1))

introduces a general linear transformation

$\begin{pmatrix} m_{00} & m_{01} & 0 \\ m_{10} & m_{11} & 0 \\ 0 & 0 & hw \end{pmatrix}$, i.e. there is no translational

part.

Operations

The main thing to do with transformations is to apply them on geometric objects. Each class *Class_2<Kernel>* representing a geometric object has a member function:

Class_2<Kernel> *transform*(*Aff_transformation_2<Kernel>* *t*).

The transformation classes provide a member function *transform()* for points, vectors, directions, and lines:

<i>Point_2<Kernel></i>	<i>t.transform(Point_2<Kernel> p)</i>
<i>Vector_2<Kernel></i>	<i>t.transform(Vector_2<Kernel> p)</i>
<i>Direction_2<Kernel></i>	<i>t.transform(Direction_2<Kernel> p)</i>
<i>Line_2<Kernel></i>	<i>t.transform(Line_2<Kernel> p)</i>

CGAL provides function operators for these member functions:

<i>Point_2<Kernel></i>	<i>t.operator()(Point_2<Kernel> p)</i>
<i>Vector_2<Kernel></i>	<i>t.operator()(Vector_2<Kernel> p)</i>
<i>Direction_2<Kernel></i>	<i>t.operator()(Direction_2<Kernel> p)</i>
<i>Line_2<Kernel></i>	<i>t.operator()(Line_2<Kernel> p)</i>

Miscellaneous

Aff_transformation_2<Kernel>

<i>t.operator*(s)</i>	composes two affine transformations.
------------------------	--------------------------------------

Aff_transformation_2<Kernel>

<i>t.inverse()</i>	gives the inverse transformation.
--------------------	-----------------------------------

<i>bool</i>	<i>t.is_even()</i>	returns <i>true</i> , if the transformation is not reflecting, i.e. the determinant of the involved linear transformation is non-negative.
-------------	--------------------	--------------------------------------------------------------------------------------------------------------------------------------------

<i>bool</i>	<i>t.is_odd()</i>	returns <i>true</i> , if the transformation is reflecting.
-------------	-------------------	------------------------------------------------------------

The matrix entries of a matrix representation of a *Aff_transformation_2<Kernel>* can be accessed through the following member functions:

<i>Kernel::FT</i>	<i>t.cartesian(int i, int j)</i>	returns entry m_{ij} in a matrix representation in which m_{22} is 1.
<i>Kernel::FT</i>	<i>t.m(int i, int j)</i>	

<i>Kernel::RT</i>	<i>t.homogeneous(int i, int j)</i>	returns entry m_{ij} in some fixed matrix representation.
<i>Kernel::RT</i>	<i>t.hm(int i, int j)</i>	

For affine transformations no I/O operators are defined.

See Also

Identity_transformation, Rotation, Scaling, Translation
rational_rotation_approximation

Example

```
typedef Cartesian<double>      K;
typedef Aff_transformation_2<K> Transformation;
typedef Point_2<K>            Point;
typedef Vector_2<K>           Vector;
typedef Direction_2<K>        Direction;

Transformation rotate(ROTATION, sin(pi), cos(pi));
Transformation rational_rotate(ROTATION, Direction(1,1), 1, 100);
Transformation translate(TRANSLATION, Vector(-2, 0));
Transformation scale(SCALING, 3);

Point q(0, 1);
q = rational_rotate(q);

Point p(1, 1);

p = rotate(p);

p = translate(p);

p = scale(p);
```

The same would have been achieved with

```
Transformation transform = scale * (translate * rotate);
p = transform(Point(1.0, 1.0));
```

See Also

CGAL::Aff_transformation_3<Kernel> page 89
CGAL::Identity_transformation page 132
CGAL::Reflection page 132
CGAL::Rotation page 133
CGAL::Scaling page 133
CGAL::Translation page 134

CGAL::Bbox_2

#include <CGAL/Bbox_2.h>

Definition

An object b of the class *Bbox_2* is a bounding box in the two-dimensional Euclidean plane \mathbb{E}^2 . This class is not templated.

Creation

Bbox_2 b (*double* x_min , *double* y_min , *double* x_max , *double* y_max);

introduces a bounding box b with lower left corner at $(xmin, ymin)$ and with upper right corner at $(xmax, ymax)$.

Operations

double $b.xmin()$

double $b.ymin()$

double $b.xmax()$

double $b.ymax()$

Bbox_2 $b.operator+(c)$ returns a bounding box of b and c .

See Also

CGAL::Bbox_3 page [88](#)

CGAL::do_overlap page [166](#)

CGAL::Circle_2<Kernel>

Definition

An object of type *Circle_2<Kernel>* is a circle in the two-dimensional Euclidean plane \mathbb{E}^2 . The circle is oriented, i.e. its boundary has clockwise or counterclockwise orientation. The boundary splits \mathbb{E}^2 into a positive and a negative side, where the positive side is to the left of the boundary. The boundary also splits \mathbb{E}^2 into a bounded and an unbounded side. Note that the circle can be degenerated, i.e. the squared radius may be zero.

Creation

Circle_2<Kernel> *c*(*Point_2<Kernel>* *center*,
Kernel::FT *squared_radius*,
Orientation *ori* = *COUNTERCLOCKWISE*)

introduces a variable *c* of type *Circle_2<Kernel>*. It is initialized to the circle with center *center*, squared radius *squared_radius* and orientation *ori*.
Precondition: *ori* \neq *COLLINEAR*, and further, *squared_radius* \geq 0.

Circle_2<Kernel> *c*(*Point_2<Kernel>* *p*, *Point_2<Kernel>* *q*, *Point_2<Kernel>* *r*);

introduces a variable *c* of type *Circle_2<Kernel>*. It is initialized to the unique circle which passes through the points *p*, *q* and *r*. The orientation of the circle is the orientation of the point triple *p*, *q*, *r*.
Precondition: *p*, *q*, and *r* are not collinear.

Circle_2<Kernel> *c*(*Point_2<Kernel>* *p*, *Point_2<Kernel>* *q*, *Orientation* *ori* = *COUNTERCLOCKWISE*);

introduces a variable *c* of type *Circle_2<Kernel>*. It is initialized to the circle with diameter \overline{pq} and orientation *ori*.
Precondition: *ori* \neq *COLLINEAR*.

Circle_2<Kernel> *c*(*Point_2<Kernel>* *center*, *Orientation* *ori* = *COUNTERCLOCKWISE*);

introduces a variable *c* of type *Circle_2<Kernel>*. It is initialized to the circle with center *center*, squared radius zero and orientation *ori*.
Precondition: *ori* \neq *COLLINEAR*.
Postcondition: *c.is_degenerate()* = *true*.

Access Functions

Point_2<Kernel> *c.center()* returns the center of *c*.

<i>Kernel::FT</i>	<i>c.squared_radius()</i>	returns the squared radius of <i>c</i> .
<i>Orientation</i>	<i>c.orientation()</i>	returns the orientation of <i>c</i> .
<i>bool operator</i>	<i>c.==(circle2)</i>	returns <i>true</i> , iff <i>c</i> and <i>circle2</i> are equal, i.e. if they have the same center, same squared radius and same orientation.
<i>bool operator</i>	<i>c.!=(circle2)</i>	returns <i>true</i> , iff <i>c</i> and <i>circle2</i> are not equal.

Predicates

<i>bool</i>	<i>c.is_degenerate()</i>	returns <i>true</i> , iff <i>c</i> is degenerate, i.e. if <i>c</i> has squared radius zero.
<i>Oriented_side</i>	<i>c.oriented_side(Point_2<Kernel> p)</i>	returns either the constant <i>ON_ORIENTED_BOUNDARY</i> , <i>ON_POSITIVE_SIDE</i> , or <i>ON_NEGATIVE_SIDE</i> , iff <i>p</i> lies on the boundary, properly on the positive side, or properly on the negative side of <i>c</i> , resp.
<i>Bounded_side</i>	<i>c.bounded_side(Point_2<Kernel> p)</i>	returns <i>ON_BOUNDED_SIDE</i> , <i>ON_BOUNDARY</i> , or <i>ON_UNBOUNDED_SIDE</i> iff <i>p</i> lies properly inside, on the boundary, or properly outside of <i>c</i> , resp.
<i>bool</i>	<i>c.has_on_positive_side(Point_2<Kernel> p)</i>	
<i>bool</i>	<i>c.has_on_negative_side(Point_2<Kernel> p)</i>	
<i>bool</i>	<i>c.has_on_boundary(Point_2<Kernel> p)</i>	
<i>bool</i>	<i>c.has_on_bounded_side(Point_2<Kernel> p)</i>	
<i>bool</i>	<i>c.has_on_unbounded_side(Point_2<Kernel> p)</i>	

Miscellaneous

<i>Circle_2<Kernel></i>	<i>c.opposite()</i>	returns the circle with the same center and squared radius as <i>c</i> but with opposite orientation.
<i>Circle_2<Kernel></i>	<i>c.orthogonal_transform(Aff_transformation_2<Kernel> at)</i>	returns the circle obtained by applying <i>at</i> on <i>c</i> . <i>Precondition:</i> <i>at</i> is an orthogonal transformation.
<i>Bbox_2</i>	<i>c.bbox()</i>	returns a bounding box containing <i>c</i> .

See Also

Kernel::Circle_2 page [220](#)

CGAL::Direction_2<Kernel>

Definition

An object of the class *Direction_2<Kernel>* is a vector in the two-dimensional vector space \mathbb{R}^2 where we forget about its length. They can be viewed as unit vectors, although there is no normalization internally, since this is error prone. Directions are used whenever the length of a vector does not matter. They also characterize a set of parallel oriented lines that have the same orientations. For example, you can ask for the direction orthogonal to an oriented plane, or the direction of an oriented line. Further, they can be used to indicate angles. The slope of a direction is $dy()/dx()$.

Creation

Direction_2<Kernel> *d*(*Vector_2<Kernel>* *v*); introduces the direction *d* of vector *v*.

Direction_2<Kernel> *d*(*Line_2<Kernel>* *l*); introduces the direction *d* of line *l*.

Direction_2<Kernel> *d*(*Ray_2<Kernel>* *r*); introduces the direction *d* of ray *r*.

Direction_2<Kernel> *d*(*Segment_2<Kernel>* *s*); introduces the direction *d* of segment *s*.

Direction_2<Kernel> *d*(*Kernel::RT* *x*, *Kernel::RT* *y*);
introduces a direction *d* passing through the origin and the point with Cartesian coordinates (*x*,*y*).

Operations

Kernel::RT *d*.delta(*int* *i*) returns values, such that *d*== *Direction_2<Kernel>*(delta(0),delta(1)).
Precondition: : $0 \leq i \leq 1$.

Kernel::RT *d*.dx() returns delta(0).

Kernel::RT *d*.dy() returns delta(1).

There is a total order on directions. We compare the angles between the positive *x*-axis and the directions in counterclockwise order.

bool *d*.operator==(*e*)
bool *d*.operator!=(*e*)
bool *d*.operator<(*e*)
bool *d*.operator>(*e*)
bool *d*.operator<=(*e*)
bool *d*.operator>=(*e*)

Furthermore, we have

<i>bool</i>	<i>d.counterclockwise_in_between(d1, d2)</i>	returns true, iff <i>d</i> is not equal to <i>d1</i> , and while rotating counterclockwise starting at <i>d1</i> , <i>d</i> is reached strictly before <i>d2</i> is reached. Note that true is returned if <i>d1</i> == <i>d2</i> , unless also <i>d</i> == <i>d1</i> .
-------------	-----------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<i>Direction_2<Kernel></i>	<i>d.operator-()</i>	The direction opposite to <i>d</i> .
----------------------------------	----------------------	--------------------------------------

Miscellaneous

<i>Vector_2<Kernel></i>	<i>d.vector()</i>	returns a vector that has the same direction as <i>d</i> .
-------------------------------	-------------------	------------------------------------------------------------

<i>Direction_2<Kernel></i>	<i>d.transform(Aff_transformation_2<Kernel> t)</i>	returns the direction obtained by applying <i>t</i> on <i>d</i> .
----------------------------------	-----------------------------------------------------------	-------------------------------------------------------------------

See Also

Kernel::Direction_2 page [351](#)

CGAL::Iso_rectangle_2<Kernel>

Definition

An object s of the data type *Iso_rectangle_2<Kernel>* is a rectangle in the Euclidean plane \mathbb{E}^2 with sides parallel to the x and y axis of the coordinate system.

Although they are represented in a canonical form by only two vertices, namely the lower left and the upper right vertex, we provide functions for “accessing” the other vertices as well. The vertices are returned in counterclockwise order.

Iso-oriented rectangles and bounding boxes are quite similar. The difference however is that bounding boxes have always double coordinates, whereas the coordinate type of an iso-oriented rectangle is chosen by the user.

Creation

Iso_rectangle_2<Kernel> r (*Point_2<Kernel>* p , *Point_2<Kernel>* q);

introduces an iso-oriented rectangle r with diagonal opposite vertices p and q . Note that the object is brought in the canonical form.

Iso_rectangle_2<Kernel> r (*Point_2<Kernel>* $left$,
Point_2<Kernel> $right$,
Point_2<Kernel> $bottom$,
Point_2<Kernel> top)

introduces an iso-oriented rectangle r whose minimal x coordinate is the one of $left$, the maximal x coordinate is the one of $right$, the minimal y coordinate is the one of $bottom$, the maximal y coordinate is the one of top .

Iso_rectangle_2<Kernel> r (*Kernel::RT* min_hx ,
Kernel::RT min_hy ,
Kernel::RT max_hx ,
Kernel::RT max_hy ,
Kernel::RT $hw = RT(1)$)

introduces an iso-oriented rectangle r with diagonal opposite vertices $(min_hx/hw, min_hy/hw)$ and $(max_hx/hw, max_hy/hw)$.
Precondition: $hw \neq 0$.

Operations

bool $r.operator==(r2)$ Test for equality: two iso-oriented rectangles are equal, iff their lower left and their upper right vertices are equal.

<i>bool</i>	<i>r.operator!=(r2)</i>	Test for inequality.
<i>Point_2<Kernel></i>	<i>r.vertex(int i)</i>	returns the <i>i</i> 'th vertex modulo 4 of <i>r</i> in counterclockwise order, starting with the lower left vertex.
<i>Point_2<Kernel></i>	<i>r.operator[](int i)</i>	returns <i>vertex(i)</i> .
<i>Point_2<Kernel></i>	<i>r.min()</i>	returns the lower left vertex of <i>r</i> ($= vertex(0)$).
<i>Point_2<Kernel></i>	<i>r.max()</i>	returns the upper right vertex of <i>r</i> ($= vertex(2)$).
<i>Kernel::FT</i>	<i>r.xmin()</i>	returns the <i>x</i> coordinate of lower left vertex of <i>r</i> .
<i>Kernel::FT</i>	<i>r.ymin()</i>	returns the <i>y</i> coordinate of lower left vertex of <i>r</i> .
<i>Kernel::FT</i>	<i>r.xmax()</i>	returns the <i>x</i> coordinate of upper right vertex of <i>r</i> .
<i>Kernel::FT</i>	<i>r.ymax()</i>	returns the <i>y</i> coordinate of upper right vertex of <i>r</i> .
<i>Kernel::FT</i>	<i>r.min_coord(int i)</i>	returns the <i>i</i> 'th Cartesian coordinate of the lower left vertex of <i>r</i> . <i>Precondition:</i> $0 \leq i \leq 1$.
<i>Kernel::FT</i>	<i>r.max_coord(int i)</i>	returns the <i>i</i> 'th Cartesian coordinate of the upper right vertex of <i>r</i> . <i>Precondition:</i> $0 \leq i \leq 1$.

Predicates

<i>bool</i>	<i>r.is_degenerate()</i>	<i>r</i> is degenerate, if all vertices are collinear.
<i>Bounded_side</i>	<i>r.bounded_side(Point_2<Kernel> p)</i>	returns either <i>ON_UNBOUNDED_SIDE</i> , <i>ON_BOUNDED_SIDE</i> , or the constant <i>ON_BOUNDARY</i> , depending on where point <i>p</i> is.
<i>bool</i>	<i>r.has_on_boundary(Point_2<Kernel> p)</i>	
<i>bool</i>	<i>r.has_on_bounded_side(Point_2<Kernel> p)</i>	
<i>bool</i>	<i>r.has_on_unbounded_side(Point_2<Kernel> p)</i>	

Miscellaneous

<i>Kernel::FT</i>	<i>r.area()</i>	returns the area of <i>r</i> .
<i>Bbox</i>	<i>r.bbox()</i>	returns a bounding box containing <i>r</i> .

Iso_rectangle_2<Kernel>

r.transform(Aff_transformation_2<Kernel> *t*)

returns the iso-oriented rectangle obtained by applying *t* on the lower left and the upper right corner of *r*.

Precondition: The angle at a rotation must be a multiple of $\pi/2$, otherwise the resulting rectangle does not have the same side length. Note that rotating about an arbitrary angle can even result in a degenerate iso-oriented rectangle.

See Also

Kernel::IsoRectangle_2.....page [382](#)

CGAL::Line_2<Kernel>

Definition

An object l of the data type $Line_2<Kernel>$ is a directed straight line in the two-dimensional Euclidean plane \mathbb{E}^2 . It is defined by the set of points with Cartesian coordinates (x, y) that satisfy the equation

$$l : ax + by + c = 0.$$

The line splits \mathbb{E}^2 in a *positive* and a *negative* side. A point p with Cartesian coordinates (px, py) is on the positive side of l , iff $apx + bpy + c > 0$, it is on the negative side of l , iff $apx + bpy + c < 0$. The positive side is to the left of l .

Creation

$Line_2<Kernel> \ l(Kernel::RT \ a, Kernel::RT \ b, Kernel::RT \ c);$

introduces a line l with the line equation in Cartesian coordinates $ax + by + c = 0$.

$Line_2<Kernel> \ l(Point_2<Kernel> \ p, Point_2<Kernel> \ q);$

introduces a line l passing through the points p and q . Line l is directed from p to q .

$Line_2<Kernel> \ l(Point_2<Kernel> \ p, Direction_2<Kernel> \ d);$

introduces a line l passing through point p with direction d .

$Line_2<Kernel> \ l(Point_2<Kernel> \ p, Vector_2<Kernel> \ v);$

introduces a line l passing through point p and oriented by v .

$Line_2<Kernel> \ l(Segment_2<Kernel> \ s);$

introduces a line l supporting the segment s , oriented from source to target.

$Line_2<Kernel> \ l(Ray_2<Kernel> \ r);$

introduces a line l supporting the ray r , with same orientation.

Operations

$bool \quad \quad \quad l.operator==(h)$

Test for equality: two lines are equal, iff they have a non empty intersection and the same direction.

<i>bool</i>	<i>l.operator!=(h)</i>	Test for inequality.
<i>Kernel::RT</i>	<i>l.a()</i>	returns the first coefficient of <i>l</i> .
<i>Kernel::RT</i>	<i>l.b()</i>	returns the second coefficient of <i>l</i> .
<i>Kernel::RT</i>	<i>l.c()</i>	returns the third coefficient of <i>l</i> .
<i>Point_2<Kernel></i>	<i>l.point(int i)</i>	returns an arbitrary point on <i>l</i> . It holds <i>point(i) == point(j)</i> , iff <i>i==j</i> . Furthermore, <i>l</i> is directed from <i>point(i)</i> to <i>point(j)</i> , for all <i>i < j</i> .
<i>Point_2<Kernel></i>	<i>l.projection(Point_2<Kernel> p)</i>	returns the orthogonal projection of <i>p</i> onto <i>l</i> .
<i>Kernel::FT</i>	<i>l.x_at_y(Kernel::FT y)</i>	returns the <i>x</i> -coordinate of the point at <i>l</i> with given <i>y</i> -coordinate. <i>Precondition: l</i> is not horizontal.
<i>Kernel::FT</i>	<i>l.y_at_x(Kernel::FT x)</i>	returns the <i>y</i> -coordinate of the point at <i>l</i> with given <i>x</i> -coordinate. <i>Precondition: l</i> is not vertical.

Predicates

<i>bool</i>	<i>l.is_degenerate()</i>	line <i>l</i> is degenerate, if the coefficients <i>a</i> and <i>b</i> of the line equation are zero.
<i>bool</i>	<i>l.is_horizontal()</i>	
<i>bool</i>	<i>l.is_vertical()</i>	
<i>Oriented_side</i>	<i>l.oriented_side(Point_2<Kernel> p)</i>	returns <i>ON_ORIENTED_BOUNDARY</i> , <i>ON_NEGATIVE_SIDE</i> , or the constant <i>ON_POSITIVE_SIDE</i> , depending on the position of <i>p</i> relative to the oriented line <i>l</i> .

For convenience we provide the following boolean functions:

<i>bool</i>	<i>l.has_on(Point_2<Kernel> p)</i>
<i>bool</i>	<i>l.has_on_positive_side(Point_2<Kernel> p)</i>
<i>bool</i>	<i>l.has_on_negative_side(Point_2<Kernel> p)</i>

Miscellaneous

<i>Vector_2<Kernel></i>	<i>l.to_vector()</i>	returns a vector having the direction of <i>l</i> .
<i>Direction_2<Kernel></i>	<i>l.direction()</i>	returns the direction of <i>l</i> .

<i>Line_2</i> < <i>Kernel</i> >	<i>l.opposite()</i>	returns the line with opposite direction.
<i>Line_2</i> < <i>Kernel</i> >	<i>l.perpendicular(Point_2</i> < <i>Kernel</i> > <i>p</i>)	returns the line perpendicular to <i>l</i> and passing through <i>p</i> , where the direction is the direction of <i>l</i> rotated counterclockwise by 90 degrees.
<i>Line_2</i> < <i>Kernel</i> >	<i>l.transform(Aff_transformation_2</i> < <i>Kernel</i> > <i>t</i>)	returns the line obtained by applying <i>t</i> on a point on <i>l</i> and the direction of <i>l</i> .

Example

Let us first define two Cartesian two-dimensional points in the Euclidean plane \mathbb{E}^2 . Their dimension and the fact that they are Cartesian is expressed by the suffix *_2* and the representation type *Cartesian*.

```
Point_2< Cartesian<double> >  p(1.0,1.0), q(4.0,7.0);
```

To define a line *l* we write:

```
Line_2< Cartesian<double> >  l(p,q);
```

See Also

Kernel::Line_2 page [399](#)

CGAL::Point_2<Kernel>

Definition

An object of the class *Point_2<Kernel>* is a point in the two-dimensional Euclidean plane \mathbb{E}^2 .

Remember that *Kernel::RT* and *Kernel::FT* denote a *RingNumberType* and a *FieldNumberType*, respectively. For the kernel model *Cartesian<T>*, the two types are the same. For the kernel model *Homogeneous<T>*, *Kernel::RT* is equal to *T*, and *Kernel::FT* is equal to *Quotient<T>*.

Types

Point_2<Kernel>::Cartesian_const_iterator

An iterator for enumerating the Cartesian coordinates of a point.

Creation

Point_2<Kernel> p(Origin ORIGIN);

introduces a variable *p* with Cartesian coordinates (0,0).

Point_2<Kernel> p(Kernel::RT hx, Kernel::RT hy, Kernel::RT hw = RT(1));

introduces a point *p* initialized to (*hx/hw*, *hy/hw*).
Precondition: *hw* \neq *Kernel::RT*(0)

Operations

bool p.operator==(q)

Test for equality. Two points are equal, iff their *x* and *y* coordinates are equal. The point can be compared with *ORIGIN*.

bool p.operator!=(q)

Test for inequality. The point can be compared with *ORIGIN*.

There are two sets of coordinate access functions, namely to the homogeneous and to the Cartesian coordinates. They can be used independently from the chosen kernel model.

Kernel::RT p.hx()

returns the homogeneous *x* coordinate.

<i>Kernel::RT</i>	<i>p.hy()</i>	
		returns the homogeneous y coordinate.
<i>Kernel::RT</i>	<i>p.hw()</i>	

returns the homogenizing coordinate.

Note that you do not lose information with the homogeneous representation, because the `FieldNumberType` is a quotient.

<i>Kernel::FT</i>	<i>p.x()</i>	returns the Cartesian x coordinate, that is hx/hw .
<i>Kernel::FT</i>	<i>p.y()</i>	returns the Cartesian y coordinate, that is hy/hw .

The following operations are for convenience and for compatibility with higher dimensional points. Again they come in a Cartesian and in a homogeneous flavor.

<i>Kernel::RT</i>	<i>p.homogeneous(int i)</i>	
		returns the i 'th homogeneous coordinate of p , starting with 0. <i>Precondition:</i> $0 \leq i \leq 2$.

<i>Kernel::FT</i>	<i>p.cartesian(int i)</i>	
		returns the i 'th Cartesian coordinate of p , starting with 0. <i>Precondition:</i> $0 \leq i \leq 1$.

<i>Kernel::FT</i>	<i>p.operator[](int i)</i>	
		returns <i>cartesian</i> (i). <i>Precondition:</i> $0 \leq i \leq 1$.

<i>Cartesian_const_iterator</i>	<i>p.cartesian_begin()</i>	
		returns an iterator to the Cartesian coordinates of p , starting with the 0th coordinate.

<i>Cartesian_const_iterator</i>	<i>p.cartesian_end()</i>	
		returns an off the end iterator to the Cartesian coordinates of p .

<i>int</i>	<i>p.dimension()</i>	
		returns the dimension (the constant 2).

<i>Bbox_2</i>	<i>p.bbox()</i>	
		returns a bounding box containing p . Note that bounding boxes are not parameterized with whatsoever.

Point_2<Kernel> *p.transform(Aff_transformation_2<Kernel> t)*
 returns the point obtained by applying *t* on *p*.

Operators

The following operations can be applied on points:

bool *operator<(p, q)*
 returns true iff *p* is lexicographically smaller than *q*, i.e. either if *p.x()* < *q.x()* or if *p.x()* == *q.x()* and *p.y()* < *q.y()*.

bool *operator>(p, q)*
 returns true iff *p* is lexicographically greater than *q*.

bool *operator<=(p, q)*
 returns true iff *p* is lexicographically smaller or equal to *q*.

bool *operator>=(p, q)*
 returns true iff *p* is lexicographically greater or equal to *q*.

Vector_2<Kernel> *operator-(p, q)*
 returns the difference vector between *q* and *p*. You can substitute *ORIGIN* for either *p* or *q*, but not for both.

Point_2<Kernel> *operator+(p, Vector_2<Kernel> v)*
 returns the point obtained by translating *p* by the vector *v*.

Point_2<Kernel> *operator-(p, Vector_2<Kernel> v)*
 returns the point obtained by translating *p* by the vector -*v*.

Example

The following declaration creates two points with Cartesian double coordinates.

```
Point_2< Cartesian<double> > p, q(1.0, 2.0);
```

The variable `p` is uninitialized and should first be used on the left hand side of an assignment.

```
p = q;  
  
std::cout << p.x() << " " << p.y() << std::endl;
```

See Also

Kernel::Point_2.....page [408](#)

CGAL::Ray_2<Kernel>

Definition

An object r of the data type $\text{Ray}_2<\text{Kernel}>$ is a directed straight ray in the two-dimensional Euclidean plane \mathbb{E}^2 . It starts in a point called the *source* of r and goes to infinity.

Creation

$\text{Ray}_2<\text{Kernel}> \ r(\text{Point}_2<\text{Kernel}> \ p, \text{Point}_2<\text{Kernel}> \ q);$

introduces a ray r with source p and passing through point q .

$\text{Ray}_2<\text{Kernel}> \ r(\text{Point}_2<\text{Kernel}> \ p, \text{Direction}_2<\text{Kernel}> \ d);$

introduces a ray r starting at source p with direction d .

$\text{Ray}_2<\text{Kernel}> \ r(\text{Point}_2<\text{Kernel}> \ p, \text{Vector}_2<\text{Kernel}> \ v);$

introduces a ray r starting at source p with the direction of v .

$\text{Ray}_2<\text{Kernel}> \ r(\text{Point}_2<\text{Kernel}> \ p, \text{Line}_2<\text{Kernel}> \ l);$

introduces a ray r starting at source p with the same direction as l .

Operations

<i>bool</i>	$r.operator==(h)$	Test for equality: two rays are equal, iff they have the same source and the same direction.
<i>bool</i>	$r.operator!=(h)$	Test for inequality.
$\text{Point}_2<\text{Kernel}>$	$r.source()$	returns the source of r .
$\text{Point}_2<\text{Kernel}>$	$r.point(\text{int } i)$	returns a point on r . $point(0)$ is the source, $point(i)$, with $i > 0$, is different from the source. <i>Precondition:</i> $i \geq 0$.
$\text{Direction}_2<\text{Kernel}>$	$r.direction()$	returns the direction of r .
$\text{Vector}_2<\text{Kernel}>$	$r.to_vector()$	returns a vector giving the direction of r .

<i>Line_2<Kernel></i>	<i>r.supporting_line()</i>	returns the line supporting <i>r</i> which has the same direction.
<i>Ray_2<Kernel></i>	<i>r.opposite()</i>	returns the ray with the same source and the opposite direction.

Predicates

<i>bool</i>	<i>r.is_degenerate()</i>	ray <i>r</i> is degenerate, if the source and the second defining point fall together (that is if the direction is degenerate).
<i>bool</i> <i>bool</i>	<i>r.is_horizontal()</i> <i>r.is_vertical()</i>	
<i>bool</i>	<i>r.has_on(Point_2<Kernel> p)</i>	A point is on <i>r</i> , iff it is equal to the source of <i>r</i> , or if it is in the interior of <i>r</i> .
<i>bool</i>	<i>r.collinear_has_on(Point_2<Kernel> p)</i>	checks if point <i>p</i> is on <i>r</i> . This function is faster than function <i>has_on()</i> if the precondition checking is disabled. <i>Precondition:</i> <i>p</i> is on the supporting line of <i>r</i> .

Miscellaneous

<i>Ray_2<Kernel></i>	<i>r.transform(Aff_transformation_2<Kernel> t)</i>	returns the ray obtained by applying <i>t</i> on the source and on the direction of <i>r</i> .
----------------------------	-----------------------------------------------------------	------------------------------------------------------------------------------------------------

See Also

Kernel::Ray_2 page [412](#)

CGAL::Segment_2<Kernel>

Definition

An object s of the data type $\text{Segment}_2<\text{Kernel}>$ is a directed straight line segment in the two-dimensional Euclidean plane \mathbb{E}^2 , i.e. a straight line segment $[p, q]$ connecting two points $p, q \in \mathbb{R}^2$. The segment is topologically closed, i.e. the end points belong to it. Point p is called the *source* and q is called the *target* of s . The length of s is the Euclidean distance between p and q . Note that there is only a function to compute the square of the length, because otherwise we had to perform a square root operation which is not defined for all number types, which is expensive, and may not be exact.

Creation

$\text{Segment}_2<\text{Kernel}> \ s(\text{Point}_2<\text{Kernel}> \ p, \text{Point}_2<\text{Kernel}> \ q);$

introduces a segment s with source p and target q . The segment is directed from the source towards the target.

Operations

$bool$	$s.operator==(q)$	Test for equality: Two segments are equal, iff their sources and targets are equal.
$bool$	$s.operator!=(q)$	Test for inequality.
$\text{Point}_2<\text{Kernel}>$	$s.source()$	returns the source of s .
$\text{Point}_2<\text{Kernel}>$	$s.target()$	returns the target of s .
$\text{Point}_2<\text{Kernel}>$	$s.min()$	returns the point of s with lexicographically smallest coordinate.
$\text{Point}_2<\text{Kernel}>$	$s.max()$	returns the point of s with lexicographically largest coordinate.
$\text{Point}_2<\text{Kernel}>$	$s.vertex(\text{int } i)$	returns source or target of s : $vertex(0)$ returns the source of s , $vertex(1)$ returns the target of s . The parameter i is taken modulo 2, which gives easy access to the other vertex.
$\text{Point}_2<\text{Kernel}>$	$s.point(\text{int } i)$	returns $vertex(i)$.
$\text{Point}_2<\text{Kernel}>$	$s.operator[] (\text{int } i)$	returns $vertex(i)$.
$\text{Kernel}::FT$	$s.squared_length()$	returns the squared length of s .
$\text{Direction}_2<\text{Kernel}>$	$s.direction()$	returns the direction from source to target of s .

<i>Vector_2<Kernel></i>	<i>s.to_vector()</i>	returns the vector <i>s.target()</i> - <i>s.source()</i> .
<i>Segment_2<Kernel></i>	<i>s.opposite()</i>	returns a segment with source and target point interchanged.
<i>Line_2<Kernel></i>	<i>s.supporting_line()</i>	returns the line <i>l</i> passing through <i>s</i> . Line <i>l</i> has the same orientation as segment <i>s</i> .

Predicates

<i>bool</i>	<i>s.is_degenerate()</i>	segment <i>s</i> is degenerate, if source and target are equal.
<i>bool</i>	<i>s.is_horizontal()</i>	
<i>bool</i>	<i>s.is_vertical()</i>	
<i>bool</i>	<i>s.has_on(Point_2<Kernel> p)</i>	A point is on <i>s</i> , iff it is equal to the source or target of <i>s</i> , or if it is in the interior of <i>s</i> .
<i>bool</i>	<i>s.collinear_has_on(Point_2<Kernel> p)</i>	checks if point <i>p</i> is on segment <i>s</i> . This function is faster than function <i>has_on()</i> . <i>Precondition:</i> <i>p</i> is on the supporting line of <i>s</i> .

Miscellaneous

<i>Bbox_2</i>	<i>s.bbox()</i>	returns a bounding box containing <i>s</i> .
<i>Segment_2<Kernel></i>	<i>s.transform(Aff_transformation_2<Kernel> t)</i>	returns the segment obtained by applying <i>t</i> on the source and the target of <i>s</i> .

See Also

Kernel::Segment_2.....page [414](#)

CGAL::Triangle_2<Kernel>

Definition

An object t of the class *Triangle_2<Kernel>* is a triangle in the two-dimensional Euclidean plane \mathbb{E}^2 . Triangle t is oriented, i.e., its boundary has clockwise or counterclockwise orientation. We call the side to the left of the boundary the positive side and the side to the right of the boundary the negative side.

The boundary of a triangle splits the plane in two open regions, a bounded one and an unbounded one.

Creation

Triangle_2<Kernel> t (*Point_2<Kernel>* p , *Point_2<Kernel>* q , *Point_2<Kernel>* r);

introduces a triangle t with vertices p , q and r .

Operations

bool $t.operator==(t2)$ Test for equality: two triangles are equal, iff there exists a cyclic permutation of the vertices of $t2$, such that they are equal to the vertices of t .

bool $t.operator!=(t2)$ Test for inequality.

Point_2<Kernel> $t.vertex(int i)$ returns the i 'th vertex modulo 3 of t .

Point_2<Kernel> $t.operator[] (int i)$ returns $vertex(i)$.

Predicates

bool $t.is_degenerate()$ triangle t is degenerate, if the vertices are collinear.

Orientation $t.orientation()$ returns the orientation of t .

Oriented_side $t.oriented_side(Point_2<Kernel> p)$
 returns *ON_ORIENTED_BOUNDARY*, or *POSITIVE_SIDE*, or the constant *ON_NEGATIVE_SIDE*, determined by the position of point p .
Precondition: t is not degenerate.

Bounded_side $t.bounded_side(Point_2<Kernel> p)$
 returns the constant *ON_BOUNDARY*, *ON_BOUNDED_SIDE*, or else *ON_UNBOUNDED_SIDE*, depending on where point p is.
Precondition: t is not degenerate.

For convenience we provide the following boolean functions:

<i>bool</i>	<i>t.has_on_positive_side(Point_2<Kernel> p)</i>
<i>bool</i>	<i>t.has_on_negative_side(Point_2<Kernel> p)</i>
<i>bool</i>	<i>t.has_on_boundary(Point_2<Kernel> p)</i>
<i>bool</i>	<i>t.has_on_bounded_side(Point_2<Kernel> p)</i>
<i>bool</i>	<i>t.has_on_unbounded_side(Point_2<Kernel> p)</i>

Precondition: *t* is not degenerate.

Miscellaneous

<i>Triangle_2<Kernel></i>	<i>t.opposite()</i>	returns a triangle where the boundary is oriented the other way round (this flips the positive and the negative side, but not the bounded and unbounded side).
<i>Kernel::FT</i>	<i>t.area()</i>	returns the signed area of <i>t</i> .
<i>Bbox_2</i>	<i>t.bbox()</i>	returns a bounding box containing <i>t</i> .
<i>Triangle_2<Kernel></i>	<i>t.transform(Aff_transformation_2<Kernel> at)</i>	returns the triangle obtained by applying <i>at</i> on the three vertices of <i>t</i> .

See Also

Kernel::Triangle_2 page [422](#)

CGAL::Vector_2<Kernel>

Definition

An object of the class *Vector_2<Kernel>* is a vector in the two-dimensional vector space \mathbb{R}^2 . Geometrically spoken, a vector is the difference of two points p_2, p_1 and denotes the direction and the distance from p_1 to p_2 .

CGAL defines a symbolic constant *NULL_VECTOR*. We will explicitly state where you can pass this constant as an argument instead of a vector initialized with zeros.

Creation

Vector_2<Kernel> *v*(*Point_2<Kernel>* *a*, *Point_2<Kernel>* *b*);

introduces the vector $b - a$.

Vector_2<Kernel> *v*(*Segment_2<Kernel>* *s*);

introduces the vector $s.target() - s.source()$.

Vector_2<Kernel> *v*(*Ray_2<Kernel>* *r*);

introduces the vector having the same direction as *r*.

Vector_2<Kernel> *v*(*Line_2<Kernel>* *l*);

introduces the vector having the same direction as *l*.

Vector_2<Kernel> *v*(*Null_vector* *NULL_VECTOR*);

introduces a null vector *v*.

Vector_2<Kernel> *v*(*Kernel::RT* *hx*, *Kernel::RT* *hy*, *Kernel::RT* *hw* = *RT*(1));

introduces a vector *v* initialized to $(hx/hw, hy/hw)$.
Precondition: $hw \neq 0$

Operations

<i>bool</i>	<i>v.operator==(w)</i>	Test for equality: two vectors are equal, iff their <i>x</i> and <i>y</i> coordinates are equal. You can compare a vector with the <i>NULL_VECTOR</i> .
-------------	-------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------

<i>bool</i>	<i>v.operator!=(w)</i>	Test for inequality. You can compare a vector with the <i>NULL_VECTOR</i> .
-------------	-------------------------	-----------------------------------------------------------------------------

There are two sets of coordinate access functions, namely to the homogeneous and to the Cartesian coordinates. They can be used independently from the chosen kernel model.

<i>Kernel::RT</i>	<i>v.hx()</i>	returns the homogeneous <i>x</i> coordinate.
<i>Kernel::RT</i>	<i>v.hy()</i>	returns the homogeneous <i>y</i> coordinate.

<i>Kernel::RT</i>	<i>v.hw()</i>	returns the homogenizing coordinate.
-------------------	---------------	--------------------------------------

Note that you do not lose information with the homogeneous representation, because the `FieldNumberType` is a quotient.

<i>Kernel::FT</i>	<i>v.x()</i>	returns the x -coordinate of v , that is hx/hw .
<i>Kernel::FT</i>	<i>v.y()</i>	returns the y -coordinate of v , that is hy/hw .

The following operations are for convenience and for compatibility with higher dimensional vectors. Again they come in a Cartesian and homogeneous flavor.

<i>Kernel::RT</i>	<i>v.homogeneous(int i)</i>	returns the i 'th homogeneous coordinate of v , starting with 0.
-------------------	-------------------------------	----------------------------------------------------------------------

Precondition: $0 \leq i \leq 2$.

<i>Kernel::FT</i>	<i>v.cartesian(int i)</i>	returns the i 'th Cartesian coordinate of v , starting at 0.
-------------------	-----------------------------	------------------------------------------------------------------

Precondition: $0 \leq i \leq 1$.

<i>Kernel::FT</i>	<i>v.operator[] (int i)</i>	returns <i>cartesian(i)</i> .
-------------------	-------------------------------	-------------------------------

Precondition: $0 \leq i \leq 1$.

<i>int</i>	<i>v.dimension()</i>	returns the dimension (the constant 2).
------------	----------------------	-----------------------------------------

<i>Direction_2<Kernel></i>	<i>v.direction()</i>	returns the direction which passes through v .
----------------------------------	----------------------	--------------------------------------------------

<i>Vector_2<Kernel></i>	<i>v.transform(Aff_transformation_2<Kernel> t)</i>	
-------------------------------	------------------------------------------------------------	--

returns the vector obtained by applying t on v .

<i>Vector_2<Kernel></i>	<i>v.perpendicular(Orientation o)</i>	
-------------------------------	-----------------------------------------	--

returns the vector perpendicular to v in clockwise or counterclockwise orientation.

Operators

The following operations can be applied to vectors:

<i>Vector_2<Kernel></i>	<i>v.operator+(w)</i>	Addition.
-------------------------------	-------------------------	-----------

<i>Vector_2<Kernel></i>	<i>v.operator-(w)</i>	Subtraction.
-------------------------------	-------------------------	--------------

<i>Vector_2<Kernel></i>	<i>v.operator-()</i>	returns the opposite vector.
-------------------------------	----------------------	------------------------------

<i>Kernel::FT</i>	<i>v.operator*(w)</i>	returns the scalar product (= inner product) of the two vectors.
-------------------	-------------------------	------------------------------------------------------------------

<i>Vector_2<Kernel></i>	<i>operator*(v, Kernel::RT s)</i>	
-------------------------------	-------------------------------------	--

Multiplication with a scalar from the right.

<i>Vector_2<Kernel></i>	<i>operator*(v, Kernel::FT s)</i>	
		Multiplication with a scalar from the right.

<i>Vector_2<Kernel></i>	<i>operator*(Kernel::RT s, v)</i>	
		Multiplication with a scalar from the left.

<i>Vector_2<Kernel></i>	<i>operator*(Kernel::FT s, v)</i>	
		Multiplication with a scalar from the left.

<i>Vector_2<Kernel></i>	<i>v.operator/(Kernel::RT s)</i>	
		Division by a scalar.

See Also

Kernel::Vector_2 page [424](#)

2.10.2 Three-dimensional Objects

CGAL::Bbox_3

```
#include <CGAL/Bbox_3.h>
```

Definition

An object b of the class *Bbox_3* is a bounding box in the three-dimensional Euclidean space \mathbb{E}^3 .

Creation

```
Bbox_3 b( double x_min, double y_min, double z_min, double x_max, double y_max, double z_max);
```

introduces a bounding box b with lexicographically smallest corner point at $(x_{min}, y_{min}, z_{min})$ lexicographically largest corner point at $(x_{max}, y_{max}, z_{max})$.

Operations

<i>double</i>	<i>b.xmin()</i>
<i>double</i>	<i>b.ymin()</i>
<i>double</i>	<i>b.zmin()</i>
<i>double</i>	<i>b.xmax()</i>
<i>double</i>	<i>b.ymax()</i>
<i>double</i>	<i>b.zmax()</i>

<i>Bbox_3</i>	<i>b.operator+(c)</i>	returns a bounding box of b and c .
---------------	-------------------------	-----------------------------------------

See Also

CGAL::Bbox_2 page [64](#)
CGAL::do_overlap page [166](#)

CGAL::Aff_transformation_3<Kernel>

Definition

The class *Aff_transformation_3<Kernel>* represents three-dimensional affine transformations. The general form of an affine transformation is based on a homogeneous representation of points. Thereby all transformations can be realized by matrix multiplication.

Multiplying the transformation matrix by a scalar does not change the represented transformation. Therefore, any transformation represented by a matrix with rational entries can be represented by a transformation matrix with integer entries as well. (Multiply the matrix with the common denominator of the rational entries.) Hence, it is sufficient to use the number type *Kernel::RT* to represent the entries of the transformation matrix.

CGAL offers several specialized affine transformations. Different constructors are provided to create them. They are parameterized with a symbolic name to denote the transformation type, followed by additional parameters. The symbolic name tags solve ambiguities in the function overloading and they make the code more readable, i.e., what type of transformation is created.

In three-dimensional space we have a 4×4 matrix $(m_{ij})_{i,j=0\dots 3}$. Entries m_{30} , m_{31} , and m_{32} are always zero and therefore do not appear in the constructors.

Creation

```
Aff_transformation_3<Kernel> t( Identity_transformation);
```

introduces an identity transformation.

```
Aff_transformation_3<Kernel> t( const Translation, Vector_3<Kernel> v);
```

introduces a translation by a vector v .

```
Aff_transformation_3<Kernel> t( const Scaling, Kernel::RT s, Kernel::RT hw = RT(1));
```

introduces a scaling by a scale factor s/hw .

```
Aff_transformation_3<Kernel> t( Kernel::RT m00,
                               Kernel::RT m01,
                               Kernel::RT m02,
                               Kernel::RT m03,
                               Kernel::RT m10,
                               Kernel::RT m11,
                               Kernel::RT m12,
                               Kernel::RT m13,
                               Kernel::RT m20,
                               Kernel::RT m21,
                               Kernel::RT m22,
                               Kernel::RT m23,
```

$$\text{Kernel}::RT\ hw = RT(1)$$

introduces a general affine transformation of the matrix form $\begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ 0 & 0 & 0 & hw \end{pmatrix}$. The part $\frac{1}{hw}$ $\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix}$ defines the scaling and rotational part of the transformation, while the vector $\frac{1}{hw} \begin{pmatrix} m_{03} \\ m_{13} \\ m_{23} \end{pmatrix}$ contains the translational part.

```
Aff_transformation_3<Kernel> t( Kernel::RT m00,
                                Kernel::RT m01,
                                Kernel::RT m02,
                                Kernel::RT m10,
                                Kernel::RT m11,
                                Kernel::RT m12,
                                Kernel::RT m20,
                                Kernel::RT m21,
                                Kernel::RT m22,
                                Kernel::RT hw = RT(1))
```

introduces a general linear transformation of the matrix form $\begin{pmatrix} m_{00} & m_{01} & m_{02} & 0 \\ m_{10} & m_{11} & m_{12} & 0 \\ m_{20} & m_{21} & m_{22} & 0 \\ 0 & 0 & 0 & hw \end{pmatrix}$, i.e. an affine transformation without translational part.

Operations

Each class *Class_3<Kernel>* representing a geometric object in 3D has a member function:

```
Class_3<Kernel> transform(Aff_transformation_3<Kernel> t).
```

The transformation classes provide a member function *transform()* for points, vectors, directions, and planes:

```
Point_3<Kernel>          t.transform( Point_3<Kernel> p)
Vector_3<Kernel>        t.transform( Vector_3<Kernel> p)
Direction_3<Kernel>     t.transform( Direction_3<Kernel> p)
Plane_3<Kernel>         t.transform( Plane_3<Kernel> p)
```

CGAL provides four function operators for these member functions:

```
Point_3<Kernel>          t.operator()( Point_3<Kernel> p)
Vector_3<Kernel>        t.operator()( Vector_3<Kernel> p)
Direction_3<Kernel>     t.operator()( Direction_3<Kernel> p)
Plane_3<Kernel>         t.operator()( Plane_3<Kernel> p)
```

Aff_transformation_3<Kernel>

t.operator(s)* composes two affine transformations.

Aff_transformation_3<Kernel>

t.inverse() gives the inverse transformation.

bool *t.is_even()* returns *true*, if the transformation is not reflecting, i.e. the determinant of the involved linear transformation is non-negative.

bool *t.is_odd()* returns *true*, if the transformation is reflecting.

The matrix entries of a matrix representation of a *Aff_transformation_3<Kernel>* can be accessed through the following member functions:

Kernel::FT *t.cartesian(int i, int j)*
Kernel::FT *t.m(int i, int j)* returns entry m_{ij} in a matrix representation in which m_{33} is 1.

Kernel::RT *t.homogeneous(int i, int j)*
Kernel::RT *t.hm(int i, int j)* returns entry m_{ij} in some fixed matrix representation.

For affine transformations no I/O operators are defined.

See Also

CGAL::Aff_transformation_2<Kernel> page 60
CGAL::Identity_transformation page 132
CGAL::Reflection page 132
CGAL::Rotation page 133
CGAL::Scaling page 133
CGAL::Translation page 134

CGAL::Direction_3<Kernel>

Definition

An object of the class *Direction_3<Kernel>* is a vector in the three-dimensional vector space \mathbb{R}^3 where we forget about their length. They can be viewed as unit vectors, although there is no normalization internally, since this is error prone. Directions are used whenever the length of a vector does not matter. They also characterize a set of parallel lines that have the same orientation or the direction normal to parallel planes that have the same orientation. For example, you can ask for the direction orthogonal to an oriented plane, or the direction of an oriented line.

Creation

<i>Direction_3<Kernel></i> <i>d</i> (<i>Vector_3<Kernel></i> <i>v</i>);	introduces a direction <i>d</i> initialised with the direction of vector <i>v</i> .
<i>Direction_3<Kernel></i> <i>d</i> (<i>Line_3<Kernel></i> <i>l</i>);	introduces the direction <i>d</i> of line <i>l</i> .
<i>Direction_3<Kernel></i> <i>d</i> (<i>Ray_3<Kernel></i> <i>r</i>);	introduces the direction <i>d</i> of ray <i>r</i> .
<i>Direction_3<Kernel></i> <i>d</i> (<i>Segment_3<Kernel></i> <i>s</i>);	introduces the direction <i>d</i> of segment <i>s</i> .
<i>Direction_3<Kernel></i> <i>d</i> (<i>Kernel::RT</i> <i>x</i> , <i>Kernel::RT</i> <i>y</i> , <i>Kernel::RT</i> <i>z</i>);	introduces a direction <i>d</i> initialised with the direction from the origin to the point with Cartesian coordinates (<i>x</i> , <i>y</i> , <i>z</i>).

Operations

<i>Kernel::RT</i>	<i>d</i> .delta(int <i>i</i>)	returns values, such that <i>d</i> == <i>Direction_3<Kernel></i> (delta(0),delta(1),delta(2)). Precondition: : 0 ≤ <i>i</i> ≤ 2.
<i>Kernel::RT</i>	<i>d</i> .dx()	returns delta(0).
<i>Kernel::RT</i>	<i>d</i> .dy()	returns delta(1).
<i>Kernel::RT</i>	<i>d</i> .dz()	returns delta(2).
bool	<i>d</i> .operator==(<i>e</i>)	Test for equality.
bool	<i>d</i> .operator!=(<i>e</i>)	Test for inequality.
<i>Direction_3<Kernel></i>	<i>d</i> .operator-()	The direction opposite to <i>d</i> .
<i>Vector_3<Kernel></i>	<i>d</i> .vector()	returns a vector that has the same direction as <i>d</i> .
<i>Direction_3<Kernel></i>	<i>d</i> .transform(<i>Aff_transformation_3<Kernel></i> <i>t</i>)	returns the direction obtained by applying <i>t</i> on <i>d</i> .

See Also

Kernel::Direction_3 page [352](#)

CGAL::Iso_cuboid_3<Kernel>

Definition

An object s of the data type *Iso_cuboid_3<Kernel>* is a cuboid in the Euclidean space \mathbb{E}^3 with edges parallel to the x , y and z axis of the coordinate system.

Although they are represented in a canonical form by only two vertices, namely the lexicographically smallest and largest vertex with respect to Cartesian xyz coordinates, we provide functions for “accessing” the other vertices as well.

Iso-oriented cuboids and bounding boxes are quite similar. The difference however is that bounding boxes have always double coordinates, whereas the coordinate type of an iso-oriented cuboid is chosen by the user.

Creation

Iso_cuboid_3<Kernel> c (*Point_3<Kernel>* p , *Point_3<Kernel>* q);

introduces an iso-oriented cuboid c with diagonal opposite vertices p and q . Note that the object is brought in the canonical form.

Iso_cuboid_3<Kernel> c (*Point_3<Kernel>* $left$,
Point_3<Kernel> $right$,
Point_3<Kernel> $bottom$,
Point_3<Kernel> top ,
Point_3<Kernel> far ,
Point_3<Kernel> $close$)

introduces an iso-oriented cuboid c whose minimal x coordinate is the one of $left$, the maximal x coordinate is the one of $right$, the minimal y coordinate is the one of $bottom$, the maximal y coordinate is the one of top , the minimal z coordinate is the one of far , the maximal z coordinate is the one of $close$.

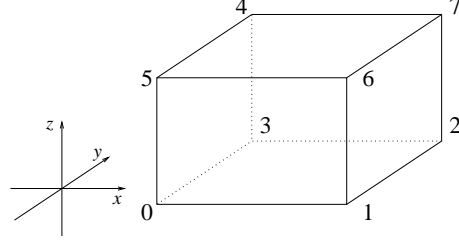
Iso_cuboid_3<Kernel> c (*Kernel::RT* min_hx ,
Kernel::RT min_hy ,
Kernel::RT min_hz ,
Kernel::RT max_hx ,
Kernel::RT max_hy ,
Kernel::RT max_hz ,
Kernel::RT $hw = RT(1)$)

introduces an iso-oriented cuboid c with diagonal opposite vertices $(min_hx/hw, min_hy/hw, min_hz/hw)$ and $(max_hx/hw, max_hy/hw, max_hz/hw)$.

Precondition: $hw \neq 0$.

Operations

<i>bool</i>	<i>c.operator==(c2)</i>	Test for equality: two iso-oriented cuboid are equal, iff their lower left and their upper right vertices are equal.
<i>bool</i>	<i>c.operator!=(c2)</i>	Test for inequality.
<i>Point_3<Kernel></i>	<i>c.vertex(int i)</i>	returns the i'th vertex modulo 8 of <i>c</i> . starting with the lower left vertex.
<i>Point_3<Kernel></i>	<i>c.operator[] (int i)</i>	returns <i>vertex(i)</i> , as indicated in the figure below:



<i>Point_3<Kernel></i>	<i>c.min()</i>	returns the smallest vertex of <i>c</i> (= <i>vertex(0)</i>).
<i>Point_3<Kernel></i>	<i>c.max()</i>	returns the largest vertex of <i>c</i> (= <i>vertex(7)</i>).
<i>Kernel::FT</i>	<i>c.xmin()</i>	returns smallest Cartesian <i>x</i> -coordinate in <i>c</i> .
<i>Kernel::FT</i>	<i>c.ymin()</i>	returns smallest Cartesian <i>y</i> -coordinate in <i>c</i> .
<i>Kernel::FT</i>	<i>c.zmin()</i>	returns smallest Cartesian <i>z</i> -coordinate in <i>c</i> .
<i>Kernel::FT</i>	<i>c.xmax()</i>	returns largest Cartesian <i>x</i> -coordinate in <i>c</i> .
<i>Kernel::FT</i>	<i>c.ymax()</i>	returns largest Cartesian <i>y</i> -coordinate in <i>c</i> .
<i>Kernel::FT</i>	<i>c.zmax()</i>	returns largest Cartesian <i>z</i> -coordinate in <i>c</i> .
<i>Kernel::FT</i>	<i>c.min_coord(int i)</i>	returns <i>i</i> -th Cartesian coordinate of the smallest vertex of <i>c</i> . <i>Precondition:</i> $0 \leq i \leq 2$.
<i>Kernel::FT</i>	<i>c.max_coord(int i)</i>	returns <i>i</i> -th Cartesian coordinate of the largest vertex of <i>c</i> . <i>Precondition:</i> $0 \leq i \leq 2$.

Predicates

<i>bool</i>	<i>c.is_degenerate()</i>	<i>c</i> is degenerate, if all vertices are collinear.
<i>Bounded_side</i>	<i>c.bounded_side(Point_3<Kernel> p)</i>	returns either <i>ON_UNBOUNDED_SIDE</i> , <i>ON_BOUNDED_SIDE</i> , or the constant <i>ON_BOUNDARY</i> , depending on where point <i>p</i> is.
<i>bool</i>	<i>c.has_on_boundary(Point_3<Kernel> p)</i>	
<i>bool</i>	<i>c.has_on_bounded_side(Point_3<Kernel> p)</i>	

bool *c.has_on_unbounded_side(Point_3<Kernel> p)*

Miscellaneous

Kernel::FT *c.volume()* returns the volume of *c*.

Bbox *c.bbox()* returns a bounding box containing *c*.

Iso_cuboid_3<Kernel> *c.transform(Aff_transformation_3<Kernel> t)*

returns the iso-oriented cuboid obtained by applying *t* on the smallest and the largest of *c*.

Precondition: The angle at a rotation must be a multiple of $\pi/2$, otherwise the resulting cuboid does not have the same size. Note that rotating about an arbitrary angle can even result in a degenerate iso-oriented cuboid.

See Also

Kernel::IsoCuboid_3 page [381](#)

CGAL::Line_3<Kernel>

Definition

An object l of the data type $Line_3<Kernel>$ is a directed straight line in the three-dimensional Euclidean space \mathbb{E}^3 .

Creation

$Line_3<Kernel> \ l(Point_3<Kernel> \ p, Point_3<Kernel> \ q);$

introduces a line l passing through the points p and q .
Line l is directed from p to q .

$Line_3<Kernel> \ l(Point_3<Kernel> \ p, Direction_3<Kernel> \ d);$

introduces a line l passing through point p with direction d .

$Line_3<Kernel> \ l(Point_3<Kernel> \ p, Vector_3<Kernel> \ v);$

introduces a line l passing through point p and oriented by v .

$Line_3<Kernel> \ l(Segment_3<Kernel> \ s);$

returns the line supporting the segment s , oriented from source to target.

$Line_3<Kernel> \ l(Ray_3<Kernel> \ r);$

returns the line supporting the ray r , with the same orientation.

Operations

$bool \quad \quad \quad l.operator==(h)$ Test for equality: two lines are equal, iff they have a non empty intersection and the same direction.

$bool \quad \quad \quad l.operator!=(h)$ Test for inequality.

$Point_3<Kernel> \quad \quad l.projection(Point_3<Kernel> \ p)$
returns the orthogonal projection of p on l .

$Point_3<Kernel> \quad \quad l.point(int \ i)$ returns an arbitrary point on l . It holds $point(i) = point(j)$, iff $i=j$.

Predicates

<i>bool</i>	<i>lis_degenerate()</i>	returns <i>true</i> iff line <i>l</i> is degenerated to a point.
-------------	-------------------------	------------------------------------------------------------------

bool *l.has_on(Point_3<Kernel> p)*
returns *true* iff *p* lies on *l*.

Miscellaneous

Plane_3<Kernel> *l.perpendicular_plane(Point_3<Kernel> p)*

returns the plane perpendicular to *l* passing through *p*.

Line_3<Kernel> *l.opposite()* returns the line with opposite direction.

Vector_3<Kernel> *l.to_vector()* returns a vector having the same direction as *l*.

Direction_3<Kernel> *l.direction()* returns the direction of *l*.

<i>Line_3<Kernel></i>	<i>l.transform(Aff_transformation_3<Kernel> t)</i>	
		returns the line obtained by applying <i>t</i> on a point on <i>l</i> and the direction of <i>l</i> .

See Also

Kernel:Line_3 page 400

CGAL::Plane_3<Kernel>

Definition

An object h of the data type $\text{Plane}_3<\text{Kernel}>$ is an oriented plane in the three-dimensional Euclidean space \mathbb{E}^3 . It is defined by the set of points with Cartesian coordinates (x, y, z) that satisfy the plane equation

$$h: ax + by + cz + d = 0.$$

The plane splits \mathbb{E}^3 in a *positive* and a *negative side*. A point p with Cartesian coordinates (px, py, pz) is on the positive side of h , iff $apx + bpy + cpz + d > 0$. It is on the negative side, iff $apx + bpy + cpz + d < 0$.

Creation

$\text{Plane}_3<\text{Kernel}> \ h(\text{Kernel}::\text{RT } a, \text{Kernel}::\text{RT } b, \text{Kernel}::\text{RT } c, \text{Kernel}::\text{RT } d);$

creates a plane h defined by the equation $apx + bpy + cpz + d = 0$. Notice that h is degenerate if $a = b = c$.

$\text{Plane}_3<\text{Kernel}> \ h(\text{Point}_3<\text{Kernel}> \ p, \text{Point}_3<\text{Kernel}> \ q, \text{Point}_3<\text{Kernel}> \ r);$

creates a plane h passing through the points p , q and r . The plane is oriented such that p , q and r are oriented in a positive sense (that is counterclockwise) when seen from the positive side of h . Notice that h is degenerate if the points are collinear.

$\text{Plane}_3<\text{Kernel}> \ h(\text{Point}_3<\text{Kernel}> \ p, \text{Vector}_3<\text{Kernel}> \ v);$

introduces a plane h that passes through point p and that is orthogonal to v .

$\text{Plane}_3<\text{Kernel}> \ h(\text{Point}_3<\text{Kernel}> \ p, \text{Direction}_3<\text{Kernel}> \ d);$

introduces a plane h that passes through point p and that has as an orthogonal direction equal to d .

$\text{Plane}_3<\text{Kernel}> \ h(\text{Line}_3<\text{Kernel}> \ l, \text{Point}_3<\text{Kernel}> \ p);$

introduces a plane h that is defined through the three points $l.\text{point}(0)$, $l.\text{point}(1)$ and p .

$\text{Plane}_3<\text{Kernel}> \ h(\text{Ray}_3<\text{Kernel}> \ r, \text{Point}_3<\text{Kernel}> \ p);$

introduces a plane h that is defined through the three points $r.\text{point}(0)$, $r.\text{point}(1)$ and p .

Plane_3<Kernel> h(Segment_3<Kernel> s, Point_3<Kernel> p);

introduces a plane *h* that is defined through the three points *s.source()*, *s.target()* and *p*.

Operations

bool *h.operator==(h2)* Test for equality: two planes are equal, iff they have a non empty intersection and the same orientation.

bool *h.operator!=(h2)* Test for inequality.

Kernel::RT *h.a()* returns the first coefficient of *h*.
Kernel::RT *h.b()* returns the second coefficient of *h*.
Kernel::RT *h.c()* returns the third coefficient of *h*.
Kernel::RT *h.d()* returns the fourth coefficient of *h*.

Line_3<Kernel> *h.perpendicular_line(Point_3<Kernel> p)*
 returns the line that is perpendicular to *h* and that passes through point *p*. The line is oriented from the negative to the positive side of *h*.

Point_3<Kernel> *h.projection(Point_3<Kernel> p)*
 returns the orthogonal projection of *p* on *h*.

Plane_3<Kernel> *h.opposite()* returns the plane with opposite orientation.

Point_3<Kernel> *h.point()* returns an arbitrary point on *h*.

Vector_3<Kernel> *h.orthogonal_vector()*
 returns a vector that is orthogonal to *h* and that is directed to the positive side of *h*.

Direction_3<Kernel> *h.orthogonal_direction()*
 returns the direction that is orthogonal to *h* and that is directed to the positive side of *h*.

Vector_3<Kernel> *h.base1()* returns a vector orthogonal to *orthogonal_vector()*.

Vector_3<Kernel> *h.base2()* returns a vector that is both orthogonal to *base1()*, and to *orthogonal_vector()*, and such that the result of *orientation(point(), point() + base1(), point()+base2(), point() + orthogonal_vector())* is positive.

2D Conversion

The following functions provide conversion between a plane and CGAL's two-dimensional space. The transformation is affine, but not necessarily an isometry. This means, the transformation preserves combinatorics, but not distances.

Point_2<Kernel> *h.to_2d(Point_3<Kernel> p)*

returns the image point of the projection of *p* under an affine transformation, which maps *h* onto the *xy*-plane, with the *z*-coordinate removed.

Point_3<Kernel> *h.to_3d(Point_2<Kernel> p)*

returns a point *q*, such that *to_2d(to_3d(p))* is equal to *p*.

Predicates

Oriented_side *h.oriented_side(Point_3<Kernel> p)*

returns either *ON_ORIENTED_BOUNDARY*, or the constant *ON_POSITIVE_SIDE*, or the constant *ON_NEGATIVE_SIDE*, determined by the position of *p* relative to the oriented plane *h*.

For convenience we provide the following boolean functions:

bool *h.has_on(Point_3<Kernel> p)*
bool *h.has_on_positive_side(Point_3<Kernel> p)*
bool *h.has_on_negative_side(Point_3<Kernel> p)*

bool *h.has_on(Line_3<Kernel> l)*

bool *h.is_degenerate()* Plane *h* is degenerate, if the coefficients *a*, *b*, and *c* of the plane equation are zero.

Miscellaneous

Plane_3<Kernel> *h.transform(Aff_transformation_3<Kernel> t)*

returns the plane obtained by applying *t* on a point of *h* and the orthogonal direction of *h*.

See Also

Kernel::Plane_3.....page [407](#)

CGAL::Point_3<Kernel>

Definition

An object of the class *Point_3<Kernel>* is a point in the three-dimensional Euclidean space \mathbb{E}^3 .

Remember that *Kernel::RT* and *Kernel::FT* denote a RingNumberType and a FieldNumberType, respectively. For the kernel model *Cartesian<T>*, the two types are the same. For the kernel model *Homogeneous<T>*, *Kernel::RT* is equal to *T*, and *Kernel::FT* is equal to *Quotient<T>*.

Types

Point_3<Kernel>::Cartesian_const_iterator

An iterator for enumerating the Cartesian coordinates of a point.

Creation

Point_3<Kernel> p(Origin ORIGIN);

introduces a point with Cartesian coordinates(0,0,0).

Point_3<Kernel> p(Kernel::RT hx, Kernel::RT hy, Kernel::RT hz, Kernel::RT hw = RT(1));

introduces a point *p* initialized to $(hx/hw, hy/hw, hz/hw)$.
Precondition: $hw \neq 0$.

Operations

bool p.operator==(q)

Test for equality: Two points are equal, iff their *x*, *y* and *z* coordinates are equal.

bool p.operator!=(q)

Test for inequality.

There are two sets of coordinate access functions, namely to the homogeneous and to the Cartesian coordinates. They can be used independently from the chosen kernel model.

Kernel::RT p.hx()

returns the homogeneous *x* coordinate.

<i>Kernel::RT</i>	<i>p.hy()</i>	
		returns the homogeneous y coordinate.
<i>Kernel::RT</i>	<i>p.hz()</i>	
		returns the homogeneous z coordinate.
<i>Kernel::RT</i>	<i>p.hw()</i>	
		returns the homogenizing coordinate.

Note that you do not lose information with the homogeneous representation, because the `FieldNumberType` is a quotient.

<i>Kernel::FT</i>	<i>p.x()</i>	returns the Cartesian x coordinate, that is hx/hw .
<i>Kernel::FT</i>	<i>p.y()</i>	returns the Cartesian y coordinate, that is hy/hw .
<i>Kernel::FT</i>	<i>p.z()</i>	returns the Cartesian z coordinate, that is hz/hw .

The following operations are for convenience and for compatibility with code for higher dimensional points. Again they come in a Cartesian and in a homogeneous flavor.

<i>Kernel::RT</i>	<i>p.homogeneous(int i)</i>	
		returns the i 'th homogeneous coordinate of p , starting with 0. <i>Precondition:</i> $0 \leq i \leq 3$.

<i>Kernel::FT</i>	<i>p.cartesian(int i)</i>	
		returns the i 'th Cartesian coordinate of p , starting with 0. <i>Precondition:</i> $0 \leq i \leq 2$.

<i>Kernel::FT</i>	<i>p.operator[](int i)</i>	
		returns <i>cartesian(i)</i> . <i>Precondition:</i> $0 \leq i \leq 2$.

<i>Cartesian_const_iterator</i>	<i>p.cartesian_begin()</i>	
		returns an iterator to the Cartesian coordinates of p , starting with the 0th coordinate.

<i>Cartesian_const_iterator</i>	<i>p.cartesian_end()</i>	
		returns an off the end iterator to the Cartesian coordinates of p .

<i>int</i>	<i>p.dimension()</i>	
		returns the dimension (the constant 3).

Bbox_3 *p.bbox()*
returns a bounding box containing *p*.

Point_3<Kernel> *p.transform(Aff_transformation_3<Kernel> t)*
returns the point obtained by applying *t* on *p*.

Operators

The following operations can be applied on points:

bool *operator<(p, q)*
returns true iff *p* is lexicographically smaller than *q* (the lexicographical order being defined on the Cartesian coordinates).

bool *operator>(p, q)*
returns true iff *p* is lexicographically greater than *q*.

bool *operator<=(p, q)*
returns true iff *p* is lexicographically smaller or equal to *q*.

bool *operator>=(p, q)*
returns true iff *p* is lexicographically greater or equal to *q*.

Vector_3<Kernel> *operator-(p, q)*
returns the difference vector between *q* and *p*. You can substitute *ORIGIN* for either *p* or *q*, but not for both.

Point_3<Kernel> *operator+(p, Vector_3<Kernel> v)*
returns the point obtained by translating *p* by the vector *v*.

Point_3<Kernel> *operator-(p, Vector_3<Kernel> v)*
returns the point obtained by translating *p* by the vector *-v*.

See Also

Kernel::Point_3.....page [410](#)

CGAL::Ray_3<Kernel>

Definition

An object r of the data type $\text{Ray}_3<\text{Kernel}>$ is a directed straight ray in the three-dimensional Euclidean space \mathbb{E}^3 . It starts in a point called the *source* of r and it goes to infinity.

Creation

$\text{Ray}_3<\text{Kernel}> \ r(\text{Point}_3<\text{Kernel}> \ p, \text{Point}_3<\text{Kernel}> \ q);$

introduces a ray r with source p and passing through point q .

$\text{Ray}_3<\text{Kernel}> \ r(\text{Point}_3<\text{Kernel}> \ p, \text{Direction}_3<\text{Kernel}> \ d);$

introduces a ray r with source p and with direction d .

$\text{Ray}_3<\text{Kernel}> \ r(\text{Point}_3<\text{Kernel}> \ p, \text{Vector}_3<\text{Kernel}> \ v);$

introduces a ray r with source p and with a direction given by v .

$\text{Ray}_3<\text{Kernel}> \ r(\text{Point}_3<\text{Kernel}> \ p, \text{Line}_3<\text{Kernel}> \ l);$

introduces a ray r starting at source p with the same direction as l .

Operations

<i>bool</i>	$r.operator==(h)$	Test for equality: two rays are equal, iff they have the same source and the same direction.
<i>bool</i>	$r.operator!=(h)$	Test for inequality.
$\text{Point}_3<\text{Kernel}>$	$r.source()$	returns the source of r
$\text{Point}_3<\text{Kernel}>$	$r.point(\text{int } i)$	returns a point on r . $point(0)$ is the source. $point(i)$, with $i > 0$, is different from the source. <i>Precondition:</i> $i \geq 0$.
$\text{Direction}_3<\text{Kernel}>$	$r.direction()$	returns the direction of r .
$\text{Vector}_3<\text{Kernel}>$	$r.to_vector()$	returns a vector giving the direction of r .

<i>Line_3<Kernel></i>	<i>r.supporting_line()</i>	returns the line supporting <i>r</i> which has the same direction.
<i>Ray_3<Kernel></i>	<i>r.opposite()</i>	returns the ray with the same source and the opposite direction.
<i>bool</i>	<i>r.is_degenerate()</i>	ray <i>r</i> is degenerate, if the source and the second defining point fall together (that is if the direction is degenerate).
<i>bool</i>	<i>r.has_on(Point_3<Kernel> p)</i>	A point is on <i>r</i> , iff it is equal to the source of <i>r</i> , or if it is in the interior of <i>r</i> .
<i>Ray_3<Kernel></i>	<i>r.transform(Aff_transformation_3<Kernel> t)</i>	returns the ray obtained by applying <i>t</i> on the source and on the direction of <i>r</i> .

See Also

Kernel::Ray_3 page [413](#)

CGAL::Segment_3<Kernel>

Definition

An object s of the data type $\text{Segment}_3<\text{Kernel}>$ is a directed straight line segment in the three-dimensional Euclidean space \mathbb{E}^3 , i.e. a straight line segment $[p, q]$ connecting two points $p, q \in \mathbb{R}^3$. The segment is topologically closed, i.e. the end points belong to it. Point p is called the *source* and q is called the *target* of s . The length of s is the Euclidean distance between p and q . Note that there is only a function to compute the square of the length, because otherwise we had to perform a square root operation which is not defined for all number types, which is expensive, and may not be exact.

Creation

$\text{Segment}_3<\text{Kernel}> \ s(\text{Point}_3<\text{Kernel}> \ p, \text{Point}_3<\text{Kernel}> \ q);$

introduces a segment s with source p and target q . It is directed from the source towards the target.

Operations

$bool$	$s.operator==(q)$	Test for equality: Two segments are equal, iff their sources and targets are equal.
$bool$	$s.operator!=(q)$	Test for inequality.
$\text{Point}_3<\text{Kernel}>$	$s.source()$	returns the source of s .
$\text{Point}_3<\text{Kernel}>$	$s.target()$	returns the target of s .
$\text{Point}_3<\text{Kernel}>$	$s.min()$	returns the point of s with smallest coordinate (lexicographically).
$\text{Point}_3<\text{Kernel}>$	$s.max()$	returns the point of s with largest coordinate (lexicographically).
$\text{Point}_3<\text{Kernel}>$	$s.vertex(int\ i)$	returns source or target of s : $vertex(0)$ returns the source, $vertex(1)$ returns the target. The parameter i is taken modulo 2, which gives easy access to the other vertex.
$\text{Point}_3<\text{Kernel}>$	$s.point(int\ i)$	returns $vertex(i)$.
$\text{Point}_3<\text{Kernel}>$	$s.operator[] (int\ i)$	returns $vertex(i)$.
$\text{Kernel}::FT$	$s.squared_length()$	returns the squared length of s .
$\text{Vector}_3<\text{Kernel}>$	$s.to_vector()$	returns the vector $s.target() - s.source()$.
$\text{Direction}_3<\text{Kernel}>$	$s.direction()$	returns the direction from source to target.

<i>Segment_3<Kernel></i>	<i>s.opposite()</i>	returns a segment with source and target interchanged.
<i>Line_3<Kernel></i>	<i>s.supporting_line()</i>	returns the line <i>l</i> passing through <i>s</i> . Line <i>l</i> has the same orientation as segment <i>s</i> , that is from the source to the target of <i>s</i> .
<i>bool</i>	<i>s.is_degenerate()</i>	segment <i>s</i> is degenerate, if source and target fall together.
<i>bool</i>	<i>s.has_on(Point_3<Kernel> p)</i>	A point is on <i>s</i> , iff it is equal to the source or target of <i>s</i> , or if it is in the interior of <i>s</i> .
<i>Bbox_3</i>	<i>s.bbox()</i>	returns a bounding box containing <i>s</i> .
<i>Segment_3<Kernel></i>	<i>s.transform(Aff_transformation_3<Kernel> t)</i>	returns the segment obtained by applying <i>t</i> on the source and the target of <i>s</i> .

See Also

Kernel::Segment_3.....page [415](#)

CGAL::Sphere_3<Kernel>

Definition

An object of type *Sphere_3<Kernel>* is a sphere in the three-dimensional Euclidean space \mathbb{E}^3 . The sphere is oriented, i.e. its boundary has clockwise or counterclockwise orientation. The boundary splits \mathbb{E}^3 into a positive and a negative side, where the positive side is to the left of the boundary. The boundary also splits \mathbb{E}^3 into a bounded and an unbounded side. Note that the sphere can be degenerated, i.e. the squared radius may be zero.

Creation

Sphere_3<Kernel> *c*(*Point_3<Kernel>* *center*,
 Kernel::FT *squared_radius*,
 Orientation *orientation* = *COUNTERCLOCKWISE*)

introduces a variable *c* of type *Sphere_3<Kernel>*. It is initialized to the sphere with center *center*, squared radius *squared_radius* and orientation *orientation*.

Precondition: *orientation* \neq *COPLANAR*, and furthermore, *squared_radius* \geq 0.

Sphere_3<Kernel> *c*(*Point_3<Kernel>* *p*, *Point_3<Kernel>* *q*, *Point_3<Kernel>* *r*, *Point_3<Kernel>* *s*);

introduces a variable *c* of type *Sphere_3<Kernel>*. It is initialized to the unique sphere which passes through the points *p*, *q*, *r* and *s*. The orientation of the sphere is the orientation of the point quadruple *p*, *q*, *r*, *s*.

Precondition: *p*, *q*, *r*, and *s* are not collinear.

Sphere_3<Kernel> *c*(*Point_3<Kernel>* *p*,
 Point_3<Kernel> *q*,
 Point_3<Kernel> *r*,
 Orientation *o* = *COUNTERCLOCKWISE*)

introduces a variable *c* of type *Sphere_3<Kernel>*. It is initialized to the smallest sphere which passes through the points *p*, *q*, and *r*. The orientation of the sphere is *o*.

Precondition: *o* is not *COPLANAR*.

Sphere_3<Kernel> *c*(*Point_3<Kernel>* *p*, *Point_3<Kernel>* *q*, *Orientation* *o* = *COUNTERCLOCKWISE*);

introduces a variable *c* of type *Sphere_3<Kernel>*. It is initialized to the smallest sphere which passes through the points *p* and *q*. The orientation of the sphere is *o*.

Precondition: *o* is not *COPLANAR*.

Sphere_3<Kernel> *c*(*Point_3<Kernel>* *center*, *Orientation* *orientation* = *COUNTERCLOCKWISE*);

introduces a variable *c* of type *Sphere_3<Kernel>*. It is initialized to the sphere with center *center*, squared radius zero and orientation *orientation*.

Precondition: *orientation* \neq *COPLANAR*.

Postcondition: *c.is_degenerate()* = *true*.

Access Functions

<i>Point_3<Kernel></i> <i>Kernel::FT</i> <i>orientation</i>	<i>c.center()</i> <i>c.squared_radius()</i> <i>c.orientation()</i>	returns the center of <i>c</i> . returns the squared radius of <i>c</i> . returns the orientation of <i>c</i> .
<i>bool operator</i>	<i>c.==(sphere2)</i>	returns <i>true</i> , iff <i>c</i> and <i>sphere2</i> are equal, i.e. if they have the same center, same squared radius and same orientation.
<i>bool operator</i>	<i>c.!=(sphere2)</i>	returns <i>true</i> , iff <i>c</i> and <i>sphere2</i> are not equal.

Predicates

<i>bool</i>	<i>c.is_degenerate()</i>	returns <i>true</i> , iff <i>c</i> is degenerate, i.e. if <i>c</i> has squared radius zero.
<i>Oriented_side</i>	<i>c.oriented_side(Point_3<Kernel> p)</i>	returns either the constant <i>ON_ORIENTED_BOUNDARY</i> , <i>ON_POSITIVE_SIDE</i> , or <i>ON_NEGATIVE_SIDE</i> , iff <i>p</i> lies on the boundary, properly on the positive side, or properly on the negative side of <i>c</i> , resp.
<i>Bounded_side</i>	<i>c.bounded_side(Point_3<Kernel> p)</i>	returns <i>ON_BOUNDED_SIDE</i> , <i>ON_BOUNDARY</i> , or <i>ON_UNBOUNDED_SIDE</i> iff <i>p</i> lies properly inside, on the boundary, or properly outside of <i>c</i> , resp.
<i>bool</i>	<i>c.has_on_positive_side(Point_3<Kernel> p)</i>	
<i>bool</i>	<i>c.has_on_negative_side(Point_3<Kernel> p)</i>	
<i>bool</i>	<i>c.has_on_boundary(Point_3<Kernel> p)</i>	
<i>bool</i>	<i>c.has_on_bounded_side(Point_3<Kernel> p)</i>	
<i>bool</i>	<i>c.has_on_unbounded_side(Point_3<Kernel> p)</i>	

Miscellaneous

<i>Sphere_3<Kernel></i>	<i>c.opposite()</i>	returns the sphere with the same center and squared radius as <i>c</i> but with opposite orientation.
-------------------------------	---------------------	-------------------------------------------------------------------------------------------------------

<i>Sphere_3<Kernel></i>	<i>c.orthogonal_transform(Aff_transformation_3<Kernel> at)</i>	<p>returns the sphere obtained by applying <i>at</i> on <i>c</i>. <i>Precondition:</i> <i>at</i> is an orthogonal transformation.</p>
-------------------------------	-----------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------

<i>Bbox_3</i>	<i>c.bbox()</i>	returns a bounding box containing <i>c</i> .
---------------	-----------------	----------------------------------------------

See Also

Kernel::Sphere_3 page [420](#)

CGAL::Tetrahedron_3<Kernel>

Definition

An object t of the class *Tetrahedron_3<Kernel>* is an oriented tetrahedron in the three-dimensional Euclidean space \mathbb{E}^3 .

It is defined by four vertices p_0, p_1, p_2 and p_3 . The orientation of a tetrahedron is the orientation of its four vertices. That means it is positive when p_3 is on the positive side of the plane defined by p_0, p_1 and p_2 .

The tetrahedron itself splits the space \mathbb{E}_3 in a *positive* and a *negative* side.

The boundary of a tetrahedron splits the space in two open regions, a bounded one and an unbounded one.

Creation

```
Tetrahedron_3<Kernel> t( Point_3<Kernel> p0,
                        Point_3<Kernel> p1,
                        Point_3<Kernel> p2,
                        Point_3<Kernel> p3)
```

introduces a tetrahedron t with vertices p_0, p_1, p_2 and p_3 .

Operations

<i>bool</i>	<i>t.operator==(t2)</i>	Test for equality: two tetrahedra t and $t2$ are equal, iff t and $t2$ have the same orientation and their sets (not sequences) of vertices are equal.
<i>bool</i>	<i>t.operator!=(t2)</i>	Test for inequality.
<i>Point_3<Kernel></i>	<i>t.vertex(int i)</i>	returns the i 'th vertex modulo 4 of t .
<i>Point_3<Kernel></i>	<i>t.operator[](int i)</i>	returns <i>vertex(int i)</i> .

Predicates

<i>bool</i>	<i>t.is_degenerate()</i>	Tetrahedron t is degenerate, if the vertices are coplanar.
<i>Orientation</i>	<i>t.orientation()</i>	
<i>Oriented_side</i>	<i>t.oriented_side(Point_3<Kernel> p)</i>	

Precondition: : t is not degenerate.

Bounded_side *t.bounded_side(Point_3<Kernel> p)*

Precondition: : *t* is not degenerate.

For convenience we provide the following boolean functions:

bool *t.has_on_positive_side(Point_3<Kernel> p)*
bool *t.has_on_negative_side(Point_3<Kernel> p)*
bool *t.has_on_boundary(Point_3<Kernel> p)*
bool *t.has_on_bounded_side(Point_3<Kernel> p)*
bool *t.has_on_unbounded_side(Point_3<Kernel> p)*

Miscellaneous

Kernel::FT *t.volume()* returns the signed volume of *t*.

Bbox_3 *t.bbox()* returns a bounding box containing *t*.

Tetrahedron_3<Kernel> *t.transform(Aff_transformation_3<Kernel> at)*

returns the tetrahedron obtained by applying *at* on the three vertices of *t*.

See Also

Kernel::Tetrahedron_3 page [421](#)

CGAL::Triangle_3<Kernel>

Definition

An object t of the class *Triangle_3<Kernel>* is a triangle in the three-dimensional Euclidean space \mathbb{E}^3 . As the triangle is not a full-dimensional object there is only a test whether a point lies on the triangle or not.

Creation

Triangle_3<Kernel> t (*Point_3<Kernel>* p , *Point_3<Kernel>* q , *Point_3<Kernel>* r);

introduces a triangle t with vertices p , q and r .

Operations

bool $t.operator==(t_2)$ Test for equality: two triangles t and t_2 are equal, iff there exists a cyclic permutation of the vertices of t_2 , such that they are equal to the vertices of t .

bool $t.operator!=(t_2)$ Test for inequality.

Point_3<Kernel> $t.vertex(int\ i)$ returns the i 'th vertex modulo 3 of t .

Point_3<Kernel> $t.operator[](int\ i)$ returns $vertex(int\ i)$.

Plane_3<Kernel> $t.supporting_plane()$ returns the supporting plane of t , with same orientation.

Predicates

bool $t.is_degenerate()$ t is degenerate if its vertices are collinear.

bool $t.has_on(Point_3<Kernel>\ p)$

A point is on t , if it is on a vertex, an edge or the face of t .

Miscellaneous

Kernel::FT $t.squared_area()$ returns a square of the area of t .

Bbox_3 $t.bbox()$ returns a bounding box containing t .

Triangle_3<Kernel> *t.transform(Aff_transformation_3<Kernel> at)*

returns the triangle obtained by applying *at* on the three vertices of *t*.

See Also

Kernel::Triangle_3 page [423](#)

CGAL::Vector_3<Kernel>

Definition

An object of the class *Vector_3<Kernel>* is a vector in the three-dimensional vector space \mathbb{R}^3 . Geometrically spoken a vector is the difference of two points p_2, p_1 and denotes the direction and the distance from p_1 to p_2 .

CGAL defines a symbolic constant *NULL_VECTOR*. We will explicitly state where you can pass this constant as an argument instead of a vector initialized with zeros.

Creation

Vector_3<Kernel> *v*(*Point_3<Kernel>* *a*, *Point_3<Kernel>* *b*);

introduces the vector $b - a$.

Vector_3<Kernel> *v*(*Segment_3<Kernel>* *s*);

introduces the vector $s.target() - s.source()$.

Vector_3<Kernel> *v*(*Ray_3<Kernel>* *r*);

introduces a vector having the same direction as *r*.

Vector_3<Kernel> *v*(*Line_3<Kernel>* *l*);

introduces a vector having the same direction as *l*.

Vector_3<Kernel> *v*(*Null_vector* *NULL_VECTOR*);

introduces a null vector *v*.

Vector_3<Kernel> *v*(*Kernel::RT* *hx*, *Kernel::RT* *hy*, *Kernel::FT* *hz*, *Kernel::RT* *hw* = *RT(1)*);

introduces a vector *v* initialized to $(hx/hw, hy/hw, hz/hw)$.

Operations

bool

v.operator==(w)

Test for equality: two vectors are equal, iff their *x*, *y* and *z* coordinates are equal. You can compare a vector with the *NULL_VECTOR*.

bool

v.operator!=(w)

Test for inequality. You can compare a vector with the *NULL_VECTOR*.

There are two sets of coordinate access functions, namely to the homogeneous and to the Cartesian coordinates. They can be used independently from the chosen kernel model.

Kernel::RT

v.hx()

returns the homogeneous *x* coordinate.

Kernel::RT

v.hy()

returns the homogeneous *y* coordinate.

Kernel::RT

v.hz()

returns the homogeneous *z* coordinate.

<i>Kernel::RT</i>	<i>v.hw()</i>	returns the homogenizing coordinate.
-------------------	---------------	--------------------------------------

Note that you do not lose information with the homogeneous representation, because the `FieldNumberType` is a quotient.

<i>Kernel::FT</i>	<i>v.x()</i>	returns the x -coordinate of v , that is hx/hw .
<i>Kernel::FT</i>	<i>v.y()</i>	returns the y -coordinate of v , that is hy/hw .
<i>Kernel::FT</i>	<i>v.z()</i>	returns the z coordinate of v , that is hz/hw .

The following operations are for convenience and for compatibility with higher dimensional vectors. Again they come in a Cartesian and homogeneous flavor.

<i>Kernel::RT</i>	<i>v.homogeneous(int i)</i>	returns the i 'th homogeneous coordinate of v , starting with 0. <i>Precondition:</i> $0 \leq i \leq 3$.
-------------------	-------------------------------	------------------------------------------------------------------------------------------------------------------

<i>Kernel::FT</i>	<i>v.cartesian(int i)</i>	returns the i 'th Cartesian coordinate of v , starting at 0. <i>Precondition:</i> $0 \leq i \leq 2$.
-------------------	-----------------------------	--------------------------------------------------------------------------------------------------------------

<i>Kernel::FT</i>	<i>v.operator[] (int i)</i>	returns <i>cartesian</i> (i). <i>Precondition:</i> $0 \leq i \leq 2$.
-------------------	-------------------------------	-------------------------------------------------------------------------------

<i>int</i>	<i>v.dimension()</i>	returns the dimension (the constant 3).
------------	----------------------	-----------------------------------------

<i>Vector_3<Kernel></i>	<i>v.transform(Aff_transformation_3<Kernel> t)</i>	returns the vector obtained by applying t on v .
-------------------------------	------------------------------------------------------------	------------------------------------------------------

<i>Direction_3<Kernel></i>	<i>v.direction()</i>	returns the direction of v .
----------------------------------	----------------------	--------------------------------

Operators

The following operations can be applied on vectors:

<i>Vector_3<Kernel></i>	<i>v.operator+(w)</i>	Addition.
-------------------------------	-------------------------	-----------

<i>Vector_3<Kernel></i>	<i>v.operator-(w)</i>	Subtraction.
-------------------------------	-------------------------	--------------

<i>Vector_3<Kernel></i>	<i>v.operator-()</i>	Returns the opposite vector.
-------------------------------	----------------------	------------------------------

<i>Kernel::FT</i>	<i>v.operator*(w)</i>	returns the scalar product (= inner product) of the two vectors.
-------------------	-------------------------	------------------------------------------------------------------

<i>Vector_3<Kernel></i>	<i>operator*(v, Kernel::RT s)</i>	Multiplication with a scalar from the right.
-------------------------------	-------------------------------------	----------------------------------------------

<i>Vector_3<Kernel></i>	<i>operator*(v, Kernel::FT s)</i>	Multiplication with a scalar from the right.
-------------------------------	-------------------------------------	----------------------------------------------

Vector_3<Kernel> *operator*(Kernel::RT s, v)*
 Multiplication with a scalar from the left.

Vector_3<Kernel> *operator*(Kernel::FT s, v)*
 Multiplication with a scalar from the left.

Vector_3<Kernel> *v.operator/(Kernel::RT s)*
 Division by a scalar.

See Also

Kernel::Vector_3 page [425](#)
CGAL::cross_product page [164](#)

2.10.3 Polymorphic Objects

CGAL::Object

`#include <CGAL/Object.h>`

Definition

Some functions can return different types of objects. A typical C++ solution to this problem is to derive all possible return types from a common base class, to return a pointer to this class and to perform a dynamic cast on this pointer. The class *Object* provides an abstraction. An object *obj* of the class *Object* can represent an arbitrary class. The only operations it provides is to make copies and assignments, so that you can put them in lists or arrays. Note that *Object* is NOT a common base class for the elementary classes. Therefore, there is no automatic conversion from these classes to *Object*. Rather this is done with the global function *make_object*. This encapsulation mechanism requires the use of *assign* or *object_cast* to use the functionality of the encapsulated class.

Creation

Object obj; introduces an empty object.

Object obj(o); Copy constructor.

Objects of type *Object* are normally created using the global function *make_object*.

Operations

Object& *obj.operator=(o)* Assignment.

bool *obj.is_empty()* returns true, if *obj* does not contain an object.

std::type_info *obj.type()* returns the type information of the contained type, or *typeid(void)* if empty.

Construction of an *Object* storing an object of type *T* can be performed using the *make_object* global function :

template <class T>
Object *make_object(T t)* Creates an object that contains *t*.

Assignment of an object of type *Object* to an object of type *T* can be done using *assign* :

template <class T>
bool *assign(T& c, o)* assigns *o* to *c* if *o* was constructed from an object of type *T*. Returns *true*, if the assignment was possible.

Another possibility to access the encapsulated object is to use *object_cast*, which avoids the default constructor and assignment required by *assign* :

template <class T>

*const T** *object_cast(const * o)*

Returns a pointer to the object of type *T* stored by *o*, if any, otherwise returns *NULL*.

template <class T>

T *object_cast(o)*

Returns a copy of the object of type *T* stored by *o*, if any, otherwise throws an exception of type *Bad_object_cast*.

Example

In the following example, the object class is used as return value for the intersection computation, as there are possibly different return values.

```
{
    typedef Cartesian<double>    K;
    typedef K::Point_2          Point_2;
    typedef K::Segment_2         Segment_2;

    Point_2 point;
    Segment_2 segment, segment_1, segment_2;

    std::cin >> segment_1 >> segment_2;

    Object obj = intersection(segment_1, segment_2);

    if (assign(point, obj)) {
        /* do something with point */
    } else if (assign(segment, obj)) {
        /* do something with segment */
    }

    /* there was no intersection */
}
```

Another way to access the object is to use *object_cast*, which allows to skip a default construction and assignment :

```
{
    typedef Cartesian<double>    K;
    typedef K::Point_2          Point_2;
    typedef K::Segment_2         Segment_2;

    Segment_2 segment_1, segment_2;

    std::cin >> segment_1 >> segment_2;

    Object obj = intersection(segment_1, segment_2);

    if (const Point_2 * point = object_cast<Point_2>(&obj)) {
        /* do something with *point */
    }
```

```

    } else if (const Segment_2 * segment = object_cast<Segment_2>(&obj)) {
        /* do something with *segment*/
    }

    /* there was no intersection */
}

```

The intersection routine itself looks roughly as follows:

```

template < class Kernel >
Object intersection(Segment_2<Kernel> s1, Segment_2<Kernel> s2)
{

    if (/* intersection is a point */ ) {

        Point_2<Kernel> p = ... ;
        return make_object(p);

    } else if (/* intersection is a segment */ ) {

        Segment_2<Kernel> s = ... ;
        return make_object(s);
    }

    /* empty intersection */
    return Object();
}

```

See Also

Kernel::Object_2.....page [401](#)
Kernel::Object_3.....page [402](#)

2.11 Constants and Enumerations

CGAL::Angle

```
#include <CGAL/enum.h>
```

```
enum Angle { OBTUSE, RIGHT, ACUTE};
```

See Also

CGAL::anglepage [135](#)

Enum

CGAL::Bounded_side

```
#include <CGAL/enum.h>
```

```
enum Bounded_side { ON_UNBOUNDED_SIDE, ON_BOUNDARY, ON_BOUNDED_SIDE};
```

Enum

CGAL::Comparison_result

```
#include <CGAL/enum.h>
```

```
enum Comparison_result { SMALLER, EQUAL, LARGER};
```

CGAL::Sign

```
#include <CGAL/enum.h>
```

```
enum Sign { NEGATIVE, ZERO, POSITIVE};
```

See Also

CGAL::Orientation page [125](#)

CGAL::Orientation

```
#include <CGAL/enum.h>
```

```
typedef Sign          Orientation;
```

See Also

[CGAL::LEFT_TURN](#)page [127](#)
[CGAL::RIGHT_TURN](#)page [127](#)
[CGAL::COLLINEAR](#)page [127](#)
[CGAL::CLOCKWISE](#)page [126](#)
[CGAL::COUNTERCLOCKWISE](#)page [126](#)
[CGAL::COPLANAR](#)page [128](#)

CGAL::Oriented_side

```
#include <CGAL/enum.h>
```

```
enum Oriented_side { ON_NEGATIVE_SIDE, ON_ORIENTED_BOUNDARY, ON_POSITIVE_SIDE};
```

CGAL::CLOCKWISE

const Orientation *CLOCKWISE = NEGATIVE;*

See Also

CGAL::COUNTERCLOCKWISEpage [126](#)

CGAL::COUNTERCLOCKWISE

const Orientation *COUNTERCLOCKWISE = POSITIVE;*

See Also

CGAL::CLOCKWISEpage [126](#)

CGAL::COLLINEAR

const Orientation *COLLINEAR = ZERO;*

See Also

CGAL::LEFT_TURN [page 127](#)
CGAL::RIGHT_TURN [page 127](#)

Constant

CGAL::LEFT_TURN

const Orientation *LEFT_TURN = POSITIVE;*

See Also

CGAL::COLLINEAR [page 127](#)
CGAL::RIGHT_TURN [page 127](#)

Constant

CGAL::RIGHT_TURN

const Orientation *RIGHT_TURN = NEGATIVE;*

See Also

CGAL::COLLINEAR [page 127](#) *CGAL::LEFT_TURN* [page 127](#)

Constant

CGAL::COPLANAR

const Orientation *COPLANAR = ZERO;*

CGAL::DEGENERATE

const Orientation *DEGENERATE = ZERO;*

CGAL::Null_vector

```
#include <CGAL/Origin.h>
```

Definition

CGAL defines a symbolic constant *NULL_VECTOR* to construct zero length vectors. *Null_vector* is the type of this constant.

See Also

CGAL::Vector_2<Kernel> page [85](#)
CGAL::Vector_3<Kernel> page [116](#)

CGAL::NULL_VECTOR

```
const Null_vector      NULL_VECTOR;
```

Definition

A symbolic constant used to construct zero length vectors.

See Also

CGAL::Vector_2<Kernel> page [85](#)
CGAL::Vector_3<Kernel> page [116](#)

CGAL::Origin

`#include <CGAL/Origin.h>`

Definition

CGAL defines a symbolic constant *ORIGIN* which denotes the point at the origin. *Origin* is the type of this constant. It is used in the conversion between points and vectors.

See Also

CGAL::Point_2<Kernel> page [75](#)
CGAL::Point_3<Kernel> page [102](#)
CGAL::Vector_2<Kernel> page [85](#)
CGAL::Vector_3<Kernel> page [116](#)
CGAL::operator+ page [187](#)
CGAL::operator- page [188](#)

CGAL::ORIGIN

`const Origin` *ORIGIN*;

Definition

A symbolic constant which denotes the point at the origin. This constant is used in the conversion between points and vectors.

Example

```
Point_2< Cartesian<Exact_NT> >  p(1.0, 1.0), q;
Vector2< Cartesian<Exact_NT> >  v;
v = p - ORIGIN;
q = ORIGIN + v;
assert( p == q );
```

See Also

<i>CGAL::Point_2<Kernel></i>	page 75
<i>CGAL::Point_3<Kernel></i>	page 102

CGAL::Identity_transformation

`#include <CGAL/aff_transformation_tags.h>`

Definition

Tag class for affine transformations.

See Also

[CGAL::Aff_transformation_2<Kernel>](#) page 60
[CGAL::Aff_transformation_3<Kernel>](#) page 89
[CGAL::Reflection](#) page 132
[CGAL::Rotation](#) page 133
[CGAL::Scaling](#) page 133
[CGAL::Translation](#) page 134

CGAL::Reflection

`#include <CGAL/aff_transformation_tags.h>`

Definition

Tag class for affine transformations.

See Also

[CGAL::Aff_transformation_2<Kernel>](#) page 60
[CGAL::Aff_transformation_3<Kernel>](#) page 89
[CGAL::Identity_transformation](#) page 132
[CGAL::Rotation](#) page 133
[CGAL::Scaling](#) page 133
[CGAL::Translation](#) page 134

CGAL::Rotation

`#include <CGAL/aff_transformation_tags.h>`

Definition

Tag class for affine transformations.

See Also

[CGAL::Aff_transformation_2<Kernel>](#) page 60
[CGAL::Aff_transformation_3<Kernel>](#) page 89
[CGAL::Identity_transformation](#) page 132
[CGAL::Rotation](#) page 133
[CGAL::Scaling](#) page 133
[CGAL::Translation](#) page 134

CGAL::Scaling

`#include <CGAL/aff_transformation_tags.h>`

Definition

Tag class for affine transformations.

See Also

[CGAL::Aff_transformation_2<Kernel>](#) page 60
[CGAL::Aff_transformation_3<Kernel>](#) page 89
[CGAL::Identity_transformation](#) page 132
[CGAL::Reflection](#) page 132
[CGAL::Rotation](#) page 133
[CGAL::Translation](#) page 134

CGAL::Translation

`#include <CGAL/aff_transformation_tags.h>`

Definition

Tag class for affine transformations.

See Also

[CGAL::Aff_transformation_2<Kernel>](#) page [60](#)
[CGAL::Aff_transformation_3<Kernel>](#) page [89](#)
[CGAL::Identity_transformation](#) page [132](#)
[CGAL::Reflection](#) page [132](#)
[CGAL::Rotation](#) page [133](#)
[CGAL::Scaling](#) page [133](#)

2.12 Global Functions

CGAL::angle

Angle *angle(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r)*

returns *OBTUSE*, *RIGHT* or *ACUTE* depending on the angle formed by the three points *p*, *q*, *r* (*q* being the vertex of the angle).

Angle *angle(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r)*

returns *OBTUSE*, *RIGHT* or *ACUTE* depending on the angle formed by the three points *p*, *q*, *r* (*q* being the vertex of the angle).

CGAL::are_ordered_along_line

```
bool are_ordered_along_line( Point_2<Kernel> p,
                             Point_2<Kernel> q,
                             Point_2<Kernel> r)
```

returns *true*, iff the three points are collinear and q lies between p and r . Note that *true* is returned, if $q=p$ or $q=r$.

```

bool      are_ordered_along_line( Point_3<Kernel> p,
                                   Point_3<Kernel> q,
                                   Point_3<Kernel> r)

```

returns *true*, iff the three points are collinear and q lies between p and r . Note that *true* is returned, if $q=p$ or $q=r$.

See Also

<i>CGAL::are_strictly_ordered_along_line</i>	page 137
<i>CGAL::collinear_are_ordered_along_line</i>	page 143
<i>CGAL::collinear_are_strictly_ordered_along_line</i>	page 144

CGAL::are_strictly_ordered_along_line

bool *are_strictly_ordered_along_line*(*Point_2<Kernel>* *p*,
Point_2<Kernel> *q*,
Point_2<Kernel> *r*)

returns *true*, iff the three points are collinear and *q* lies strictly between *p* and *r*. Note that *false* is returned, if $q==p$ or $q==r$.

bool *are_strictly_ordered_along_line*(*Point_3<Kernel>* *p*,
Point_3<Kernel> *q*,
Point_3<Kernel> *r*)

returns *true*, iff the three points are collinear and *q* lies strictly between *p* and *r*. Note that *false* is returned, if $q==p$ or $q==r$.

See Also

CGAL::are_ordered_along_line page [136](#)
CGAL::collinear_are_ordered_along_line page [143](#)
CGAL::collinear_are_strictly_ordered_along_line page [144](#)

CGAL::area*Kernel::FT**area(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r)*

returns the signed area of the triangle defined by the points p , q and r .

CGAL::bisector

Line_2<Kernel>

bisector(Point_2<Kernel> p, Point_2<Kernel> q)

constructs the bisector line of the two points p and q . The bisector is oriented in such a way that p lies on its positive side.

Precondition: p and q are not equal.

Line_2<Kernel>

bisector(Line_2<Kernel> l1, Line_2<Kernel> l2)

constructs the bisector of the two lines $l1$ and $l2$. In the general case, the bisector has the direction of the vector which is the sum of the normalized directions of the two lines, and which passes through the intersection of $l1$ and $l2$. If $l1$ and $l2$ are parallel, then the bisector is defined as the line which has the same direction as $l1$, and which is at the same distance from $l1$ and $l2$. This function requires that *Kernel::RT* supports the *sqrt()* operation.

Plane_3<Kernel>

bisector(Point_3<Kernel> p, Point_3<Kernel> q)

constructs the bisector plane of the two points p and q . The bisector is oriented in such a way that p lies on its positive side.

Precondition: p and q are not equal.

Plane_3<Kernel>

bisector(Plane_3<Kernel> h1, Plane_3<Kernel> h2)

constructs the bisector of the two planes $h1$ and $h2$. In the general case, the bisector has a normal vector which has the same direction as the sum of the normalized normal vectors of the two planes, and passes through the intersection of $h1$ and $h2$. If $h1$ and $h2$ are parallel, then the bisector is defined as the plane which has the same oriented normal vector as $h1$, and which is at the same distance from $h1$ and $h2$. This function requires that *Kernel::RT* supports the *sqrt()* operation.

CGAL::centroid

Point_2<Kernel> *centroid(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r)*
 compute the centroid of the points p , q , and r .

Point_2<Kernel> *centroid(Point_2<Kernel> p,*
 Point_2<Kernel> q,
 Point_2<Kernel> r,
 Point_2<Kernel> s)
 compute the centroid of the points p , q , r , and s .

Point_2<Kernel> *centroid(Triangle_2<Kernel> t)*
 compute the centroid of the triangle t .

Point_3<Kernel> *centroid(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r)*
 compute the centroid of the points p , q , and r .

Point_3<Kernel> *centroid(Point_3<Kernel> p,*
 Point_3<Kernel> q,
 Point_3<Kernel> r,
 Point_3<Kernel> s)
 compute the centroid of the points p , q , r , and s .

Point_3<Kernel> *centroid(Triangle_3<Kernel> t)*
 compute the centroid of the triangle t .

Point_3<Kernel> *centroid(Tetrahedron_3<Kernel> t)*
 compute the centroid of the tetrahedron t .

CGAL::circumcenter

<i>Point_2<Kernel></i>	<i>circumcenter(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r)</i>	<p>compute the center of the circle passing through the points p, q, and r.</p> <p><i>Precondition:</i> p, q, and r are not collinear.</p>
<i>Point_2<Kernel></i>	<i>circumcenter(Triangle_2<Kernel> t)</i>	<p>compute the center of the circle passing through the vertices of t.</p> <p><i>Precondition:</i> t is not degenerate.</p>
<i>Point_3<Kernel></i>	<i>circumcenter(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r)</i>	<p>compute the center of the circle passing through the points p, q, and r.</p> <p><i>Precondition:</i> p, q, and r are not collinear.</p>
<i>Point_3<Kernel></i>	<i>circumcenter(Triangle_3<Kernel> t)</i>	<p>compute the center of the circle passing through the vertices of t.</p> <p><i>Precondition:</i> t is not degenerate.</p>
<i>Point_3<Kernel></i>	<i>circumcenter(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r, Point_3<Kernel> s)</i>	<p>compute the center of the sphere passing through the points p, q, r, and s.</p> <p><i>Precondition:</i> p, q, r, and s are not coplanar.</p>
<i>Point_3<Kernel></i>	<i>circumcenter(Tetrahedron_3<Kernel> t)</i>	<p>compute the center of the sphere passing through the vertices of t.</p> <p><i>Precondition:</i> t is not degenerate.</p>

CGAL::collinear

bool *collinear*(*Point_2*<*Kernel*> *p*, *Point_2*<*Kernel*> *q*, *Point_2*<*Kernel*> *r*)

returns *true*, iff *p*, *q*, and *r* are collinear.

bool *collinear*(*Point_3*<*Kernel*> *p*, *Point_3*<*Kernel*> *q*, *Point_3*<*Kernel*> *r*)

returns *true*, iff *p*, *q*, and *r* are collinear.

See Also

CGAL::left_turn page [177](#)
CGAL::orientation page [494](#)
CGAL::right_turn page [196](#)

CGAL::collinear_are_ordered_along_line

bool *collinear_are_ordered_along_line*(*Point_2<Kernel>* *p*,
Point_2<Kernel> *q*,
Point_2<Kernel> *r*)

returns *true*, iff *q* lies between *p* and *r*.
Precondition: *p*, *q* and *r* are collinear.

bool *collinear_are_ordered_along_line*(*Point_3<Kernel>* *p*,
Point_3<Kernel> *q*,
Point_3<Kernel> *r*)

returns *true*, iff *q* lies between *p* and *r*.
Precondition: *p*, *q* and *r* are collinear.

See Also

CGAL::are_ordered_along_line page [136](#)
CGAL::are_strictly_ordered_along_line page [137](#)
CGAL::collinear_are_strictly_ordered_along_line page [144](#)

bool *collinear_are_strictly_ordered_along_line*(*Point_2<Kernel>* *p*,
 Point_2<Kernel> *q*,
 Point_2<Kernel> *r*)

returns *true*, iff *q* lies strictly between *p* and *r*.
Precondition: *p*, *q* and *r* are collinear.

bool *collinear_are_strictly_ordered_along_line*(*Point_3<Kernel>* *p*,
 Point_3<Kernel> *q*,
 Point_3<Kernel> *r*)

returns *true*, iff *q* lies strictly between *p* and *r*.
Precondition: *p*, *q* and *r* are collinear.

See Also

<i>CGAL::are_ordered_along_line</i>	page 136
<i>CGAL::are_strictly_ordered_along_line</i>	page 137
<i>CGAL::collinear_are_ordered_along_line</i>	page 143

CGAL::compare_slopes

<i>Comparison_result</i>	<i>compare_slopes(Line_2<Kernel> l1, Line_2<Kernel> l2)</i> compares the slopes of the lines <i>l1</i> and <i>l2</i>
<i>Comparison_result</i>	<i>compare_slopes(Segment_2<Kernel> s1, Segment_2<Kernel> s2)</i> compares the slopes of the segments <i>s1</i> and <i>s2</i>

CGAL::compare_x

Comparison_result *compare_x(Point_2<Kernel> p, Point_2<Kernel> q)*
 compares the x -coordinates of p and q .

Comparison_result *compare_x(Point_3<Kernel> p, Point_3<Kernel> q)*
 compares the x -coordinates of p and q .

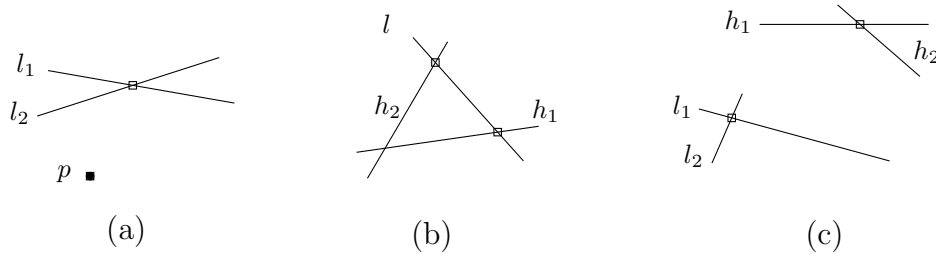


Figure 2.1: Comparison of the x or y -coordinates of the (implicitly given) points in the boxes.

Comparison_result *compare_x(Point_2<Kernel> p, Line_2<Kernel> l1, Line_2<Kernel> l2)*
 compares the x -coordinates of p and the intersection of lines $l1$ and $l2$ (Figure 2.1 (a)).

Comparison_result *compare_x(Line_2<Kernel> l, Line_2<Kernel> h1, Line_2<Kernel> h2)*
 compares the x -coordinates of the intersection of line l with line $h1$ and with line $h2$ (Figure 2.1 (b)).

Comparison_result *compare_x(Line_2<Kernel> l1,
 Line_2<Kernel> l2,
 Line_2<Kernel> h1,
 Line_2<Kernel> h2)*
 compares the x -coordinates of the intersection of lines $l1$ and $l2$ and the intersection of lines $h1$ and $h2$ (Figure 2.1 (c)).

See Also

CGAL::compare_xy page 151
CGAL::compare_xyz page 152
CGAL::compare_x_at_y page 153
CGAL::compare_y page 155
CGAL::compare_yx page 159

<i>CGAL::compare_y_at_x</i>	page 157
<i>CGAL::compare_z</i>	page 160

CGAL::compare_xy

Comparison_result *compare_xy*(*Point_2*<*Kernel*> *p*, *Point_2*<*Kernel*> *q*)

Compares the Cartesian coordinates of points *p* and *q* lexicographically in *xy* order: first *x*-coordinates are compared, if they are equal, *y*-coordinates are compared.

Comparison_result *compare_xy*(*Point_3*<*Kernel*> *p*, *Point_3*<*Kernel*> *q*)

Compares the Cartesian coordinates of points *p* and *q* lexicographically in *xy* order: first *x*-coordinates are compared, if they are equal, *y*-coordinates are compared.

See Also

CGAL::compare_xyz page 152
CGAL::compare_x page 149
CGAL::compare_x_at_y page 153
CGAL::compare_y page 155
CGAL::compare_yx page 159
CGAL::compare_y_at_x page 157
CGAL::compare_z page 160

CGAL::compare_xyz

Comparison_result *compare_xyz(Point_3<Kernel> p, Point_3<Kernel> q)*

Compares the Cartesian coordinates of points p and q lexicographically in xyz order: first x -coordinates are compared, if they are equal, y -coordinates are compared, and if both x - and y - coordinate are equal, z -coordinates are compared.

See Also

CGAL::compare_xy page [151](#)
CGAL::compare_x page [149](#)
CGAL::compare_x_at_y page [153](#)
CGAL::compare_y page [155](#)
CGAL::compare_yx page [159](#)
CGAL::compare_y_at_x page [157](#)
CGAL::compare_z page [160](#)

CGAL::compare_x_at_y

Comparison_result

`compare_x_at_y(Point_2<Kernel> p, Line_2<Kernel> h)`

compares the x -coordinates of p and the horizontal projection of p on h (Figure 2.2 (a)).

Precondition: h is not horizontal.

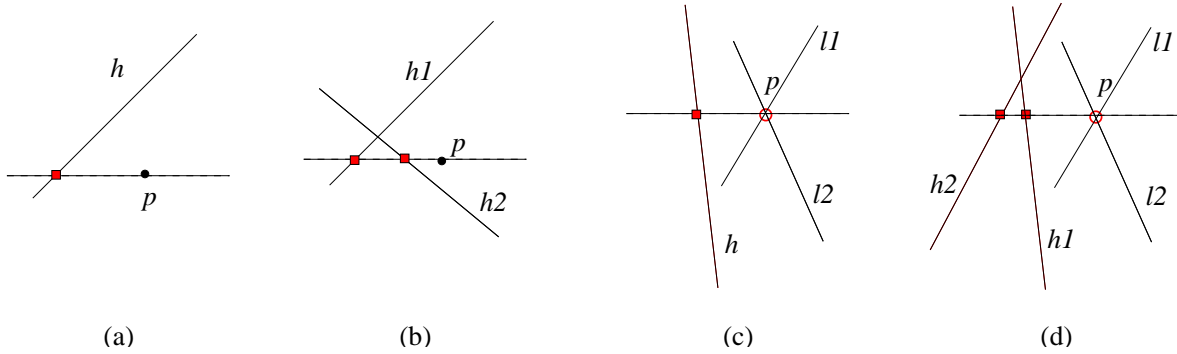


Figure 2.2: Comparison of the x -coordinates of the (implicitly given) points in the boxes, at a y -coordinate. The y -coordinate is either given explicitly (disc) or implicitly (circle).

Comparison_result

`compare_x_at_y(Point_2<Kernel> p, Line_2<Kernel> h1, Line_2<Kernel> h2)`

This function compares the x -coordinates of the horizontal projection of p on $h1$ and on $h2$ (Figure 2.2 (b)).

Precondition: $h1$ and $h2$ are not horizontal.

Comparison_result

`compare_x_at_y(Line_2<Kernel> l1, Line_2<Kernel> l2, Line_2<Kernel> h)`

Let p be the intersection of lines $l1$ and $l2$. This function compares the x -coordinates of p and the horizontal projection of p on h (Figure 2.2 (c)).

Precondition: $l1$ and $l2$ intersect and are not horizontal; h is not horizontal.

Comparison_result

`compare_x_at_y(Line_2<Kernel> l1,
Line_2<Kernel> l2,
Line_2<Kernel> h1,
Line_2<Kernel> h2)`

Let p be the intersection of lines $l1$ and $l2$. This function compares the x -coordinates of the horizontal projection of p on $h1$ and on $h2$ (Figure 2.2 (d)).

Precondition: $l1$ and $l2$ intersect and are not horizontal; $h1$ and $h2$ are not horizontal.

See Also

<i>CGAL::compare_xy</i>	page 151
<i>CGAL::compare_xyz</i>	page 152
<i>CGAL::compare_x</i>	page 149
<i>CGAL::compare_y</i>	page 155
<i>CGAL::compare_yx</i>	page 159
<i>CGAL::compare_y_at_x</i>	page 157
<i>CGAL::compare_z</i>	page 160

CGAL::compare_y

Comparison_result *compare_y*(*Point_2<Kernel>* *p*, *Point_2<Kernel>* *q*)
 compares Cartesian y-coordinates of *p* and *q*.

Comparison_result *compare_y*(*Point_3<Kernel>* *p*, *Point_3<Kernel>* *q*)
 compares Cartesian y-coordinates of *p* and *q*.

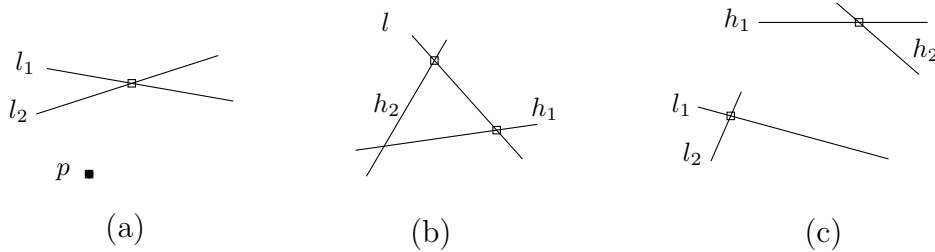


Figure 2.3: Comparison of the x or y-coordinates of the (implicitly given) points in the boxes.

Comparison_result *compare_y*(*Point_2<Kernel>* *p*, *Line_2<Kernel>* *l1*, *Line_2<Kernel>* *l2*)
 compares the y-coordinates of *p* and the intersection of lines *l1* and *l2* (Figure 2.3 (a)).

Comparison_result *compare_y*(*Line_2<Kernel>* *l*, *Line_2<Kernel>* *h1*, *Line_2<Kernel>* *h2*)
 compares the y-coordinates of the intersection of line *l* with line *h1* and with line *h2* (Figure 2.3 (b)).

Comparison_result *compare_y*(*Line_2<Kernel>* *l1*,
 Line_2<Kernel> *l2*,
 Line_2<Kernel> *h1*,
 Line_2<Kernel> *h2*)
 compares the y-coordinates of the intersection of lines *l1* and *l2* and the intersection of lines *h1* and *h2* (Figure 2.3 (c)).

See Also

CGAL::compare_xy page 151
CGAL::compare_xyz page 152
CGAL::compare_x page 149
CGAL::compare_x_at_y page 153
CGAL::compare_yx page 159

<i>CGAL::compare_y_at_x</i>	page 157
<i>CGAL::compare_z</i>	page 160

CGAL::compare_y_at_x

Comparison_result

`compare_y_at_x(Point_2<Kernel> p, Line_2<Kernel> h)`

compares the y-coordinates of p and the vertical projection of p on h (Figure 2.4 (d)).

Precondition: h is not vertical.

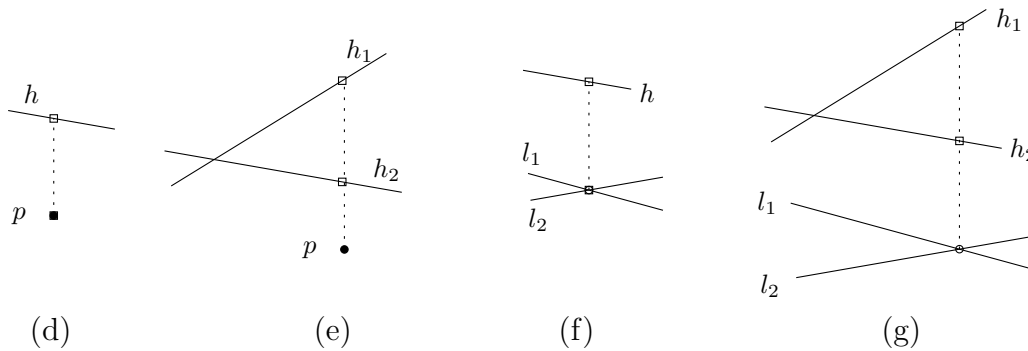


Figure 2.4: Comparison of the y-coordinates of the (implicitly given) points in the boxes, at an x -coordinate. The x -coordinate is either given explicitly (disc) or implicitly (circle).

Comparison_result

`compare_y_at_x(Point_2<Kernel> p, Line_2<Kernel> h1, Line_2<Kernel> h2)`

compares the y-coordinates of the vertical projection of p on $h1$ and on $h2$ (Figure 2.4 (e)).

Precondition: $h1$ and $h2$ are not vertical.

Comparison_result

`compare_y_at_x(Line_2<Kernel> l1, Line_2<Kernel> l2, Line_2<Kernel> h)`

Let p be the intersection of lines $l1$ and $l2$. This function compares the y-coordinates of p and the vertical projection of p on h (Figure 2.4 (f)).

Precondition: $l1, l2$ intersect and h is not vertical.

Comparison_result

`compare_y_at_x(Line_2<Kernel> l1,
Line_2<Kernel> l2,
Line_2<Kernel> h1,
Line_2<Kernel> h2)`

Let p be the intersection of lines $l1$ and $l2$. This function compares the y-coordinates of the vertical projection of p on $h1$ and on $h2$ (Figure 2.4 (g)).

Precondition: $l1$ and $l2$ intersect; $h1$ and $h2$ are not vertical.

Comparison_result *compare_y_at_x*(*Point_2*<*Kernel*> *p*, *Segment_2*<*Kernel*> *s*)

compares the y-coordinates of *p* and the vertical projection of *p* on *s*. If *s* is vertical, then return *EQUAL* when *p* lies on *s*, *SMALLER* when *p* lies under *s*, and *LARGER* otherwise.

Precondition: *p* is within the x range of *s*.

Comparison_result *compare_y_at_x*(*Point_2*<*Kernel*> *p*,
 Segment_2<*Kernel*> *s1*,
 Segment_2<*Kernel*> *s2*)

compares the y-coordinates of the vertical projection of *p* on *s1* and on *s2*. If *s1* or *s2* is vertical, then return *EQUAL* if they intersect, otherwise return *SMALLER* if *s1* lies below *s2*, and return *LARGER* otherwise.

Precondition: *p* is within the x range of *s1* and *s2*.

See Also

CGAL::compare_xy page 151

CGAL::compare_xyz page 152

CGAL::compare_x page 149

CGAL::compare_x_at_y page 153

CGAL::compare_y page 155

CGAL::compare_yx page 159

CGAL::compare_z page 160

CGAL::compare_yx

Comparison_result *compare_yx*(*Point_2<Kernel> p*, *Point_2<Kernel> q*)

Compares the Cartesian coordinates of points p and q lexicographically in yx order: first y -coordinates are compared, if they are equal, x -coordinates are compared.

See Also

CGAL::compare_xy page [151](#)
CGAL::compare_xyz page [152](#)
CGAL::compare_x page [149](#)
CGAL::compare_x_at_y page [153](#)
CGAL::compare_y page [155](#)
CGAL::compare_y_at_x page [157](#)
CGAL::compare_z page [160](#)

CGAL::compare_z

Comparison_result *compare_z(Point_3<Kernel> p, Point_3<Kernel> q)*

compares the z -coordinates of p and q .

See Also

CGAL::compare_xy page [151](#)
CGAL::compare_xyz page [152](#)
CGAL::compare_x page [149](#)
CGAL::compare_x_at_y page [153](#)
CGAL::compare_y page [155](#)
CGAL::compare_yx page [159](#)
CGAL::compare_y_at_x page [157](#)

CGAL::coplanar

bool *coplanar*(*Point_3*<*Kernel*> *p*,
 Point_3<*Kernel*> *q*,
 Point_3<*Kernel*> *r*,
 Point_3<*Kernel*> *s*)

returns *true*, if *p*, *q*, *r*, and *s* are coplanar.

See Also

CGAL::coplanar_orientation page [162](#)
CGAL::coplanar_side_of_bounded_circle page [163](#)

Orientation

```
coplanar_orientation( Point_3<Kernel> p,
                      Point_3<Kernel> q,
                      Point_3<Kernel> r,
                      Point_3<Kernel> s)
```

Let P be the plane defined by the points p , q , and r . Note that the order defines the orientation of P . The function computes the orientation of points p , q , and s in P : If p , q , s are collinear, *COLLINEAR* is returned. If P and the plane defined by p , q , and s have the same orientation, *POSITIVE* is returned; otherwise *NEGATIVE* is returned. *Precondition*: p , q , r , and s are coplanar and p , q , and r are not collinear.

Orientation

```
coplanar_orientation( Point_3<Kernel> p,
                      Point_3<Kernel> q,
                      Point_3<Kernel> r)
```

If p, q, r are collinear, then *COLLINEAR* is returned. If not, then p, q, r define a plane P . The return value in this case is either *POSITIVE* or *NEGATIVE*, but we don't specify it explicitly. However, we guarantee that all calls to this predicate over 3 points in P will return a coherent orientation if considered a 2D orientation in P .

See Also

<i>CGAL::coplanar</i>	page 161
<i>CGAL::coplanar_side_of_bounded_circle</i>	page 163
<i>CGAL::orientation</i>	page 494

CGAL::coplanar_side_of_bounded_circle

Bounded_side

```
coplanar_side_of_bounded_circle( Point_3<Kernel> p,
                                  Point_3<Kernel> q,
                                  Point_3<Kernel> r,
                                  Point_3<Kernel> s)
```

returns the bounded side of the circle defined by p , q , and r on which s lies.

Precondition: p, q, r , and s are coplanar and p, q , and r are not collinear.

See Also

CGAL::coplanar_orientation.....page 162

CGAL::side_of_bounded_circle.....page 197

CGAL::cross_product

Vector_3<Kernel> *cross_product*(*Vector_3<Kernel>* *u*, *Vector_3<Kernel>* *v*)
returns the cross product of *u* and *v*.

CGAL::do_intersect

```
#include <CGAL/intersections.h>
```

```
bool do_intersect( Type1<Kernel> obj1, Type2<Kernel> obj2)
```

checks whether *obj1* and *obj2* intersect. Two objects *obj1* and *obj2* intersect if there is a point *p* that is part of both *obj1* and *obj2*. The intersection region of those two objects is defined as the set of all points *p* that are part of both *obj1* and *obj2*. Note that for objects like triangles and polygons that enclose a bounded region, this region is part of the object.

The types *Type1* and *Type2* can be any of the following:

- *Point_2<Kernel>*
- *Line_2<Kernel>*
- *Ray_2<Kernel>*
- *Segment_2<Kernel>*
- *Triangle_2<Kernel>*
- *Iso_rectangle_2<Kernel>*

Also, in three-dimensional space *Type1* can be *Plane_3<Kernel>* or *Triangle_3<Kernel>* and *Type2* any of

- *Plane_3<Kernel>*
- *Line_3<Kernel>*
- *Ray_3<Kernel>*
- *Segment_3<Kernel>*
- *Triangle_3<Kernel>*

Finally, *Type1* can be of type *Triangle_3<Kernel>* and *Type2* of type *Tetrahedron_3<Kernel>*.

See Also

CGAL::intersection page [945](#)

CGAL::do_overlap

```
#include <CGAL/Bbox_2.h>
```

```
bool do_overlap( Bbox_2 bb1, Bbox_2 bb2)
```

returns *true* iff *bb1* and *bb2* overlap, i.e., iff their intersection is non-empty.

```
#include <CGAL/Bbox_3.h>
```

```
bool do_overlap( Bbox_3 bb1, Bbox_3 bb2)
```

returns *true* iff *bb1* and *bb2* overlap, i.e., iff their intersection is non-empty.

CGAL::enum_cast

```
#include <CGAL/functions_on_enums.h>
```

```
template < typename T, typename U >
```

```
T          enum_cast( U u)
```

converts between the various enums provided by the CGAL kernel. The conversion preserves the order of the values.

See Also

CGAL::Sign page [124](#)
CGAL::Comparison_result page [124](#)
CGAL::Orientation page [125](#)
CGAL::Oriented_side page [125](#)
CGAL::Bounded_side page [123](#)
CGAL::Angle page [123](#)
CGAL::Uncertain<T> page ??

CGAL::has_larger_distance_to_point

```
bool has_larger_distance_to_point( Point_2<Kernel> p,
                                   Point_2<Kernel> q,
                                   Point_2<Kernel> r)
```

returns *true* iff the distance between q and p is larger than the distance between r and p .

```

bool has_larger_distance_to_point( Point_3<Kernel> p,
                                   Point_3<Kernel> q,
                                   Point_3<Kernel> r)

```

returns *true* iff the distance between q and p is larger than the distance between r and p .

See Also

<i>CGAL::compare_distance_to_point</i>	page 145
<i>CGAL::compare_signed_distance_to_line</i>	page 146
<i>CGAL::compare_signed_distance_to_plane</i>	page 147
<i>CGAL::has_larger_signed_distance_to_line</i>	page 169
<i>CGAL::has_larger_signed_distance_to_plane</i>	page 170
<i>CGAL::has_smaller_distance_to_point</i>	page 171
<i>CGAL::has_smaller_signed_distance_to_line</i>	page 172
<i>CGAL::has_smaller_signed_distance_to_plane</i>	page 173

CGAL::has_larger_signed_distance_to_line

bool *has_larger_signed_distance_to_line*(*Line_2*<*Kernel*> *l*,
Point_2<*Kernel*> *p*,
Point_2<*Kernel*> *q*)

returns *true* iff the signed distance of *p* and *l* is larger than the signed distance of *q* and *l*.

bool *has_larger_signed_distance_to_line*(*Point_2*<*Kernel*> *p*,
Point_2<*Kernel*> *q*,
Point_2<*Kernel*> *r*,
Point_2<*Kernel*> *s*)

returns *true* iff the signed distance of *r* and *l* is larger than the signed distance of *s* and *l*, where *l* is the directed line through points *p* and *q*.

See Also

CGAL::compare_distance_to_point page 145
CGAL::compare_signed_distance_to_line page 146
CGAL::compare_signed_distance_to_plane page 147
CGAL::has_larger_distance_to_point page 168
CGAL::has_larger_signed_distance_to_line page 169
CGAL::has_larger_signed_distance_to_plane page 170
CGAL::has_smaller_distance_to_point page 171
CGAL::has_smaller_signed_distance_to_line page 172
CGAL::has_smaller_signed_distance_to_plane page 173

CGAL::has_larger_signed_distance_to_plane

bool *has_larger_signed_distance_to_plane*(*Plane_3*<*Kernel*> *h*,
Point_3<*Kernel*> *p*,
Point_3<*Kernel*> *q*)

returns *true* iff the signed distance of *p* and *h* is larger than the signed distance of *q* and *h*.

bool *has_larger_signed_distance_to_plane*(*Point_3*<*Kernel*> *p*,
Point_3<*Kernel*> *q*,
Point_3<*Kernel*> *r*,
Point_3<*Kernel*> *s*,
Point_3<*Kernel*> *t*)

returns *true* iff the signed distance of *s* and *h* is larger than the signed distance of *t* and *h*, where *h* is the oriented plane through *p*, *q* and *r*.

See Also

CGAL::compare_distance_to_point page [145](#)
CGAL::compare_signed_distance_to_line page [146](#)
CGAL::compare_signed_distance_to_plane page [147](#)
CGAL::has_larger_distance_to_point page [168](#)
CGAL::has_larger_signed_distance_to_line page [169](#)
CGAL::has_smaller_distance_to_point page [171](#)
CGAL::has_smaller_signed_distance_to_line page [172](#)
CGAL::has_smaller_signed_distance_to_plane page [173](#)

CGAL::has_smaller_distance_to_point

bool *has_smaller_distance_to_point*(*Point_2<Kernel>* *p*,
Point_2<Kernel> *q*,
Point_2<Kernel> *r*)

returns *true* iff the distance between *q* and *p* is smaller than the distance between *r* and *p*.

bool *has_smaller_distance_to_point*(*Point_3<Kernel>* *p*,
Point_3<Kernel> *q*,
Point_3<Kernel> *r*)

returns *true* iff the distance between *q* and *p* is smaller than the distance between *r* and *p*.

See Also

CGAL::compare_distance_to_point page 145
CGAL::compare_signed_distance_to_line page 146
CGAL::compare_signed_distance_to_plane page 147
CGAL::has_larger_distance_to_point page 168
CGAL::has_larger_signed_distance_to_line page 169
CGAL::has_larger_signed_distance_to_plane page 170
CGAL::has_smaller_signed_distance_to_line page 172
CGAL::has_smaller_signed_distance_to_plane page 173

CGAL::has_smaller_signed_distance_to_line

bool *has_smaller_signed_distance_to_line*(*Line_2*<*Kernel*> *l*,
Point_2<*Kernel*> *p*,
Point_2<*Kernel*> *q*)

returns *true* iff the signed distance of *p* and *l* is smaller than the signed distance of *q* and *l*.

bool *has_smaller_signed_distance_to_line*(*Point_2*<*Kernel*> *p*,
Point_2<*Kernel*> *q*,
Point_2<*Kernel*> *r*,
Point_2<*Kernel*> *s*)

returns *true* iff the signed distance of *r* and *l* is smaller than the signed distance of *s* and *l*, where *l* is the oriented line through *p* and *q*.

See Also

CGAL::compare_distance_to_point page 145
CGAL::compare_signed_distance_to_line page 146
CGAL::compare_signed_distance_to_plane page 147
CGAL::has_larger_distance_to_point page 168
CGAL::has_larger_signed_distance_to_line page 169
CGAL::has_larger_signed_distance_to_plane page 170
CGAL::has_smaller_distance_to_point page 171
CGAL::has_smaller_signed_distance_to_plane page 173

CGAL::has_smaller_signed_distance_to_plane

bool *has_smaller_signed_distance_to_plane*(*Plane_3*<*Kernel*> *h*,
Point_3<*Kernel*> *p*,
Point_3<*Kernel*> *q*)

returns *true* iff the signed distance of *p* and *h* is smaller than the signed distance of *q* and *h*.

bool *has_smaller_signed_distance_to_plane*(*Point_3*<*Kernel*> *p*,
Point_3<*Kernel*> *q*,
Point_3<*Kernel*> *r*,
Point_3<*Kernel*> *s*,
Point_3<*Kernel*> *t*)

returns *true* iff the signed distance of *p* and *h* is smaller than the signed distance of *q* and *h*, where *h* is the oriented plane through *p*, *q* and *r*.

See Also

CGAL::compare_distance_to_point page 145
CGAL::compare_signed_distance_to_line page 146
CGAL::compare_signed_distance_to_plane page 147
CGAL::has_larger_distance_to_point page 168
CGAL::has_larger_signed_distance_to_line page 169
CGAL::has_larger_signed_distance_to_plane page 170
CGAL::has_smaller_distance_to_point page 171
CGAL::has_smaller_signed_distance_to_line page 172

CGAL::intersection

```
#include <CGAL/intersections.h>
```

```
Object intersection( Type1<Kernel> obj1, Type2<Kernel> obj2)
```

Two objects *obj1* and *obj2* intersect if there is a point *p* that is part of both *obj1* and *obj2*. The intersection region of those two objects is defined as the set of all points *p* that are part of both *obj1* and *obj2*. Note that for objects like triangles and polygons that enclose a bounded region, this region is considered part of the object. If a segment lies completely inside a triangle, then those two objects intersect and the intersection region is the complete segment.

The possible value for types *Type1* and *Type2* and the possible return values wrapped in *Object* are the following:

type A	type B	return type
<i>Line_2</i>	<i>Line_2</i>	<i>Point_2</i> <i>Line_2</i>
<i>Segment_2</i>	<i>Line_2</i>	<i>Point_2</i> <i>Segment_2</i>
<i>Segment_2</i>	<i>Segment_2</i>	<i>Point_2</i> <i>Segment_2</i>
<i>Ray_2</i>	<i>Line_2</i>	<i>Point_2</i> <i>Ray_2</i>
<i>Ray_2</i>	<i>Segment_2</i>	<i>Point_2</i> <i>Segment_2</i>
<i>Ray_2</i>	<i>Ray_2</i>	<i>Point_2</i> <i>Segment_2</i> <i>Ray_2</i>
<i>Triangle_2</i>	<i>Line_2</i>	<i>Point_2</i> <i>Segment_2</i>
<i>Triangle_2</i>	<i>Segment_2</i>	<i>Point_2</i> <i>Segment_2</i>
<i>Triangle_2</i>	<i>Ray_2</i>	<i>Point_2</i> <i>Segment_2</i>
<i>Triangle_2</i>	<i>Triangle_2</i>	<i>Point_2</i> <i>Segment_2</i> <i>Triangle_2</i> <i>std::vector<Point_2></i>
<i>Iso_rectangle_2</i>	<i>Line_2</i>	<i>Point_2</i> <i>Segment_2</i>
<i>Iso_rectangle_2</i>	<i>Segment_2</i>	<i>Point_2</i> <i>Segment_2</i>
<i>Iso_rectangle_2</i>	<i>Ray_2</i>	<i>Point_2</i> <i>Segment_2</i>

continued

<i>Iso_rectangle_2</i>	<i>Iso_rectangle_2</i>	<i>Iso_rectangle_2</i>
<i>Plane_3</i>	<i>Line_3</i>	<i>Point_3</i> <i>Line_3</i>
<i>Plane_3</i>	<i>Ray_3</i>	<i>Point_3</i> <i>Ray_3</i>
<i>Plane_3</i>	<i>Segment_3</i>	<i>Point_3</i> <i>Segment_3</i>
<i>Plane_3</i>	<i>Plane_3</i>	<i>Line_3</i> <i>Plane_3</i>

There is also an intersection function between 3 planes.

Object *intersection(Plane_3<Kernel> pl1, Plane_3<Kernel> pl2, Plane_3<Kernel> pl3)*

returns the intersection of 3 planes, which can be either a point, a line, a plane, or empty.

Example

The following example demonstrates the most common use of *intersection* routines.

```
#include <CGAL/intersections.h>

void foo(CGAL::Segment_2<Kernel> seg, CGAL::Line_2<Kernel> line)
{
    CGAL::Object result;
    CGAL::Point_2<Kernel> ipoint;
    CGAL::Segment_2<Kernel> iseg;

    result = CGAL::intersection(seg, line);
    if (CGAL::assign(ipoint, result)) {

        // handle the point intersection case.

    } else
        if (CGAL::assign(iseg, result)) {

            // handle the segment intersection case.

        } else {

            // handle the no intersection case.

        }
}
```

See Also

CGAL::assign page ??
CGAL::do_intersect page [943](#)
CGAL::Object page [120](#)

CGAL::left_turn

bool *left_turn*(*Point_2*<*Kernel*> *p*, *Point_2*<*Kernel*> *q*, *Point_2*<*Kernel*> *r*)
returns *true* iff *p*, *q*, and *r* form a left turn.

See Also

CGAL::collinearpage [142](#)
CGAL::orientationpage [494](#)
CGAL::right_turnpage [196](#)

CGAL::lexicographically_xyz_smaller

bool *lexicographically_xyz_smaller*(*Point_3*<*Kernel*> *p*, *Point_3*<*Kernel*> *q*)

returns *true* iff *p* is lexicographically smaller than *q* with respect to *xyz* order.

See Also

CGAL::compare_xyz page [152](#)
CGAL::lexicographically_xyz_smaller_or_equal page [179](#)

CGAL::lexicographically_xyz_smaller_or_equal

bool *lexicographically_xyz_smaller_or_equal*(*Point_3*<*Kernel*> *p*, *Point_3*<*Kernel*> *q*)

returns *true* iff *p* is lexicographically not larger than *q* with respect to *xyz* order.

See Also

CGAL::compare_xyz page [152](#)
CGAL::lexicographically_xyz_smaller page [178](#)

CGAL::lexicographically_xy_larger

bool

lexicographically_xy_larger(Point_2<Kernel> p, Point_2<Kernel> q)

returns *true* iff *p* is lexicographically larger than *q* with respect to *xy* order.

See Also

CGAL::compare_xy page [151](#)

CGAL::lexicographically_xy_larger_or_equal page [181](#)

CGAL::lexicographically_xy_smaller page [182](#)

CGAL::lexicographically_xy_smaller_or_equal page [183](#)

CGAL::lexicographically_xy_larger_or_equal

bool *lexicographically_xy_larger_or_equal(Point_2<Kernel> p, Point_2<Kernel> q)*

returns *true* iff *p* is lexicographically not smaller than *q* with respect to *xy* order.

See Also

CGAL::compare_xy page [151](#)
CGAL::lexicographically_xy_larger page [180](#)
CGAL::lexicographically_xy_smaller page [182](#)
CGAL::lexicographically_xy_smaller_or_equal page [183](#)

CGAL::lexicographically_xy_smaller

bool

lexicographically_xy_smaller(Point_2<Kernel> p, Point_2<Kernel> q)

returns *true* iff *p* is lexicographically smaller than *q* with respect to *xy* order.

See Also

CGAL::compare_xy page [151](#)
CGAL::lexicographically_xy_larger page [180](#)
CGAL::lexicographically_xy_larger_or_equal page [181](#)
CGAL::lexicographically_xy_smaller_or_equal page [183](#)

CGAL::lexicographically_xy_smaller_or_equal

bool *lexicographically_xy_smaller_or_equal*(*Point_2*<*Kernel*> *p*, *Point_2*<*Kernel*> *q*)

returns *true* iff *p* is lexicographically not larger than *q* with respect to *xy* order.

See Also

CGAL::compare_xy page [151](#)
CGAL::lexicographically_xy_larger page [180](#)
CGAL::lexicographically_xy_larger_or_equal page [181](#)
CGAL::lexicographically_xy_smaller page [182](#)

CGAL::max_vertex

Point_2<Kernel> *max_vertex(Iso_box_2<Kernel> ir)*

computes the vertex with the lexicographically largest coordinates of the iso rectangle *ir*.

Point_3<Kernel> *max_vertex(Iso_cuboid_3<Kernel> ic)*

computes the vertex with the lexicographically largest coordinates of the iso cuboid *ic*.

CGAL::midpoint

Point_2<Kernel> *midpoint(Point_2<Kernel> p, Point_2<Kernel> q)*
computes the midpoint of the segment pq .

Point_3<Kernel> *midpoint(Point_3<Kernel> p, Point_3<Kernel> q)*
computes the midpoint of the segment pq .

CGAL::min_vertex

Point_2<Kernel> *min_vertex(Iso_box_2<Kernel> ir)*

computes the vertex with the lexicographically smallest coordinates of the iso rectangle *ir*.

Point_3<Kernel> *min_vertex(Iso_cuboid_3<Kernel> ic)*

computes the vertex with the lexicographically smallest coordinates of the iso cuboid *ic*.

CGAL::operator+

Point_2<Kernel> *operator+(Point_2<Kernel> p, Vector_2<Kernel> v)*

returns the point obtained by translating p by vector v .

Point_3<Kernel> *operator+(Point_3<Kernel> p, Vector_3<Kernel> v)*

returns a point obtained by translating p by vector v .

See Also

CGAL::operator- page [188](#)
*CGAL::operator** page [189](#)

CGAL::operator-

Vector_2<Kernel> *operator-(Point_2<Kernel> p, Point_2<Kernel> q)*

returns the difference vector between q and p . You can substitute *ORIGIN* for either p or q , but not for both.

Point_2<Kernel> *operator-(Point_2<Kernel> p, Vector_2<Kernel> v)*

returns the point obtained by translating p by the vector $-v$.

Vector_3<Kernel> *operator-(Point_3<Kernel> p, Point_3<Kernel> q)*

returns the difference vector between q and p . You can substitute *ORIGIN* for either p or q , but not both.

Point_3<Kernel> *operator-(Point_3<Kernel> p, Vector_3<Kernel> v)*

returns a point obtained by translating p by the vector $-v$.

See Also

CGAL::operator+ page [187](#)
*CGAL::operator** page [189](#)

CGAL::operator*

Vector_2<Kernel> *operator*(Kernel::RT s, Vector_2<Kernel> w)*

Multiplication with a scalar from the left.

Vector_3<Kernel> *operator*(Kernel::RT s, Vector_3<Kernel> w)*

Multiplication with a scalar from the left.

See Also

CGAL::operator+ page [187](#)
CGAL::operator- page [188](#)

CGAL::opposite

```
#include <CGAL/functions_on_enums.h>
```

```
Oriented_side                    opposite( Oriented_side o)
```

returns the opposite side (for example *ON_POSITIVE_SIDE* if *o*==*ON_NEGATIVE_SIDE*), or *ON_ORIENTED_BOUNDARY* if *o*==*ON_ORIENTED_BOUNDARY*.

```
Bounded_side                    opposite( Bounded_side o)
```

returns the opposite side (for example *BOUNDED_SIDE* if *o*==*UNBOUNDED_SIDE*), or returns *ON_BOUNDARY* if *o*==*ON_BOUNDARY*.

CGAL::orthogonal_vector

Vector_3<Kernel> *orthogonal_vector(Plane_3<Kernel> p)*

computes an orthogonal vector of the plane p , which is directed to the positive side of this plane.

Vector_3<Kernel> *orthogonal_vector(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r)*

computes an orthogonal vector of the plane defined by p , q and r , which is directed to the positive side of this plane.

CGAL::orientation

Orientation *orientation*(*Point_2*<*Kernel*> *p*, *Point_2*<*Kernel*> *q*, *Point_2*<*Kernel*> *r*)

returns *LEFT_TURN*, if *r* lies to the left of the oriented line *l* defined by *p* and *q*, returns *RIGHT_TURN* if *r* lies to the right of *l*, and returns *COLLINEAR* if *r* lies on *l*.

Orientation *orientation*(*Point_3*<*Kernel*> *p*,
 Point_3<*Kernel*> *q*,
 Point_3<*Kernel*> *r*,
 Point_3<*Kernel*> *s*)

returns *POSITIVE*, if *s* lies on the positive side of the oriented plane *h* defined by *p*, *q*, and *r*, returns *NEGATIVE* if *s* lies on the negative side of *h*, and returns *COPLANAR* if *s* lies on *h*.

See Also

CGAL::collinear page [142](#) *CGAL::left_turn* page [177](#) *CGAL::right_turn* page [196](#)

CGAL::parallel

<i>bool</i>	<i>parallel(Line_2<Kernel> l1, Line_2<Kernel> l2)</i>	returns <i>true</i> , if <i>l1</i> and <i>l2</i> are parallel or if one of those (or both) is degenerate.
<i>bool</i>	<i>parallel(Ray_2<Kernel> r1, Ray_2<Kernel> r2)</i>	returns <i>true</i> , if <i>r1</i> and <i>r2</i> are parallel or if one of those (or both) is degenerate.
<i>bool</i>	<i>parallel(Segment_2<Kernel> s1, Segment_2<Kernel> s2)</i>	returns <i>true</i> , if <i>s1</i> and <i>s2</i> are parallel or if one of those (or both) is degenerate.
<i>bool</i>	<i>parallel(Line_3<Kernel> l1, Line_3<Kernel> l2)</i>	returns <i>true</i> , if <i>l1</i> and <i>l2</i> are parallel or if one of those (or both) is degenerate.
<i>bool</i>	<i>parallel(Plane_3<Kernel> h1, Plane_3<Kernel> h2)</i>	returns <i>true</i> , if <i>h1</i> and <i>h2</i> are parallel or if one of those (or both) is degenerate.
<i>bool</i>	<i>parallel(Ray_3<Kernel> r1, Ray_3<Kernel> r2)</i>	returns <i>true</i> , if <i>r1</i> and <i>r2</i> are parallel or if one of those (or both) is degenerate.
<i>bool</i>	<i>parallel(Segment_3<Kernel> s1, Segment_3<Kernel> s2)</i>	returns <i>true</i> , if <i>s1</i> and <i>s2</i> are parallel or if one of those (or both) is degenerate.

CGAL::quotient_cartesian_to_homogeneous

```
#include <CGAL/cartesian_homogeneous_conversion.h>
```

```
Point_2< Homogeneous<RT> >
```

```
quotient_cartesian_to_homogeneous( Point_2< Cartesian< Quotient<RT> > > cp)
```

converts 2d point *cp* with Cartesian representation with number type *Quotient<RT>* into a 2d point with homogeneous representation with number type *RT*.

```
Point_3< Homogeneous<RT> >
```

```
quotient_cartesian_to_homogeneous( Point_3< Cartesian< Quotient<RT> > > cp)
```

converts 3d point *cp* with Cartesian representation with number type *Quotient<RT>* into a 3d point with homogeneous representation with number type *RT*.

See Also

<i>CGAL::Cartesian<FieldNumberType></i>	page 41
<i>CGAL::Cartesian_converter<K1, K2, NTConverter></i>	page 42
<i>CGAL::cartesian_to_homogeneous</i>	page 43
<i>CGAL::Homogeneous<RingNumberType></i>	page 48
<i>CGAL::Homogeneous_converter<K1, K2, RTConverter, FTConverter></i>	page 49
<i>CGAL::homogeneous_to_cartesian</i>	page 50
<i>CGAL::homogeneous_to_quotient_cartesian</i>	page 51
<i>CGAL::Simple_cartesian<FieldNumberType></i>	page 55
<i>CGAL::Simple_homogeneous<RingNumberType></i>	page 56

CGAL::rational_rotation_approximation

```
#include <CGAL/rational_rotation.h>
```

```
template <RingNumberType>
void rational_rotation_approximation( RingNumberType dirx,
                                     RingNumberType diry,
                                     RingNumberType & sin_num,
                                     RingNumberType & cos_num,
                                     RingNumberType & denom,
                                     RingNumberType eps_num,
                                     RingNumberType eps_den)
```

computes integers *sin_num*, *cos_num* and *denom*, such that *sin_num/denom* approximates the sine of direction (*dirx,diry*). The difference between the sine and the approximating rational is bounded by *eps_num/eps_den*.

Precondition: *eps_num* \neq 0.

Implementation

The approximation is based on Farey sequences as described in the rational rotation method presented by Canny and Ressler at the 8th SoCG 1992. We use a slower version which needs no division operation in the approximation.

See Also

CGAL::Aff_transformation_2<Kernel> page [60](#)

CGAL::right_turn

bool *right_turn*(*Point_2*<*Kernel*> *p*, *Point_2*<*Kernel*> *q*, *Point_2*<*Kernel*> *r*)
returns *true* iff *p*, *q*, and *r* form a right turn.

See Also

CGAL::collinear page [142](#)
CGAL::left_turn page [177](#)
CGAL::orientation page [494](#)

CGAL::side_of_bounded_circle

Bounded_side

side_of_bounded_circle(*Point_2*<*Kernel*> *p*,
Point_2<*Kernel*> *q*,
Point_2<*Kernel*> *r*,
Point_2<*Kernel*> *t*)

returns the relative position of point *t* to the circle defined by *p*, *q* and *r*. The order of the points *p*, *q* and *r* does not matter.

Precondition: *p*, *q* and *r* are not collinear.

Bounded_side

side_of_bounded_circle(*Point_2*<*Kernel*> *p*,
Point_2<*Kernel*> *q*,
Point_2<*Kernel*> *t*)

returns the position of the point *t* relative to the circle that has *pq* as its diameter.

See Also

CGAL::coplanar_side_of_bounded_circle page [163](#)
CGAL::side_of_oriented_circle page [199](#)

CGAL::side_of_bounded_sphere

[illegible]

returns the relative position of point t to the sphere defined by p, q, r , and s . The order of the points p, q, r , and s does not matter.

Precondition: p, q, r and s are not coplanar.

$$\text{Bounded_side} \quad \text{side_of_bounded_sphere}(\text{Point_3}\langle\text{Kernel}\rangle p,$$
$$\text{Point_3}\langle\text{Kernel}\rangle q,$$
$$\text{Point_3}\langle\text{Kernel}\rangle r,$$
$$\text{Point_3}\langle\text{Kernel}\rangle t)$$

returns the position of the point t relative to the sphere passing through p , q , and r and whose center is in the plane defined by these three points.

[illegible]

returns the position of the point t relative to the sphere that has pq as its diameter.

See Also

CGAL::side_of_oriented_sphere page 497

returns the relative position of point *test* to the oriented sphere defined by *p*, *q*, *r* and *s*. The order of the points *p*, *q*, *r*, and *s* is important, since it determines the orientation of the implicitly constructed sphere. If the points *p*, *q*, *r* and *s* are positive oriented, positive side is the bounded interior of the sphere.

Precondition: *p*, *q*, *r* and *s* are not coplanar.

CGAL::side_of_bounded_sphere page 496

CGAL::squared_distance

```
#include <CGAL/squared_distance_2.h>
#include <CGAL/squared_distance_3.h>
```

```
Kernel::FT          squared_distance( Type1<Kernel> obj1, Type2<Kernel> obj2)
```

computes the square of the Euclidean distance between two geometric objects. For arbitrary geometric objects *obj1* and *obj2* the squared distance is defined as the minimal *squared_distance(p1, p2)*, where *p1* is a point of *obj1* and *p2* is a point of *obj2*. Note that for objects that have an inside (a bounded region), this inside is part of the object. So, the squared distance from a point inside is zero, not the squared distance to the closest point on the boundary.

In 2D, the types *Type1* and *Type2* can be any of the following:

- *Point_2*
- *Line_2*
- *Ray_2*
- *Segment_2*
- *Triangle_2*

In 3D, the types *Type1* and *Type2* can be any of the following:

- *Point_3*
- *Line_3*
- *Ray_3*
- *Segment_3*
- *Plane_3*

See Also

<i>CGAL::compare_distance_to_point</i>	page 145
<i>CGAL::compare_signed_distance_to_line</i>	page 146
<i>CGAL::compare_signed_distance_to_plane</i>	page 147
<i>CGAL::has_larger_distance_to_point</i>	page 168
<i>CGAL::has_larger_signed_distance_to_line</i>	page 169
<i>CGAL::has_larger_signed_distance_to_plane</i>	page 170
<i>CGAL::has_smaller_distance_to_point</i>	page 171
<i>CGAL::has_smaller_signed_distance_to_line</i>	page 172
<i>CGAL::has_smaller_signed_distance_to_plane</i>	page 173

CGAL::squared_radius

- FT* `squared_radius(Point_2<Kernel> p, Point_2<Kernel> q, Point_2<Kernel> r)`
- compute the squared radius of the circle passing through the points p , q , and r .
Precondition: p , q , and r are not collinear.
- FT* `squared_radius(Point_2<Kernel> p, Point_2<Kernel> q)`
- compute the squared radius of the smallest circle passing through p , and q , i.e. one fourth of the squared distance between p and q .
- FT* `squared_radius(Point_3<Kernel> p,
Point_3<Kernel> q,
Point_3<Kernel> r,
Point_3<Kernel> s)`
- compute the squared radius of the sphere passing through the points p , q , r and s .
Precondition: p , q , r and s are not coplanar.
- FT* `squared_radius(Point_3<Kernel> p, Point_3<Kernel> q, Point_3<Kernel> r)`
- compute the squared radius of the sphere passing through the points p , q , and r and whose center is in the same plane as those three points.
- FT* `squared_radius(Point_3<Kernel> p, Point_3<Kernel> q)`
- compute the squared radius of the smallest circle passing through p , and q , i.e. one fourth of the squared distance between p and q .

See Also

`CGAL::Circle_2<Kernel>` [page 65](#)
`CGAL::Sphere_3<Kernel>` [page 109](#)

CGAL::volume

Kernel::FT *volume*(*Point_3<Kernel> p0*,
 Point_3<Kernel> p1,
 Point_3<Kernel> p2,
 Point_3<Kernel> p3)

Computes the signed volume of the tetrahedron defined by the four points *p0*, *p1*, *p2* and *p3*.

See Also

CGAL::Tetrahedron_3<Kernel>page [112](#)

CGAL::x_equal

bool *x_equal*(*Point_2<Kernel> p*, *Point_2<Kernel> q*)
 returns *true*, iff *p* and *q* have the same *x*-coordinate.

bool *x_equal*(*Point_3<Kernel> p*, *Point_3<Kernel> q*)
 returns *true*, iff *p* and *q* have the same *x*-coordinate.

See Also

CGAL::compare_x page [149](#)
CGAL::y_equal page [205](#)
CGAL::z_equal page [206](#)

CGAL::y_equal

bool *y_equal*(*Point_2*<*Kernel*> *p*, *Point_2*<*Kernel*> *q*)
returns *true*, iff *p* and *q* have the same y-coordinate.

bool *y_equal*(*Point_3*<*Kernel*> *p*, *Point_3*<*Kernel*> *q*)
returns *true*, iff *p* and *q* have the same y-coordinate.

See Also

CGAL::compare_y page [155](#)
CGAL::x_equal page [204](#)
CGAL::z_equal page [206](#)

CGAL::z_equal

bool

z_equal(Point_3<Kernel> p, Point_3<Kernel> q)

returns *true*, iff *p* and *q* have the same *z*-coordinate.

See Also

CGAL::compare_z page [160](#)

CGAL::x_equal page [204](#)

CGAL::y_equal page [205](#)

2.13 Kernel Function Object Concepts

Kernel::Angle_2

A model for this must provide:

Angle *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*
returns *OBTUSE*, *RIGHT* or *ACUTE* depending on the
angle formed by the three points *p*, *q*, *r* (*q* being the vertex
of the angle).

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::anglepage [135](#)

Kernel::Angle_3

A model for this must provide:

Angle `fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)`
 returns *OBTUSE*, *RIGHT* or *ACUTE* depending on the angle formed by the three points *p*, *q*, *r* (*q* being the vertex of the angle).

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::anglepage [135](#)

Kernel::AreOrderedAlongLine_2

A model for this must provide:

bool *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*
 returns *true*, iff the three points are collinear and *q* lies between *p* and *r*. Note that *true* is returned, if $q=p$ or $q=r$.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::are_ordered_along_line [page 136](#)

Kernel::AreOrderedAlongLine_3

A model for this must provide:

bool *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*
 returns *true*, iff the three points are collinear and *q* lies between *p* and *r*. Note that *true* is returned, if $q=p$ or $q=r$.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::are_ordered_along_line page [136](#)

Kernel::AreParallel_2

A model for this must provide:

bool *fo.operator()*(*Kernel::Line_2* *l1*, *Kernel::Line_2* *l2*)
returns *true*, if *l1* and *l2* are parallel or if one of those (or both) is degenerate.

bool *fo.operator()*(*Kernel::Ray_2* *r1*, *Kernel::Ray_2* *r2*)
returns *true*, if *r1* and *r2* are parallel or if one of those (or both) is degenerate.

bool *fo.operator()*(*Kernel::Segment_2* *s1*, *Kernel::Segment_2* *s2*)
returns *true*, if *s1* and *s2* are parallel or if one of those (or both) is degenerate.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::parallel.....page [193](#)

Kernel::AreParallel_3

A model for this must provide:

bool *fo.operator()(Kernel::Line_3 l1, Kernel::Line_3 l2)*
returns *true*, if *l1* and *l2* are parallel or if one of those (or both) is degenerate.

bool *fo.operator()(Kernel::Plane_3 h1, Kernel::Plane_3 h2)*
returns *true*, if *h1* and *h2* are parallel or if one of those (or both) is degenerate.

bool *fo.operator()(Kernel::Ray_3 r1, Kernel::Ray_3 r2)*
returns *true*, if *r1* and *r2* are parallel or if one of those (or both) is degenerate.

bool *fo.operator()(Kernel::Segment_3 s1, Kernel::Segment_3 s2)*
returns *true*, if *s1* and *s2* are parallel or if one of those (or both) is degenerate.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::parallel.....page [193](#)

Kernel::AreStrictlyOrderedAlongLine_2

A model for this must provide:

bool *fo.operator()*(*Kernel::Point_2* *p*, *Kernel::Point_2* *q*, *Kernel::Point_2* *r*)
 returns *true*, iff the three points are collinear and *q* lies strictly between *p* and *r*. Note that *false* is returned, if $q==p$ or $q==r$.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::are_strictly_ordered_along_line page [137](#)

Kernel::AreStrictlyOrderedAlongLine_3

A model for this must provide:

bool *fo.operator()*(*Kernel::Point_3* *p*, *Kernel::Point_3* *q*, *Kernel::Point_3* *r*)

returns *true*, iff the three points are collinear and *q* lies strictly between *p* and *r*. Note that *false* is returned, if $q==p$ or $q==r$.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::are_strictly_ordered_along_line page [137](#)

Kernel::Assign_2

A model for this must provide:

```
template <class T>
bool fo.operator()( T& t, Kernel::Object_2 o)
```

assigns *o* to *t* if *o* was constructed from an object of type *T*. Returns *true*, if the assignment was possible.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::assign page ??
 CGAL::Object page [120](#)
 Kernel::Object_2 page [401](#)
 Kernel::Intersect_2 page [375](#)

Kernel::Assign_3

A model for this must provide:

```
template <class T>
bool fo.operator()( T& t, Kernel::Object_3 o)
```

assigns *o* to *t* if *o* was constructed from an object of type *T*. Returns *true*, if the assignment was possible.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::assign page ??
CGAL::Object page [120](#)
Kernel::Object_3 page [402](#)
Kernel::Intersect_3 page [376](#)

Kernel::BoundedSide_2

A model for this must provide:

Bounded_side *fo.operator()(Kernel::Circle_2 c, Kernel::Point_2 p)*

returns either *ON_UNBOUNDED_SIDE*, *ON_BOUNDED_SIDE*, or the constant *ON_BOUNDARY*, depending on where point *p* is relative to circle *c*.

Bounded_side *fo.operator()(Kernel::Triangle_2 t, Kernel::Point_2 p)*

returns either *ON_UNBOUNDED_SIDE*, *ON_BOUNDED_SIDE*, or the constant *ON_BOUNDARY*, depending on where point *p* is relative to triangle *t*.

Bounded_side *fo.operator()(Kernel::Iso_rectangle_2 r, Kernel::Point_2 p)*

returns either *ON_UNBOUNDED_SIDE*, *ON_BOUNDED_SIDE*, or the constant *ON_BOUNDARY*, depending on where point *p* is relative to rectangle *r*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Circle_2<Kernel>page [65](#)
CGAL::Triangle_2<Kernel>page [83](#)
CGAL::Iso_rectangle_2<Kernel>page [69](#)

Kernel::BoundedSide_3

A model for this must provide:

Bounded_side *fo.operator()(Kernel::Sphere_3 s, Kernel::Point_3 p)*

returns either *ON_UNBOUNDED_SIDE*, *ON_BOUNDED_SIDE*, or the constant *ON_BOUNDARY*, depending on where point *p* is with respect to sphere *s*.

Bounded_side *fo.operator()(Kernel::Tetrahedron_3 t, Kernel::Point_3 p)*

returns either *ON_UNBOUNDED_SIDE*, *ON_BOUNDED_SIDE*, or the constant *ON_BOUNDARY*, depending on where point *p* is with respect to tetrahedron *t*.

Bounded_side *fo.operator()(Kernel::Iso_cuboid_3 c, Kernel::Point_3 p)*

returns either *ON_UNBOUNDED_SIDE*, *ON_BOUNDED_SIDE*, or the constant *ON_BOUNDARY*, depending on where point *p* is with respect to iso-cuboid *c*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Iso_cuboid_3<Kernel> page [94](#)
CGAL::Sphere_3<Kernel> page [109](#)
CGAL::Tetrahedron_3<Kernel> page [112](#)

Kernel::Circle_2

A type representing circles in two dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

<i>CGAL::Circle_2<Kernel></i>	page 65
Kernel::BoundedSide_2	page 218
Kernel::ComputeSquaredRadius_2	page 256
Kernel::ConstructCenter_2	page 273
Kernel::ConstructCircle_2	page 277
Kernel::ConstructOppositeCircle_2	page 297
Kernel::Equal_2	page 361
Kernel::HasOnBoundary_2	page 363
Kernel::HasOnBoundedSide_2	page 365
Kernel::HasOnNegativeSide_2	page 367
Kernel::HasOnPositiveSide_2	page 369
Kernel::HasOnUnboundedSide_2	page 371
Kernel::IsDegenerate_2	page 377
Kernel::OrientedSide_2	page 405

Kernel::CollinearAreOrderedAlongLine_2

A model for this must provide:

bool *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*

returns *true*, iff *q* lies between *p* and *r*.

Precondition: *p*, *q* and *r* are collinear.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::collinear_are_ordered_along_line page [143](#)

Kernel::CollinearAreOrderedAlongLine_3

A model for this must provide:

bool *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*

returns *true*, iff *q* lies between *p* and *r*.

Precondition: *p*, *q* and *r* are collinear.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::collinear_are_ordered_along_line [page 143](#)

Kernel::CollinearAreStrictlyOrderedAlongLine_2

A model for this must provide:

bool *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*
 returns *true*, iff *q* lies strictly between *p* and *r*.
Precondition: *p*, *q* and *r* are collinear.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::collinear_are_strictly_ordered_along_line page [144](#)

Kernel::CollinearAreStrictlyOrderedAlongLine_3

A model for this must provide:

bool *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*

returns *true*, iff *q* lies strictly between *p* and *r*.
Precondition: *p*, *q* and *r* are collinear.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::collinear_are_strictly_ordered_along_linepage [144](#)

Kernel::CollinearHasOn_2

A model for this must provide:

bool *fo.operator()(Kernel::Ray_2 r, Kernel::Point_2 p)*
 checks if point *p* is on *r*.
Precondition: *p* is on the supporting line of *r*.

bool *fo.operator()(Kernel::Segment_2 s, Kernel::Point_2 p)*
 checks if point *p* is on *s*.
Precondition: *p* is on the supporting line of *s*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Ray_2<Kernel>page [79](#)
CGAL::Segment_2<Kernel>page [81](#)

Kernel::Collinear_2

A model for this must provide:

bool *fo.operator()*(*Kernel::Point_2* *p*, *Kernel::Point_2* *q*, *Kernel::Point_2* *r*)
returns *true*, if *p*, *q*, and *r* are collinear.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::collinearpage [142](#)

Kernel::Collinear_3

A model for this must provide:

bool *fo.operator()*(*Kernel::Point_3* *p*, *Kernel::Point_3* *q*, *Kernel::Point_3* *r*)
returns *true*, if *p*, *q*, and *r* are collinear.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::collinearpage [142](#)

Kernel::CompareAngleWithXAxis_2

A model for this must provide:

Comparison_result *fo.operator()(Kernel::Direction_2 d, Kernel::Direction_2 e)*

compares the angles between the positive x -axis and the directions in counterclockwise order.

Refines

AdaptableFunctor (with two arguments)

Kernel::CompareDistance_2

A model for this must provide:

<i>Comparison_result</i>	<i>fo.operator()</i> (<i>Kernel::Point_2</i> <i>p</i> , <i>Kernel::Point_2</i> <i>q</i> , <i>Kernel::Point_2</i> <i>r</i>)
	compares the distances of points <i>q</i> and <i>r</i> to point <i>p</i>

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::compare_distance_to_point page 145

<i>Comparison_result</i>	<i>fo.operator()</i> (<i>Kernel::Point_3</i> <i>p</i> , <i>Kernel::Point_3</i> <i>q</i> , <i>Kernel::Point_3</i> <i>r</i>)
	compares the distances of points <i>q</i> and <i>r</i> to point <i>p</i>

AdaptableFunctor (with three arguments)

CGAL::compare_distance_to_point page 145

Kernel::CompareSlope_2

A model for this must provide:

<i>Comparison_result</i>	<i>fo.operator()</i> (<i>Kernel::Line_2 l1</i> , <i>Kernel::Line_2 l2</i>)
	compares the slopes of the lines <i>l1</i> and <i>l2</i>

<i>Comparison_result</i>	<i>fo.operator()</i> (<i>Kernel::Segment_2 s1</i> , <i>Kernel::Segment_2 s2</i>)
	compares the slopes of the segments <i>s1</i> and <i>s2</i>

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_slopes page 148

Kernel::CompareXAtY_2

A model for this must provide:

Comparison_result *fo.operator()*(*Kernel::Point_2* *p*, *Kernel::Line_2* *h*)

compares the x -coordinates of p and the horizontal projection of p on h (Figure 2.5 (a)).

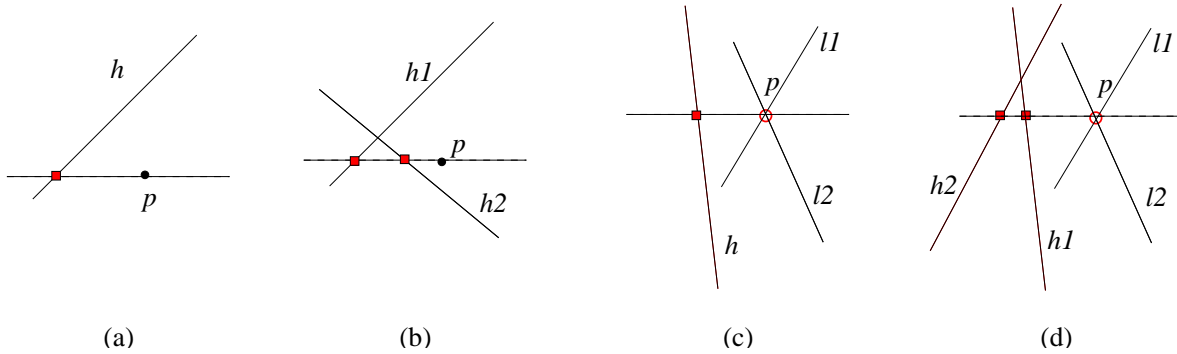


Figure 2.5: Comparison of the x -coordinates of the (implicitly given) points in the boxes, at a y -coordinate. The y -coordinate is either given explicitly (disc) or implicitly (circle).

Comparison_result *fo.operator()*(*Kernel::Point_2* *p*, *Kernel::Line_2* *h1*, *Kernel::Line_2* *h2*)

compares the x -coordinates of the horizontal projection of p on $h1$ and on $h2$ (Figure 2.5 (b)).

Comparison_result *fo.operator()*(*Kernel::Line_2* *l1*, *Kernel::Line_2* *l2*, *Kernel::Line_2* *h*)

Let p be the intersection of lines $l1$ and $l2$. This function compares the x -coordinates of p and the horizontal projection of p on h (Figure 2.5 (c)).

Comparison_result *fo.operator()*(*Kernel::Line_2* *l1*,
Kernel::Line_2 *l2*,
Kernel::Line_2 *h1*,
Kernel::Line_2 *h2*)

Let p be the intersection of lines $l1$ and $l2$. This function compares the x -coordinates of the horizontal projection of p on $h1$ and on $h2$ (Figure 2.5 (d)).

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::compare_x_at_y page [153](#)

Kernel::CompareXYZ_3

A model for this must provide:

Comparison_result *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

Compares the Cartesian coordinates of points *p* and *q* lexicographically in *xy* order: first *x*-coordinates are compared, if they are equal, *y*-coordinates are compared. If they are equal, *z*-coordinates are compared.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_xyz page [152](#)

Kernel::CompareXY_2

A model for this must provide:

Comparison_result *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*

Compares the Cartesian coordinates of points p and q lexicographically in xy order: first x -coordinates are compared, if they are equal, y -coordinates are compared.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_xy page [151](#)

Kernel::CompareXY_3

A model for this must provide:

Comparison_result *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

Compares the Cartesian coordinates of points p and q lexicographically in xy order: first x -coordinates are compared, if they are equal, y -coordinates are compared.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_xy page [151](#)

Kernel::CompareX_2

A model for this must provide:

Comparison_result *fo.operator()*(*Kernel::Point_2* *p*, *Kernel::Point_2* *q*)

compares the Cartesian *x*-coordinates of points *p* and *q*

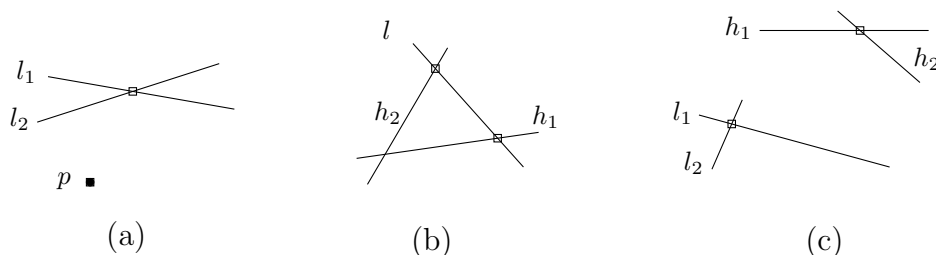


Figure 2.6: Comparison of the *x* or *y*-coordinates of the (implicitly given) points in the boxes.

Comparison_result *fo.operator()*(*Kernel::Point_2* *p*, *Kernel::Line_2* *l1*, *Kernel::Line_2* *l2*)

compares the *x*-coordinates of *p* and the intersection of lines *l1* and *l2* (Figure 2.6 (a)).

Comparison_result *fo.operator()*(*Kernel::Line_2* *l*, *Kernel::Line_2* *h1*, *Kernel::Line_2* *h2*)

compares the *x*-coordinates of the intersection of line *l* with line *h1* and with line *h2* (Figure 2.6 (b)).

Comparison_result *fo.operator()*(*Kernel::Line_2* *l1*,
Kernel::Line_2 *l2*,
Kernel::Line_2 *h1*,
Kernel::Line_2 *h2*)

compares the *x*-coordinates of the intersection of lines *l1* and *l2* and the intersection of lines *h1* and *h2* (Figure 2.6 (c)).

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_x page 149

Kernel::CompareX_3

A model for this must provide:

Comparison_result *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

Compares the Cartesian x -coordinates of points p and q

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_x page [149](#)

Kernel::CompareYAtX_2

A model for this must provide:

Comparison_result $fo.operator()(\text{Kernel::Point_2 } p, \text{Kernel::Line_2 } h)$

compares the y-coordinates of p and the vertical projection of p on h (Figure 2.7 (d)).
Precondition: h is not vertical.

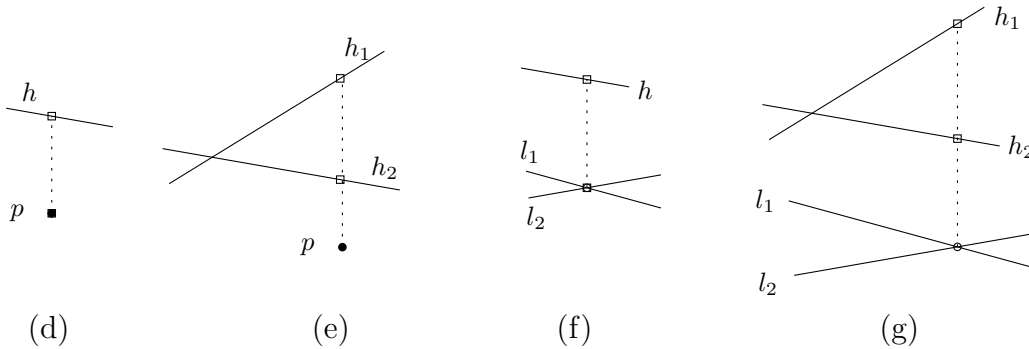


Figure 2.7: Comparison of the y-coordinates of the (implicitly given) points in the boxes, at an x -coordinate. The x -coordinate is either given explicitly (disc) or implicitly (circle).

Comparison_result $fo.operator()(\text{Kernel::Point_2 } p, \text{Kernel::Line_2 } h1, \text{Kernel::Line_2 } h2)$

This function compares the y-coordinates of the vertical projection of p on $h1$ and on $h2$ (Figure 2.4 (e)).
Precondition: $h1$ and $h2$ are not vertical.

Comparison_result $fo.operator()(\text{Kernel::Line_2 } l1, \text{Kernel::Line_2 } l2, \text{Kernel::Line_2 } h)$

Let p be the intersection of lines $l1$ and $l2$. This function compares the y-coordinates of p and the vertical projection of p on h (Figure 2.4 (f)).
Precondition: $l1, l2$ intersect and h is not vertical.

Comparison_result $fo.operator()(\text{Kernel::Line_2 } l1, \text{Kernel::Line_2 } l2, \text{Kernel::Line_2 } h1, \text{Kernel::Line_2 } h2)$

Let p be the intersection of lines $l1$ and $l2$. This function compares the y-coordinates of the vertical projection of p on $h1$ and on $h2$ (Figure 2.4 (g)).
Precondition: $l1$ and $l2$ intersect; $h1$ and $h2$ are not vertical.

Comparison_result *fo.operator()(Kernel::Point_2 p, Kernel::Segment_2 s)*

compares the y-coordinates of *p* and the vertical projection of *p* on *s*. If *s* is vertical, then return *EQUAL* when *p* lies on *s*, *SMALLER* when *p* lies under *s*, and *LARGER* otherwise.

Precondition: *p* is within the x range of *s*.

Comparison_result *fo.operator()(Kernel::Point_2 p, Kernel::Segment_2 s1, Kernel::Segment_2 s2)*

This function compares the y-coordinates of the vertical projection of *p* on *s1* and on *s2*. If *s1* or *s2* is vertical, then return *EQUAL* if they intersect, otherwise return *SMALLER* if *s1* lies below *s2*, and return *LARGER* otherwise.

Precondition: *p* is within the x range of *s1* and *s2*.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::compare_y_at_x page [157](#)

Kernel::CompareY_2

A model for this must provide:

Comparison_result *fo.operator()*(*Kernel::Point_2* *p*, *Kernel::Point_2* *q*)

Compares the Cartesian y-coordinates of points *p* and *q*

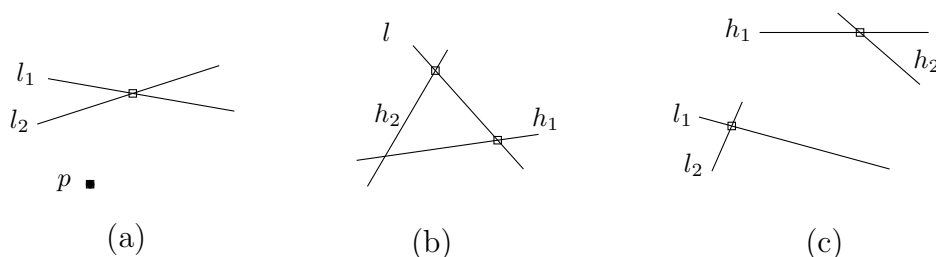


Figure 2.8: Comparison of the x or y-coordinates of the (implicitly given) points in the boxes.

Comparison_result *fo.operator()*(*Kernel::Point_2* *p*, *Kernel::Line_2* *l1*, *Kernel::Line_2* *l2*)

compares the y-coordinates of *p* and the intersection of lines *l1* and *l2* (Figure 2.8 (a)).

Comparison_result *fo.operator()*(*Kernel::Line_2* *l*, *Kernel::Line_2* *h1*, *Kernel::Line_2* *h2*)

compares the y-coordinates of the intersection of line *l* with line *h1* and with line *h2* (Figure 2.8 (b)).

Comparison_result *fo.operator()*(*Kernel::Line_2* *l1*,
Kernel::Line_2 *l2*,
Kernel::Line_2 *h1*,
Kernel::Line_2 *h2*)

compares the y-coordinates of the intersection of lines *l1* and *l2* and the intersection of lines *h1* and *h2* (Figure 2.8 (c)).

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_y page 155

Kernel::CompareY_3

A model for this must provide:

Comparison_result *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

Compares the Cartesian y-coordinates of points *p* and *q*

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_y page [155](#)

Kernel::CompareZ_3

A model for this must provide:

Comparison_result *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

Compares the Cartesian z -coordinates of points p and q

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_z page [160](#)

Kernel::ComputeA_2

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Line_2 l)*

returns the coefficient a of the line with equation $ax + by + c = 0$.

Refines

AdaptableFunctor

Kernel::ComputeB_2

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Line_2 l)*

returns the coefficient b of the line with equation $ax + by + c = 0$.

Refines

AdaptableFunctor

Kernel::ComputeC_2

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Line_2 l)*

returns the coefficient c of the line with equation $ax + by + c = 0$.

Refines

AdaptableFunctor

Kernel::ComputeArea_2

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*

returns the signed area of the triangle defined by the points *p*, *q* and *r*.

Kernel::FT *fo.operator() (Kernel::Iso_rectangle_2 r)*
returns the area of *r*.

Kernel::FT *fo.operator() (Kernel::Triangle_2 t)*
returns the signed area of *t*.

Refines

AdaptableFunctor (with one argument)

See Also

<i>CGAL::Iso_rectangle_2<Kernel></i>	page 69
<i>CGAL::Triangle_2<Kernel></i>	page 83

Kernel::FT *fo.operator() (Kernel::Triangle_3 t)*
returns the area of *t*.

Kernel::FT *fo.operator()*(*Kernel::Point_3* *p*, *Kernel::Point_3* *q*, *Kernel::Point_3* *r*)
returns the area of the triangle *p*, *q*, *r*.

AdaptableFunctor (with one argument)

CGAL::Triangle_3<Kernel> page 114

Kernel::ComputeScalarProduct_2

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Vector_2 v, Kernel::Vector_2 w)*

returns the scalar (inner) product of the two vectors *v* and *w*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Vector_2<Kernel> page [85](#)

Kernel::ComputeScalarProduct_3

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Vector_3 v, Kernel::Vector_3 w)*

returns the scalar (inner) product of the two vectors v and w .

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Vector_3<Kernel> page [116](#)

Kernel::ComputeSquaredArea_3

A model for this must provide:

<i>Kernel::FT</i>	<i>fo.operator() (Kernel::Triangle_3 t)</i>	
		returns the square of the area of <i>t</i> .

Kernel::FT *fo.operator()*(*Kernel::Point_3* *p*, *Kernel::Point_3* *q*, *Kernel::Point_3* *r*)

returns the square of the area of the triangle *p*, *q*, *r*.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Triangle_3<Kernel> page 114

Kernel::ComputeSquaredDistance_2

A model for this must provide:

Kernel::FT *fo.operator()(Type1 obj1, Type2 obj2)*

returns the squared distance between two geometrical objects of type *Type1* and *Type2*

for all pairs *Type1* and *Type2*, where the types *Type1* and *Type2* can be any of the following:

- *Kernel::Point_2*
- *Kernel::Line_2*
- *Kernel::Ray_2*
- *Kernel::Segment_2*
- *Kernel::Triangle_2*

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::squared_distance page [498](#)

Kernel::ComputeSquaredDistance_3

A model for this must provide:

Kernel::FT *fo.operator()(Type1 obj1, Type2 obj2)*

returns the squared distance between two geometrical objects of type *Type1* and *Type2*

for all pairs *Type1* and *Type2*, where the types *Type1* and *Type2* can be any of the following:

- *Kernel::Point_3*
- *Kernel::Line_3*
- *Kernel::Ray_3*
- *Kernel::Segment_3*
- *Kernel::Plane_3*

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::squared_distance page [498](#)

<i>Kernel::FT</i>	<i>fo.operator() (Kernel::Vector_2 v)</i>	
		returns the squared length of <i>v</i> .

Kernel::FT *fo.operator() (Kernel::Segment_2 s)*

returns the squared length of *s*.

AdaptableFunctor (with one argument)

<i>CGAL::Vector_2<Kernel></i>	page 85
<i>CGAL::Segment_2<Kernel></i>	page 81

Kernel::ComputeSquaredLength_3

A model for this must provide:

Kernel::FT *fo.operator() (Kernel::Vector_3 v)*
returns the squared length of *v*.

<i>Kernel::FT</i>	<i>fo.operator() (Kernel::Segment_3 s)</i>
	returns the squared length of <i>s</i> .

Refines

AdaptableFunctor (with one argument)

See Also

<i>CGAL::Vector_3<Kernel></i>	page 116
<i>CGAL::Segment_3<Kernel></i>	page 107

Kernel::ComputeSquaredRadius_2

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Circle_2 c)*

returns the squared radius of *c*.

Kernel::FT *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*

returns the squared radius of the circle passing through *p*,
q and *r*.

Precondition: *p*, *q* and *r* are not collinear.

Kernel::FT *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*

returns the squared radius of the smallest circle passing
through *p*, and *q*, i.e. one fourth of the squared distance
between *p* and *q*.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Circle_2<Kernel> [page 65](#)
CGAL::squared_radius [page 202](#)

Kernel::ComputeSquaredRadius_3

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Sphere_3 s)*
returns the squared radius of *s*.

<i>Kernel::FT</i>	<i>fo.operator()(</i> <i>Kernel::Point_3 p,</i> <i>Kernel::Point_3 q,</i> <i>Kernel::Point_3 r,</i> <i>Kernel::Point_3 s)</i>
	<p>returns the squared radius of the sphere passing through p, q, r and s.</p> <p><i>Precondition:</i> p, q, r and s are not coplanar.</p>

<i>Kernel::FT</i>	<i>fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)</i>
	returns the squared radius of the sphere passing through <i>p</i> , <i>q</i> and <i>r</i> , and whose center is in the plane defined by these three points.

<i>Kernel::FT</i>	<i>fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)</i>	returns the squared radius of the smallest circle passing through <i>p</i> , and <i>q</i> , i.e. one fourth of the squared distance between <i>p</i> and <i>q</i> .
-------------------	-------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------

Refines

AdaptableFunctor (with one argument)

See Also

<i>CGAL::Sphere_3<Kernel></i>	page 109
<i>CGAL::squared_radius</i>	page 202

Kernel::FT *fo.operator() (Kernel::Iso_cuboid_3 c)*
returns the volume of *c*.

Kernel::FT *fo.operator() (Kernel::Tetrahedron_3 t)*
returns the signed volume of *t*.

Kernel::FT

fo.operator()(Kernel::Point_3 p0,
Kernel::Point_3 p1,
Kernel::Point_3 p2,
Kernel::Point_3 p3)

returns the signed volume of the tetrahedron defined by
the four points *p0*, *p1*, *p2*, *p3*.

Refines

AdaptableFunctor (with one argument)

See Also

<i>CGAL::Iso_cuboid_3<Kernel></i>	page 94
<i>CGAL::Tetrahedron_3<Kernel></i>	page 112

Kernel::ComputeX_2

A model for this must provide:

Kernel::FT *fo.operator() (Kernel::Point_2 p)*

returns the x-coordinate of the point.

Kernel::FT *fo.operator() (Kernel::Vector_2 v)*
returns the *x*-coordinate of the vector.

Refines

AdaptableFunctor

Kernel::FT *fo.operator() (Kernel::Point_2 p)* returns the y-coordinate of the point.

Kernel::FT *fo.operator() (Kernel::Vector_2 v)*

returns the y-coordinate of the vector.

AdaptableFunctor

Kernel::ComputeXmin_2

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Iso_rectangle_2 r)*
returns the smallest *x*-coordinate of the isorectangle.

Refines

AdaptableFunctor

Kernel::ComputeYmin_2

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Iso_rectangle_2 r)*
returns the smallest y-coordinate of the isorectangle.

Refines

AdaptableFunctor

Kernel::ComputeXmax_2

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Iso_rectangle_2 r)*
returns the largest *x*-coordinate of the isorectangle.

Refines

AdaptableFunctor

Kernel::ComputeYmax_2

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Iso_rectangle_2 r)*
returns the largest y-coordinate of the isorectangle.

Refines

AdaptableFunctor

Kernel::ComputeYAtX_2

A model for this must provide:

Kernel::FT *fo.operator()(Kernel::Line_2 l, Kernel::FT x)*

returns the y-coordinate of the point at *l* with given *x*-coordinate.

Precondition: *l* is not vertical.

Refines

AdaptableFunctor

See Also

CGAL::compare_y_at_x page [157](#)

Kernel::ConstructBaseVector_3

A model for this must provide:

Kernel::Vector_3 *fo.operator()(Kernel::Plane_3 h, int index)*

when *index* == 1, returns a vector *b1* that is orthogonal to the normal *n* to plane *h*; when *index* == 2, retrns a vector *b2* that is orthogonal to *n* and *b1* and such that for an arbitrary point *p* on the plane *h*, the orientation of *p*, *p* + *b1*, *p* + *b2*, and *p* + *n* is positive.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Plane_3<Kernel> page [99](#)

Kernel::ConstructBbox_2

A model for this must provide:

CGAL::Bbox_2 *fo.operator()(Kernel::Point_2 p)*
returns the bounding box of *p*.

CGAL::Bbox_2 *fo.operator()(Kernel::Segment_2 s)*
returns the bounding box of *s*.

CGAL::Bbox_2 *fo.operator()(Kernel::Triangle_2 t)*
returns the bounding box of *t*.

CGAL::Bbox_2 *fo.operator()(Kernel::Iso_rectangle_2 i)*
returns the bounding box of *i*.

CGAL::Bbox_2 *fo.operator()(Kernel::Circle_2 c)*
returns the bounding box of *c*.

Refines

AdaptableFunctor (with one argument)

Kernel::ConstructBbox_3

A model for this must provide:

CGAL::Bbox_3 *fo.operator()(Kernel::Point_3 p)*
returns the bounding box of *p*.

CGAL::Bbox_3 *fo.operator()(Kernel::Segment_3 s)*
returns the bounding box of *s*.

CGAL::Bbox_3 *fo.operator()(Kernel::Triangle_3 t)*
returns the bounding box of *t*.

CGAL::Bbox_3 *fo.operator()(Kernel::Tetrahedron_3 t)*
returns the bounding box of *t*.

CGAL::Bbox_3 *fo.operator()(Kernel::Iso_Cuboid_3 i)*
returns the bounding box of *i*.

CGAL::Bbox_3 *fo.operator()(Kernel::Sphere_3 s)*
returns the bounding box of *s*.

Refines

AdaptableFunctor (with one argument)

Kernel::ConstructBisector_2

A model for this must provide:

Kernel::Line_2 *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*

constructs the bisector of p and q . The bisector is oriented in such a way that p lies on its positive side.

Precondition: p and q are not equal.

Kernel::Line_2 *fo.operator()(Kernel::Line_2 l1, Kernel::Line_2 l2)*

constructs the bisector of the two lines $l1$ and $l2$. In the general case, the bisector has the direction of the vector which is the sum of the normalized directions of the two lines, and which passes through the intersection of $l1$ and $l2$. If $l1$ and $l2$ are parallel, then the bisector is defined as the line which has the same direction as $l1$, and which is at the same distance from $l1$ and $l2$. This function requires that *Kernel::RT* supports the *sqrt()* operation.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::bisector page [139](#)

Kernel::ConstructBisector_3

A model for this must provide:

Kernel::Plane_3 *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

constructs the bisector plane of p and q . The bisector is oriented in such a way that p lies on its positive side.

Precondition: p and q are not equal.

Kernel::Plane_3 *fo.operator()(Kernel::Plane_3 h1, Kernel::Plane_3 h2)*

constructs the bisector of the two planes $h1$ and $h2$. In the general case, the bisector has a normal vector which has the same direction as the sum of the normalized normal vectors of the two planes, and passes through the intersection of $h1$ and $h2$. If $h1$ and $h2$ are parallel, then the bisector is defined as the plane which has the same oriented normal vector as $h1$, and which is at the same distance from $h1$ and $h2$. This function requires that *Kernel::RT* supports the *sqrt()* operation.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::bisector page [139](#)

Kernel::ConstructCartesianConstIterator_2

A model for this must provide:

Kernel::Cartesian_const_iterator_2

fo.operator()(Kernel::Point_2 p)

returns an iterator on the 0'th Cartesian coordinate of *p*.

Kernel::Cartesian_const_iterator_2

fo.operator()(Kernel::Point_2 p, int)

returns the past the end iterator of the Cartesian coordinates of *p*.

Refines

AdaptableFunctor (with one argument)

See Also

Kernel::CartesianConstIterator_2 page [345](#)

Kernel::ConstructCartesianConstIterator_3

A model for this must provide:

Kernel::Cartesian_const_iterator_3

fo.operator()(Kernel::Point_3 p)

returns an iterator on the 0'th Cartesian coordinate of *p*.

Kernel::Cartesian_const_iterator_3

fo.operator()(Kernel::Point_3 p, int)

returns the past the end iterator of the Cartesian coordinates of *p*.

Refines

AdaptableFunctor (with one argument)

See Also

Kernel::CartesianConstIterator_3page ??

Kernel::ConstructCenter_2

A model for this must provide:

```
Kernel::Point_2      fo.operator()( Kernel::Circle_2 c)
                                compute the center of the circle c.
```

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Circle_2<Kernel>page [65](#)

$$\text{Kernel}::\text{Point}_3 \quad \text{fo.operator()}(\text{Kernel}::\text{Sphere}_3 \text{ s})$$

compute the center of the sphere s .

AdaptableFunctor (with one argument)

CGAL::Sphere_3<Kernel> page 109

Kernel::ConstructCentroid_2

A model for this must provide:

Kernel::Point_2 *fo.operator()*(*Kernel::Point_2* *p*, *Kernel::Point_2* *q*, *Kernel::Point_2* *r*)

compute the centroid of the points *p*, *q*, and *r*.

```
Kernel::Point_2
    fo.operator()( Kernel::Point_2 p,
                   Kernel::Point_2 q,
                   Kernel::Point_2 r,
                   Kernel::Point_2 s)

compute the centroid of the points p, q, r and s.
```

<i>Kernel::Point_2</i>	<i>fo.operator() (Kernel::Triangle_2 t)</i>	
		compute the centroid of the triangle <i>t</i> .

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::centroid page 2345

Kernel::ConstructCentroid_3

A model for this must provide:

Kernel::Point_3 *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*
compute the centroid of the points *p*, *q*, and *r*.

Kernel::Point_3 *fo.operator()(Kernel::Point_3 p,*
 Kernel::Point_3 q,
 Kernel::Point_3 r,
 Kernel::Point_3 s)
compute the centroid of the points *p*, *q*, *r* and *s*.

Kernel::Point_3 *fo.operator()(Kernel::Triangle_3 t)*
compute the centroid of the triangle *t*.

Kernel::Point_3 *fo.operator()(Kernel::Tetrahedron_3 t)*
compute the centroid of the tetrahedron *t*.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::centroid page [2345](#)

A model for this must provide:

introduces a variable of type *Kernel::Circle_2*. It is initialized to the circle with center *center*, squared radius *squared_radius* and orientation *orientation*.
Precondition: *orientation* \neq *COLLINEAR*, and further, *squared_radius* > 0 .

introduces a variable of type *Kernel::Circle_2*. It is initialized to the unique circle which passes through the points p , q and r . The orientation of the circle is the orientation of the point triple p , q , r .
Precondition: p , q , and r are not collinear.

introduces a variable of type *Kernel::Circle_2*. It is initialized to the circle with diameter \overline{pq} and orientation *orientation*.
Precondition: *orientation* \neq *COLLINEAR*.

introduces a variable of type *Kernel::Circle_2*. It is initialized to the circle with center *center*, squared radius zero and orientation *orientation*.
Precondition: *orientation* \neq *COLLINEAR*.
Postcondition: *.is_degenerate()* = *true*.

AdaptableFunctor (with three arguments)

CGAL::Circle_2<Kernel> page 65

Kernel::ConstructCircumcenter_2

A model for this must provide:

Kernel::Point_2 *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*

compute the center of the circle passing through the points p , q , and r .
Precondition: p , q , and r are not collinear.

Kernel::Point_2 *fo.operator()(Kernel::Triangle_2 t)*

compute the center of the circle passing through the three vertices of t .
Precondition: t is not degenerate.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::circumcenter page [141](#)

Kernel::ConstructCircumcenter_3

A model for this must provide:

Kernel::Point_3 *fo.operator()*(*Kernel::Point_3* *p*,
 Kernel::Point_3 *q*,
 Kernel::Point_3 *r*,
 Kernel::Point_3 *s*)

compute the center of the sphere passing through the points *p*, *q*, *r*, and *s*.

Precondition: *p*, *q*, *r*, and *s* are not coplanar.

Kernel::Point_3 *fo.operator()*(*Kernel::Tetrahedron_3* *t*)

compute the center of the sphere passing through the vertices of *t*.

Precondition: *t* is not degenerate.

Kernel::Point_3 *fo.operator()*(*Kernel::Point_3* *p*, *Kernel::Point_3* *q*, *Kernel::Point_3* *r*)

compute the center of the circle passing through the points *p*, *q* and *r*.

Precondition: *p*, *q* and *r* are not collinear.

Kernel::Point_3 *fo.operator()*(*Kernel::Triangle_3* *t*)

compute the center of the circle passing through the vertices of *t*.

Precondition: *t* is not degenerate.

Refines

AdaptableFunctor (with four arguments)

See Also

CGAL::circumcenter page [141](#)

Kernel::Vector_3 *fo.operator()*(*Kernel::Vector_3* *v*, *Kernel::Vector_3* *w*)
computes the cross product of *v* and *w*.

AdaptableFunctor (with two arguments)

CGAL::cross_product page 164

Kernel::ConstructDifferenceOfVectors_2

A model for this must provide:

Kernel::Vector_2 *fo.operator()(Kernel::Vector_2 v1, Kernel::Vector_2 v2)*

introduces the vector $v1 - v2$.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Vector_2<Kernel> page [85](#)

Kernel::ConstructDirection_2

A model for this must provide:

Kernel::Direction_2 *fo.operator()(Kernel::Vector_2 v)*
introduces the direction of vector *v*.

Kernel::Direction_2 *fo.operator()(Kernel::Line_2 l)*
introduces the direction of line *l*.

Kernel::Direction_2 *fo.operator()(Kernel::Ray_2 r)*
introduces the direction of ray *r*.

Kernel::Direction_2 *fo.operator()(Kernel::Segment_2 s)*
introduces the direction of segment *s*.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Direction_2<Kernel>page [67](#)

Kernel::ConstructDirection_3

A model for this must provide:

Kernel::Direction_3 *fo.operator()(Kernel::Vector_3 v)*

introduces a direction initialised with the direction of vector *v*.

Kernel::Direction_3 *fo.operator()(Kernel::Line_3 l)*

introduces the direction of line *l*.

Kernel::Direction_3 *fo.operator()(Kernel::Ray_3 r)*

introduces the direction of ray *r*.

Kernel::Direction_3 *fo.operator()(Kernel::Segment_3 s)*

introduces the direction of segment *s*.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Direction_3<Kernel>page [92](#)

$$\text{Kernel::Iso_cuboid_3} \quad fo.operator()(\text{Kernel::Point_3 } p, \text{Kernel::Point_3 } q)$$

introduces an iso-oriented cuboid with diagonal opposite vertices p and q such that p is the lexicographically smallest point in the cuboid.

Kernel::Iso_cuboid_3 *fo.operator() (Kernel::Point_3 left,*
 Kernel::Point_3 right,
 Kernel::Point_3 bottom,
 Kernel::Point_3 top,
 Kernel::Point_3 far,
 Kernel::Point_3 close)

introduces an iso-oriented cuboid *fo* whose minimal *x* coordinate is the one of *left*, the maximal *x* coordinate is the one of *right*, the minimal *y* coordinate is the one of *bottom*, the maximal *y* coordinate is the one of *top*, the minimal *z* coordinate is the one of *far*, the maximal *z* coordinate is the one of *close*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Iso_cuboid_3<Kernel> page 94

Kernel::ConstructIsoRectangle_2

A model for this must provide:

$$\text{Kernel::Iso_rectangle_2} \quad fo.operator()(\text{Kernel::Point_2 } p, \text{Kernel::Point_2 } q)$$

introduces an iso-oriented rectangle with diagonal opposite vertices p and q such that p is the lexicographically smallest point in the rectangle.

```
Kernel::Iso_rectangle_2   fo.operator() ( Kernel::Point_2 left,
                             Kernel::Point_2 right,
                             Kernel::Point_2 bottom,
                             Kernel::Point_2 top)
```

introduces an iso-oriented rectangle *fo* whose minimal *x* coordinate is the one of *left*, the maximal *x* coordinate is the one of *right*, the minimal *y* coordinate is the one of *bottom*, the maximal *y* coordinate is the one of *top*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Iso_rectangle_2<Kernel> page 69

Kernel::ConstructLiftedPoint_3

A model for this must provide:

Kernel::Point_3 *fo.operator()(Kernel::Plane_3 h, Kernel::Point_2 p)*

returns a point q on plane h , such that the projection of this point onto the xy -plane is p .

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Plane_3<Kernel> [page 99](#)

Kernel::ConstructLine_2

A model for this must provide:

$$\text{Kernel::Line_2} \quad fo.operator()(\text{Kernel::Point_2 } p, \text{Kernel::Point_2 } q)$$

introduces a line passing through the points p and q . Line is directed from p to q .

```
Kernel::Line_2 fo.operator() ( Kernel::Point_2 p, Kernel::Direction_2 d)
```

introduces a line passing through point p with direction d .

```
Kernel::Line_2      fo.operator()( Kernel::Point_2 p, Kernel::Vector_2 v)
```

introduces a line passing through point p and oriented by ν .

$$Kernel::Line_2 \quad fo.operator() (Kernel::Segment_2 s)$$

introduces a line supporting the segment s , oriented from source to target.

$$Kernel::Line_2 \quad fo.operator() (Kernel::Ray_2 \ r)$$

introduces a line supporting the ray r , with same orientation.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Line_2<Kernel> page 72

Kernel::ConstructLine_3

A model for this must provide:

Kernel::Line_3 *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*
introduces a line passing through the points *p* and *q*. Line is directed from *p* to *q*.

Kernel::Line_3 *fo.operator()(Kernel::Point_3 p, Kernel::Vector_3 v)*
introduces a line passing through point *p* and oriented by *v*.

Kernel::Line_3 *fo.operator()(Kernel::Point_3 p, Kernel::Direction_3 d)*
introduces a line passing through point *p* with direction *d*.

Kernel::Line_3 *fo.operator()(Kernel::Segment_3 s)*
returns the line supporting the segment *s*, oriented from source to target.

Kernel::Line_3 *fo.operator()(Kernel::Ray_3 r)*
returns the line supporting the ray *r*, with the same orientation.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Line_3<Kernel> page [97](#)

Kernel::ConstructMaxVertex_2

A model for this must provide:

Kernel::Point_2 *fo.operator()(Kernel::Iso_rectangle_2 r)*
 returns the vertex of *r* with lexicographically largest co-ordinates.

Kernel::Point_2 *fo.operator()(Kernel::Segment_2 s)*
 returns the vertex of *s* with lexicographically largest co-ordinates.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Iso_rectangle_2<Kernel> page [69](#)
CGAL::Segment_2<Kernel> page [81](#)

Kernel::ConstructMaxVertex_3

A model for this must provide:

Kernel::Point_3 *fo.operator()(Kernel::Iso_cuboid_3 c)*

returns the vertex of *c* with lexicographically largest coordinates.

Kernel::Point_3 *fo.operator()(Kernel::Segment_3 s)*

returns the vertex of *s* with lexicographically largest coordinates.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Iso_cuboid_3<Kernel> page [94](#)
CGAL::Segment_3<Kernel> page [107](#)

Kernel::ConstructMidpoint_2

A model for this must provide:

$$\text{Kernel::Point_2} \quad fo.operator()(\text{Kernel::Point_2 } p, \text{Kernel::Point_2 } q)$$

computes the midpoint of the segment pq .

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::midpoint.....page 493

Kernel::Point_3 *fo.operator() (Kernel::Point_3 p, Kernel::Point_3 q)*

computes the midpoint of the segment pq .

AdaptableFunctor (with two arguments)

CGAL::midpoint page 493

Kernel::ConstructMinVertex_2

A model for this must provide:

Kernel::Point_2 *fo.operator()(Kernel::Iso_rectangle_2 r)*
 returns the vertex of *r* with lexicographically smallest co-ordinates.

Kernel::Point_2 *fo.operator()(Kernel::Segment_2 s)*
 returns the vertex of *s* with lexicographically smallest co-ordinates.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Iso_rectangle_2<Kernel> page [69](#)
CGAL::Segment_2<Kernel> page [81](#)

Kernel::ConstructMinVertex_3

A model for this must provide:

Kernel::Point_3 *fo.operator()(Kernel::Iso_cuboid_3 c)*

returns the vertex of *c* with lexicographically smallest coordinates.

Kernel::Point_3 *fo.operator()(Kernel::Segment_3 s)*

returns the vertex of *s* with lexicographically smallest coordinates.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Iso_cuboid_3<Kernel> page [94](#)
CGAL::Segment_3<Kernel> page [107](#)

Kernel::ConstructObject_2

A model for this must provide:

```
template <class T>
Object_2          fo.operator()( T t)          constructs an object that contains t and returns it.
```

Refines

AdaptableFunctor (with one argument)

See Also

<i>CGAL::make_object</i>	page ??
<i>CGAL::Object</i>	page 120
Kernel::Assign_2	page 216
Kernel::Assign_3	page 217
Kernel::Object_2.....	page 401
Kernel::Object_3.....	page 402

Kernel::ConstructObject_3

A model for this must provide:

template <class T>
Object_3 *fo.operator()(T t)* constructs an object that contains *t* and returns it.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::make_object page ??
CGAL::Object page [120](#)
Kernel::Assign_2 page [216](#)
Kernel::Assign_3 page [217](#)
Kernel::Object_2 page [401](#)
Kernel::Object_3 page [402](#)

Kernel::ConstructOppositeCircle_2

A model for this must provide:

Kernel::Circle_2 *fo.operator()(Kernel::Circle_2 c)*

returns the circle with the same center and squared radius as *c*, but with opposite orientation.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Circle_2<Kernel>page [65](#)

$$\text{Kernel}::\text{Direction_2} \quad \text{fo.operator()}(\text{Kernel}::\text{Direction_2 } d)$$

returns the direction opposite to d .

AdaptableFunctor (with one argument)

CGAL::Direction_2<Kernel>page 67

Kernel::ConstructOppositeDirection_3

A model for this must provide:

```
Kernel::Direction_3      fo.operator()( Kernel::Direction_3 d)
                                returns the direction opposite to d.
```

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Direction_3<Kernel>.....page [92](#)

Kernel::ConstructOppositeLine_2

A model for this must provide:

Kernel::Line_2 *fo.operator()(Kernel::Line_2 l)*

returns the line representing the same set of points as *l*,
but with opposite direction.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Line_2<Kernel> [page 72](#)

Kernel::ConstructOppositeLine_3

A model for this must provide:

Kernel::Line_3 *fo.operator()(Kernel::Line_3 l)*

returns the line representing the same set of points as *l*,
but with opposite direction.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Line_3<Kernel> page [97](#)

Kernel::ConstructOppositePlane_3

A model for this must provide:

Kernel::Plane_3 *fo.operator()(Kernel::Plane_3 p)*

returns the plane representing the same set of points as *p*,
but with opposite orientation.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Plane_3<Kernel> [page 99](#)

Kernel::ConstructOppositeRay_2

A model for this must provide:

Kernel::Ray_2 *fo.operator()(Kernel::Ray_2 r)*

returns the ray with the same source as *r*, but in opposite direction.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Ray_2<Kernel>page [79](#)

Kernel::ConstructOppositeRay_3

A model for this must provide:

Kernel::Ray_3 *fo.operator()(Kernel::Ray_3 r)*

returns the ray with the same source as *r*, but in opposite direction.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Ray_3<Kernel>page [105](#)

Kernel::ConstructOppositeSegment_2

A model for this must provide:

Kernel::Segment_2 *fo.operator()(Kernel::Segment_2 s)*

returns the segment representing the same set of points as *s*, but with opposite orientation.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Segment_2<Kernel>page [81](#)

Kernel::ConstructOppositeSegment_3

A model for this must provide:

Kernel::Segment_3 *fo.operator()(Kernel::Segment_3 s)*

returns the segment representing the same set of points as *s*, but with opposite orientation.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Segment_3<Kernel>.....page [107](#)

Kernel::ConstructOppositeSphere_3

A model for this must provide:

Kernel::Sphere_3 *fo.operator()(Kernel::Sphere_3 s)*

returns the sphere with the same center and squared radius as *s*, but with opposite orientation.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Sphere_3<Kernel> [page 109](#)

Kernel::ConstructOppositeTriangle_2

A model for this must provide:

Kernel::Triangle_2 *fo.operator()(Kernel::Triangle_2 t)*

returns the triangle with opposite orientation to *t* (this flips the positive and the negative side, but not bounded and unbounded side).

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Triangle_2<Kernel>page [83](#)

Kernel::ConstructOppositeVector_2

A model for this must provide:

```
Kernel::Vector_2      fo.operator()( Kernel::Vector_2 v)
                        returns the vector -v.
```

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Vector_2<Kernel> page [85](#)

Kernel::Vector_3 *fo.operator() (Kernel::Vector_3 v)*

returns the vector -v.

AdaptableFunctor (with one argument)

CGAL::Vector_3<Kernel> page 116

Kernel::ConstructOrthogonalVector_3

A model for this must provide:

Kernel::Vector_3 *fo.operator()(Kernel::Plane_3 p)*

returns a vector that is orthogonal to the plane *p* and directed to the positive side of *p*.

Kernel::Vector_3 *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*

returns a vector that is orthogonal to the plane defined by *Kernel::ConstructPlane_3()(p, q, r)* and directed to the positive side of this plane.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Plane_3<Kernel> page [99](#)
Kernel::ConstructCrossProductVector_3 page [280](#)

Kernel::ConstructPerpendicularDirection_2

A model for this must provide:

Kernel::Direction_2 *fo.operator()(Kernel::Direction_2 d, Orientation o)*

introduces a direction orthogonal to *d*. If *o* is *CLOCKWISE*, *d* is rotated clockwise; if *o* is *COUNTERCLOCKWISE*, *d* is rotated counterclockwise.
Precondition: o is not *COLLINEAR*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Direction_2<Kernel>page [67](#)

Kernel::ConstructPerpendicularLine_2

A model for this must provide:

Kernel::Line_2 *fo.operator()(Kernel::Line_2 l, Kernel::Point_2 p)*

returns the line perpendicular to *l* and passing through *p*, where the direction is the direction of *l* rotated counter-clockwise by 90 degrees.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Line_2<Kernel> [page 72](#)

Kernel::ConstructPerpendicularLine_3

A model for this must provide:

Kernel::Line_3 *fo.operator()(Kernel::Plane_3 pl, Kernel::Point_3 p)*

returns the line that is perpendicular to *pl* and that passes through point *p*. The line is oriented from the negative to the positive side of *pl*

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Plane_3<Kernel> [page 99](#)

Kernel::ConstructPerpendicularPlane_3

A model for this must provide:

Kernel::Plane_3 *fo.operator()(Kernel::Line_3 l, Kernel::Point_3 p)*

returns the plane perpendicular to *l* passing through *p*,
such that the normal direction of the plane coincides with
the direction of the line.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Plane_3<Kernel> page [99](#)

Kernel::ConstructPerpendicularVector_2

A model for this must provide:

Kernel::Vector_2 *fo.operator()(Kernel::Vector_2 v, Orientation o)*

returns *v* rotated clockwise by 90 degrees, if *o* is *CLOCKWISE*, and rotated counterclockwise otherwise.
Precondition: o is not *COLLINEAR*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Vector_2<Kernel> page [85](#)

Kernel::ConstructPlane_3

A model for this must provide:

Kernel::Plane_3 *fo.operator()(Kernel::RT a, Kernel::RT b, Kernel::RT c, Kernel::RT d)*

creates a plane defined by the equation $apx + bpy + cz + d = 0$. Notice that is degenerate if $a = b = c$.

Kernel::Plane_3 *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*

creates a plane passing through the points p , q and r . The plane is oriented such that p , q and r are oriented in a positive sense (that is counterclockwise) when seen from the positive side of the plane. Notice that is degenerate if the points are collinear.

Kernel::Plane_3 *fo.operator()(Kernel::Point_3 p, Kernel::Direction_3 d)*

introduces a plane that passes through point p and that has as an orthogonal direction equal to d .

Kernel::Plane_3 *fo.operator()(Kernel::Point_3 p, Kernel::Vector_3 v)*

introduces a plane that passes through point p and that is orthogonal to v .

Kernel::Plane_3 *fo.operator()(Kernel::Line_3 l, Kernel::Point_3 p)*

introduces a plane that is defined through the three points $l.point(0)$, $l.point(1)$ and p .

Kernel::Plane_3 *fo.operator()(Kernel::Ray_3 r, Kernel::Point_3 p)*

introduces a plane that is defined through the three points $r.point(0)$, $r.point(1)$ and p .

Kernel::Plane_3 *fo.operator()(Kernel::Segment_3 s, Kernel::Point_3 p)*

introduces a plane that is defined through the three points $s.source()$, $s.target()$ and p .

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Plane_3<Kernel> page [99](#)

Kernel::ConstructPointOn_2

A model for this must provide:

Kernel::Point_2 *fo.operator()(Kernel::Line_2 l, int i)*

returns an arbitrary point on *l*. It holds *point(i) == point(j)*, iff *i==j*. Furthermore, is directed from *point(i)* to *point(j)*, for all *i < j*.

Kernel::Point_2 *fo.operator()(Kernel::Ray_2 r, int i)*

returns a point on *r*. *point(0)* is the source, *point(i)*, with *i > 0*, is different from the source.
Precondition: i ≥ 0.

Kernel::Point_2 *fo.operator()(Kernel::Segment_2 s, int i)*

returns source or target of *s*: *point(0)* returns the source of *s*, *point(1)* returns the target of *s*. The parameter *i* is taken modulo 2, which gives easy access to the other end point.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Line_2<Kernel> page [72](#)
CGAL::Ray_2<Kernel> page [79](#)
CGAL::Segment_2<Kernel> page [81](#)

Kernel::ConstructPointOn_3

A model for this must provide:

Kernel::Point_3 *fo.operator()(Kernel::Line_3 l, int i)*

returns an arbitrary point on *l*. It holds *point(i) == point(j)*, iff *i==j*. Furthermore, is directed from *point(i)* to *point(j)*, for all *i < j*.

Kernel::Point_3 *fo.operator()(Kernel::Plane_3 h)*

returns an arbitrary point on *h*.

Kernel::Point_3 *fo.operator()(Kernel::Ray_3 r, int i)*

returns a point on *r*. *point(0)* is the source, *point(i)*, with *i > 0*, is different from the source.
Precondition: i ≥ 0.

Kernel::Point_3 *fo.operator()(Kernel::Segment_3 s, int i)*

returns source or target of *s*: *point(0)* returns the source of *s*, *point(1)* returns the target of *s*. The parameter *i* is taken modulo 2, which gives easy access to the other end point.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Line_3<Kernel> page [97](#)
CGAL::Plane_3<Kernel> page [99](#)
CGAL::Ray_3<Kernel> page [105](#)
CGAL::Segment_3<Kernel> page [107](#)

Kernel::ConstructPoint_2

A model for this must provide:

$$Kernel::Point_2 \quad fo.operator() (Origin ORIGIN)$$

introduces a variable with Cartesian coordinates $(0,0)$.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Point_2<Kernel> page 75

Kernel::ConstructPoint_3

A model for this must provide:

Kernel::Point_3 *fo.operator()(Origin ORIGIN)*

introduces a point with Cartesian coordinates(0,0,0).

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Point_3<Kernel> page [102](#)

Kernel::ConstructProjectedPoint_2

A model for this must provide:

```
Kernel::Point_2      fo.operator()( Kernel::Line_2 l, Kernel::Point_2 p)
```

returns the orthogonal projection of p onto l .

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Line_2<Kernel> page 72

Kernel::Point_3 *fo.operator()*(*Kernel::Line_3* *l*, *Kernel::Point_3* *p*)

returns the orthogonal projection of *p* onto *l*.

Kernel::Point_3 *fo.operator() (Kernel::Plane_3 h, Kernel::Point_3 p)*

returns the orthogonal projection of *p* onto *h*.

AdaptableFunctor (with two arguments)

<i>CGAL::Line_3<Kernel></i>	page 97
<i>CGAL::Plane_3<Kernel></i>	page 99

Kernel::ConstructProjectedXYPoint_2

A model for this must provide:

Kernel::Point_2 *fo.operator()(Kernel::Plane_3 h, Kernel::Point_3 p)*

returns the image point of the projection of p under an affine transformation, which maps h onto the xy -plane, with the z -coordinate removed.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Plane_3<Kernel> page [99](#)

Kernel::ConstructRay_2

A model for this must provide:

Kernel::Ray_2 *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*
introduces a ray with source *p* and passing through point *q*.

Kernel::Ray_2 *fo.operator()(Kernel::Point_2 p, Kernel::Vector_2 v)*
introduces a ray starting at source *p* with the direction given by *v*.

Kernel::Ray_2 *fo.operator()(Kernel::Point_2 p, Kernel::Direction_2 d)*
introduces a ray starting at source *p* with direction *d*.

Kernel::Ray_2 *fo.operator()(Kernel::Point_2 p, Kernel::Line_2 l)*
introduces a ray starting at source *p* with the same direction as *l*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Ray_2<Kernel> page [79](#)

Kernel::ConstructRay_3

A model for this must provide:

$$\text{Kernel::Ray_3} \quad fo.operator()(\text{Kernel::Point_3 } p, \text{Kernel::Point_3 } q)$$

introduces a ray with source p and passing through point q .

```
Kernel::Ray_3      fo.operator()( Kernel::Point_3 p, Kernel::Vector_3 v)
```

introduces a ray with source p and with the direction given by v .

$$\text{Kernel::Ray}_3 \quad fo.operator()(\text{Kernel::Point}_3 p, \text{Kernel::Direction}_3 d)$$

introduces a ray with source p and with direction d .

```
Kernel::Ray_3      fo.operator()( Kernel::Point_3 p, Kernel::Line_3 l)
```

introduces a ray with source p and with the same direction as l .

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Ray_3<Kernel> page 105

```
Kernel::Vector_2 fo.operator()( Kernel::Vector_2 v, Kernel::RT scale)
```

produces the vector v scaled by a factor $scale$.

```
Kernel::Vector_2 fo.operator()( Kernel::Vector_2 v, Kernel::FT scale)
```

produces the vector v scaled by a factor $scale$.

AdaptableFunctor (with two arguments)

CGAL::Vector_2<Kernel> page 85

Kernel::ConstructScaledVector_3

A model for this must provide:

Kernel::Vector_3 *fo.operator()(Kernel::Vector_3 v, Kernel::RT scale)*
 produces the vector *v* scaled by a factor *scale*.

Kernel::Vector_3 *fo.operator()(Kernel::Vector_3 v, Kernel::FT scale)*
 produces the vector *v* scaled by a factor *scale*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Vector_3<Kernel> page [116](#)

Kernel::ConstructSumOfVectors_2

A model for this must provide:

Kernel::Vector_2 *fo.operator()(Kernel::Vector_2 v1, Kernel::Vector_2 v2)*

introduces the vector $v1 + v2$.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Vector_2<Kernel> page [85](#)

Kernel::ConstructSegment_2

A model for this must provide:

Kernel::Segment_2 *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*

introduces a segment with source p and target q . The segment is directed from the source towards the target.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Segment_2<Kernel>page [81](#)

Kernel::ConstructSegment_3

A model for this must provide:

Kernel::Segment_3 *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

introduces a segment with source *p* and target *q*. It is directed from the source towards the target.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Segment_3<Kernel>.....page [107](#)

Kernel::ConstructSphere_3

A model for this must provide:

Kernel::Sphere_3 *fo.operator()*(*Kernel::Point_3* center,
 Kernel::FT squared_radius,
 Orientation orientation = COUNTERCLOCKWISE)

introduces a sphere initialized to the sphere with center *center*, squared radius *squared_radius* and orientation *orientation*.

Precondition: *orientation* \neq COPLANAR, and furthermore, *squared_radius* \geq 0.

Kernel::Sphere_3 *fo.operator()*(*Kernel::Point_3* p,
 Kernel::Point_3 q,
 Kernel::Point_3 r,
 Kernel::Point_3 s)

introduces a sphere initialized to the unique sphere which passes through the points *p*, *q*, *r* and *s*. The orientation of the sphere is the orientation of the point quadruple *p*, *q*, *r*, *s*.

Precondition: *p*, *q*, *r*, and *s* are not collinear.

Kernel::Sphere_3 *fo.operator()*(*Kernel::Point_3* p,
 Kernel::Point_3 q,
 Kernel::Point_3 r,
 Orientation o = COUNTERCLOCKWISE)

introduces a sphere initialized to the smallest sphere which passes through the points *p*, *q*, and *r*. The orientation of the sphere is *o*.

Precondition: *o* is not COPLANAR.

Kernel::Sphere_3 *fo.operator()*(*Kernel::Point_3* p,
 Kernel::Point_3 q,
 Orientation o = COUNTERCLOCKWISE)

introduces a sphere initialized to the smallest sphere which passes through the points *p* and *q*. The orientation of the sphere is *o*.

Precondition: *o* is not COPLANAR.

Kernel::Sphere_3 *fo.operator()*(*Kernel::Point_3* center,
 Orientation orientation = COUNTERCLOCKWISE)

introduces a sphere *s* initialized to the sphere with center *center*, squared radius zero and orientation *orientation*.

Precondition: *orientation* \neq COPLANAR.

Postcondition: *s.is_degenerate()* = true.

Refines

AdaptableFunctor (with four arguments)

See Also

CGAL::Sphere_3<Kernel> [page 109](#)

Kernel::ConstructSupportingPlane_3

A model for this must provide:

Kernel::Plane_3 *fo.operator()(Kernel::Triangle_3 t)*

returns the supporting plane of *t*, with same orientation.

Refines

AdaptableFunctor (with one argument)

See Also

CGAL::Triangle_3<Kernel> page [114](#)

Kernel::ConstructTetrahedron_3

A model for this must provide:

```
Kernel::Tetrahedron_3    fo.operator()( Kernel::Point_3 p0,
                                         Kernel::Point_3 p1,
                                         Kernel::Point_3 p2,
                                         Kernel::Point_3 p3)
```

introduces a tetrahedron with vertices p_0 , p_1 , p_2 and p_3 .

Refines

AdaptableFunctor (with four arguments)

See Also

CGAL::Tetrahedron_3<Kernel> page [112](#)

Kernel::ConstructTranslatedPoint_2

A model for this must provide:

```
Kernel::Point_2      fo.operator()( Kernel::Point_2 p, Kernel::Vector_2 v)
```

returns the point obtained by translating p by the vector v .

```
Kernel::Point_2      fo.operator()( Origin o, Kernel::Vector_2 v)
```

returns the point obtained by translating a point at the origin by the vector v .

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Point_2<Kernel> page 75

$$\text{Kernel::Point_3} \quad fo.operator()(\text{Kernel::Point_3 } p, \text{Kernel::Vector_3 } v)$$
$$\text{Kernel::Point_2} \quad fo.operator()(\text{Origin } o, \text{Kernel::Vector_2 } v)$$

Refines

See Also

CGAL::Point_3<Kernel> page 102

Kernel::ConstructTriangle_2

A model for this must provide:

Kernel::Triangle_2 *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*

introduces a triangle with vertices *p*, *q* and *r*.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::Triangle_2<Kernel>.....page [83](#)

$$\text{Kernel::Triangle_3} \quad fo.operator()(\text{Kernel::Point_3 } p, \text{Kernel::Point_3 } q, \text{Kernel::Point_3 } r)$$

introduces a triangle with vertices p , q and r .

AdaptableFunctor (with three arguments)

CGAL::Triangle_3<Kernel> page 114

Kernel::ConstructVector_2

A model for this must provide:

Kernel::Vector_2 *fo.operator()(Kernel::Point_2 a, Kernel::Point_2 b)*
introduces the vector $b - a$.

Kernel::Vector_2 *fo.operator()(Origin o, Kernel::Point_2 b)*
introduces the vector b .

Kernel::Vector_2 *fo.operator()(Kernel::Point_2 a, Origin o)*
introduces the vector $-a$.

Kernel::Vector_2 *fo.operator()(Kernel::Segment_2 s)*
introduces the vector $s.target() - s.source()$.

Kernel::Vector_2 *fo.operator()(Kernel::Ray_2 r)*
introduces a vector having the same direction as r .

Kernel::Vector_2 *fo.operator()(Kernel::Line_2 l)*
introduces a vector having the same direction as l .

Kernel::Vector_2 *fo.operator()(Null_vector NULL_VECTOR)*
introduces a null vector .

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Vector_2<Kernel> page [85](#)
Kernel::ConstructScaledVector_2 page [328](#)

Kernel::ConstructVector_3

A model for this must provide:

Kernel::Vector_3 *fo.operator()(Kernel::Point_3 a, Kernel::Point_3 b)*

introduces the vector $b - a$.

Kernel::Vector_3 *fo.operator()(Origin o, Kernel::Point_3 b)*

introduces the vector b .

Kernel::Vector_3 *fo.operator()(Kernel::Point_3 a, Origin o)*

introduces the vector $-a$.

Kernel::Vector_3 *fo.operator()(Kernel::Segment_3 s)*

introduces the vector $s.target() - s.source()$.

Kernel::Vector_3 *fo.operator()(Kernel::Ray_3 r)*

introduces a vector having the same direction as r .

Kernel::Vector_3 *fo.operator()(Kernel::Line_3 l)*

introduces a vector having the same direction as l .

Kernel::Vector_3 *fo.operator()(Null_vector NULL_VECTOR)*

introduces a null vector .

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Vector_3<Kernel> page [116](#)
Kernel::ConstructScaledVector_3 page [329](#)

Kernel::ConstructVertex_2

A model for this must provide:

Kernel::Point_2 *fo.operator()(Kernel::Segment_2 s, int i)*

returns source or target of *s*: *fo(s,0)* returns the source of *s*, *fo(s,1)* returns the target of *s*. The parameter *i* is taken modulo 2.

Kernel::Point_2 *fo.operator()(Kernel::Iso_rectangle_2 r, int i)*

returns the *i*'th vertex of *r* in counterclockwise order, starting with the lower left vertex. The parameter *i* is taken modulo 4.

Kernel::Point_2 *fo.operator()(Kernel::Triangle_2 t, int i)*

returns the *i*'th vertex of *t*. The parameter *i* is taken modulo 3.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Iso_rectangle_2<Kernel> page [69](#)
CGAL::Segment_2<Kernel> page [81](#)
CGAL::Triangle_2<Kernel> page [83](#)

Kernel::ConstructVertex_3

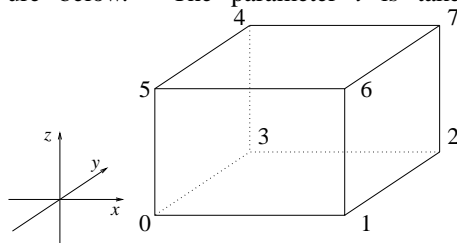
A model for this must provide:

Kernel::Point_3 *fo.operator() (Kernel::Segment_3 s, int i)*

returns source or target of s : $fo(s,0)$ returns the source of s , $fo(s,1)$ returns the target of s . The parameter i is taken modulo 2.

Kernel::Point_3 *fo.operator() (Kernel::Iso_cuboid_3 c, int i)*

returns the i 'th vertex of c , as indicated in the figure below. The parameter i is taken modulo 8.



Kernel::Point_3 *fo.operator() (Kernel::Triangle_3 t, int i)*

returns the i 'th vertex of t . The parameter i is taken modulo 3.

Kernel::Point_3 *fo.operator() (Kernel::Tetrahedron_3 t, int i)*

returns the i 'th vertex of t . The parameter i is taken modulo 4.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Iso_cuboid_3<Kernel> page [94](#)
CGAL::Segment_3<Kernel> page [107](#)
CGAL::Tetrahedron_3<Kernel> page [112](#)
CGAL::Triangle_3<Kernel> page [114](#)

Kernel::CartesianConstIterator_2

A type representing an iterator to the Cartesian coordinates of a point in two dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

Kernel::ConstructCartesianConstIterator_2 page [271](#)

Kernel::CartesianConstIterator_3

A type representing an iterator to the Cartesian coordinates of a point in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

Kernel::ConstructCartesianConstIterator_3 page [272](#)

Kernel::CoplanarOrientation_3

A model for this must provide:

Orientation `fo.operator()(Kernel::Point_3 p,
Kernel::Point_3 q,
Kernel::Point_3 r,
Kernel::Point_3 s)`

Let P be the plane defined by the points p , q , and r . Note that the order defines the orientation of P . The function computes the orientation of points p , q , and s in P : Iff p , q , s are collinear, *COLLINEAR* is returned. Iff P and the plane defined by p , q , and s have the same orientation, *POSITIVE* is returned; otherwise *NEGATIVE* is returned. *Precondition*: p , q , r , and s are coplanar and p , q , and r are not collinear.

Orientation `fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)`

If p, q, r are collinear, then *COLLINEAR* is returned. If not, then p, q, r define a plane P . The return value in this case is either *POSITIVE* or *NEGATIVE*, but we don't specify it explicitly. However, we guarantee that all calls to this predicate over 3 points in P will return a coherent orientation if considered a 2D orientation in P .

Refines

AdaptableFunctor (with four arguments)

See Also

CGAL::coplanar_orientationpage [162](#)

$$\text{Bounded_side} \quad fo.operator() (Kernel::Point_3 p, \\ Kernel::Point_3 q, \\ Kernel::Point_3 r, \\ Kernel::Point_3 s)$$

Precondition: p , q , r , and s are coplanar and p , q , and r are not collinear.

AdaptableFunctor (with four arguments)

CGAL::coplanar_side_of_bounded_circle page 163

Kernel::Coplanar_3

A model for this must provide:

```
bool                                fo.operator()( Kernel::Point_3 p,
                                                    Kernel::Point_3 q,
                                                    Kernel::Point_3 r,
                                                    Kernel::Point_3 s)

                                returns true, if p, q, r, and s are coplanar.
```

Refines

AdaptableFunctor (with four arguments)

See Also

CGAL::coplanarpage [161](#)

Kernel::CounterclockwiseInBetween_2

A model for this must provide:

```
bool                                fo.operator()( Kernel::Direction_2 d,
                                                    Kernel::Direction_2 d1,
                                                    Kernel::Direction_2 d2)
```

returns *true* iff *d* is not equal to *d1*, and while rotating counterclockwise starting at *d1*, *d* is reached strictly before *d2* is reached. Note that true is returned if *d1* == *d2*, unless also *d* == *d1*.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::Direction_2<Kernel>page [67](#)

Kernel::Direction_2

A type representing directions in two dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

<i>CGAL::Direction_2<Kernel></i>	page 67
Kernel::CompareAngleWithXAxis_2	page 228
Kernel::ComputeDx_2	page ??
Kernel::ComputeDy_2	page ??
Kernel::ConstructDirection_2	page 282
Kernel::ConstructOppositeDirection_2	page 298
Kernel::ConstructPerpendicularDirection_2	page 312
Kernel::CounterclockwiseInBetween_2	page 350
Kernel::Equal_2	page 361

Kernel::Direction_3

A type representing directions in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

CGAL::Direction_3<Kernel> page [92](#)
Kernel::ConstructDirection_3 page [283](#)
Kernel::ConstructOppositeDirection_3 page [299](#)
Kernel::Equal_2 page [361](#)

Kernel::DoIntersect_2

A model for this must provide

bool *fo.operator()(Type1 obj1, Type2 obj2)*

determines if two geometrical objects of type *Type1* and *Type2* intersect or not

for all pairs *Type1* and *Type2*, where the types *Type1* and *Type2* can be any of the following:

- *Kernel::Point_2*
- *Kernel::Line_2*
- *Kernel::Ray_2*
- *Kernel::Segment_2*
- *Kernel::Triangle_2*
- *Kernel::Iso_rectangle_2*

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::do_intersect page [943](#)

Kernel::DoIntersect_3

A model for this must provide

bool *fo.operator()(Type1 obj1, Type2 obj2)* determines if two geometrical objects of type *Type1* and *Type2* intersect or not

for all pairs *Type1* and *Type2*, where the type *Type1* is *Kernel::Plane_3* or *Kernel::Triangle_3* and *Type2* can be any of the following:

- *Kernel::Plane_3*
- *Kernel::Line_3*
- *Kernel::Ray_3*
- *Kernel::Segment_3*
- *Kernel::Triangle_3*

And also for *Type1* of type *Triangle_3<Kernel>* and *Type2* of type *Tetrahedron_3<Kernel>*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::do_intersect [page 943](#)

Kernel::EqualXY_3

A model for this must provide:

bool *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

returns true iff *p* and *q* have the same Cartesian *x*-coordinate and the same Cartesian *y*-coordinate.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_xy page [151](#)

Kernel::EqualX_2

A model for this must provide:

bool *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*
returns true iff *p* and *q* have the same Cartesian *x*-coordinate.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::x_equalpage [204](#)

Kernel::EqualX_3

A model for this must provide:

bool *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*
 returns true iff *p* and *q* have the same Cartesian *x*-coordinate.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::x_equalpage [204](#)

Kernel::EqualY_2

A model for this must provide:

bool *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*
 returns true iff *p* and *q* have the same Cartesian y-coordinate.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::y_equalpage [205](#)

Kernel::EqualY_3

A model for this must provide:

bool *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*
 returns true iff *p* and *q* have the same Cartesian y-coordinate.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::y_equalpage [205](#)

Kernel::EqualZ_3

A model for this must provide:

bool *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*
 returns true iff *p* and *q* have the same Cartesian *z*-coordinate.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::z_equal page [206](#)

Kernel::Equal_2

A model for this must provide the following operations. For all of them $fo(x,y)$ returns true iff x and y are equal.

```
bool                fo.operator()( Kernel::Point_2 x, Kernel::Point_2 y)

bool                fo.operator()( Kernel::Vector_2 x, Kernel::Vector_2 y)

bool                fo.operator()( Kernel::Direction_2 x, Kernel::Direction_2 y)

bool                fo.operator()( Kernel::Line_2 x, Kernel::Line_2 y)

bool                fo.operator()( Kernel::Ray_2 x, Kernel::Ray_2 y)

bool                fo.operator()( Kernel::Segment_2 x, Kernel::Segment_2 y)

bool                fo.operator()( Kernel::Circle_2 x, Kernel::Circle_2 y)

bool                fo.operator()( Kernel::Triangle_2 x, Kernel::Triangle_2 y)

bool                fo.operator()( Kernel::Iso_rectangle_2 x, Kernel::Iso_rectangle_2 y)
```

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Circle_2<Kernel> page [65](#)
CGAL::Direction_2<Kernel> page [67](#)
CGAL::Iso_rectangle_2<Kernel> page [69](#)
CGAL::Line_2<Kernel> page [72](#)
CGAL::Point_2<Kernel> page [75](#)
CGAL::Ray_2<Kernel> page [79](#)
CGAL::Segment_2<Kernel> page [81](#)
CGAL::Triangle_2<Kernel> page [83](#)
CGAL::Vector_2<Kernel> page [85](#)

Kernel::Equal_3

A model for this must provide the following operations. For all of them *fo(x,y)* returns true iff *x* and *y* are equal.

<i>bool</i>	<i>fo.operator()(Kernel::Point_3 x, Kernel::Point_3 y)</i>
<i>bool</i>	<i>fo.operator()(Kernel::Vector_3 x, Kernel::Vector_3 y)</i>
<i>bool</i>	<i>fo.operator()(Kernel::Direction_3 x, Kernel::Direction_3 y)</i>
<i>bool</i>	<i>fo.operator()(Kernel::Line_3 x, Kernel::Line_3 y)</i>
<i>bool</i>	<i>fo.operator()(Kernel::Plane_3 x, Kernel::Plane_3 y)</i>
<i>bool</i>	<i>fo.operator()(Kernel::Ray_3 x, Kernel::Ray_3 y)</i>
<i>bool</i>	<i>fo.operator()(Kernel::Segment_3 x, Kernel::Segment_3 y)</i>
<i>bool</i>	<i>fo.operator()(Kernel::Sphere_3 x, Kernel::Sphere_3 y)</i>
<i>bool</i>	<i>fo.operator()(Kernel::Triangle_3 x, Kernel::Triangle_3 y)</i>
<i>bool</i>	<i>fo.operator()(Kernel::Tetrahedron_3 x, Kernel::Tetrahedron_3 y)</i>
<i>bool</i>	<i>fo.operator()(Kernel::Iso_cuboid_3 x, Kernel::Iso_cuboid_3 y)</i>

Refines

AdaptableFunctor (with two arguments)

See Also

<i>CGAL::Direction_3<Kernel></i>	page 92
<i>CGAL::Iso_cuboid_3<Kernel></i>	page 94
<i>CGAL::Line_3<Kernel></i>	page 97
<i>CGAL::Plane_3<Kernel></i>	page 99
<i>CGAL::Point_3<Kernel></i>	page 102
<i>CGAL::Ray_3<Kernel></i>	page 105
<i>CGAL::Segment_3<Kernel></i>	page 107
<i>CGAL::Sphere_3<Kernel></i>	page 109
<i>CGAL::Tetrahedron_3<Kernel></i>	page 112
<i>CGAL::Triangle_3<Kernel></i>	page 114
<i>CGAL::Vector_3<Kernel></i>	page 116

Kernel::HasOnBoundary_2

A model for this must provide:

bool *fo.operator()(Kernel::Circle_2 c, Kernel::Point_2 p)*
returns true iff *p* lies on the boundary of *c*.

bool *fo.operator()(Kernel::Iso_rectangle_2 i, Kernel::Point_2 p)*
returns true iff *p* lies on the boundary of *i*.

bool *fo.operator()(Kernel::Triangle_2 t, Kernel::Point_2 p)*
returns true iff *p* lies on the boundary of *t*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Circle_2<Kernel>page [65](#)
CGAL::Iso_rectangle_2<Kernel>page [69](#)
CGAL::Triangle_2<Kernel>page [83](#)

Kernel::HasOnBoundary_3

A model for this must provide:

bool *fo.operator()(Kernel::Sphere_3 s, Kernel::Point_3 p)*
returns true iff *p* lies on the boundary of *s*.

bool *fo.operator()(Kernel::Tetrahedron_3 t, Kernel::Point_3 p)*
returns true iff *p* lies on the boundary of *t*.

bool *fo.operator()(Kernel::Iso_cuboid_3 c, Kernel::Point_3 p)*
returns true iff *p* lies on the boundary of *c*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Iso_cuboid_3<Kernel> page [94](#)
CGAL::Sphere_3<Kernel> page [109](#)
CGAL::Tetrahedron_3<Kernel> page [112](#)

Kernel::HasOnBoundedSide_2

A model for this must provide:

bool *fo.operator()(Kernel::Circle_2 c, Kernel::Point_2 p)*
returns true iff *p* lies on the bounded side of *c*.

bool *fo.operator()(Kernel::Iso_rectangle_2 i, Kernel::Point_2 p)*
returns true iff *p* lies on the bounded side of *i*.

bool *fo.operator()(Kernel::Triangle_2 t, Kernel::Point_2 p)*
returns true iff *p* lies on the bounded side of *t*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Circle_2<Kernel> page [65](#)
CGAL::Iso_rectangle_2<Kernel> page [69](#)
CGAL::Triangle_2<Kernel> page [83](#)

Kernel::HasOnBoundedSide_3

A model for this must provide:

bool *fo.operator()(Kernel::Sphere_3 s, Kernel::Point_3 p)*
returns true iff *p* lies on the bounded side of *s*.

bool *fo.operator()(Kernel::Tetrahedron_3 t, Kernel::Point_3 p)*
returns true iff *p* lies on the bounded side of *t*.

bool *fo.operator()(Kernel::Iso_cuboid_3 c, Kernel::Point_3 p)*
returns true iff *p* lies on the bounded side of *c*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Iso_cuboid_3<Kernel> page [94](#)
CGAL::Sphere_3<Kernel> page [109](#)
CGAL::Tetrahedron_3<Kernel> page [112](#)

Kernel::HasOnNegativeSide_2

A model for this must provide:

bool *fo.operator()(Kernel::Circle_2 c, Kernel::Point_2 p)*
returns true iff *p* lies on the negative side of *c*.

bool *fo.operator()(Kernel::Line_2 l, Kernel::Point_2 p)*
returns true iff *p* lies on the negative side of *l* (*l* is considered a halfspace).

bool *fo.operator()(Kernel::Triangle_2 t, Kernel::Point_2 p)*
returns true iff *p* lies on the negative side of *t*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Circle_2<Kernel>page [65](#)
CGAL::Line_2<Kernel> page [72](#)
CGAL::Triangle_2<Kernel>page [83](#)

Kernel::HasOnNegativeSide_3

A model for this must provide:

bool *fo.operator() (Kernel::Plane_3 h, Kernel::Point_3 p)*
returns true iff *p* lies on the negative side of *h* (*h* is considered a halfspace).

bool *fo.operator() (Kernel::Sphere_3 s, Kernel::Point_3 p)*
returns true iff *p* lies on the negative side of *s*.

bool *fo.operator() (Kernel::Tetrahedron_3 t, Kernel::Point_3 p)*
returns true iff *p* lies on the negative side of *t*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Plane_3<Kernel> page [99](#)
CGAL::Sphere_3<Kernel> page [109](#)
CGAL::Tetrahedron_3<Kernel> page [112](#)

Kernel::HasOnPositiveSide_2

A model for this must provide:

bool *fo.operator()(Kernel::Circle_2 c, Kernel::Point_2 p)*
returns true iff *p* lies on the positive side of *c*.

bool *fo.operator()(Kernel::Line_2 l, Kernel::Point_2 p)*
returns true iff *p* lies on the positive side of *l* (*l* is considered a halfspace).

bool *fo.operator()(Kernel::Triangle_2 t, Kernel::Point_2 p)*
returns true iff *p* lies on the positive side of *t*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Circle_2<Kernel>page [65](#)
CGAL::Line_2<Kernel> page [72](#)
CGAL::Triangle_2<Kernel>page [83](#)

Kernel::HasOnPositiveSide_3

A model for this must provide:

bool *fo.operator()(Kernel::Plane_3 h, Kernel::Point_3 p)*
returns true iff *p* lies on the positive side of *h* (*h* is considered a halfspace).

bool *fo.operator()(Kernel::Sphere_3 s, Kernel::Point_3 p)*
returns true iff *p* lies on the positive side of *s*.

bool *fo.operator()(Kernel::Tetrahedron_3 t, Kernel::Point_3 p)*
returns true iff *p* lies on the positive side of *t*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Plane_3<Kernel> page [99](#)
CGAL::Sphere_3<Kernel> page [109](#)
CGAL::Tetrahedron_3<Kernel> page [112](#)

Kernel::HasOnUnboundedSide_2

A model for this must provide:

bool *fo.operator()(Kernel::Circle_2 c, Kernel::Point_2 p)*
returns true iff *p* lies on the unbounded side of *c*.

bool *fo.operator()(Kernel::Iso_rectangle_2 i, Kernel::Point_2 p)*
returns true iff *p* lies on the unbounded side of *i*.

bool *fo.operator()(Kernel::Triangle_2 t, Kernel::Point_2 p)*
returns true iff *p* lies on the unbounded side of *t*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Circle_2<Kernel>page [65](#)
CGAL::Iso_rectangle_2<Kernel>page [69](#)
CGAL::Triangle_2<Kernel>page [83](#)

Kernel::HasOnUnboundedSide_3

A model for this must provide:

bool *fo.operator()(Kernel::Sphere_3 s, Kernel::Point_3 p)*
returns true iff *p* lies on the unbounded side of *s*.

bool *fo.operator()(Kernel::Tetrahedron_3 t, Kernel::Point_3 p)*
returns true iff *p* lies on the unbounded side of *t*.

bool *fo.operator()(Kernel::Iso_cuboid_3 c, Kernel::Point_3 p)*
returns true iff *p* lies on the unbounded side of *c*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Iso_cuboid_3<Kernel> page [94](#)
CGAL::Sphere_3<Kernel> page [109](#)
CGAL::Tetrahedron_3<Kernel> page [112](#)

Kernel::HasOn_2

A model for this must provide:

bool *fo.operator()(Kernel::Line_2 l, Kernel::Point_2 p)*
returns true iff *p* lies on *l*.

bool *fo.operator()(Kernel::Ray_2 r, Kernel::Point_2 p)*
returns true iff *p* lies on *r*.

bool *fo.operator()(Kernel::Segment_2 s, Kernel::Point_2 p)*
returns true iff *p* lies on *s*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Line_2<Kernel> page [72](#)
CGAL::Ray_2<Kernel> page [79](#)
CGAL::Segment_2<Kernel> page [81](#)

Kernel::HasOn_3

A model for this must provide:

bool *fo.operator()(Kernel::Line_3 l, Kernel::Point_3 p)*
returns true iff *p* lies on *l*.

bool *fo.operator()(Kernel::Ray_3 r, Kernel::Point_3 p)*
returns true iff *p* lies on *r*.

bool *fo.operator()(Kernel::Segment_3 s, Kernel::Point_3 p)*
returns true iff *p* lies on *s*.

bool *fo.operator()(Kernel::Plane_3 pl, Kernel::Point_3 p)*
returns true iff *p* lies on *pl*.

bool *fo.operator()(Kernel::Triangle_3 t, Kernel::Point_3 p)*
returns true iff *p* lies on *t*.

Refines

AdaptableFunctor (with two arguments)

See Also

<i>CGAL::Line_3<Kernel></i>	page 97
<i>CGAL::Plane_3<Kernel></i>	page 99
<i>CGAL::Ray_3<Kernel></i>	page 105
<i>CGAL::Segment_3<Kernel></i>	page 107
<i>CGAL::Sphere_3<Kernel></i>	page 109
<i>CGAL::Triangle_3<Kernel></i>	page 114

Kernel::Intersect_2

A model for this must provide

Kernel::Object_2 *fo.operator()(Type1 obj1, Type2 obj2)*

computes the intersection region of two geometrical objects of type *Type1* and *Type2*

for all pairs *Type1* and *Type2*, where the types *Type1* and *Type2* can be any of the following:

- *Kernel::Line_2*
- *Kernel::Ray_2*
- *Kernel::Segment_2*
- *Kernel::Triangle_2*
- *Kernel::Iso_rectangle_2*

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::intersection page [945](#)

Kernel::Intersect_3

A model for this must provide

Kernel::Object_3 *fo.operator()(Type1 obj1, Type2 obj2)*

computes the intersection region of two geometrical objects of type *Type1* and *Type2*

for all pairs *Type1* and *Type2*, where the type *Type1* is *Kernel::Plane_3* and *Type2* can be any of the following:

- *Kernel::Plane_3*
- *Kernel::Line_3*
- *Kernel::Ray_3*
- *Kernel::Segment_3*

Kernel::Object_3 *fo.operator()(Kernel::Plane_3 pl1, Kernel::Plane_3 pl2, Kernel::Plane_3 pl3)*

computes the intersection of three planes. The result can be either a *Kernel::Point_3*, a *Kernel::Line_3*, a *Kernel::Plane_3*, or empty.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::intersection page [945](#)

Kernel::IsDegenerate_2

A model for this must provide:

bool *fo.operator()(Kernel::Circle_2 o)*
returns true iff *o* is degenerate.

bool *fo.operator()(Kernel::Iso_rectangle_2 o)*
returns true iff *o* is degenerate.

bool *fo.operator()(Kernel::Line_2 o)*
returns true iff *o* is degenerate.

bool *fo.operator()(Kernel::Ray_2 o)*
returns true iff *o* is degenerate.

bool *fo.operator()(Kernel::Segment_2 o)*
returns true iff *o* is degenerate.

bool *fo.operator()(Kernel::Triangle_2 o)*
returns true iff *o* is degenerate.

Refines

AdaptableFunctor (with one argument)

See Also

<i>CGAL::Circle_2<Kernel></i>	page 65
<i>CGAL::Iso_rectangle_2<Kernel></i>	page 69
<i>CGAL::Line_2<Kernel></i>	page 72
<i>CGAL::Ray_2<Kernel></i>	page 79
<i>CGAL::Segment_2<Kernel></i>	page 81
<i>CGAL::Triangle_2<Kernel></i>	page 83

Kernel::IsDegenerate_3

A model for this must provide:

bool *fo.operator()(Kernel::Iso_cuboid_3 o)*
returns true iff *o* is degenerate.

bool *fo.operator()(Kernel::Line_3 o)*
returns true iff *o* is degenerate.

bool *fo.operator()(Kernel::Plane_3 o)*
returns true iff *o* is degenerate.

bool *fo.operator()(Kernel::Ray_3 o)*
returns true iff *o* is degenerate.

bool *fo.operator()(Kernel::Segment_3 o)*
returns true iff *o* is degenerate.

bool *fo.operator()(Kernel::Sphere_3 o)*
returns true iff *o* is degenerate.

bool *fo.operator()(Kernel::Tetrahedron_3 o)*
returns true iff *o* is degenerate.

bool *fo.operator()(Kernel::Triangle_3 o)*
returns true iff *o* is degenerate.

Refines

AdaptableFunctor (with one argument)

See Also

<i>CGAL::Iso_cuboid_3<Kernel></i>	page 94
<i>CGAL::Line_3<Kernel></i>	page 97
<i>CGAL::Plane_3<Kernel></i>	page 99
<i>CGAL::Point_3<Kernel></i>	page 102
<i>CGAL::Ray_3<Kernel></i>	page 105
<i>CGAL::Segment_3<Kernel></i>	page 107
<i>CGAL::Sphere_3<Kernel></i>	page 109
<i>CGAL::Tetrahedron_3<Kernel></i>	page 112
<i>CGAL::Triangle_3<Kernel></i>	page 114

Kernel::IsHorizontal_2

A model for this must provide:

bool *fo.operator()(Kernel::Line_2 o)*
returns true iff *o* is horizontal.

bool *fo.operator()(Kernel::Ray_2 o)*
returns true iff *o* is horizontal.

bool *fo.operator()(Kernel::Segment_2 o)*
returns true iff *o* is horizontal.

Refines

AdapatableFunctor (with one argument)

See Also

CGAL::Line_2<Kernel> page [72](#)
CGAL::Ray_2<Kernel> page [79](#)
CGAL::Segment_2<Kernel> page [81](#)

Kernel::IsoCuboid_3

A type representing isocuboids in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

<i>CGAL::Iso_cuboid_3<Kernel></i>	page 94
Kernel::BoundedSide_3	page 219
Kernel::ComputeVolume_3	page 258
Kernel::ConstructIsoCuboid_3	page 284
Kernel::ConstructVertex_3	page 344
Kernel::Equal_2	page 361
Kernel::HasOnBoundary_3	page 364
Kernel::HasOnBoundedSide_3	page 366
Kernel::HasOnUnboundedSide_3	page 372
Kernel::IsDegenerate_3	page 378

Kernel::IsoRectangle_2

A type representing isorectangles in two dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

<i>CGAL::Iso_rectangle_2<Kernel></i>	page 69
Kernel::ConstructIsoRectangle_2	page 285
Kernel::ComputeXmin_2	page 261
Kernel::ComputeXmax_2	page 263
Kernel::ComputeYmin_2	page 262
Kernel::ComputeYmax_2	page 264
Kernel::BoundedSide_2	page 218
Kernel::ComputeArea_2	page 247
Kernel::ConstructIsoRectangle_2	page 285
Kernel::ConstructVertex_2	page 343
Kernel::DoIntersect_2	page 353
Kernel::Equal_2	page 361
Kernel::HasOnBoundary_2	page 363
Kernel::HasOnBoundedSide_2	page 365
Kernel::HasOnUnboundedSide_2	page 371
Kernel::Intersect_2	page 375
Kernel::IsDegenerate_2	page 377

Kernel::IsVertical_2

A model for this must provide:

bool *fo.operator()(Kernel::Line_2 o)*
returns true iff *o* is vertical.

bool *fo.operator()(Kernel::Ray_2 o)*
returns true iff *o* is vertical.

bool *fo.operator()(Kernel::Segment_2 o)*
returns true iff *o* is vertical.

Refines

AdapatableFunctor (with one argument)

See Also

CGAL::Line_2<Kernel> page [72](#)
CGAL::Ray_2<Kernel> page [79](#)
CGAL::Segment_2<Kernel> page [81](#)

Kernel::LeftTurn_2

A model for this must provide:

bool *fo.operator()*(*Kernel::Point_2* *p*, *Kernel::Point_2* *q*, *Kernel::Point_2* *r*)
returns *true*, iff the three points *p*, *q* and *r* form a left turn.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::left_turn page [177](#)

Kernel::LessDistanceToPoint_2

A model for this must provide:

bool *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q, Kernel::Point_2 r)*
 returns true iff the distance of *q* to *p* is smaller than the distance of *r* to *p*.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::has_smaller_distance_to_point page [171](#)

Kernel::LessDistanceToPoint_3

A model for this must provide:

bool *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q, Kernel::Point_3 r)*
 returns true iff the distance of *q* to *p* is smaller than the distance of *r* to *p*.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::has_smaller_distance_to_point page [171](#)

Kernel::LessRotateCCW_2

A model for this must provide:

bool *fo.operator()*(*Kernel::Point_2* *p*, *Kernel::Point_2* *q*, *Kernel::Point_2* *r*)

returns true iff the three points *p*, *q* and *r* form a left turn or if they are collinear and the distance of *q* to *p* is larger than the distance of *r* to *p*, where *p* is the point passed to the object at construction.

Precondition: *p* does not lie in the interior of the segment *rq*, i.e. *p* is an extreme point with respect to $\{p, q, r\}$.

Refines

AdaptableFunctor (with three arguments)

Kernel::LessSignedDistanceToLine_2

A model for this must provide:

bool *fo.operator()*(*Kernel::Line_2* *l*, *Kernel::Point_2* *p*, *Kernel::Point_2* *q*)
returns *true* if the signed distance from *p* and the oriented line *l* is smaller than the signed distance of *q* and *l*.

bool *fo.operator()*(*Kernel::Point_2* *p*,
Kernel::Point_2 *q*,
Kernel::Point_2 *r*,
Kernel::Point_2 *s*)
returns *true* if the signed distance from *r* and the oriented line *l* defined by *p* and *q* is smaller than the signed distance of *s* and *l*.
Precondition: *p*! = *q*.

Refines

AdaptableFunctor (with four arguments)

See Also

CGAL::has_smaller_signed_distance_to_line page [172](#)

Kernel::LessSignedDistanceToPlane_3

A model for this must provide:

bool *fo.operator()*(*Kernel::Plane_3* *p*, *Kernel::Point_3* *q*, *Kernel::Point_3* *r*)

returns true, iff the signed distance from point *q* to plane *p* is smaller than the signed distance from point *r* to *p*.

bool *fo.operator()*(*Kernel::Point_3* *p1*,
Kernel::Point_3 *p2*,
Kernel::Point_3 *p3*,
Kernel::Point_3 *q*,
Kernel::Point_3 *r*)

returns true, iff the signed distance from point *q* to the plane *p* defined by *p1*, *p2*, *p3* is smaller than the signed distance from point *r* to *p*.
Precondition: *p*, *q*, and *r* are not collinear.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::has_smaller_signed_distance_to_plane page [173](#)

Kernel::LessXYZ_3

A model for this must provide:

bool *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

returns true iff the x -coordinate of p is smaller than the x -coordinate of q or if they are the same and the y -coordinate of p is smaller than the y -coordinate of q , or, if both x - and y -coordinate are identical and the z -coordinate of p is smaller than the z -coordinate of q .

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::lexicographically_xyz_smaller [page 178](#)

Kernel::LessXY_2

A model for this must provide:

bool *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*

returns true iff the x -coordinate of p is smaller than the x -coordinate of q or if they are the same and the y -coordinate of p is smaller than the y -coordinate of q .

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::lexicographically_xy_smaller page [182](#)

Kernel::LessXY_3

A model for this must provide:

bool *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

returns true iff the x -coordinate of p is smaller than the x -coordinate of q or if they are the same and the y -coordinate of p is smaller than the y -coordinate of q .

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_xy page [151](#)

Kernel::LessX_2

A model for this must provide:

bool *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*
 returns true iff the *x*-coordinate of *p* is smaller than the
x-coordinate of *q*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_x page [149](#)

Kernel::LessX_3

A model for this must provide:

bool *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*

returns true iff the x -coordinate of p is smaller than the x -coordinate of q .

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_x [page 149](#)

Kernel::LessYX_2

A model for this must provide:

bool *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*

returns true iff the y -coordinate of p is smaller than the y -coordinate of q or if they are the same and the x -coordinate of p is smaller than the x -coordinate of q .

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_yx page [159](#)

Kernel::LessY_2

A model for this must provide:

bool *fo.operator()(Kernel::Point_2 p, Kernel::Point_2 q)*
 returns true iff the y-coordinate of *p* is smaller than the
 y-coordinate of *q*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_y [page 155](#)

Kernel::LessY_3

A model for this must provide:

bool

fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)

returns true iff the y-coordinate of *p* is smaller than the y-coordinate of *q*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_y.....page [155](#)

Kernel::LessZ_3

A model for this must provide:

bool *fo.operator()(Kernel::Point_3 p, Kernel::Point_3 q)*
 returns true iff the *z*-coordinate of *p* is smaller than the
z-coordinate of *q*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::compare_z [page 160](#)

Kernel::Line_2

A type representing straight lines (and halfspaces) in two dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

<i>CGAL::Line_2<Kernel></i>	page 72
Kernel::CompareXAtY_2	page 232
Kernel::ComputeSquaredDistance_2	page 252
Kernel::CompareYAtX_2	page 239
Kernel::ConstructBisector_2	page 269
Kernel::ConstructDirection_2	page 282
Kernel::ConstructLine_2	page 287
Kernel::ConstructOppositeLine_2	page 300
Kernel::ConstructPerpendicularLine_2	page 313
Kernel::ConstructPointOn_2	page 319
Kernel::ConstructProjectedPoint_2	page 323
Kernel::DoIntersect_2	page 353
Kernel::Equal_2	page 361
Kernel::HasOnNegativeSide_2	page 367
Kernel::HasOnPositiveSide_2	page 369
Kernel::HasOn_2	page 373
Kernel::Intersect_2	page 375
Kernel::IsDegenerate_2	page 377
Kernel::IsHorizontal_2	page 380
Kernel::IsVertical_2	page 383
Kernel::OrientedSide_2	page 405

Kernel::Line_3

A type representing straight lines in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

<i>CGAL::Line_3<Kernel></i>	page 97
Kernel::ComputeSquaredDistance_3	page 253
Kernel::ConstructDirection_3	page 283
Kernel::ConstructLine_3	page 288
Kernel::ConstructOppositeLine_3	page 301
Kernel::ConstructPerpendicularLine_3	page 314
Kernel::ConstructPlane_3	page 317
Kernel::ConstructPointOn_3	page 320
Kernel::ConstructProjectedPoint_3	page 324
Kernel::DoIntersect_3	page 354
Kernel::Equal_3	page 362
Kernel::HasOn_3	page 374
Kernel::Intersect_3	page 376
Kernel::IsDegenerate_3	page 378

Kernel::Object_2

A type representing different types of objects in two dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

CGAL::Object page [120](#)
Kernel::Assign_2 page [216](#)
Kernel::ConstructObject_2 page [295](#)
Kernel::Intersect_2 page [375](#)

Kernel::Object_3

A type representing different types of objects in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

CGAL::Object page [120](#)
Kernel::Assign_3 page [217](#)
Kernel::ConstructObject_3 page [296](#)
Kernel::Intersect_3 page [376](#)

Kernel::Orientation_2

A model for this must provide:

Orientation $fo.operator()(\text{Kernel::Point_2 } p, \text{Kernel::Point_2 } q, \text{Kernel::Point_2 } r)$

returns *LEFT_TURN*, if r lies to the left of the oriented line l defined by p and q , returns *RIGHT_TURN* if r lies to the right of l , and returns *COLLINEAR* if r lies on l .

Orientation $fo.operator()(\text{Kernel::Vector_2 } u, \text{Kernel::Vector_2 } v)$

returns *LEFT_TURN* if u and v form a left turn, returns *RIGHT_TURN* if u and v form a right turn, and returns *COLLINEAR* if u and v are collinear.

Refines

AdaptableFunctor (with three arguments)

See Also

CGAL::orientationpage [494](#)

Kernel::Orientation_3

A model for this must provide:

Orientation *fo.operator()(Kernel::Point_3 p,*
 Kernel::Point_3 q,
 Kernel::Point_3 r,
 Kernel::Point_3 s)

returns *POSITIVE*, if *s* lies on the positive side of the oriented plane *h* defined by *p*, *q*, and *r*, returns *NEGATIVE* if *s* lies on the negative side of *h*, and returns *COPLANAR* if *s* lies on *h*.

Orientation *fo.operator()(Kernel::Vector_3 u, Kernel::Vector_3 v, Kernel::Vector_3 w)*

returns *POSITIVE* if *u*, *v* and *w* are positively oriented, returns *NEGATIVE* if *u*, *v* and *w* are negatively oriented, and returns *COPLANAR* if *u*, *v* and *w* are coplanar.

Refines

AdaptableFunctor (with four arguments)

See Also

CGAL::orientationpage [494](#)

Kernel::OrientedSide_2

A model for this must provide:

Oriented_side *fo.operator()(Kernel::Circle_2 c, Kernel::Point_2 p)*

returns *ON_ORIENTED_BOUNDARY*, *ON_NEGATIVE_SIDE*, or the constant *ON_POSITIVE_SIDE*, depending on the position of *p* relative to the oriented circle *c*.

Oriented_side *fo.operator()(Kernel::Line_2 l, Kernel::Point_2 p)*

returns *ON_ORIENTED_BOUNDARY*, *ON_NEGATIVE_SIDE*, or the constant *ON_POSITIVE_SIDE*, depending on the position of *p* relative to the oriented line *l*.

Oriented_side *fo.operator()(Kernel::Triangle_2 t, Kernel::Point_2 p)*

returns *ON_ORIENTED_BOUNDARY*, *ON_NEGATIVE_SIDE*, or the constant *ON_POSITIVE_SIDE*, depending on the position of *p* relative to the oriented triangle *t*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Circle_2<Kernel>page [65](#)
CGAL::Line_2<Kernel> page [72](#)
CGAL::Triangle_2<Kernel>page [83](#)

Kernel::OrientedSide_3

A model for this must provide:

Oriented_side *fo.operator()(Kernel::Plane_3 h, Kernel::Point_3 p)*

returns *ON_ORIENTED_BOUNDARY*, *ON_NEGATIVE_SIDE*, or the constant *ON_POSITIVE_SIDE*, depending on the position of *p* relative to the oriented plane *h*.

Oriented_side *fo.operator()(Kernel::Tetrahedron_3 t, Kernel::Point_3 p)*

returns *ON_ORIENTED_BOUNDARY*, *ON_NEGATIVE_SIDE*, or the constant *ON_POSITIVE_SIDE*, depending on the position of *p* relative to the oriented tetrahedron *t*.

Oriented_side *fo.operator()(Kernel::Sphere_3 s, Kernel::Point_3 p)*

returns *ON_ORIENTED_BOUNDARY*, *ON_NEGATIVE_SIDE*, or the *ON_POSITIVE_SIDE*, depending on the position of *p* relative to the oriented sphere *s*.

Refines

AdaptableFunctor (with two arguments)

See Also

CGAL::Plane_3<Kernel> page [99](#)
CGAL::Sphere_3<Kernel> page [109](#)
CGAL::Tetrahedron_3<Kernel> page [112](#)

Kernel::Plane_3

A type representing planes (and halfspaces) in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

<i>CGAL::Plane_3<Kernel></i>	page 99
Kernel::ComputeSquaredDistance_3	page 253
Kernel::ConstructBaseVector_3	page 266
Kernel::ConstructBisector_3	page 270
Kernel::ConstructLiftedPoint_3	page 286
Kernel::ConstructOppositePlane_3	page 302
Kernel::ConstructOrthogonalVector_3	page 311
Kernel::ConstructPerpendicularLine_3	page 314
Kernel::ConstructPerpendicularPlane_3	page 315
Kernel::ConstructPlane_3	page 317
Kernel::ConstructPointOn_3	page 320
Kernel::ConstructProjectedPoint_3	page 324
Kernel::ConstructProjectedXYPoint_2	page 325
Kernel::DoIntersect_3	page 354
Kernel::Equal_3	page 362
Kernel::HasOnNegativeSide_3	page 368
Kernel::HasOnPositiveSide_3	page 370
Kernel::HasOn_3	page 374
Kernel::Intersect_3	page 376
Kernel::IsDegenerate_3	page 378
Kernel::LessSignedDistanceToPlane_3	page 389
Kernel::OrientedSide_3	page 406

Kernel::Point_2

A type representing points in two dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

Kernel::Angle_2	page 208
Kernel::AreOrderedAlongLine_2	page 210
Kernel::AreStrictlyOrderedAlongLine_2	page 214
Kernel::Collinear_2	page 226
Kernel::CollinearAreOrderedAlongLine_2	page 221
Kernel::CollinearAreStrictlyOrderedAlongLine_2	page 223
Kernel::CompareDistance_2	page 229
Kernel::CompareXAtY_2	page 232
Kernel::CompareXY_2	page 235
Kernel::CompareX_2	page 237
Kernel::CompareYAtX_2	page 239
Kernel::CompareY_2	page 241
Kernel::ComputeSquaredDistance_2	page 252
Kernel::ComputeSquaredRadius_2	page 256
Kernel::ComputeX_2	page 259
Kernel::ComputeY_2	page 260
Kernel::ComputeHx_2	page ??
Kernel::ComputeHy_2	page ??
Kernel::ConstructBisector_2	page 269
Kernel::ConstructCircumcenter_2	page 278
Kernel::ConstructLiftedPoint_3	page 286
Kernel::ConstructMidpoint_2	page 291
Kernel::ConstructPointOn_2	page 319
Kernel::ConstructPoint_2	page 321
Kernel::ConstructProjectedPoint_2	page 323
Kernel::ConstructProjectedXYPoint_2	page 325
Kernel::ConstructTranslatedPoint_2	page 337
Kernel::DoIntersect_2	page 353
Kernel::Equal_2	page 361
Kernel::EqualX_2	page 356
Kernel::EqualY_2	page 358
Kernel::LeftTurn_2	page 384
Kernel::LessDistanceToPoint_2	page 385
Kernel::LessRotateCCW_2	page 387
Kernel::LessSignedDistanceToLine_2	page 388
Kernel::LessX_2	page 393
Kernel::LessXY_2	page 391
Kernel::LessY_2	page 396
Kernel::LessYX_2	page 395
Kernel::Orientation_2	page 403
Kernel::SideOfBoundedCircle_2	page 416

Kernel::SideOfOrientedCircle_2.....page [418](#)

Kernel::Point_3

A type representing points in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

Kernel::Angle_3	page 209
Kernel::AreOrderedAlongLine_3	page 211
Kernel::AreStrictlyOrderedAlongLine_3	page 215
Kernel::Collinear_3	page 227
Kernel::CollinearAreOrderedAlongLine_3	page 222
Kernel::CollinearAreStrictlyOrderedAlongLine_3	page 224
Kernel::CompareDistance_3	page 230
Kernel::CompareXYZ_3	page 234
Kernel::CompareXY_3	page 236
Kernel::CompareX_3	page 238
Kernel::CompareY_3	page 242
Kernel::CompareZ_3	page 243
Kernel::ComputeSquaredDistance_3	page 253
Kernel::ComputeSquaredRadius_3	page 257
Kernel::ConstructBisector_3	page 270
Kernel::ConstructCentroid_3	page 276
Kernel::ConstructCircumcenter_3	page 279
Kernel::ConstructLiftedPoint_3	page 286
Kernel::ConstructMidpoint_3	page 292
Kernel::ConstructPointOn_3	page 320
Kernel::ConstructPoint_3	page 322
Kernel::ConstructProjectedPoint_3	page 324
Kernel::ConstructTranslatedPoint_3	page 338
Kernel::CoplanarOrientation_3	page 347
Kernel::CoplanarSideOfBoundedCircle_3	page 348
Kernel::Coplanar_3	page 349
Kernel::EqualXY_3	page 355
Kernel::EqualX_3	page 357
Kernel::EqualY_3	page 359
Kernel::EqualZ_3	page 360
Kernel::Equal_2	page 361
Kernel::LessDistanceToPoint_3	page 386
Kernel::LessSignedDistanceToPlane_3	page 389
Kernel::LessXYZ_3	page 390
Kernel::LessXY_3	page 392
Kernel::LessX_3	page 394
Kernel::LessY_3	page 397
Kernel::LessZ_3	page 398
Kernel::Orientation_3	page 404
Kernel::SideOfBoundedSphere_3	page 417

Kernel::SideOfOrientedSphere_3 page [419](#)

Kernel::Ray_2

A type representing rays in two dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

<i>CGAL::Ray_2<Kernel></i>	page 79
Kernel::CollinearHasOn_2	page 225
Kernel::ComputeSquaredDistance_2	page 252
Kernel::ConstructDirection_2	page 282
Kernel::ConstructLine_2	page 287
Kernel::ConstructOppositeRay_2	page 303
Kernel::ConstructPointOn_2	page 319
Kernel::ConstructRay_2	page 326
Kernel::ConstructSource_2	page ??
Kernel::ConstructSecondPoint_2	page ??
Kernel::DoIntersect_2	page 353
Kernel::Equal_2	page 361
Kernel::HasOn_2	page 373
Kernel::Intersect_2	page 375
Kernel::IsDegenerate_2	page 377
Kernel::IsHorizontal_2	page 380
Kernel::IsVertical_2	page 383

Kernel::Ray_3

A type representing rays in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

<i>CGAL::Ray_3<Kernel></i>	page 105
Kernel::ComputeSquaredDistance_3	page 253
Kernel::ConstructDirection_3	page 283
Kernel::ConstructLine_3	page 288
Kernel::ConstructOppositeRay_3	page 304
Kernel::ConstructPlane_3	page 317
Kernel::ConstructPointOn_3	page 320
Kernel::ConstructRay_3	page 327
Kernel::DoIntersect_3	page 354
Kernel::Equal_3	page 362
Kernel::HasOn_3	page 374
Kernel::Intersect_3	page 376
Kernel::IsDegenerate_3	page 378

Kernel::Segment_2

A type representing segments in two dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

<i>CGAL::Segment_2<Kernel></i>	page 81
Kernel::CollinearHasOn_2	page 225
Kernel::ComputeSquaredDistance_2	page 252
Kernel::ComputeSquaredLength_2	page 254
Kernel::ConstructDirection_2	page 282
Kernel::ConstructLine_2	page 287
Kernel::ConstructOppositeSegment_2	page 305
Kernel::ConstructPointOn_2	page 319
Kernel::ConstructSegment_2	page 331
Kernel::ConstructSource_2	page ??
Kernel::ConstructTarget_2	page ??
Kernel::ConstructVertex_2	page 343
Kernel::DoIntersect_2	page 353
Kernel::Equal_2	page 361
Kernel::HasOn_2	page 373
Kernel::Intersect_2	page 375
Kernel::IsDegenerate_2	page 377
Kernel::IsHorizontal_2	page 380
Kernel::IsVertical_2	page 383

Kernel::Segment_3

A type representing segments in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

<i>CGAL::Segment_3<Kernel></i>	page 107
Kernel::ComputeSquaredDistance_3	page 253
Kernel::ComputeSquaredLength_3	page 255
Kernel::ConstructDirection_3	page 283
Kernel::ConstructLine_3	page 288
Kernel::ConstructOppositeSegment_3	page 306
Kernel::ConstructPlane_3	page 317
Kernel::ConstructPointOn_3	page 320
Kernel::ConstructSegment_3	page 332
Kernel::ConstructVertex_3	page 344
Kernel::DoIntersect_3	page 354
Kernel::Equal_3	page 362
Kernel::HasOn_3	page 374
Kernel::Intersect_3	page 376
Kernel::IsDegenerate_3	page 378

```
Bounded_side      fo.operator()( Kernel::Point_2 p,
                                   Kernel::Point_2 q,
                                   Kernel::Point_2 r,
                                   Kernel::Point_2 t)
```

Precondition: p, q and r are not collinear.

returns the position of the point t relative to the circle that has pq as its diameter.

Refines

AdaptableFunctor (with four arguments)

See Also

CGAL::side_of_bounded_circle.....page 197

Kernel::SideOfBoundedSphere_3

A model for this must provide:

```
Bounded_side      fo.operator()( Kernel::Point_3 p,
                                   Kernel::Point_3 q,
                                   Kernel::Point_3 r,
                                   Kernel::Point_3 s,
                                   Kernel::Point_3 t)
```

returns the relative position of point t to the sphere defined by p, q, r , and s . The order of the points p, q, r , and s does not matter.

Precondition: p, q, r and s are not coplanar.

<i>Bounded_side</i>	<i>fo.operator()</i> (<i>Kernel::Point_3</i> <i>p</i> , <i>Kernel::Point_3</i> <i>q</i> , <i>Kernel::Point_3</i> <i>r</i> , <i>Kernel::Point_3</i> <i>t</i>)
---------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------

returns the position of the point t relative to the sphere passing through p , q , and r and whose center is in the plane defined by these three points.

<i>Bounded_side</i>	<i>fo.operator()</i> (<i>Kernel::Point_3</i> <i>p</i> , <i>Kernel::Point_3</i> <i>q</i> , <i>Kernel::Point_3</i> <i>t</i>)
---------------------	------------------------------------------------------------------------------------------------------------------------------

returns the position of the point t relative to the sphere that has pq as its diameter.

Refines

AdaptableFunctor (with five arguments)

See Also

CGAL::side_of_bounded_sphere page 496

Kernel::SideOfOrientedSphere_3

A model for this must provide:

<i>Oriented_side</i>	<i>fo.operator()</i> (<i>Kernel::Point_3</i> <i>p</i> , <i>Kernel::Point_3</i> <i>q</i> , <i>Kernel::Point_3</i> <i>r</i> , <i>Kernel::Point_3</i> <i>s</i> , <i>Kernel::Point_3</i> <i>t</i>)
----------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

returns the relative position of point t to the oriented sphere defined by p, q, r and s . The order of the points p, q, r , and s is important, since it determines the orientation of the implicitly constructed sphere. If the points p, q, r and s are positive oriented, positive side is the bounded interior of the sphere.

Precondition: p, q, r and s are not coplanar.

Refines

AdaptableFunctor (with five arguments)

See Also

CGAL::side_of_oriented_sphere page 497

Kernel::Sphere_3

A type representing spheres in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

<i>CGAL::Sphere_3<Kernel></i>	page 109
Kernel::BoundedSide_3	page 219
Kernel::ComputeSquaredRadius_3	page 257
Kernel::ConstructCenter_3	page 274
Kernel::ConstructOppositeSphere_3	page 307
Kernel::ConstructSphere_3	page 333
Kernel::Equal_2	page 361
Kernel::HasOnBoundary_3	page 364
Kernel::HasOnBoundedSide_3	page 366
Kernel::HasOnNegativeSide_3	page 368
Kernel::HasOnPositiveSide_3	page 370
Kernel::HasOnUnboundedSide_3	page 372
Kernel::IsDegenerate_3	page 378
Kernel::OrientedSide_3	page 406

Kernel::Tetrahedron_3

A type representing tetrahedra in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

<i>CGAL::Tetrahedron_3<Kernel></i>	page 112
Kernel::BoundedSide_3	page 219
Kernel::ComputeVolume_3	page 258
Kernel::ConstructCentroid_3	page 276
Kernel::ConstructTetrahedron_3	page 336
Kernel::ConstructVertex_3	page 344
Kernel::Equal_2	page 361
Kernel::HasOnBoundary_3	page 364
Kernel::HasOnBoundedSide_3	page 366
Kernel::HasOnNegativeSide_3	page 368
Kernel::HasOnPositiveSide_3	page 370
Kernel::HasOnUnboundedSide_3	page 372
Kernel::IsDegenerate_3	page 378
Kernel::OrientedSide_3	page 406

Kernel::Triangle_2

A type representing triangles in two dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

<i>CGAL::Triangle_2<Kernel></i>	page 83
Kernel::BoundedSide_2	page 218
Kernel::ComputeArea_2	page 247
Kernel::ComputeSquaredDistance_2	page 252
Kernel::ConstructCentroid_2	page 275
Kernel::ConstructOppositeTriangle_2	page 308
Kernel::ConstructTriangle_2	page 339
Kernel::ConstructVertex_2	page 343
Kernel::DoIntersect_2	page 353
Kernel::Equal_2	page 361
Kernel::HasOnBoundary_2	page 363
Kernel::HasOnBoundedSide_2	page 365
Kernel::HasOnNegativeSide_2	page 367
Kernel::HasOnPositiveSide_2	page 369
Kernel::HasOnUnboundedSide_2	page 371
Kernel::Intersect_2	page 375
Kernel::IsDegenerate_2	page 377
Kernel::OrientedSide_2	page 405

Kernel::Triangle_3

A type representing triangles in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

<i>CGAL::Triangle_3<Kernel></i>	page 114
Kernel::ComputeSquaredArea_3	page 251
Kernel::ConstructCentroid_3	page 276
Kernel::ConstructSupportingPlane_3	page 335
Kernel::ConstructTriangle_3	page 340
Kernel::ConstructVertex_3	page 344
Kernel::DoIntersect_3	page 354
Kernel::Equal_3	page 362
Kernel::HasOn_3	page 374
Kernel::IsDegenerate_3	page 378

Kernel::Vector_2

Definition

A type representing vectors in two dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

<i>CGAL::Vector_2<Kernel></i>	page 85
Kernel::ComputeX_2	page 259
Kernel::ComputeY_2	page 260
Kernel::ComputeHx_2	page ??
Kernel::ComputeHy_2	page ??
Kernel::ConstructDirection_2	page 282
Kernel::ConstructOppositeVector_2	page 309
Kernel::ConstructPerpendicularVector_2	page 316
Kernel::ConstructScaledVector_2	page 328
Kernel::ConstructDividedVector_2	page ??
Kernel::ConstructSumOfVectors_2	page 330
Kernel::ConstructDifferenceOfVectors_2	page 281
Kernel::ConstructVector_2	page 341
Kernel::Equal_2	page 361
Kernel::Orientation_2	page 403

Kernel::Vector_3

Definition

A type representing vectors in three dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

See Also

<i>CGAL::Vector_3<Kernel></i>	page 116
Kernel::ConstructCrossProductVector_3	page 280
Kernel::ConstructDirection_3	page 283
Kernel::ConstructOppositeVector_3	page 310
Kernel::ConstructOrthogonalVector_3	page 311
Kernel::ConstructScaledVector_3	page 329
Kernel::ConstructVector_3	page 342
Kernel::Equal_3	page 362
Kernel::Orientation_3	page 404

2.14 Tag Classes

CGAL::Tag_false

Definition

The class *Tag_false* is used to define a tag indicating, for example, that a certain feature is not available in a class.

```
#include <CGAL/tags.h>
```

```
bool check_tag( Tag_false ) returns false.
```

See Also

CGAL::Tag_truepage [426](#)
CGAL::Number_type_traits<NT> in the Support Library Manual.

CGAL::Tag_true

Definition

The class *Tag_true* is used to define a tag indicating, for example, that a certain feature is available in a class.

```
#include <CGAL/tags.h>
```

```
bool check_tag( Tag_true ) returns true.
```

See Also

CGAL::Tag_falsepage [426](#)
CGAL::Number_type_traits<NT> in the Support Library Manual.

Chapter 3

dD Kernel

Michael Seel

3.1 Introduction

This part of the reference manual covers the higher-dimensional kernel. The kernel contains objects of constant size, such as point, vector, direction, line, ray, segment, circle. With each type comes a set of functions which can be applied to an object of this type. You will typically find access functions (e.g. to the coordinates of a point), tests of the position of a point relative to the object, a function returning the bounding box, the length, or the area of an object, and so on. The CGAL kernel further contains basic operations such as affine transformations, detection and computation of intersections, and distance computations. Note that this section partly recapitulates facts already mentioned for the lower-dimensional kernel.

3.1.1 Robustness

The correctness proof of nearly all geometric algorithms presented in theory papers assumes exact computation with real numbers. This leads to a fundamental problem with the implementation of geometric algorithms. Naively, often the exact real arithmetic is replaced by inexact floating-point arithmetic in the implementation. This often leads to acceptable results for many input data. However, even for the implementation of the simplest geometric algorithms this simplification occasionally does not work. Rounding errors introduced by inaccurate arithmetic may lead to inconsistent decisions, causing unexpected failures for some correct input data. There are many approaches to this problem, one of them is to compute exactly (compute so accurate that all decisions made by the algorithm are exact) which is possible in many cases but more expensive than standard floating-point arithmetic. C. M. Hoffmann [[Hof89a](#), [Hof89b](#)] illustrates some of the problems arising in the implementation of geometric algorithms and discusses some approaches to solve them. A more recent overview is given in [[Sch00](#)]. The exact computation paradigm is discussed by Yap and Dubé [[YD95](#)] and Yap [[Yap97](#)].

In CGAL you can choose the underlying number types and arithmetic. You can use different types of arithmetic simultaneously and the choice can be easily changed, e.g. for testing. So you can choose between implementations with fast but occasionally inexact arithmetic and implementations guaranteeing exact computation and exact results. Of course you have to pay for the exactness in terms of execution time and storage space. See the section on number types in the Support Library for more details on number types and their capabilities and performance.

3.1.2 Genericity

To increase generic usage of objects and predicates the higher-dimensional kernel makes heavy use of iterator ranges as defined in the STL for modelling tuples. Iterators conceptualize C++ pointers.

For an iterator range $[first, last)$ we define $T = tuple [first, last)$ as the ordered tuple $(T[0], T[1], \dots, T[d-1])$ where $S[i] = * + +^{(i)}first$ (the element obtained by i times forwarding the iterator by operator $++$ and then dereferencing it to get the value to which it points). We write $d = size [first, last)$ and $S = set [first, last)$ to denote the unordered set of elements of the corresponding tuple.

This extends the syntax of random access iterators to input iterators. If we index the tuple as above then we require that $++^{(d+1)}first = last$.

3.2 Kernel Representations

Our object of study is the d -dimensional affine Euclidean space, where d is a parameter of our geometry. Objects in that space are sets of points. A common way to represent the points is the use of Cartesian coordinates, which assumes a reference frame (an origin and d orthogonal axes). In that framework, a point is represented by a d -tuple $(c_0, c_1, \dots, c_{d-1})$, and so are vectors in the underlying linear space. Each point is represented uniquely by such Cartesian coordinates.

Another way to represent points is by homogeneous coordinates. In that framework, a point is represented by a $(d+1)$ -tuple (h_0, h_1, \dots, h_d) . Via the formulae $c_i = h_i/h_d$, the corresponding point with Cartesian coordinates $(c_0, c_1, \dots, c_{d-1})$ can be computed. Note that homogeneous coordinates are not unique. For $\lambda \neq 0$, the tuples (h_0, h_1, \dots, h_d) and $(\lambda \cdot h_0, \lambda \cdot h_1, \dots, \lambda \cdot h_d)$ represent the same point. For a point with Cartesian coordinates $(c_0, c_1, \dots, c_{d-1})$ a possible homogeneous representation is $(c_0, c_1, \dots, c_{d-1}, 1)$. Homogeneous coordinates in fact allow to represent objects in a more general space, the projective space \mathbb{P}_d . In CGAL, we do not compute in projective geometry. Rather, we use homogeneous coordinates to avoid division operations, since the additional coordinate can serve as a common denominator.

3.2.1 Genericity through Parameterization

Almost all the kernel objects (and the corresponding functions) are templates with a parameter that allows the user to choose the representation of the kernel objects. A type that is used as an argument for this parameter must fulfill certain requirements on syntax and semantics. The list of requirements defines an abstract kernel concept. In CGAL such a kernel concept is often also called a *representation class* and denoted by R . A representation class provides the actual implementations of the kernel objects. For all kernel objects $Kernel_object$, the types $CGAL::Kernel_object<R>$ and $R::Kernel_object$ are identical.

CGAL offers two families of concrete models for the concept representation class, one based on the Cartesian representation of points and one based on the homogeneous representation of points. The interface of the kernel objects is designed such that it works well with both Cartesian and homogeneous representation, for example, points have a constructor with a range of coordinates plus a common denominator (the $d+1$ homogeneous coordinates of the point). The common interfaces parameterized with a representation class allow one to develop code independent of the chosen representation. We said “families” of models, because both families are parameterized too. A user can choose the number type used to represent the coordinates and the linear algebra module used to calculate the result of predicates and constructions.

For reasons that will become evident later, a representation class provides two typenames for number types,

namely $R::FT$ and $R::RT$ and one typename for the linear algebra module $R::LA$.¹ The type $R::FT$ must fulfill the requirements on what is called a *field type* in CGAL. This roughly means that $R::FT$ is a type for which operations $+$, $-$, $*$ and $/$ are defined with semantics (approximately) corresponding to those of a field in a mathematical sense. Note that, strictly speaking, the built-in type *int* does not fulfill the requirements on a field type, since *ints* correspond to elements of a ring rather than a field, especially operation $/$ is not the inverse of $*$. The requirements on the type $R::RT$ are weaker. This type must fulfill the requirements on what is called an *Euclidean ring type* in CGAL. This roughly means that $R::RT$ is a type for which operations $+$, $-$, $*$ are defined with semantics (approximately) corresponding to those of a ring in a mathematical sense. A very limited division operation $/$ must be available as well. It must work for exact (i.e., no remainder) integer divisions only. Furthermore, both number types should fulfill CGAL's requirements on a number type.

3.2.2 Cartesian Kernel

With `Cartesian_d<FieldNumberType,LinearAlgebra>` you can choose Cartesian representation of coordinates. The type `LinearAlgebra` must be a linear algebra module working on numbers of type `FieldNumberType`. The second parameter defaults to module delivered with the kernel so for short a user can just write `Cartesian_d<FieldNumberType>` when not providing her own linear algebra.

When you choose Cartesian representation you have to declare at least the type of the coordinates. A number type used with the `Cartesian_d` representation class should be a *field type* as described above. Both `Cartesian<FieldNumberType>::FT` and `Cartesian<FieldNumberType>::RT` are mapped to number type `FieldNumberType`. `Cartesian_d<FieldNumberType,LinearAlgebra>::LA` is mapped to the type `LinearAlgebra`. `Cartesian<FieldNumberType>` uses reference counting internally to save copying costs.

3.2.3 Homogeneous Kernel

As we mentioned before, homogeneous coordinates permit to avoid division operations in numerical computations, since the additional coordinate can serve as a common denominator. Avoiding divisions can be useful for exact geometric computation. With `Homogeneous_d<RingNumberType,LinearAlgebra>` you can choose homogeneous representation of coordinates with the kernel objects. As for Cartesian representation you have to declare at the same time the type used to store the homogeneous coordinates. Since the homogeneous representation allows one to avoid the divisions, the number type associated with a homogeneous representation class must be a model for the weaker concept Euclidean ring type only.

The type `LinearAlgebra` must be a linear algebra module working on numbers of type `RingNumberType`. Again the second parameter defaults to module delivered with the kernel so for short one can just write `Homogeneous_d<RingNumberType>` when replacing the default is no issue.

However, some operations provided by this kernel involve division operations, for example computing squared distances or returning a Cartesian coordinate. To keep the requirements on the number type parameter of `Homogeneous` low, the number type `Quotient<RingNumberType>` is used instead. This number type turns a ring type into a field type. It maintains numbers as quotients, i.e. a numerator and a denominator. Thereby, divisions are circumvented. With `Homogeneous_d<RingNumberType>`, `Homogeneous_d<RingNumberType>::FT` is equal to `Quotient<RingNumberType>` while `Homogeneous_d<RingNumberType>::RT` is equal to `RingNumberType`. `Homogeneous_d<RingNumberType,LinearAlgebra>::LA` is mapped to the type `LinearAlgebra`.

¹The double colon `::` is the C++ scope operator.

3.2.4 Naming conventions

The use of representation classes not only avoids problems, it also makes all CGAL classes very uniform. They **always** consist of:

1. The *capitalized base name* of the geometric object, such as *Point*, *Segment*, *Triangle*.
2. Followed by *_d*.
3. A *representation class* as parameter, which itself is parameterized with a number type, such as *Cartesian_d<double>* or *Homogeneous_d<leda_integer>*.

3.2.5 Kernel as a Traits Class

Algorithms and data structures in the basic library of CGAL are parameterized by a traits class that subsumes the objects on which the algorithm or data structure operates as well as the operations to do so. For most of the algorithms and data structures in the basic library you can use a kernel as a traits class. For some algorithms you even do not have to specify the kernel; it is detected automatically using the types of the geometric objects passed to the algorithm. In some other cases, the algorithms or data structures needs more than is provided by a kernel. In these cases, a kernel can not be used as a traits class.

3.2.6 Choosing a Kernel

If you start with integral Cartesian coordinates, many geometric computations will involve integral numerical values only. Especially, this is true for geometric computations that evaluate only predicates, which are tantamount to determinant computations. Examples are triangulation of point sets and convex hull computation.

The dimension d of our affine space determines the dimension of the matrix computations in the mathematical evaluation of predicates. As rounding errors accumulate fast the homogeneous representation used with multi-precision integers is the kernel of choice for well-behaved algorithms. Note, that unless you use an arbitrary precision integer type, incorrect results might arise due to overflow.

If new points are to be constructed, for example the intersection point of two lines, computation of Cartesian coordinates usually involves divisions, so you need to use a field type with Cartesian representation or have to switch to homogeneous representation. *double* is a possible, but imprecise field type. You can also put any ring type into *Quotient* to get a field type and put it into *Cartesian*, but you better put the ring type into *Homogeneous*. *leda_rational* and *leda_real* are valid field types, too.

Still other people will prefer the built-in type *double*, because they need speed and can live with approximate results, or even algorithms that, from time to time, crash or compute incorrect results due to accumulated rounding errors.

3.2.7 Inclusion of Header Files

You need just to include a representation class to obtain the geometric objects of the kernel that you would like to use with the representation class, i.e., *CGAL/Cartesian_d.h* or *CGAL/Homogeneous_d.h*

3.3 Kernel Geometry

3.3.1 Points and Vectors

In CGAL, we strictly distinguish between points, vectors and directions. A *point* is a point in the Euclidean space \mathbb{E}^d , a *vector* is the difference of two points p_2, p_1 and denotes the direction and the distance from p_1 to p_2 in the vector space \mathbb{R}^d , and a *direction* is a vector where we forget about its length. They are different mathematical concepts. For example, they behave different under affine transformations and an addition of two points is meaningless in affine geometry. By putting them in different classes we not only get cleaner code, but also type checking by the compiler which avoids ambiguous expressions. Hence, it pays twice to make this distinction.

CGAL defines a symbolic constant *ORIGIN* of type *Origin* which denotes the point at the origin. This constant is used in the conversion between points and vectors. Subtracting it from a point p results in the locus vector of p .

```
double coord[] = {1.0, 1.0, 1.0, 1.0};
Point_d< Cartesian_d<double> > p(4, coord, coord+4), q(4);
Vector_d< Cartesian_d<double> > v(4);
v = p - ORIGIN;
q = ORIGIN + v;
assert( p == q );
```

In order to obtain the point corresponding to a vector v you simply have to add v to *ORIGIN*. If you want to determine the point q in the middle between two points p_1 and p_2 , you can write²

```
q = p_1 + (p_2 - p_1) / 2.0;
```

Note that these constructions do not involve any performance overhead for the conversion with the currently available representation classes.

3.3.2 Kernel Objects

Besides points (*Point_d<R>*), vectors (*Vector_d<R>*), and directions (*Direction_d<R>*), CGAL provides lines, rays, segments, hyperplanes, and spheres.

Lines (*Line_d<R>*) in CGAL are oriented. A ray (*Ray_d<R>*) is a semi-infinite interval on a line, and this line is oriented from the finite endpoint of this interval towards any other point in this interval. A segment (*Segment_d<R>*) is a bounded interval on a directed line, and the endpoints are ordered so that they induce the same direction as that of the line.

Hyperplanes are affine subspaces of dimension $d - 1$ in \mathbb{E}^d , passing through d points. Hyperplanes are oriented and partition space into a positive side and a negative side. In CGAL, there are no special classes for halfspaces. Halfspaces are supposed to be represented by oriented hyperplanes. All kernel objects are equality comparable via *operator==* and *operator!=*. For those oriented objects whose orientation can be reversed (segments, lines, hyperplanes, spheres) there is also a global function *weak_equality* that allows to test for point set equality disregarding the orientation.

²you might call *midpoint(p_1, p_2)* instead

3.3.3 Orientation and Relative Position

Geometric objects in CGAL have member functions that test the position of a point relative to the object. Full dimensional objects and their boundaries are represented by the same type, e.g. halfspaces and hyperplanes are not distinguished, neither are balls and spheres. Such objects split the ambient space into two full-dimensional parts, a bounded part and an unbounded part (e.g. spheres), or two unbounded parts (e.g. hyperplanes). By default these objects are oriented, i.e., one of the resulting parts is called the positive side, the other one is called the negative side. Both of these may be unbounded.

For these objects there is a function *oriented_side()* that determines whether a test point is on the positive side, the negative side, or on the oriented boundary. This function returns a value of type *Oriented_side*.

Those objects that split the space in a bounded and an unbounded part, have a member function *bounded_side()* with return type *Bounded_side*.

If an object is lower dimensional, e.g. a segment in d -dimensional space, there is only a test whether a point belongs to the object or not. This member function, which takes a point as an argument and returns a boolean value, is called *has_on()*

3.4 Predicates and Constructions

3.4.1 Predicates

Predicates are at the heart of a geometry kernel. They are basic units for the composition of geometric algorithms and encapsulate decisions. Hence their correctness is crucial for the control flow and hence for the correctness of an implementation of a geometric algorithm. CGAL uses the term predicate in a generalized sense. Not only components returning a Boolean value are called predicates but also components returning an enumeration type like a *Comparison_result* or an *Orientation*. We say components, because predicates are implemented both as functions and function objects (also called functors and provided by a kernel class).

CGAL provides predicates for the orientation of point sets (*orientation*), for comparing points according to some given order, especially for comparing Cartesian coordinates (e.g. *lexicographically_xy_smaller*), in-sphere tests, and predicates to compare distances.

3.4.2 Constructions

Functions and function objects that generate objects that are neither of type *bool* nor enum types are called constructions. Constructions involve computation of new numerical values and may be imprecise due to rounding errors unless a kernel with an exact number type is used.

Affine transformations (*Aff_transformation_d<R>*) allow to generate new object instances under arbitrary affine transformations. These transformations include translations, rotations (within planes) and scaling. Most of the geometric objects in a kernel have a member function *transform(Aff_transformation t)* which applies the transformation to the object instance.

CGAL also provides a set of functions that detect or compute the intersection between objects and functions to calculate their squared distance. Moreover, some member functions of kernel objects are constructions.

So there are routines that compute the square of the Euclidean distance, but no routines that compute the distance itself. Why? First of all, the two values can be derived from each other quite easily (by taking the square root

or taking the square). So, supplying only the one and not the other is only a minor inconvenience for the user. Second, often either value can be used. This is for example the case when (squared) distances are compared. Third, the library wants to stimulate the use of the squared distance instead of the distance. The squared distance can be computed in more cases and the computation is cheaper. We do this by not providing the perhaps more natural routine, The problem of a distance routine is that it needs the *sqrt* operation. This has two drawbacks:

- The *sqrt* operation can be costly. Even if it is not very costly for a specific number type and platform, avoiding it is always cheaper.
- There are number types on which no *sqrt* operation is defined, especially integer types and rationals.

3.4.3 Intersection and Polymorphic Return Values

Intersections on kernel objects currently cover only those objects that are part of flats (*Segment_d*<*R*>, *Ray_d*<*R*>, *Line_c*<*R*>, and *Hyperplane_d*<*R*>). For any pair of objects *o1*, *o2* of these types the operation *intersection(o1,o2)* returns a polymorphic object that wraps the result of the intersection operation.

The class *Object* provides the polymorphic abstraction. An object *obj* of type *Object* can represent an arbitrary class. The only operations it provides is to make copies and assignments, so that you can put them in lists or arrays. Note that *Object* is NOT a common base class for the elementary classes. Therefore, there is no automatic conversion from these classes to *Object*. Rather this is done with the global function *make_object()*. This encapsulation mechanism requires the use of *assign* to unwrap the encapsulated class.

Example

In the following example, the object type is used as a return value for the intersection computation, as there are possibly different return values.

```
Point_d< Cartesian_d<double> > p;
Segment_d< Cartesian_d<double> > s, s1, s2;
std::cin >> s1 >> s2;
Object obj = intersection(s1, s2);
if ( assign(p, obj) ) {
    /* do something with p */
} else if ( (assign(s, obj) ) {
    /* do something with s */
}
/* there was no intersection */
```

3.4.4 Constructive Predicates

For testing where a point *p* lies with respect to a hyperplane defined by an array *P* of points *p*₁, ... , *p*_{*d*}, one may be tempted to construct the hyperplane *Hyperplane_d*<*R*>(*d*,*P*,*P*+*d*) and use the method *oriented_side(p)*. This may pay off if many tests with respect to the plane are made. Nevertheless, unless the number type is exact, the constructed plane is only approximated, and round-off errors may lead *oriented_side(p)* to return an orientation which is different from the orientation of *p*₁, ... , *p*_{*d*}, *p*.

In CGAL, we provide predicates in which such geometric decisions are made directly with a reference to the input points in *P* without an intermediary object like a plane. For the above test, the recommended way to get the result is to use *orientation(P',P'+d)*, where *P'* is an array containing the points *p*₁, ... , *p*_{*d*}, *p*.

For exact number types like *leda_real*, the situation is different. If several tests are to be made with the same plane, it pays off to construct the plane and to use *oriented_side(p)*.

dD Kernel

Reference Manual

Michael Seel

Concepts

Kernel_d	page 499
LinearAlgebraTraits_d	page 437
Vector	page 440
Matrix	page 442

Classes

CGAL::Point_d<R>	page ??
CGAL::Vector_d<R>	page ??
CGAL::Direction_d<R>	page ??
CGAL::Segment_d<R>	page ??
CGAL::Ray_d<R>	page ??
CGAL::Line_d<R>	page ??
CGAL::Hyperplane_d<R>	page ??
CGAL::Sphere_d<R>	page ??
CGAL::Aff_transformation_d<R>	page ??

Predicates and Construction on Points

CGAL::affine_rank	page 478
CGAL::affinely_independent	page 477
CGAL::center_of_sphere	page 479
CGAL::compare_lexicographically	page 480
CGAL::contained_in_affine_hull	page 481
CGAL::contained_in_simplex	page 483
CGAL::lexicographically_smaller	page 487
CGAL::lexicographically_smaller_or_equal	page 488
CGAL::lift_to_paraboloid	page 489
CGAL::midpoint	page 493
CGAL::orientation	page 494
CGAL::side_of_bounded_sphere	page 496
CGAL::side_of_oriented_sphere	page 497

CGAL::squared_distance page [498](#)

Predicates and Construction on Vectors

CGAL::linear_base page [491](#)

CGAL::linear_rank page [492](#)

CGAL::linearly_independent page [490](#)

Intersection operations

CGAL::do_intersect page [943](#)

CGAL::intersection page [945](#)

3.5 Linear Algebra Concepts and Classes

LinearAlgebraTraits_d

Definition

The data type *LinearAlgebraTraits_d* encapsulates two classes *Matrix*, *Vector* and many functions of basic linear algebra. An instance of data type *Matrix* is a matrix of variables of type *NT*. Accordingly, *Vector* implements vectors of variables of type *NT*. Most functions of linear algebra are *checkable*, i.e., the programs can be asked for a proof that their output is correct. For example, if the linear system solver declares a linear system $Ax = b$ unsolvable it also returns a vector c such that $c^T A = 0$ and $c^T b \neq 0$.

Types

LinearAlgebraTraits_d::NT the number type of the components.

LinearAlgebraTraits_d::Vector the vector type.

LinearAlgebraTraits_d::Matrix the matrix type.

Operations

static Matrix *LA.transpose(Matrix M)*

returns M^T (a $M.column_dimension() \times M.column_dimension()$ - matrix).

static bool *LA.inverse(Matrix M, Matrix& I, NT& D, Vector& c)*

determines whether M has an inverse. It also computes either the inverse as $(1/D) \cdot I$ or when no inverse exists, a vector c such that $c^T \cdot M = 0$.

Precondition: M is square.

static Matrix *LA.inverse(Matrix M, NT& D)*

returns the inverse matrix of M . More precisely, $1/D$ times the matrix returned is the inverse of M .

Precondition: $determinant(M) \neq 0$.

Precondition: M is square.

static bool *LA.linear_solver(Matrix M, Vector b, Vector& x, NT& D)*

as above, but without the witness *c*
Precondition: M.row_dimension() = b.dimension().

static bool *LA.is_solvable(Matrix M, Vector b)*

determines whether the system $M \cdot x = b$ is solvable
Precondition: M.row_dimension() = b.dimension().

static bool *LA.homogeneous_linear_solver(Matrix M, Vector& x)*

determines whether the homogeneous linear system $M \cdot x = 0$ has a non - trivial solution. If yes, then *x* is such a solution.

static int *LA.homogeneous_linear_solver(Matrix M, Matrix& spanning_vecs)*

determines the solution space of the homogeneous linear system $M \cdot x = 0$. It returns the dimension of the solution space. Moreover the columns of *spanning_vecs* span the solution space.

static int *LA.independent_columns(Matrix M, std::vector<int>& columns)*

returns the indices of a maximal subset of independent columns of *M*.

static int *LA.rank(Matrix M)*

returns the rank of matrix *M*

Has Models

CGAL::Linear_algebraHd<RT> [page 446](#)
CGAL::Linear_algebraCd<FT> [page 445](#)

Vector

Definition

An instance of data type *Vector* is a vector of variables of number type *NT*. Together with the type *Matrix* it realizes the basic operations of linear algebra.

Types

<i>Vector::NT</i>	the ring type of the components.
<i>Vector::iterator</i>	the iterator type for accessing components.
<i>Vector::const_iterator</i>	the const iterator type for accessing components.

Creation

<i>Vector</i> <i>v</i> ;	creates an instance <i>v</i> of type <i>Vector</i> .
<i>Vector</i> <i>v</i> (<i>int d</i>);	creates an instance <i>v</i> of type <i>Vector</i> . <i>v</i> is initialized to a vector of dimension <i>d</i> .
<i>Vector</i> <i>v</i> (<i>int d</i> , <i>NT x</i>);	creates an instance <i>v</i> of type <i>Vector</i> . <i>v</i> is initialized to a vector of dimension <i>d</i> with entries <i>x</i> .
<i>template</i> <class <i>Forward_iterator</i> > <i>Vector</i> <i>v</i> (<i>Forward_iterator first</i> , <i>Forward_iterator last</i>);	
	creates an instance <i>v</i> of type <i>Vector</i> ; <i>v</i> is initialized to the vector with entries set [<i>first</i> , <i>last</i>).
	<i>Requirement: Forward_iterator</i> has value type <i>NT</i> .

Operations

<i>int</i>	<i>v.dimension()</i>	returns the dimension of <i>v</i> .
<i>bool</i>	<i>v.is_zero()</i>	returns true iff <i>v</i> is the zero vector.
<i>NT&</i>	<i>v[int i]</i>	returns the <i>i</i> -th component of <i>v</i> . <i>Precondition:</i> $0 \leq i \leq v.dimension() - 1$.
<i>iterator</i>	<i>v.begin()</i>	iterator to the first component.
<i>iterator</i>	<i>v.end()</i>	iterator beyond the last component.

The same operations *begin()*, *end()* exist for *const_iterator*.

<i>Vector</i>	$v + vI$	Addition. <i>Precondition:</i> $v.dimension() == vI.dimension()$.
<i>Vector</i>	$v - vI$	Subtraction. <i>Precondition:</i> $v.dimension() = vI.dimension()$.
<i>NT</i>	$v * vI$	Inner Product. <i>Precondition:</i> $v.dimension() = vI.dimension()$.
<i>Vector</i>	$-v$	Negation.
<i>Vector&</i>	$v += vI$	Addition plus assignment. <i>Precondition:</i> $v.dimension() == vI.dimension()$.
<i>Vector&</i>	$v -= vI$	Subtraction plus assignment. <i>Precondition:</i> $v.dimension() == vI.dimension()$.
<i>Vector&</i>	$v *= NT\ s$	Scalar multiplication plus assignment.
<i>Vector&</i>	$v /= NT\ s$	Scalar division plus assignment.
<i>Vector</i>	$NT\ r * v$	Componentwise multiplication with number r .
<i>Vector</i>	$v * NT\ r$	Componentwise multiplication with number r .

Matrix

Definition

An instance of data type *Matrix* is a matrix of variables of number type *NT*. The types *Matrix* and *Vector* together realize many functions of basic linear algebra.

Types

<i>Matrix:: NT</i>	the ring type of the components.
<i>Matrix:: iterator</i>	bidirectional iterator for accessing all components row-wise.
<i>Matrix:: row_iterator</i>	random access iterator for accessing row entries.
<i>Matrix:: column_iterator</i>	random access iterator for accessing column entries.

There also constant versions of the above iterators: *const_iterator*, *row_const_iterator*, and *column_const_iterator*.

<i>Matrix:: Identity</i>	a tag class for identity initialization
<i>Matrix:: Vector</i>	the vector type used.

Creation

<i>Matrix M</i> ;	creates an instance <i>M</i> of type <i>Matrix</i> .
<i>Matrix M</i> (<i>int n</i>);	creates an instance <i>M</i> of type <i>Matrix</i> of dimension $n \times n$ initialized to the zero matrix.
<i>Matrix M</i> (<i>int m</i> , <i>int n</i>);	creates an instance <i>M</i> of type <i>Matrix</i> of dimension $m \times n$ initialized to the zero matrix.
<i>Matrix M</i> (<i>std::pair<int,int> p</i>);	creates an instance <i>M</i> of type <i>Matrix</i> of dimension $p.first \times p.second$ initialized to the zero matrix.
<i>Matrix M</i> (<i>int n</i> , <i>Identity</i> , <i>NT x</i> = <i>NT(1)</i>);	creates an instance <i>M</i> of type <i>Matrix</i> of dimension $n \times n$ initialized to the identity matrix (times <i>x</i>).
<i>Matrix M</i> (<i>int m</i> , <i>int n</i> , <i>NT x</i>);	creates an instance <i>M</i> of type <i>Matrix</i> of dimension $m \times n$ initialized to the matrix with <i>x</i> entries.

```
template <class Forward_iterator>
```

```
Matrix M( Forward_iterator first, Forward_iterator last);
```

creates an instance M of type *Matrix*. Let S be the ordered set of n column-vectors of common dimension m as given by the iterator range $[first, last)$. M is initialized to an $m \times n$ matrix with the columns as specified by S .

Precondition: *Forward_iterator* has a value type V from which we require to provide a iterator type $V::const_iterator$, to have $V::value_type == NT$.

Note that *Vector* or $std::vector<NT>$ fulfill these requirements.

```
Matrix M( std::vector< Vector > A);
```

creates an instance M of type *Matrix*. Let A be an array of n column-vectors of common dimension m . M is initialized to an $m \times n$ matrix with the columns as specified by A .

Operations

<i>int</i>	<i>M.row_dimension()</i>	returns n , the number of rows of M .
<i>int</i>	<i>M.column_dimension()</i>	returns m , the number of columns of M .
<i>std::pair<int,int></i>	<i>M.dimension()</i>	returns (m, n) , the dimension pair of M .
<i>Vector</i>	<i>M.row(int i)</i>	returns the i -th row of M (an m - vector). <i>Precondition:</i> $0 \leq i \leq m - 1$.
<i>Vector</i>	<i>M.column(int i)</i>	returns the i -th column of M (an n - vector). <i>Precondition:</i> $0 \leq i \leq n - 1$.
<i>NT&</i>	<i>M(int i, int j)</i>	returns $M_{i,j}$. <i>Precondition:</i> $0 \leq i \leq m - 1$ and $0 \leq j \leq n - 1$.
<i>void</i>	<i>M.swap_rows(int i, int j)</i>	swaps rows i and j . <i>Precondition:</i> $0 \leq i \leq m - 1$ and $0 \leq j \leq m - 1$.
<i>void</i>	<i>M.swap_columns(int i, int j)</i>	swaps columns i and j . <i>Precondition:</i> $0 \leq i \leq n - 1$ and $0 \leq j \leq n - 1$.
<i>row_iterator</i>	<i>M.row_begin(int i)</i>	an iterator pointing to the first entry of the i th row. <i>Precondition:</i> $0 \leq i \leq m - 1$.
<i>row_iterator</i>	<i>M.row_end(int i)</i>	an iterator pointing beyond the last entry of the i th row. <i>Precondition:</i> $0 \leq i \leq m - 1$.

<i>column_iterator</i>	<i>M.column_begin(int i)</i>	an iterator pointing to the first entry of the <i>i</i> th column. <i>Precondition:</i> $0 \leq i \leq n - 1$.
<i>column_iterator</i>	<i>M.column_end(int i)</i>	an iterator pointing beyond the last entry of the <i>i</i> th column. <i>Precondition:</i> $0 \leq i \leq n - 1$.
<i>iterator</i>	<i>M.begin()</i>	an iterator pointing to the first entry of <i>M</i> .
<i>iterator</i>	<i>M.end()</i>	an iterator pointing beyond the last entry of <i>M</i> .

The same operations exist for *row_const_iterator*, *column_const_iterator* and *const_iterator*.

<i>bool</i>	<i>M == M1</i>	Test for equality.
<i>bool</i>	<i>M != M1</i>	Test for inequality.

Arithmetic Operators

<i>Matrix</i>	<i>M + M1</i>	Addition. <i>Precondition:</i> <i>M.row_dimension() == M1.row_dimension()</i> and <i>M.column_dimension() == M1.column_dimension()</i> .
<i>Matrix</i>	<i>M - M1</i>	Subtraction. <i>Precondition:</i> <i>M.row_dimension() == M1.row_dimension()</i> and <i>M.column_dimension() == M1.column_dimension()</i> .
<i>Matrix</i>	<i>-M</i>	Negation.
<i>Matrix</i>	<i>M * M1</i>	Multiplication. <i>Precondition:</i> <i>M.column_dimension() == M1.row_dimension()</i> .
<i>Vector</i>	<i>M * Vector vec</i>	Multiplication with vector. <i>Precondition:</i> <i>M.column_dimension() == vec.dimension()</i> .
<i>Matrix</i>	<i>NT x * M</i>	Multiplication of every entry with <i>x</i> .
<i>Matrix</i>	<i>M * NT x</i>	Multiplication of every entry with <i>x</i> .

CGAL::Linear_algebraCd<FT>

Definition

The class *Linear_algebraCd*<FT> serves as the default traits class for the LA parameter of *CGAL::Cartesian_d*<FT,LA>. It implements linear algebra for field number types *FT*.

```
#include <CGAL/Linear_algebraCd.h>
```

Is Model for the Concepts

LinearAlgebraTraits_d page [437](#)

Requirements

FT must be a field number type.

Operations

Fits all operation requirements of the concept.

CGAL::Linear_algebraHd<RT>

Definition

The class *Linear_algebraHd*<*RT*> serves as the default traits class for the LA parameter of *CGAL::Homogeneous_d*<*RT*,*LA*>. It implements linear algebra for Euclidean ring number types *RT*.

```
#include <CGAL/Linear_algebraHd.h>
```

Is Model for the Concepts

LinearAlgebraTraits_d page [437](#)

Requirements

To make a ring number type *RT* work with this class it has to provide a division *operator/* with remainder.

Operations

Fits all operation requirements of the concept.

3.6 Kernel Objects

CGAL::Point_d<Kernel>

Definition

An instance of data type *Point_d<Kernel>* is a point of Euclidean space in dimension d . A point $p = (p_0, \dots, p_{d-1})$ in d -dimensional space can be represented by homogeneous coordinates (h_0, h_1, \dots, h_d) of number type RT such that $p_i = h_i/h_d$, which is of type FT . The homogenizing coordinate h_d is positive.

We call p_i , $0 \leq i < d$ the i -th Cartesian coordinate and h_i , $0 \leq i \leq d$, the i -th homogeneous coordinate. We call d the dimension of the point.

Types

Point_d<Kernel>:: RT the ring type.

Point_d<Kernel>:: FT the field type.

Point_d<Kernel>:: LA the linear algebra layer.

Point_d<Kernel>:: Cartesian_const_iterator
a read-only iterator for the Cartesian coordinates.

Point_d<Kernel>:: Homogeneous_const_iterator
a read-only iterator for the homogeneous coordinates.

Creation

Point_d<Kernel> p; introduces a variable p of type *Point_d<Kernel>*.

Point_d<Kernel> p(int d, Origin);
introduces a variable p of type *Point_d<Kernel>* in d -dimensional space, initialized to the origin.

template <class InputIterator>

Point_d<Kernel> p(int d, InputIterator first, InputIterator last);

introduces a variable p of type $Point_d<Kernel>$ in dimension d . If $size\ [first, last) == d$ this creates a point with Cartesian coordinates $set\ [first, last)$. If $size\ [first, last) == p+1$ the range specifies the homogeneous coordinates $H = set\ [first, last) = (\pm h_0, \pm h_1, \dots, \pm h_d)$ where the sign chosen is the sign of h_d .

Precondition: d is nonnegative, $[first, last)$ has d or $d+1$ elements where the last has to be non-zero.

Requirement: The value type of $InputIterator$ is RT .

template <class InputIterator>

Point_d<Kernel> p(int d, InputIterator first, InputIterator last, RT D);

introduces a variable p of type $Point_d<Kernel>$ in dimension d initialized to the point with homogeneous coordinates as defined by $H = set\ [first, last)$ and $D: (\pm H[0], \pm H[1], \dots, \pm H[d-1], \pm D)$. The sign chosen is the sign of D .

Precondition: D is non-zero, the iterator range defines a d -tuple of RT .

Requirement: The value type of $InputIterator$ is RT .

Point_d<Kernel> p(RT x, RT y, RT w = 1);

introduces a variable p of type $Point_d<Kernel>$ in 2-dimensional space.

Precondition: $w \neq 0$.

Point_d<Kernel> p(RT x, RT y, RT z, RT w);

introduces a variable p of type $Point_d<Kernel>$ in 3-dimensional space.

Precondition: $w \neq 0$.

Operations

<i>int</i>	<i>p.dimension()</i>	returns the dimension of p .
<i>FT</i>	<i>p.cartesian(int i)</i>	returns the i -th Cartesian coordinate of p . <i>Precondition:</i> $0 \leq i < d$.
<i>FT</i>	<i>p[int i]</i>	returns the i -th Cartesian coordinate of p . <i>Precondition:</i> $0 \leq i < d$.
<i>RT</i>	<i>p.homogeneous(int i)</i>	returns the i -th homogeneous coordinate of p . <i>Precondition:</i> $0 \leq i \leq d$.
<i>Cartesian_const_iterator</i>	<i>p.cartesian_begin()</i>	returns an iterator pointing to the zeroth Cartesian coordinate p_0 of p .

<i>Cartesian_const_iterator</i>	<i>p.cartesian_end()</i>	returns an iterator pointing beyond the last Cartesian coordinate of <i>p</i> .
---------------------------------	--------------------------	---------------------------------------------------------------------------------

Homogeneous_const_iterator

p.homogeneous_begin()

returns an iterator pointing to the zeroth homogeneous coordinate h_0 of *p*.

Homogeneous_const_iterator

p.homogeneous_end()

returns an iterator pointing beyond the last homogeneous coordinate of *p*.

Point_d<Kernel>

p.transform(Aff_transformation_d<Kernel> t)

returns $t(p)$.

Arithmetic Operators, Tests and IO

Vector_d<Kernel>

p - Origin o

returns the vector $p - O$.

Vector_d<Kernel>

p - q

returns $p - q$.
Precondition: $p.dimension() == q.dimension()$.

Point_d<Kernel>

p + Vector_d<Kernel> v

returns $p + v$.
Precondition: $p.dimension() == v.dimension()$.

Point_d<Kernel>

p - Vector_d<Kernel> v

returns $p - v$.
Precondition: $p.dimension() == v.dimension()$.

Point_d<Kernel>&

p += Vector_d<Kernel> v

adds v to p .

Precondition: $p.dimension() == v.dimension()$.

Point_d<Kernel>&

p -= Vector_d<Kernel> v

subtracts v from p .

Precondition: $p.dimension() == v.dimension()$.

bool $p == Origin$ returns true if p is the origin.

Downward compatibility

We provide operations of the lower dimensional interface $x()$, $y()$, $z()$, $hx()$, $hy()$, $hz()$, $hw()$.

Implementation

Points are implemented by arrays of RT items. All operations like creation, initialization, tests, point - vector arithmetic, input and output on a point p take time $O(p.dimension())$. $dimension()$, coordinate access and conversions take constant time. The space requirement for points is $O(p.dimension())$.

CGAL::Vector_d<Kernel>

Definition

An instance of data type *Vector_d<Kernel>* is a vector of Euclidean space in dimension d . A vector $r = (r_0, \dots, r_{d-1})$ can be represented in homogeneous coordinates (h_0, \dots, h_d) of number type RT , such that $r_i = h_i/h_d$ which is of type FT . We call the r_i 's the Cartesian coordinates of the vector. The homogenizing coordinate h_d is positive.

This data type is meant for use in computational geometry. It realizes free vectors as opposed to position vectors (type *Point_d*). The main difference between position vectors and free vectors is their behavior under affine transformations, e.g., free vectors are invariant under translations.

Types

Vector_d<Kernel>::RT the ring type.

Vector_d<Kernel>::FT the field type.

Vector_d<Kernel>::LA the linear algebra layer.

Vector_d<Kernel>::Cartesian_const_iterator
a read-only iterator for the Cartesian coordinates.

Vector_d<Kernel>::Homogeneous_const_iterator
a read-only iterator for the homogeneous coordinates.

Vector_d<Kernel>::Base_vector
construction tag.

Creation

Vector_d<Kernel> v; introduces a variable v of type *Vector_d<Kernel>*.

Vector_d<Kernel> v(int d, Null_vector);
introduces the zero vector v of type *Vector_d<Kernel>* in d -dimensional space.
For the creation flag *CGAL::NULL_VECTOR* can be used.

template <class InputIterator>

Vector_d<Kernel> v(int d, InputIterator first, InputIterator last);

introduces a variable v of type *Vector_d<Kernel>* in dimension d . If *size [first,last) == d* this creates a vector with Cartesian coordinates *set [first,last)*. If *size [first,last) == p+1* the range specifies the homogeneous coordinates $H = \text{set } [first, last) = (\pm h_0, \pm h_1, \dots, \pm h_d)$ where the sign chosen is the sign of h_d .

Precondition: d is nonnegative, *[first,last)* has d or $d+1$ elements where the last has to be non-zero.

Requirement: The value type of *InputIterator* is *RT*.

template <class InputIterator>

Vector_d<Kernel> v(int d, InputIterator first, InputIterator last, RT D);

introduces a variable v of type *Vector_d<Kernel>* in dimension d initialized to the vector with homogeneous coordinates as defined by $H = \text{set } [first, last)$ and $D: (\pm H[0], \pm H[1], \dots, \pm H[d-1], \pm D)$. The sign chosen is the sign of D .

Precondition: D is non-zero, the iterator range defines a d -tuple of *RT*.

Requirement: The value type of *InputIterator* is *RT*.

Vector_d<Kernel> v(int d, Base_vector, int i);

returns a variable v of type *Vector_d<Kernel>* initialized to the i -th base vector of dimension d .

Precondition: $0 \leq i < d$.

Vector_d<Kernel> v(RT x, RT y, RT w = 1);

introduces a variable v of type *Vector_d<Kernel>* in 2-dimensional space.

Precondition: $w \neq 0$.

Vector_d<Kernel> v(RT x, RT y, RT z, RT w);

introduces a variable v of type *Vector_d<Kernel>* in 3-dimensional space.

Precondition: $w \neq 0$.

Operations

<i>int</i>	<i>v.dimension()</i>	returns the dimension of v .
<i>FT</i>	<i>v.cartesian(int i)</i>	returns the i -th Cartesian coordinate of v . <i>Precondition:</i> $0 \leq i < d$.
<i>FT</i>	<i>v[int i]</i>	returns the i -th Cartesian coordinate of v . <i>Precondition:</i> $0 \leq i < d$.
<i>RT</i>	<i>v.homogeneous(int i)</i>	returns the i -th homogeneous coordinate of v . <i>Precondition:</i> $0 \leq i \leq d$.

<i>FT</i>	<i>v.squared_length()</i>	returns the square of the length of <i>v</i> .
<i>Cartesian_const_iterator</i>	<i>v.cartesian_begin()</i>	returns an iterator pointing to the zeroth Cartesian coordinate of <i>v</i> .
<i>Cartesian_const_iterator</i>	<i>v.cartesian_end()</i>	returns an iterator pointing beyond the last Cartesian coordinate of <i>v</i> .
<i>Homogeneous_const_iterator</i>	<i>v.homogeneous_begin()</i>	returns an iterator pointing to the zeroth homogeneous coordinate of <i>v</i> .
<i>Homogeneous_const_iterator</i>	<i>v.homogeneous_end()</i>	returns an iterator pointing beyond the last homogeneous coordinate of <i>v</i> .
<i>Direction_d<Kernel></i>	<i>v.direction()</i>	returns the direction of <i>v</i> .
<i>Vector_d<Kernel></i>	<i>v.transform(Aff_transformation_d<Kernel> t)</i>	returns $t(v)$.

Arithmetic Operators, Tests and IO

<i>Vector_d<Kernel>&</i>	<i>v *= RT n</i>	multiplies all Cartesian coordinates by <i>n</i> .
<i>Vector_d<Kernel>&</i>	<i>v *= FT r</i>	multiplies all Cartesian coordinates by <i>r</i> .
<i>Vector_d<Kernel></i>	<i>v / RT n</i>	returns the vector with Cartesian coordinates $v_i/n, 0 \leq i < d$.
<i>Vector_d<Kernel></i>	<i>v / FT r</i>	returns the vector with Cartesian coordinates $v_i/r, 0 \leq i < d$.
<i>Vector_d<Kernel>&</i>	<i>v /= RT n</i>	divides all Cartesian coordinates by <i>n</i> .
<i>Vector_d<Kernel>&</i>	<i>v /= FT r</i>	divides all Cartesian coordinates by <i>r</i> .
<i>FT</i>	<i>v * w</i>	inner product, i.e., $\sum_{0 \leq i < d} v_i w_i$, where v_i and w_i are the Cartesian coordinates of <i>v</i> and <i>w</i> respectively.
<i>Vector_d<Kernel></i>	<i>v + w</i>	returns the vector with Cartesian coordinates $v_i + w_i, 0 \leq i < d$.
<i>Vector_d<Kernel>&</i>	<i>v += w</i>	addition plus assignment.

<i>Vector_d<Kernel></i>	$v - w$	returns the vector with Cartesian coordinates $v_i - w_i, 0 \leq i < d$.
<i>Vector_d<Kernel>&</i>	$v -= w$	subtraction plus assignment.
<i>Vector_d<Kernel></i>	$-v$	returns the vector in opposite direction.
<i>bool</i>	<i>v.is_zero()</i>	returns true if <i>v</i> is the zero vector.

Downward compatibility

We provide all operations of the lower dimensional interface *x()*, *y()*, *z()*, *hx()*, *hy()*, *hz()*, *hw()*.

<i>Vector_d<Kernel></i>	<i>RT n * v</i>	returns the vector with Cartesian coordinates nv_i .
<i>Vector_d<Kernel></i>	<i>FT r * v</i>	returns the vector with Cartesian coordinates $rv_i, 0 \leq i < d$.

Implementation

Vectors are implemented by arrays of variables of type *RT*. All operations like creation, initialization, tests, vector arithmetic, input and output on a vector *v* take time $O(v.dimension())$. coordinate access, *dimension()* and conversions take constant time. The space requirement of a vector is $O(v.dimension())$.

CGAL::Direction_d<Kernel>

Definition

A *Direction_d* is a vector in the d -dimensional vector space where we forget about its length. We represent directions in d -dimensional space as a tuple (h_0, \dots, h_d) of variables of type RT which we call the homogeneous coordinates of the direction. The coordinate h_d must be positive. The Cartesian coordinates of a direction are $c_i = h_i/h_d$ for $0 \leq i < d$, which are of type FT . Two directions are equal if their Cartesian coordinates are positive multiples of each other. Directions are in one-to-one correspondence to points on the unit sphere.

Types

Direction_d<Kernel>::RT the ring type.

Direction_d<Kernel>::FT the field type.

Direction_d<Kernel>::LA the linear algebra layer.

Direction_d<Kernel>::Delta_const_iterator
a read-only iterator for the deltas of *dir*.

Direction_d<Kernel>::Base_direction
construction tag.

Creation

Direction_d<Kernel> dir; introduces a variable *dir* of type *Direction_d<Kernel>*.

Direction_d<Kernel> dir(Vector_d<Kernel> v);
introduces a variable *dir* of type *Direction_d<Kernel>* initialized to the direction of *v*.

template <class InputIterator>
Direction_d<Kernel> dir(int d, InputIterator first, InputIterator last);
introduces a variable *dir* of type *Direction_d<Kernel>* in dimension d with representation tuple *set* $[first, last)$.
Precondition: d is nonnegative, $[first, last)$ has d elements.
Requirement: The value type of *InputIterator* is RT .

Direction_d<Kernel> *dir*(*int d*, *Base_direction*, *int i*);

returns a variable *dir* of type *Direction_d<Kernel>* initialized to the direction of the *i*-th base vector of dimension *d*.
Precondition: $0 \leq i < d$.

Direction_d<Kernel> *dir*(*RT x*, *RT y*);

introduces a variable *dir* of type *Direction_d<Kernel>* in 2-dimensional space.

Direction_d<Kernel> *dir*(*RT x*, *RT y*, *RT z*);

introduces a variable *dir* of type *Direction_d<Kernel>* in 3-dimensional space.

Operations

<i>int</i>	<i>dir.dimension()</i>	returns the dimension of <i>dir</i> .
<i>RT</i>	<i>dir.delta(int i)</i>	returns the <i>i</i> -th component of <i>dir</i> . <i>Precondition</i> : $0 \leq i < d$.
<i>RT</i>	<i>dir[int i]</i>	returns the <i>i</i> -th delta of <i>dir</i> . <i>Precondition</i> : $0 \leq i < d$.
<i>Delta_const_iterator</i>	<i>dir.deltas_begin()</i>	returns an iterator pointing to the first delta of <i>dir</i> .
<i>Delta_const_iterator</i>	<i>dir.deltas_end()</i>	returns an iterator pointing beyond the last delta of <i>dir</i> .
<i>Vector_d<Kernel></i>	<i>dir.vector()</i>	returns a vector pointing in direction <i>dir</i> .
<i>bool</i>	<i>dir.is_degenerate()</i>	returns true iff <i>dir.delta(i)=0</i> for all $0 \leq i < d$.
<i>Direction_d<Kernel></i>	<i>dir.transform(Aff_transformation_d<Kernel> t)</i>	returns <i>t(p)</i> .
<i>Direction_d<Kernel></i>	<i>dir.opposite()</i>	returns the direction opposite to <i>dir</i> .
<i>Direction_d<Kernel></i>	$-dir$	returns the direction opposite to <i>dir</i> .

Downward compatibility

We provide the operations of the lower dimensional interface *dx()*, *dy()*, *dz()*.

Implementation

Directions are implemented by arrays of integers as an item type. All operations like creation, initialization, tests, inversion, input and output on a direction d take time $O(d.dimension())$. $dimension()$, coordinate access and conversion take constant time. The space requirement is $O(d.dimension())$.

CGAL::Line_d<Kernel>

Definition

An instance of data type *Line_d* is an oriented line in *d*-dimensional Euclidean space.

Types

Line_d<Kernel>::R the representation type.

Line_d<Kernel>::RT the ring type.

Line_d<Kernel>::FT the field type.

Line_d<Kernel>::LA the linear algebra layer.

Creation

Line_d<Kernel> l; introduces a variable *l* of type *Line_d<Kernel>*.

Line_d<Kernel> l(Point_d<Kernel> p, Point_d<Kernel> q);

introduces a line through *p* and *q* and oriented from *p* to *q*.

Precondition: *p* and *q* are distinct and have the same dimension.

Line_d<Kernel> l(Point_d<Kernel> p, Direction_d<Kernel> dir);

introduces a line through *p* with direction *dir*.

Precondition: *p.dimension()==dir.dimension()*, *dir* is not degenerate.

Line_d<Kernel> l(Segment_d<Kernel> s);

introduces a variable *l* of type *Line_d<Kernel>* and initializes it to the line through *s.source()* and *s.target()* with direction from *s.source()* to *s.target()*.

Precondition: *s* is not degenerate.

Line_d<Kernel> l(Ray_d<Kernel> r);

introduces a variable *l* of type *Line_d<Kernel>* and initializes it to the line through *r.point(1)* and *r.point(2)*.

Operations

<i>int</i>	<i>l.dimension()</i>	returns the dimension of the ambient space.
<i>Point_d<Kernel></i>	<i>l.point(int i)</i>	returns an arbitrary point on <i>l</i> . It holds that <i>point(i) == point(j)</i> , iff <i>i==j</i> . Furthermore, <i>l</i> is directed from <i>point(i)</i> to <i>point(j)</i> , for all <i>i < j</i> .
<i>Line_d<Kernel></i>	<i>l.opposite()</i>	returns the line (<i>point(2),point(1)</i>) of opposite direction.
<i>Direction_d<Kernel></i>	<i>l.direction()</i>	returns the direction of <i>l</i> .
<i>Line_d<Kernel></i>	<i>l.transform(Aff_transformation_d<Kernel> t)</i>	returns <i>t(l)</i> . Precondition: <i>l.dimension()==t.dimension()</i> .
<i>Line_d<Kernel></i>	<i>l + Vector_d<Kernel> v</i>	returns <i>l+v</i> , i.e., <i>l</i> translated by vector <i>v</i> . Precondition: <i>l.dimension()==v.dimension()</i> .
<i>Point_d<Kernel></i>	<i>l.projection(Point_d<Kernel> p)</i>	returns the point of intersection of <i>l</i> with the hyperplane that is orthogonal to <i>l</i> and that contains <i>p</i> . Precondition: <i>l.dimension()==p.dimension()</i> .
<i>bool</i>	<i>l.has_on(Point_d<Kernel> p)</i>	returns true if <i>p</i> lies on <i>l</i> and false otherwise. Precondition: <i>l.dimension()==p.dimension()</i> .

Non-Member Functions

<i>bool</i>	<i>weak_equality(l1, l2)</i>	Test for equality as unoriented lines. Precondition: <i>l1.dimension()==l2.dimension()</i> .
<i>bool</i>	<i>parallel(l1, l2)</i>	returns true if <i>l1</i> and <i>l2</i> are parallel as unoriented lines and false otherwise. Precondition: <i>l1.dimension()==l2.dimension()</i> .

Implementation

Lines are implemented by a pair of points as an item type. All operations like creation, initialization, tests, direction calculation, input and output on a line *l* take time $O(l.dimension())$. *dimension()*, coordinate and point access, and identity test take constant time. The operations for intersection calculation also take time $O(l.dimension())$. The space requirement is $O(l.dimension())$.

CGAL::Ray_d<Kernel>

Definition

An instance of data type *Ray_d* is a ray in *d*-dimensional Euclidean space. It starts in a point called the source of *r* and it goes to infinity.

Types

Ray_d<Kernel>:: R the representation type.

Ray_d<Kernel>:: RT the ring type.

Ray_d<Kernel>:: FT the field type.

Ray_d<Kernel>:: LA the linear algebra layer.

Creation

Ray_d<Kernel> r; introduces some ray in *d*-dimensional space.

Ray_d<Kernel> r(Point_d<Kernel> p, Point_d<Kernel> q);

introduces a ray through *p* and *q* and starting at *p*.

Precondition: *p* and *q* are distinct and have the same dimension.

Precondition: *p.dimension()==q.dimension()*.

Ray_d<Kernel> r(Point_d<Kernel> p, Direction_d<Kernel> dir);

introduces a ray starting in *p* with direction *dir*.

Precondition: *p* and *dir* have the same dimension and *dir* is not degenerate.

Precondition: *p.dimension()==dir.dimension()*.

Ray_d<Kernel> r(Segment_d<Kernel> s);

introduces a ray through *s.source()* and *s.target()* and starting at *s.source()*.

Precondition: *s* is not degenerate.

Operations

int *r.dimension()* returns the dimension of the ambient space.

Point_d<Kernel> *r.source()* returns the source point of *r*.

<i>Point_d<Kernel></i>	<i>r.point(int i)</i>	returns a point on <i>r</i> . <i>point(0)</i> is the source. <i>point(i)</i> , with $i > 0$, is different from the source. <i>Precondition: $i \geq 0$.</i>
<i>Direction_d<Kernel></i>	<i>r.direction()</i>	returns the direction of <i>r</i> .
<i>Line_d<Kernel></i>	<i>r.supporting_line()</i>	returns the supporting line of <i>r</i> .
<i>Ray_d<Kernel></i>	<i>r.opposite()</i>	returns the ray with direction opposite to <i>r</i> and starting in <i>source</i> .
<i>Ray_d<Kernel></i>	<i>r.transform(Aff_transformation_d<Kernel> t)</i>	returns $t(r)$. <i>Precondition: $r.dimension()=t.dimension()$.</i>
<i>Ray_d<Kernel></i>	<i>r + Vector_d<Kernel> v</i>	returns $r+v$, i.e., <i>r</i> translated by vector <i>v</i> . <i>Precondition: $r.dimension()=v.dimension()$.</i>
<i>bool</i>	<i>r.has_on(Point_d<Kernel> p)</i>	A point is on <i>r</i> , iff it is equal to the source of <i>r</i> , or if it is in the interior of <i>r</i> . <i>Precondition: $r.dimension()=p.dimension()$.</i>

Non-Member Functions

<i>bool</i>	<i>parallel(r1, r2)</i>	returns true if the unoriented supporting lines of <i>r1</i> and <i>r2</i> are parallel and false otherwise. <i>Precondition: $r1.dimension()=r2.dimension()$.</i>
-------------	---------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Implementation

Rays are implemented by a pair of points as an item type. All operations like creation, initialization, tests, direction calculation, input and output on a ray *r* take time $O(r.dimension())$. *dimension()*, coordinate and point access, and identity test take constant time. The space requirement is $O(r.dimension())$.

CGAL::Segment_d<Kernel>

Definition

An instance s of the data type *Segment_d* is a directed straight line segment in d -dimensional Euclidean space connecting two points p and q . p is called the source point and q is called the target point of s , both points are called endpoints of s . A segment whose endpoints are equal is called *degenerate*.

Types

Segment_d<Kernel>::R the representation type.

$$Segment_d<Kernel>:: RT \quad \text{the ring type.}$$

$Segment_d<Kernel>:: FT$ the field type.

Segment_d<Kernel>::LA the linear algebra layer.

Creation

Segment_d<Kernel> *s*;

introduces a variable *s* of type *Segment_d<Kernel>*.

Segment_d<Kernel> *s* (*Point_d<Kernel>* *p*, *Point_d<Kernel>* *q*);

introduces a variable *s* of type *Segment_d<Kernel>* which is initialized to the segment (*p*,*q*).

Precondition: *p.dimension()==q.dimension()*.

Segment_d<Kernel> *s* (*Point_d<Kernel>* *p*, *Vector_d<Kernel>* *v*);

introduces a variable *s* of type *Segment_d<Kernel>* which is initialized to the segment (*p*,*p*+*v*).

Precondition: *p.dimension()*==*v.dimension()*.

Operations

<i>int</i>	<i>s.dimension()</i>	returns the dimension of the ambient space.
------------	----------------------	---------------------------------------------

<i>Point_d<Kernel></i>	<i>s.source()</i>	returns the source point of segment <i>s</i> .
------------------------------	-------------------	------------------------------------------------

<i>Point_d</i> < <i>Kernel</i> >	<i>s.target()</i>	returns the target point of segment <i>s</i> .
----------------------------------	-------------------	------------------------------------------------

<i>Point_d<Kernel></i>	<i>s.vertex(int i)</i>	returns source or target of <i>s</i> : <i>vertex(0)</i> returns the source, <i>vertex(1)</i> returns the target. The parameter <i>i</i> is taken modulo 2, which gives easy access to the other vertex. <i>Precondition: $i \geq 0$.</i>
<i>Point_d<Kernel></i>	<i>s.point(int i)</i>	returns <i>vertex(i)</i> .
<i>Point_d<Kernel></i>	<i>s[int i]</i>	returns <i>vertex(i)</i> .
<i>Point_d<Kernel></i>	<i>s.min()</i>	returns the lexicographically smaller vertex.
<i>Point_d<Kernel></i>	<i>s.max()</i>	returns the lexicographically larger vertex.
<i>Segment_d<Kernel></i>	<i>s.opposite()</i>	returns the segment (<i>target()</i> , <i>source()</i>).
<i>Direction_d<Kernel></i>	<i>s.direction()</i>	returns the direction from source to target. <i>Precondition: s is non-degenerate.</i>
<i>Vector_d<Kernel></i>	<i>s.vector()</i>	returns the vector from source to target.
<i>FT</i>	<i>s.squared_length()</i>	returns the square of the length of <i>s</i> .
<i>bool</i>	<i>s.has_on(Point_d<Kernel> p)</i>	returns true if <i>p</i> lies on <i>s</i> and false otherwise. <i>Precondition: $s.dimension()=p.dimension()$.</i>
<i>Line_d<Kernel></i>	<i>s.supporting_line()</i>	returns the supporting line of <i>s</i> . <i>Precondition: s is non-degenerate.</i>
<i>Segment_d<Kernel></i>	<i>s.transform(Aff_transformation_d<Kernel> t)</i>	returns <i>t(s)</i> . <i>Precondition: $s.dimension()=t.dimension()$.</i>
<i>Segment_d<Kernel></i>	<i>s + Vector_d<Kernel> v</i>	returns <i>s + v</i> , i.e., <i>s</i> translated by vector <i>v</i> . <i>Precondition: $s.dimension()=v.dimension()$.</i>
<i>bool</i>	<i>s.is_degenerate()</i>	returns true if <i>s</i> is degenerate i.e. <i>s.source()=s.target()</i> .

Non-Member Functions

<i>bool</i>	<i>weak_equality(s1, s2)</i>	Test for equality as unoriented segments. <i>Precondition: $s1.dimension()=s2.dimension()$.</i>
-------------	-------------------------------	---------------------------------------------------------------------------------------------------------------

<i>bool</i>	<i>parallel(s1, s2)</i>	return true if one of the segments is degenerate or if the un-oriented supporting lines are parallel. <i>Precondition: s1.dimension()==s2.dimension().</i>
<i>bool</i>	<i>common_endpoint(s1, s2, Point_d<Kernel>& common)</i>	if <i>s1</i> and <i>s2</i> touch in a common end point, this point is assigned to <i>common</i> and the result is <i>true</i> , otherwise the result is <i>false</i> . If <i>s1==s2</i> then one of the endpoints is returned. <i>Precondition: s1.dimension()==s2.dimension().</i>

Implementation

Segments are implemented by a pair of points as an item type. All operations like creation, initialization, tests, the calculation of the direction and source - target vector, input and output on a segment *s* take time $O(s.dimension())$. *dimension()*, coordinate and end point access, and identity test take constant time. The operations for intersection calculation also take time $O(s.dimension())$. The space requirement is $O(s.dimension())$.

CGAL::Hyperplane_d<Kernel>

Definition

An instance of data type *Hyperplane_d* is an oriented hyperplane in d - dimensional space. A hyperplane h is represented by coefficients (c_0, c_1, \dots, c_d) of type *RT*. At least one of c_0 to c_{d-1} must be non-zero. The plane equation is $\sum_{0 \leq i < d} c_i x_i + c_d = 0$, where x_0 to x_{d-1} are Cartesian point coordinates. For a particular x the sign of $\sum_{0 \leq i < d} c_i x_i + c_d$ determines the position of a point x with respect to the hyperplane (on the hyperplane, on the negative side, or on the positive side).

There are two equality predicates for hyperplanes. The (weak) equality predicate (*weak_equality*) declares two hyperplanes equal if they consist of the same set of points, the strong equality predicate (*operator==*) requires in addition that the negative halfspaces agree. In other words, two hyperplanes are strongly equal if their coefficient vectors are positive multiples of each other and they are (weakly) equal if their coefficient vectors are multiples of each other.

Types

Hyperplane_d<Kernel>:: RT

the ring type.

Hyperplane_d<Kernel>:: FT

the field type.

Hyperplane_d<Kernel>:: LA

the linear algebra layer.

Hyperplane_d<Kernel>:: Coefficient_const_iterator

a read-only iterator for the coefficients.

Creation

Hyperplane_d<Kernel> h; introduces a variable h of type *Hyperplane_d<Kernel>*.

template <class InputIterator>

Hyperplane_d<Kernel> h(int d, InputIterator first, InputIterator last, RT D);

introduces a variable h of type *Hyperplane_d<Kernel>* initialized to the hyperplane with coefficients set $[first, last)$ and D .

Precondition: $size [first, last) == d$.

Requirement: The value type of *InputIterator* is *RT*.

template <class InputIterator>

Hyperplane_d<Kernel> h(int d, InputIterator first, InputIterator last);

introduces a variable *h* of type *Hyperplane_d<Kernel>* initialized to the hyperplane with coefficients set *[first,last)*.

Precondition: size *[first,last) == d+1*.

Requirement: The value type of *InputIterator* is *RT*.

template <class ForwardIterator>

*Hyperplane_d<Kernel> h(ForwardIterator first,
ForwardIterator last,
Point_d<Kernel> o,
Oriented_side side = Oriented_side(0))*

constructs some hyperplane that passes through the points in set *[first,last)*. If *side* is *ON_POSITIVE_SIDE* or *ON_NEGATIVE_SIDE* then *o* is on that side of the constructed hyperplane.

Precondition: A hyperplane with the stated properties must exist.

Requirement: The value type of *ForwardIterator* is *Point_d<Kernel>*.

Hyperplane_d<Kernel> h(Point_d<Kernel> p, Direction_d<Kernel> dir);

constructs the hyperplane with normal direction *dir* that passes through *p*. The direction *dir* points into the positive side.

Precondition: *p.dimension() == dir.dimension()* and *dir* is not degenerate.

Hyperplane_d<Kernel> h(RT a, RT b, RT c);

introduces a variable *h* of type *Hyperplane_d<Kernel>* in 2-dimensional space with equation $ax + by + c = 0$.

Hyperplane_d<Kernel> h(RT a, RT b, RT c, RT d);

introduces a variable *h* of type *Hyperplane_d<Kernel>* in 3-dimensional space with equation $ax + by + cz + d = 0$.

Operations

int *h.dimension()* returns the dimension of *h*.

RT *h[int i]* returns the *i*-th coefficient of *h*.
Precondition: $0 \leq i \leq d$.

RT *h.coefficient(int i)*

returns the *i*-th coefficient of *h*.
Precondition: $0 \leq i \leq d$.

<i>Coefficient_const_iterator</i>	<i>h.coefficients_begin()</i>	returns an iterator pointing to the first coefficient.
<i>Coefficient_const_iterator</i>	<i>h.coefficients_end()</i>	returns an iterator pointing beyond the last coefficient.
<i>Vector_d<Kernel></i>	<i>h.orthogonal_vector()</i>	returns the orthogonal vector of <i>h</i> . It points from the negative halfspace into the positive halfspace and its homogeneous coordinates are $(c_0, \dots, c_{d-1}, 1)$.
<i>Direction_d<Kernel></i>	<i>h.orthogonal_direction()</i>	returns the orthogonal direction of <i>h</i> . It points from the negative halfspace into the positive halfspace.
<i>Oriented_side</i>	<i>h.oriented_side(Point_d<Kernel> p)</i>	returns the side of the hyperplane <i>h</i> containing <i>p</i> . <i>Precondition:</i> <i>h.dimension() == p.dimension()</i> .
<i>bool</i>	<i>h.has_on(Point_d<Kernel> p)</i>	returns true iff point <i>p</i> lies on the hyperplane <i>h</i> . <i>Precondition:</i> <i>h.dimension() == p.dimension()</i> .
<i>bool</i>	<i>h.has_on_boundary(Point_d<Kernel> p)</i>	returns true iff point <i>p</i> lies on the boundary of hyperplane <i>h</i> . <i>Precondition:</i> <i>h.dimension() == p.dimension()</i> .
<i>bool</i>	<i>h.has_on_positive_side(Point_d<Kernel> p)</i>	returns true iff point <i>p</i> lies on the positive side of hyperplane <i>h</i> . <i>Precondition:</i> <i>h.dimension() == p.dimension()</i> .
<i>bool</i>	<i>h.has_on_negative_side(Point_d<Kernel> p)</i>	returns true iff point <i>p</i> lies on the negative side of hyperplane <i>h</i> . <i>Precondition:</i> <i>h.dimension() == p.dimension()</i> .

Hyperplane_d<Kernel> *h.transform(Aff_transformation_d<Kernel> t)*

returns $t(h)$.

Precondition: $h.dimension() == t.dimension()$.

Non-Member Functions

bool *weak_equality(h1, h2)*

test for weak equality.

Precondition: $h1.dimension() == h2.dimension()$.

Implementation

Hyperplanes are implemented by arrays of integers as an item type. All operations like creation, initialization, tests, vector arithmetic, input and output on a hyperplane h take time $O(h.dimension())$. coordinate access and $dimension()$ take constant time. The space requirement is $O(h.dimension())$.

CGAL::Sphere_d<Kernel>

Definition

An instance S of the data type $Sphere_d$ is an oriented sphere in some d -dimensional space. A sphere is defined by $d + 1$ points (class $Point_d<Kernel>$). We use A to denote the array of the defining points. A set A of defining points is *legal* if either the points are affinely independent or if the points are all equal. Only a legal set of points defines a sphere in the geometric sense and hence many operations on spheres require the set of defining points to be legal. The orientation of S is equal to the orientation of the defining points, i.e., $orientation(A)$.

Types

$Sphere_d<Kernel>:: R$	the representation type.
$Sphere_d<Kernel>:: RT$	the ring type.
$Sphere_d<Kernel>:: FT$	the field type.
$Sphere_d<Kernel>:: LA$	the linear algebra layer.
$Sphere_d<Kernel>:: point_iterator$	a read-only iterator for points defining the sphere.

Creation

$Sphere_d<Kernel> S;$ introduces a variable S of type $Sphere_d<Kernel>$.

`template <class ForwardIterator>`

$Sphere_d<Kernel> S(int d, ForwardIterator first, ForwardIterator last);$

introduces a variable S of type $Sphere_d<Kernel>$. S is initialized to the sphere through the points in $A = tuple [first, last)$.

Precondition: A consists of $d + 1$ d -dimensional points.

Requirement: The value type of `ForwardIterator` is $Point_d<Kernel>$.

Operations

int $S.dimension()$ returns the dimension of the ambient space.

$Point_d<Kernel> S.point(int i)$ returns the i th defining point.
Precondition: $0 \leq i \leq dim$.

<i>point_iterator</i>	<i>S.points_begin()</i>	returns an iterator pointing to the first defining point.
<i>point_iterator</i>	<i>S.points_end()</i>	returns an iterator pointing beyond the last defining point.
<i>bool</i>	<i>S.is_degenerate()</i>	returns true iff the defining points are not full dimensional.
<i>bool</i>	<i>S.is_legal()</i>	returns true iff the set of defining points is legal. A set of defining points is legal iff their orientation is non-zero or if they are all equal.
<i>Point_d<Kernel></i>	<i>S.center()</i>	returns the center of <i>S</i> . <i>Precondition: S is legal.</i>
<i>FT</i>	<i>S.squared_radius()</i>	returns the squared radius of the sphere. <i>Precondition: S is legal.</i>
<i>Orientation</i>	<i>S.orientation()</i>	returns the orientation of <i>S</i> .
<i>Oriented_side</i>	<i>S.oriented_side(Point_d<Kernel> p)</i>	returns either the constant <i>ON_ORIENTED_BOUNDARY</i> , <i>ON_POSITIVE_SIDE</i> , or <i>ON_NEGATIVE_SIDE</i> , iff <i>p</i> lies on the boundary, properly on the positive side, or properly on the negative side of sphere, resp. <i>Precondition: S.dimension()==p.dimension().</i>
<i>Bounded_side</i>	<i>S.bounded_side(Point_d<Kernel> p)</i>	returns <i>ON_BOUNDED_SIDE</i> , <i>ON_BOUNDARY</i> , or <i>ON_UNBOUNDED_SIDE</i> iff <i>p</i> lies properly inside, on the boundary, or properly outside of sphere, resp. <i>Precondition: S.dimension()==p.dimension().</i>
<i>bool</i>	<i>S.has_on_positive_side(Point_d<Kernel> p)</i>	returns <i>S.oriented_side(p)==ON_POSITIVE_SIDE</i> . <i>Precondition: S.dimension()==p.dimension().</i>

S.has_on_negative_side(Point_d<Kernel> p)

returns $S.oriented_side(p) == ON_NEGATIVE_SIDE$.

Precondition: $S.dimension()=p.dimension()$.

$$S.has_on_boundary(Point_d<Kernel> p)$$

returns $S.oriented_side(p) == ON_ORIENTED_BOUNDARY$, which is the same as $S.bounded_side(p) == ON_BOUNDARY$.

Precondition: $S.dimension()=p.dimension()$.

S.has_on_bounded_side(Point_d<Kernel> p)

returns $S.bounded_side(p) == ON_BOUNDED_SIDE$.

Precondition: $S.dimension()=p.dimension()$.

$$S.has_on_unbounded_side(Point_d<Kernel> p)$$

returns $S.bounded_side(p) == ON_UNBOUNDED_SIDE$.

Precondition: $S.dimension()=p.dimension()$.

Sphere_d<Kernel>

S.opposite()

returns the sphere with the same center and squared radius as S but with opposite orientation.

Sphere_d<Kernel>

$$S + Vector_d \langle Kernel \rangle v$$

returns the sphere translated by v .

Precondition: $S.dimension() == v.dimension()$.

Non-Member Functions

```
bool      weak_equality( S1, S2)
```

Test for equality as unoriented spheres.

Precondition: $S1.dimension()=S2.dimension()$.

Implementation

Spheres are implemented by a vector of points as a handle type. All operations like creation, initialization, tests, input and output of a sphere s take time $O(s.dimension())$. $dimension()$, point access take constant time. The $center()$ -operation takes time $O(d^3)$ on its first call and constant time thereafter. The sidedness and orientation tests take time $O(d^3)$. The space requirement for spheres is $O(s.dimension())$ neglecting the storage room of the points.

CGAL::Iso_box_d<Kernel>

Definition

An object b of the data type $Iso_box_d<Kernel>$ is an iso-box in the Euclidean space \mathbb{E}^d with edges parallel to the axes of the coordinate system.

Creation

$Iso_box_d<Kernel> \ b(\text{const } Point_d<Kernel>\& \ p, \text{const } Point_d<Kernel>\& \ q);$

introduces an iso-oriented iso-box b with diagonal opposite vertices p and q .

Operations

$bool \qquad b.operator==(\text{const } Iso_box_d<Kernel>\& \ b2) \text{ const}$

Test for equality: two iso-oriented cuboid are equal, iff their lower left and their upper right vertices are equal.

$bool \qquad b.operator!=(\text{const } Iso_box_d<Kernel>\& \ b2) \text{ const}$

Test for inequality.

$\text{const } Point_d<Kernel>\& \quad b.min() \text{ const}$ returns the smallest vertex of b .

$\text{const } Point_d<Kernel>\& \quad b.max() \text{ const}$ returns the largest vertex of b .

Predicates

$bool \qquad b.is_degenerate() \text{ const}$

b is degenerate, if all vertices are collinear.

$Bounded_side \quad b.bounded_side(\text{const } Point_d<Kernel>\& \ p) \text{ const}$

returns either $ON_UNBOUNDED_SIDE$, $ON_BOUNDED_SIDE$, or the constant $ON_BOUNDARY$, depending on where point p is.

$bool \quad b.has_on_boundary(\text{const } Point_d<Kernel>\& \ p) \text{ const}$

$bool \quad b.has_on_bounded_side(\text{const } Point_d<Kernel>\& \ p) \text{ const}$

$bool \quad b.has_on_unbounded_side(\text{const } Point_d<Kernel>\& \ p) \text{ const}$

Miscellaneous

Kernel::FT

b.volume() const

returns the volume of *b*.

CGAL::Aff_transformation_d<Kernel>

Definition

An instance of the data type *Aff_transformation_d<Kernel>* is an affine transformation of d -dimensional space. It is specified by a square matrix M of dimension $d + 1$. All entries in the last row of M except the diagonal entry must be zero; the diagonal entry must be non-zero. A point p with homogeneous coordinates $(p[0], \dots, p[d])$ can be transformed into the point $p.transform(A) = Mp$, where A is an affine transformation created from M by the constructors below.

Types

Aff_transformation_d<Kernel>:: RT

the ring type.

Aff_transformation_d<Kernel>:: FT

the field type.

Aff_transformation_d<Kernel>:: LA

the linear algebra layer.

Aff_transformation_d<Kernel>:: Matrix

the matrix type.

Creation

Aff_transformation_d<Kernel> t;

introduces some transformation.

Aff_transformation_d<Kernel> t(int d, Identity_transformation);

introduces the identity transformation in d -dimensional space.

Aff_transformation_d<Kernel> t(Matrix M);

introduces the transformation of d -space specified by matrix M .

Precondition: M is a square matrix of dimension $d + 1$ where entries in the last row of M except the diagonal entry must be zero; the diagonal entry must be non-zero.

template <typename Forward_iterator>

Aff_transformation_d<Kernel> t(Scaling, Forward_iterator start, Forward_iterator end);

introduces the transformation of d -space specified by a diagonal matrix with entries *set* $[start, end)$ on the diagonal (a scaling of the space).

Precondition: *set* $[start, end)$ is a vector of dimension $d + 1$.

Aff_transformation_d<Kernel> t(Translation, Vector_d<Kernel> v);

introduces the translation by vector v .

Aff_transformation_d<Kernel> t(int d, Scaling, RT num, RT den);

returns a scaling by a scale factor num/den .

Precondition: $den \neq 0$.

Aff_transformation_d<Kernel> t(int d, Rotation, RT sin_num, RT cos_num, RT den, int e1 = 0, int e2 = 1);

returns a planar rotation with sine and cosine values sin_num/den and cos_num/den in the plane spanned by the base vectors b_{e1} and b_{e2} in d -space. Thus the default use delivers a planar rotation in the x - y plane.

Precondition: $sin_num^2 + cos_num^2 = den^2$ and $0 \leq e_1 < e_2 < d$.

Precondition: $den \neq 0$

*Aff_transformation_d<Kernel> t(int d,
Rotation,
Direction_d<Kernel> dir,
RT num,
RT den,
int e1 = 0,
int e2 = 1)*

returns a planar rotation within a two-dimensional linear subspace. The subspace is spanned by the base vectors b_{e1} and b_{e2} in d -space. The rotation parameters are given by the 2-dimensional direction *dir*, such that the difference between the sines and cosines of the rotation given by *dir* and the approximated rotation are at most num/den each.

Precondition: $dir.dimension()=2$, $!dir.is_degenerate()$ and $num < den$ is positive, $den \neq 0$, $0 \leq e_1 < e_2 < d$.

Operations

<i>int</i>	<i>t.dimension()</i>	the dimension of the underlying space
<i>Matrix</i>	<i>t.matrix()</i>	returns the transformation matrix
<i>Aff_transformation_d<Kernel></i>	<i>t.inverse()</i>	returns the inverse transformation. <i>Precondition:</i> <i>t.matrix()</i> is invertible.

Aff_transformation_d<Kernel>

$t * s$

composition of transformations. Note that transformations are not necessarily commutative. $t*s$ is the transformation which transforms first by t and then by s .

Implementation

Affine Transformations are implemented by matrices of number type RT as a handle type. All operations like creation, initialization, input and output on a transformation t take time $O(t.dimension()^2)$. $dimension()$ takes constant time. The operations for inversion and composition have the cubic costs of the used matrix operations. The space requirement is $O(t.dimension()^2)$.

3.7 Global Kernel Functions

CGAL::affinely_independent

```
#include <CGAL/predicates_d.h>
```

```
template <class ForwardIterator>
```

```
bool          affinely_independent( ForwardIterator first, ForwardIterator last)
```

returns true iff the points in $A = \text{tuple } [first, last)$ are affinely independent.

Precondition: The objects are of the same dimension.

Requirement: The value type of *ForwardIterator* is *Point_d<R>*

CGAL::affine_rank

```
#include <CGAL/predicates_d.h>
```

```
template <class ForwardIterator>
```

```
int affine_rank( ForwardIterator first, ForwardIterator last)
```

computes the affine rank of the points in $A = \text{tuple}[first, last)$.

Precondition: The objects in A are of the same dimension.

Requirement: The value type of *ForwardIterator* is *Point_d<R>*.

CGAL::center_of_sphere

```
#include <CGAL/constructions_d.h>
```

```
template <class ForwardIterator>
```

```
Point_d<R> center_of_sphere( ForwardIterator first, ForwardIterator last)
```

returns the center of the sphere spanned by the points in $A = \text{tuple}[first, last)$.

Precondition: A contains $d + 1$ affinely independent points of dimension d .

Requirement: The value type of *ForwardIterator* is *Point_d<R>*.

CGAL::compare_lexicographically

```
#include <CGAL/predicates_d.h>
```

```
Comparison_result    compare_lexicographically( Point_d<R> p, Point_d<R> q)
```

Compares the Cartesian coordinates of points p and q lexicographically in ascending order of its Cartesian components $p[i]$ and $q[i]$ for $i = 0, \dots, d - 1$.

Precondition: $p.dimension() == q.dimension()$

CGAL::contained_in_affine_hull

```
#include <CGAL/predicates_d.h>
```

```
template <class ForwardIterator>
```

```
bool contained_in_affine_hull( ForwardIterator first,  
                              ForwardIterator last,  
                              Point_d<R> p)
```

determines whether p is contained in the affine hull of the points in $A = \text{tuple } [first, last)$.

Precondition: The objects in A are of the same dimension.

Requirement: The value type of *ForwardIterator* is *Point_d<R>*.

CGAL::contained_in_linear_hull

```
#include <CGAL/predicates_d.h>
```

```
template <class ForwardIterator>
```

```
bool contained_in_linear_hull( ForwardIterator first,  
                              ForwardIterator last,  
                              Vector_d<R> v)
```

determines whether v is contained in the linear hull of the vectors in $A = \text{tuple } [first, last)$.

Precondition: The objects in A are of the same dimension.

Requirement: The value type of *ForwardIterator* is *Vector_d<R>*.

CGAL::contained_in_simplex

```
#include <CGAL/predicates_d.h>
```

```
template <class ForwardIterator>
```

```
bool contained_in_simplex( ForwardIterator first, ForwardIterator last, Point_d<R> p)
```

determines whether p is contained in the simplex of the points in $A = \text{tuple } [first, last)$.

Precondition: The objects in A are of the same dimension and affinely independent.

Requirement: The value type of *ForwardIterator* is *Point_d<R>*.

CGAL::do_intersect

```
#include <CGAL/intersections_d.h>
```

```
bool do_intersect( Type1<R> obj1, Type2<R> obj2)
```

checks whether *obj1* and *obj2* intersect. Two objects *obj1* and *obj2* intersect if there is a point *p* that is part of both *obj1* and *obj2*. The intersection region of those two objects is defined as the set of all points *p* that are part of both *obj1* and *obj2*.

Precondition: the objects are of the same dimension.

The types *Type1* and *Type2* can be any of the following:

- *Point_d<R>*
- *Line_d<R>*
- *Ray_d<R>*
- *Segment_d<R>*
- *Hyperplane_d<R>*

See Also

intersection

CGAL::intersection

```
#include <CGAL/intersections_d.h>
```

Object *intersection*(*Type1*<*R*> *f1*, *Type2*<*R*> *f2*)

returns the intersection result of *f1* and *f2* by means of the polymorphic wrapper type *Object*. The returned object can be tested for the intersection result and assigned by means of the operation *bool assign*(*T*& *t*, *Object* *o*).

Precondition: The objects are of the same dimension.

The possible value for types *Type1* and *Type2* and the possible return values wrapped in *Object* are the following:

Type1	Type2	Return Type
<i>Line_d</i>	<i>Line_d</i>	<i>Point_d</i> , <i>Line_d</i>
<i>Segment_d</i>	<i>Line_d</i>	<i>Point_d</i> , <i>Segment_d</i>
<i>Segment_d</i>	<i>Segment_d</i>	<i>Point_d</i> , <i>Segment_d</i>
<i>Ray_d</i>	<i>Line_d</i>	<i>Point_d</i> , <i>Ray_d</i>
<i>Ray_d</i>	<i>Segment_d</i>	<i>Point_d</i> , <i>Segment_d</i>
<i>Ray_d</i>	<i>Ray_d</i>	<i>Point_d</i> , <i>Segment_d</i> , <i>Ray_d</i>
<i>Hyperplane_d</i>	<i>Line_d</i>	<i>Point_3</i> , <i>Line_3</i>
<i>Hyperplane_d</i>	<i>Ray_d</i>	<i>Point_d</i> , <i>Ray_d</i>
<i>Hyperplane_d</i>	<i>Segment_d</i>	<i>Point_d</i> , <i>Segment_d</i>

Example

The following example demonstrates the most common use of *intersection* routines.

```
#include <CGAL/intersections_d.h>

template <class R>
void foo(Segment_d<R> seg, Line_d<R> lin)
{
    Point_d<R> ipnt; Segment_d<R> iseg;
    Object result = intersection(seg, lin);
    if ( assign(ipnt, result) ) {
        // handle the point intersection case.
    } else if ( assign(iseg, result) ) {
        // handle the segment intersection case.
    } else {
        // handle the no intersection case.
    }
}
```

See Also

do_intersect, *Kernel::Intersect_d*

CGAL::lexicographically_smaller

```
#include <CGAL/predicates_d.h>
```

```
bool                                lexicographically_smaller( Point_d<R> p, Point_d<R> q)
```

returns *true* iff *p* is lexicographically smaller than *q* with respect to Cartesian lexicographic order of points.

Precondition: *p.dimension()* == *q.dimension()*.

CGAL::lexicographically_smaller_or_equal

```
#include <CGAL/predicates_d.h>
```

```
bool lexicographically_smaller_or_equal( Point_d<R> p, Point_d<R> q)
```

returns *true* iff p is lexicographically smaller than q with respect to Cartesian lexicographic order of points or equal to q .

Precondition: $p.\text{dimension}() == q.\text{dimension}()$.

CGAL::lift_to_paraboloid

```
#include <CGAL/constructions_d.h>
```

```
Point_d<R> lift_to_paraboloid( Point_d<R> p)
```

returns the projection of $p = (x_0, \dots, x_{d-1})$ onto the paraboloid of revolution which is the point $(p_0, \dots, p_{d-1}, \sum_{0 \leq i < d} p_i^2)$ in $(d+1)$ -space.

CGAL::linearly_independent

```
#include <CGAL/predicates_d.h>
```

```
template <class ForwardIterator>
```

```
bool linearly_independent( ForwardIterator first, ForwardIterator last)
```

decides whether the vectors in $A = \text{tuple } [first, last)$ are linearly independent.

Precondition: The objects in A are of the same dimension.

Requirement: The value type of *ForwardIterator* is *Vector_d<R>*.

CGAL::linear_base

```
#include <CGAL/constructions_d.h>
```

```
template <class ForwardIterator, class OutputIterator>
```

```
OutputIterator linear_base( ForwardIterator first, ForwardIterator last, OutputIterator result)
```

computes a basis of the linear space spanned by the vectors in $A = \text{tuple } [first, last)$ and returns it via an iterator range starting in *result*. The returned iterator marks the end of the output.

Precondition: A contains vectors of the same dimension d .

Requirement: The value type of *ForwardIterator* and *OutputIterator* is $\text{Vector}_d<R>$.

CGAL::linear_rank

```
#include <CGAL/predicates_d.h>
```

```
template <class ForwardIterator>
```

```
int linear_rank( ForwardIterator first, ForwardIterator last)
```

computes the linear rank of the vectors in $A = \text{tuple}[first, last)$.

Precondition: The objects are of the same dimension.

Requirement: The value type of *ForwardIterator* is *Vector_d<R>*.

CGAL::midpoint

```
#include <CGAL/constructions_d.h>
```

```
Point_d<R> midpoint( Point_d<R> p, Point_d<R> q)
```

computes the midpoint of the segment pq .

Precondition:

Precondition: $p.dimension() == q.dimension()$.

CGAL::orientation

```
#include <CGAL/predicates_d.h>
```

```
template <class ForwardIterator>
```

```
Orientation orientation( ForwardIterator first, ForwardIterator last)
```

determines the orientation of the points of the tuple $A = \text{tuple } [first, last)$ where A consists of $d + 1$ points in d -space. This is the sign of the determinant

$$\begin{vmatrix} 1 & 1 & 1 & 1 \\ A[0] & A[1] & \dots & A[d] \end{vmatrix}$$

where $A[i]$ denotes the Cartesian coordinate vector of the i -th point in A .

Precondition: $\text{size } [first, last) == d + 1$ and $A[i].\text{dimension}() == d \forall 0 \leq i \leq d$.

Requirement: The value type of *ForwardIterator* is *Point_d<R>*.

CGAL::project_along_d_axis

```
#include <CGAL/constructions_d.h>
```

```
Point_d<R> project_along_d_axis( Point_d<R> p)
```

returns p projected along the d -axis onto the hyperspace spanned by the first $d - 1$ standard base vectors.

CGAL::side_of_bounded_sphere

```
#include <CGAL/predicates_d.h>
```

```
template <class ForwardIterator>
```

```
Bounded_side      side_of_bounded_sphere( ForwardIterator first,  
                                           ForwardIterator last,  
                                           Point_d<R> p)
```

returns the relative position of point p to the sphere defined by $A = \text{tuple } [first, last)$. The order of the points of A does not matter.

Precondition: $\text{orientation}(first, last)$ is not *ZERO*.

Requirement: The value type of *ForwardIterator* is *Point_d<R>*.

CGAL::side_of_oriented_sphere

```
#include <CGAL/predicates_d.h>
```

```
template <class ForwardIterator>
```

```
Oriented_side      side_of_oriented_sphere( ForwardIterator first,
                                             ForwardIterator last,
                                             Point_d<R> p)
```

returns the relative position of point p to the oriented sphere defined by the points in $A = \text{tuple } [first, last)$. The order of the points in A is important, since it determines the orientation of the implicitly constructed sphere. If the points in A are positively oriented, the positive side is the bounded interior of the sphere.

Precondition: A contains $d + 1$ points in d -space.

Requirement: The value type of *ForwardIterator* is *Point_d<R>*.

CGAL::squared_distance

```
#include <CGAL/constructions_d.h>
```

```
FT squared_distance( Point_d<R> p, Point_d<R> q)
```

computes the square of the Euclidean distance between the two points p and q .

Precondition: The dimensions of p and q are the same.

3.8 Kernel Concept

Kernel_d

The concept of a *kernel* is defined by a set of requirements on the provision of certain types and access member functions to create objects of these types. The types are function object classes to be used within the algorithms and data structures in the basic library of CGAL. This allows you to use any model of a kernel as a traits class in the CGAL algorithms and data structures, unless they require types beyond those provided by a kernel.

Kernel_d subsumes the concept of a *d-dimensional kernel*.

A kernel provides types, construction objects, and generalized predicates. The former replace constructors of the kernel classes and constructive procedures in the kernel. There are also function objects replacing operators, especially for equality testing.

Types

<i>Kernel_d:: FT</i>	a number type that is a model for <i>FieldNumberType</i>
<i>Kernel_d:: RT</i>	a number type that is a model for <i>RingNumberType</i>

Coordinate Access

<i>Kernel_d:: Cartesian_const_iterator_d</i>	a type that allows to iterate over the Cartesian coordinates
----------------------------------------------	--------------------------------------------------------------

Geometric Objects

Kernel_d:: Point_d
Kernel_d:: Vector_d
Kernel_d:: Direction_d
Kernel_d:: Hyperplane_d
Kernel_d:: Line_d
Kernel_d:: Ray_d
Kernel_d:: Segment_d
Kernel_d:: Iso_box_d
Kernel_d:: Sphere_d
Kernel_d:: Aff_transformation_d

Constructions

Kernel_d:: Construct_point_d
Kernel_d:: Construct_vector_d
Kernel_d:: Construct_direction_d
Kernel_d:: Construct_hyperplane_d
Kernel_d:: Construct_segment_d
Kernel_d:: Construct_iso_box_d
Kernel_d:: Construct_line_d

Kernel_d:: Construct_ray_d
Kernel_d:: Construct_sphere_d
Kernel_d:: Construct_aff_transformation_d
Kernel_d:: Construct_cartesian_const_iterator_d

Generalized Predicates

Kernel_d:: Affine_rank_d
Kernel_d:: Affinely_independent_d
Kernel_d:: Barycentric_coordinates_d
Kernel_d:: Center_of_sphere_d
Kernel_d:: Compare_lexicographically_d
Kernel_d:: Component_accessor_d
Kernel_d:: Contained_in_affine_hull_d
Kernel_d:: Contained_in_linear_hull_d
Kernel_d:: Contained_in_simplex_d
Kernel_d:: Equal_d
Kernel_d:: Has_on_positive_side_d
Kernel_d:: Intersect_d
Kernel_d:: Less_lexicographically_d
Kernel_d:: Less_or_equal_lexicographically_d
Kernel_d:: Lift_to_paraboloid_d
Kernel_d:: Linear_base_d
Kernel_d:: Linear_rank_d
Kernel_d:: Linearly_independent_d
Kernel_d:: Midpoint_d
Kernel_d:: Orientation_d
Kernel_d:: Oriented_side_d
Kernel_d:: Orthogonal_vector_d
Kernel_d:: Point_of_sphere_d
Kernel_d:: Point_to_vector_d
Kernel_d:: Position_on_line_d
Kernel_d:: Project_along_d_axis_d
Kernel_d:: Side_of_bounded_sphere_d
Kernel_d:: Side_of_oriented_sphere_d
Kernel_d:: Squared_distance_d
Kernel_d:: Value_at_d
Kernel_d:: Vector_to_point_d

Operations

The following member functions return function objects of the types listed above. The name of the access function is the name of the type returned with an *_object* suffix and no capital letter at the beginning. We only give two examples to show the scheme. For the functors *Construct_point_d* and *Orientation_d* the corresponding functions are:

Kernel::Construct_point_d *kernel.construct_point_d_object()*
Kernel::Orientation_d *kernel.orientation_d_object()*

Has Models

Cartesian_d<FieldNumberType>, Homogeneous_d<RingNumberType>

Kernel::Affinely_independent_d

A model for this must provide:

```
template <class ForwardIterator>
bool fo( ForwardIterator first, ForwardIterator last)
```

returns true iff the points in $A = \text{tuple } [first, last)$ are affinely independent.

Precondition: The objects are of the same dimension.

Requirement: The value type of *ForwardIterator* is *Kernel::Point_d*.

Kernel::Affine_rank_d

A model for this must provide:

```
template <class ForwardIterator>
int fo( ForwardIterator first, ForwardIterator last)
```

computes the affine rank of the points in $A = \text{tuple}[first, last)$.

Precondition: The objects are of the same dimension.

Requirement: The value type of *ForwardIterator* is *Kernel::Point_d*.

Kernel::CartesianConstIterator_d

A type representing an iterator to the Cartesian coordinates of a point in d dimensions.

Refines

CopyConstructible, Assignable, DefaultConstructible

Is Model for the Concepts

BidirectionalIterator

See Also

Kernel::ConstructCartesianConstIterator_d page [507](#)

Kernel::Center_of_sphere_d

A model for this must provide:

```
template <class ForwardIterator>
```

```
Kernel::Point_d fo( ForwardIterator first, ForwardIterator last)
```

computes the affine rank of the points in $A = \text{tuple}[first, last)$.

Precondition: The objects are of the same dimension.

Requirement: The value type of *ForwardIterator* is *Kernel::Point_d*.

Kernel::Compare_lexicographically_d

A model for this must provide:

Comparison_result *fo*(*Kernel::Point_d* *p*, *Kernel::Point_d* *q*)

Compares the Cartesian coordinates of points *p* and *q* lexicographically in ascending order of its Cartesian components *p*[*i*] and *q*[*i*] for *i* = 0, ..., *d* − 1.

Precondition: The objects are of the same dimension.

Kernel::Component_accessor_d

A model for this must provide:

<i>int</i>	<i>fo.dimension(Kernel::Point_d p)</i>	returns the dimension of <i>p</i> .
<i>Kernel::RT</i>	<i>fo.homogeneous(Kernel::Point_d p, int i)</i>	returns the <i>i</i> th homogeneous coordinate of <i>p</i> . <i>Precondition: $0 \leq i \leq \text{dimension}(p)$.</i>
<i>Kernel::FT</i>	<i>fo.cartesian(Kernel::Point_d p, int i)</i>	returns the <i>i</i> th Cartesian coordinate of <i>p</i> . <i>Precondition: $0 \leq i < \text{dimension}(p)$.</i>

Kernel::ConstructCartesianConstIterator_d

A model for this must provide:

Kernel::Cartesian_const_iterator_d

fo(Kernel::Point_d p)

returns an iterator on the 0'th Cartesian coordinate of *p*.

Kernel::Cartesian_const_iterator_d

fo(Kernel::Point_d p, int)

returns the past the end iterator of the Cartesian coordinates of *p*.

Refines

AdaptableFunctor (with one argument)

See Also

Kernel::CartesianConstIterator_d page [503](#)

Kernel::Contained_in_affine_hull_d

A model for this must provide:

```
template <class ForwardIterator>
```

```
Bounded_side          fo( ForwardIterator first, ForwardIterator last, Kernel::Point_d p)
```

determines whether p is contained in the affine hull of the points in $A = \text{tuple } [first, last)$.

Precondition: The objects are of the same dimension.

Requirement: The value type of *ForwardIterator* is *Kernel::Point_d*.

Kernel::Contained_in_linear_hull_d

A model for this must provide:

```
template <class ForwardIterator>
```

```
Bounded_side          fo( ForwardIterator first, ForwardIterator last, Kernel::Vector_d v)
```

determines whether v is contained in the linear hull of the vectors in $A = \text{tuple } [first, last)$.

Precondition: The objects are of the same dimension.

Requirement: The value type of *ForwardIterator* is *Kernel::Vector_d*.

Kernel::Contained_in_simplex_d

A model for this must provide:

```
template <class ForwardIterator>
```

```
Bounded_side          fo( ForwardIterator first, ForwardIterator last, Kernel::Point_d p)
```

determines whether p is contained in the simplex of the points in $A = \text{tuple } [first, last)$.

Precondition: The objects in A are of the same dimension and affinely independent.

Requirement: The value type of *ForwardIterator* is *Kernel::Point_d*.

Kernel::Equal_d

A model for this must provide:

bool

fo(Kernel::Point_d p, Kernel::Point_d q)

returns true iff p and q are equal (as d -dimensional points).

Precondition: p and q have the same dimension.

Kernel::Has_on_positive_side_d

A model for this must provide:

```
template <class Kernel_object>
bool fo( Kernel_object o, Kernel::Point_d p)
```

returns true iff p is on the positive side of o .
 $Kernel_object$ may be any of $Kernel::Sphere_d$,
 $Kernel::Hyperplane_d$.
Precondition: p and o have the same dimension.

Kernel::Intersect_d

A model for this must provide:

```
template <class Kernel_object>
Object      fo( Kernel_object p, Kernel_object q)
```

returns the result of the intersection of p and q in form of a polymorphic object. *Kernel_object* may be any of *Kernel::Segment_d*, *Kernel::Ray_d*, *Kernel::Line_d*, *Kernel::Hyperplane_d*.

Precondition: p and q have the same dimension.

Kernel::Less_lexicographically_d

A model for this must provide:

bool *fo(Kernel::Point_d p, Kernel::Point_d q)*

returns *true* iff *p* is lexicographically smaller than *q* with respect to Cartesian lexicographic order of points.
Precondition: *p* and *q* have the same dimension.

Kernel::Less_or_equal_lexicographically_d

A model for this must provide:

bool *fo*(*Kernel::Point_d* *p*, *Kernel::Point_d* *q*)

returns *true* iff *p* is lexicographically smaller than *q* with respect to Cartesian lexicographic order of points or equal to *q*.

Precondition: *p* and *q* have the same dimension.

Kernel::Lift_to_paraboloid_d

A model for this must provide:

Kernel::Point_d \rightarrow *fo(Kernel::Point_d p)*

returns $p = (x_0, \dots, x_{d-1})$ lifted to the paraboloid of revolution which is the point $(p_0, \dots, p_{d-1}, \sum_{0 \leq i < d} p_i^2)$ in $(d+1)$ -space.

Kernel::Linearly_independent_d

A model for this must provide:

```
template <class ForwardIterator>
bool fo( ForwardIterator first, ForwardIterator last)
```

decides whether the vectors in $A = \text{tuple } [first, last)$ are linearly independent.

Precondition: The objects in A are of the same dimension.

Requirement: The value type of *ForwardIterator* is *Kernel_d::Vector_d*.

Kernel::Linear_base_d

A model for this must provide:

```
template <class ForwardIterator, class OutputIterator>
int fo( ForwardIterator first, ForwardIterator last, OutputIterator result)
```

computes a basis of the linear space spanned by the vectors in $A = \text{tuple } [first, last)$ and returns it via an iterator range starting in *result*. The returned iterator marks the end of the output.

Precondition: A contains vectors of the same dimension d .

Requirement: The value type of *ForwardIterator* and *OutputIterator* is *Kernel::Vector_d*.

Kernel::Linear_rank_d

A model for this must provide:

```
template <class ForwardIterator>
```

```
int fo( ForwardIterator first, ForwardIterator last)
```

computes the linear rank of the vectors in $A = \text{tuple}[first, last)$.

Precondition:

Precondition: A contains vectors of the same dimension d .

Requirement: The value type of *ForwardIterator* is *Kernel::Vector_d*.

Kernel::Midpoint_d

A model for this must provide:

Kernel::Point_d *fo(Kernel::Point_d p, Kernel::Point_d q)*

computes the midpoint of the segment pq .

Precondition: p and q have the same dimension.

Kernel::Orientation_d

A model for this must provide:

```
template <class ForwardIterator>
```

```
Orientation fo( ForwardIterator first, ForwardIterator last)
```

determines the orientation of the points of the tuple $A = \text{tuple } [first, last)$ where A consists of $d + 1$ points in d -space. This is the sign of the determinant

$$\begin{vmatrix} 1 & 1 & 1 & 1 \\ A[0] & A[1] & \dots & A[d] \end{vmatrix}$$

where $A[i]$ denotes the cartesian coordinate vector of the i -th point in A .

Precondition: $\text{size } [first, last) == d + 1$ and $A[i].\text{dimension}() == d \forall 0 \leq i \leq d$.

Requirement: The value type of *ForwardIterator* is *Kernel::Point_d*.

Kernel::Oriented_side_d

A model for this must provide:

```
template <class Kernel_object>
Oriented_side fo( Kernel_object o, Kernel::Point_d p)
```

returns the side of p with respect to o . *Kernel_object* may be any of *Kernel::Sphere_d* or *Kernel::Hyperplane_d*.
Precondition: p and o have the same dimension.

Kernel::Orthogonal_vector_d

A model for this must provide:

Kernel::Vector_d *fo(Kernel::Hyperplane_d h)*
 computes an orthogonal vector to *h*.

Kernel::Point_of_sphere_d

A model for this must provide:

```
bool fo( Kernel::Sphere_d s, int i)
    returns the ith point defining the sphere s.
```

Kernel::Point_to_vector_d

A model for this must provide:

Kernel::Vector_d *fo(Kernel::Point_d p)*

converts p to its geometric vector.

Kernel::Project_along_d_axis_d

A model for this must provide:

Kernel::Point_d \rightarrow *fo(Kernel::Point_d p)*

returns p projected along the d -axis onto the hyperspace spanned by the first $d - 1$ standard base vectors.

Kernel::Side_of_bounded_sphere_d

A model for this must provide:

```
template <class ForwardIterator>
```

```
Bounded_side          fo( ForwardIterator first, ForwardIterator last, Kernel::Point_d p)
```

returns the relative position of point p to the sphere defined by $A = \text{tuple } [first, last]$. The order of the points of A does not matter.

Precondition: $\text{orientation}(first, last)$ is not ZERO.

Requirement: The value type of *ForwardIterator* is *Kernel::Point_d*.

Kernel::Side_of_oriented_sphere_d

A model for this must provide:

```
template <class ForwardIterator>
```

```
Bounded_side          fo( ForwardIterator first, ForwardIterator last, Kernel::Point_d p)
```

returns the relative position of point p to the oriented sphere defined by the points in $A = \text{tuple } [first, last)$. The order of the points in A is important, since it determines the orientation of the implicitly constructed sphere. If the points in A are positively oriented, the positive side is the bounded interior of the sphere.

Precondition: A contains $d + 1$ points in d -space.

Requirement: The value type of *ForwardIterator* is *Kernel::Point_d*.

Kernel::Squared_distance_d

A model for this must provide:

Kernel::FT *fo(Kernel::Point_d p, Kernel::Point_d q)*

computes the square of the Euclidean distance between the two points p and q .

Precondition: The dimensions of p and q are the same.

Kernel::Value_at_d

A model for this must provide:

Kernel::FT $fo(\textit{Kernel::Hyperplane_d } h, \textit{Kernel::Point_d } p)$

computes the value of h evaluated at p .

Precondition: p and h have the same dimension.

Kernel::Vector_to_point_d

A model for this must provide:

Kernel::Point_d *fo(Kernel::Vector_d v)*

converts v to the affine point $0 + v$.

Chapter 4

2D Circular Kernel

Sylvain Pion and Monique Teillaud

Contents

4.1	Introduction	533
4.2	Software Design	533
4.3	Examples	534
4.3.1	Computing an Arrangement of Random Circles	534
4.3.2	Constructing an Arrangement of Circles and Segments	535
4.3.3	Using the Predefined Kernel	537
4.4	Design and Implementation History	538

4.1 Introduction

The goal of the circular kernel is to offer to the user a large set of functionalities on circles and circular arcs in the plane. All the choices (interface, robustness, representation, and so on) made here are consistent with the choices made in the CGAL kernel, for which we refer the user to the 2D kernel manual.

In this first release, all functionalities necessary for computing an arrangement of circular arcs and these line segments are defined. Three traits classes are provided for the CGAL arrangement package.

4.2 Software Design

The design is done in such a way that the algebraic concepts and the geometric concepts are clearly separated. *Circular_kernel_2* has therefore two template parameters:

- the *LinearKernel*, from which the circular kernel derives, provides all elementary geometric objects like points, lines, circles, and elementary functionality on them. It must be a model of the CGAL two dimensional *Kernel* concept.
- the second parameter is the algebraic kernel, which is responsible for computations on polynomials and algebraic numbers. It has to be a model of concept *AlgebraicKernelForCircles*. The robustness of the package relies on the fact that the algebraic kernel provides exact computations on algebraic objects.

The circular kernel uses the extensibility scheme presented in the 2D kernel manual (see Section 2.5). The types of *LinearKernel* are inherited by the circular kernel and some types are taken from the *AlgebraicKernelForCircles* parameter. Three new main geometric objects are introduced by *Circular_kernel_2*: circular arcs, points of circular arcs (used in particular for endpoints of arcs and intersection points between arcs) and line segments whose endpoints are points of this new type.

In fact, the circular kernel is documented as a concept, *CircularKernel*, and two models are provided:

- *Circular_kernel_2<LinearKernel,AlgebraicKernelForCircles>*, the basic kernel,
- and a predefined filtered kernel *Exact_circular_kernel_2*, that is based on similar techniques as *Exact_predicates_exact_constructions_kernel*.

More work is in progress to increase the efficiency of this filtered kernel and provide other filtering techniques.

4.3 Examples

4.3.1 Computing an Arrangement of Random Circles

This example shows how to construct incrementally an arrangement of circles, using the traits class for arrangement of circular arcs provided with the package.

```
#include <CGAL/basic.h>
#include <CGAL/Cartesian.h>

#include <CGAL/Random.h>
#include <CGAL/point_generators_2.h>

#include <CGAL/MP_Float.h>

#include <CGAL/Algebraic_kernel_2_2.h>

#include <CGAL/Circular_kernel.h>

#include <CGAL/Arr_circular_arc_traits.h>

#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_naive_point_location.h>

typedef CGAL::Quotient<CGAL::MP_Float> NT;
typedef CGAL::Cartesian<NT> Linear_k;

typedef CGAL::Algebraic_kernel_for_circles_2_2<NT> Algebraic_k;
typedef CGAL::Circular_kernel_2<Linear_k,Algebraic_k> Circular_k;

typedef Circular_k::Point_2 Point_2;
typedef Circular_k::Circle_2 Circle_2;
typedef Circular_k::Circular_arc_2 Circular_arc_2;
typedef std::vector<Circular_arc_2> ArcContainer;
```

```

typedef CGAL::Arr_circular_arc_traits<Circular_k> Traits;

typedef CGAL::Arrangement_2<Traits> Arr;
typedef CGAL::Arr_naive_point_location<Arr> Point_location;

int main(){

    CGAL::Random generatorOfgenerator;
    int random_seed = generatorOfgenerator.get_int(0, 123456);
    std::cout << "random_seed = " << random_seed << std::endl;
    CGAL::Random theRandom(random_seed);
    int random_max = 128;
    int random_min = -128;
    ArcContainer ac;
    int x, y;

    for (int i = 0; i < 10; i++) {
        x = theRandom.get_int(random_min, random_max);
        y = theRandom.get_int(random_min, random_max);
        ac.push_back( Circle_2( Point_2(x,y), x*x + y*y));
    }

    Arr _pm;
    Point_location _pl(_pm);
    for (ArcContainer::const_iterator it=ac.begin(); it != ac.end(); ++it) {
        insert_curve(_pm,*it,_pl);
    };

    return 0;
};

```

4.3.2 Constructing an Arrangement of Circles and Segments

In this example, the traits class using the `boost::variant` is used in order to provide arrangements with curves that can be either circular arcs or line segments.

```

#include <CGAL/basic.h>
#include <CGAL/Cartesian.h>

#include <CGAL/Random.h>
#include <CGAL/point_generators_2.h>

#include <CGAL/MP_Float.h>
#include <CGAL/Gmpq.h>

#include <CGAL/Algebraic_kernel_2_2.h>

#include <CGAL/Circular_kernel.h>

#include <CGAL/Arr_circular_line_arc_traits.h>

```

```

#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_naive_point_location.h>

typedef CGAL::Gmpq NT;
typedef CGAL::Cartesian<NT> Linear_k;

typedef CGAL::Algebraic_kernel_for_circles_2_2<NT> Algebraic_k;
typedef CGAL::Circular_kernel_2<Linear_k,Algebraic_k> Circular_k;

typedef Circular_k::Point_2 Point_2;
typedef Circular_k::Circle_2 Circle_2;
typedef Circular_k::Circular_arc_2 Circular_arc_2;
typedef Circular_k::Line_arc_2 Line_arc_2;

typedef boost::variant< Circular_arc_2, Line_arc_2> Arc;
typedef std::vector< Arc> ArcContainer;

typedef CGAL::Arr_circular_line_arc_traits
<Circular_k, Circular_arc_2, Line_arc_2> Traits;

typedef CGAL::Arrangement_2<Traits> Arr;
typedef CGAL::Arr_naive_point_location<Arr> Point_location;

int main(){

    CGAL::Random generatorOfgenerator;
    int random_seed = generatorOfgenerator.get_int(0, 123456);
    std::cout << "random_seed = " << random_seed << std::endl;
    CGAL::Random theRandom(random_seed);
    int random_max = 128;
    int random_min = -128;
    ArcContainer ac;
    int x1, y1, x2, y2;

    for (int i = 0; i < 10; i++) {
        x1 = theRandom.get_int(random_min,random_max);
        y1 = theRandom.get_int(random_min,random_max);
        do{
            x2 = theRandom.get_int(random_min,random_max);
            y2 = theRandom.get_int(random_min,random_max);
        } while((x1 == x2) && (y1 == y2));
        std::cout << x1 << " " << y1 << " " << x2 << " " << y2 << std::endl;

        boost::variant< Circular_arc_2, Line_arc_2 >
            v = Line_arc_2(Point_2(x1,y1), Point_2(x2,y2));
        ac.push_back( v);
    }

    for (int i = 0; i < 10; i++) {
        x1 = theRandom.get_int(random_min,random_max);
        y1 = theRandom.get_int(random_min,random_max);

        boost::variant< Circular_arc_2, Line_arc_2 >

```



```

        v = Circle_2( Point_2(x1,y1), x1*x1 + y1*y1);
        ac.push_back(v);
    }

    Arr _pm;
    Point_location _pl(_pm);
    for (ArcContainer::const_iterator it=ac.begin(); it != ac.end(); ++it) {
        insert_curve(_pm,*it,_pl);
    };

    return 0;
};

```

4.3.3 Using the Predefined Kernel

```

#include <CGAL/basic.h>

#include <CGAL/Random.h>
#include <CGAL/point_generators_2.h>

#include <CGAL/Exact_circular_kernel.h>

#include <CGAL/Arr_circular_line_arc_traits.h>

#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_naive_point_location.h>

typedef CGAL::Exact_circular_kernel_2          Circular_k;

typedef Circular_k::Point_2                    Point_2;
typedef Circular_k::Circle_2                   Circle_2;
typedef Circular_k::Circular_arc_2             Circular_arc_2;
typedef Circular_k::Line_arc_2                 Line_arc_2;

typedef boost::variant< Circular_arc_2, Line_arc_2> Arc;
typedef std::vector< Arc> ArcContainer;

typedef CGAL::Arr_circular_line_arc_traits
    <Circular_k, Circular_arc_2, Line_arc_2> Traits;

typedef CGAL::Arrangement_2<Traits>            Arr;
typedef CGAL::Arr_naive_point_location<Arr>    Point_location;

int main(){

    CGAL::Random generatorOfgenerator;
    int random_seed = generatorOfgenerator.get_int(0, 123456);
    std::cout << "random_seed = " << random_seed << std::endl;
    CGAL::Random theRandom(random_seed);
    int random_max = 128;
    int random_min = -128;

```

```

ArcContainer ac;
int x1, y1, x2, y2;

for (int i = 0; i < 10; i++) {
    x1 = theRandom.get_int(random_min,random_max);
    y1 = theRandom.get_int(random_min,random_max);
    do{
        x2 = theRandom.get_int(random_min,random_max);
        y2 = theRandom.get_int(random_min,random_max);
    } while((x1 == x2) && (y1 == y2));
    std::cout << x1 << " " << y1 << " " << x2 << " " << y2 << std::endl;

    boost::variant< Circular_arc_2, Line_arc_2 >
        v = Line_arc_2(Point_2(x1,y1), Point_2(x2,y2));
    ac.push_back( v);
}

for (int i = 0; i < 10; i++) {
    x1 = theRandom.get_int(random_min,random_max);
    y1 = theRandom.get_int(random_min,random_max);

    boost::variant< Circular_arc_2, Line_arc_2 >
        v = Circle_2( Point_2(x1,y1), x1*x1 + y1*y1);
    ac.push_back(v);
}

Arr_pm;
Point_location _pl(_pm);
for (ArcContainer::const_iterator it=ac.begin(); it != ac.end(); ++it) {
    insert_curve(_pm,*it,_pl);
};

return 0;
};

```

4.4 Design and Implementation History

The first pieces of prototype code were comparisons of algebraic numbers of degree 2, written by Olivier Devillers [DFMT00, DFMT02], and that are still used in the current implementation of *CGAL::Root_of_2*.

Some work was then done in the direction of a “kernel” for CGAL.¹ and the first design emerged in [EKP⁺04].

The code of this package was written by Sylvain Pion and Monique Teillaud who also wrote the manual. Athanasios Kakargias worked on an prototype version of this kernel in 2003. Julien Hazebrouck participated in the implementation of this kernel in July and August 2005.

Some work is in progress on the implementation of a 3D kernel for circles, circular arcs and spherical patches (with the participation of Julien Hazebrouck and Damien Leroy).

¹Monique Teillaud, First Prototype of a CGAL Geometric Kernel with Circular Arcs, Technical Report ECG-TR-182203-01, 2002
Sylvain Pion and Monique Teillaud, Towards a CGAL-like kernel for curves, Technical Report ECG-TR-302206-01, 2003

This work was partially supported by the IST Programme of the EU as a Shared-cost RTD (FET Open) Project under Contract No IST-2000-26473 (ECG - Effective Computational Geometry for Curves and Surfaces). This work is now partially supported by the IST Programme of the 6th Framework Programme of the EU as a STREP (FET Open Scheme) Project under Contract No IST-006413 (ACS - Algorithms for Complex Shapes).

2D Circular Kernel Reference Manual

Sylvain Pion and Monique Teillaud

4.5 Geometric Concepts

CircularKernel	page 545
LinearKernel	page 550

Functors

CircularKernel::ConstructLine_2	page 560
CircularKernel::ConstructCircle_2	page 561
CircularKernel::ConstructCircularArcPoint_2	page 562
CircularKernel::ConstructLineArc_2	page 563
CircularKernel::ConstructCircularArc_2	page 564
CircularKernel::ConstructCircularMinVertex_2	page 565
CircularKernel::ConstructCircularMaxVertex_2	page 566
CircularKernel::ConstructCircularSourceVertex_2	page 567
CircularKernel::ConstructCircularTargetVertex_2	page 568
CircularKernel::ConstructBbox_2	page 569
CircularKernel::CompareX_2	page 570
CircularKernel::CompareY_2	page 571
CircularKernel::CompareXY_2	page 572
CircularKernel::Equal_2	page 578
CircularKernel::CompareYatX_2	page 573
CircularKernel::CompareYtoRight_2	page 574
CircularKernel::HasOn_2	page 579
CircularKernel::DoOverlap_2	page 580
CircularKernel::InXRange_2	page 581

CircularKernel::IsVertical_2	page 582
CircularKernel::IsXMonotone_2	page 583
CircularKernel::IsYMonotone_2	page 584
CircularKernel::MakeXMonotone_2	page 575
CircularKernel::Intersect_2	page 576
CircularKernel::Split_2	page 577
CircularKernel::GetEquation	page 585

4.6 Algebraic Concepts

AlgebraicKernelForCircles	page 586
---------------------------------	----------

Functors

AlgebraicKernelForCircles::ConstructPolynomial_1_2	page 599
AlgebraicKernelForCircles::ConstructPolynomialForCircles_2_2	page 600
AlgebraicKernelForCircles::CompareX	page 595
AlgebraicKernelForCircles::CompareY	page 596
AlgebraicKernelForCircles::CompareXY	page 597
AlgebraicKernelForCircles::SignAt	page 598
AlgebraicKernelForCircles::XCriticalPoints	page 602
AlgebraicKernelForCircles::YCriticalPoints	page 603
AlgebraicKernelForCircles::Solve	page 601

4.7 Geometric Kernels and Classes

Kernels

CGAL::Circular_kernel_2<LinearKernel,AlgebraicKernelForCircles>	page 548
CGAL::Exact_circular_kernel_2	page 549

Points

CGAL::Circular_arc_point_2<CircularKernel>	page 558
--------------------------------------------------	----------

Arcs

<i>CGAL::Circular_arc_2<CircularKernel></i>	page 554
<i>CGAL::Line_arc_2<CircularKernel></i>	page 556

4.8 Algebraic Kernel and Classes

Kernel

<i>CGAL::Algebraic_kernel_for_circles_2_2<RT></i>	page 588
---------------------------------------------------------------	----------

Polynomials

<i>CGAL::Polynomial_1_2<RT></i>	page 592
<i>CGAL::Polynomial_for_circles_2_2<FT></i>	page 594

Roots of Polynomials

<i>CGAL::Root_of_2<RT></i>	page 2582
<i>CGAL::Root_for_circles_2_2<FT></i>	page 590
<i>CGAL::Root_of_traits_2<RT></i>	page 2581

4.9 Traits Classes for CGAL Arrangements

<i>CGAL::Arr_circular_arc_traits<CircularKernel></i>	page 604
<i>CGAL::Arr_line_arc_traits<CircularKernel></i>	page 605
<i>CGAL::Arr_circular_line_arc_traits<CircularKernel></i>	page 606

4.10 Alphabetical List of Reference Pages

<i>AlgebraicKernelForCircles::CompareXY</i>	page 597
<i>AlgebraicKernelForCircles::CompareX</i>	page 595
<i>AlgebraicKernelForCircles::CompareY</i>	page 596
<i>AlgebraicKernelForCircles::ConstructPolynomialForCircles_2_2</i>	page 600
<i>AlgebraicKernelForCircles::ConstructPolynomial_1_2</i>	page 599
<i>AlgebraicKernelForCircles::PolynomialForCircles_2_2</i>	page 593
<i>AlgebraicKernelForCircles::Polynomial_1_2</i>	page 591
<i>AlgebraicKernelForCircles::RootForCircles_2_2</i>	page 589
<i>AlgebraicKernelForCircles::SignAt</i>	page 598
<i>AlgebraicKernelForCircles::Solve</i>	page 601
<i>AlgebraicKernelForCircles::XCriticalPoints</i>	page 602
<i>AlgebraicKernelForCircles::YCriticalPoints</i>	page 603

<i>AlgebraicKernelForCircles</i>	page 586
<i>Algebraic_kernel_for_circles_2_2<RT></i>	page 588
<i>Arr_circular_arc_traits<CircularKernel></i>	page 604
<i>Arr_circular_line_arc_traits<CircularKernel></i>	page 606
<i>Arr_line_arc_traits<CircularKernel></i>	page 605
<i>CircularKernel::CircularArcPoint_2</i>	page 553
<i>CircularKernel::CircularArc_2</i>	page 551
<i>CircularKernel::CompareXY_2</i>	page 572
<i>CircularKernel::CompareX_2</i>	page 570
<i>CircularKernel::CompareYatX_2</i>	page 573
<i>CircularKernel::CompareYtoRight_2</i>	page 574
<i>CircularKernel::CompareY_2</i>	page 571
<i>CircularKernel::ConstructBbox_2</i>	page 569
<i>CircularKernel::ConstructCircle_2</i>	page 561
<i>CircularKernel::ConstructCircularArcPoint_2</i>	page 562
<i>CircularKernel::ConstructCircularArc_2</i>	page 564
<i>CircularKernel::ConstructCircularMaxVertex_2</i>	page 566
<i>CircularKernel::ConstructCircularMinVertex_2</i>	page 565
<i>CircularKernel::ConstructCircularSourceVertex_2</i>	page 567
<i>CircularKernel::ConstructCircularTargetVertex_2</i>	page 568
<i>CircularKernel::ConstructLineArc_2</i>	page 563
<i>CircularKernel::ConstructLine_2</i>	page 560
<i>CircularKernel::DoOverlap_2</i>	page 580
<i>CircularKernel::Equal_2</i>	page 578
<i>CircularKernel::GetEquation</i>	page 585
<i>CircularKernel::HasOn_2</i>	page 579
<i>CircularKernel::Intersect_2</i>	page 576
<i>CircularKernel::InXRange_2</i>	page 581
<i>CircularKernel::IsVertical_2</i>	page 582
<i>CircularKernel::IsXMonotone_2</i>	page 583
<i>CircularKernel::IsYMonotone_2</i>	page 584
<i>CircularKernel::LineArc_2</i>	page 552
<i>CircularKernel::MakeXMonotone_2</i>	page 575
<i>CircularKernel::Split_2</i>	page 577
<i>CircularKernel</i>	page 545
<i>Circular_arc_2<CircularKernel></i>	page 554
<i>Circular_arc_point_2<CircularKernel></i>	page 558
<i>Circular_kernel_2<LinearKernel,AlgebraicKernelForCircles></i>	page 548
<i>Exact_circular_kernel_2</i>	page 549
<i>LinearKernel</i>	page 550
<i>Line_arc_2<CircularKernel></i>	page 556
<i>Polynomial_1_2<RT></i>	page 592
<i>Polynomial_for_circles_2_2<FT></i>	page 594
<i>Root_for_circles_2_2<FT></i>	page 590

CircularKernel

Refines

Kernel

Has Models

CGAL::Circular_kernel_2 <*LinearKernel*, *AlgebraicKernelForCircles*>

CGAL::Exact_circular_kernel_2

Types

A model of *CircularKernel* is supposed to provide some basic types

CircularKernel::Linear_kernel

Model of *LinearKernel*.

CircularKernel::Algebraic_kernel

Model of *AlgebraicKernelForCircles*.

CircularKernel::RT

Model of *RingNumberType*.

CircularKernel::FT

Model of *FieldNumberType*.

CircularKernel::Root_of_2

Model of *RootOf_2*.

CircularKernel::Root_for_circles_2_2

Model of *AlgebraicKernelForCircles::RootForCircles_2_2*.

CircularKernel::Polynomial_1_2

Model of *AlgebraicKernelForCircles::Polynomial_1_2*.

CircularKernel::Polynomial_for_circles_2_2

Model of *AlgebraicKernelForCircles::PolynomialForCircles_2_2*.

and to define the following geometric objects

CircularKernel::Point_2

Model of *Kernel::Point_2*.

CircularKernel::Circle_2

Model of *Kernel::Circle_2*.

CircularKernel::Line_arc_2

Model of *CircularKernel::LineArc_2*.

CircularKernel::Circular_arc_2

Model of *CircularKernel::CircularArc_2*.

CircularKernel::Circular_arc_point_2

Model of *CircularKernel::CircularArcPoint_2*.

Moreover, a model of *CircularKernel* must provide predicates, constructions and other functionalities.

Predicates

CircularKernel::Compare_x_2

Model of *CircularKernel::CompareX_2*.

CircularKernel::Compare_y_2

Model of *CircularKernel::CompareY_2*.

CircularKernel::Compare_xy_2

Model of *CircularKernel::CompareXY_2*.

CircularKernel::Equal_2

Model of *CircularKernel::Equal_2*.

CircularKernel::Compare_y_at_x_2

Model of *CircularKernel::CompareYatX_2*.

<i>CircularKernel:: Compare_y_to_right_2</i>	Model of <i>CircularKernel::CompareYtoRight_2</i> .
<i>CircularKernel:: Has_on_2</i>	Model of <i>CircularKernel::HasOn_2</i> .
<i>CircularKernel:: Do_overlap</i>	Model of <i>CircularKernel::DoOverlap_2</i> .
<i>CircularKernel:: In_x_range_2</i>	Model of <i>CircularKernel::InXRange_2</i> .
<i>CircularKernel:: Is_vertical_2</i>	Model of <i>CircularKernel::IsVertical_2</i> .
<i>CircularKernel:: Is_x_monotone_2</i>	Model of <i>CircularKernel::IsXMonotone_2</i> .
<i>CircularKernel:: Is_y_monotone_2</i>	Model of <i>CircularKernel::IsYMonotone_2</i> .

Constructions

<i>CircularKernel:: Construct_line_2</i>	Model of <i>CircularKernel::ConstructLine_2</i> .
<i>CircularKernel:: Construct_circle_2</i>	Model of <i>CircularKernel::ConstructCircle_2</i> .
<i>CircularKernel:: Construct_circular_arc_point_2</i>	Model of <i>CircularKernel::ConstructCircularArcPoint_2</i> .
<i>CircularKernel:: Construct_line_arc_2</i>	Model of <i>CircularKernel::ConstructLineArc_2</i> .
<i>CircularKernel:: Construct_circular_arc_2</i>	Model of <i>CircularKernel::ConstructCircularArc_2</i> .
<i>CircularKernel:: Construct_circular_min_vertex_2</i>	Model of <i>CircularKernel::ConstructCircularMinVertex_2</i> .
<i>CircularKernel:: Construct_circular_max_vertex_2</i>	Model of <i>CircularKernel::ConstructCircularMaxVertex_2</i> .
<i>CircularKernel:: Construct_circular_source_vertex_2</i>	Model of <i>CircularKernel::ConstructCircularSourceVertex_2</i> .
<i>CircularKernel:: Construct_circular_target_vertex_2</i>	Model of <i>CircularKernel::ConstructCircularTargetVertex_2</i> .

Link with the algebraic kernel

<i>CircularKernel:: Get_equation</i>	Model of <i>CircularKernel::GetEquation</i> .
--------------------------------------	-----------------------------------------------

Operations

As in the *Kernel* concept, for each of the function objects above, there must exist a member function that requires no arguments and returns an instance of that function object. The name of the member function is the uncapitalized name of the type returned with the suffix *_object* appended. For example, for the function object *CircularKernel::Construct_circular_arc_2* the following member function must exist:

Construct_circular_arc_2

ck.construct_circular_arc_2_object()

See Also

Kernel page [35](#)

CGAL::Circular_kernel_2<LinearKernel,AlgebraicKernelForCircles>

`#include <CGAL/Circular_kernel.h>`

Is Model for the Concepts

CircularKernel

Parameters

The circular kernel is parameterized by a *LinearKernel* parameter (and derives from it), in order to reuse all needed functionalities on basic linear objects provided by one of the CGAL kernels. It also allows other implementations of these basic functionalities.

The second parameter, *AlgebraicKernelForCircles*, is meant to provide the circular kernel with all the algebraic functionalities required for the manipulation of algebraic curves.

Inherits From

LinearKernel

Types

The circular kernel uses basic number types of the algebraic kernel:
`typedef AlgebraicKernelForCircles::RT` *RT*; Ring number type.

`typedef AlgebraicKernelForCircles::FT` *FT*; Field number type.

In fact, the two number types *AlgebraicKernelForCircles::RT* and *LinearKernel::RT* must coincide, as well as *AlgebraicKernelForCircles::FT* and *LinearKernel::FT*.

The following types are available, as well as all the functionality on them described in the *CircularKernel* concept.

```
typedef Line_arc_2<Circular_kernel_2>      Line_arc_2;
typedef Circular_arc_2<Circular_kernel_2>   Circular_arc_2;
typedef Circular_arc_point_2<Circular_kernel_2>
                                           Circular_arc_point_2;
```

See Also

LinearKernel [page 550](#)
AlgebraicKernelForCircles [page 586](#)
CGAL::Exact_circular_kernel_2 [page 549](#)

CGAL::Exact_circular_kernel_2

#include <CGAL/Exact_circular_kernel.h>

Definition

A typedef to a (filtered) circular kernel that provides both exact geometric predicates and exact geometric constructions.

Is Model for the Concepts

CircularKernel

See Also

CGAL::Circular_kernel_2<LinearKernel,AlgebraicKernelForCircles> page [548](#)

LinearKernel

Definition

The geometric kernel parameter of *CGAL::Circular_kernel_2* is supposed to be a model of the (*two-dimensional*) *Kernel* concept, so that the circular kernel provides all functionalities of a CGAL kernel.

Has Models

All CGAL kernels

See Also

Kernel page [35](#)
CircularKernel page [545](#)

CircularKernel::CircularArc_2

Concept for arcs of circles.

Refines

CopyConstructible, Assignable, DefaultConstructible

Has Models

CGAL::Circular_arc_2<CircularKernel>

CircularKernel::LineArc_2

Definition

Concept for line segments supported by a line that is a model of *Kernel::Line_2* and whose endpoints are models of the *CircularKernel::CircularArcPoint_2* concept.

Refines

CopyConstructible, Assignable, DefaultConstructible

Has Models

CGAL::Line_arc_2<*CircularKernel*>

CircularKernel::CircularArcPoint_2

Definition

Concept for points on circles, circular arcs or line arcs.

Refines

CopyConstructible, Assignable, DefaultConstructible

Has Models

CGAL::Circular_arc_point_2<CircularKernel>

CGAL::Circular_arc_2<CircularKernel>

```
#include <CGAL/Circular_arc_2.h>
```

Is Model for the Concepts

```
CircularKernel::CircularArc_2
```

Creation

```
Circular_arc_2<CircularKernel> ca( CircularKernel::Circle_2 c);
```

Constructs an arc from a full circle.

```
Circular_arc_2<CircularKernel> ca( CircularKernel::Circle_2 c,
    CircularKernel::Circular_arc_point_2 p1,
    CircularKernel::Circular_arc_point_2 p2)
```

Constructs the circular arc supported by *c*, that is oriented counterclockwise, whose source is *p1* and whose target is *p2*.

Precondition: *p1* and *p2* lie on *c*.

Access Functions

```
CircularKernel::Circle_2 ca.supporting_circle()
```

A circular arc is supposed to be oriented counterclockwise, from *source* to *target*.

```
CircularKernel::Circular_arc_point_2 ca.source()
CircularKernel::Circular_arc_point_2 ca.target()
```

When the methods *source* and *target* return the same point, then the arc is in fact a full circle.

When an arc is x-monotone, its left and right points can be accessed directly:

```
CircularKernel::Circular_arc_point_2 ca.left()           Precondition: ca.is_x_monotone().
CircularKernel::Circular_arc_point_2 ca.right()          Precondition: ca.is_x_monotone().
```

Query Functions

```
bool ca.is_x_monotone() Tests whether the arc is x-monotone.
bool ca.is_y_monotone() Tests whether the arc is y-monotone.
```

I/O

istream& *std::istream*& *is* >> *Circular_arc_2* & *ca*
ostream& *std::ostream*& *os* << *Circular_arc_2* *ca*

See Also

CGAL::Circular_arc_point_2<*CircularKernel*> page [558](#)
CGAL::Line_arc_2<*CircularKernel*> page [556](#)

CGAL::Line_arc_2<CircularKernel>

```
#include <CGAL/Line_arc_2.h>
```

Is Model for the Concepts

```
CircularKernel::LineArc2
```

Creation

```
Line_arc_2<CircularKernel> la( CircularKernel::Line_2 l,
                               CircularKernel::Circular_arc_point_2 p1,
                               CircularKernel::Circular_arc_point_2 p2)
```

Construct the line segment supported by l , whose source is $p1$ and whose target is $p2$.
Precondition: $p1$ and $p2$ lie on l .

```
Line_arc_2<CircularKernel> la( CircularKernel::Line_2 l,
                               CircularKernel::Point_2 p1,
                               CircularKernel::Point_2 p2)
```

Same.

```
Line_arc_2<CircularKernel> la( CircularKernel::Segment_2 s);
```

Access Functions

```
CircularKernel::Line_2          la.supporting_line()
```

```
CircularKernel::Circular_arc_point_2 la.source()
```

```
CircularKernel::Circular_arc_point_2 la.target()
```

```
CircularKernel::Circular_arc_point_2 la.left()
```

```
CircularKernel::Circular_arc_point_2 la.right()
```

Query Functions

```
bool          la.is_vertical()
```

I/O

```
istream&          std::istream& is >> Line_arc_2 & ca
ostream&          std::ostream& os << Line_arc_2 ca
```

See Also

CGAL::Circular_arc_point_2<CircularKernel> page [558](#)
CGAL::Circular_arc_2<CircularKernel> page [554](#)

CGAL::Circular_arc_point_2<CircularKernel>

```
#include <CGAL/Circular_arc_point_2.h>
```

Is Model for the Concepts

```
CircularKernel::CircularArcPoint_2
```

Creation

```
Circular_arc_point_2<CircularKernel> p( CircularKernel::Point_2 q);
```

```
Circular_arc_point_2<CircularKernel> p( CircularKernel::Root_for_circles_2_2 r);
```

Access Functions

<i>CircularKernel::Root_of_2</i>	<i>p.x()</i>	<i>x</i> -coordinate of the point.
<i>CircularKernel::Root_of_2</i>	<i>p.y()</i>	<i>y</i> -coordinate of the point.

<i>Bbox_2</i>	<i>p.bbox()</i>	Returns a bounding box around the point.
---------------	-----------------	------------------------------------------

Operations

<i>bool</i>	<i>p == q</i>	Test for equality. Two points are equal, iff their <i>x</i> and <i>y</i> coordinates are equal.
-------------	---------------	-------------------------------------------------------------------------------------------------

<i>bool</i>	<i>p != q</i>	Test for nonequality.
-------------	---------------	-----------------------

<i>bool</i>	<i>p < q</i>	Returns true iff <i>p</i> is lexicographically smaller than <i>q</i> , i.e. either if <i>p.x()</i> < <i>q.x()</i> or if <i>p.x()</i> == <i>q.x()</i> and <i>p.y()</i> < <i>q.y()</i> .
-------------	-----------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<i>bool</i>	<i>p > q</i>	Returns true iff <i>p</i> is lexicographically greater than <i>q</i> .
-------------	-----------------	------------------------------------------------------------------------

<i>bool</i>	<i>p <= q</i>	Returns true iff <i>p</i> is lexicographically smaller than or equal to <i>q</i> .
-------------	------------------	------------------------------------------------------------------------------------

<i>bool</i>	<i>p >= q</i>	Returns true iff <i>p</i> is lexicographically greater than or equal to <i>q</i> .
-------------	------------------	------------------------------------------------------------------------------------

I/O

<i>istream&</i>	<i>std::istream& is >> Circular_arc_point_2 & cp</i>
<i>ostream&</i>	<i>std::ostream& os << Circular_arc_point_2 ce</i>

See Also

CGAL::Circular_arc_2<CircularKernel> [page 554](#)
CGAL::Line_arc_2<CircularKernel> [page 556](#)

CircularKernel::ConstructLine_2

Refines

Kernel::ConstructLine_2

A model *fo* of this type must provide:

CircularKernel::Line_2 *fo*(*CircularKernel::Polynomial_1_2*)

Constructs a line from an equation.

See Also

CircularKernel::GetEquation page [585](#)

CircularKernel::ConstructCircle_2

Refines

Kernel::ConstructCircle_2

A model *fo* of this type must provide:

CircularKernel::Circle_2

fo(CircularKernel::Polynomial_for_circles_2_2)

Constructs a circle from an equation.

See Also

CircularKernel::GetEquation page [585](#)

CircularKernel::ConstructCircularArcPoint_2

A model *fo* of this type must provide:

CircularKernel::Circular_arc_point_2

fo(CircularKernel::Root_for_circles_2_2 r)

CircularKernel::Circular_arc_point_2

fo(CircularKernel::Point_2 p)

CircularKernel::ConstructLineArc_2

A model *fo* of this type must provide:

CircularKernel::Line_arc_2

```
fo.operator()( CircularKernel::Line_2 l,
                CircularKernel::Circular_arc_point_2 p1,
                CircularKernel::Circular_arc_point_2 p2)
```

Constructs the line segment supported by *l*, whose source is *p1* and whose target is *p2*.

Precondition: *p1* and *p2* lie on *l*.

CircularKernel::Line_arc_2

```
fo( CircularKernel::Segment_2 s)
```

CircularKernel::Line_arc_2

```
fo( CircularKernel::Point_2 p1, CircularKernel::Point_2 p2)
```

CircularKernel::Line_arc_2

```
fo.operator()( CircularKernel::Line_2 l,
                CircularKernel::Circle_2 c1,
                bool b1,
                CircularKernel::Circle_2 c2,
                bool b2)
```

Constructs the line segment whose supporting line is *l*, whose source endpoint is the b_1^h intersection of *l* with *c1*, and whose target endpoint is the b_2^h intersection of *l* and *c2*, where intersections are ordered lexicographically.

Precondition: *l* intersects both *c1* and *c2*, and the arc defined by the intersections has non-zero length.

CircularKernel::Line_arc_2

```
fo.operator()( CircularKernel::Line_2 l,
                CircularKernel::Line_2 l1,
                CircularKernel::Line_2 l2)
```

Same, for intersections defined by lines instead of circles.

CircularKernel::ConstructCircularArc_2

A model *fo* of this type must provide:

CircularKernel::Circular_arc_2

fo(*CircularKernel::Circle_2* *c*)

Constructs an arc from a full circle.

CircularKernel::Circular_arc_2

fo.operator()(*CircularKernel::Circle_2* *c*,
CircularKernel::Circular_arc_point_2 *p1*,
CircularKernel::Circular_arc_point_2 *p2*)

Construct the circular arc supported by *c*, that is oriented counterclockwise, whose source is *p1* and whose target is *p2*.

Precondition: *p1* and *p2* lie on *c*.

CircularKernel::Circular_arc_2

fo.operator()(*CircularKernel::Circle_2* *c*,
CircularKernel::Circle_2 *c1*,
bool *b1*,
CircularKernel::Circle_2 *c2*,
bool *b2*)

Constructs the unique circular arc that is oriented counterclockwise, whose supporting circle is *c*, and whose source endpoint is the intersection of *c* and *c1* with index *b1*, and whose target is the intersection of *c* and *c2* of index *b2*, where intersections are ordered lexicographically.

Precondition: *c* intersects both *c1* and *c2*, and the arc defined by the intersections has non-zero length.

CircularKernel::Circular_arc_2

fo.operator()(*CircularKernel::Circle_2* *c*,
CircularKernel::Line_2 *l1*,
bool *b1*,
CircularKernel::Line_2 *l2*,
bool *b2*)

Same, for intersections defined by lines instead of circles.

CircularKernel::ConstructCircularMinVertex_2

A model *fo* of this type must provide:

CircularKernel::Circular_arc_point_2

fo(CircularKernel::Circular_arc_2 c)

Constructs the *x*-minimal vertex of *c*.
Precondition: The arc *c* is *x*-monotone.

CircularKernel::Circular_arc_point_2

fo(CircularKernel::Line_arc_2 l)

Same, for a line segment.

CircularKernel::ConstructCircularMaxVertex_2

A model *fo* of this type must provide:

CircularKernel::Circular_arc_point_2

fo(CircularKernel::Circular_arc_2 c)

Constructs the *x*-maximal vertex of *c*.

Precondition: The arc *c* is *x*-monotone.

CircularKernel::Circular_arc_point_2

fo(CircularKernel::Line_arc_2 l)

Same, for a line segment.

CircularKernel::ConstructCircularSourceVertex_2

A model *fo* of this type must provide:

CircularKernel::Circular_arc_point_2

fo(CircularKernel::Circular_arc_2 c)

Constructs the source vertex of *c*.

CircularKernel::Circular_arc_point_2

fo(CircularKernel::Line_arc_2 l)

Same, for a line segment.

CircularKernel::ConstructCircularTargetVertex_2

A model *fo* of this type must provide:

CircularKernel::Circular_arc_point_2

fo(CircularKernel::Circular_arc_2 c)

Constructs the target vertex of *c*.

CircularKernel::Circular_arc_point_2

fo(CircularKernel::Line_arc_2 l)

Same, for a line segment.

CircularKernel::ConstructBbox_2

A model *fo* of this type must provide operators to construct a bounding box of geometric objects:

CGAL::Bbox_2 *fo*(*CircularKernel::Circular_arc_point_2* *p*)

CGAL::Bbox_2 *fo*(*CircularKernel::Line_arc_2* *l*)

CGAL::Bbox_2 *fo*(*CircularKernel::Circular_arc_2* *c*)

CircularKernel::CompareX_2

Refines

Kernel::CompareX_2

An object *fo* of this type must provide in addition:

```
Comparison_result      fo.operator()( CircularKernel::Circular_arc_point_2 p,
                    CircularKernel::Circular_arc_point_2 q)
```

Compares the x -coordinates of p and q .

See Also

<i>CircularKernel::CompareY_2</i>	page 571
<i>CircularKernel::CompareXY_2</i>	page 572
<i>CircularKernel::Equal_2</i>	page 578

Refines

An object *fo* of this type must provide in addition:

Compares the y -coordinates of p and q .

<i>CircularKernel::CompareX_2</i>	page 570
<i>CircularKernel::CompareXY_2</i>	page 572
<i>CircularKernel::Equal_2</i>	page 578

CircularKernel::CompareXY_2

Refines

Kernel::CompareXY_2

An object *fo* of this type must provide in addition:

[illegible]

Compares p and q according to the lexicographic ordering on x - and y -coordinates.

See Also

<i>CircularKernel::CompareX_2</i>	page 570
<i>CircularKernel::CompareY_2</i>	page 571
<i>CircularKernel::Equal_2</i>	page 578

CircularKernel::CompareYatX_2

An object *fo* of this type must provide two operators that compare a point *p* and an arc *a* on the vertical line passing through *p*.

Comparison_result *fo*(*CircularKernel::Circular_arc_point_2* *p*, *CircularKernel::Circular_arc_2* *a*)

For a circular arc.

Precondition: The arc *a* must be monotone and *p* must be in the vertical range of *a*.

Comparison_result *fo*(*CircularKernel::Circular_arc_point_2* *p*, *CircularKernel::Line_arc_2* *a*)

Same for a segment.

CircularKernel::MakeXMonotone_2

A model *fo* of this type must provide:

```
template < class OutputIterator >
OutputIterator fo( CircularKernel::Circular_arc_2 ca, OutputIterator oit)
```

Splits the arc *ca* into monotone arcs that are returned through the output iterator.

For the sake of completeness, the *operator()* must also be defined for a *Line_arc_2*. In this case, the input line arc itself is the only arc returned through the *OutputIterator*.

CircularKernel::Intersect_2

Refines

Kernel::Intersect_2

A model *fo* of this type must provide:

```
template < class OutputIterator >
OutputIterator fo( Type1 obj1, Type2 obj2, OutputIterator intersections)
```

Copies in the output iterator the intersection elements between the two objects. *intersections* iterates on elements of type *CGAL::Object*.

where *Type_1* and *Type_2* can both be either

- *CircularKernel::Line_arc_2* or
- *CircularKernel::Circle_2* or
- *CircularKernel::Circular_arc_2*.

Depending on the types *Type_1* and *Type_2*, these elements can be assigned to

- *std::pair<CircularKernel::Circular_arc_point_2, unsigned>*, where the unsigned integer is the multiplicity of the corresponding intersection point between *obj_1* and *obj_2* or
- *CircularKernel::Circular_arc_2* in case of an overlap.

CircularKernel::Split_2

A model *fo* of this type must provide:

```
void fo.operator()( CircularKernel::Circular_arc_2 a,
                    CircularKernel::Circular_arc_point_2 p,
                    CircularKernel::Circular_arc_2 &a1,
                    CircularKernel::Circular_arc_2 &a2)
```

Splits arc *a* at point *p*, which creates arcs *a1* and *a2*.
Precondition: *a* is *x*-monotone, and *p* lies on *a*.

```
void fo.operator()( CircularKernel::Line_arc_2 l,
                    CircularKernel::Circular_arc_point_2 p,
                    CircularKernel::Line_arc_2 &l1,
                    CircularKernel::Line_arc_2 &l2)
```

Same for a line arc.

CircularKernel::Equal_2

Definition

Testing equality between objects.

Refines

Kernel::Equal_2

An object *fo* of this type must provide in addition:

bool *fo.operator()*(*CircularKernel::Circular_arc_point_2* p0,
CircularKernel::Circular_arc_point_2 p1)

For two points.

bool *fo*(*CircularKernel::Circular_arc_2* a0, *CircularKernel::Circular_arc_2* a1)

For two arcs.

bool *fo*(*CircularKernel::Line_arc_2* a0, *CircularKernel::Line_arc_2* a1)

For two segments.

For the sake of completeness, the *operator()* must also be defined for a *Line_arc_2* and a *Circular_arc_2* as arguments (in any order), and it always returns *false*.

See Also

CircularKernel::CompareX_2 page [570](#)
CircularKernel::CompareY_2 page [571](#)
CircularKernel::CompareXY_2 page [572](#)

CircularKernel::HasOn_2

Definition

To test whether a point lies on a curve.

An object *fo* of this type must provide:

bool *fo*(*CircularKernel::Line_2* l, *CircularKernel::Circular_arc_point_2* p)

For a line.

bool *fo*(*CircularKernel::Circle_2* c, *CircularKernel::Circular_arc_point_2* p)

For a circle.

bool *fo*(*CircularKernel::Line_arc_2* l, *CircularKernel::Circular_arc_point_2* p)

For a line arc.

bool *fo*(*CircularKernel::Circular_arc_2* c, *CircularKernel::Circular_arc_point_2* p)

For a circular arc.

Precondition: c is x-monotone.

CircularKernel::DoOverlap_2

Definition

Testing whether the interiors of two curves overlap.

An object *fo* of this type must provide:

bool *fo*(*CircularKernel::Line_arc_2* l0, *CircularKernel::Line_arc_2* l1)

For two line arcs.

bool *fo*(*CircularKernel::Circular_arc_2* a0, *CircularKernel::Circular_arc_2* a1)

For two circular arcs.

Precondition: a_0 and a_1 are *x*-monotone.

CircularKernel::InXRange_2

Definition

To test whether a point lies in the vertical range of a curve.

An object *fo* of this type must provide:

bool *fo*(CircularKernel::Line_arc_2 l, CircularKernel::Circular_arc_point_2 p)

For a line arc.

bool *fo*(CircularKernel::Circular_arc_2 c, CircularKernel::Circular_arc_point_2 p)

For a circular arc.

Precondition: *c* is *x*-monotone.

CircularKernel::IsVertical_2

Refines

Kernel::IsVertical_2

An object *fo* of this type must provide:

bool *fo(CircularKernel::Line_arc_2 l)*
For a line arc.

bool *fo(CircularKernel::Circular_arc_2 c)*
For a circular arc, always returns *false*.

CircularKernel::IsXMonotone_2

An object *fo* of this type must provide:

<i>bool</i>	<i>fo</i> (CircularKernel::Circular_arc_2 <i>c</i>)	Tests whether the arc is <i>x</i> -monotone.
<i>bool</i>	<i>fo</i> (CircularKernel::Line_arc_2 <i>l</i>)	For a line arc, always returns <i>true</i> .

CircularKernel::IsYMonotone_2

An object *fo* of this type must provide:

<i>bool</i>	<i>fo</i> (<i>CircularKernel::Circular_arc_2</i> <i>c</i>)	Tests whether the arc is y-monotone.
<i>bool</i>	<i>fo</i> (<i>CircularKernel::Line_arc_2</i> <i>l</i>)	For a line arc, always returns <i>true</i> .

CircularKernel::GetEquation

A model *fo* of this type must provide:

CircularKernel::Polynomial_1_2

fo(CircularKernel::Line_2 c)

Returns the equation of the line.

CircularKernel::Polynomial_for_circles_2_2

fo(CircularKernel::Circle_2 c)

Returns the equation of the circle.

See Also

CircularKernel::ConstructLine_2page [560](#)

CircularKernel::ConstructCircle_2 page [561](#)

AlgebraicKernelForCircles

Definition

The *AlgebraicKernelForCircles* concept is meant to provide the curved kernel with all the algebraic functionalities required for the manipulation of circular arcs.

Has Models

Algebraic_kernel_for_circles_2_2

Types

A model of *AlgebraicKernelForCircles* is supposed to provide

AlgebraicKernelForCircles:: RT
AlgebraicKernelForCircles:: FT

A model of *RingNumberType*.
 A model of *FieldNumberType*<*RT*>.

AlgebraicKernelForCircles:: Polynomial_1_2

A model of *AlgebraicKernelForCircles::Polynomial_1_2*, for bivariate polynomials of degree up to 1.

AlgebraicKernelForCircles:: Polynomial_for_circles_2_2

A model of *AlgebraicKernelForCircles::PolynomialForCircles_2_2*, for bivariate polynomials of degree up to 2 that can store equations of circles.

AlgebraicKernelForCircles:: Root_of_2

A model of *RootOf_2*, for algebraic numbers of degree up to 2.

AlgebraicKernelForCircles:: Root_for_circles_2_2

A model of *AlgebraicKernelForCircles::RootForCircles_2_2*, for solutions of systems of two models of *AlgebraicKernelForCircles::PolynomialForCircles_2_2*.

AlgebraicKernelForCircles:: Construct_polynomial_1_2

A model of *AlgebraicKernelForCircles::ConstructPolynomial_1_2*.

AlgebraicKernelForCircles:: Construct_polynomial_for_circles_2_2

A model of *AlgebraicKernelForCircles::ConstructPolynomialForCircles_2_2*.

AlgebraicKernelForCircles:: Compare_x

A model of the concept *AlgebraicKernelForCircles::CompareX*.

<i>AlgebraicKernelForCircles:: Compare_y</i>	A model of the concept <i>AlgebraicKernelForCircles::CompareY</i> .
<i>AlgebraicKernelForCircles:: Compare_xy</i>	A model of the concept <i>AlgebraicKernelForCircles::CompareXY</i> .
<i>AlgebraicKernelForCircles:: Sign_at</i>	A model of the concept <i>AlgebraicKernelForCircles::SignAt</i> .
<i>AlgebraicKernelForCircles:: X_critical_points</i>	A model of the concept <i>AlgebraicKernelForCircles::XCriticalPoints</i> .
<i>AlgebraicKernelForCircles:: Y_critical_points</i>	A model of the concept <i>AlgebraicKernelForCircles::YCriticalPoints</i> .
<i>AlgebraicKernelForCircles:: Solve</i>	A model of the concept <i>AlgebraicKernelForCircles::Solve</i> .

See Also

<i>CircularKernel</i>	page 545
<i>CGAL::Circular_kernel_2<LinearKernel,AlgebraicKernelForCircles></i>	page 548

CGAL::Algebraic_kernel_for_circles_2_2<RT>

#include <CGAL/Algebraic_kernel_2_2.h>

Is Model for the Concepts

AlgebraicKernelForCircles

AlgebraicKernelForCircles::RootForCircles_2_2

Definition

Concept to represent the roots of a system of two equations of degree 2 in two variables x and y that are models of concept *AlgebraicKernelForCircles::PolynomialForCircles_2_2*

Operations

The comparison operator `==` must be provided.

bool $p == q$

Has Models

CGAL::Root_for_circles_2_2

See Also

AlgebraicKernelForCircles page [586](#)

CGAL::Root_for_circles_2_2<FT>

#include <CGAL/Root_for_circles_2_2.h>

Is Model for the Concepts

AlgebraicKernelForCircles::RootForCircles_2_2

AlgebraicKernelForCircles::Polynomial_1_2

Definition

Concept to represent bivariate polynomials of degree 1 whose coefficients are of a type that is a model of the concept *RingNumberType*.

Refines

CopyConstructible, Assignable, DefaultConstructible

Has Models

CGAL::Polynomial_1_2

See Also

AlgebraicKernelForCircles page [586](#)

CGAL::Polynomial_1_2<RT>

#include <CGAL/Polynomials_1_2.h>

Is Model for the Concepts

AlgebraicKernelForCircles::Polynomial_1_2

AlgebraicKernelForCircles::PolynomialForCircles_2_2

Definition

Concept to represent bivariate polynomials of degree up to 2 capable of storing equations of circles, whose center's coordinates, as well as the square of the radius, are of a type that is a model of the concept *FieldNumberType*.

Refines

CopyConstructible, Assignable, DefaultConstructible

Creation

Operations

The comparison operator `==` must be provided.

bool *AlgebraicKernelForCircles::PolynomialForCircles_2_2 p == q*

Has Models

CGAL::Polynomial_for_circles_2_2

See Also

AlgebraicKernelForCircles page [586](#)

CGAL::Polynomial_for_circles_2_2<FT>

#include <CGAL/Polynomials_2_2.h>

Is Model for the Concepts

AlgebraicKernelForCircles::PolynomialForCircles_2_2

See Also

CGAL::Root_of_2<RT> page [2582](#)
AlgebraicKernelForCircles page [586](#)

AlgebraicKernelForCircles::CompareX

A model *fo* of this type must provide:

```
template < class OutputIterator >
CGAL::Comparison_result
```

```
fo.operator()( AlgebraicKernelForCircles::Root_for_circles_2_2 r1,
               AlgebraicKernelForCircles::Root_for_circles_2_2 r2)
```

Compares the *x* (first) variables of two *Root_for_circles_2_2*.

See Also

AlgebraicKernelForCircles::CompareY [page 596](#)
AlgebraicKernelForCircles::CompareXY [page 597](#)
CircularKernel::CompareX_2 [page 570](#)

AlgebraicKernelForCircles::CompareY

Definition

A model *fo* of this type must provide:

```
template < class OutputIterator >
CGAL::Comparison_result
```

```
fo.operator()( AlgebraicKernelForCircles::Root_for_circles_2_2 r1,
               AlgebraicKernelForCircles::Root_for_circles_2_2 r2)
```

Compares the y (second) variables of two *Root_for_circles_2_2*.

See Also

AlgebraicKernelForCircles::CompareX page [595](#)
AlgebraicKernelForCircles::CompareXY page [597](#)
CircularKernel::CompareY_2 page [571](#)

AlgebraicKernelForCircles::CompareXY

Definition

A model *fo* of this type must provide:

```
template < class OutputIterator >
CGAL::Comparison_result
```

```
fo.operator()( AlgebraicKernelForCircles::Root_for_circles_2_2 r1,
               AlgebraicKernelForCircles::Root_for_circles_2_2 r2)
```

Compares two *Root_for_circles_2_2* lexicographically.

See Also

AlgebraicKernelForCircles::CompareX page [595](#)
AlgebraicKernelForCircles::CompareY page [596](#)
CircularKernel::CompareXY_2 page [572](#)

AlgebraicKernelForCircles::SignAt

Definition

A model *fo* of this type must provide:

```
template < class OutputIterator >
CGAL::Sign          fo.operator()( AlgebraicKernelForCircles::Polynomial_1_2 p,
                                   AlgebraicKernelForCircles::Root_for_circles_2_2 r)
```

Computes the sign of polynomial *p* evaluated at a root *r*.

```
template < class OutputIterator >
CGAL::Sign          fo.operator()( AlgebraicKernelForCircles::Polynomial_for_circles_2_2 p,
                                   AlgebraicKernelForCircles::Root_for_circles_2_2 r)
```

Same as previous.

AlgebraicKernelForCircles::ConstructPolynomial_1_2

A model *fo* of this type must provide:

AlgebraicKernelForCircles::Polynomial_1_2

```
fo.operator()( AlgebraicKernelForCircles::RT a,
               AlgebraicKernelForCircles::RT b,
               AlgebraicKernelForCircles::RT c)
```

Constructs polynomial $ax+by+c$.

See Also

CircularKernel::ConstructLine_2page [560](#)
CircularKernel::GetEquationpage [585](#)

AlgebraicKernelForCircles::ConstructPolynomialForCircles_2_2

A model *fo* of this type must provide:

AlgebraicKernelForCircles::PolynomialForCircles_2_2

```
fo.operator()( const AlgebraicKernelForCircles::FT a,
               const AlgebraicKernelForCircles::FT b,
               const AlgebraicKernelForCircles::FT rsq)
```

Constructs polynomial $(x-a)^2 + (y-b)^2 - rsq$.

See Also

CircularKernel::ConstructCircle_2 page [561](#)
CircularKernel::GetEquation page [585](#)

AlgebraicKernelForCircles::Solve

Definition

A model *fo* of this type must provide:

```
template < class OutputIterator >
OutputIterator      fo.operator()( AlgebraicKernelForCircles::Polynomial_1_2 p1,
                                   AlgebraicKernelForCircles::Polynomial_1_2 p2,
                                   OutputIterator res)
```

Copies in the output iterator the common roots of *p1* and *p2*, with their multiplicity, as objects of type *std::pair< AlgebraicKernelForCircles::Root_for_circles_2_2, int>*.

```
template < class OutputIterator >
OutputIterator      fo.operator()( AlgebraicKernelForCircles::Polynomial_1_2 p1,
                                   AlgebraicKernelForCircles::Polynomial_for_circles_2_2 p2,
                                   OutputIterator res)
```

Same as previous.

```
template < class OutputIterator >
OutputIterator      fo.operator()( AlgebraicKernelForCircles::Polynomial_for_circles_2_2 p1,
                                   AlgebraicKernelForCircles::Polynomial_1_2 p2,
                                   OutputIterator res)
```

Same as previous.

```
template < class OutputIterator >
OutputIterator      fo.operator()( AlgebraicKernelForCircles::Polynomial_for_circles_2_2 p1,
                                   AlgebraicKernelForCircles::Polynomial_for_circles_2_2 p2,
                                   OutputIterator res)
```

Same as previous.

AlgebraicKernelForCircles::XCriticalPoints

Definition

A model *fo* of this type must provide:

```
template < class OutputIterator >
OutputIterator      fo.operator()( AlgebraicKernelForCircles::Polynomial_for_circles_2_2 p,
                                   OutputIterator res)
```

Copies in the output iterator the *x*-critical points of polynomial *p*, as objects of type *AlgebraicKernelForCircles::Root_for_circles_2_2*.

```
template < class OutputIterator >
AlgebraicKernelForCircles::Root_for_circles_2_2
```

```
fo( AlgebraicKernelForCircles::Polynomial_for_circles_2_2 p, bool i)
```

Computes the *i*th *x*-critical point of polynomial *p*.

AlgebraicKernelForCircles::YCriticalPoints

Definition

A model *fo* of this type must provide:

```
template < class OutputIterator >
OutputIterator      fo.operator()( AlgebraicKernelForCircles::Polynomial_for_circles_2_2 p,
                                   OutputIterator res)
```

Copies in the output iterator the y-critical points of polynomial *p*, as objects of type *AlgebraicKernelForCircles::Root_for_circles_2_2*.

```
template < class OutputIterator >
AlgebraicKernelForCircles::Root_for_circles_2_2
```

```
fo( AlgebraicKernelForCircles::Polynomial_for_circles_2_2 p, bool i)
```

Computes the *i*th y-critical point of polynomial *p*.

CGAL::Arr_circular_arc_traits<CircularKernel>

#include <CGAL/Arr_circular_arc_traits.h>

Definition

This class is a traits class for CGAL arrangements, built on top of a model of concept *CircularKernel*. It provides curves of type *CGAL::Circular_arc_2<CircularKernel>*.

Is Model for the Concepts

ArrangementTraits_2

CGAL::Arr_line_arc_traits<CircularKernel>

#include <CGAL/Arr_line_arc_traits.h>

Definition

This class is a traits class for CGAL arrangements, built on top of a model of concept *CircularKernel*. It provides curves of type *CGAL::Line_arc_2<CircularKernel>*.

Is Model for the Concepts

ArrangementTraits_2

CGAL::Arr_circular_line_arc_traits<CircularKernel>

`#include <CGAL/Arr_circular_line_arc_traits.h>`

Definition

This class is a traits class for CGAL arrangements, built on top of a model of concept *CircularKernel*. It provides curves that can be of both types *CGAL::Line_arc_2<CircularKernel>* or *CGAL::Circular_arc_2<CircularKernel>*.

It uses the `boost::variant`.

Is Model for the Concepts

ArrangementTraits_2

Part III

Convex Hull Algorithms

Chapter 5

2D Convex Hulls and Extreme Points

Susan Hert and Stefan Schirra

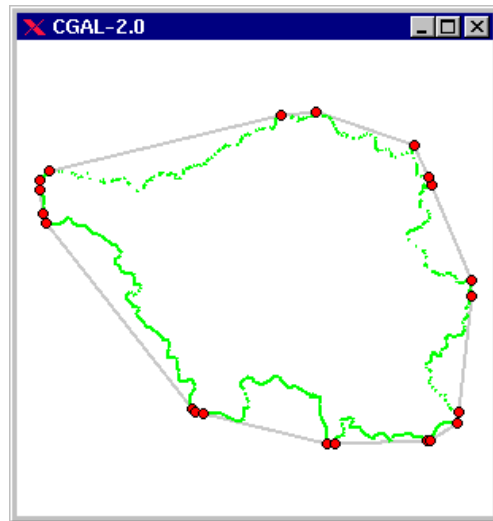
Contents

5.1 Introduction	609
5.2 Convex Hull	610
5.3 Example using Graham-Andrew's Algorithm	610
5.4 Extreme Points and Hull Subsequences	611
5.5 Traits Classes	611
5.6 Convexity Checking	612

5.1 Introduction

A subset $S \subseteq \mathbb{R}^2$ is convex if for any two points p and q in the set the line segment with endpoints p and q is contained in S . The convex hull of a set S is the smallest convex set containing S . The convex hull of a set of points P is a convex polygon with vertices in P . A point in P is an extreme point (with respect to P) if it is a vertex of the convex hull of P . A set of points is said to be strongly convex if it consists of only extreme points.

This chapter describes the functions provided in CGAL for producing convex hulls in two dimensions as well as functions for checking if sets of points are strongly convex or not. There are also a number of functions described for computing particular extreme points and subsequences of hull points, such as the lower and upper hull of a set of points.



5.2 Convex Hull

CGAL provides implementations of several classical algorithms for computing the counterclockwise sequence of extreme points for a set of points in two dimensions (*i.e.*, the counterclockwise sequence of points on the convex hull). The algorithms have different asymptotic running times and require slightly different sets of geometric primitives. Thus you may choose the algorithm that best fits your setting.

Each of the convex hull functions presents the same interface to the user. That is, the user provides a pair of iterators, *first* and *beyond*, an output iterator *result*, and a traits class *traits*. The points in the range $[first, beyond)$ define the input points whose convex hull is to be computed. The counterclockwise sequence of extreme points is written to the sequence starting at position *result*, and the past-the-end iterator for the resulting set of points is returned. The traits classes for the functions specify the types of the input points and the geometric primitives that are required by the algorithms. All functions provide an interface in which this class need not be specified and defaults to types and operations defined in the kernel in which the input point type is defined.

Given a sequence of n input points with h extreme points, the function `convex_hull_2` uses either the output-sensitive $O(nh)$ algorithm of Bykat [Byk78] (a non-recursive version of the quickhull [BDH96] algorithm) or the algorithm of Akl and Toussaint, which requires $O(n \log n)$ time in the worst case. The algorithm chosen depends on the kind of iterator used to specify the input points. These two algorithms are also available via the functions `ch_bykat` and `ch_akl_toussaint`, respectively. Also available are the $O(n \log n)$ Graham-Andrew scan algorithm [And79, Meh84] (`ch_graham_andrew`), the $O(nh)$ Jarvis march algorithm [Jar73] (`ch_jarvis`), and Eddy's $O(nh)$ algorithm [Edd77] (`ch_eddy`), which corresponds to the two-dimensional version of the quickhull algorithm. The linear-time algorithm of Melkman for producing the convex hull of simple polygonal chains (or polygons) is available through the function `ch_melkman`.

5.3 Example using Graham-Andrew's Algorithm

In the following example a convex hull is constructed from point data read from standard input using *Graham-Andrew* algorithm. The resulting convex polygon is shown at the standard output console. The same results could be achieved by substituting the function `CGAL::ch_graham_andrew` by other function like `CGAL::ch_bykat`.

```
// file: examples/Convex_hull_2/ch_example_from_cin_to_cout.C
```

```

#include <CGAL/Cartesian.h>
#include <CGAL/ch_graham_andrew.h>

typedef    CGAL::Point_2<CGAL::Cartesian<double> >    Point_2;

int main()
{
    CGAL::set_ascii_mode(std::cin);
    CGAL::set_ascii_mode(std::cout);
    std::istream_iterator< Point_2 >  in_start( std::cin );
    std::istream_iterator< Point_2 >  in_end;
    std::ostream_iterator< Point_2 >  out( std::cout, "\n" );
    CGAL::ch_graham_andrew( in_start, in_end, out );
    return 0;
}

```

5.4 Extreme Points and Hull Subsequences

In addition to the functions for producing convex hulls, there are a number of functions for computing sets and sequences of points related to the convex hull. The functions *lower_hull_points_2* and *upper_hull_points_2* provide the computation of the counterclockwise sequence of extreme points on the lower hull and upper hull, respectively. The algorithm used in these functions is Andrew's variant of Graham's scan algorithm [And79, Meh84], which has worst-case running time of $O(n \log n)$.

There are also functions available for computing certain subsequences of the sequence of extreme points on the convex hull. The function *ch_jarvis_march* generates the counterclockwise ordered subsequence of extreme points between a given pair of points and *ch_graham_andrew_scan* computes the sorted sequence of extreme points that are not left of the line defined by the first and last input points.

Finally, a set of functions (*ch_nswe_point*, *ch_ns_point*, *ch_we_point*, *ch_n_point*, *ch_s_point*, *ch_w_point*, *ch_e_point*) is provided for computing extreme points of a 2D point set in the coordinate directions.

5.5 Traits Classes

Each of the functions used to compute convex hulls or extreme points is parameterized by a traits class, which specifies the types and geometric primitives to be used in the computation. There are several implementations of 2D traits classes provided in the library. The class *Convex_hull_traits_2<R>* corresponds to the default traits class that provides the types and predicates presented in the 2-dimensional CGAL kernel in which the input points lie. The class *Convex_hull_constructive_traits<R>* is a second traits class based on CGAL primitives but differs from *Convex_hull_traits_2* in that some of its primitives reuse intermediate results to speed up computation. In addition, there are three projective traits classes (*Convex_hull_projective_xy_traits_2*, *Convex_hull_projective_xz_traits_2*, and *Convex_hull_projective_yz_traits_2*), which may be used to compute the convex hull of a set of three-dimensional points projected into each of the three coordinate planes.

5.6 Convexity Checking

The functions *is_ccw_strongly_convex_2* and *is_cw_strongly_convex_2* check whether a given sequence of 2D points forms a (counter)clockwise strongly convex polygon.. These are used in postcondition testing of the two-dimensional convex hull functions.

2D Convex Hulls and Extreme Points

Reference Manual

Susan Hert and Stefan Schirra

A subset $S \subseteq \mathbb{R}^2$ is convex if for any two points p and q in the set the line segment with endpoints p and q is contained in S . The convex hull of a set S is the smallest convex set containing S . The convex hull of a set of points P is a convex polygon with vertices in P . A point in P is an extreme point (with respect to P) if it is a vertex of the convex hull of P .

CGAL provides functions for computing convex hulls in two dimensions as well as functions for testing if a given set of points is strongly convex or not. There are also a number of functions available for computing particular extreme points in 2D and subsequences of the hull points, such as the lower hull or upper hull of a set of points.

5.7 Classified Reference Pages

Assertions

The assertion flags for the convex hull and extreme point algorithms use *CH* in their names (e.g., *CGAL_CH_NO_POSTCONDITIONS*). For the convex hull algorithms, the postcondition check tests only convexity (if not disabled), but not containment of the input points in the polygon or polyhedron defined by the output points. The latter is considered an expensive checking and can be enabled by defining *CGAL_CH_CHECK_EXPENSIVE*.

Concepts

ConvexHullTraits_2.....page [641](#)

Traits Classes

CGAL::Convex_hull_constructive_traits_2<R>page [643](#)
CGAL::Convex_hull_projective_xy_traits_2<Point_3>page [644](#)
CGAL::Convex_hull_projective_xz_traits_2<Point_3>page [645](#)
CGAL::Convex_hull_projective_yz_traits_2<Point_3>page [646](#)
CGAL::Convex_hull_traits_2<R>page [647](#)

Convex Hull Functions

<i>CGAL::ch_aki_toussaint</i>	page 616
<i>CGAL::ch_bykat</i>	page 618
<i>CGAL::ch_eddy</i>	page 620
<i>CGAL::ch_graham_andrew</i>	page 623
<i>CGAL::ch_jarvis</i>	page 627
<i>CGAL::ch_melkman</i>	page 631
<i>CGAL::convex_hull_2</i>	page 639

Convexity Checking Functions

<i>CGAL::is_ccw_strongly_convex_2</i>	page 648
<i>CGAL::is_cw_strongly_convex_2</i>	page 649

Hull Subsequence Functions

<i>CGAL::ch_graham_andrew_scan</i>	page 625
<i>CGAL::ch_jarvis_march</i>	page 629
<i>CGAL::lower_hull_points_2</i>	page 650
<i>CGAL::upper_hull_points_2</i>	page 652

Extreme Point Functions

<i>CGAL::ch_e_point</i>	page 622
<i>CGAL::ch_nswe_point</i>	page 633
<i>CGAL::ch_n_point</i>	page 635
<i>CGAL::ch_ns_point</i>	page 634
<i>CGAL::ch_s_point</i>	page 636
<i>CGAL::ch_w_point</i>	page 638
<i>CGAL::ch_we_point</i>	page 637

5.8 Alphabetical List of Reference Pages

<i>ch_akl_toussaint</i>	page 616
<i>ch_bykat</i>	page 618
<i>ch_eddy</i>	page 620
<i>ch_e_point</i>	page 622
<i>ch_graham_andrew_scan</i>	page 625
<i>ch_graham_andrew</i>	page 623
<i>ch_jarvis_march</i>	page 629
<i>ch_jarvis</i>	page 627
<i>ch_melkman</i>	page 631
<i>ch_nswe_point</i>	page 633
<i>ch_ns_point</i>	page 634
<i>ch_n_point</i>	page 635
<i>ch_s_point</i>	page 636
<i>ch_we_point</i>	page 637
<i>ch_w_point</i>	page 638
<i>ConvexHullTraits_2</i>	page 641
<i>convex_hull_2</i>	page 639
<i>Convex_hull_constructive_traits_2<R></i>	page 643
<i>Convex_hull_projective_xy_traits_2<Point_3></i>	page 644
<i>Convex_hull_projective_xz_traits_2<Point_3></i>	page 645
<i>Convex_hull_projective_yz_traits_2<Point_3></i>	page 646
<i>Convex_hull_traits_2<R></i>	page 647
<i>is_ccw_strongly_convex_2</i>	page 648
<i>is_cw_strongly_convex_2</i>	page 649
<i>lower_hull_points_2</i>	page 650
<i>upper_hull_points_2</i>	page 652

CGAL::ch_aki_toussaint

Definition

The function *ch_aki_toussaint* generates the counterclockwise sequence of extreme points from a given set of input points.

```
#include <CGAL/ch_aki_toussaint.h>
```

```
template <class ForwardIterator, class OutputIterator, class Traits>
OutputIterator      ch_aki_toussaint( ForwardIterator first,
                                     ForwardIterator beyond,
                                     OutputIterator result,
                                     Traits ch_traits = Default_traits())
```

generates the counterclockwise sequence of extreme points of the points in the range *[first,beyond)*. The resulting sequence is placed starting at position *result*, and the past-the-end iterator for the resulting sequence is returned. It is not specified at which point the cyclic sequence of extreme points is cut into a linear sequence.

Precondition: The source range *[first,beyond)* does not contain *result*.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

Requirements

1. *ForwardIterator::value_type* and *OutputIterator::value_type* are equivalent to *Traits::Point_2*.
2. *Traits* defines the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:
 - *Traits::Point_2*,
 - *Traits::Less_xy_2*,
 - *Traits::Less_yx_2*,
 - *Traits::Left_turn_2*,
 - *Traits::Equal_2*.

See Also

CGAL::ch_bykat page 618
CGAL::ch_eddy page 620
CGAL::ch_graham_andrew page 623
CGAL::ch_jarvis page 627
CGAL::ch_melkman page 631
CGAL::convex_hull_2 page 639

Implementation

This function uses the algorithm of Akl and Toussaint [[AT78](#)] that requires $O(n \log n)$ time for n input points.

CGAL::ch_bykat

Definition

The function *ch_bykat* generates the counterclockwise sequence of extreme points from a given set of input points.

```
#include <CGAL/ch_bykat.h>
```

```
template <class InputIterator, class OutputIterator, class Traits>
OutputIterator      ch_bykat( InputIterator first,
                             InputIterator beyond,
                             OutputIterator result,
                             Traits ch_traits = Default_traits)
```

generates the counterclockwise sequence of extreme points of the points in the range *[first,beyond)*. The resulting sequence is placed starting at position *result*, and the past-the-end iterator for the resulting sequence is returned. It is not specified at which point the cyclic sequence of extreme points is cut into a linear sequence.

Precondition: The source range *[first,beyond)* does not contain *result*.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

Requirements

1. *InputIterator::value_type* and *OutputIterator::value_type* are equivalent to *Traits::Point_2*.
2. *Traits* defines the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:
 - *Traits::Point_2*,
 - *Traits::Less_signed_distance_to_line_2*,
 - *Traits::Left_turn_2*,
 - *Traits::Less_xy_2*,
 - *Traits::Equal_2*.

See Also

CGAL::ch_akl_toussaint page 616
CGAL::ch_eddy page 620
CGAL::ch_graham_andrew page 623
CGAL::ch_jarvis page 627
CGAL::ch_melkman page 631
CGAL::convex_hull_2 page 639

Implementation

This function implements the non-recursive variation of Eddy's algorithm [Edd77] described in [Byk78]. This algorithm requires $O(nh)$ time in the worst case for n input points with h extreme points.

CGAL::ch_eddy

Definition

The function *ch_eddy* generates the counterclockwise sequence of extreme points from a given set of input points.

```
#include <CGAL/ch_eddy.h>
```

```
template <class InputIterator, class OutputIterator, class Traits>
OutputIterator      ch_eddy( InputIterator first,
                             InputIterator beyond,
                             OutputIterator result,
                             Traits ch_traits = Default_traits)
```

generates the counterclockwise sequence of extreme points of the points in the range *[first,beyond)*. The resulting sequence is placed starting at position *result*, and the past-the-end iterator for the resulting sequence is returned. It is not specified at which point the cyclic sequence of extreme points is cut into a linear sequence.

Precondition: The source range *[first,beyond)* does not contain *result*.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

Requirements

1. *InputIterator::value_type* and *OutputIterator::value_type* are equivalent to *Traits::Point_2*.
2. *Traits* defines the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:
 - *Traits::Point_2*,
 - *Traits::Equal_2*,
 - *Traits::Less_signed_distance_to_line_2*,
 - *Traits::Left_turn_2*,
 - *Traits::Less_xy_2*.

See Also

CGAL::ch_akl_toussaint page [616](#)
CGAL::ch_bykat page [618](#)
CGAL::ch_graham_andrew page [623](#)
CGAL::ch_jarvis page [627](#)
CGAL::ch_melkman page [631](#)
CGAL::convex_hull_2 page [639](#)

Implementation

This function implements Eddy's algorithm [Edd77], which is the two-dimensional version of the quickhull algorithm [BDH96] . This algorithm requires $O(nh)$ time in the worst case for n input points with h extreme points.

CGAL::ch_e_point

Definition

The function *ch_e_point* finds a point of a given set of input points with maximal x coordinate.

```
#include <CGAL/ch_selected_extreme_points_2.h>
```

```
template <class ForwardIterator>
void      ch_e_point( ForwardIterator first,
                     ForwardIterator beyond,
                     ForwardIterator& e,
                     Traits ch_traits = Default_traits)
```

traverses the range $[first, beyond)$. After execution, the value of e is an iterator in the range such that $*e \geq_{xy} *it$ for all iterators it in the range.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

Requirements

Traits defines a type *Traits::Less_xy_2* as described in the concept *ConvexHullTraits_2* and the corresponding member function that returns an instance of this type.

See Also

CGAL::ch_nsw_e_pointpage [633](#)
CGAL::ch_n_pointpage [635](#)
CGAL::ch_ns_pointpage [634](#)
CGAL::ch_s_pointpage [636](#)
CGAL::ch_w_pointpage [638](#)
CGAL::ch_we_pointpage [637](#)

CGAL::ch_graham_andrew

Definition

The function `ch_graham_andrew` generates the counterclockwise sequence of extreme points from a given set of input points.

```
#include <CGAL/ch_graham_andrew.h>
```

```
template <class InputIterator, class OutputIterator, class Traits>
OutputIterator      ch_graham_andrew( InputIterator first,
                                     InputIterator beyond,
                                     OutputIterator result,
                                     Traits ch_traits = Default_traits)
```

generates the counterclockwise sequence of extreme points of the points in the range $[first, beyond)$. The resulting sequence is placed starting at position *result*, and the past-the-end iterator for the resulting sequence is returned. It is not specified at which point the cyclic sequence of extreme points is cut into a linear sequence.

Precondition: The source range $[first, beyond)$ does not contain *result*.

The default traits class *Default_traits* is the kernel in which the type *InputIterator::value_type* is defined.

Requirements

1. *InputIterator::value_type* and *OutputIterator::value_type* are equivalent to *Traits::Point_2*.
2. *Traits* defines the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:
 - *Traits::Point_2*,
 - *Traits::Less_xy_2*,
 - *Traits::Left_turn_2*,
 - *Traits::Equal_2*.

See Also

CGAL::ch_akl_toussaint page [616](#)
CGAL::ch_bykat page [618](#)
CGAL::ch_eddy page [620](#)
CGAL::ch_graham_andrew_scan page [625](#)
CGAL::ch_jarvis page [627](#)
CGAL::ch_melkman page [631](#)
CGAL::convex_hull_2 page [639](#)

<i>CGAL::lower_hull_points_2</i>	page 650
<i>CGAL::upper_hull_points_2</i>	page 652

Implementation

This function implements Andrew’s variant of the Graham scan algorithm [[And79](#)] and follows the presentation of Mehlhorn [[Meh84](#)]. This algorithm requires $O(n \log n)$ time in the worst case for n input points.

CGAL::ch_graham_andrew_scan

Definition

The function `ch_graham_andrew_scan` generates the counterclockwise sequence of extreme points from a given set of input points that are not left of the line defined by the first and last points in this sequence.

```
#include <CGAL/ch_graham_andrew.h>
```

```
template <class BidirectionalIterator, class OutputIterator, class Traits>
OutputIterator      ch_graham_andrew_scan( BidirectionalIterator first,
                                           BidirectionalIterator beyond,
                                           OutputIterator result,
                                           Traits ch_traits = Default_traits)
```

generates the counterclockwise sequence of extreme points that are not left of pq , where p is the value of *first* and q is the value of *beyond* $- 1$. The resulting sequence is placed starting at *result* with p ; point q is omitted. The past-the-end iterator for the sequence is returned.

Precondition: The range $[first, beyond)$ contains at least two different points. The points in $[first, beyond)$ are “sorted” with respect to pq , i.e., the sequence of points in $[first, beyond)$ define a counterclockwise polygon, for which the Graham-Sklansky-procedure [Sk172] works.

The default traits class *Default_traits* is the kernel in which the type *BidirectionalIterator::value_type* is defined.

Requirements

1. *BidirectionalIterator::value_type* and *OutputIterator::value_type* are equivalent to *Traits::Point_2*.
2. *Traits* defines the following two types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:
 - *Traits::Point_2*,
 - *Traits::Left_turn_2*.

See Also

CGAL::ch_graham_andrew page 623
 CGAL::lower_hull_points_2 page 650
 CGAL::upper_hull_points_2 page 652

Implementation

The function uses Andrew’s variant of the Graham scan algorithm [And79] . This algorithm requires $O(n \log n)$ time in the worst case for n input points.

Example

In the following example *ch_graham_andrew_scan()* is used to realize Anderson's variant [And78] of the Graham Scan [Gra72]. The points are sorted counterclockwise around the leftmost point using the *Less_rotate_ccw_2* predicate, as defined in the concept *ConvexHullTraits_2*. According to the definition of *Less_rotate_ccw_2*, the leftmost point is the last point in the sorted sequence and its predecessor on the convex hull is the first point in the sorted sequence. It is not hard to see that the preconditions of *ch_graham_andrew_scan()* are satisfied. Anderson's variant of the Graham scan is usually inferior to Andrew's variant because of its higher arithmetic demand.

```
template <class InputIterator, class OutputIterator, class Traits>
OutputIterator
ch_graham_anderson( InputIterator first, InputIterator beyond,
                    OutputIterator result, const Traits& ch_traits)
{
    typedef typename Traits::Less_xy_2          Less_xy_2;
    typedef typename Traits::Point_2           Point_2;
    typedef typename Traits::Less_rotate_ccw_2  Less_rotate_ccw_2;

    if (first == beyond) return result;
    std::vector< Point_2 > V;
    copy( first, beyond, back_inserter(V) );
    typename std::vector< Point_2 >::iterator it =
        std::min_element(V.begin(), V.end(), Less_xy_2());
    std::sort( V.begin(), V.end(), CGAL::bind_1(Less_rotate_ccw_2(), *it) );
    if ( *(V.begin()) == *(V.rbegin()) )
    {
        *result = *(V.begin()); ++result;
        return result;
    }
    return ch_graham_andrew_scan( V.begin(), V.end(), result, ch_traits);
}
```

CGAL::ch_jarvis

Definition

The function *ch_jarvis* generates the counterclockwise sequence of extreme points from a given set of input points.

```
#include <CGAL/ch_jarvis.h>
```

```
template <class InputIterator, class OutputIterator, class Traits>
OutputIterator      ch_jarvis( InputIterator first,
                              InputIterator beyond,
                              OutputIterator result,
                              Traits ch_traits = Default_traits)
```

generates the counterclockwise sequence of extreme points of the points in the range *[first,beyond)*. The resulting sequence is placed starting at position *result*, and the past-the-end iterator for the resulting sequence is returned. It is not specified at which point the cyclic sequence of extreme points is cut into a linear sequence.

Precondition: The source range *[first,beyond)* does not contain *result*.

The default traits class *Default_traits* is the kernel in which the type *InputIterator::value_type* is defined.

Requirements

1. *InputIterator::value_type* and *OutputIterator::value_type* are equivalent to *Traits::Point_2*.
2. *Traits* defines the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:
 - *Traits::Point_2*,
 - *Traits::Equal_2*,
 - *Traits::Less_rotate_ccw_2*,
 - *Traits::Less_xy_2*.

See Also

CGAL::ch_akl_toussaint page 616
CGAL::ch_bykat page 618
CGAL::ch_eddy page 620
CGAL::ch_graham_andrew page 623
CGAL::ch_jarvis_march page 629
CGAL::ch_melkman page 631
CGAL::convex_hull_2 page 639

Implementation

This function uses the Jarvis march (gift-wrapping) algorithm [Jar73]. This algorithm requires $O(nh)$ time in the worst case for n input points with h extreme points.

CGAL::ch_jarvis_march

Definition

The function *ch_jarvis_march* generates the counterclockwise sequence of extreme points from a given set of input points that line between two input points.

```
#include <CGAL/ch_jarvis.h>
```

```
template <class ForwardIterator, class OutputIterator, class Traits>
OutputIterator ch_jarvis_march( ForwardIterator first,
                               ForwardIterator beyond,
                               Traits::Point_2 start_p,
                               Traits::Point_2 stop_p,
                               OutputIterator result,
                               Traits ch_traits = Default_traits)
```

generates the counterclockwise subsequence of extreme points between *start_p* and *stop_p* of the points in the range *[first,beyond)*, starting at position *result* with point *start_p*. The last point generated is the point preceding *stop_p* in the counterclockwise order of extreme points.

Precondition: *start_p* and *stop_p* are extreme points with respect to the points in the range *[first,beyond)* and *stop_p* is an element of range *[first,beyond)*.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

Requirements

1. *ForwardIterator::value_type* and *OutputIterator::value_type* are equivalent to *Traits::Point_2*.
2. *Traits* defines the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:
 - *Traits::Point_2*,
 - *Traits::Equal_2*,
 - *Traits::Less_rotate_ccw_2*.

See Also

CGAL::ch_jarvispage [627](#)
CGAL::lower_hull_points_2page [650](#)
CGAL::upper_hull_points_2page [652](#)

Implementation

The function uses the Jarvis march (gift-wrapping) algorithm [Jar73]. This algorithm requires $O(nh)$ time in the worst case for n input points with h extreme points.

CGAL::ch_melkman

Definition

The function *ch_melkman* computes the counterclockwise sequence of extreme points of a sequence of points that forms a simple polyline or polygon.

```
#include <CGAL/ch_melkman.h>
```

```
template <class InputIterator, class OutputIterator>
OutputIterator      ch_melkman( InputIterator first,
                               InputIterator last,
                               OutputIterator result,
                               Traits ch_traits = Default_traits)
```

generates the counterclockwise sequence of extreme points of the points in the range $[first, beyond)$. The resulting sequence is placed starting at position *result*, and the past-the-end iterator for the resulting sequence is returned.

Precondition: The source range $[first, beyond)$ corresponds to a simple polyline. $[first, beyond)$ does not contain *result*

The default traits class *Default_traits* is the kernel in which the type *InputIterator::value_type* is defined.

Requirements

1. *InputIterator::value_type* and *OutputIterator::value_type* are equivalent to *Traits::Point_2*.
2. *Traits* contains the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:
 - *Traits::Point_2*,
 - *Traits::Equal_2*,
 - *Traits::Less_xy_2*,
 - *Traits::Left_turn_2*.

See Also

CGAL::ch_akl_toussaint page 616
CGAL::ch_bykat page 618
CGAL::ch_eddy page 620
CGAL::ch_graham_andrew page 623
CGAL::ch_jarvis page 627
CGAL::ch_melkman page 631
CGAL::convex_hull_2 page 639

Implementation

It uses an implementation of Melkman's algorithm [Mel87]. Running time of this is linear.

CGAL::ch_nswe_point

Definition

The function *ch_nswe_point* finds the four extreme points of a given set of input points using a linear scan of the input points. That is, it determines the points with maximal y, minimal y, minimal x, and maximal x coordinates.

```
#include <CGAL/ch_selected_extreme_points_2.h>
```

```
template <class ForwardIterator>
void ch_nswe_point( ForwardIterator first,
                   ForwardIterator beyond,
                   ForwardIterator& n,
                   ForwardIterator& s,
                   ForwardIterator& w,
                   ForwardIterator& e,
                   Traits ch_traits = Default_traits)
```

traverses the range $[first, beyond)$. After execution, the value of n is an iterator in the range such that $*n \geq_{yx} *it$ for all iterators it in the range. Similarly, for s , w , and e the inequalities $*s \leq_{yx} *it$, $*w \leq_{xy} *it$, and $*e \geq_{xy} *it$ hold for all iterators it in the range.

Requirements

Traits contains the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:

- *Traits::Less_xy_2*,
- *Traits::Less_yx_2*.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

See Also

CGAL::ch_e_point page [622](#)
CGAL::ch_n_point page [635](#)
CGAL::ch_ns_point page [634](#)
CGAL::ch_s_point page [636](#)
CGAL::ch_w_point page [638](#)
CGAL::ch_we_point page [637](#)

CGAL::ch_ns_point

Definition

The function *ch_ns_point* finds the points of a given set of input points with minimal and maximal x coordinates.

```
#include <CGAL/ch_selected_extreme_points_2.h>
```

```
template <class ForwardIterator>
void      ch_ns_point( ForwardIterator first,
                      ForwardIterator beyond,
                      ForwardIterator& n,
                      ForwardIterator& s,
                      Traits ch_traits = Default_traits)
```

traverses the range $[first, beyond)$. After execution, the value of n is an iterator in the range such that $*n \geq_{yx} *it$ for all iterators it in the range. Similarly, for s the inequality $*s \leq_{yx} *it$ holds for all iterators in the range.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

Requirements

Traits defines the type *Traits::Less_yx_2* as specified in the concept *ConvexHullTraits_2* and the corresponding member function that returns an instance of this type.

See Also

CGAL::ch_e_point page [622](#)
CGAL::ch_nswe_point page [633](#)
CGAL::ch_n_point page [635](#)
CGAL::ch_s_point page [636](#)
CGAL::ch_w_point page [638](#)
CGAL::ch_we_point page [637](#)

CGAL::ch_n_point

Definition

The function *ch_n_point* finds a point in a given set of input points with maximal y coordinate.

```
#include <CGAL/ch_selected_extreme_points_2.h>
```

```
template <class ForwardIterator>
void      ch_n_point( ForwardIterator first,
                      ForwardIterator beyond,
                      ForwardIterator& n,
                      Traits ch_traits = Default_traits)
```

traverses the range $[first, beyond)$. After execution, the value of *n* is an iterator in the range such that $*n \geq_{yx} *it$ for all iterators *it* in the range.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

Requirements

Traits defines the type *Traits::Less_yx_2* as specified in the concept *ConvexHullTraits_2* and the corresponding member function that returns an instance of this type.

See Also

CGAL::ch_e_point page [622](#)
CGAL::ch_nswe_point page [633](#)
CGAL::ch_ns_point page [634](#)
CGAL::ch_s_point page [636](#)
CGAL::ch_w_point page [638](#)
CGAL::ch_we_point page [637](#)

CGAL::ch_s_point

Definition

The function *ch_s_point* finds a points in a given set of input points with minimal y coordinates.

```
#include <CGAL/ch_selected_extreme_points_2.h>
```

```
template <class ForwardIterator>
void      ch_s_point( ForwardIterator first,
                     ForwardIterator beyond,
                     ForwardIterator& s,
                     Traits ch_traits = Default_traits)
```

traverses the range *[first,beyond)*. After execution, the value of *s* is an iterator in the range such that $*s \leq_{yx} *it$ for all iterators *it* in the range.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

Requirements

Traits defines the type *Traits::Less_yx_2* as specified in the concept *ConvexHullTraits_2* and the corresponding member function that returns an instance of this type.

See Also

CGAL::ch_e_point page [622](#)
CGAL::ch_nswe_point page [633](#)
CGAL::ch_n_point page [635](#)
CGAL::ch_ns_point page [634](#)
CGAL::ch_w_point page [638](#)
CGAL::ch_we_point page [637](#)

CGAL::ch_we_point

Definition

The function *ch_we_point* finds two points of a given set of input points with minimal and maximal x coordinates.

```
#include <CGAL/ch_selected_extreme_points_2.h>
```

```
template <class ForwardIterator>
void      ch_we_point( ForwardIterator first,
                      ForwardIterator beyond,
                      ForwardIterator& w,
                      ForwardIterator& e,
                      Traits ch_traits = Default_traits)
```

traverses the range $[first, beyond)$. After execution, the value of w is an iterator in the range such that $*w \leq_{xy} *it$ for all iterators it in the range. Similarly, for e the inequality $*e \geq_{xy} *it$ holds for all iterators in the range.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

Requirements

Traits defines the type *Traits::Less_xy_2* as specified in the concept *ConvexHullTraits_2* and the corresponding member function that returns an instance of this type.

See Also

CGAL::ch_e_point page [622](#)
CGAL::ch_nswe_point page [633](#)
CGAL::ch_n_point page [635](#)
CGAL::ch_ns_point page [634](#)
CGAL::ch_s_point page [636](#)
CGAL::ch_w_point page [638](#)

CGAL::ch_w_point

Definition

The function *ch_w_point* finds a point in a given set of input points with minimal x coordinate.

```
#include <CGAL/ch_selected_extreme_points_2.h>
```

```
template <class ForwardIterator>
void ch_w_point( ForwardIterator first,
                 ForwardIterator beyond,
                 ForwardIterator& w,
                 Traits ch_traits = Default_traits)
```

traverses the range $[first, beyond)$. After execution, the value of w is an iterator in the range such that $*w \leq_{xy} *it$ for all iterators it in the range.

Requirements

Traits defines the type *Traits::Less_xy_2* as specified in the concept *ConvexHullTraits_2* and the corresponding member function that returns an instance of this type.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

See Also

CGAL::ch_e_point page [622](#)
CGAL::ch_nsw_e_point page [633](#)
CGAL::ch_n_point page [635](#)
CGAL::ch_ns_point page [634](#)
CGAL::ch_s_point page [636](#)
CGAL::ch_we_point page [637](#)

CGAL::convex_hull_2

Definition

The function *convex_hull_2* generates the counterclockwise sequence of extreme points from a given set of input points.

```
#include <CGAL/convex_hull_2.h>
```

```
template <class InputIterator, class OutputIterator>
OutputIterator          convex_hull_2( InputIterator first,
                                      InputIterator beyond,
                                      OutputIterator result,
                                      Traits ch_traits = Default_traits)
```

generates the counterclockwise sequence of extreme points of the points in the range *[first,beyond)*. The resulting sequence is placed starting at position *result*, and the past-the-end iterator for the resulting sequence is returned. It is not specified at which point the cyclic sequence of extreme points is cut into a linear sequence.

Precondition: The source range *[first,beyond)* does not contain *result*.

The default traits class *Default_traits* is the kernel in which the type *InputIterator::value_type* is defined.

Requirements

1. *InputIterator::value_type* and *OutputIterator::value_type* are equivalent to *Traits::Point_2*.
2. *Traits* contains the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:
 - *Traits::Point_2*,
 - *Traits::Equal_2*,
 - *Traits::Less_xy_2*,
 - *Traits::Less_yx_2*,
 - *Traits::Left_turn_2*.

See Also

CGAL::ch_akl_toussaint page [616](#)
CGAL::ch_bykat page [618](#)
CGAL::ch_eddy page [620](#)
CGAL::ch_graham_andrew page [623](#)
CGAL::ch_jarvis page [627](#)
CGAL::ch_melkman page [631](#)

Implementation

One of two algorithms is used, depending on the type of iterator used to specify the input points. For input iterators, the algorithm used is that of Bykat [Byk78], which has a worst-case running time of $O(nh)$, where n is the number of input points and h is the number of extreme points. For all other types of iterators, the $O(n \log n)$ algorithm of Akl and Toussaint [AT78] is used.

Example

In the following example we use the STL-compliant interface of *CGAL::Polygon_2* to construct the convex hull polygon from the sequence of extreme points. Point data are read from standard input, the convex hull polygon is shown in a CGAL window. Remember, that when no traits class is specified for the function *convex_hull_2*, the kernel from which the input points come is used as the default traits class.

ConvexHullTraits_2

Definition

All convex hull and extreme point algorithms provided in CGAL are parameterized with a traits class *Traits*, which defines the primitives (objects and predicates) that the convex hull algorithms use. ConvexHullTraits_2 defines the complete set of primitives required in these functions. The specific subset of these primitives required by each function is specified with each function.

Types

<i>ConvexHullTraits_2:: Point_2</i>	The point type on which the convex hull functions operate.
<i>ConvexHullTraits_2:: Equal_2</i>	Binary predicate object type comparing <i>Point_2</i> s. Must provide <i>bool operator()(Point_2 p, Point_2 q)</i> where <i>true</i> is returned iff $p ==_{xy} q$, false otherwise.
<i>ConvexHullTraits_2:: Less_xy_2</i>	Binary predicate object type comparing <i>Point_2</i> s lexicographically. Must provide <i>bool operator()(Point_2 p, Point_2 q)</i> where <i>true</i> is returned iff $p <_{xy} q$. We have $p <_{xy} q$, iff $p_x < q_x$ or $p_x = q_x$ and $p_y < q_y$, where p_x and p_y denote x and y coordinate of point p , respectively.
<i>ConvexHullTraits_2:: Less_yx_2</i>	Same as <i>Less_xy_2</i> with the roles of x and y interchanged.
<i>ConvexHullTraits_2:: Left_turn_2</i>	Predicate object type that must provide <i>bool operator()(Point_2 p, Point_2 q, Point_2 r)</i> , which returns <i>true</i> iff r lies to the left of the oriented line through p and q .
<i>ConvexHullTraits_2:: Less_signed_distance_to_line_2</i>	Predicate object type that must provide <i>bool operator()(Point_2 p, Point_2 q, Point_2 r, Point_2 s)</i> , which returns <i>true</i> iff the signed distance from r to the line l_{pq} through p and q is smaller than the distance from s to l_{pq} . It is used to compute the point right of a line with maximum unsigned distance to the line. The predicate must provide a total order compatible with convexity, <i>i.e.</i> , for any line segment s one of the endpoints of s is the smallest point among the points on s , with respect to the order given by <i>Less_signed_distance_to_line_2</i> .
<i>ConvexHullTraits_2:: Less_rotate_ccw_2</i>	Predicate object type that must provide <i>bool operator()(Point_2 e, Point_2 p, Point_2 q)</i> , where <i>true</i> is returned iff a tangent at e to the point set $\{e, p, q\}$ hits p before q when rotated counterclockwise around e . Ties are broken such that the point with larger distance to e is smaller!

Creation

Only a copy constructor is required.

```
ConvexHullTraits_2 traits( & t);
```

Operations

The following member functions to create instances of the above predicate object types must exist.

```
Equal_2                traits.equal_2_object()  
Less_xy_2             traits.less_xy_2_object()  
Less_yx_2             traits.less_yx_2_object()  
Less_signed_distance_to_line_2  
                        traits.less_signed_distance_to_line_2_object()  
Less_rotate_ccw_2     traits.less_rotate_ccw_2_object()  
Left_turn_2           traits.left_turn_2_object()
```

Has Models

```
CGAL::Convex_hull_constructive_traits_2<R> ..... page 643  
CGAL::Convex_hull_projective_xy_traits_2<Point_3> ..... page 644  
CGAL::Convex_hull_projective_xz_traits_2<Point_3> ..... page 645  
CGAL::Convex_hull_projective_yz_traits_2<Point_3> ..... page 646  
CGAL::Convex_hull_traits_2<R> ..... page 647
```

See Also

```
IsStronglyConvexTraits_3 ..... page 679
```

CGAL::Convex_hull_constructive_traits_2<R>

Definition

The class *Convex_hull_constructive_traits_2<R>* serves as a traits class for all the two-dimensional convex hull and extreme point calculation function. Unlike the class *CGAL::Convex_hull_traits_2<R>*, this class makes use of previously computed results to avoid redundancy. For example, in the sidedness tests, lines (of type *R::Line_2*) are constructed, which is equivalent to the precomputation of subdeterminants of the orientation-determinant for three points.

```
#include <CGAL/convex_hull_constructive_traits_2.h>
```

Is Model for the Concepts

ConvexHullTraits_2.....page [641](#)

Types

<i>typedef R::Point_2</i>	<i>Point_2;</i>
<i>typedef R::Less_xy_2</i>	<i>Less_xy_2;</i>
<i>typedef R::Less_yx_2</i>	<i>Less_yx_2;</i>
<i>typedef CGAL::r_Less_dist_to_line<R></i>	<i>Less_signed_distance_to_line_2;</i>
<i>typedef R::Less_rotate_ccw</i>	<i>Less_rotate_ccw_2;</i>
<i>typedef R::Left_turn_2</i>	<i>Left_turn_2;</i>
<i>typedef R::Equal_2</i>	<i>Equal_2;</i>

Creation

Convex_hull_constructive_traits_2<R> traits; default constructor.

Operations

<i>Less_xy_2</i>	<i>traits.less_xy_2_object()</i>
<i>Less_yx_2</i>	<i>traits.less_yx_2_object()</i>
<i>Less_signed_distance_to_line_2</i>	<i>traits.less_signed_distance_to_line_2_object()</i>
<i>Less_rotate_ccw_2</i>	<i>traits.less_rotate_ccw_2_object()</i>
<i>Left_turn_2</i>	<i>traits.left_turn_2_object()</i>
<i>Equal_2</i>	<i>traits.equal_2_object()</i>

See Also

CGAL::Convex_hull_projective_xy_traits_2<Point_3>page [644](#)
CGAL::Convex_hull_projective_xz_traits_2<Point_3>page [645](#)
CGAL::Convex_hull_projective_yz_traits_2<Point_3>page [646](#)
CGAL::Convex_hull_traits_2<R>page [647](#)

CGAL::Convex_hull_projective_xy_traits_2<Point_3>

Definition

The class *Convex_hull_projective_xy_traits_2<Point_3>* serves as a traits class for all the two-dimensional convex hull and extreme point calculation function. This class can be used to compute the convex hull of a set of 3D points projected onto the *xy* plane (*i.e.*, by ignoring the *z* coordinate).

```
#include <CGAL/Convex_hull_projective_xy_traits_2.h>
```

Is Model for the Concepts

ConvexHullTraits_2.....page [641](#)

Types

<i>typedef Point_3</i>	<i>Point_2;</i>
<i>typedef Less_xy_plane_xy_2<Point_3></i>	<i>Less_xy_2;</i>
<i>typedef Less_yx_plane_xy_2<Point_3></i>	<i>Less_yx_2;</i>
<i>typedef Less_dist_to_line_plane_xy_2<Point_3></i>	<i>Less_signed_distance_to_line_2;</i>
<i>typedef Less_rotate_ccw_plane_xy_2<Point_3></i>	<i>Less_rotate_ccw_2;</i>
<i>typedef Left_turn_plane_xy_2<Point_3></i>	<i>Left_turn_2;</i>
<i>typedef Equal_xy_plane_xy_2<Point_3></i>	<i>Equal_2;</i>

Creation

Convex_hull_projective_xy_traits_2<Point_3> traits; default constructor.

Operations

<i>Less_xy_2</i>	<i>traits.less_xy_2_object()</i>
<i>Less_yx_2</i>	<i>traits.less_yx_2_object()</i>
<i>Less_signed_distance_to_line_2</i>	<i>traits.less_signed_distance_to_line_2_object()</i>
<i>Less_rotate_ccw_2</i>	<i>traits.less_rotate_ccw_2_object()</i>
<i>Left_turn_2</i>	<i>traits.left_turn_2_object()</i>
<i>Equal_2</i>	<i>traits.equal_2_object()</i>

See Also

CGAL::Convex_hull_constructive_traits_2<R>page [643](#)
CGAL::Convex_hull_projective_xz_traits_2<Point_3>page [645](#)
CGAL::Convex_hull_projective_yz_traits_2<Point_3>page [646](#)
CGAL::Convex_hull_traits_2<R>page [647](#)

CGAL::Convex_hull_projective_xz_traits_2<Point_3>

Definition

The class *Convex_hull_projective_xz_traits_2<Point_3>* serves as a traits class for all the two-dimensional convex hull and extreme point calculation function. This class can be used to compute the convex hull of a set of 3D points projected onto the *xz* plane (i.e., by ignoring the *y* coordinate).

```
#include <CGAL/Convex_hull_projective_xz_traits_2.h>
```

Is Model for the Concepts

ConvexHullTraits_2.....page [641](#)

Types

<i>typedef Point_3</i>	<i>Point_2;</i>
<i>typedef Less_xy_plane_xz_2<Point_3></i>	<i>Less_xy_2;</i>
<i>typedef Less_yx_plane_xz_2<Point_3></i>	<i>Less_yx_2;</i>
<i>typedef Less_dist_to_line_plane_xz_2<Point_3></i>	<i>Less_signed_distance_to_line_2;</i>
<i>typedef Less_rotate_ccw_plane_xz_2<Point_3></i>	<i>Less_rotate_ccw_2;</i>
<i>typedef Left_turn_plane_xz_2<Point_3></i>	<i>Left_turn_2;</i>
<i>typedef Equal_xy_plane_xz_2<Point_3></i>	<i>Equal_2;</i>

Creation

Convex_hull_projective_xz_traits_2<Point_3> traits; default constructor.

Operations

<i>Less_xy_2</i>	<i>traits.less_xy_2_object()</i>
<i>Less_yx_2</i>	<i>traits.less_yx_2_object()</i>
<i>Less_signed_distance_to_line_2</i>	<i>traits.less_signed_distance_to_line_2_object()</i>
<i>Less_rotate_ccw_2</i>	<i>traits.less_rotate_ccw_2_object()</i>
<i>Left_turn_2</i>	<i>traits.left_turn_2_object()</i>
<i>Equal_2</i>	<i>traits.equal_2_object()</i>

See Also

CGAL::Convex_hull_constructive_traits_2<R>page [643](#)
CGAL::Convex_hull_projective_xy_traits_2<Point_3>page [644](#)
CGAL::Convex_hull_projective_yz_traits_2<Point_3>page [646](#)
CGAL::Convex_hull_traits_2<R>page [647](#)

CGAL::Convex_hull_projective_yz_traits_2<Point_3>

Definition

The class *Convex_hull_projective_yz_traits_2<Point_3>* serves as a traits class for all the two-dimensional convex hull and extreme point calculation function. This class can be used to compute the convex hull of a set of 3D points projected onto the *yz* plane (i.e., by ignoring the *x* coordinate).

```
#include <CGAL/Convex_hull_projective_yz_traits_2.h>
```

Is Model for the Concepts

ConvexHullTraits_2.....page [641](#)

Types

<i>typedef Point_3</i>	<i>Point_2;</i>
<i>typedef Less_xy_plane_yz_2<Point_3></i>	<i>Less_xy_2;</i>
<i>typedef Less_yx_plane_yz_2<Point_3></i>	<i>Less_yx_2;</i>
<i>typedef Less_dist_to_line_plane_yz_2<Point_3></i>	<i>Less_signed_distance_to_line_2;</i>
<i>typedef Less_rotate_ccw_plane_yz_2<Point_3></i>	<i>Less_rotate_ccw_2;</i>
<i>typedef Left_turn_plane_yz_2<Point_3></i>	<i>Left_turn_2;</i>
<i>typedef Equal_xy_plane_yz_2<Point_3></i>	<i>Equal_2;</i>

Creation

Convex_hull_projective_yz_traits_2<Point_3> traits; default constructor.

Operations

<i>Less_xy_2</i>	<i>traits.less_xy_2_object()</i>
<i>Less_yx_2</i>	<i>traits.less_yx_2_object()</i>
<i>Less_signed_distance_to_line_2</i>	<i>traits.less_signed_distance_to_line_2_object()</i>
<i>Less_rotate_ccw_2</i>	<i>traits.less_rotate_ccw_2_object()</i>
<i>Left_turn_2</i>	<i>traits.left_turn_2_object()</i>
<i>Equal_2</i>	<i>traits.equal_2_object()</i>

See Also

CGAL::Convex_hull_constructive_traits_2<R>page [643](#)
CGAL::Convex_hull_projective_xz_traits_2<Point_3>page [645](#)
CGAL::Convex_hull_projective_xy_traits_2<Point_3>page [644](#)
CGAL::Convex_hull_traits_2<R>page [647](#)

CGAL::Convex_hull_traits_2<R>

Definition

The class *Convex_hull_traits_2<R>* serves as a traits class for all the two-dimensional convex hull and extreme point calculation function. This class corresponds to the default traits class for these functions.

```
#include <CGAL/convex_hull_traits_2.h>
```

Is Model for the Concepts

ConvexHullTraits_2.....page [641](#)

Types

<i>typedef R::Point_2</i>	<i>Point_2;</i>
<i>typedef R::Less_xy</i>	<i>Less_xy_2;</i>
<i>typedef R::Less_yx</i>	<i>Less_yx_2;</i>
<i>typedef R::Less_signed_distance_to_line_2</i>	<i>Less_signed_distance_to_line_2;</i>
<i>typedef R::Less_rotate_ccw_2</i>	<i>Less_rotate_ccw_2;</i>
<i>typedef R::Left_turn_2</i>	<i>Left_turn_2;</i>
<i>typedef R::Equal_2</i>	<i>Equal_2;</i>

Creation

Convex_hull_traits_2<R> traits(*Convex_hull_traits_2& t*); copy constructor.

Operations

<i>Less_xy_2</i>	<i>traits.less_xy_2_object()</i>
<i>Less_yx_2</i>	<i>traits.less_yx_2_object()</i>
<i>Less_signed_distance_to_line_2</i>	<i>traits.less_signed_distance_to_line_2_object()</i>
<i>Less_rotate_ccw_2</i>	<i>traits.less_rotate_ccw_2_object()</i>
<i>Left_turn_2</i>	<i>traits.left_turn_2_object()</i>
<i>Equal_2</i>	<i>traits.equal_2_object()</i>

See Also

CGAL::Convex_hull_constructive_traits_2<R>page [643](#)
CGAL::Convex_hull_projective_xy_traits_2<Point_3>page [644](#)
CGAL::Convex_hull_projective_xz_traits_2<Point_3>page [645](#)
CGAL::Convex_hull_projective_yz_traits_2<Point_3>page [646](#)

CGAL::is_ccw_strongly_convex_2

Definition

The function *is_ccw_strongly_convex_2* determines if a given sequence of points defines a counterclockwise-oriented, strongly convex polygon. A set of points is said to be strongly convex if it consists of only extreme points (*i.e.*, vertices of the convex hull).

```
#include <CGAL/convexity_check_2.h>
```

```
template <class ForwardIterator, class Traits>
```

```
bool is_ccw_strongly_convex_2( ForwardIterator first,
                               ForwardIterator beyond,
                               Traits ch_traits = Default_traits)
```

returns *true*, iff the point elements in $[first, beyond)$ form a counterclockwise-oriented strongly convex polygon.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

Requirements

Traits contains the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:

- *Traits::Less_xy_2*,
- *Traits::Equal_2*,
- *Traits::Left_turn_2*.

See Also

CGAL::is_cw_strongly_convex_2 page [649](#)

CGAL::is_strongly_convex_3 page [678](#)

Implementation

The algorithm requires $O(n)$ time for a set of n input points.

CGAL::is_cw_strongly_convex_2

Definition

The function *is_cw_strongly_convex_2* determines if a given sequence of points defines a clockwise-oriented, strongly convex polygon. A set of points is said to be strongly convex if it consists of only extreme points (*i.e.*, vertices of the convex hull).

```
#include <CGAL/convexity_check_2.h>
```

```
template <class ForwardIterator, class Traits>
```

```
bool is_cw_strongly_convex_2( ForwardIterator first,
                             ForwardIterator beyond,
                             Traits ch_traits = Default_traits)
```

returns *true*, iff the point elements in $[first, beyond)$ form a clockwise-oriented strongly convex polygon.

The default traits class *Default_traits* is the kernel in which the type *ForwardIterator::value_type* is defined.

Requirements

Traits contains the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:

- *Traits::Equal_2*,
- *Traits::Less_xy_2*,
- *Traits::Left_turn_2*.

See Also

CGAL::is_ccw_strongly_convex_2 page [648](#)

CGAL::is_strongly_convex_3 page [678](#)

Implementation

The algorithm requires $O(n)$ time for a set of n input points.

CGAL::lower_hull_points_2

Definition

The function *lower_hull_points_2* generates the counterclockwise sequence of extreme points on the lower hull of a given set of input points.

```
#include <CGAL/convex_hull_2.h>
```

```
template <class InputIterator, class OutputIterator>
OutputIterator          lower_hull_points_2( InputIterator first,
                                           InputIterator beyond,
                                           OutputIterator result,
                                           Traits ch_traits = Default_traits)
```

generates the counterclockwise sequence of extreme points on the lower hull of the points in the range *[first, beyond)*. The resulting sequence is placed starting at position *result*, and the past-the-end iterator for the resulting sequence is returned. The sequence starts with the leftmost point; the rightmost point is not included. If there is only one extreme point (*i.e.*, leftmost and rightmost point are equal) the extreme point is reported.

Precondition: The source range *[first,beyond)* does not contain *result*.

The default traits class *Default_traits* is the kernel in which the type *InputIterator::value_type* is defined.

The different treatment by *CGAL::upper_hull_points_2* of the case that all points are equal ensures that concatenation of lower and upper hull points gives the sequence of extreme points.

Requirements

1. *InputIterator::value_type* and *OutputIterator::value_type* are equivalent to *Traits::Point_2*.
2. *Traits* contains the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:
 - *Traits::Point_2*,
 - *Traits::Equal_2*,
 - *Traits::Less_xy_2*,
 - *Traits::Less_yx_2*,
 - *Traits::Left_turn_2*.

See Also

CGAL::ch_graham_andrew page [623](#)
CGAL::ch_graham_andrew_scan page [625](#)
CGAL::upper_hull_points_2 page [652](#)

Implementation

This function uses Andrew's variant of Graham's scan algorithm [And79, Meh84]. The algorithm has worst-case running time of $O(n \log n)$ for n input points.

CGAL::upper_hull_points_2

Definition

The function *upper_hull_points_2* generates the counterclockwise sequence of extreme points on the upper hull of a given set of input points.

```
#include <CGAL/convex_hull_2.h>
```

```
template <class InputIterator, class OutputIterator>
OutputIterator      upper_hull_points_2( InputIterator first,
                                         InputIterator beyond,
                                         OutputIterator result,
                                         Traits ch_traits = Default_traits)
```

generates the counterclockwise sequence of extreme points on the upper hull of the points in the range *[first, beyond)*. The resulting sequence is placed starting at position *result*, and the past-the-end iterator for the resulting sequence is returned. The sequence starts with the rightmost point, the leftmost point is not included. If there is only one extreme point (*i.e.*, the leftmost and rightmost point are equal), the extreme point is not reported.

Precondition: The source range *[first,beyond)* does not contain *result*.

The default traits class *Default_traits* is the kernel in which the type *InputIterator::value_type* is defined.

The different treatment by *CGAL::lower_hull_points_2* of the case that all points are equal ensures that concatenation of lower and upper hull points gives the sequence of extreme points.

Requirements

1. *InputIterator::value_type* and *OutputIterator::value_type* are equivalent to *Traits::Point_2*.
2. *Traits* contains the following subset of types from the concept *ConvexHullTraits_2* and their corresponding member functions that return instances of these types:
 - *Traits::Point_2*,
 - *Traits::Equal_2*,
 - *Traits::Less_xy_2*,
 - *Traits::Less_yx_2*,
 - *Traits::Left_turn_2*.

See Also

CGAL::ch_graham_andrew page [623](#)
CGAL::ch_graham_andrew_scan page [625](#)
CGAL::lower_hull_points_2 page [650](#)

Implementation

This function uses Andrew's variant of Graham's scan algorithm [And79, Meh84]. The algorithm has worst-case running time of $O(n \log n)$ for n input points.

Chapter 6

3D Convex Hulls

Contents

6.1	Introduction	655
6.2	Static Convex Hull Construction	655
6.2.1	Traits Class	656
6.2.2	Convexity Checking	656
6.2.3	Example	656
6.3	Incremental Convex Hull Construction	657
6.3.1	Example	657
6.4	Dynamic Convex Hull Construction	659
6.4.1	Example	659

6.1 Introduction

A subset $S \subseteq \mathbb{R}^3$ is convex if for any two points p and q in the set the line segment with endpoints p and q is contained in S . The convex hull of a set S is the smallest convex set containing S . The convex hull of a set of points $P \in \mathbb{R}^3$ is a convex polytope with vertices in P . A point in P is an extreme point (with respect to P) if it is a vertex of the convex hull of P . A set of points is said to be strongly convex if it consists of only extreme points.

This chapter describes the functions provided in CGAL for producing convex hulls in three dimensions as well as functions for checking if sets of points are strongly convex are not. One can compute the convex hull of a set of points in three dimensions in one of three ways in CGAL: using a static algorithm, using an incremental construction algorithm, or using a triangulation to get a fully dynamic computation.

6.2 Static Convex Hull Construction

The function `convex_hull_3` provides an implementation of the quickhull algorithm [BDH96] for three dimensions. There are two versions of this function available, one that can be used when it is known that the output will be a polyhedron (*i.e.*, there are more than three points and they are not all collinear) and one that handles all degenerate cases and returns a `CGAL::Object`, which may be a point, a segment, a triangle, or a polyhedron. Both versions accept a range of input iterators defining the set of points whose convex hull is to be computed and a traits class defining the geometric types and predicates used in computing the hull.

6.2.1 Traits Class

The function `convex_hull_3` is parameterized by a traits class, which specifies the types and geometric primitives to be used in the computation. The default for this traits class is `Convex_hull_traits_3`.

6.2.2 Convexity Checking

The function `is_strongly_convex_3` implements the algorithm of Mehlhorn *et al.* [MNS⁺96] to determine if the vertices of a given polytope constitute a strongly convex point set or not. This function is used in postcondition testing for `convex_hull_3`.

6.2.3 Example

The following program computes the convex hull of a set of 250 random points chosen from a sphere of radius 100. It then determines if the resulting hull is a segment or a polyhedron.

```
// file: examples/Convex_hull_3/ch_quickhull_3_ex.C

#include <CGAL/Homogeneous.h>
#include <CGAL/point_generators_3.h>
#include <CGAL/copy_n.h>
#include <CGAL/Convex_hull_traits_3.h>
#include <CGAL/convex_hull_3.h>
#include <vector>

#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz RT;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float RT;
#endif

typedef CGAL::Homogeneous<RT> K;
typedef CGAL::Convex_hull_traits_3<K> Traits;
typedef Traits::Polyhedron_3 Polyhedron_3;
typedef K::Segment_3 Segment_3;

// define point creator
typedef K::Point_3 Point_3;
typedef CGAL::Creator_uniform_3<double, Point_3> PointCreator;

int main()
{
    CGAL::Random_points_in_sphere_3<Point_3, PointCreator> gen(100.0);

    // generate 250 points randomly on a sphere of radius 100.0
    // and copy them to a vector
    std::vector<Point_3> points;
```



```

CGAL::copy_n( gen, 250, std::back_inserter(points) );

// define object to hold convex hull
CGAL::Object ch_object;

// compute convex hull
CGAL::convex_hull_3(points.begin(), points.end(), ch_object);

// determine what kind of object it is
Segment_3 segment;
Polyhedron_3 polyhedron;
if ( CGAL::assign(segment, ch_object) )
    std::cout << "convex hull is a segment " << std::endl;
else if ( CGAL::assign (polyhedron, ch_object) )
    std::cout << "convex hull is a polyhedron " << std::endl;
else
    std::cout << "convex hull error!" << std::endl;

return 0;
}

```

6.3 Incremental Convex Hull Construction

The function *convex_hull_incremental_3* provides an interface similar to *convex_hull_3* for the d -dimensional incremental construction algorithm [CMS93], implemented by the class *CGAL::Convex_hull_d<R>* that is specialized to three dimensions. This function accepts an iterator range over a set of input points and returns a polyhedron, but it does not have a traits class in its interface. It uses the kernel class *Kernel* used in the polyhedron type to define an instance of the adapter traits class *CGAL::Convex_hull_d_traits_3<Kernel>*.

In most cases, the function *convex_hull_3* will be faster than *convex_hull_incremental_3*. The latter is provided mainly for comparison purposes.

To use the full functionality available with the d -dimensional class *CGAL::Convex_hull_d<R>* in three dimensions (e.g., the ability to insert new points and to query if a point lies in the convex hull or not), you can instantiate the class *CGAL::Convex_hull_d<K>* with the adapter traits class *CGAL::Convex_hull_d_traits_3<K>*, as shown in the following example.

6.3.1 Example

```

// Copyright (c) 2002 Max Planck Institut fuer Informatik (Germany).
// All rights reserved.
//
// This file is part of CGAL (www.cgal.org); you may redistribute it under
// the terms of the Q Public License version 1.0.
// See the file LICENSE.QPL distributed with CGAL.
//
// Licensees holding a valid commercial license may use this file in
// accordance with the commercial license agreement provided with the software.
//
// This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE

```

```

// WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
//
// $URL: svn+ssh://scm.gforge.inria.fr/svn/cgal/branches/CGAL-3.2-branch/Convex_hull_3/demo/Convex_hull_3
// $Id: incremental_hull_3_demo.C 28567 2006-02-16 14:30:13Z lsaboret $
//
//
// Author(s)      : Susan Hert
//

#include <CGAL/Homogeneous.h>
#include <CGAL/point_generators_3.h>
#include <CGAL/Convex_hull_d.h>
#include <CGAL/Convex_hull_d_traits_3.h>
#include <CGAL/Convex_hull_d_to_polyhedron_3.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/copy_n.h>
#include <CGAL/IO/Geomview_stream.h>
#include <CGAL/IO/Polyhedron_geomview_ostream.h>
#include <vector>
#include <cassert>

#ifdef CGAL_USE_GEOMVIEW

#ifdef CGAL_USE_LEDA
#include <CGAL/leda_integer.h>
typedef leda_integer RT;
#else
#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz RT;
#else
// NOTE: the choice of double here for a number type may cause problems
//       for degenerate point sets
#include <CGAL/double.h>
typedef double RT;
#endif
#endif

typedef CGAL::Homogeneous<RT>          K;
typedef K::Point_3                    Point_3;
typedef CGAL::Polyhedron_3< K>        Polyhedron_3;

typedef CGAL::Convex_hull_d_traits_3<K> Hull_traits_3;
typedef CGAL::Convex_hull_d< Hull_traits_3 > Convex_hull_3;
typedef CGAL::Creator_uniform_3<double, Point_3> Creator;

int main ()
{
    Convex_hull_3 CH(3); // create instance of the class with dimension == 3

    // generate 250 points randomly on a sphere of radius 100
    // and insert them into the convex hull

```

```

CGAL::Random_points_in_sphere_3<Point_3, Creator> gen(100);

for (int i = 0; i < 250 ; i++, ++gen)
    CH.insert(*gen);

assert(CH.is_valid());

// define polyhedron to hold convex hull and create it
Polyhedron_3 P;
CGAL::convex_hull_d_to_polyhedron_3(CH,P);

// display polyhedron in a geomview window
CGAL::Geomview_stream geomview;
geomview << CGAL::RED;
geomview << P;

std::cout << "Press any key to end the program: ";
std::cout.flush();
char ch;
std::cin.get(ch);

return 0;
}

#else

int main() {
    std::cerr <<
        "This demo requires geomview, which is not present on this platform\n";
    return 0;
}

#endif

```

6.4 Dynamic Convex Hull Construction

Fully dynamic maintenance of a convex hull can be achieved by using the class *CGAL::Delaunay_triangulation_3*. This class supports insertion and removal of points (*i.e.*, vertices of the triangulation) and the convex hull edges are simply the finite edges of infinite faces. The following example illustrates the dynamic construction of a convex hull. First, random points from a sphere of a certain radius are generated and are inserted into a triangulation. Then the number of points of the convex hull are obtained by counting the number of triangulation vertices incident to the infinite vertex. Some of the points are removed and then the number of points remaining on the hull are determined. Notice that the vertices incident to the infinite vertex of the triangulation are on the convex hull but it may be that not all of them are vertices of the hull.

6.4.1 Example

```

// file: examples/Convex_hull_3/dynamic_hull_3_ex.C

#include <CGAL/Simple_cartesian.h>

```

```

#include <CGAL/Filtered_kernel.h>
#include <CGAL/point_generators_3.h>
#include <CGAL/Delaunay_triangulation_3.h>
#include <CGAL/copy_n.h>

#include <list>

typedef CGAL::Simple_cartesian<double> SK;
typedef CGAL::Filtered_kernel<SK> FK;
struct K : public FK {};

typedef K::Point_3 Point_3;
typedef CGAL::Delaunay_triangulation_3<K> Delaunay;
typedef Delaunay::Vertex_handle Vertex_handle;

int main()
{
    CGAL::Random_points_in_sphere_3<Point_3> gen(100.0);
    std::list<Point_3> points;

    // generate 250 points randomly on a sphere of radius 100.0
    // and insert them into the triangulation
    CGAL::copy_n(gen, 250, std::back_inserter(points) );
    Delaunay T;
    T.insert(points.begin(), points.end());

    std::list<Vertex_handle> vertices;
    T.incident_vertices(T.infinite_vertex(), std::back_inserter(vertices));
    std::cout << "This convex hull of the 250 points has "
              << vertices.size() << " points on it." << std::endl;

    // remove 25 of the input points
    std::list<Vertex_handle>::iterator v_set_it = vertices.begin();
    for (int i = 0; i < 25; i++)
    {
        T.remove(*v_set_it);
        v_set_it++;
    }

    vertices.clear();
    T.incident_vertices(T.infinite_vertex(), std::back_inserter(vertices));
    std::cout << "After removal of 25 points, there are "
              << vertices.size() << " points on the convex hull." << std::endl;
    return 0;
}

```

3D Convex Hulls

Reference Manual

Susan Hert and Stefan Schirra

A subset $S \subseteq \mathbb{R}^3$ is convex if for any two points p and q in the set the line segment with endpoints p and q is contained in S . The convex hull of a set S is the smallest convex set containing S . The convex hull of a set of points P is a convex polytope with vertices in P . A point in P is an extreme point (with respect to P) if it is a vertex of the convex hull of P .

CGAL provides functions for computing convex hulls in two, three and arbitrary dimensions as well as functions for testing if a given set of points is strongly convex or not. This chapter describes the functions available for three dimensions.

Assertions

The assertion flags for the convex hull and extreme point algorithms use *CH* in their names (e.g., *CGAL_CH_NO_POSTCONDITIONS*). For the convex hull algorithms, the postcondition check tests only convexity (if not disabled), but not containment of the input points in the polygon or polyhedron defined by the output points. The latter is considered an expensive checking and can be enabled by defining *CGAL_CH_CHECK_EXPENSIVE*.

6.5 Classified Reference Pages

Concepts

ConvexHullPolyhedron_3	page 672
ConvexHullPolyhedronFacet_3	page 669
ConvexHullPolyhedronHalfedge_3	page 670
ConvexHullPolyhedronVertex_3	page 671
ConvexHullTraits_3	page 674
IsStronglyConvexTraits_3	page 679

Traits Classes

<i>CGAL::Convex_hull_traits_3<R></i>	page 676
--------------------------------------------------	--------------------------

Convex Hull Functions

<i>CGAL::convex_hull_3</i>	page 664
<i>CGAL::convex_hull_incremental_3</i>	page 667

Convexity Checking Function

<i>CGAL::is_strongly_convex_3</i>	page 678
-----------------------------------------	--------------------------

6.6 Alphabetical List of Reference Pages

<i>ConvexHullPolyhedronFacet_3</i>	page 669
<i>ConvexHullPolyhedronHalfedge_3</i>	page 670
<i>ConvexHullPolyhedronVertex_3</i>	page 671
<i>ConvexHullPolyhedron_3</i>	page 672
<i>ConvexHullTraits_3</i>	page 674
<i>convex_hull_3</i>	page 664
<i>convex_hull_incremental_3</i>	page 667
<i>Convex_hull_traits_3<R></i>	page 676
<i>IsStronglyConvexTraits_3</i>	page 679
<i>is_strongly_convex_3</i>	page 678

CGAL::convex_hull_3

Definition

The function *convex_hull_3* computes the convex hull of a given set of three-dimensional points. Two versions of this function are available. The first can be used when it is known that the result will be a polyhedron and the second when a degenerate hull may also be possible.

```
#include <CGAL/convex_hull_3.h>
```

```
template <class InputIterator, class Polyhedron_3, class Traits>
void convex_hull_3( InputIterator first,
                   InputIterator last,
                   Polyhedron_3& P,
                   Traits ch_traits = Default_traits)
```

computes the convex hull of the set of points in the range $[first, last)$. The polyhedron P is cleared, then the convex hull is stored in P and the plane equations of each face are computed.

Precondition: : There are at least four points in the range $[first, last)$ not all of which are collinear.

```
template <class InputIterator, class Polyhedron_3, class Traits>
void convex_hull_3( InputIterator first,
                   InputIterator last,
                   Object& ch_object,
                   Traits ch_traits = Default_traits)
```

computes the convex hull of the set of points in the range $[first, last)$. The result, which may be a point, a segment, a triangle, or a polyhedron, is stored in *ch_object*. When the result is a polyhedron, the plane equations of each face are computed.

Requirements

Both functions require the following:

1. *InputIterator::value_type* is equivalent to *Traits::Point_3*.
2. *Traits* is a model of the concept *ConvexHullTraits_3*. When it is known that the input points are not all coplanar, the types *Traits_xy*, *Traits_yx*, and *Traits_yz* need not be provided. For the purposes of checking the postcondition that the convex hull is valid, *Traits* should also be a model of the concept *IsStronglyConvexTraits_3*.

Both functions have an additional requirement for the polyhedron that is to be constructed. For the first version this is that:

- *Polyhedron_3* is a model of `ConvexHullPolyhedron_3`,

and for the second, it is required that

- *Traits* defines a type *Polyhedron_3* that is a model of `ConvexHullPolyhedron_3`.

The default traits class for both versions of *convex_hull_3* is *Convex_hull_traits_3<R>*, with the representation *R* determined by *InputIterator::value_type*.

See Also

CGAL::convex_hull_incremental_3 page [667](#)
CGAL::ch_eddy page [620](#)
CGAL::convex_hull_2 page [639](#)

Implementation

The algorithm implemented by these functions is the quickhull algorithm of Barnard *et al.* [[BDH96](#)].

Example

The following program computes the convex hull of a set of 250 random points chosen from a sphere of radius 100. It then determines if the resulting hull is a segment or a polyhedron. Notice that the traits class is not necessary in the call to *convex_hull_3* but is used in the definition of *Polyhedron_3*.

```
// file: examples/Convex_hull_3/ch_quickhull_3_ex.C

#include <CGAL/Homogeneous.h>
#include <CGAL/point_generators_3.h>
#include <CGAL/copy_n.h>
#include <CGAL/Convex_hull_traits_3.h>
#include <CGAL/convex_hull_3.h>
#include <vector>

#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz RT;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float RT;
#endif

typedef CGAL::Homogeneous<RT> K;
typedef CGAL::Convex_hull_traits_3<K> Traits;
typedef Traits::Polyhedron_3 Polyhedron_3;
typedef K::Segment_3 Segment_3;

// define point creator
typedef K::Point_3 Point_3;
typedef CGAL::Creator_uniform_3<double, Point_3> PointCreator;
```

```

int main()
{
    CGAL::Random_points_in_sphere_3<Point_3, PointCreator> gen(100.0);

    // generate 250 points randomly on a sphere of radius 100.0
    // and copy them to a vector
    std::vector<Point_3> points;
    CGAL::copy_n( gen, 250, std::back_inserter(points) );

    // define object to hold convex hull
    CGAL::Object ch_object;

    // compute convex hull
    CGAL::convex_hull_3(points.begin(), points.end(), ch_object);

    // determine what kind of object it is
    Segment_3 segment;
    Polyhedron_3 polyhedron;
    if ( CGAL::assign(segment, ch_object) )
        std::cout << "convex hull is a segment " << std::endl;
    else if ( CGAL::assign( polyhedron, ch_object) )
        std::cout << "convex hull is a polyhedron " << std::endl;
    else
        std::cout << "convex hull error!" << std::endl;

    return 0;
}

```

CGAL::convex_hull_incremental_3

Definition

The function *convex_hull_incremental_3* computes the convex hull polyhedron from a set of given three-dimensional points.

```
#include <CGAL/convex_hull_incremental_3.h>
```

```
template <class InputIterator, class Polyhedron>
void convex_hull_incremental_3( InputIterator first,
                               InputIterator beyond,
                               Polyhedron& P,
                               bool test_correctness = false)
```

computes the convex hull polyhedron of the points in the range $[first, beyond)$ and assigns it to P . If *test_correctness* is set to *true*, the tests described in [MNS⁺96] are used to determine the correctness of the resulting polyhedron.

Requirements

1. *Polyhedron* must provide a type *Polyhedron::Traits* that defines the following types
 - *Polyhedron::Traits::R*, which is a model of the representation class R required by *CGAL::Convex_hull_d_traits_3<R>*
 - *Polyhedron::Traits::Point*
2. *InputIterator::value_type* must be the same as *Polyhedron::Traits::Point*

See Also

CGAL::convex_hull_3 page [664](#)
CGAL::convex_hull_2 page [639](#)

Implementation

This function uses the d -dimensional convex hull incremental construction algorithm [CMS93] with d fixed to 3. The algorithm requires $O(n^2)$ time in the worst case and $O(n \log n)$ expected time.

See Also

CGAL::Convex_hull_d<R> page [687](#)

Example

The following example computes the convex hull of a set of 250 random points chosen uniformly in a sphere of radius 100.

```
// file: examples/Convex_hull_3/incremental_hull_3_ex.C

#include <CGAL/Homogeneous.h>
#include <CGAL/point_generators_3.h>
#include <CGAL/copy_n.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/convex_hull_incremental_3.h>
#include <vector>

#ifdef CGAL_USE_GMP
#include <CGAL/Gmpz.h>
typedef CGAL::Gmpz RT;
#else
#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float RT;
#endif

typedef CGAL::Homogeneous<RT> K;
typedef K::Point_3 Point_3;
typedef CGAL::Polyhedron_3< K> Polyhedron;
typedef CGAL::Creator_uniform_3<int, Point_3> Creator;

int main()
{
    CGAL::Random_points_in_sphere_3<Point_3, Creator> gen(100.0);

    std::vector<Point_3> V;
    // generate 250 points randomly on a sphere of radius 100.0 and copy
    // them to a vector
    CGAL::copy_n( gen, 250, std::back_inserter(V) );

    Polyhedron P; // define polyhedron to hold convex hull

    // compute convex hull
    CGAL::convex_hull_incremental_3( V.begin(), V.end(), P, true);

    return 0;
}
```

ConvexHullPolyhedronFacet_3

Definition

The requirements of the facet type of a polyhedron to be built by the function *convex_hull_3*.

Has Models

CGAL::Polyhedron_3<Traits>::Facetpage [811](#)

Types

ConvexHullPolyhedronFacet_3::Plane plane equation type stored in facets.

ConvexHullPolyhedronFacet_3::Halfedge_handle
handle to halfedge.

ConvexHullPolyhedronFacet_3::Halfedge_around_facet_circulator
circulator of halfedges around a facet.

Creation

ConvexHullPolyhedronFacet_3 f; default constructor.

Operations

Plane& *f.plane()*
const Plane& *f.plane() const* plane equation.

Halfedge_handle *f.halfedge()* an incident halfedge that points to *f*.
Halfedge_around_facet_circulator

f.facet_begin() circulator of halfedges around the facet (counterclockwise).

See Also

CGAL::Polyhedron_3<Traits> page [799](#)
ConvexHullPolyhedronVertex_3 page [671](#)
ConvexHullPolyhedronHalfedge_3 page [670](#)

ConvexHullPolyhedronHalfedge_3

Definition

The requirements of the halfedge type required for the polyhedron built by the function *convex_hull_3*.

Has Models

CGAL::Polyhedron_3<Traits>::Halfedge.....page 813

Creation

ConvexHullPolyhedronHalfedge_3 *h*; default constructor.

Operations

<i>Halfedge_handle</i>	<i>h.opposite()</i>	the opposite halfedge.
------------------------	---------------------	------------------------

<i>Halfedge_handle</i>	<i>h.next()</i>	the next halfedge around the facet.
------------------------	-----------------	-------------------------------------

<i>Halfedge_handle</i>	<i>h.prev()</i>	the previous halfedge around the facet.
------------------------	-----------------	-----------------------------------------

<i>bool</i>	<i>h.is_border()</i> <i>const</i>	is true if <i>h</i> is a border halfedge.
-------------	-----------------------------------	-------------------------------------------

<i>Halfedge_around_facet_circulator</i>	
<i>h.facet_begin()</i>	circulator of halfedges around the facet (counterclockwise).

<i>Vertex_handle</i>	<i>h.vertex()</i>	the incident vertex of <i>h</i> .
----------------------	-------------------	-----------------------------------

<i>Facet_handle</i>	<i>h.facet()</i>	the incident facet of <i>h</i> . If <i>h</i> is a border halfedge the result is default construction of the handle.
---------------------	------------------	---------------------------------------------------------------------------------------------------------------------

See Also

CGAL::Polyhedron_3<Traits> page 799

ConvexHullPolyhedronVertex_3 page 671

ConvexHullPolyhedronFacet_3.....page 669

ConvexHullPolyhedronVertex_3

Definition

The requirements of the vertex type of the polyhedron to be built by the function *convex_hull_3*.

Has Models

CGAL::Polyhedron_3<Traits>::Vertex page [816](#).

Creation

ConvexHullPolyhedronVertex_3 *v*; default constructor.

Operations

<i>Point&</i>	<i>v.point()</i>	
<i>const Point&</i>	<i>v.point() const</i>	the point.

See Also

CGAL::Polyhedron_3<Traits> page [799](#)
ConvexHullPolyhedronFacet_3 page [669](#)
ConvexHullPolyhedronHalfedge_3 page [670](#)

ConvexHullPolyhedron_3

Definition

The requirements of the polyhedron type to be built by the function *convex_hull_3*.

Has Models

CGAL::Polyhedron_3<Traits> page [799](#)

Types

<i>ConvexHullPolyhedron_3:: Point_3</i>	type of point stored in a vertex
<i>ConvexHullPolyhedron_3:: Vertex</i>	a model of <i>ConvexHullPolyhedronVertex_3</i>
<i>ConvexHullPolyhedron_3:: Halfedge</i>	a model of <i>ConvexHullPolyhedronHalfedge_3</i>
<i>ConvexHullPolyhedron_3:: Facet</i>	a model of <i>ConvexHullPolyhedronFacet_3</i>
<i>ConvexHullPolyhedron_3:: Halfedge_data_structure</i>	halfedge data structure
<i>ConvexHullPolyhedron_3:: Halfedge_handle</i>	handle to halfedge
<i>ConvexHullPolyhedron_3:: Halfedge_iterator</i>	iterator for halfedge
<i>ConvexHullPolyhedron_3:: Facet_handle</i>	handle to facet
<i>ConvexHullPolyhedron_3:: Facet_iterator</i>	iterator for facet

Creation

Only a default constructor is required.

ConvexHullPolyhedron_3 *p*;

Operations

<i>Facet_iterator</i>	<i>p.facets_begin()</i>	iterator over all facets (excluding holes).
<i>Facet_iterator</i>	<i>p.facets_end()</i>	past-the-end iterator.
<i>Halfedge_iterator</i>	<i>p.P.halfedges_begin()</i>	iterator over all halfedges.
<i>Halfedge_iterator</i>	<i>p.P.halfedges_end()</i>	past-the-end iterator.
<i>Halfedge_handle</i>	<i>p.make_tetrahedron(Point_3 p1, Point_3 p2, Point_3 p3, Point_3 p4)</i>	

adds a new tetrahedron to the polyhedral surface with its vertices initialized with *p1*, *p2*, *p3* and *p4*. Returns that halfedge of the tetrahedron which incident vertex is initialized with *p1*, the incident vertex of the next halfedge with *p2*, and the vertex thereafter with *p3*. The remaining fourth vertex is initialized with *p4*.

<i>void</i>	<i>p.erase_facet(Halfedge_handle h)</i>	removes the incident facet of <i>h</i> and changes all halfedges incident to the facet into border edges or removes them from the polyhedral surface if they were already border edges.
<i>Halfedge_handle</i>	<i>p.add_vertex_and_facet_to_border(Halfedge_handle h, Halfedge_handle g)</i>	creates a new facet within the hole incident to <i>h</i> and <i>g</i> by connecting the tip of <i>g</i> with the tip of <i>h</i> with two new halfedges and a new vertex and filling this separated part of the hole with a new facet, such that the new facet is incident to <i>g</i> . Returns the halfedge of the new edge that is incident to the new facet and the new vertex.
<i>Halfedge_handle</i>	<i>p.add_facet_to_border(Halfedge_handle h, Halfedge_handle g)</i>	creates a new facet within the hole incident to <i>h</i> and <i>g</i> by connecting the tip of <i>g</i> with the tip of <i>h</i> with a new halfedge and filling this separated part of the hole with a new facet, such that the new facet is incident to <i>g</i> . Returns the halfedge of the new edge that is incident to the new facet.
<i>Halfedge_handle</i>	<i>p.fill_hole(Halfedge_handle h)</i>	fills a hole with a newly created facet. Makes all border halfedges of the hole denoted by <i>h</i> incident to the new facet. Returns <i>h</i> .
<i>void</i>	<i>p.delegate(Modifier_base<Halfedge_data_structure>& m)</i>	calls the <i>operator()</i> of the modifier <i>m</i> . See <i>Modifier_base</i> in the Support Library Manual for a description of modifier design and its usage.

ConvexHullTraits_3

Definition

Requirements of the traits class to be used with the function *convex_hull_3*.

Types

<i>ConvexHullTraits_3:: Point_3</i>	The point type on which the convex hull algorithm operates
<i>ConvexHullTraits_3:: Plane_3</i>	a 3D plane
<i>ConvexHullTraits_3:: Segment_3</i>	a 3D segment
<i>ConvexHullTraits_3:: Triangle_3</i>	a 3D triangle
<i>ConvexHullTraits_3:: Vector_3</i>	a 3D vector
<i>ConvexHullTraits_3:: Construct_plane_3</i>	Function object type that provides <i>Plane_3 operator()(Point_3 p, Point_3 q, Point_3 r)</i> , which constructs and returns a plane passing through <i>p</i> , <i>q</i> , and <i>r</i> and oriented in a positive sense when seen from the positive side of the plane.
<i>ConvexHullTraits_3:: Construct_segment_3</i>	Function object type that provides <i>Segment_3 operator()(Point_3 p, Point_3 q)</i> , which constructs and returns the segment with source <i>p</i> and target <i>q</i> .
<i>ConvexHullTraits_3:: Construct_triangle_3</i>	Function object type that provides <i>Triangle_3 operator()(Point_3 p, Point_3 q, Point_3 r)</i> , which constructs and returns the triangle with vertices <i>p</i> , <i>q</i> , and <i>r</i>
<i>ConvexHullTraits_3:: Construct_vector_3</i>	Function object type that provides <i>Vector_3 operator()(Point_3 p, Point_3 q)</i> , which constructs and returns the vector <i>q-p</i>
<i>ConvexHullTraits_3:: Equal_3</i>	Predicate object type that provides <i>bool operator()(Point_3 p, Point_3 q)</i> , which determines if points <i>p</i> and <i>q</i> are equal or not
<i>ConvexHullTraits_3:: Collinear_3</i>	Predicate object type that provides <i>bool operator()(Point_3 p, Point_3 q, Point_3 r)</i> , which determines if points <i>p</i> , <i>q</i> and <i>r</i> are collinear or not
<i>ConvexHullTraits_3:: Coplanar_3</i>	Predicate object type that provides <i>bool operator()(Point_3 p, Point_3 q, Point_3 r, Point_3 s)</i> , which determines if points <i>p</i> , <i>q</i> , <i>r</i> , and <i>s</i> are coplanar or not
<i>ConvexHullTraits_3:: Has_on_positive_side_3</i>	Predicate object type that provides <i>bool operator()(Plane_3 h, Point_3 q)</i> , which determines if the point <i>q</i> is on the positive side of the halfspace <i>h</i>

<i>ConvexHullTraits_3:: Less_distance_to_point_3</i>	Predicate object type that provides a constructor taking a single <i>Point_3</i> object and <i>bool operator()(Point_3 q, Point_3 r)</i> , which returns true iff the distance from <i>q</i> to <i>p</i> is smaller than the distance from <i>r</i> to <i>p</i> , where <i>p</i> is the point passed to the object at construction.
------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<i>ConvexHullTraits_3:: Less_signed_distance_to_plane_3</i>	Predicate object type that provides <i>bool operator()(Plane_3 p, Point_3 q, Point_3 r)</i> , which returns true iff the signed distance from <i>q</i> to <i>p</i> is smaller than the signed distance from <i>r</i> to <i>p</i>
-------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

To handle the degenerate case when all points are coplanar, the following three types that are default-constructable are necessary:

<i>ConvexHullTraits_3:: Traits_xy</i>	A model of <i>ConvexHullTraits_2</i> for points projected into the <i>xy</i> -plane
---------------------------------------	-------------------------------------------------------------------------------------

<i>ConvexHullTraits_3:: Traits_xz</i>	A model of <i>ConvexHullTraits_2</i> for points projected into the <i>xz</i> -plane
---------------------------------------	-------------------------------------------------------------------------------------

<i>ConvexHullTraits_3:: Traits_yz</i>	A model of <i>ConvexHullTraits_2</i> for points projected into the <i>yz</i> -plane
---------------------------------------	-------------------------------------------------------------------------------------

One also needs the following function object to help choose which of the above traits classes to use:

<i>ConvexHullTraits_3:: Max_coordinate_3</i>	Function object type that provides <i>int operator()(Vector_3 v)</i> , which returns the index (0, 1, or 2 for <i>x</i> , <i>y</i> , or <i>z</i> , respectively) of the coordinate of <i>v</i> with maximum absolute value.
----------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

These types need not be provided when it is known that the points are not all coplanar.

Creation

Only a copy constructor is required.

```
ConvexHullTraits_3 traits( & ch);
```

Operations

For each of the above function and predicate object types, *Func_obj_type*, a function must exist with the name *func_obj_type_object* that creates an instance of the function or predicate object type. For example:

```
Construct_plane_3 traits.construct_plane_3_object()
```

Has Models

CGAL::Convex_hull_traits_3<R> page [676](#)

CGAL::Convex_hull_traits_3<R>

Definition

The class *Convex_hull_traits_3<R>* serves as a traits class for the function *convex_hull_3*. This is the default traits class for this function.

```
#include <CGAL/Convex_hull_traits_3.h>
```

Is Model for the Concepts

ConvexHullTraits_3 page [674](#)
IsStronglyConvexTraits_3 page [679](#)

Types

<i>typedef R::Point_3</i>	<i>Point_3;</i>
<i>typedef R::Segment_3</i>	<i>Segment_3;</i>
<i>typedef R::Triangle_3</i>	<i>Triangle_3;</i>
<i>typedef R::Plane_3</i>	<i>Plane_3;</i>
<i>typedef R::Vector_3</i>	<i>Vector_3;</i>
<i>typedef Polyhedron_default_traits_3<R></i>	<i>Poly_traits;</i>
<i>typedef Halfedge_data_structure_polyhedron_default_3<R></i>	<i>HDS;</i>
<i>typedef Polyhedron_3<Poly_traits, HDS></i>	<i>Polyhedron_3;</i>
<i>typedef R::Construct_segment_3</i>	<i>Construct_segment_3;</i>
<i>typedef R::Construct_ray_3</i>	<i>Construct_ray_3;</i>
<i>typedef R::Construct_plane_3</i>	<i>Construct_plane_3;</i>
<i>typedef R::Construct_vector_3</i>	<i>Construct_vector_3;</i>
<i>typedef R::Construct_triangle_3</i>	<i>Construct_triangle_3;</i>
<i>typedef R::Construct_centroid_3</i>	<i>Construct_centroid_3;</i>
<i>typedef R::Construct_orthogonal_vector_3</i>	<i>Construct_orthogonal_vector_3;</i>
<i>R::Equal_3</i>	<i>Equal_3;</i>
<i>R::Collinear_3</i>	<i>Collinear_3;</i>
<i>R::Coplanar_3</i>	<i>Coplanar_3;</i>
<i>R::Less_distance_to_point_3</i>	<i>Less_distance_to_point_3;</i>
<i>R::Has_on_positive_side_3</i>	<i>Has_on_positive_side_3;</i>
<i>R::Less_signed_dist_to_plane_3</i>	<i>Less_signed_distance_to_plane_3;</i>
<i>Convex_hull_projective_xy_traits_2<Point_3></i>	<i>Traits_xy;</i>
<i>Convex_hull_projective_xz_traits_2<Point_3></i>	<i>Traits_xz;</i>
<i>Convex_hull_projective_yz_traits_2<Point_3></i>	<i>Traits_yz;</i>
<i>R::Ray_3</i>	<i>Ray_3;</i>
<i>R::Has_on_3</i>	<i>Has_on_3;</i>
<i>R::Oriented_side_3</i>	<i>Oriented_side_3;</i>
<i>R::Do_intersect_3</i>	<i>Do_intersect_3;</i>

Creation

Convex_hull_traits_3<R> traits(*Convex_hull_traits_2& t*); copy constructor.

Operations

<i>Construct_segment_3</i>	<i>traits.construct_segment_3_object()</i>
<i>Construct_ray_3</i>	<i>traits.construct_ray_3_object()</i>
<i>Construct_plane_3</i>	<i>traits.construct_plane_3_object()</i>
<i>Construct_triangle_3</i>	<i>traits.construct_triangle_3_object()</i>
<i>Construct_vector_3</i>	<i>traits.construct_vector_3_object()</i>
<i>Construct_centroid_3</i>	<i>traits.construct_centroid_3_object()</i>
<i>Construct_orthogonal_vector_3</i>	<i>traits.construct_orthogonal_vector_3_object()</i>
<i>Equal_3</i>	<i>traits.equal_3_object()</i>
<i>Collinear_3</i>	<i>traits.collinear_3_object()</i>
<i>Coplanar_3</i>	<i>traits.coplanar_3_object()</i>
<i>Has_on_3</i>	<i>traits.has_on_3_object()</i>
<i>Less_distance_to_point_3</i>	<i>traits.less_distance_to_point_3_object()</i>
<i>Has_on_positive_side_3</i>	<i>traits.has_on_positive_side_3_object()</i>
<i>Oriented_side_3</i>	<i>traits.oriented_side_3_object()</i>
<i>Do_intersect_3</i>	<i>traits.do_intersect_3_object()</i>
<i>Less_signed_distance_to_plane_3</i>	<i>traits.less_signed_distance_to_plane_3_object()</i>

See Also

CGAL::convex_hull_2 [page 639](#)

CGAL::is_strongly_convex_3

Definition

The function *is_strongly_convex_3* determines if the vertices of a given polyhedron represents a strongly convex set of points or not. A set of points is said to be strongly convex if it consists of only extreme points (*i.e.*, vertices of the convex hull).

```
#include <CGAL/convexity_check_3.h>
```

```
template<class Polyhedron_3, class Traits>
bool is_strongly_convex_3( Polyhedron_3& P, Traits traits = Default_traits)
```

determines if the set of vertices of the polyhedron *P* represent a strongly convex set of points or not.

Precondition: The equations of the facet planes of the polyhedron must have already been computed.

The default traits class is the kernel in which the type *Polyhedron_3::Point_3* is defined.

Requirements

1. *Polyhedron_3::Point_3* is equivalent to *Traits::Point_3*.
2. *Traits* is a model of the concept *IsStronglyConvexTraits_3*
3. *Polyhedron_3* must define the following types:

- *Polyhedron_3::Facet_iterator*
- *Polyhedron_3::Vertex_iterator*

and the following member functions:

- *facets_begin()*
- *facets_end()*
- *vertices_begin()*
- *vertices_end()*

The vertex type of *Polyhedron_3* must be a model of *ConvexHullPolyhedronVertex_3*; the facet type must be *ConvexHullPolyhedronFacet_3*.

See Also

CGAL::is_ccw_strongly_convex_2 page 648
CGAL::is_cw_strongly_convex_2 page 649

Implementation

This function implements the tests described in [MNS⁺96] to determine convexity and requires $O(e + f)$ time for a polyhedron with e edges and f faces.

IsStronglyConvexTraits_3

Definition

Requirements of the traits class used by the function *is_strongly_convex_3*, which is used for postcondition checking by *convex_hull_3*.

Types

<i>IsStronglyConvexTraits_3:: Plane_3</i>	a 3D plane
<i>IsStronglyConvexTraits_3:: Point_3</i>	a 3D point
<i>IsStronglyConvexTraits_3:: Ray_3</i>	a 3D ray
<i>IsStronglyConvexTraits_3:: Triangle_3</i>	a 3D triangle
<i>IsStronglyConvexTraits_3:: Construct_ray_3</i>	Function object type that provides <i>Ray_3 operator()(Point_3 p, Point_3 q)</i> , which constructs and returns the ray based at point <i>p</i> and passing through <i>q</i>
<i>IsStronglyConvexTraits_3:: Construct_triangle_3</i>	Function object type that provides <i>Triangle_3 operator()(Point_3 p, Point_3 q, Point_3 r)</i> , which constructs and returns the triangle with vertices <i>p</i> , <i>q</i> , and <i>r</i>
<i>IsStronglyConvexTraits_3:: Coplanar_3</i>	Predicate object type that provides <i>bool operator()(Point_3 p, Point_3 q, Point_3 r, Point_3 s)</i> , which determines if points <i>p</i> , <i>q</i> , <i>r</i> , and <i>s</i> are coplanar or not
<i>IsStronglyConvexTraits_3:: Do_intersect_3</i>	Function object type that provides <i>CGAL::Object operator()(Triangle_3 t, Ray_3 r)</i> , which returns <i>true</i> iff <i>t</i> and <i>r</i> intersect.
<i>IsStronglyConvexTraits_3:: Has_on_positive_side_3</i>	Predicate object type that provides <i>bool operator()(Plane_3 h, Point_3 q)</i> , which determines if the point <i>q</i> is on the positive side of the halfspace <i>h</i>
<i>IsStronglyConvexTraits_3:: Oriented_side_3</i>	Predicate object type that provides <i>Oriented_side operator()(Plane_3 p, Point_3 q)</i> , which determines the position of point <i>q</i> relative to plane <i>p</i>

Creation

Only a copy constructor is required.

```
IsStronglyConvexTraits_3 traits( & t);
```

Operations

For each of the above function and predicate object types, *Func_obj_type*, a function must exist with the name *func_obj_type_object* that creates an instance of the function or predicate object type. For example:

Construct_ray_3 *traits.construct_ray_3_object()*

Has Models

CGAL::Convex_hull_traits_3<R> page [676](#)

See Also

ConvexHullTraits_3 page [674](#)

CGAL::is_strongly_convex_3 page [678](#)

Chapter 7

dD Convex Hulls and Delaunay Triangulations

Susan Hert and Michael Seel

Contents

7.1 Introduction	681
7.2 dD Convex Hull	681
7.3 Delaunay Triangulation	682

7.1 Introduction

A subset $S \subseteq \mathbb{R}^d$ is convex if for any two points p and q in the set the line segment with endpoints p and q is contained in S . The convex hull of a set S is the smallest convex set containing S . The convex hull of a set of points P is a convex polytope with vertices in P . A point in P is an extreme point (with respect to P) if it is a vertex of the convex hull of P . A set of points is said to be strongly convex if it consist of only extreme points.

This chapter describes the class provided in CGAL for producing convex hull in arbitrary dimensions. There is an intimate relationship between the Delaunay triangulation of a point set S and the convex hull of $lift(S)$: The nearest site Delaunay triangulation is the projection of the lower hull and the furthest site Delaunay triangulation is the upper hull. Here we also describe the companion class to the convex hull class that computes nearest and furthest site Delaunay triangulations.

7.2 dD Convex Hull

The class `CGAL::Convex_hull_d<R>` is used to represent the convex hull of a set of points in d -dimensional space. This class supports incremental construction of hulls, and provides a rich interface for exploration. There are also output routines for hulls of dimension 2 and 3.

The convex hull class is parameterized by a traits class that provides d -dimensional data types and predicates. The class `Convex_hull_d_traits_3` adapts any low-dimensional standard kernel model *e.g.*, `Homogeneous<RT>` or `Cartesian<FT>` for use with `Convex_hull_d`, where the dimension is fixed to three. The validity of the

computed convex hull can be checked using the member function *is_valid*, which implements the algorithm of Mehlhorn *et al.* [MNS⁺96] to determine if the vertices of a given polytope constitute a strongly convex point set or not.

The implementation follows the papers [CMS93] and [BMS94].

7.3 Delaunay Triangulation

There is a class type with a thorough interface providing the construction and exploration of closest and furthest site Delaunay simplicial complexes in arbitrary higher dimension. The class *CGAL::Delaunay_d<R, Lifted_R>* provides an implementation via the lifting map to higher dimensional convex hulls.. The class supports incremental construction of Delaunay triangulations and various kind of query operations (in particular, nearest and furthest neighbor queries and range queries with spheres and simplices).

dD Convex Hulls and Delaunay Triangulations Reference Manual

Susan Hert and Michael Seel

A subset $S \subseteq \mathbb{R}^3$ is convex if for any two points p and q in the set the line segment with endpoints p and q is contained in S . The convex hull of a set S is the smallest convex set containing S . The convex hull of a set of points P is a convex polytope with vertices in P . A point in P is an extreme point (with respect to P) if it is a vertex of the convex hull of P .

CGAL provides functions for computing convex hulls in two, three and arbitrary dimensions as well as functions for testing if a given set of points is strongly convex or not. This chapter describes the class available for arbitrary dimensions and its companion class for computing the nearest and furthest side Delaunay triangulation.

7.4 Classified Reference Pages

Concepts

ConvexHullTraits_d	page 685
DelaunayLiftedTraits_d	page 695
DelaunayTraits_d	page 698

Classes

CGAL::Convex_hull_d_traits_3<R>	page 694
CGAL::Convex_hull_d<R>	page 687
CGAL::Delaunay_d< R, Lifted_R >	page 700

7.5 Alphabetical List of Reference Pages

<i>ConvexHullTraits_d</i>	page 685
<i>Convex_hull_d</i> < <i>R</i> >	page 687
<i>Convex_hull_d_traits_3</i> < <i>R</i> >	page 694
<i>DelaunayLiftedTraits_d</i>	page 695
<i>DelaunayTraits_d</i>	page 698
<i>Delaunay_d</i> < <i>R</i> , <i>Lifted_R</i> >	page 700

ConvexHullTraits_d

Definition

Requirements of the traits class to be used with the class *Convex_hull_d*.

Types

<i>ConvexHullTraits_d:: Point_d</i>	the dD point type on which the convex hull algorithm operates
<i>ConvexHullTraits_d:: Hyperplane_d</i>	a dD plane
<i>ConvexHullTraits_d:: Vector_d</i>	a dD vector
<i>ConvexHullTraits_d:: Ray_d</i>	a dD ray
<i>ConvexHullTraits_d:: RT</i>	an arithmetic ring type
<i>ConvexHullTraits_d:: Construct_vector_d</i>	Function object type that provides <i>Vector_d operator()(int d, CGAL::Null_vector)</i> , which constructs and returns the null vector.
<i>ConvexHullTraits_d:: Construct_hyperplane_d</i>	Function object type that provides <i>Hyperplane_d operator()(ForwardIterator first, ForwardIterator last, Point_d p, CGAL::Oriented_side side)</i> , which constructs and returns a hyperplane passing through the points in <i>tuple[first,last)</i> and oriented such that <i>p</i> is on the side <i>side</i> of the returned hyperplane. When <i>side==ON_ORIENTED_BOUNDARY</i> then any hyperplane containing the tuple is returned.
<i>ConvexHullTraits_d:: Vector_to_point_d</i>	Function object type that provides <i>Point_d operator()(Vector_d v)</i> , which constructs and returns the point defined by $0 + v$.
<i>ConvexHullTraits_d:: Point_to_vector_d</i>	Function object type that provides <i>Vector_d operator()(Point_d v)</i> , which constructs and returns the vector defined by $p - 0$.
<i>ConvexHullTraits_d:: Orientation_d</i>	Function object type that provides <i>Orientation operator()(ForwardIterator first, ForwardIterator last)</i> , which determines the orientation of the points <i>tuple[first,last)</i> .
<i>ConvexHullTraits_d:: Orthogonal_vector_d</i>	Function object type that provides <i>Vector_d operator()(Hyperplane_d h)</i> , which constructs and returns a vector orthogonal to <i>h</i> and pointing from the boundary into its positive halfspace.
<i>ConvexHullTraits_d:: Oriented_side_d</i>	Predicate object type that provides <i>Oriented_side operator()(Hyperplane_d h, Point_d p)</i> , which determines the oriented side of <i>p</i> with respect to <i>h</i> .

<i>ConvexHullTraits_d:: Has_on_positive_side_d</i>	Predicate object type that provides <i>bool operator()(Hyperplane_d h, Point_d p)</i> , which return true iff <i>p</i> lies in the positive halfspace determined by <i>h</i> .
<i>ConvexHullTraits_d:: Affinely_independent_d</i>	Predicate object type that provides <i>bool operator()(ForwardIterator first, ForwardIterator last)</i> , which determines if the points <i>tuple[first,last)</i> are affinely independent.
<i>ConvexHullTraits_d:: Contained_in_simplex_d</i>	Predicate object type that provides <i>bool operator()(ForwardIterator first, ForwardIterator last, Point_d p)</i> , which determines if <i>p</i> is contained in the closed simplex defined by the points in <i>tuple[first,last)</i> .
<i>ConvexHullTraits_d:: Contained_in_affined_hull_d</i>	Predicate object type that provides <i>bool operator()(ForwardIterator first, ForwardIterator last, Point_d p)</i> , which determines if <i>p</i> is contained in the affine hull of the points in <i>tuple[first,last)</i> .
<i>ConvexHullTraits_d:: Intersect_d</i>	Predicate object type that provides <i>Object operator()(Ray_d r, Hyperplane_d h)</i> , which determines if <i>r</i> and <i>h</i> intersect and returns the corresponding polymorphic object.

Creation

A default constructor and copy constructor is required.

Operations

For each of the above function and predicate object types, *Func_obj_type*, a function must exist with the name *func_obj_type_object* that creates an instance of the function or predicate object type. For example:

Construct_vector_d *traits.construct_vector_d_object()*

Has Models

CGAL::Cartesian_d<FT,LA>

CGAL::Homogeneous_d<RT,LA>

CGAL::Convex_hull_d_traits_3<R> page 694

CGAL::Convex_hull_d<R>

Definition

An instance C of type *Convex_hull_d<R>* is the convex hull of a multi-set S of points in d -dimensional space. We call S the underlying point set and d or *dim* the dimension of the underlying space. We use d_{cur} to denote the affine dimension of S . The data type supports incremental construction of hulls.

The closure of the hull is maintained as a simplicial complex, i.e., as a collection of simplices. The intersection of any two is a face of both¹. In the sequel we reserve the word simplex for the simplices of dimension d_{cur} . For each simplex there is a handle of type *Simplex_handle* and for each vertex there is a handle of type *Vertex_handle*. Each simplex has $1 + d_{cur}$ vertices indexed from 0 to d_{cur} ; for a simplex s and an index i , $C.vertex(s,i)$ returns the i -th vertex of s . For any simplex s and any index i of s there may or may not be a simplex t opposite to the i -th vertex of s . The function $C.opposite_simplex(s,i)$ returns t if it exists and returns *Simplex_handle()* (the undefined handle) otherwise. If t exists then s and t share d_{cur} vertices, namely all but the vertex with index i of s and the vertex with index $C.index_of_vertex_in_opposite_simplex(s,i)$ of t . Assume that t exists and let $j = C.index_of_vertex_in_opposite_simplex(s,i)$. Then $s = C.opposite_simplex(t,j)$ and $i = C.index_of_vertex_in_opposite_simplex(t,j)$.

The boundary of the hull is also a simplicial complex. All simplices in this complex have dimension $d_{cur} - 1$. For each boundary simplex there is a handle of type *Facet_handle*. Each facet has d_{cur} vertices indexed from 0 to $d_{cur} - 1$. If $d_{cur} > 1$ then for each facet f and each index i , $0 \leq i < d_{cur}$, there is a facet g opposite to the i -th vertex of f . The function $C.opposite_facet(f,i)$ returns g . Two neighboring facets f and g share $d_{cur} - 1$ vertices, namely all but the vertex with index i of f and the vertex with index $C.index_of_vertex_in_opposite_facet(f,i)$ of g . Let $j = C.index_of_vertex_in_opposite_facet(f,i)$. Then $f = C.opposite_facet(g,j)$ and $i = C.index_of_vertex_in_opposite_facet(g,j)$.

Types

<i>Convex_hull_d<R>:: R</i>	the representation class.
<i>Convex_hull_d<R>:: Point_d</i>	the point type.
<i>Convex_hull_d<R>:: Hyperplane_d</i>	the hyperplane type.
<i>Convex_hull_d<R>:: Simplex_handle</i>	handle for simplices.
<i>Convex_hull_d<R>:: Facet_handle</i>	handle for facets.
<i>Convex_hull_d<R>:: Vertex_handle</i>	handle for vertices.
<i>Convex_hull_d<R>:: Simplex_iterator</i>	iterator for simplices.
<i>Convex_hull_d<R>:: Facet_iterator</i>	iterator for facets.
<i>Convex_hull_d<R>:: Vertex_iterator</i>	iterator for vertices.

¹The empty set if a facet of every simplex.

Convex_hull_d<R>:: Hull_vertex_iterator iterator for vertices that are part of the convex hull.

Note that each iterator fits the handle concept, i.e. iterators can be used as handles. Note also that all iterator and handle types come also in a const flavor, e.g., *Vertex_const_iterator* is the constant version of *Vertex_iterator*. Const correctness requires to use the const version whenever the convex hull object is referenced as constant. The *Hull_vertex_iterator* is convertible to *Vertex_iterator* and *Vertex_handle*.

Convex_hull_d<R>:: Point_const_iterator const iterator for all inserted points.

Convex_hull_d<R>:: Hull_point_const_iterator
const iterator for all points that are part of the convex hull.

Creation

Convex_hull_d<R> *C*(*int d*, *R Kernel* = *R*());

creates an instance *C* of type *Convex_hull_d*. The dimension of the underlying space is *d* and *S* is initialized to the empty point set. The traits class *R* specifies the models of all types and the implementations of all geometric primitives used by the convex hull class. The default model is one of the *d*-dimensional representation classes (e.g., *Homogeneous_d*).

The data type *Convex_hull_d* offers neither copy constructor nor assignment operator.

Requirements

R is a model of the concept *ConvexHullTraits_d*.

Operations

All operations below that take a point *x* as argument have the common precondition that *x* is a point of ambient space.

int *C.dimension()*
returns the dimension of ambient space

int *C.current_dimension()*
returns the affine dimension *d_{cur}* of *S*.

Point_d *C.associated_point(Vertex_handle v)*
returns the point associated with vertex *v*.

<i>Vertex_handle</i>	<i>C.vertex_of_simplex(Simplex_handle s, int i)</i>	<p>returns the vertex corresponding to the i-th vertex of s.</p> <p><i>Precondition:</i> $0 \leq i \leq dcur$.</p>
<i>Point_d</i>	<i>C.point_of_simplex(Simplex_handle s, int i)</i>	<p>same as <i>C.associated_point(C.vertex_of_simplex(s,i))</i>.</p>
<i>Simplex_handle</i>	<i>C.opposite_simplex(Simplex_handle s, int i)</i>	<p>returns the simplex opposite to the i-th vertex of s (<i>Simplex_handle()</i> if there is no such simplex).</p> <p><i>Precondition:</i> $0 \leq i \leq dcur$.</p>
<i>int</i>	<i>C.index_of_vertex_in_opposite_simplex(Simplex_handle s, int i)</i>	<p>returns the index of the vertex opposite to the i-th vertex of s.</p> <p><i>Precondition:</i> $0 \leq i \leq dcur$ and there is a simplex opposite to the i-th vertex of s.</p>
<i>Simplex_handle</i>	<i>C.simplex(Vertex_handle v)</i>	<p>returns a simplex of which v is a node. Note that this simplex is not unique.</p>
<i>int</i>	<i>C.index(Vertex_handle v)</i>	<p>returns the index of v in <i>simplex(v)</i>.</p>
<i>Vertex_handle</i>	<i>C.vertex_of_facet(Facet_handle f, int i)</i>	<p>returns the vertex corresponding to the i-th vertex of f.</p> <p><i>Precondition:</i> $0 \leq i < dcur$.</p>
<i>Point_d</i>	<i>C.point_of_facet(Facet_handle f, int i)</i>	<p>same as <i>C.associated_point(C.vertex_of_facet(f,i))</i>.</p>
<i>Facet_handle</i>	<i>C.opposite_facet(Facet_handle f, int i)</i>	<p>returns the facet opposite to the i-th vertex of f (<i>Facet_handle()</i> if there is no such facet).</p> <p><i>Precondition:</i> $0 \leq i < dcur$ and $dcur > 1$.</p>

<i>int</i>	<i>C.index_of_vertex_in_opposite_facet(Facet_handle f, int i)</i> returns the index of the vertex opposite to the i -th vertex of f . <i>Precondition:</i> $0 \leq i < dcur$ and $dcur > 1$.
<i>Hyperplane_d</i>	<i>C.hyperplane_supporting(Facet_handle f)</i> returns a hyperplane supporting facet f . The hyperplane is oriented such that the interior of C is on the negative side of it. <i>Precondition:</i> f is a facet of C and $dcur > 1$.
<i>Vertex_handle</i>	<i>C.insert(Point_d x)</i> adds point x to the underlying set of points. If x is equal to (the point associated with) a vertex of the current hull this vertex is returned and its associated point is changed to x . If x lies outside the current hull, a new vertex v with associated point x is added to the hull and returned. In all other cases, i.e., if x lies in the interior of the hull or on the boundary but not on a vertex, the current hull is not changed and <i>Vertex_handle()</i> is returned. If <i>CGAL_CHECK_EXPENSIVE</i> is defined then the validity check <i>is_valid(true)</i> is executed as a post condition.
<i>template <typename Forward_iterator></i> <i>void</i>	<i>C.insert(Forward_iterator first, Forward_iterator last)</i> adds $S = set [first, last)$ to the underlying set of points. If any point $S[i]$ is equal to (the point associated with) a vertex of the current hull its associated point is changed to $S[i]$.
<i>bool</i>	<i>C.is_dimension_jump(Point_d x)</i> returns true if x is not contained in the affine hull of S .
<i>std::list<Facet_handle></i>	<i>C.facets_visible_from(Point_d x)</i> returns the list of all facets that are visible from x . <i>Precondition:</i> x is contained in the affine hull of S .
<i>Bounded_side</i>	<i>C.bounded_side(Point_d x)</i> returns <i>ON_BOUNDED_SIDE</i> (<i>ON_BOUNDARY</i> , <i>ON_UNBOUNDED_SIDE</i>) if x is contained in the interior (lies on the boundary, is contained in the exterior) of C . <i>Precondition:</i> x is contained in the affine hull of S .

<i>void</i>	<i>C.clear(int d)</i>	reinitializes <i>C</i> to an empty hull in <i>d</i> -dimensional space.
<i>int</i>	<i>C.number_of_vertices()</i>	returns the number of vertices of <i>C</i> .
<i>int</i>	<i>C.number_of_facets()</i>	returns the number of facets of <i>C</i> .
<i>int</i>	<i>C.number_of_simplices()</i>	returns the number of bounded simplices of <i>C</i> .
<i>void</i>	<i>C.print_statistics()</i>	gives information about the size of the current hull and the number of visibility tests performed.
<i>bool</i>	<i>C.is_valid(bool throw_exceptions = false)</i>	checks the validity of the data structure. If <i>throw_exceptions == true</i> then the program throws the following exceptions to inform about the problem. <i>chull_has_center_on_wrong_side_of_hull_facet</i> the hyperplane supporting a facet has the wrong orientation. <i>chull_has_local_non_convexity</i> a ridge is locally non convex. <i>chull_has_double_coverage</i> the hull has a winding number larger than 1.

Lists and Iterators

<i>Vertex_iterator</i>	<i>C.vertices_begin()</i>	an iterator of <i>C</i> to start the iteration over all vertices of the complex.
<i>Vertex_iterator</i>	<i>C.vertices_end()</i>	the past the end iterator for vertices.
<i>Simplex_iterator</i>	<i>C.simplices_begin()</i>	an iterator of <i>C</i> to start the iteration over all simplices of the complex.
<i>Simplex_iterator</i>	<i>C.simplices_end()</i>	the past the end iterator for simplices.
<i>Facet_iterator</i>	<i>C.facets_begin()</i>	an iterator of <i>C</i> to start the iteration over all facets of the complex.

<i>Facet_iterator</i>	<i>C.facets_end()</i>	the past the end iterator for facets.
<i>Hull_vertex_iterator</i>	<i>C.hull_vertices_begin()</i>	an iterator to start the iteration over all vertices of <i>C</i> that are part of the convex hull.
<i>Hull_vertex_iterator</i>	<i>C.hull_vertices_end()</i>	the past the end iterator for hull vertices.
<i>Point_const_iterator</i>	<i>C.points_begin()</i>	returns the start iterator for all points that have been inserted to construct <i>C</i> .
<i>Point_const_iterator</i>	<i>C.points_end()</i>	returns the past the end iterator for points.
<i>Hull_point_const_iterator</i>	<i>C.hull_points_begin()</i>	returns an iterator to start the iteration over all points in the convex hull <i>C</i> . Included are points in the interior of facets.
<i>Hull_point_const_iterator</i>	<i>C.hull_points_end()</i>	returns the past the end iterator for points in the convex hull.
<i>template <typename Visitor></i> <i>void</i>	<i>C.visit_all_facets(Visitor V)</i>	each facet of <i>C</i> is visited by the visitor object <i>V</i> . <i>V</i> has to have a function call operator: <i>void operator()(Facet_handle) const</i>
<i>std::list<Point_d></i>	<i>C.all_points()</i>	returns a list of all points that have been inserted to construct <i>C</i> .
<i>std::list<Vertex_handle></i>	<i>C.all_vertices()</i>	returns a list of all vertices of <i>C</i> (also interior ones).
<i>std::list<Simplex_handle></i>	<i>C.all_simplices()</i>	returns a list of all simplices in <i>C</i> .
<i>std::list<Facet_handle></i>	<i>C.all_facets()</i>	returns a list of all facets of <i>C</i> .

Iteration Statements

forall_ch_vertices(v, C) { “the vertices of C are successively assigned to v ” }

forall_ch_simplices(s, C) { “the simplices of C are successively assigned to s ” }

forall_ch_facets(f, C) { “the facets of C are successively assigned to f ” }

Implementation

The implementation of type *Convex_hull_d* is based on [CMS93] and [BMS94]. The details of the implementation can be found in the implementation document available at the download site of this package.

The time and space requirements are input dependent. Let C_1, C_2, C_3, \dots be the sequence of hulls constructed and for a point x let k_i be the number of facets of C_i that are visible from x and that are not already facets of C_{i-1} . Then the time for inserting x is $O(\dim \sum_i k_i)$ and the number of new simplices constructed during the insertion of x is the number of facets of the hull which were not already facets of the hull before the insertion.

The data type *Convex_hull_d* is derived from *Regular_complex_d*. The space requirement of regular complexes is essentially $12(\dim + 2)$ bytes times the number of simplices plus the space for the points. *Convex_hull_d* needs an additional $8 + (4 + x)\dim$ bytes per simplex where x is the space requirement of the underlying number type and an additional 12 bytes per point. The total is therefore $(16 + x)\dim + 32$ bytes times the number of simplices plus $28 + x \cdot \dim$ bytes times the number of points.

Low Dimensional Conversion Routine

```
include <CGAL/Convex_hull_d_to_polyhedron_3.h>
```

```
template <class R, class T, class HDS>
void convex_hull_d_to_polyhedron_3( C, Polyhedron_3<T,HDS>& P)
```

converts the convex hull C to polyedral surface stored in P .

Precondition: $\dim == 3$ and $dcur == 3$.

Low Dimensional Output Routines

```
include <CGAL/IO/Convex_hull_d_window_stream.h>
```

```
template <class R>
void d2_show( C, CGAL::Window_stream& W)
```

draws the convex hull C in window W .

Precondition: $\dim == 2$.

```
template <class R>
void d3_surface_map( C, GRAPH< typename ::Point_d ,int>& G)
```

constructs the representation of the surface of C as a bidi-rected LEDA graph G .

Precondition: $\dim == 3$.

CGAL::Convex_hull_d_traits_3<R>

Definition

The class *Convex_hull_d_traits_3<R>* serves as a traits class for the class *Convex_hull_d*. This is a traits class that adapts any low-dimensional standard kernel model, e.g. *Homogeneous<RT>* or *Cartesian<FT>* for the fixed 3-dimensional usage of *Convex_hull_d*.

```
#include <CGAL/Convex_hull_d_traits_3.h>
```

Is Model for the Concepts

ConvexHullTraits_d

Creation

Convex_hull_d_traits_3<R> traits; default constructor.

DelaunayLiftedTraits_d

Definition

Requirements of the second traits class to be used with the class *Delaunay_d*.

Types

<i>DelaunayLiftedTraits_d:: Point_d</i>	the dD point type on which the Delaunay algorithm operates
<i>DelaunayLiftedTraits_d:: Hyperplane_d</i>	a dD plane
<i>DelaunayLiftedTraits_d:: Vector_d</i>	a dD vector
<i>DelaunayLiftedTraits_d:: Ray_d</i>	a dD ray
<i>DelaunayLiftedTraits_d:: RT</i>	a arithmetic ring type
<i>DelaunayLiftedTraits_d:: Construct_vector_d</i>	Function object type that provides <i>Vector_d operator()(int d, CGAL::Null_vector)</i> , which constructs and returns the null vector.
<i>DelaunayLiftedTraits_d:: Construct_hyperplane_d</i>	Function object type that provides <i>Hyperplane_d operator()(ForwardIterator first, ForwardIterator last, Point_d p, CGAL::Oriented_side side)</i> , which constructs and returns a hyperplane passing through the points in <i>tuple[first,last)</i> and oriented such that <i>p</i> is on the side <i>side</i> of the returned hyperplane. When <i>side==ON_ORIENTED_BOUNDARY</i> then any hyperplane containing the tuple is returned.
<i>DelaunayLiftedTraits_d:: Vector_to_point_d</i>	Function object type that provides <i>Point_d operator()(Vector_d v)</i> , which constructs and returns the point defined by $0 + v$.
<i>DelaunayLiftedTraits_d:: Point_to_vector_d</i>	Function object type that provides <i>Vector_d operator()(Point_d p)</i> , which constructs and returns the vector defined by $p - 0$.
<i>DelaunayLiftedTraits_d:: Orientation_d</i>	Function object type that provides <i>Orientation operator()(ForwardIterator first, ForwardIterator last)</i> , which determines the orientation of the points <i>tuple[first,last)</i> .
<i>DelaunayLiftedTraits_d:: Orthogonal_vector_d</i>	Function object type that provides <i>Vector_d operator()(Hyperplane_d h)</i> , which constructs and returns a vector orthogonal to <i>h</i> and pointing from the boundary into its positive halfspace.
<i>DelaunayLiftedTraits_d:: Oriented_side_d</i>	Predicate object type that provides <i>Oriented_side operator()(Hyperplane_d h, Point_d p)</i> , which determines the oriented side of <i>p</i> with respect to <i>h</i> .

<i>DelaunayLiftedTraits_d:: Has_on_positive_side_d</i>	Predicate object type that provides <i>bool operator()(Hyperplane_d h, Point_d p)</i> , which return true iff p lies in the positive halfspace determined by h .
<i>DelaunayLiftedTraits_d:: Affinely_independent_d</i>	Predicate object type that provides <i>bool operator()(ForwardIterator first, ForwardIterator last)</i> , which determines if the points <i>tuple[first,last)</i> are affinely independent.
<i>DelaunayLiftedTraits_d:: Contained_in_simplex_d</i>	Predicate object type that provides <i>bool operator()(ForwardIterator first, ForwardIterator last, Point_d p)</i> , which determines if p is contained in the closed simplex defined by the points in <i>tuple[first,last)</i> .
<i>DelaunayLiftedTraits_d:: Contained_in_affined_hull_d</i>	Predicate object type that provides <i>bool operator()(ForwardIterator first, ForwardIterator last, Point_d p)</i> , which determines if p is contained in the affine hull of the points in <i>tuple[first,last)</i> .
<i>DelaunayLiftedTraits_d:: Intersect_d</i>	Predicate object type that provides <i>Object operator()(Ray_d r, Hyperplane_d h)</i> , which determines if r and h intersect and returns the corresponding polymorphic object.
The previous requirements are all identical to the concept <i>ConvexHullTraits_d</i> . The Delaunay class adds the following requirements.	
<i>DelaunayLiftedTraits_d:: Project_along_d_axis_d</i>	Predicate object type that provides <i>DelaunayTraits_d::Point_d operator()(Point_d p)</i> , which determines the $d - 1$ -dimensional point from the d -dimensional point p while discarding the last coordinate.
<i>DelaunayLiftedTraits_d:: Lift_to_paraboloid_d</i>	Predicate object type that provides <i>Point_d operator()(DelaunayTraits_d::Point_d p)</i> , which determines the d -dimensional point from the $d - 1$ -dimensional point p while lifting it to the paraboloid of revolution.
<i>DelaunayLiftedTraits_d:: Component_accessor_d</i>	Predicate object type that provides <i>RT homogeneous(Vector_d v,int i)</i> and <i>int dimension(Vector_d v)</i> , where the former determines the i th coordinate of v and the latter the dimension of v .

Creation

A default constructor and copy constructor is required.

Operations

For each of the above function and predicate object types, *Func_obj_type*, a function must exist with the name *func_obj_type_object* that creates an instance of the function or predicate object type. For example:

Construct_vector_d *traits.construct_vector_d_object()*

Has Models

CGAL::Cartesian_d<FT,LA>
CGAL::Homogeneous_d<RT,LA>

DelaunayTraits_d

Definition

Requirements of the first traits class to be used with the class *Delaunay_d*.

Types

<i>DelaunayTraits_d:: Point_d</i>	the dD point type on which the delaunay algorithm operates
<i>DelaunayTraits_d:: Sphere_d</i>	a dD sphere
<i>DelaunayTraits_d:: FT</i>	an arithmetic field type
<i>DelaunayTraits_d:: Point_of_sphere_d</i>	Predicate object type that provides <i>Point_d operator()(Sphere_d s, int i)</i> , which returns the <i>i</i> th point defining sphere <i>s</i> .
<i>DelaunayTraits_d:: Construct_sphere_d</i>	Predicate object type that provides <i>Sphere_d operator()(int d, ForwardIterator first, ForwardIterator last)</i> , which returns a dD sphere through the points in <i>tuple[first,last)</i> .
<i>DelaunayTraits_d:: Contained_in_simplex_d</i>	Predicate object type that provides <i>bool operator()(ForwardIterator first, ForwardIterator last, Point_d p)</i> , which determines if <i>p</i> is contained in the closed simplex defined by the points in <i>tuple[first,last)</i> .
<i>DelaunayTraits_d:: Squared_distance_d</i>	Predicate object type that provides <i>FT operator()(Point_d p, Point_d q)</i> , which determines the squared distance from <i>p</i> to <i>q</i> .
<i>DelaunayTraits_d:: Affinely_independent_d</i>	Predicate object type that provides <i>bool operator()(ForwardIterator first, ForwardIterator last)</i> , which determines if the points in <i>tuple[first,last)</i> are affinely independent.

Creation

A default constructor and copy constructor is required.

Operations

For each of the above function and predicate object types, *Func_obj_type*, a function must exist with the name *func_obj_type_object* that creates an instance of the function or predicate object type. For example:

Construct_sphere_d *traits.construct_sphere_d_object()*

Has Models

CGAL::Cartesian_d<FT,LA>

CGAL::Homogeneous_d<RT,LA>

CGAL::Delaunay_d< R, Lifted_R >

Definition

An instance DT of type $Delaunay_d< R, Lifted_R >$ is the nearest and furthest site Delaunay triangulation of a set S of points in some d -dimensional space. We call S the underlying point set and d or dim the dimension of the underlying space. We use $dcur$ to denote the affine dimension of S . The data type supports incremental construction of Delaunay triangulations and various kind of query operations (in particular, nearest and furthest neighbor queries and range queries with spheres and simplices).

A Delaunay triangulation is a simplicial complex. All simplices in the Delaunay triangulation have dimension $dcur$. In the nearest site Delaunay triangulation the circumsphere of any simplex in the triangulation contains no point of S in its interior. In the furthest site Delaunay triangulation the circumsphere of any simplex contains no point of S in its exterior. If the points in S are co-circular then any triangulation of S is a nearest as well as a furthest site Delaunay triangulation of S . If the points in S are not co-circular then no simplex can be a simplex of both triangulations. Accordingly, we view DT as either one or two collection(s) of simplices. If the points in S are co-circular there is just one collection: the set of simplices of some triangulation. If the points in S are not co-circular there are two collections. One collection consists of the simplices of a nearest site Delaunay triangulation and the other collection consists of the simplices of a furthest site Delaunay triangulation.

For each simplex of maximal dimension there is a handle of type *Simplex_handle* and for each vertex of the triangulation there is a handle of type *Vertex_handle*. Each simplex has $1 + dcur$ vertices indexed from 0 to $dcur$. For any simplex s and any index i , $DT.vertex_of(s,i)$ returns the i -th vertex of s . There may or may not be a simplex t opposite to the vertex of s with index i . The function $DT.opposite_simplex(s,i)$ returns t if it exists and returns *Simplex_handle()* otherwise. If t exists then s and t share $dcur$ vertices, namely all but the vertex with index i of s and the vertex with index $DT.index_of_vertex_in_opposite_simplex(s,i)$ of t . Assume that $t = DT.opposite_simplex(s,i)$ exists and let $j = DT.index_of_vertex_in_opposite_simplex(s,i)$. Then $s = DT.opposite_simplex(t,j)$ and $i = DT.index_of_vertex_in_opposite_simplex(t,j)$. In general, a vertex belongs to many simplices.

Any simplex of DT belongs either to the nearest or to the furthest site Delaunay triangulation or both. The test $DT.simplex_of_nearest(dt_simplex\ s)$ returns true if s belongs to the nearest site triangulation and the test $DT.simplex_of_furthest(dt_simplex\ s)$ returns true if s belongs to the furthest site triangulation.

Inherits From

Convex_hull_d<Lifted_R>

Types

<i>Delaunay_d< R, Lifted_R >:: Simplex_handle</i>	handles to the simplices of the complex.
<i>Delaunay_d< R, Lifted_R >:: Vertex_handle</i>	handles to vertices of the complex.
<i>Delaunay_d< R, Lifted_R >:: Point_d</i>	the point type
<i>Delaunay_d< R, Lifted_R >:: Sphere_d</i>	the sphere type

```
enum Delaunay_voronoi_kind { NEAREST, FURTHEST};
```

```
interface flags
```

To use these types you can typedef them into the global scope after instantiation of the class. We use *Vertex_handle* instead of *Delaunay_d< R, Lifted_R >::Vertex_handle* from now on. Similarly we use *Simplex_handle*.

Delaunay_d< R, Lifted_R >:: Point_const_iterator the iterator for points.

Delaunay_d< R, Lifted_R >:: Vertex_iterator the iterator for vertices.

Delaunay_d< R, Lifted_R >:: Simplex_iterator the iterator for simplices.

Creation

```
Delaunay_d< R, Lifted_R > DT( int d, R k1 = R(), Lifted_R k2 = Lifted_R());
```

creates an instance *DT* of type *Delaunay_d*. The dimension of the underlying space is *d* and *S* is initialized to the empty point set. The traits class *R* specifies the models of all types and the implementations of all geometric primitives used by the Delaunay class. The traits class *Lifted_R* specifies the models of all types and the implementations of all geometric primitives used by the base class of *Delaunay_d< R, Lifted_R >*. The second template parameter defaults to the first: *Delaunay_d<R> = Delaunay_d<R, Lifted_R = R >*.

The data type *Delaunay_d* offers neither copy constructor nor assignment operator.

Requirements

R is a model of the concept *DelaunayTraits_d* . *Lifted_R* is a model of the concept *DelaunayLiftedTraits_d* .

Operations

All operations below that take a point *x* as an argument have the common precondition that *x.dimension()* = *DT.dimension()*.

int *DT.dimension()* returns the dimension of ambient space

int *DT.current_dimension()*

returns the affine dimension of the current point set, i.e., -1 if *S* is empty, 0 if *S* consists of a single point, 1 if all points of *S* lie on a common line, etcetera.

bool *DT.is_simplex_of_nearest(Simplex_handle s)*

returns true if *s* is a simplex of the nearest site triangulation.

<i>bool</i>	<i>DT.is_simplex_of_furthest(Simplex_handle s)</i>	returns true if <i>s</i> is a simplex of the furthest site triangulation.
<i>Vertex_handle</i>	<i>DT.vertex_of_simplex(Simplex_handle s, int i)</i>	returns the vertex associated with the <i>i</i> -th node of <i>s</i> . <i>Precondition:</i> $0 \leq i \leq d_{cur}$.
<i>Point_d</i>	<i>DT.associated_point(Vertex_handle v)</i>	returns the point associated with vertex <i>v</i> .
<i>Point_d</i>	<i>DT.point_of_simplex(Simplex_handle s, int i)</i>	returns the point associated with the <i>i</i> -th vertex of <i>s</i> . <i>Precondition:</i> $0 \leq i \leq d_{cur}$.
<i>Simplex_handle</i>	<i>DT.opposite_simplex(Simplex_handle s, int i)</i>	returns the simplex opposite to the <i>i</i> -th vertex of <i>s</i> (<i>Simplex_handle()</i> if there is no such simplex). <i>Precondition:</i> $0 \leq i \leq d_{cur}$.
<i>int</i>	<i>DT.index_of_vertex_in_opposite_simplex(Simplex_handle s, int i)</i>	returns the index of the vertex opposite to the <i>i</i> -th vertex of <i>s</i> . <i>Precondition:</i> $0 \leq i \leq d_{cur}$.
<i>Simplex_handle</i>	<i>DT.simplex(Vertex_handle v)</i>	returns a simplex of the nearest site triangulation incident to <i>v</i> .
<i>int</i>	<i>DT.index(Vertex_handle v)</i>	returns the index of <i>v</i> in <i>DT.simplex(v)</i> .
<i>bool</i>	<i>DT.contains(Simplex_handle s, Point_d x)</i>	returns true if <i>x</i> is contained in the closure of simplex <i>s</i> .
<i>bool</i>	<i>DT.empty()</i>	decides whether <i>DT</i> is empty.
<i>void</i>	<i>DT.clear()</i>	reinitializes <i>DT</i> to the empty Delaunay triangulation.

<i>Vertex_handle</i>	<i>DT.insert(Point_d x)</i>	inserts point x into DT and returns the corresponding <i>Vertex_handle</i> . More precisely, if there is already a vertex v in DT positioned at x (i.e., <i>associated_point</i> (v) is equal to x) then <i>associated_point</i> (v) is changed to x (i.e., <i>associated_point</i> (v) is made identical to x) and if there is no such vertex then a new vertex v with <i>associated_point</i> (v) = x is added to DT . In either case, v is returned.
<i>Simplex_handle</i>	<i>DT.locate(Point_d x)</i>	returns a simplex of the nearest site triangulation containing x in its closure (returns <i>Simplex_handle</i> () if x lies outside the convex hull of S).
<i>Vertex_handle</i>	<i>DT.lookup(Point_d x)</i>	if DT contains a vertex v with <i>associated_point</i> (v) = x the result is v otherwise the result is <i>Vertex_handle</i> ().
<i>Vertex_handle</i>	<i>DT.nearest_neighbor(Point_d x)</i>	computes a vertex v of DT that is closest to x , i.e., $\text{dist}(x, \text{associated_point}(v)) = \min\{\text{dist}(x, \text{associated_point}(u)) \mid u \in S\}.$
<i>std::list<Vertex_handle></i>	<i>DT.range_search(Sphere_d C)</i>	returns the list of all vertices contained in the closure of sphere C .
<i>std::list<Vertex_handle></i>	<i>DT.range_search(std::vector<Point_d> A)</i>	returns the list of all vertices contained in the closure of the simplex whose corners are given by A . <i>Precondition:</i> A must consist of $d + 1$ affinely independent points in base space.
<i>std::list<Simplex_handle></i>	<i>DT.all_simplices(Delaunay_voronoi_kind k = NEAREST)</i>	returns a list of all simplices of either the nearest or the furthest site Delaunay triangulation of S .

std::list<Vertex_handle>

DT.all_vertices(Delaunay_voronoi_kind k = NEAREST)

returns a list of all vertices of either the nearest or the furthest site Delaunay triangulation of *S*.

std::list<Point_d>

DT.all_points()

returns *S*.

Point_const_iterator

DT.points_begin()

returns the start iterator for points in *DT*.

Point_const_iterator

DT.points_end()

returns the past the end iterator for points in *DT*.

Simplex_iterator

DT.simplices_begin(Delaunay_voronoi_kind k = NEAREST)

returns the start iterator for simplices of *DT*.

Simplex_iterator

DT.simplices_end()

returns the past the end iterator for simplices of *DT*.

Implementation

The data type is derived from *Convex_hull_d* via the lifting map. For a point *x* in *d*-dimensional space let *lift(x)* be its lifting to the unit paraboloid of revolution. There is an intimate relationship between the Delaunay triangulation of a point set *S* and the convex hull of *lift(S)*: The nearest site Delaunay triangulation is the projection of the lower hull and the furthest site Delaunay triangulation is the upper hull. For implementation details we refer the reader to the implementation report available from the CGAL server.

The space requirement is the same as for convex hulls. The time requirement for an insert is the time to insert the lifted point into the convex hull of the lifted points.

Example

The abstract data type *Delaunay_d* has a default instantiation by means of the *d*-dimensional geometric kernel.

```
#include <CGAL/Homogeneous_d.h>
#include <CGAL/leda_integer.h>
#include <CGAL/Delaunay_d.h>

typedef leda_integer RT;
typedef CGAL::Homogeneous_d<RT> Kernel;
typedef CGAL::Delaunay_d<Kernel> Delaunay_d;
typedef Delaunay_d::Point_d Point;
typedef Delaunay_d::Simplex_handle Simplex_handle;
typedef Delaunay_d::Vertex_handle Vertex_handle;

int main()
{
    Delaunay_d T(2);
    Vertex_handle v1 = T.insert(Point_d(2,11));
    ...
}
```


Traits requirements

Delaunay_d< *R*, *Lifted_R* > requires the following types from the kernel traits *Lifted_R*:

RT Point_d Vector_d Ray_d Hyperplane_d

and uses the following function objects from the kernel traits:

Construct_hyperplane_d
Construct_vector_d
Vector_to_point_d / Point_to_vector_d
Orientation_d
Orthogonal_vector_d
Oriented_side_d / Has_on_positive_side_d
Affinely_independent_d
Contained_in_simplex_d
Contained_in_affine_hull_d
Intersect_d
Lift_to_paraboloid_d / Project_along_d_axis_d
Component_accessor_d

Delaunay_d< *R*, *Lifted_R* > requires the following types from the kernel traits *R*:

FT Point_d Sphere_d

and uses the following function objects from the kernel traits *R*:

Construct_sphere_d
Squared_distance_d
Point_of_sphere_d
Affinely_independent_d
Contained_in_simplex_d

Low Dimensional Output Routines

include <CGAL/IO/Delaunay_d_window_stream.h>

template <typename *R*, typename *Lifted_R*>

template <typename *R*, typename *Lifted_R*> void

d2_show(*Delaunay_d*<*R*,*Lifted_R*> *D*,
CGAL::Window_stream& *W*,
typename *Delaunay_d*<*R*,*Lifted_R*>::Delaunay_voronoi_kind *k* = *Delaunay_d*<*R*,*Lifted_R*>::NEAREST)

draws the underlying simplicial complex *D* into window *W*.

Precondition: *dim* == 2.

template <typename *R*, typename *Lifted_R*>

```
template <typename R, typename Lifted_R> void
```

```
    d2_map( Delaunay_d<R,Lifted_R> D,
            GRAPH< typename Delaunay_d<R,Lifted_R>::Point_d, int > & DTG,
            typename Delaunay_d<R,Lifted_R>::Delaunay_voronoi_kind k = Delaunay_
d<R,Lifted_R>::NEAREST)
```

constructs a LEDA graph representation of the nearest (*kind* = *NEAREST* or the furthest (*kind* = *FURTHEST*) site Delaunay triangulation.

Precondition: *dim()* == 2.

Part IV

Polygons and Polyhedra

Chapter 8

2D Polygons

Geert-Jan Giezeman and Wieger Wesselink

8.1 Introduction

A polygon is a closed chain of edges. Several algorithms are available for polygons. For some of those algorithms, it is necessary that the polygon is simple. A polygon is simple if edges don't intersect, except consecutive edges, which intersect in their common vertex.

The following algorithms are available:

- find the leftmost, rightmost, topmost and bottommost vertex.
- compute the (signed) area.
- check if a polygon is simple.
- check if a polygon is convex.
- find the orientation (clockwise or counterclockwise)
- check if a point lies inside a polygon.

All those operations take two forward iterators as parameters in order to describe the polygon. These parameters have a point type as value type.

The type *Polygon_2* can be used to represent polygons. Polygons are dynamic. Vertices can be modified, inserted and erased. They provide the algorithms described above as member functions. Moreover, they provide ways of iterating over the vertices and edges.

Currently, the *Polygon_2* class is a nice wrapper around the algorithms, but little more. Especially, computed values are not cached. That is, when the *is_simple()* member function is called twice or more, the result is computed each time anew. It is possible to set a preprocessor flag to alter this behaviour. In the future, caching will become the default.

8.2 Example

The following code fragment creates a polygon and does some checks.

```

// file: examples/Polygon/Polygon.C

#include <CGAL/Cartesian.h>
#include <CGAL/Polygon_2.h>
#include <iostream>

typedef CGAL::Cartesian<double> K;
typedef K::Point_2 Point;
typedef CGAL::Polygon_2<K> Polygon;
using std::cout; using std::endl;

int main()
{
    Point points[] = { Point(0,0), Point(5.1,0), Point(1,1), Point(0.5,6) };
    Polygon pgn(points, points+4);

    // check if the polygon is simple.
    cout << "The polygon is " <<
        (pgn.is_simple() ? " " : "not ") << "simple." << endl;

    // check if the polygon is convex
    cout << "The polygon is " <<
        (pgn.is_convex() ? " " : "not ") << "convex." << endl;

    return 0;
}

```

```

// file: examples/Polygon/polygon_algorithms.C

#include <CGAL/Cartesian.h>
#include <CGAL/Polygon_2_algorithms.h>
#include <iostream>

typedef CGAL::Cartesian<double> K;
typedef K::Point_2 Point;
using std::cout; using std::endl;

void check_inside(Point pt, Point *pgn_begin, Point *pgn_end, K traits)
{
    cout << "The point " << pt;
    switch(CGAL::bounded_side_2(pgn_begin, pgn_end, pt, traits)) {
        case CGAL::ON_BOUNDED_SIDE :
            cout << " is inside the polygon.\n";
            break;
        case CGAL::ON_BOUNDARY:
            cout << " is on the polygon boundary.\n";
            break;
        case CGAL::ON_UNBOUNDED_SIDE:
            cout << " is outside the polygon.\n";
            break;
    }
}

```

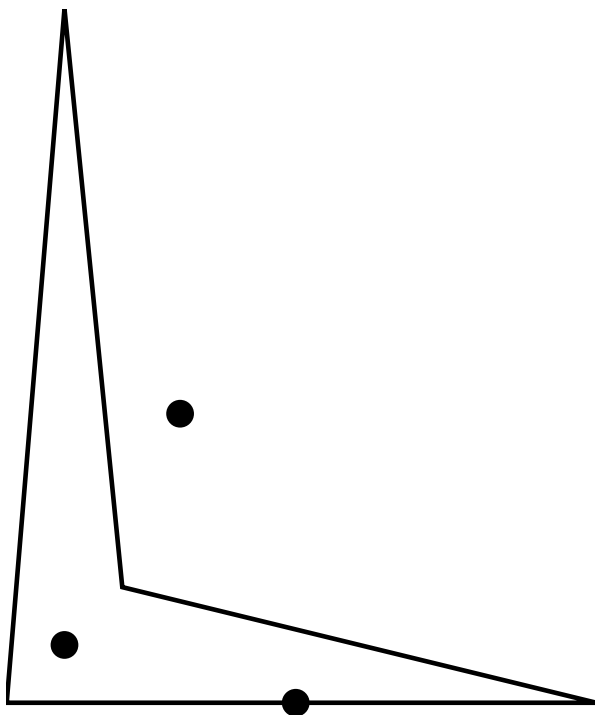


Figure 8.1: A polygon and some points

```

}

int main()
{
    Point points[] = { Point(0,0), Point(5.1,0), Point(1,1), Point(0.5,6)};

    // check if the polygon is simple.
    cout << "The polygon is "
        << (CGAL::is_simple_2(points, points+4, K()) ? "" : "not ")
        << "simple." << endl;

    check_inside(Point(0.5, 0.5), points, points+4, K());
    check_inside(Point(1.5, 2.5), points, points+4, K());
    check_inside(Point(2.5, 0), points, points+4, K());

    return 0;
}

```


2D Polygons

Reference Manual

Geert-Jan Giezeman and Wiegner Wesselink

Assertions

The assertion flags for the polygons and polygon operations use *POLYGON* in their names (e.g., *CGAL_POLYGON_NO_ASSERTIONS*).

8.3 Classified Reference Pages

Concepts

PolygonTraits_2 page [724](#)

Classes

CGAL::Polygon_2<PolygonTraits_2, Container> page [726](#)

Global Functions

CGAL::area_2 page [715](#)
CGAL::bbox_2 page [716](#)
CGAL::bottom_vertex_2 page [717](#)
CGAL::bounded_side_2 page [718](#)
CGAL::is_convex_2 page [719](#)
CGAL::is_simple_2 page [720](#)
CGAL::left_vertex_2 page [721](#)
CGAL::orientation_2 page [722](#)
CGAL::oriented_side_2 page [723](#)
CGAL::polygon_area_2 page [732](#)
CGAL::right_vertex_2 page [733](#)
CGAL::top_vertex_2 page [734](#)

8.4 Alphabetical List of Reference Pages

<i>area_2</i>	page 715
<i>bbox_2</i>	page 716
<i>bottom_vertex_2</i>	page 717
<i>bounded_side_2</i>	page 718
<i>is_convex_2</i>	page 719
<i>is_simple_2</i>	page 720
<i>left_vertex_2</i>	page 721
<i>orientation_2</i>	page 722
<i>oriented_side_2</i>	page 723
<i>PolygonTraits_2</i>	page 724
<i>Polygon_2</i> < <i>PolygonTraits_2</i> , <i>Container</i> >	page 726
<i>polygon_area_2</i>	page 732
<i>right_vertex_2</i>	page 733
<i>top_vertex_2</i>	page 734

CGAL::area_2

Definition

The function *area_2* computes the signed area of a polygon.

```
#include <CGAL/Polygon_2_algorithms.h>
```

```
template <class ForwardIterator, class Traits>
void      area_2( ForwardIterator first,
                  ForwardIterator last,
                  typename Traits::FT &result,
                  Traits traits)
```

Computes the signed area of the polygon defined by the range of points *first* ... *last*. The area is returned in the parameter *result*. The sign is positive for counterclockwise polygons, negative for clockwise polygons. If the polygon is not simple, the area is not well defined. The functionality is also available by the *polygon_area_2* function, which returns the area instead of taking it as a parameter.

Requirements

1. *Traits* is a model of the concept *PolygonTraits_2*. Only the following members of this traits class are used:
 - *Compute_area_2* : Computes the signed area of the oriented triangle defined by 3 *Point_2* passed as arguments.
 - *FT*
 - *compute_area_2_object*
2. *ForwardIterator::value_type* should be *Traits::Point_2*,

See Also

CGAL::polygon_area_2 page [732](#)
PolygonTraits_2 page [724](#)
CGAL::orientation_2 page [722](#)
CGAL::Polygon_2<PolygonTraits_2, Container> page [726](#)

CGAL::bbox_2

Definition

```
#include <CGAL/Polygon_2_algorithms.h>
```

```
template <class InputIterator, class Traits>
Bbox_2          bbox_2( InputIterator first, InputIterator last, Traits traits)
```

Returns the bounding box of the range $[first, last)$.

Requirements

1. *Traits* is a model of the concept `PolygonTraits_2`. In fact, only the members *Construct_bbox_2* and *construct_bbox_2_object* are used.
2. *InputIterator::value_type* should be *Traits::Point_2*,

See Also

CGAL::Polygon_2<PolygonTraits_2, Container> page [726](#)

CGAL::bottom_vertex_2

Definition

```
#include <CGAL/Polygon_2_algorithms.h>
```

```
template <class ForwardIterator, class Traits>
```

```
ForwardIterator bottom_vertex_2( ForwardIterator first, ForwardIterator last, Traits traits)
```

Returns an iterator to the bottommost point from the range $[first, last)$. In case of a tie, the point with the smallest x -coordinate is taken.

Requirements

1. *Traits* is a model of the concept `PolygonTraits_2`. In fact, only the members `Less_yx_2` and `less_yx_2_object` are used.
2. `ForwardIterator::value_type` should be `Traits::Point_2`,

See Also

`CGAL::left_vertex_2` page [721](#)
`CGAL::right_vertex_2` page [733](#)
`CGAL::top_vertex_2` page [734](#)
`CGAL::Polygon_2<PolygonTraits_2, Container>` page [726](#)
`PolygonTraits_2` page [724](#)

Example

CGAL::bounded_side_2

Definition

The function *bounded_side_2* computes if a point lies inside a polygon.

```
#include <CGAL/Polygon_2_algorithms.h>
```

```
template <class ForwardIterator, class Traits>
Bounded_side          bounded_side_2( ForwardIterator first,
                                      ForwardIterator last,
                                      Traits::Point_2 point,
                                      Traits traits)
```

The function *bounded_side_2* computes if a point lies inside a polygon. The polygon is defined by the sequence of points *first*...*last*. Being inside is defined by the odd-even rule. If we take a ray starting at the point and extending to infinity (in any direction), we count the number of intersections. If this number is odd, the point is inside, otherwise it is outside. If the point is on a polygon edge, a special value is returned. A simple polygon divides the plane in an unbounded and a bounded region. According to the definition points in the bounded region are inside the polygon.

Requirements

1. *Traits* is a model of the concept *PolygonTraits_2*. Only the following members of this traits class are used:

- *Compare_x_2*
- *Compare_y_2*
- *Orientation_2*
- *compare_x_2_object*
- *compare_y_2_object*
- *orientation_2_object*

2. *ForwardIterator::value_type* should be *Traits::Point_2*,

Implementation

The running time is linear in the number of vertices of the polygon. A horizontal ray is taken to count the number of intersections. Special care is taken that the result is correct even if there are degeneracies (if the ray passes through a vertex).

See Also

PolygonTraits_2 page [724](#)
CGAL::oriented_side_2 page [723](#)
CGAL::Polygon_2<PolygonTraits_2, Container> page [726](#)
CGAL::Bounded_side

CGAL::is_convex_2

Definition

The function *is_convex_2* computes if a polygon is convex.

```
#include <CGAL/Polygon_2_algorithms.h>
```

```
template <class ForwardIterator, class Traits>
bool is_convex_2( ForwardIterator first, ForwardIterator last, Traits traits)
```

Requirements

1. *Traits* is a model of the concept *PolygonTraits_2*. Only the following members of this traits class are used:
 - *Less_xy_2*
 - *Orientation_2*
 - *less_xy_2_object*
 - *orientation_2_object*
2. *ForwardIterator::value_type* should be *Traits::Point_2*,

See Also

PolygonTraits_2 page [724](#)
CGAL::Polygon_2<*PolygonTraits_2*, *Container*> page [726](#)

CGAL::is_simple_2

Definition

The function *is_simple_2* computes if a polygon is simple, that is, does not self-intersect.

```
#include <CGAL/Polygon_2_algorithms.h>
```

```
template <class ForwardIterator, class Traits>
inline bool is_simple_2( ForwardIterator first, ForwardIterator last, Traits traits)
```

A polygon is called simple if the edges do not intersect, except consecutive edge in their common vertex. This function checks if the polygon defined by the range *first* ... *last* is simple.

Requirements

1. *Traits* is a model of the concept *PolygonTraits_2*. Only the following members of this traits class are used:
 - *Point_2*
 - *Less_xy_2*
 - *Orientation_2*
 - *less_xy_2*
 - *orientation_2*
2. *ForwardIterator::value_type* should be *Traits::Point_2*,

Implementation

The simplicity test is implemented by means of a plane sweep algorithm. The algorithm is quite robust when used with inexact number types. The running time is $O(n \log n)$, where n is the number of vertices of the polygon.

See Also

PolygonTraits_2 page [724](#)
CGAL::Polygon_2<*PolygonTraits_2*, *Container*> page [726](#)

CGAL::left_vertex_2

Definition

```
#include <CGAL/Polygon_2_algorithms.h>
```

```
template <class ForwardIterator, class Traits>
```

```
ForwardIterator left_vertex_2( ForwardIterator first, ForwardIterator last, Traits traits)
```

Returns an iterator to the leftmost point from the range $[first, last)$. In case of a tie, the point with the smallest y-coordinate is taken.

Requirements

1. *Traits* is a model of the concept *PolygonTraits_2*. In fact, only the members *Less_xy_2* and *less_xy_2_object* are used.
2. *ForwardIterator::value_type* should be *Traits::Point_2*,

See Also

CGAL::right_vertex_2 page [733](#)

CGAL::top_vertex_2 page [734](#)

CGAL::bottom_vertex_2 page [717](#)

CGAL::Polygon_2<*PolygonTraits_2*, *Container*> page [726](#)

CGAL::orientation_2

Definition

The function *orientation_2* computes if a polygon is clockwise or counterclockwise oriented.

```
#include <CGAL/Polygon_2_algorithms.h>
```

```
template <class ForwardIterator, class Traits>
```

```
Orientation          orientation_2( ForwardIterator first, ForwardIterator last, Traits traits)
```

Precondition: *is_simple_2*(first, last, traits);

Requirements

1. *Traits* is a model of the concept *PolygonTraits_2*. Only the following members of this traits class are used:
 - *Less_xy_2*
 - *less_xy_2_object*
 - *orientation_2_object*
2. *ForwardIterator::value_type* should be *Traits::Point_2*,

See Also

PolygonTraits_2 page [724](#)
CGAL::is_simple_2 page [720](#)
CGAL::Polygon_2<*PolygonTraits_2*, *Container*> page [726](#)
CGAL::Orientation

CGAL::oriented_side_2

Definition

The function *oriented_side_2* computes on which side of a polygon a point lies.

```
#include <CGAL/Polygon_2_algorithms.h>
```

```
template <class ForwardIterator, class Traits>
Oriented_side          oriented_side_2( ForwardIterator first,
                                       ForwardIterator last,
                                       Traits::Point_2 point,
                                       Traits traits)
```

Requirements

1. *Traits* is a model of the concept *PolygonTraits_2*. Only the following members of this traits class are used:
 - *Less_xy_2*
 - *Compare_x_2*
 - *Compare_y_2*
 - *Orientation_2*
 - *less_xy_2_object*
 - *compare_x_2_object*
 - *compare_y_2_object*
 - *orientation_2_object*
2. *ForwardIterator::value_type* should be *Traits::Point_2*,

See Also

PolygonTraits_2 page [724](#)
CGAL::bounded_side_2 page [718](#)
CGAL::is_simple_2 page [720](#)
CGAL::Polygon_2<PolygonTraits_2, Container> page [726](#)
CGAL::Oriented_side

PolygonTraits_2

Definition

The *Polygon_2* class and the functions that implement the functionality found in that class each are parameterized by a traits class that defines the primitives used in the algorithms. The concept *PolygonTraits_2* defines this common set of requirements.

The requirements of *PolygonTraits_2* are a subset of the kernel requirements. We only list the types and methods which are required and refer to the description of the kernel concept for details.

Types

PolygonTraits_2:: FT

PolygonTraits_2:: Point_2

The point type.

PolygonTraits_2:: Segment_2

The segment type.

PolygonTraits_2:: Construct_segment_2

PolygonTraits_2:: Equal_2

PolygonTraits_2:: Less_xy_2

PolygonTraits_2:: Less_yx_2

PolygonTraits_2:: Compare_x_2

PolygonTraits_2:: Compare_y_2

PolygonTraits_2:: Orientation_2

PolygonTraits_2:: Compute_area_2

Computes the signed area of the oriented triangle defined by 3 *Point_2* passed as arguments.

Creation

A default constructor and copy constructor are required.

Operations

The following functions that create instances of the above predicate object types must exist.

Equal_2 *traits.equal_2_object()*

<i>Less_xy_2</i>	<i>traits.less_xy_2_object()</i>
<i>Less_yx_2</i>	<i>traits.less_yx_2_object()</i>
<i>Compare_y_2</i>	<i>traits.compare_y_2_object()</i>
<i>Compare_x_2</i>	<i>traits.compare_x_2_object()</i>
<i>Orientation_2</i>	<i>traits.orientation_2_object()</i>
<i>Compute_area_2</i>	<i>traits.compute_area_2_object()</i>
<i>Construct_segment_2</i>	<i>traits.construct_segment_2_object()</i>

Has Models

The kernels supplied by CGAL are models of PolygonTraits_2.

See Also

CGAL::Polygon_2<PolygonTraits_2, Container> [page 726](#)

CGAL::Polygon_2<PolygonTraits_2, Container>

Definition

The class *Polygon_2<PolygonTraits_2, Container>* implements polygons. The *Polygon_2<PolygonTraits_2, Container>* is parameterised by a traits class and a container class. The latter can be any class that fulfills the requirements for an STL container. It defaults to the vector class.

```
#include <CGAL/Polygon_2.h>
```

Types

Polygon_2<PolygonTraits_2, Container>:: Traits The traits type.

Polygon_2<PolygonTraits_2, Container>:: Container

The container type.

typedef Traits::FT *FT*;

The number type, which is the *field type* of the points of the polygon.

typedef Traits::Point_2 *Point_2*;

typedef Traits::Segment_2

The point type of the polygon.

Segment_2;

The type of a segment between two points of the polygon.

The following types denote iterators that allow to traverse the vertices and edges of a polygon. Since it is questionable whether a polygon should be viewed as a circular or as a linear data structure both circulators and iterators are defined. The circulators and iterators are non-mutable.¹ The iterator category is in all cases bidirectional, except for *Vertex_iterator*, which has the same iterator category as *Container::iterator*. **N.B.** In fact all of them should have the same iterator category as *Container::iterator*. However, due to compiler problems this is currently not possible.

For vertices we define

Polygon_2<PolygonTraits_2, Container>:: Vertex_iterator

Polygon_2<PolygonTraits_2, Container>:: Vertex_circulator

Their value type is *Point_2*.

For edges we define

Polygon_2<PolygonTraits_2, Container>:: Edge_const_circulator

Polygon_2<PolygonTraits_2, Container>:: Edge_const_iterator

Their value type is *Segment_2*.

¹At least conceptually. The enforcement depends on preprocessor flags.

Creation

Polygon_2<*PolygonTraits_2*, *Container*> *pgn*(*Traits* *p_traits* = *Traits*());

Creates an empty polygon *pgn*.

template <*class InputIterator*>
Polygon_2<*PolygonTraits_2*, *Container*> *pgn*(*InputIterator* *first*,
 InputIterator *last*,
 Traits *p_traits* = *Traits*())

Introduces a polygon *pgn* with vertices from the sequence defined by the range [*first*,*last*). The value type of *InputIterator* must be *Point_2*.

Modifiers

void *pgn.set*(*Vertex_iterator* *pos*, *Point_2* *x*)

Acts as **pos* = *x*, except that that would be illegal because the iterator is not mutable.

Vertex_iterator *pgn.insert*(*Vertex_iterator* *i*, *Point_2* *q*)

Inserts the vertex *q* before *i*. The return value points to the inserted vertex.

<i>template <class InputIterator></i>	<i>pgn.insert(Vertex_iterator i, InputIterator first, InputIterator last)</i>	Inserts the vertices in the range <i>[first, last)</i> before <i>i</i> . The value type of points in the range <i>[first, last)</i> must be <i>Point_2</i> .
<i>void</i>	<i>pgn.push_back(Point_2 q)</i>	Has the same semantics as <i>p.insert(p.vertices_end(), q)</i> .
<i>void</i>	<i>pgn.erase(Vertex_iterator i)</i>	Erases the vertex pointed to by <i>i</i> .
<i>void</i>	<i>pgn.erase(Vertex_iterator first, Vertex_iterator last)</i>	Erases the vertices in the range <i>[first, last)</i> .
<i>void</i>	<i>pgn.clear()</i>	Erases all vertices.
<i>void</i>	<i>pgn.reverse_orientation()</i>	Reverses the orientation of the polygon. The vertex pointed to by <i>p.vertices_begin()</i> remains the same.

Access Functions

The following methods of the class return circulators and iterators that allow to traverse the vertices and edges.

<i>Vertex_iterator</i>	<i>pgn.vertices_begin()</i>	Returns a constant iterator that allows to traverse the vertices of the polygon <i>p</i> .
<i>Vertex_iterator</i>	<i>pgn.vertices_end()</i>	Returns the corresponding past-the-end iterator.
<i>Vertex_circulator</i>	<i>pgn.vertices_circulator()</i>	Returns a mutable circulator that allows to traverse the vertices of the polygon <i>p</i> .
<i>Edge_const_iterator</i>	<i>pgn.edges_begin()</i>	Returns a non-mutable iterator that allows to traverse the edges of the polygon <i>p</i> .
<i>Edge_const_iterator</i>	<i>pgn.edges_end()</i>	Returns the corresponding past-the-end iterator.
<i>Edge_const_circulator</i>	<i>pgn.edges_circulator()</i>	Returns a non-mutable circulator that allows to traverse the edges of the polygon <i>p</i> .

Predicates

<i>bool</i>	<i>pgn.is_simple()</i>	Returns whether <i>p</i> is a simple polygon.
<i>bool</i>	<i>pgn.is_convex()</i>	Returns whether <i>p</i> is convex.
<i>Orientation</i>	<i>pgn.orientation()</i>	Returns the orientation of <i>pgn</i> . If the number of vertices <i>p.size()</i> < 3 then <i>COLLINEAR</i> is returned. <i>Precondition: p.is_simple().</i>
<i>Oriented_side</i>	<i>pgn.oriented_side(Point_2 q)</i>	Returns <i>POSITIVE_SIDE</i> , or <i>NEGATIVE_SIDE</i> , or <i>ON_ORIENTED_BOUNDARY</i> , depending on where point <i>q</i> is. <i>Precondition: p.is_simple().</i>
<i>Bounded_side</i>	<i>pgn.bounded_side(Point_2 q)</i>	Returns the symbolic constant <i>ON_BOUNDED_SIDE</i> , <i>ON_BOUNDARY</i> or <i>ON_UNBOUNDED_SIDE</i> , depending on where point <i>q</i> is. <i>Precondition: p.is_simple().</i>
<i>Bbox_2</i>	<i>pgn.bbox()</i>	Returns the smallest bounding box containing <i>pgn</i> .
<i>Traits::FT</i>	<i>pgn.area()</i>	Returns the signed area of the polygon <i>pgn</i> . This means that the area is positive for counter clockwise polygons and negative for clockwise polygons.
<i>Vertex_iterator</i>	<i>pgn.left_vertex()</i>	Returns the leftmost vertex of the polygon <i>p</i> with the smallest y-coordinate.
<i>Vertex_iterator</i>	<i>pgn.right_vertex()</i>	Returns the rightmost vertex of the polygon <i>p</i> with the largest y-coordinate.
<i>Vertex_iterator</i>	<i>pgn.top_vertex()</i>	Returns topmost vertex of the polygon <i>p</i> with the largest x-coordinate.
<i>Vertex_iterator</i>	<i>pgn.bottom_vertex()</i>	Returns the bottommost vertex of the polygon <i>p</i> with the smallest x-coordinate.

For convenience we provide the following boolean functions:

<i>bool</i>	<i>pgn.is_counterclockwise_oriented()</i>
<i>bool</i>	<i>pgn.is_clockwise_oriented()</i>
<i>bool</i>	<i>pgn.is_collinear_oriented()</i>
<i>bool</i>	<i>pgn.has_on_positive_side(Point_2 q)</i>
<i>bool</i>	<i>pgn.has_on_negative_side(Point_2 q)</i>
<i>bool</i>	<i>pgn.has_on_boundary(Point_2 q)</i>

bool *pgn.has_on_bounded_side(Point_2 q)*
bool *pgn.has_on_unbounded_side(Point_2 q)*

Random access methods

These methods are only available for random access containers.

Point_2 *pgn.vertex(int i)* Returns a (const) reference to the *i*-th vertex.

Point_2 *pgn[int i]* Returns a (const) reference to the *i*-th vertex.

Segment_2 *pgn.edge(int i)* Returns a const reference to the *i*-th edge.

Miscellaneous

int *pgn.size()* Returns the number of vertices of the polygon *pgn*.

bool *pgn.is_empty()* Returns *p.size() == 0*.

Container *pgn.container()* Returns a const reference to the sequence of vertices of the polygon *pgn*.

Globally defined operators

template <class Traits, class Container1, class Container2>

bool *Polygon_2<Traits,Container1> p1 == Polygon_2<Traits,Container2> p2*

Test for equality: two polygons are equal iff there exists a cyclic permutation of the vertices of *p2* such that they are equal to the vertices of *p1*. Note that the template argument *Container* of *p1* and *p2* may be different.

template <class Traits, class Container1, class Container2>

bool *Polygon_2<Traits,Container1> p1 != Polygon_2<Traits,Container2> p2*

Test for inequality.

template <class Transformation, class Traits, class Container>

Polygon_2<Traits,Container>

transform(Transformation t, Polygon_2<Traits,Container> p)

Returns the image of the polygon *p* under the transformation *t*.

Implementation

The methods *is_simple*, *is_convex*, *orientation*, *oriented_side*, *bounded_side*, *bbox*, *area*, *left_vertex*, *right_vertex*, *top_vertex* and *bottom_vertex* are all implemented using the algorithms on sequences of 2D points. See the corresponding global functions for information about which algorithms were used and what complexity they have.

Example

The following code fragment creates a polygon and checks if it is convex.

```
// file: examples/Polygon/Polygon.C

#include <CGAL/Cartesian.h>
#include <CGAL/Polygon_2.h>
#include <iostream>

typedef CGAL::Cartesian<double> K;
typedef K::Point_2 Point;
typedef CGAL::Polygon_2<K> Polygon;
using std::cout; using std::endl;

int main()
{
    Point points[] = { Point(0,0), Point(5.1,0), Point(1,1), Point(0.5,6) };
    Polygon pgn(points, points+4);

    // check if the polygon is simple.
    cout << "The polygon is " <<
        (pgn.is_simple() ? "" : "not ") << "simple." << endl;

    // check if the polygon is convex
    cout << "The polygon is " <<
        (pgn.is_convex() ? "" : "not ") << "convex." << endl;

    return 0;
}
```

CGAL::polygon_area_2

Definition

The function *polygon_area_2* computes the signed area of a polygon.

```
#include <CGAL/Polygon_2_algorithms.h>
```

```
template <class ForwardIterator, class Traits>
typename Traits::FT      polygon_area_2( ForwardIterator first, ForwardIterator last, Traits traits)
```

Computes the signed area of the polygon defined by the range of points *first* ... *last*. The sign is positive for counterclockwise polygons, negative for clockwise polygons. If the polygon is not simple, the area is not well defined.

Requirements

1. *Traits* is a model of the concept *PolygonTraits_2*. Only the following members of this traits class are used:
 - *Compute_area_2* : Computes the signed area of the oriented triangle defined by 3 *Point_2* passed as arguments.
 - *FT*
 - *compute_area_2_object*
2. *ForwardIterator::value_type* should be *Traits::Point_2*,

See Also

PolygonTraits_2 page [724](#)
CGAL::orientation_2 page [722](#)
CGAL::Polygon_2<*PolygonTraits_2*, *Container*> page [726](#)

CGAL::right_vertex_2

Definition

```
#include <CGAL/Polygon_2_algorithms.h>
```

```
template <class ForwardIterator, class Traits>
```

```
ForwardIterator      right_vertex_2( ForwardIterator first, ForwardIterator last, Traits traits)
```

Returns an iterator to the rightmost point from the range $[first, last)$. In case of a tie, the point with the largest y-coordinate is taken.

Requirements

1. *Traits* is a model of the concept `PolygonTraits_2`. In fact, only the members `Less_xy_2` and `less_xy_2_object` are used.
2. `ForwardIterator::value_type` should be `Traits::Point_2`,

See Also

`CGAL::left_vertex_2` page [721](#)
`CGAL::top_vertex_2` page [734](#)
`CGAL::bottom_vertex_2` page [717](#)
`CGAL::Polygon_2<PolygonTraits_2, Container>` page [726](#)

CGAL::top_vertex_2

Definition

```
#include <CGAL/Polygon_2_algorithms.h>
```

```
template <class ForwardIterator, class Traits>
```

```
ForwardIterator top_vertex_2( ForwardIterator first, ForwardIterator last, Traits traits)
```

Returns an iterator to the topmost point from the range *[first,last)*. In case of a tie, the point with the largest *x*-coordinate is taken.

Requirements

1. *Traits* is a model of the concept *PolygonTraits_2*. In fact, only the members *Less_yx_2* and *less_yx_2_object* are used.
2. *ForwardIterator::value_type* should be *Traits::Point_2*,

See Also

CGAL::left_vertex_2 page [721](#)
CGAL::right_vertex_2 page [733](#)
CGAL::bottom_vertex_2 page [717](#)
CGAL::Polygon_2<*PolygonTraits_2*, *Container*> page [726](#)

Chapter 9

2D Polygon Partitioning

Susan Hert

Contents

9.1 Introduction	735
9.2 Monotone Partitioning	735
9.3 Convex Partitioning	736

9.1 Introduction

A *partition* of a polygon P is a set of polygons such that the interiors of the polygons do not intersect and the union of the polygons is equal to the interior of the original polygon P . This chapter describes functions for partitioning planar polygons into two types of subpolygons — y -monotone polygons and convex polygons. The partitions are produced without introducing new (Steiner) vertices.

All the partitioning functions present the same interface to the user. That is, the user provides a pair of input iterators, *first* and *beyond*, an output iterator *result*, and a traits class *traits*. The points in the range $[first, beyond)$ are assumed to define a simple polygon whose vertices are in counterclockwise order. The computed partition polygons, whose vertices are also oriented counterclockwise, are written to the sequence starting at position *result* and the past-the-end iterator for the resulting sequence of polygons is returned. The traits classes for the functions specify the types of the input points and output polygons as well as a few other types and function objects that are required by the various algorithms.

9.2 Monotone Partitioning

A *y-monotone polygon* is a polygon whose vertices v_1, \dots, v_n can be divided into two chains v_1, \dots, v_k and v_k, \dots, v_n, v_1 , such that any horizontal line intersects either chain at most once. For producing a y -monotone partition of a given polygon, the sweep-line algorithm presented in [dBvKOS97] is implemented by the function `y_monotone_partition_2`. This algorithm runs in $O(n \log n)$ time and requires $O(n)$ space. This algorithm does not guarantee a bound on the number of polygons produced with respect to the optimal number.

For checking the validity of the partitions produced by `y_monotone_partition_2`, we provide a function `is_y_monotone_2`, which determines if a sequence of points in 2D defines a y -monotone polygon or not. For examples of the use of these functions, see the corresponding reference pages.

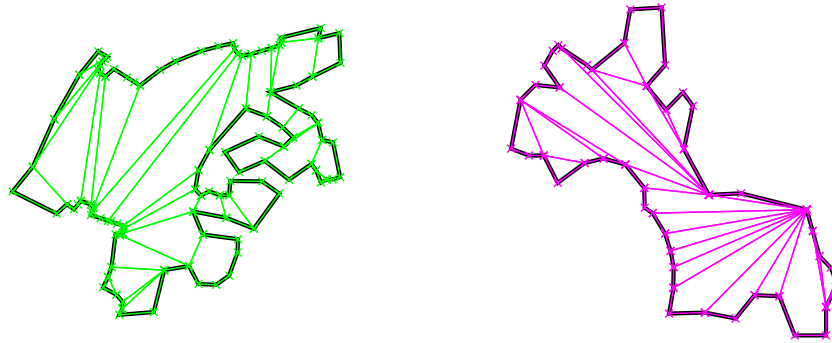


Figure 9.1: Examples of an optimal convex partition (left) and an approximately optimal convex partition (right).

9.3 Convex Partitioning

Three functions are provided for producing convex partitions of polygons. One produces a partition that is optimal in the number of pieces. The other two functions produce approximately optimal convex partitions. Both these functions produce convex decompositions by first decomposing the polygon into simpler polygons; the first uses a triangulation and the second a monotone partition. These two functions both guarantee that they will produce no more than four times the optimal number of convex pieces but they differ in their runtime complexities. Though the triangulation-based approximation algorithm often results in fewer convex pieces, this is not always the case.

An optimal convex partition can be produced using the function *optimal_convex_partition_2*. This function provides an implementation of Greene’s dynamic programming algorithm for optimal partitioning [Gre83]. This algorithm requires $O(n^4)$ time and $O(n^3)$ space in the worst case.

The function *approx_convex_partition_2* implements the simple approximation algorithm of Hertel and Mehlhorn [HM83] that produces a convex partitioning of a polygon from a triangulation by throwing out unnecessary triangulation edges. The triangulation used in this function is one produced by the 2-dimensional constrained triangulation package of CGAL. For a given triangulation, this convex partitioning algorithm requires $O(n)$ time and space to construct a decomposition into no more than four times the optimal number of convex pieces.

The sweep-line approximation algorithm of Greene [Gre83], which, given a monotone partition of a polygon, produces a convex partition in $O(n \log n)$ time and $O(n)$ space, is implemented by the function *greene_approx_convex_partition_2*. The function *y_monotone_partition_2* described in Section 9.2 is used to produce the monotone partition. This algorithm provides the same worst-case approximation guarantee as the algorithm of Hertel and Mehlhorn implemented with *approx_convex_partition_2* but can sometimes produce better results (*i.e.*, convex partitions with fewer pieces).

Examples of the uses of all of these functions are provided with the corresponding reference pages.

2D Polygon Partitioning

Reference Manual

Susan Hert

A *partition* of a polygon is a set of polygons such that the interiors of the polygons do not intersect and the union of the polygons is equal to the interior of the original polygon. Functions are available for partitioning planar polygons into two types of subpolygons — y-monotone polygons and convex polygons.

The function that produces a y-monotone partitioning is based on the algorithm presented in [dBvKOS97] which requires $O(n \log n)$ time and $O(n)$ space for a polygon with n vertices and guarantees nothing about the number of polygons produced with respect to the optimal number. Three functions are provided for producing convex partitionings. Two of these functions produce approximately optimal partitions and one results in an optimal partition, where “optimal” is defined in terms of the number of partition polygons. The two functions that implement approximation algorithms are guaranteed to produce no more than four times the optimal number of convex pieces. The optimal partitioning function provides an implementation of Greene’s dynamic programming algorithm [Gre83], which requires $O(n^4)$ time and $O(n^3)$ space to produce a convex partitioning. One of the approximation algorithms is also due to Greene [Gre83] and requires $O(n \log n)$ time and $O(n)$ space to produce a convex partitioning given a y-monotone partitioning. The other approximation algorithm is a result of Hertel and Mehlhorn [HM83], which requires $O(n)$ time and space to produce a convex partitioning from a triangulation of a polygon. Each of the partitioning functions uses a traits class to supply the primitive types and predicates used by the algorithms.

Assertions

The assertion flags for this package use *PARTITION* in their names (e.g., *CGAL_PARTITION_NO_POSTCONDITIONS*). The precondition checks for the planar polygon partitioning functions are: counterclockwise ordering of the input vertices and simplicity of the polygon these vertices represent. The postcondition checks are: simplicity, counterclockwise orientation, and convexity (or y-monotonicity) of the partition polygons and validity of the partition (i.e., the partition polygons are nonoverlapping and the union of these polygons is the same as the original polygon).

9.4 Classified Reference Pages

Concepts

ConvexPartitionIsValidTraits_2 page 745
IsYMonotoneTraits_2 page 752
OptimalConvexPartitionTraits_2 page 759

PartitionTraits_2	page 765
PartitionIsValidTraits_2	page 763
YMonotonePartitionIsValidTraits_2	page 777
YMonotonePartitionTraits_2	page 778

Function Object Concepts

PolygonIsValid	page 771
----------------------	--------------------------

Classes

<i>CGAL::Partition_is_valid_traits_2</i> <Traits, PolygonIsValid>	page 767
<i>CGAL::Partition_traits_2</i> <R>	page 769

Function Object Classes

<i>CGAL::Is_convex_2</i> <Traits>	page 753
<i>CGAL::Is_vacuously_valid</i> <Traits>	page 754
<i>CGAL::Is_y_monotone_2</i> <Traits>	page 755

Functions

<i>CGAL::approx_convex_partition_2</i>	page 740
<i>CGAL::convex_partition_is_valid_2</i>	page 743
<i>CGAL::greene_approx_convex_partition_2</i>	page 746
<i>CGAL::is_y_monotone_2</i>	page 749
<i>CGAL::optimal_convex_partition_2</i>	page 756
<i>CGAL::partition_is_valid_2</i>	page 761
<i>CGAL::y_monotone_partition_2</i>	page 772
<i>CGAL::y_monotone_partition_is_valid_2</i>	page 775

9.5 Alphabetical List of Reference Pages

<i>approx_convex_partition_2</i>	page 740
<i>ConvexPartitionIsValidTraits_2</i>	page 745
<i>convex_partition_is_valid_2</i>	page 743
<i>greene_approx_convex_partition_2</i>	page 746
<i>IsYMonotoneTraits_2</i>	page 752
<i>Is_convex_2</i> <Traits>	page 753
<i>Is_vacuously_valid</i> <Traits>	page 754

<i>Is_y_monotone_2<Traits></i>	page 755
<i>is_y_monotone_2</i>	page 749
<i>OptimalConvexPartitionTraits_2</i>	page 759
<i>optimal_convex_partition_2</i>	page 756
<i>PartitionIsValidTraits_2</i>	page 763
<i>PartitionTraits_2</i>	page 765
<i>partition_is_valid_2</i>	page 761
<i>Partition_is_valid_traits_2<Traits, PolygonIsValid></i>	page 767
<i>Partition_traits_2<R></i>	page 769
<i>PolygonIsValid</i>	page 771
<i>YMonotonePartitionIsValidTraits_2</i>	page 777
<i>YMonotonePartitionTraits_2</i>	page 778
<i>y_monotone_partition_2</i>	page 772
<i>y_monotone_partition_is_valid_2</i>	page 775

CGAL::approx_convex_partition_2

Definition

Function that produces a set of convex polygons that represent a partitioning of a polygon defined on a sequence of points. The number of convex polygons produced is no more than four times the minimal number.

```
#include <CGAL/partition_2.h>
```

```
template <class InputIterator, class OutputIterator, class Traits>
OutputIterator      approx_convex_partition_2( InputIterator first,
                                              InputIterator beyond,
                                              OutputIterator result,
                                              Traits traits = Default_traits)
```

computes a partition of the polygon defined by the points in the range $[first, beyond)$ into convex polygons. The counterclockwise-oriented partition polygons are written to the sequence starting at position *result*. The past-the-end iterator for the resulting sequence of polygons is returned.

Precondition: The points in the range $[first, beyond)$ define a simple counterclockwise-oriented polygon.

Requirements

1. *Traits* is a model of the concept `PartitionTraits_2` and, for the purposes of checking the postcondition that the partition produced is valid, it should also be a model of the concept `ConvexPartitionIsValidTraits_2`.
2. `OutputIterator::value_type` should be `Traits::Polygon_2`.
3. `InputIterator::value_type` should be `Traits::Point_2`, which should also be the type of the points stored in an object of type `Traits::Polygon_2`.
4. Points in the range $[first, beyond)$ must define a simple polygon whose vertices are oriented counterclockwise.

The default traits class `Default_traits` is `Partition_traits_2`, with the representation type determined by `InputIterator1::value_type`.

See Also

[CGAL::convex_partition_is_valid_2](#) page 743
[CGAL::greene_approx_convex_partition_2](#) page 746
[CGAL::optimal_convex_partition_2](#) page 756
[CGAL::partition_is_valid_2](#) page 761
[CGAL::Partition_is_valid_traits_2<Traits, PolygonIsValid>](#) page 767
[CGAL::y_monotone_partition_2](#) page 772

Implementation

This function implements the algorithm of Hertel and Mehlhorn [HM83] and is based on the class `CGAL::Constrained_triangulation_2`. Given a triangulation of the polygon, the function requires $O(n)$ time and space for a polygon with n vertices.

Example

The following program computes an approximately optimal convex partitioning of a polygon using the default traits class and stores the partition polygons in the list *partition_polys*.

```
// file: examples/Partition_2/approx_convex_partition.C

#include <CGAL/basic.h>
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Partition_traits_2.h>
#include <CGAL/partition_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_polygon_2.h>
#include <cassert>
#include <list>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Partition_traits_2<K> Traits;
typedef Traits::Point_2 Point_2;
typedef Traits::Polygon_2 Polygon_2;
typedef Polygon_2::Vertex_iterator Vertex_iterator;
typedef std::list<Polygon_2> Polygon_list;
typedef CGAL::Creator_uniform_2<int, Point_2> Creator;
typedef CGAL::Random_points_in_square_2<Point_2, Creator> Point_generator;

void make_polygon(Polygon_2& polygon)
{
    polygon.push_back(Point_2(391, 374));
    polygon.push_back(Point_2(240, 431));
    polygon.push_back(Point_2(252, 340));
    polygon.push_back(Point_2(374, 320));
    polygon.push_back(Point_2(289, 214));
    polygon.push_back(Point_2(134, 390));
    polygon.push_back(Point_2( 68, 186));
    polygon.push_back(Point_2(154, 259));
    polygon.push_back(Point_2(161, 107));
    polygon.push_back(Point_2(435, 108));
    polygon.push_back(Point_2(208, 148));
    polygon.push_back(Point_2(295, 160));
    polygon.push_back(Point_2(421, 212));
    polygon.push_back(Point_2(441, 303));
}

int main()
{
```

```

Polygon_2    polygon;
Polygon_list partition_polys;

/*
   CGAL::random_polygon_2(50, std::back_inserter(polygon),
                        Point_generator(100));
*/
make_polygon(polygon);
CGAL::approx_convex_partition_2(polygon.vertices_begin(),
                                polygon.vertices_end(),
                                std::back_inserter(partition_polys));
assert(CGAL::convex_partition_is_valid_2(polygon.vertices_begin(),
                                         polygon.vertices_end(),
                                         partition_polys.begin(),
                                         partition_polys.end()));

return 0;
}

```

CGAL::convex_partition_is_valid_2

Definition

Function that determines if a given set of polygons represents a valid convex partitioning for a given sequence of points that represent a simple, counterclockwise-oriented polygon. A convex partition is valid if the polygons do not overlap, the union of the polygons is the same as the original polygon given by the sequence of points, and if each partition polygon is convex.

```
#include <CGAL/partition_is_valid_2.h>
```

```
template<class InputIterator, class ForwardIterator, class Traits>
bool      convex_partition_is_valid_2( InputIterator point_first,
                                      InputIterator point_beyond,
                                      ForwardIterator poly_first,
                                      ForwardIterator poly_beyond,
                                      Traits traits = Default_traits)
```

determines if the polygons in the range `[poly_first, poly_beyond)` define a valid convex partition of the polygon defined by the points in the range `[point_first, point_beyond)`. The function returns *true* iff the partition is valid and otherwise returns *false*.

Precondition: The points in the range `[point_first, point_beyond)` define a simple, counterclockwise-oriented polygon.

Requirements

1. *Traits* is a model of the concept `ConvexPartitionIsValidTraits_2`.
2. `InputIterator::value_type` should be `Traits::Point_2`, which should also be the type of the points stored in an object of type `Traits::Polygon_2`.
3. `ForwardIterator::value_type` should be `Traits::Polygon_2`.

The default traits class `Default_traits` is `Partition_traits_2`, with the representation type determined by `InputIterator::value_type`.

See Also

`CGAL::approx_convex_partition_2` page 740
`CGAL::greene_approx_convex_partition_2` page 746
`CGAL::optimal_convex_partition_2` page 756
`CGAL::partition_is_valid_2` page 761
`CGAL::is_convex_2`

Implementation

This function calls *partition_is_valid_2* using the function object *Is_convex_2* to determine the convexity of each partition polygon. Thus the time required by this function is $O(n \log n + e \log e)$ where n is the total number of vertices in the partition polygons and e the total number of edges.

Example

See the example presented with the function *approx_convex_partition_2* for an illustration of the use of this function.

_____ *advanced* _____

ConvexPartitionIsValidTraits_2

Definition

Requirements of a traits class used by *convex_partition_is_valid_2* for testing the validity of a convex partition of a polygon.

Types

All types required by the concept `PartitionIsValidTraits_2` are required except the function object type *Is_valid*. The following type is required instead:

ConvexPartitionIsValidTraits_2::Is_convex_2 Model of the concept `PolygonIsValid` that tests if a sequence of points is convex or not.

Creation

Only a copy constructor is required.

ConvexPartitionIsValidTraits_2 traits(& tr);

Operations

The following function that creates an instance of the above predicate object type must exist instead of the function *is_valid_object* required by `PartitionIsValidTraits_2`.

Is_convex_2 traits.is_convex_2_object(t)

Has Models

CGAL::Partition_traits_2<R> page 769

See Also

CGAL::approx_convex_partition_2 page 740
CGAL::greene_approx_convex_partition_2 page 746
CGAL::Is_convex_2<Traits> page 753
CGAL::optimal_convex_partition_2 page 756

CGAL::greene_approx_convex_partition_2

Definition

Function that produces a set of convex polygons that represent a partitioning of a polygon defined on a sequence of points. The number of convex polygons produced is no more than four times the minimal number.

```
#include <CGAL/partition_2.h>
```

```
template <class InputIterator, class OutputIterator, class Traits>
OutputIterator      greene_approx_convex_partition_2( InputIterator first,
                                                    InputIterator beyond,
                                                    OutputIterator result,
                                                    Traits traits = Default_traits)
```

computes a partition of the polygon defined by the points in the range $[first, beyond)$ into convex polygons. The counterclockwise-oriented partition polygons are written to the sequence starting at position *result*. The past-the-end iterator for the resulting sequence of polygons is returned.

Precondition: The points in the range $[first, beyond)$ define a simple, counterclockwise-oriented polygon.

Requirements

1. *Traits* is a model of the concepts *PartitionTraits_2* and *YMonotonePartitionTraits_2*. For the purpose of checking the validity of the y-monotone partition produced as a preprocessing step for the convex partitioning, it must also be a model of *YMonotonePartitionIsValidTraits_2*. For the purpose of checking the postcondition that the convex partition is valid, *Traits* must also be a model of *ConvexPartitionIsValidTraits_2*.
2. *OutputIterator::value_type* is equivalent to *Traits::Polygon_2*.
3. *InputIterator::value_type* is equivalent to *Traits::Point_2*, which should also be equivalent to the type of the points stored in an object of type *Traits::Polygon_2*.

The default traits class *Default_traits* is *Partition_traits_2*, with the representation type determined by *InputIterator::value_type*.

See Also

CGAL::approx_convex_partition_2 [page 740](#)
CGAL::convex_partition_is_valid_2 [page 743](#)
CGAL::optimal_convex_partition_2 [page 756](#)
CGAL::partition_is_valid_2 [page 761](#)
CGAL::y_monotone_partition_2 [page 772](#)

Implementation

This function implements the approximation algorithm of Greene [Gre83] and requires $O(n \log n)$ time and $O(n)$ space to produce a convex partitioning given a y -monotone partitioning of a polygon with n vertices. The function `y_monotone_partition_2` is used to produce the monotone partition.

Example

The following program computes an approximately optimal convex partitioning of a polygon using the default traits class and stores the partition polygons in the list `partition_polys`.

```
// file: examples/Partition_2/greene_approx_convex_partition_2.C

#include <CGAL/basic.h>
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Partition_traits_2.h>
#include <CGAL/partition_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_polygon_2.h>
#include <cassert>
#include <list>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Partition_traits_2<K> Traits;
typedef Traits::Point_2 Point_2;
typedef Traits::Polygon_2 Polygon_2;
typedef Polygon_2::Vertex_iterator Vertex_iterator;
typedef std::list<Polygon_2> Polygon_list;
typedef CGAL::Creator_uniform_2<int, Point_2> Creator;
typedef CGAL::Random_points_in_square_2< Point_2, Creator > Point_generator;

void make_polygon(Polygon_2& polygon)
{
    polygon.push_back(Point_2(391, 374));
    polygon.push_back(Point_2(240, 431));
    polygon.push_back(Point_2(252, 340));
    polygon.push_back(Point_2(374, 320));
    polygon.push_back(Point_2(289, 214));
    polygon.push_back(Point_2(134, 390));
    polygon.push_back(Point_2( 68, 186));
    polygon.push_back(Point_2(154, 259));
    polygon.push_back(Point_2(161, 107));
    polygon.push_back(Point_2(435, 108));
    polygon.push_back(Point_2(208, 148));
    polygon.push_back(Point_2(295, 160));
    polygon.push_back(Point_2(421, 212));
    polygon.push_back(Point_2(441, 303));
}

int main()
{
    Polygon_2    polygon;
```

```

Polygon_list partition_polys;
Traits        partition_traits;

/*
   CGAL::random_polygon_2(50, std::back_inserter(polygon),
                        Point_generator(100));
*/
make_polygon(polygon);
CGAL::greene_approx_convex_partition_2(polygon.vertices_begin(),
                                       polygon.vertices_end(),
                                       std::back_inserter(partition_polys),
                                       partition_traits);
assert(CGAL::convex_partition_is_valid_2(polygon.vertices_begin(),
                                       polygon.vertices_end(),
                                       partition_polys.begin(),
                                       partition_polys.end(),
                                       partition_traits));

return 0;
}

```

CGAL::is_y_monotone_2

Definition

Function for testing the y-monotonicity of a sequence of points.

```
#include <CGAL/is_y_monotone_2.h>
```

```
template<class InputIterator, class Traits>
bool is_y_monotone_2( InputIterator first, InputIterator beyond, Traits traits)
```

Determines if the sequence of points in the range *[first, beyond)* define a y-monotone polygon or not. If so, the function returns *true*, otherwise it returns *false*.

Requirements

1. *Traits* is a model of the concept *IsYMonotoneTraits_2*.
2. *InputIterator::value_type* should be *Traits::Point_2*.

The default traits class *Default_traits* is the kernel in which the type *InputIterator::value_type* is defined.

See Also

CGAL::Is_y_monotone_2<Traits> page [755](#)
CGAL::y_monotone_partition_2 page [772](#)
CGAL::y_monotone_partition_is_valid_2 page [775](#)

Implementation

This function requires $O(n)$ time for a polygon with n vertices.

Example

The following program computes a y-monotone partitioning of a polygon using the default traits class and stores the partition polygons in the list *partition_polys*. It then asserts that each of the partition polygons is, in fact, a y-monotone polygon and that the partition is valid. (Note that the assertions are superfluous unless the postcondition checking done by *y_monotone_partition_2* has been turned off during compilation.)

```
// file: examples/Partition_2/y_monotone_partition_2.C

#include <CGAL/basic.h>
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Partition_traits_2.h>
#include <CGAL/partition_2.h>
```

```

#include <CGAL/point_generators_2.h>
#include <CGAL/random_polygon_2.h>
#include <cassert>
#include <list>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Partition_traits_2<K> Traits;
typedef Traits::Point_2 Point_2;
typedef Traits::Polygon_2 Polygon_2;
typedef std::list<Polygon_2> Polygon_list;
typedef CGAL::Creator_uniform_2<int, Point_2> Creator;
typedef CGAL::Random_points_in_square_2<Point_2, Creator> Point_generator;

void make_polygon(Polygon_2& polygon)
{
    polygon.push_back(Point_2(391, 374));
    polygon.push_back(Point_2(240, 431));
    polygon.push_back(Point_2(252, 340));
    polygon.push_back(Point_2(374, 320));
    polygon.push_back(Point_2(289, 214));
    polygon.push_back(Point_2(134, 390));
    polygon.push_back(Point_2( 68, 186));
    polygon.push_back(Point_2(154, 259));
    polygon.push_back(Point_2(161, 107));
    polygon.push_back(Point_2(435, 108));
    polygon.push_back(Point_2(208, 148));
    polygon.push_back(Point_2(295, 160));
    polygon.push_back(Point_2(421, 212));
    polygon.push_back(Point_2(441, 303));
}

int main( )
{
    Polygon_2    polygon;
    Polygon_list partition_polys;

    /*
    CGAL::random_polygon_2(50, std::back_inserter(polygon),
                          Point_generator(100));
    */

    make_polygon(polygon);
    CGAL::y_monotone_partition_2(polygon.vertices_begin(),
                                polygon.vertices_end(),
                                std::back_inserter(partition_polys));

    std::list<Polygon_2>::const_iterator poly_it;
    for (poly_it = partition_polys.begin(); poly_it != partition_polys.end();
         poly_it++)
    {
        assert(CGAL::is_y_monotone_2((*poly_it).vertices_begin(),
                                     (*poly_it).vertices_end()));
    }
}

```

```
    assert(CGAL::partition_is_valid_2(polygon.vertices_begin(),
                                      polygon.vertices_end(),
                                      partition_polys.begin(),
                                      partition_polys.end()));

    return 0;
}
```

IsYMonotoneTraits_2

Definition

Requirements of a traits class to be used with the function *is_y_monotone_2* that tests whether a sequence of 2D points defines a y-monotone polygon or not.

Types

The following two types are required:

IsYMonotoneTraits_2:: Point_2

The point type of the polygon vertices.

IsYMonotoneTraits_2:: Less_yx_2

Predicate object type that compares *Point_2*s lexicographically. Must provide *bool operator()(Point_2 p, Point_2 q)* where *true* is returned iff $p <_{xy} q$. We have $p <_{xy} q$, iff $p_x < q_x$ or $p_x = q_x$ and $p_y < q_y$, where p_x and p_y denote x and y coordinate of point p resp.

Creation

Only a copy constructor is required.

IsYMonotoneTraits_2 traits(& tr);

Operations

The following function that creates an instance of the above predicate object type must exist:

Less_yx_2 traits.less_yx_2_object()

Has Models

CGAL::Partition_traits_2<R> page [769](#)
CGAL::Kernel_traits_2

See Also

CGAL::Is_y_monotone_2<Traits> page [755](#)
CGAL::y_monotone_partition_2 page [772](#)
CGAL::y_monotone_partition_is_valid_2 page [775](#)

CGAL::Is_convex_2<Traits>

Definition

Function object class for testing if a sequence of points represents a convex polygon or not.

`#include <CGAL/polygon_function_objects.h>`

Is Model for the Concepts

PolygonIsValid page [771](#)

Creation

`Is_convex_2<Traits> f(Traits t);` *Traits* satisfies the requiriements of the function `is_convex_2`

Operations

`template<class InputIterator>`
`bool f(InputIterator first, InputIterator beyond)`

returns *true* iff the points of type *Triats::Point_2* in the range `[first,beyond)` define a convex polygon.

See Also

`CGAL::convex_partition_is_valid_2` page [743](#)
`CGAL::Partition_is_valid_traits_2<Traits, PolygonIsValid>` page [767](#)

Implementation

This test requires $O(n)$ time for a polygon with n vertices.

CGAL::Is_vacuously_valid<Traits>

Definition

Function object class that indicates all sequences of points are valid.

```
#include <CGAL/polygon_function_objects.h>
```

Is Model for the Concepts

PolygonIsValid page [771](#)

Creation

```
Is_vacuously_valid<Traits> f( Traits t);
```

Operations

```
template<class InputIterator>
bool      f( InputIterator first, InputIterator beyond)      returns true.
```

See Also

CGAL::partition_is_valid_2 page [761](#)
CGAL::Partition_is_valid_traits_2<Traits, PolygonIsValid> page [767](#)

Implementation

This test requires $O(1)$ time.

CGAL::Is_y_monotone_2<Traits>

Definition

Function object class that tests whether a sequence of points represents a y-monotone polygon or not.

#include <CGAL/polygon_function_objects.h>

Is Model for the Concepts

PolygonIsValid page [771](#)

Creation

Is_y_monotone_2<Traits> f(Traits t); *Traits* is a model of the concept IsYMonotoneTraits_2

Operations

template<class InputIterator>
bool f(InputIterator first, InputIterator beyond)

returns *true* iff the points of type *Traits::Point_2* in the range *[first,beyond)* define a y-monotone polygon.

See Also

CGAL::convex_partition_is_valid_2 page [743](#)
CGAL::Partition_is_valid_traits_2<Traits, PolygonIsValid> page [767](#)

Implementation

This test requires $O(n)$ time for a polygon with n vertices.

CGAL::optimal_convex_partition_2

Function that produces a set of convex polygons that represent a partitioning of a polygon defined on a sequence of points. The number of convex polygons produced is minimal.

```
#include <CGAL/partition_2.h>
```

```
template <class InputIterator, class OutputIterator, class Traits>
OutputIterator      optimal_convex_partition_2( InputIterator first,
                                                InputIterator beyond,
                                                OutputIterator result,
                                                Traits traits = Default_traits)
```

computes a partition of the polygon defined by the points in the range $[first, beyond)$ into convex polygons. The counterclockwise-oriented partition polygons are written to the sequence starting at position *result*. The past-the-end iterator for the resulting sequence of polygons is returned.

Precondition: The points in the range $[first, beyond)$ define a simple, counterclockwise-oriented polygon.

Requirements

1. *Traits* is a model of the concept `OptimalConvexPartitionTraits_2`. For the purposes of checking the post-condition that the partition is valid, *Traits* should also be a model of `ConvexPartitionIsValidTraits_2`.
2. `OutputIterator::value_type` should be `Traits::Polygon_2`.
3. `InputIterator::value_type` should be `Traits::Point_2`, which should also be the type of the points stored in an object of type `Traits::Polygon_2`.

The default traits class `Default_traits` is `Partition_traits_2`, with the representation type determined by `InputIterator::value_type`.

See Also

`CGAL::approx_convex_partition_2` page [740](#)
`CGAL::convex_partition_is_valid_2` page [743](#)
`CGAL::greene_approx_convex_partition_2` page [746](#)
`CGAL::partition_is_valid_2` page [761](#)
`CGAL::Partition_is_valid_traits_2<Traits, PolygonIsValid>` page [767](#)

Implementation

This function implements the dynamic programming algorithm of Greene [[Gre83](#)], which requires $O(n^4)$ time and $O(n^3)$ space to produce a partitioning of a polygon with n vertices.

Example

The following program computes an optimal convex partitioning of a polygon using the default traits class and stores the partition polygons in the list *partition_polys*. It then asserts that the partition produced is valid. The traits class used for testing the validity is derived from the traits class used to produce the partition with the function object class *CGAL::Is_convex_2* used to define the required *Is_valid* type. (Note that this assertion is superfluous unless the postcondition checking for *optimal_convex_partition_2* has been turned off.)

```
// file: examples/Partition_2/optimal_convex_partition_2.C

#include <CGAL/basic.h>
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Partition_traits_2.h>
#include <CGAL/Partition_is_valid_traits_2.h>
#include <CGAL/polygon_function_objects.h>
#include <CGAL/partition_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_polygon_2.h>
#include <cassert>
#include <list>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Partition_traits_2<K> Traits;
typedef CGAL::Is_convex_2<Traits> Is_convex_2;
typedef Traits::Polygon_2 Polygon_2;
typedef Traits::Point_2 Point_2;
typedef Polygon_2::Vertex_const_iterator Vertex_iterator;
typedef std::list<Polygon_2> Polygon_list;
typedef CGAL::Partition_is_valid_traits_2<Traits, Is_convex_2> Validity_traits;

typedef CGAL::Creator_uniform_2<int, Point_2> Creator;
typedef CGAL::Random_points_in_square_2<Point_2, Creator> Point_generator;

void make_polygon(Polygon_2& polygon)
{
    polygon.push_back(Point_2(391, 374));
    polygon.push_back(Point_2(240, 431));
    polygon.push_back(Point_2(252, 340));
    polygon.push_back(Point_2(374, 320));
    polygon.push_back(Point_2(289, 214));
    polygon.push_back(Point_2(134, 390));
    polygon.push_back(Point_2( 68, 186));
    polygon.push_back(Point_2(154, 259));
    polygon.push_back(Point_2(161, 107));
    polygon.push_back(Point_2(435, 108));
    polygon.push_back(Point_2(208, 148));
    polygon.push_back(Point_2(295, 160));
    polygon.push_back(Point_2(421, 212));
    polygon.push_back(Point_2(441, 303));
}

int main()
{
```

```

Polygon_2          polygon;
Polygon_list       partition_polys;
Traits            partition_traits;
Validity_traits    validity_traits;

/*
   CGAL::random_polygon_2(50, std::back_inserter(polygon),
                        Point_generator(100));
*/
make_polygon(polygon);
CGAL::optimal_convex_partition_2(polygon.vertices_begin(),
                                polygon.vertices_end(),
                                std::back_inserter(partition_polys),
                                partition_traits);
assert(CGAL::partition_is_valid_2(polygon.vertices_begin(),
                                polygon.vertices_end(),
                                partition_polys.begin(),
                                partition_polys.end(),
                                validity_traits));

return 0;
}

```

OptimalConvexPartitionTraits_2

Definition

Requirements of a traits class to be used with the function *optimal_convex_partition_2* that computes an optimal convex partition of a polygon.

Generalizes

PartitionTraits_2 page 765

Types

In addition to the types listed with the concept PartitionTraits_2, the following types are required:

OptimalConvexPartitionTraits_2:: Segment_2 A segment type

OptimalConvexPartitionTraits_2:: Ray_2 A ray type

OptimalConvexPartitionTraits_2:: Object_2 A general object type that can be either a point or a segment

OptimalConvexPartitionTraits_2:: Construct_segment_2

Function object type that provides *Segment_2* *operator()(Point_2 p, Point_2 q)*, which constructs and returns the segment defined by the points *p* and *q*.

OptimalConvexPartitionTraits_2:: Construct_ray_2

Function object type that provides *Ray_2* *operator()(Point_2 p, Point_2 q)*, which constructs and returns the ray from point *p* through point *q*.

OptimalConvexPartitionTraits_2:: Collinear_are_ordered_along_line_2

Predicate object type that determines orderings of *Point_2*s on a line. Must provide *bool operator()(Point_2 p, Point_2 q, Point_2 r)* that returns *true*, iff *q* lies between *p* and *r* and *p*, *q*, and *r* satisfy the precondition that they are collinear.

OptimalConvexPartitionTraits_2:: Are_strictly_ordered_along_line_2

Predicate object type that determines orderings of *Point_2*s. Must provide *bool operator()(Point_2 p, Point_2 q, Point_2 r)* that returns *true*, iff the three points are collinear and *q* lies strictly between *p* and *r*. Note that *false* should be returned if *q==p* or *q==r*.

<i>OptimalConvexPartitionTraits_2:: Intersect_2</i>	Function object type that provides <i>Object_2 operator()(Segment_2 s1, Segment_2 s2)</i> that returns the intersection of two segments (which may be either a segment or a point).
<i>OptimalConvexPartitionTraits_2:: Assign_2</i>	Function object type that provides <i>bool operator()(Segment_2 s1, Object_2 o)</i> that returns <i>true</i> if <i>o</i> is a segment and assigns the value of <i>o</i> to <i>s1</i> ; returns <i>false</i> otherwise.

Creation

Only a copy constructor is required.

OptimalConvexPartitionTraits_2 traits(& tr);

Operations

In addition to the functions required by *PartitionTraits_2*, the following functions that create instances of the above function object types must exist:

<i>Collinear_are_ordered_along_line_2</i>	<i>traits.collinear_are_ordered_along_line_2_object()</i>
<i>Construct_segment_2</i>	<i>traits.construct_segment_2_object()</i>
<i>Construct_ray_2</i>	<i>traits.construct_ray_2_object()</i>
<i>Are_strictly_ordered_along_line_2</i>	<i>traits.are_strictly_ordered_along_line_2_object()</i>

Has Models

CGAL::Partition_traits_2<R> page [769](#)

See Also

CGAL::convex_partition_is_valid_2 page [743](#)
CGAL::Partition_is_valid_traits_2<Traits, PolygonIsValid> page [767](#)

CGAL::partition_is_valid_2

Definition

Function that determines if a given set of polygons represents a valid partition for a given sequence of points that define a simple, counterclockwise-oriented polygon. A valid partition is one in which the polygons are nonoverlapping and the union of the polygons is the same as the original polygon.

```
#include <CGAL/partition_is_valid_2.h>
```

```
template<class InputIterator, class ForwardIterator, class Traits>
bool partition_is_valid_2( InputIterator point_first,
                          InputIterator point_beyond,
                          ForwardIterator poly_first,
                          ForwardIterator poly_beyond,
                          Traits traits = Default_traits)
```

returns *true* iff the polygons in the range $[poly_first, poly_beyond)$ define a valid partition of the polygon defined by the points in the range $[point_first, point_beyond)$ and *false* otherwise. Each polygon must also satisfy the property tested by *Traits::Is_valid()*.

Precondition: Points in the range $[point_first, point_beyond)$ define a simple, counterclockwise-oriented polygon.

Requirements

1. *Traits* is a model of the concept *PartitionIsValidTraits_2* and the concept defining the requirements for the validity test implemented by *Traits::Is_valid()*.
2. *InputIterator::value_type* should be *Traits::Point_2*, which should also be the type of the points stored in an object of type *Traits::Polygon_2*.
3. *ForwardIterator::value_type* should be *Traits::Polygon_2*.

The default traits class *Default_traits* is *Partition_traits_2*, with the representation type determined by *InputIterator::value_type*.

See Also

CGAL::approx_convex_partition_2 page 740
CGAL::greene_approx_convex_partition_2 page 746
CGAL::is_y_monotone_2 page 749
CGAL::optimal_convex_partition_2 page 756
CGAL::Partition_is_valid_traits_2<*Traits*, *PolygonIsValid*> page 767
CGAL::y_monotone_partition_2 page 772
CGAL::is_convex_2

Implementation

This function requires $O(n \log n + e \log e + \sum_{i=1}^p m_i)$ where n is the total number of vertices of the p partition polygons, e is the total number of edges of the partition polygons and m_i is the time required by *Traits::Is_valid()* to test if partition polygon p_i is valid.

Example

See the example presented with the function *optimal_convex_partition_2* for an illustration of the use of this function.

_____ *advanced* _____

PartitionIsValidTraits_2

Definition

Requirements of a traits class that is used by *partition_is_valid_2*, *convex_partition_is_valid_2*, and *y_monotone_partition_is_valid_2* for testing if a given set of polygons are nonoverlapping and if their union is a polygon that is the same as a polygon represented by a given sequence of points. Note that the traits class for *partition_is_valid_2* may have to satisfy additional requirements if each partition polygon is to be tested for having a particular property; see, for example, the descriptions of the function *is_convex_2* and the concept *YMonotonePartitionTraits_2* for the additional requirements for testing for convexity and y-monotonicity, respectively.

Types

<i>PartitionIsValidTraits_2:: Point_2</i>	The point type on which the partitioning algorithm operates.
<i>PartitionIsValidTraits_2:: Polygon_2</i>	The polygon type created by the partitioning function. This type should provide a nested type <i>Vertex_const_iterator</i> that is the type of the non-mutable iterator over the polygon vertices.
<i>PartitionIsValidTraits_2:: Is_valid</i>	A model of the concept <i>PolygonIsValid</i>
<i>PartitionIsValidTraits_2:: Less_xy_2</i>	Predicate object type that compares <i>Point_2</i> s lexicographically. Must provide <i>bool operator()(Point_2 p, Point_2 q)</i> where <i>true</i> is returned iff $p <_{xy} q$. We have $p <_{xy} q$, iff $p_x < q_x$ or $p_x = q_x$ and $p_y < q_y$, where p_x and p_y denote the x and y coordinates of point p , respectively.
<i>PartitionIsValidTraits_2:: Left_turn_2</i>	Predicate object type that provides <i>bool operator()(Point_2 p, Point_2 q, Point_2 r)</i> , which returns <i>true</i> iff r lies to the left of the oriented line through p and q .
<i>PartitionIsValidTraits_2:: Orientation_2</i>	Predicate object type that provides <i>CGAL::Orientation operator()(Point_2 p, Point_2 q, Point_2 r)</i> that returns <i>CGAL::LEFT_TURN</i> , if r lies to the left of the oriented line l defined by p and q , returns <i>CGAL::RIGHT_TURN</i> if r lies to the right of l , and returns <i>CGAL::COLLINEAR</i> if r lies on l .

Creation

Only a copy constructor is required.

```
PartitionIsValidTraits_2 traits( & tr);
```

Operations

The following functions that create instances of the above predicate object types must exist.

Orientation_2 *traits.is_valid_object()*

Less_xy_2 *traits.less_xy_2_object()*

Left_turn_2 *traits.left_turn_2_object()*

Orientation_2 *traits.orientation_2_object()*

Has Models

CGAL::Partition_is_valid_traits_2<*Traits*, *PolygonIsValid*> page [767](#)

See Also

CGAL::approx_convex_partition_2 page [740](#)

CGAL::greene_approx_convex_partition_2 page [746](#)

CGAL::optimal_convex_partition_2 page [756](#)

CGAL::y_monotone_partition_2 page [772](#)

└────────── *advanced* ─────────┘

PartitionTraits_2

Definition

The polygon partitioning functions are each parameterized by a traits class that defines the primitives used in the algorithms. Many requirements are common to all traits classes. The concept `PartitionTraits_2` defines this common set of requirements.

Types

<code>PartitionTraits_2:: Point_2</code>	The point type on which the partitioning algorithm operates.
<code>PartitionTraits_2:: Polygon_2</code>	The polygon type to be created by the partitioning algorithm. For testing the validity postcondition of the partition, this type should provide a nested type <i>Vertex_const_iterator</i> that is the type of the iterator over the polygon vertices and member functions <i>Vertex_const_iterator vertices_begin()</i> and <i>Vertex_const_iterator vertices_end()</i> .
<code>PartitionTraits_2:: Less_xy_2</code>	Predicate object type that compares <i>Point_2</i> s lexicographically. Must provide <i>bool operator()(Point_2 p, Point_2 q)</i> where <i>true</i> is returned iff $p <_{xy} q$. We have $p <_{xy} q$, iff $p_x < q_x$ or $p_x = q_x$ and $p_y < q_y$, where p_x and p_y denote the x and y coordinates of point p , respectively.
<code>PartitionTraits_2:: Less_yx_2</code>	Same as <i>Less_xy_2</i> with the roles of x and y interchanged.
<code>PartitionTraits_2:: Left_turn_2</code>	Predicate object type that provides <i>bool operator()(Point_2 p, Point_2 q, Point_2 r)</i> , which returns <i>true</i> iff r lies to the left of the oriented line through p and q .
<code>PartitionTraits_2:: Orientation_2</code>	Predicate object type that provides <i>CGAL::Orientation operator()(Point_2 p, Point_2 q, Point_2 r)</i> that returns <i>CGAL::LEFT_TURN</i> , if r lies to the left of the oriented line l defined by p and q , returns <i>CGAL::RIGHT_TURN</i> if r lies to the right of l , and returns <i>CGAL::COLLINEAR</i> if r lies on l .
<code>PartitionTraits_2:: Compare_y_2</code>	Predicate object type that provides <i>CGAL::Comparison_result operator()(Point_2 p, Point_2 q)</i> to compare the y values of two points. The operator must return <i>CGAL::SMALLER</i> if $p_y < q_y$, <i>CGAL::LARGER</i> if $p_y > q_y$ and <i>CGAL::EQUAL</i> if $p_y = q_y$.
<code>PartitionTraits_2:: Compare_x_2</code>	The same as <i>Compare_y_2</i> , except that x coordinates are compared instead of y .

Creation

A copy constructor and default constructor are required.

```
PartitionTraits_2 traits;
```

```
PartitionTraits_2 traits( & tr);
```

Operations

The following functions that create instances of the above predicate object types must exist.

```
Less_yx_2          traits.less_yx_2_object()
```

```
Less_xy_2          traits.less_xy_2_object()
```

```
Left_turn_2       traits.left_turn_2_object()
```

```
Orientation_2     traits.orientation_2_object()
```

```
Compare_y_2       traits.compare_y_2_object()
```

```
Compare_x_2       traits.compare_x_2_object()
```

Has Models

CGAL::Partition_traits_2<*R*> page [769](#)

See Also

CGAL::approx_convex_partition_2 page [740](#)

CGAL::greene_approx_convex_partition_2 page [746](#)

CGAL::optimal_convex_partition_2 page [756](#)

CGAL::y_monotone_partition_2 page [772](#)

CGAL::Partition_is_valid_traits_2<Traits, PolygonIsValid>

Definition

Class that derives a traits class for *partition_is_valid_2* from a given traits class by defining the validity testing function object in terms of a supplied template parameter.

```
#include <CGAL/Partition_is_valid_traits_2.h>
```

Inherits From

Traits

Is Model for the Concepts

PartitionIsValidTraits_2

Types

```
typedef PolygonIsValid      Is_valid;
typedef Traits::Point_2     Point_2;
typedef Traits::Polygon_2   Polygon_2;
typedef Traits::Less_xy_2   Less_xy_2;
typedef Traits::Left_turn_2 Left_turn_2;
typedef Traits::Orientation_2 Orientation_2;
```

Operations

The constructors and member functions for creating instances of the above types are inherited from *Traits*. In addition, the following member function is defined:

<i>Is_valid</i>	<i>traits.is_valid_object(Traits traits)</i>
-----------------	-----------------------------------------------

function returning an instance of *Is_valid*

See Also

<i>CGAL::Is_convex_2<Traits></i>	page 753
<i>CGAL::Is_vacuously_valid<Traits></i>	page 754
<i>CGAL::Is_y_monotone_2<Traits></i>	page 755
<i>CGAL::Partition_traits_2<R></i>	page 769

Example

See the example presented with the function *optimal_convex_partition_2* for an illustration of the use of this traits class.

_____ *advanced* _____

CGAL::Partition_traits_2<R>

Definition

Traits class that can be used with all the 2-dimensional polygon partitioning algorithms. It is parameterized by a representation class *R*.

```
#include <CGAL/Partition_traits_2.h>
```

Is Model for the Concepts

ConvexPartitionIsValidTraits_2	page 745
IsYMonotoneTraits_2	page 752
OptimalConvexPartitionTraits_2	page 759
PartitionTraits_2	page 765
YMonotonePartitionIsValidTraits_2	page 777
YMonotonePartitionTraits_2	page 778

Types

<i>typedef</i> R::Line_2	Line_2;
<i>typedef</i> R::Segment_2	Segment_2;
<i>typedef</i> R::Ray_2	Ray_2;
<i>typedef</i> R::Less_yx_2	Less_yx_2;
<i>typedef</i> R::Less_xy_2	Less_xy_2;
<i>typedef</i> R::Left_turn_2	Left_turn_2;
<i>typedef</i> R::Orientation_2	Orientation_2;
<i>typedef</i> R::Compare_y_2	Compare_y_2;
<i>typedef</i> R::Compare_x_2	Compare_x_2;
<i>typedef</i> R::Construct_line_2	Construct_line_2;
<i>typedef</i> R::Construct_ray_2	Construct_ray_2;
<i>typedef</i> R::Construct_segment_2	Construct_segment_2;
<i>typedef</i> R::Collinear_are_ordered_along_line_2	Collinear_are_ordered_along_line_2;
<i>typedef</i> R::Are_strictly_ordered_along_line_2	Are_strictly_ordered_along_line_2;
<i>typedef</i> CGAL::Polygon_traits_2<R>	Poly_Traits;
<i>typedef</i> Poly_Traits::Point_2	Point_2;
<i>typedef</i> std::list<Point_2>	Container;
<i>typedef</i> CGAL::Polygon_2<Poly_Traits, Container>	Polygon_2;
<i>typedef</i> R::Less_xy_2	Less_xy;
<i>typedef</i> Poly_Traits::Vector_2	Vector_2;
<i>typedef</i> R::FT	FT;
<i>typedef</i> Partition_traits_2<R>	Self;
<i>typedef</i> CGAL::Is_convex_2<Self>	Is_convex_2;
<i>typedef</i> CGAL::Is_y_monotone_2<Self>	Is_y_monotone_2;

Creation

A default constructor and copy constructor are defined.

Partition_traits_2<*R*> *traits*;

Partition_traits_2<*R*> *traits*(*Partition_traits_2*& *tr*);

Operations

For each predicate object type *Pred_object_type* listed above (i.e., *Less_yx_2*, *Less_xy_2*, *Left_turn_2*, *Orientation_2*, *Compare_y_2*, *Compare_x_2*, *Construct_line_2*, *Construct_ray_2*, *Construct_segment_2*, *Collinear_are_ordered_along_line_2*, *Are_strictly_ordered_along_line_2*, *Is_convex_2*, *Is_y_monotone_2*) there is a corresponding function of the following form defined:

Pred_object_type *traits.pred_object_type_object*() Returns an instance of *Pred_object_type*.

See Also

CGAL::approx_convex_partition_2 page [740](#)
CGAL::convex_partition_is_valid_2 page [743](#)
CGAL::greene_approx_convex_partition_2 page [746](#)
CGAL::optimal_convex_partition_2 page [756](#)
CGAL::partition_is_valid_2 page [761](#)
CGAL::Partition_is_valid_traits_2<*Traits*, *PolygonIsValid*> page [767](#)
CGAL::y_monotone_partition_2 page [772](#)
CGAL::y_monotone_partition_is_valid_2 page [775](#)

PolygonIsValid

Definition

Function object that determines if a sequence of points represents a valid partition polygon or not, where “valid” can assume any of several meanings (*e.g.*, convex or *y*-monotone).

Creation

PolygonIsValid *f*(*Traits* *t*);

Traits is a model of the concept required by the function that checks for validity of the polygon.

Operations

```
template<class InputIterator>
bool f( InputIterator first, InputIterator beyond)
```

returns *true* iff the points of type *Traits::Point_2* in the range [*first*,*beyond*) define a valid polygon.

Has Models

CGAL::Is_convex_2<*Traits*> page [753](#)
CGAL::Is_y_monotone_2<*Traits*> page [755](#)

See Also

CGAL::approx_convex_partition_2 page [740](#)
CGAL::convex_partition_is_valid_2 page [743](#)
CGAL::greene_approx_convex_partition_2 page [746](#)
CGAL::optimal_convex_partition_2 page [756](#)
CGAL::partition_is_valid_2 page [761](#)
CGAL::y_monotone_partition_2 page [772](#)
CGAL::y_monotone_partition_is_valid_2 page [775](#)

CGAL::y_monotone_partition_2

Definition

Function that produces a set of y-monotone polygons that represent a partitioning of a polygon defined on a sequence of points.

```
#include <CGAL/partition_2.h>
```

```
template <class InputIterator, class OutputIterator, class Traits>
OutputIterator      y_monotone_partition_2( InputIterator first,
                                           InputIterator beyond,
                                           OutputIterator result,
                                           Traits traits = Default_traits)
```

computes a partition of the polygon defined by the points in the range $[first, beyond)$ into y-monotone polygons. The counterclockwise-oriented partition polygons are written to the sequence starting at position *result*. The past-the-end iterator for the resulting sequence of polygons is returned.

Precondition: The points in the range $[first, beyond)$ define a simple, counterclockwise-oriented polygon.

Requirements

1. *Traits* is a model of the concept *YMonotonePartitionTraits_2* and, for the purposes of checking the post-condition that the partition is valid, it should also be a model of *YMonotonePartitionIsValidTraits_2*.
2. *OutputIterator::value_type* should be *Traits::Polygon_2*.
3. *InputIterator::value_type* should be *Traits::Point_2*, which should also be the type of the points stored in an object of type *Traits::Polygon_2*.

The default traits class *Default_traits* is *Partition_traits_2*, with the representation type determined by *InputIterator::value_type*.

See Also

CGAL::approx_convex_partition_2 page [740](#)
 CGAL::greene_approx_convex_partition_2 page [746](#)
 CGAL::optimal_convex_partition_2 page [756](#)
 CGAL::partition_is_valid_2 page [761](#)
 CGAL::y_monotone_partition_is_valid_2 page [775](#)

Implementation

This function implements the algorithm presented by de Berg *et al.* [[dBvKOS97](#)] which requires $O(n \log n)$ time and $O(n)$ space for a polygon with n vertices.

Example

The following program computes a y-monotone partitioning of a polygon using the default traits class and stores the partition polygons in the list *partition_polys*. It then asserts that each partition polygon produced is, in fact, y-monotone and that the partition is valid. (Note that these assertions are superfluous unless the postcondition checking for *y_monotone_partition_2* has been turned off.)

```
// file: examples/Partition_2/y_monotone_partition_2.C

#include <CGAL/basic.h>
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Partition_traits_2.h>
#include <CGAL/partition_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_polygon_2.h>
#include <cassert>
#include <list>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Partition_traits_2<K> Traits;
typedef Traits::Point_2 Point_2;
typedef Traits::Polygon_2 Polygon_2;
typedef std::list<Polygon_2> Polygon_list;
typedef CGAL::Creator_uniform_2<int, Point_2> Creator;
typedef CGAL::Random_points_in_square_2<Point_2, Creator> Point_generator;

void make_polygon(Polygon_2& polygon)
{
    polygon.push_back(Point_2(391, 374));
    polygon.push_back(Point_2(240, 431));
    polygon.push_back(Point_2(252, 340));
    polygon.push_back(Point_2(374, 320));
    polygon.push_back(Point_2(289, 214));
    polygon.push_back(Point_2(134, 390));
    polygon.push_back(Point_2( 68, 186));
    polygon.push_back(Point_2(154, 259));
    polygon.push_back(Point_2(161, 107));
    polygon.push_back(Point_2(435, 108));
    polygon.push_back(Point_2(208, 148));
    polygon.push_back(Point_2(295, 160));
    polygon.push_back(Point_2(421, 212));
    polygon.push_back(Point_2(441, 303));
}

int main( )
{
    Polygon_2    polygon;
    Polygon_list partition_polys;

    /*
        CGAL::random_polygon_2(50, std::back_inserter(polygon),
                               Point_generator(100));
    */
}
```

```

*/
make_polygon(polygon);
CGAL::y_monotone_partition_2(polygon.vertices_begin(),
                             polygon.vertices_end(),
                             std::back_inserter(partition_polys));

std::list<Polygon_2>::const_iterator poly_it;
for (poly_it = partition_polys.begin(); poly_it != partition_polys.end();
     poly_it++)
{
    assert(CGAL::is_y_monotone_2((*poly_it).vertices_begin(),
                                (*poly_it).vertices_end()));
}

assert(CGAL::partition_is_valid_2(polygon.vertices_begin(),
                                polygon.vertices_end(),
                                partition_polys.begin(),
                                partition_polys.end()));

return 0;
}

```

CGAL::y_monotone_partition_is_valid_2

Definition

Function that determines if a given set of polygons represents a valid y-monotone partitioning for a given sequence of points that define a simple, counterclockwise-oriented polygon. A valid partition is one in which the polygons are nonoverlapping and the union of the polygons is the same as the original polygon and each polygon is y-monotone

```
#include <CGAL/partition_is_valid_2.h>
```

```
template<class InputIterator, class ForwardIterator, class Traits>
bool y_monotone_partition_is_valid_2( InputIterator point_first,
                                     InputIterator point_beyond,
                                     ForwardIterator poly_first,
                                     ForwardIterator poly_beyond,
                                     Traits traits = Default_traits)
```

determines if the polygons in the range $[poly_first, poly_beyond)$ define a valid y-monotone partition of the polygon represented by the points in the range $[point_first, point_beyond)$. The function returns *true* iff the partition is valid and otherwise returns false.

Precondition: Points in the range $[point_first, point_beyond)$ define a simple, counterclockwise-oriented polygon.

Requirements

1. *Traits* is a model of the concept *YMonotonePartitionIsValidTraits_2*.
2. *InputIterator::value_type* should be *Traits::Point_2*, which should also be the type of the points stored in an object of type *Traits::Polygon_2*.
3. *ForwardIterator::value_type* should be *Traits::Polygon_2*.

The default traits class *Default_traits* is *Partition_traits_2*, with the representation type determined by *InputIterator::value_type*.

See Also

CGAL::y_monotone_partition_2 page 772
 CGAL::is_y_monotone_2 page 749
 CGAL::partition_is_valid_2 page 761
 CGAL::Partition_is_valid_traits_2<Traits, PolygonIsValid> page 767

Implementation

This function uses the function *partition_is_valid_2* together with the function object *Is_y_monotone_2* to determine if each polygon is *y*-monotone or not. Thus the time required is $O(n \log n + e \log e)$ where n is the total number of vertices of the partition polygons and e is the total number of edges.

Example

See the example presented with the function *y_monotone_partition_2* for an illustration of the use of this function.

_____ *advanced* _____

YMonotonePartitionIsValidTraits_2

Definition

Requirements of a traits class that is used by *y_monotone_partition_is_valid_2* for testing the validity of a y-monotone partition of a polygon.

Types

All types required by the concept *PartitionIsValidTraits_2* are required except the function object type *Is_valid*. The following type is required instead:

YMonotonePartitionIsValidTraits_2::Is_y_monotone_2

Model of the concept *PolygonIsValid* that tests if a sequence of points is y-monotone or not.

Creation

Only a copy constructor is required.

YMonotonePartitionIsValidTraits_2 traits(& tr);

Operations

The following function that creates an instance of the above predicate object type must exist instead of the function *is_valid_object* required by *PartitionIsValidTraits_2*.

Is_y_monotone_2 traits.is_y_monotone_2_object(t)

Has Models

CGAL::Partition_traits_2<R> page [769](#)

See Also

CGAL::partition_is_valid_2 page [761](#)

CGAL::y_monotone_partition_2 page [772](#)

YMonotonePartitionTraits_2

Definition

Requirements of a traits class to be used with the function `y_monotone_partition_2`.

Generalizes

PartitionTraits_2 page [765](#)

Types

In addition to the types defined for the concept PartitionTraits_2, the following types are also required:

YMonotonePartitionTraits_2:: Line_2

YMonotonePartitionTraits_2:: Compare_x_at_y_2

Predicate object type that provides *CGAL::Comparison_result operator()(Point_2 p, Line_2 h)* to compare the *x* coordinate of *p* and the horizontal projection of *p* on *h*.

YMonotonePartitionTraits_2:: Construct_line_2

Function object type that provides *Line_2 operator()(Point_2 p, Point_2 q)*, which constructs and returns the line defined by the points *p* and *q*.

YMonotonePartitionTraits_2:: Is_horizontal_2

Function object type that provides *bool operator()(Line_2 l)*, which returns *true* iff the line *l* is horizontal.

Creation

A copy constructor and default constructor are required.

YMonotonePartitionTraits_2 traits;

YMonotonePartitionTraits_2 traits(YMonotonePartitionTraits tr);

Operations

In addition to the functions required for the concept PartitionTraits_2, the following functions that create instances of the above function object types must exist.

Construct_line_2 *traits.construct_line_2_object()*

Compare_x_at_y_2 *traits.compare_x_at_y_2_object()*

Is_horizontal_2 *traits.is_horizontal_2_object()*

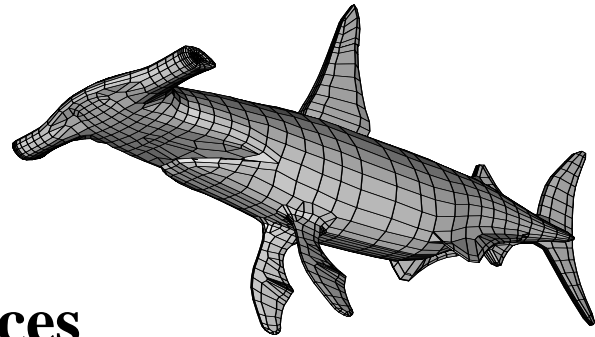
Has Models

CGAL::Partition_traits_2<R> page [769](#)

Chapter 10

3D Polyhedral Surfaces

Lutz Kettner



Contents

10.1 Introduction	781
10.2 Definition	782
10.3 Example Programs	782
10.3.1 First Example Using Defaults	783
10.3.2 Example with Geometry in Vertices	783
10.3.3 Example for Affine Transformation	784
10.3.4 Example Computing Plane Equations	784
10.3.5 Example with a Vector Instead of a List Representation	785
10.3.6 Example with Circulator Writing Object File Format (OFF)	786
10.3.7 Example Using Euler Operators to Build a Cube	787
10.4 File I/O	789
10.5 Extending Vertices, Halfedges, and Facets	790
10.6 Advanced Example Programs	792
10.6.1 Example Creating a Subdivision Surface	792
10.6.2 Example Using the Incremental Builder and Modifier Mechanism	795

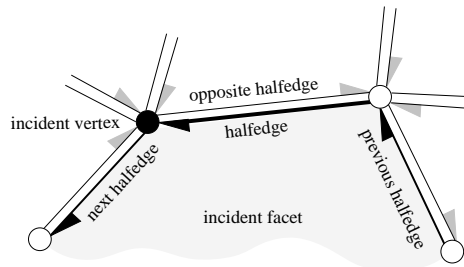
10.1 Introduction

Polyhedral surfaces in three dimensions are composed of vertices, edges, facets and an incidence relationship on them. The organization beneath is a halfedge data structure, which restricts the class of representable surfaces to orientable 2-manifolds – with and without boundary. If the surface is closed we call it a *polyhedron*, for example, see the above model of a hammerhead:

The polyhedral surface is realized as a container class that manages vertices, halfedges, facets with their incidences, and that maintains the combinatorial integrity of them. It is based on the highly flexible design of the halfedge data structure, see the introduction in Chapter 11 and [Ket99]. However, the polyhedral surface can be used and understood without knowing the underlying design. Some of the examples in this chapter introduce also gradually into first applications of this flexibility.

10.2 Definition

A polyhedral surface $CGAL::Polyhedron_3<PolyhedronTraits_3>$ in three dimensions consists of vertices V , edges E , facets F and an incidence relation on them. Each edge is represented by two halfedges with opposite orientations. The incidences stored with a halfedge are illustrated in the following figure:



Vertices represent points in space. Edges are straight line segments between two endpoints. Facets are planar polygons without holes. Facets are defined by the circular sequence of halfedges along their boundary. The polyhedral surface itself can have holes (with at least two facets surrounding it since a single facet cannot have a hole). The halfedges along the boundary of a hole are called *border halfedges* and have no incident facet. An edge is a *border edge* if one of its halfedges is a border halfedge. A surface is *closed* if it contains no border halfedges. A closed surface is a boundary representation for polyhedra in three dimensions. The convention is that the halfedges are oriented counterclockwise around facets as seen from the outside of the polyhedron. An implication is that the halfedges are oriented clockwise around the vertices. The notion of the solid side of a facet as defined by the halfedge orientation extends to polyhedral surfaces with border edges although they do not define a closed object. If normal vectors are considered for the facets, normals point outwards (following the right-hand rule).

The strict definition can be found in [Ket99]. One implication of this definition is that the polyhedral surface is always an orientable and oriented 2-manifold with border edges, i.e., the neighborhood of each point on the polyhedral surface is either homeomorphic to a disc or to a half disc, except for vertices where many holes and surfaces with boundary can join. Another implication is that the smallest representable surface avoiding self intersections is a triangle (for polyhedral surfaces with border edges) or a tetrahedron (for polyhedra). Boundary representations of orientable 2-manifolds are closed under Euler operations. They are extended with operations that create or close holes in the surface.

Other intersections besides the incidence relation are not allowed. However, this is not automatically verified in the operations, since self intersections are not easy to check efficiently. $CGAL::Polyhedron_3<PolyhedronTraits_3>$ does only maintain the combinatorial integrity of the polyhedral surface (using Euler operations) and does not consider the coordinates of the points or any other geometric information.

$CGAL::Polyhedron_3<PolyhedronTraits_3>$ can represent polyhedral surfaces as well as polyhedra. The interface is designed in such a way that it is easy to ignore border edges and work only with polyhedra.

10.3 Example Programs

The polyhedral surface is based on the highly flexible design of the halfedge data structure. Examples for this flexibility can be found in Section 10.5 and in Section 11.3. This design is not a prerequisite to understand the following examples. See also the Section 10.6 below for some advanced examples.

10.3.1 First Example Using Defaults

The first example instantiates a polyhedron using a kernel as traits class. It creates a tetrahedron and stores the reference to one of its halfedges in a *Halfedge_handle*. Handles, also known as *trivial iterators*, are used to keep references to halfedges, vertices, or facets for future use. There is also a *Halfedge_iterator* type for enumerating halfedges. Such an iterator type can be used wherever a handle is required. Respective *Halfedge_const_handle* and *Halfedge_const_iterator* for a constant polyhedron and similar handles and iterators with *Vertex_* and *Facet_* prefix are provided too.

The example continues with a test if the halfedge actually refers to a tetrahedron. This test checks the connected component referred to by the halfedge *h* and not the polyhedral surface as a whole. This example works only on the combinatorial level of a polyhedral surface. The next example adds the geometry.

```
// file: examples/Polyhedron/polyhedron_prog_simple.C

#include <CGAL/Simple_cartesian.h>
#include <CGAL/Polyhedron_3.h>

typedef CGAL::Simple_cartesian<double>      Kernel;
typedef CGAL::Polyhedron_3<Kernel>          Polyhedron;
typedef Polyhedron::Halfedge_handle          Halfedge_handle;

int main() {
    Polyhedron P;
    Halfedge_handle h = P.make_tetrahedron();
    if ( P.is_tetrahedron(h) )
        return 0;
    return 1;
}
```

10.3.2 Example with Geometry in Vertices

We add geometry to our construction of a tetrahedron. Four points are passed as arguments to the construction. This example demonstrates in addition the use of the vertex iterator and the access to the point in the vertices. Note the natural access notation *v->point()*. Similarly, all information stored in a vertex, halfedge, and facet can be accessed with a member function given a handle or iterator. For example, given a halfedge handle *h* we can write *h->next()* to get a halfedge handle to the next halfedge, *h->opposite()* for the opposite halfedge, *h->vertex()* for the incident vertex at the tip of *h*, and so on. The output of the program will be “1 0 0\n0 1 0\n0 0 1\n0 0 0\n”.

```
// file: examples/Polyhedron/polyhedron_prog_tetra.C

#include <CGAL/Simple_cartesian.h>
#include <CGAL/Polyhedron_3.h>
#include <iostream>

typedef CGAL::Simple_cartesian<double>      Kernel;
typedef Kernel::Point_3                      Point_3;
typedef CGAL::Polyhedron_3<Kernel>          Polyhedron;
typedef Polyhedron::Vertex_iterator          Vertex_iterator;
```

```

int main() {
    Point_3 p( 1.0, 0.0, 0.0);
    Point_3 q( 0.0, 1.0, 0.0);
    Point_3 r( 0.0, 0.0, 1.0);
    Point_3 s( 0.0, 0.0, 0.0);

    Polyhedron P;
    P.make_tetrahedron( p, q, r, s);
    CGAL::set_ascii_mode( std::cout);
    for ( Vertex_iterator v = P.vertices_begin(); v != P.vertices_end(); ++v)
        std::cout << v->point() << std::endl;
    return 0;
}

```

The polyhedron offers a point iterator for convenience. The above for loop simplifies to a single statement by using *std::copy* and the ostream iterator adaptor.

```

std::copy( P.points_begin(), P.points_end(),
           std::ostream_iterator<Point_3>(std::cout, "\n"));

```

10.3.3 Example for Affine Transformation

An affine transformation A can act as a functor transforming points and a point iterator is conveniently defined for polyhedral surfaces. So, assuming we want only the point coordinates of a polyhedron P transformed, *std::transform* does the job in a single line.

```

std::transform( P.points_begin(), P.points_end(), P.points_begin(), A);

```

10.3.4 Example Computing Plane Equations

The polyhedral surface has already provisions to store a plane equation for each facet. However, it does not provide a function to compute plane equations.

This example computes the plane equations of a polyhedral surface. The actual computation is implemented in the `compute_plane_equations` function. Depending on the arithmetic (exact/inexact) and the shape of the facets (convex/non-convex) different methods are useful. We assume here strictly convex facets and exact arithmetic. In our example a homogeneous representation with `int` coordinates is sufficient. The four plane equations of the tetrahedron are the output of the program.

```

// file: examples/Polyhedron/polyhedron_prog_planes.C

#include <CGAL/Homogeneous.h>
#include <CGAL/Polyhedron_3.h>
#include <iostream>
#include <algorithm>

struct Plane_equation {
    template <class Facet>

```



```

typename Facet::Plane_3 operator()( Facet& f) {
    typename Facet::Halfedge_handle h = f.halfedge();
    typedef typename Facet::Plane_3 Plane;
    return Plane( h->vertex()->point(),
                 h->next()->vertex()->point(),
                 h->next()->next()->vertex()->point());
}
};

typedef CGAL::Homogeneous<int>      Kernel;
typedef Kernel::Point_3             Point_3;
typedef Kernel::Plane_3             Plane_3;
typedef CGAL::Polyhedron_3<Kernel> Polyhedron;

int main() {
    Point_3 p( 1, 0, 0);
    Point_3 q( 0, 1, 0);
    Point_3 r( 0, 0, 1);
    Point_3 s( 0, 0, 0);
    Polyhedron P;
    P.make_tetrahedron( p, q, r, s);
    std::transform( P.facets_begin(), P.facets_end(), P.planes_begin(),
                   Plane_equation());
    CGAL::set_pretty_mode( std::cout);
    std::copy( P.planes_begin(), P.planes_end(),
              std::ostream_iterator<Plane_3>( std::cout, "\n"));
    return 0;
}

```

10.3.5 Example with a Vector Instead of a List Representation

The polyhedron class template has actually four parameters, where three of them have default values. Using the default values explicitly in our examples above for three parameter—ignoring the fourth parameter, which would be a standard allocator for container class— the definition of a polyhedron looks like:

```

typedef CGAL::Polyhedron_3< Traits,
                          CGAL::Polyhedron_items_3,
                          CGAL::HalfedgeDS_default> Polyhedron;

```

The *CGAL::Polyhedron_items_3* class contains the types used for vertices, edges, and facets. The *CGAL::HalfedgeDS_default* class defines the halfedge data structure used, which is a list-based representation in this case. An alternative is a vector-based representation. Using a vector provides random access for the elements in the polyhedral surface and is more space efficient, but elements cannot be deleted arbitrarily. Using a list allows arbitrary deletions, but provides only bidirectional iterators and is less space efficient. The following example creates again a tetrahedron with given points, but in a vector-based representation.

The vector-based representation resizes automatically if the reserved capacity is not sufficient for the new items created. Upon resizing all handles, iterators, and circulators become invalid. Their correct update in the halfedge data structure is costly, thus it is advisable to reserve enough space in advance as indicated with the alternative constructor in the comment.

advanced

Note that the polyhedron and not the underlying halfedge data structure triggers the resize operation, since the resize operation requires some preconditions, such as valid incidences, to be fulfilled that only the polyhedron can guarantee.

advanced

```
// file: examples/Polyhedron/polyhedron_prog_vector.C

#include <CGAL/Cartesian.h>
#include <CGAL/HalfedgeDS_vector.h>
#include <CGAL/Polyhedron_3.h>
#include <iostream>

typedef CGAL::Cartesian<double>          Kernel;
typedef Kernel::Point_3                  Point_3;
typedef CGAL::Polyhedron_3< Kernel,
                           CGAL::Polyhedron_items_3,
                           CGAL::HalfedgeDS_vector> Polyhedron;

int main() {
    Point_3 p( 1.0, 0.0, 0.0);
    Point_3 q( 0.0, 1.0, 0.0);
    Point_3 r( 0.0, 0.0, 1.0);
    Point_3 s( 0.0, 0.0, 0.0);

    Polyhedron P;    // alternative constructor: Polyhedron P(4,12,4);
    P.make_tetrahedron( p, q, r, s);
    CGAL::set_ascii_mode( std::cout);
    std::copy( P.points_begin(), P.points_end(),
               std::ostream_iterator<Point_3>( std::cout, "\n"));
    return 0;
}
```

10.3.6 Example with Circulator Writing Object File Format (OFF)

We create a tetrahedron and write it to *std::cout* using the Object File Format (OFF) [Phi96]. This example makes use of STL algorithms (*std::copy*, *std::distance*), STL *std::ostream_iterator*, and CGAL circulators. The polyhedral surface provides convenient circulators for the counterclockwise circular sequence of halfedges around a facet and the clockwise circular sequence of halfedges around a vertex.

However, the computation of the vertex index in the inner loop of the facet output is not advisable with the *std::distance* function, since it takes linear time for non random-access iterators, which leads to quadratic runtime. For better runtime the vertex index needs to be stored separately and computed once before writing the facets. It can be stored, for example, in the vertex itself or in a hash-structure. See also the following Section 10.4 for file I/O.

```
// file: examples/Polyhedron/polyhedron_prog_off.C

#include <CGAL/Simple_cartesian.h>
#include <CGAL/Polyhedron_3.h>
```

```

#include <iostream>

typedef CGAL::Simple_cartesian<double>          Kernel;
typedef Kernel::Point_3                         Point_3;
typedef CGAL::Polyhedron_3<Kernel>             Polyhedron;
typedef Polyhedron::Facet_iterator             Facet_iterator;
typedef Polyhedron::Halfedge_around_facet_circulator Halfedge_facet_circulator;

int main() {
    Point_3 p( 0.0, 0.0, 0.0);
    Point_3 q( 1.0, 0.0, 0.0);
    Point_3 r( 0.0, 1.0, 0.0);
    Point_3 s( 0.0, 0.0, 1.0);

    Polyhedron P;
    P.make_tetrahedron( p, q, r, s);

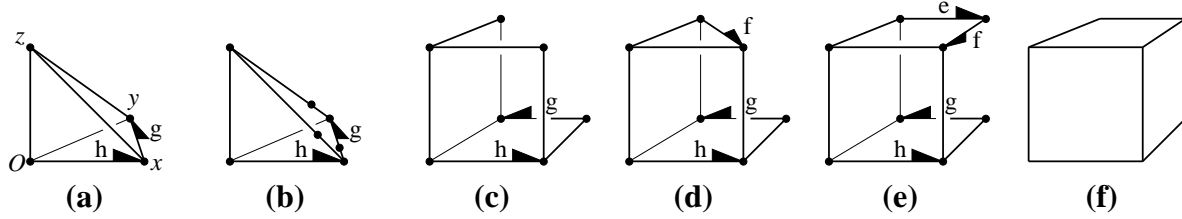
    // Write polyhedron in Object File Format (OFF).
    CGAL::set_ascii_mode( std::cout);
    std::cout << "OFF" << std::endl << P.size_of_vertices() << ' '
               << P.size_of_facets() << " 0" << std::endl;
    std::copy( P.points_begin(), P.points_end(),
               std::ostream_iterator<Point_3>( std::cout, "\n"));
    for ( Facet_iterator i = P.facets_begin(); i != P.facets_end(); ++i) {
        Halfedge_facet_circulator j = i->facet_begin();
        // Facets in polyhedral surfaces are at least triangles.
        CGAL_assertion( CGAL::circulator_size(j) >= 3);
        std::cout << CGAL::circulator_size(j) << ' ';
        do {
            std::cout << ' ' << std::distance(P.vertices_begin(), j->vertex());
        } while ( ++j != i->facet_begin());
        std::cout << std::endl;
    }
    return 0;
}

```

10.3.7 Example Using Euler Operators to Build a Cube

Euler operators are the natural way of modifying polyhedral surfaces. We provide a set of operations for polyhedra: *split_facet()*, *join_facet()*, *split_vertex()*, *join_vertex()*, *split_loop()*, and *join_loop()*. We add further convenient operators, such as *split_edge()*. However, they could be implemented using the six operators above. Furthermore, we provide more operators to work with polyhedral surfaces with border edges, for example, creating and deleting holes. We refer to the references manual for the definition and illustrative figures of the Euler operators.

The following example implements a function that appends a unit cube to a polyhedral surface. To keep track of the different steps during the creation of the cube a sequence of sketches might help with labels for the different handles that occur in the program code. The following Figure shows six selected steps from the creation sequence. These steps are also marked in the program code.



```
// file: examples/Polyhedron/polyhedron_prog_cube.C

#include <CGAL/Simple_cartesian.h>
#include <CGAL/Polyhedron_3.h>
#include <iostream>

template <class Poly>
typename Poly::Halfedge_handle make_cube_3( Poly& P) {
    // appends a cube of size [0,1]^3 to the polyhedron P.
    CGAL_precondition( P.is_valid());
    typedef typename Poly::Point_3      Point;
    typedef typename Poly::Halfedge_handle Halfedge_handle;
    Halfedge_handle h = P.make_tetrahedron( Point( 1, 0, 0),
                                           Point( 0, 0, 1),
                                           Point( 0, 0, 0),
                                           Point( 0, 1, 0));
    Halfedge_handle g = h->next()->opposite()->next();           // Fig. (a)
    P.split_edge( h->next());
    P.split_edge( g->next());
    P.split_edge( g);                                           // Fig. (b)
    h->next()->vertex()->point()      = Point( 1, 0, 1);
    g->next()->vertex()->point()      = Point( 0, 1, 1);
    g->opposite()->vertex()->point() = Point( 1, 1, 0);           // Fig. (c)
    Halfedge_handle f = P.split_facet( g->next(),
                                       g->next()->next()->next()); // Fig. (d)
    Halfedge_handle e = P.split_edge( f);
    e->vertex()->point() = Point( 1, 1, 1);                       // Fig. (e)
    P.split_facet( e, f->next()->next());                         // Fig. (f)
    CGAL_postcondition( P.is_valid());
    return h;
}

typedef CGAL::Simple_cartesian<double>      Kernel;
typedef CGAL::Polyhedron_3<Kernel>          Polyhedron;
typedef Polyhedron::Halfedge_handle         Halfedge_handle;

int main() {
    Polyhedron P;
    Halfedge_handle h = make_cube_3( P);
    return (P.is_tetrahedron(h) ? 1 : 0);
}
```

10.4 File I/O

Simple file I/O for polyhedral surfaces is already provided in the library. The file I/O considers so far only the topology of the surface and its point coordinates. It ignores a possible plane equation or any user-added attributes, such as color.

The default file format supported in CGAL for output as well as for input is the Object File Format, OFF, with file extension `.off`, which is also understood by GeomView [Phi96]. For OFF an ASCII and a binary format exist. The format can be selected with the CGAL modifiers for streams, `set_ascii_mode` and `set_binary_mode` respectively. The modifier `set_pretty_mode` can be used to allow for (a few) structuring comments in the output. Otherwise, the output would be free of comments. The default for writing is ASCII without comments. Both, ASCII and binary format, can be read independent of the stream setting. Since this file format is the default format, `istream` operators are provided for it.

```
#include <CGAL/IO/Polyhedron_istream.h>
```

```
template <class PolyhedronTraits_3>
ostream& out << CGAL::Polyhedron_3<PolyhedronTraits_3> P
```

```
template <class PolyhedronTraits_3>
istream& in >> CGAL::Polyhedron_3<PolyhedronTraits_3>& P
```

Additional formats supported for writing are OpenInventor (`.iv`) [Wer94], VRML 1.0 and 2.0 (`.wrl`) [BPP95, VRM96, HW96], and Wavefront Advanced Visualizer object format (`.obj`). Another convenient output function writes a polyhedral surface to a GeomView process spawned from the CGAL program. These output functions are provided as stream operators, now acting on the stream type of the respective format.

```
#include <CGAL/IO/Polyhedron_inventor_ostream.h>
#include <CGAL/IO/Polyhedron_VRML_1_ostream.h>
#include <CGAL/IO/Polyhedron_VRML_2_ostream.h>
#include <CGAL/IO/Polyhedron_geomview_ostream.h>
```

```
template <class PolyhedronTraits_3>
Inventor_ostream& out << CGAL::Polyhedron_3<PolyhedronTraits_3> P
```

```
template <class PolyhedronTraits_3>
VRML_1_ostream& out << CGAL::Polyhedron_3<PolyhedronTraits_3> P
```

```
template <class PolyhedronTraits_3>
VRML_2_ostream& out << CGAL::Polyhedron_3<PolyhedronTraits_3> P
```

```
template <class PolyhedronTraits_3>
Geomview_stream& out << CGAL::Polyhedron_3<PolyhedronTraits_3> P
```

All these file formats have in common that they represent a surface as a set of facets. Each facet is a list of indices pointing into a set of vertices. Vertices are represented as coordinate triples. The file I/O for polyhedral surfaces `CGAL::Polyhedron_3` imposes certain restrictions on these formats. They must represent a permissible polyhedral surface, e.g., a 2-manifold and no isolated vertices, see Section 10.1.

Some example programs around the different file formats are provided in the distribution under `examples/Polyhedron_IO/` and `demo/Polyhedron_IO/`. We show an example converting OFF input into VRML 1.0 output.

```
// examples/Polyhedron_IO/polyhedron2vrm1.C
// -----

#include <CGAL/Simple_cartesian.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>
#include <CGAL/IO/Polyhedron_VRML_1_ostream.h>
#include <iostream>

typedef CGAL::Simple_cartesian<double> Kernel;
typedef CGAL::Polyhedron_3<Kernel> Polyhedron;

int main() {
    Polyhedron P;
    std::cin >> P;
    CGAL::VRML_1_ostream out( std::cout);
    out << P;
    return ( std::cin && std::cout) ? 0 : 1;
}
```

10.5 Extending Vertices, Halfedges, and Facets

In Section [10.3.5](#) we have seen how to change the default list representation

```
typedef CGAL::Polyhedron_3< Traits,
                           CGAL::Polyhedron_items_3,
                           CGAL::HalfedgeDS_default> Polyhedron;
```

to a vector based representation of the underlying halfedge data structure. Now we want to look a bit closer at the second template argument, `Polyhedron_items_3`, that specifies what kind of vertex, halfedge, and facet is used. The implementation of `Polyhedron_items_3` looks a bit involved with nested wrapper class templates. But ignoring this technicality, what remains are three local typedefs that define the `Vertex`, the `Halfedge`, and the `Face` for the polyhedral surface. Note that we use here `Face` instead of `facet`. `Face` is the term used for the halfedge data structure. Only the top layer of the polyhedral surface gives alias names renaming `face` to `facet`.

```
class Polyhedron_items_3 {
public:
    template < class Refs, class Traits>
    struct Vertex_wrapper {
        typedef typename Traits::Point_3 Point;
        typedef CGAL::HalfedgeDS_vertex_base<Refs, CGAL::Tag_true, Point> Vertex;
    };
    template < class Refs, class Traits>
    struct Halfedge_wrapper {
        typedef CGAL::HalfedgeDS_halfedge_base<Refs> Halfedge;
    };
    template < class Refs, class Traits>
    struct Face_wrapper {
        typedef typename Traits::Plane_3 Plane;
        typedef CGAL::HalfedgeDS_face_base<Refs, CGAL::Tag_true, Plane> Face;
    };
};
```

```
};
};
```

If we look up in the reference manual the definitions of the three classes used in the typedefs, we will see the confirmation that the default polyhedron uses all supported incidences, a point in the vertex class, and a plane equation in the face class. Note how the wrapper class provides two template parameters, `Refs`, which we discuss a bit later, and `Traits`, which is the geometric traits class used by the polyhedral surface and which provides us here with the types for the point and the plane equation.

Using this example code we can write our own items class. Instead, we illustrate an easier way if we only want to exchange one class. We use a simpler face without the plane equation but with a color attribute added. To simplify the creation of a vertex, halfedge, or face class, it is always recommended to derive from one of the given base classes. Even if the base class would contain no data it would provide convenient type definitions. So, we derive from the base class, repeat the mandatory constructors if necessary—which is not the case for faces but would be for vertices—and add the color attribute.

```
template <class Refs>
struct My_face : public CGAL::HalfedgeDS_face_base<Refs> {
    CGAL::Color color;
};
```

The new items class is derived from the old items class and the wrapper containing the face typedef gets overridden. Note that the name of the wrapper and its template parameters are fixed. They cannot be changed even if, as in this example, a template parameter is not used.

```
struct My_items : public CGAL::Polyhedron_items_3 {
    template <class Refs, class Traits>
    struct Face_wrapper {
        typedef My_face<Refs> Face;
    };
};
```

When we use our new items class with the polyhedral surface, our new face class is used in the halfedge data structure and the color attribute is available in the type `Polyhedron::Facet`. However, `Polyhedron::Facet` is not the same type as our local face typedef for `My_face`, but it is derived therefrom. Thus, everything that we put in the local face type except constructors is then available in the `Polyhedron::Facet` type. For more details, see the Chapter 11 on the halfedge data structure design.

Pulling all pieces together, the full example program illustrates how easy the color attribute can be accessed once it is defined.

```
// file: examples/Polyhedron/polyhedron_prog_color.C

#include <CGAL/Simple_cartesian.h>
#include <CGAL/IO/Color.h>
#include <CGAL/Polyhedron_3.h>

// A face type with a color member variable.
template <class Refs>
struct My_face : public CGAL::HalfedgeDS_face_base<Refs> {
```

```

        CGAL::Color color;
};

// An items type using my face.
struct My_items : public CGAL::Polyhedron_items_3 {
    template <class Refs, class Traits>
    struct Face_wrapper {
        typedef My_face<Refs> Face;
    };
};

typedef CGAL::Simple_cartesian<double>      Kernel;
typedef CGAL::Polyhedron_3<Kernel, My_items> Polyhedron;
typedef Polyhedron::Halfedge_handle         Halfedge_handle;

int main() {
    Polyhedron P;
    Halfedge_handle h = P.make_tetrahedron();
    h->facet()->color = CGAL::RED;
    return 0;
}

```

We come back to the first template parameter, *Refs*, of the wrapper classes. This parameter provides us with local types that allow us to make further references between vertices, halfedges, and facets, which have not already been prepared for in the current design. These local types are *Vertex_handle*, *Halfedge_handle*, *Face_handle*, and there respective *..._const_handle*. We add now a new vertex reference to a face class as follows. Encapsulation and access functions could be added for a more thorough design, but we omit that here for the sake of brevity. The integration of the face class with the items class works as illustrated above.

```

template <class Refs>
struct My_face : public CGAL::HalfedgeDS_face_base<Refs> {
    typedef typename Refs::Vertex_handle Vertex_handle;
    Vertex_handle vertex_ref;
};

```

More advanced examples can be found in the Section 11.3 illustrating further the design of the halfedge data structure.

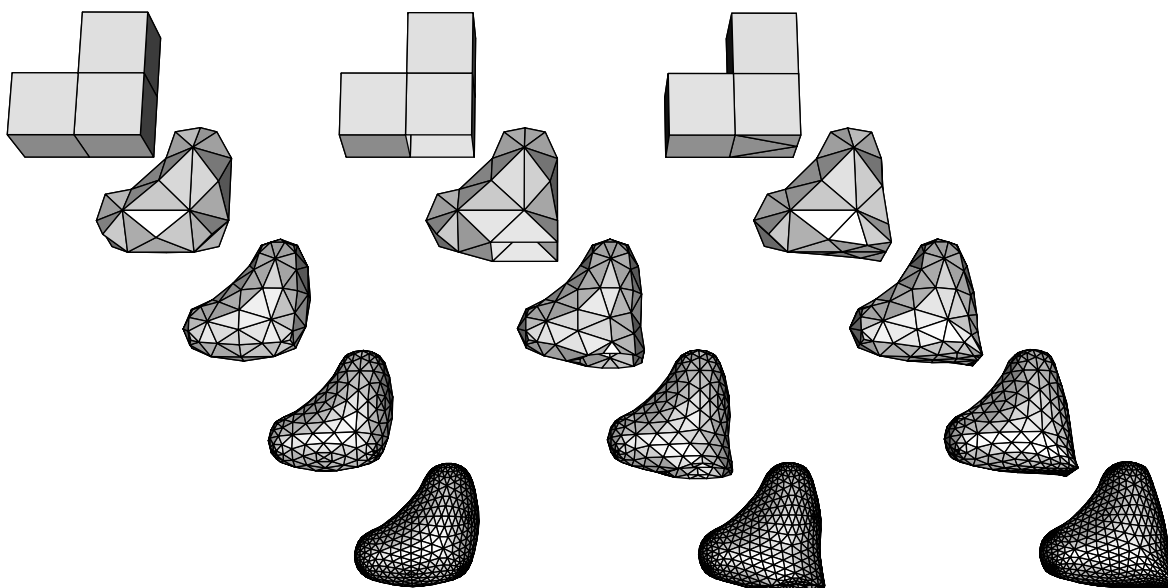
10.6 Advanced Example Programs

10.6.1 Example Creating a Subdivision Surface

This program reads a polyhedral surface from the standard input and writes a refined polyhedral surface to the standard output. Input and output are in the Object File Format, OFF, with the common file extension *.off*, which is also understood by GeomView [Phi96].

The refinement is a single step of the $\sqrt{3}$ -scheme for creating a subdivision surface [Kob00]. Each step subdivides a facet into triangles around a new center vertex, smoothes the position of the old vertices, and flips the old edges. The program is organized along this outline. In each of these parts, the program efficiently uses the

knowledge that the newly created vertices, edges, and facets have been added to the end of the sequences. The program needs additional processing memory only for the smoothing step of the old vertices.



The above figure shows three example objects, each subdivided four times. The initial object for the left sequence is the closed surface of three unit cubes glued together to a corner. The example program shown here can handle only closed surfaces, but the extended example `examples/Polyhedron/polyhedron_prog_subdiv_with_boundary.C` handles surfaces with boundary. So, the middle sequence starts with the same surface where one of the facets has been removed. The boundary subdivides to a nice circle. The third sequence creates a sharp edge using a trick in the object presentation. The sharp edge is actually a hole whose vertex coordinates pinch the hole shut to form an edge. The example directory `examples/Polyhedron/` contains the OFF files used here.

```
// file: examples/Polyhedron/polyhedron_prog_subdiv.C

#include <CGAL/Cartesian.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>
#include <iostream>
#include <algorithm>
#include <vector>
#include <cmath>

typedef CGAL::Cartesian<double>          Kernel;
typedef Kernel::Vector_3                  Vector;
typedef Kernel::Point_3                   Point;
typedef CGAL::Polyhedron_3<Kernel>        Polyhedron;

typedef Polyhedron::Vertex                 Vertex;
typedef Polyhedron::Vertex_iterator        Vertex_iterator;
typedef Polyhedron::Halfedge_handle        Halfedge_handle;
typedef Polyhedron::Edge_iterator          Edge_iterator;
typedef Polyhedron::Facet_iterator         Facet_iterator;
typedef Polyhedron::Halfedge_around_vertex_const_circulator HV_circulator;
```

```

typedef Polyhedron::Halfedge_around_facet_circulator      HF_circulator;

void create_center_vertex( Polyhedron& P, Facet_iterator f) {
    Vector vec( 0.0, 0.0, 0.0);
    std::size_t order = 0;
    HF_circulator h = f->facet_begin();
    do {
        vec = vec + ( h->vertex()->point() - CGAL::ORIGIN);
        ++ order;
    } while ( ++h != f->facet_begin());
    CGAL_assertion( order >= 3); // guaranteed by definition of polyhedron
    Point center = CGAL::ORIGIN + (vec / order);
    Halfedge_handle new_center = P.create_center_vertex( f->halfedge());
    new_center->vertex()->point() = center;
}

struct Smooth_old_vertex {
    Point operator()( const Vertex& v) const {
        CGAL_precondition((CGAL::circulator_size( v.vertex_begin()) & 1) == 0);
        std::size_t degree = CGAL::circulator_size( v.vertex_begin()) / 2;
        double alpha = ( 4.0 - 2.0 * std::cos( 2.0 * CGAL_PI / degree)) / 9.0;
        Vector vec = (v.point() - CGAL::ORIGIN) * ( 1.0 - alpha);
        HV_circulator h = v.vertex_begin();
        do {
            vec = vec + ( h->opposite()->vertex()->point() - CGAL::ORIGIN)
                * alpha / degree;
            ++ h;
            CGAL_assertion( h != v.vertex_begin()); // even degree guaranteed
            ++ h;
        } while ( h != v.vertex_begin());
        return (CGAL::ORIGIN + vec);
    }
};

void flip_edge( Polyhedron& P, Halfedge_handle e) {
    Halfedge_handle h = e->next();
    P.join_facet( e);
    P.split_facet( h, h->next()->next());
}

void subdiv( Polyhedron& P) {
    if ( P.size_of_facets() == 0)
        return;
    // We use that new vertices/halfedges/facets are appended at the end.
    std::size_t nv = P.size_of_vertices();
    Vertex_iterator last_v = P.vertices_end();
    -- last_v; // the last of the old vertices
    Edge_iterator last_e = P.edges_end();
    -- last_e; // the last of the old edges
    Facet_iterator last_f = P.facets_end();
    -- last_f; // the last of the old facets

    Facet_iterator f = P.facets_begin(); // create new center vertices
    do {

```

```

        create_center_vertex( P, f);
    } while ( f++ != last_f);

    std::vector<Point> pts; // smooth the old vertices
    pts.reserve( nv); // get intermediate space for the new points
    ++ last_v; // make it the past-the-end position again
    std::transform( P.vertices_begin(), last_v, std::back_inserter( pts),
        Smooth_old_vertex());
    std::copy( pts.begin(), pts.end(), P.points_begin());

    Edge_iterator e = P.edges_begin(); // flip the old edges
    ++ last_e; // make it the past-the-end position again
    while ( e != last_e) {
        Halfedge_handle h = e;
        ++e; // careful, incr. before flip since flip destroys current edge
        flip_edge( P, h);
    };
    CGAL_postcondition( P.is_valid());
}

int main() {
    Polyhedron P;
    std::cin >> P;
    P.normalize_border();
    if ( P.size_of_border_edges() != 0) {
        std::cerr << "The input object has border edges. Cannot subdivide."
            << std::endl;
        std::exit(1);
    }
    subdiv( P);
    std::cout << P;
    return 0;
}

```

10.6.2 Example Using the Incremental Builder and Modifier Mechanism

A utility class *CGAL::Polyhedron_incremental_builder_3* helps in creating polyhedral surfaces from a list of points followed by a list of facets that are represented as indices into the point list. This is particularly useful for implementing file reader for common file formats. It is used here to create a triangle.

A modifier mechanism allows to access the internal representation of the polyhedral surface, i.e., the halfedge data structure, in a controlled manner. A modifier is basically a callback mechanism using a function object. When called, the function object receives the internal halfedge data structure as a parameter and can modify it. On return, the polyhedron can check the halfedge data structure for validity. Such a modifier object must always return with a halfedge data structure that is a valid polyhedral surface. The validity check is implemented as an expensive postcondition at the end of the *delegate()* member function, i.e., it is not called by default, only when expensive checks are activated.

In this example, *Build_triangle* is such a function object derived from *CGAL::Modifier_base<HalfedgeDS>*. The *delegate()* member function of the polyhedron accepts this function object and calls its *operator()* with a reference to its internally used halfedge data structure. Thus, this member function in *Build_triangle* can create the triangle in the halfedge data structure.

```

// file: examples/Polyhedron/polyhedron_prog_incr_builder.C

#include <CGAL/Simple_cartesian.h>
#include <CGAL/Polyhedron_incremental_builder_3.h>
#include <CGAL/Polyhedron_3.h>

// A modifier creating a triangle with the incremental builder.
template <class HDS>
class Build_triangle : public CGAL::Modifier_base<HDS> {
public:
    Build_triangle() {}
    void operator()( HDS& hds) {
        // Postcondition: 'hds' is a valid polyhedral surface.
        CGAL::Polyhedron_incremental_builder_3<HDS> B( hds, true);
        B.begin_surface( 3, 1, 6);
        typedef typename HDS::Vertex    Vertex;
        typedef typename Vertex::Point Point;
        B.add_vertex( Point( 0, 0, 0));
        B.add_vertex( Point( 1, 0, 0));
        B.add_vertex( Point( 0, 1, 0));
        B.begin_facet();
        B.add_vertex_to_facet( 0);
        B.add_vertex_to_facet( 1);
        B.add_vertex_to_facet( 2);
        B.end_facet();
        B.end_surface();
    }
};

typedef CGAL::Simple_cartesian<double>    Kernel;
typedef CGAL::Polyhedron_3<Kernel>        Polyhedron;
typedef Polyhedron::HalfedgeDS            HalfedgeDS;

int main() {
    Polyhedron P;
    Build_triangle<HalfedgeDS> triangle;
    P.delegate( triangle);
    CGAL_assertion( P.is_triangle( P.halfedges_begin()));
    return 0;
}

```

3D Polyhedral Surfaces

Reference Manual

Lutz Kettner

Polyhedral surfaces in three dimensions are composed of vertices, edges, facets and an incidence relationship on them. The organization beneath is a halfedge data structure, which restricts the class of representable surfaces to orientable 2-manifolds – with and without boundary. If the surface is closed we call it a *polyhedron*.

The polyhedral surface is realized as a container class managing vertices, halfedges, facets with their incidences, and maintaining the combinatorial integrity of them. Its local types for the vertices, halfedges and facets are documented separately. A default traits class, a default items class and an incremental builder conclude the references. The polyhedral surface is based on the highly flexible design of the halfedge data structure, see the reference for *HalfedgeDS* in Chapter 11 or [Ket99], but the default instantiation of the polyhedral surface can be used without knowing the halfedge data structure.

10.7 Classified Reference Pages

Concepts

PolyhedronTraits_3 page 827
PolyhedronItems_3 page 822

Classes

CGAL::Polyhedron_3<Traits> page 799
CGAL::Polyhedron_3<Traits>::Vertex page 816
CGAL::Polyhedron_3<Traits>::Halfedge page 813
CGAL::Polyhedron_3<Traits>::Facet page 811
CGAL::Polyhedron_traits_3<Kernel> page 828
CGAL::Polyhedron_traits_with_normals_3<Kernel> page 830
CGAL::Polyhedron_items_3 page 824
CGAL::Polyhedron_min_items_3 page 826
CGAL::Polyhedron_incremental_builder_3<HDS> page 818

Functions

template <class Traits>

ostream& ostream& out << CGAL::Polyhedron_3<Traits> P page [832](#)

template <class Traits>
istream& istream& in >> CGAL::Polyhedron_3<Traits>& P page [833](#)

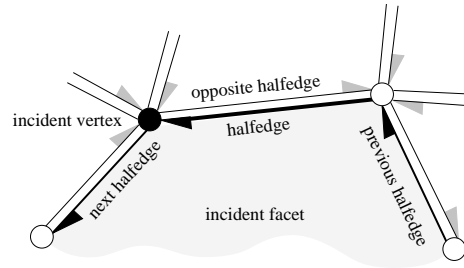
10.8 Alphabetical List of Reference Pages

<i>Facet</i>	page 811
<i>Halfedge</i>	page 813
<i>operator<<</i>	page 2800
<i>operator>></i>	page 2792
<i>PolyhedronItems_3</i>	page 822
<i>PolyhedronTraits_3</i>	page 827
<i>Polyhedron_3<Traits></i>	page 799
<i>Polyhedron_incremental_builder_3<HDS></i>	page 818
<i>Polyhedron_items_3</i>	page 824
<i>Polyhedron_min_items_3</i>	page 826
<i>Polyhedron_traits_3<Kernel></i>	page 828
<i>Polyhedron_traits_with_normals_3<Kernel></i>	page 830
<i>Vertex</i>	page 816

CGAL::Polyhedron_3<Traits>

Definition

A polyhedral surface *Polyhedron_3<Traits>* consists of vertices V , edges E , facets F and an incidence relation on them. Each edge is represented by two halfedges with opposite orientations.



Vertices represent points in 3d-space. Edges are straight line segments between two endpoints. Facets are planar polygons without holes defined by the circular sequence of halfedges along their boundary. The polyhedral surface itself can have holes. The halfedges along the boundary of a hole are called *border halfedges* and have no incident facet. An edge is a *border edge* if one of its halfedges is a border halfedge. A surface is *closed* if it contains no border halfedges. A closed surface is a boundary representation for polyhedra in three dimensions. The convention is that the halfedges are oriented counterclockwise around facets as seen from the outside of the polyhedron. An implication is that the halfedges are oriented clockwise around the vertices. The notion of the solid side of a facet as defined by the halfedge orientation extends to polyhedral surfaces with border edges although they do not define a closed object. If normal vectors are considered for the facets, normals point outwards (following the right hand rule).

The strict definition can be found in [Ket99]. One implication of this definition is that the polyhedral surface is always an orientable and oriented 2-manifold with border edges, i.e., the neighborhood of each point on the polyhedral surface is either homeomorphic to a disc or to a half disc, except for vertices where many holes and surfaces with boundary can join. Another implication is that the smallest representable surface is a triangle (for polyhedral surfaces with border edges) or a tetrahedron (for polyhedra). Boundary representations of orientable 2-manifolds are closed under Euler operations. They are extended with operations that create or close holes in the surface.

Other intersections besides the incidence relation are not allowed, although they are not automatically handled, since self intersections are not easy to check efficiently. *Polyhedron_3<Traits>* does only maintain the combinatorial integrity of the polyhedral surface (using Euler operations) and does not consider the coordinates of the points or any geometric information.

The class *Polyhedron_3<Traits>* can represent polyhedral surfaces as well as polyhedra. The interface is designed in such a way that it is easy to ignore border edges and work only with polyhedra.

The sequence of edges can be ordered in the data structure on request such that the sequence starts with the non-border edges and ends with the border edges. Border edges are then itself ordered such that the halfedge which is incident to the facet comes first and the halfedge incident to the hole comes thereafter. This normalization step counts simultaneously the number of border edges. This number is zero if and only if the surface is a closed polyhedron. Note that this class does not maintain this counter nor the halfedge order during further modifications. There is no automatic caching done for auxiliary information.

```
#include <CGAL/Polyhedron_3.h>
```

Parameters

The full template declaration of *Polyhedron_3*<*Traits*> states four template parameters:

```
template < class PolyhedronTraits_3,
           class PolyhedronItems_3 = CGAL::Polyhedron_items_3,
           template < class T, class I> class HalfedgeDS = CGAL::HalfedgeDS_default,
           class Alloc = CGAL_ALLOCATOR(int)>
class Polyhedron_3;
```

The first parameter requires a model of the *PolyhedronTraits_3* concept as argument, for example *CGAL::Polyhedron_traits_3*. The second parameter expects a model of the *PolyhedronItems_3* concept. By default, the class *CGAL::Polyhedron_items_3* is preselected. The third parameter is a class template. A model of the *HalfedgeDS* concept is expected. By default, the class *CGAL::HalfedgeDS_default* is preselected, which is a list based implementation of the halfedge data structure. The fourth parameter *Alloc* requires a standard allocator for STL container classes. The *rebind* mechanism from *Alloc* will be used to create appropriate allocators internally. A default is provided with the macro *CGAL_ALLOCATOR(int)* from the *<CGAL/memory.h>* header file.

Types

<i>Polyhedron_3</i> < <i>Traits</i> >:: <i>Traits</i>	traits class selected for <i>PolyhedronTraits_3</i> .
<i>Polyhedron_3</i> < <i>Traits</i> >:: <i>Items</i>	items class selected for <i>PolyhedronItems_3</i> .
<i>Polyhedron_3</i> < <i>Traits</i> >:: <i>HalfedgeDS</i>	instantiated halfedge data structure.
<i>Polyhedron_3</i> < <i>Traits</i> >:: <i>size_type</i>	size type of <i>HalfedgeDS</i> .
<i>Polyhedron_3</i> < <i>Traits</i> >:: <i>difference_type</i>	difference type of <i>HalfedgeDS</i> .
<i>Polyhedron_3</i> < <i>Traits</i> >:: <i>iterator_category</i>	iterator category of <i>HalfedgeDS</i> for all iterators.
<i>Polyhedron_3</i> < <i>Traits</i> >:: <i>circulator_category</i>	circulator category of all circulators; bidirectional category if the <i>Items::Halfedge</i> provides a <i>prev()</i> member function, otherwise forward category.
<i>Polyhedron_3</i> < <i>Traits</i> >:: <i>allocator_type</i>	allocator type <i>Alloc</i> .
<i>Polyhedron_3</i> < <i>Traits</i> >:: <i>Vertex</i>	vertex type.
<i>Polyhedron_3</i> < <i>Traits</i> >:: <i>Halfedge</i>	halfedge type.
<i>Polyhedron_3</i> < <i>Traits</i> >:: <i>Facet</i>	facet type.
<i>Polyhedron_3</i> < <i>Traits</i> >:: <i>Point_3</i>	point stored in vertices.
<i>Polyhedron_3</i> < <i>Traits</i> >:: <i>Plane_3</i>	plane equation stored in facets (if supported).

The following handles, iterators, and circulators have appropriate non-mutable counterparts, i.e., *const_handle*, *const_iterator*, and *const_circulator*. The mutable types are assignable to their non-mutable counterparts. Both circulators are assignable to the *Halfedge_iterator*. The iterators are assignable to the respective handle types. Wherever the handles appear in function parameter lists, the corresponding iterators can be used as well. For convenience, the *Edge_iterator* enumerates every other halfedge. It is based on the *CGAL::N_step_adaptor* class. For convenience, the *Point_iterator* enumerates all points in the polyhedral surface in the same order as the *Vertex_iterator*, but with the value type *Point*. It is based on the *CGAL::Iterator_project* adaptor. Similarly, a *Plane_iterator* is provided.

<i>Polyhedron_3</i> < <i>Traits</i> >:: <i>Vertex_handle</i>	handle to vertex.
--------------------------------------------------------------	-------------------

<i>Polyhedron_3<Traits>:: Halfedge_handle</i>	handle to halfedge.
<i>Polyhedron_3<Traits>:: Facet_handle</i>	handle to facet.
<i>Polyhedron_3<Traits>:: Vertex_iterator</i>	iterator over all vertices.
<i>Polyhedron_3<Traits>:: Halfedge_iterator</i>	iterator over all halfedges.
<i>Polyhedron_3<Traits>:: Facet_iterator</i>	iterator over all facets.
<i>Polyhedron_3<Traits>:: Halfedge_around_vertex_circulator</i>	
	circulator of halfedges around a vertex (cw).
<i>Polyhedron_3<Traits>:: Halfedge_around_facet_circulator</i>	
	circulator of halfedges around a facet (ccw).
<i>Polyhedron_3<Traits>:: Edge_iterator</i>	iterator over all edges (every other halfedge).
<i>Polyhedron_3<Traits>:: Point_iterator</i>	iterator over all points.
<i>Polyhedron_3<Traits>:: Plane_iterator</i>	iterator over all plane equations.

— advanced —

Types for Tagging Optional Features

The following types are equal to either *CGAL::Tag_true* or *CGAL::Tag_false*, depending on whether the named feature is supported or not.

<i>Polyhedron_3<Traits>:: Supports_vertex_halfedge</i>	<i>Vertex::halfedge()</i> .
<i>Polyhedron_3<Traits>:: Supports_vertex_point</i>	<i>Vertex::point()</i> .
<i>Polyhedron_3<Traits>:: Supports_halfedge_prev</i>	<i>Halfedge::prev()</i> .
<i>Polyhedron_3<Traits>:: Supports_halfedge_vertex</i>	<i>Halfedge::vertex()</i> .
<i>Polyhedron_3<Traits>:: Supports_halfedge_facet</i>	<i>Halfedge::facet()</i> .
<i>Polyhedron_3<Traits>:: Supports_facet_halfedge</i>	<i>Facet::halfedge()</i> .
<i>Polyhedron_3<Traits>:: Supports_facet_plane</i>	<i>Facet::plane()</i> .
<i>Polyhedron_3<Traits>:: Supports_removal</i>	supports removal of individual elements.

— advanced —

Creation

Polyhedron_3<Traits> *P*(*Traits traits* = *Traits()*);

Polyhedron_3<Traits> *P*(*size_type v*, *size_type h*, *size_type f*, *Traits traits* = *Traits()*);

a polyhedron *P* with storage reserved for *v* vertices, *h* halfedges, and *f* facets.
The reservation sizes are a hint for optimizing storage allocation.

void *P.reserve*(*size_type v*, *size_type h*, *size_type f*)

reserve storage for *v* vertices, *h* halfedges, and *f* facets. The reservation sizes are a hint for optimizing storage allocation. If the *capacity* is already greater than the requested size nothing happens. If the *capacity* changes all iterators and circulators might invalidate.

<i>Halfedge_handle</i>	<i>P.make_tetrahedron()</i>	a tetrahedron is added to the polyhedral surface. Returns a halfedge of the tetrahedron.
<i>Halfedge_handle</i>	<i>P.make_tetrahedron(Point p1, Point p2, Point p3, Point p4)</i>	a tetrahedron is added to the polyhedral surface with its vertices initialized to p_1, p_2, p_3 , and p_4 . Returns that halfedge of the tetrahedron which incident vertex is initialized to p_1 . The incident vertex of the next halfedge is p_2 , and the vertex thereafter is p_3 . The remaining fourth vertex is initialized to p_4 .
<i>Halfedge_handle</i>	<i>P.make_triangle()</i>	a triangle with border edges is added to the polyhedral surface. Returns a non-border halfedge of the triangle.
<i>Halfedge_handle</i>	<i>P.make_triangle(Point p1, Point p2, Point p3)</i>	a triangle with border edges is added to the polyhedral surface with its vertices initialized to p_1, p_2 , and p_3 . Returns that non-border halfedge of the triangle which incident vertex is initialized to p_1 . The incident vertex of the next halfedge is p_2 , and the vertex thereafter is p_3 .

Access Member Functions

<i>bool</i>	<i>P.empty()</i>	returns true if P is empty.
<i>size_type</i>	<i>P.size_of_vertices()</i>	number of vertices.
<i>size_type</i>	<i>P.size_of_halfedges()</i>	number of halfedges (incl. border halfedges).
<i>size_type</i>	<i>P.size_of_facets()</i>	number of facets.
<i>size_type</i>	<i>P.capacity_of_vertices()</i>	space reserved for vertices.
<i>size_type</i>	<i>P.capacity_of_halfedges()</i>	space reserved for halfedges.
<i>size_type</i>	<i>P.capacity_of_facets()</i>	space reserved for facets.
<i>size_t</i>	<i>P.bytes()</i>	bytes used for the polyhedron.
<i>size_t</i>	<i>P.bytes_reserved()</i>	bytes reserved for the polyhedron.
<i>allocator_type</i>	<i>P.get_allocator()</i>	allocator object.
<i>Vertex_iterator</i>	<i>P.vertices_begin()</i>	iterator over all vertices.
<i>Vertex_iterator</i>	<i>P.vertices_end()</i>	past-the-end iterator.
<i>Halfedge_iterator</i>	<i>P.halfedges_begin()</i>	iterator over all halfedges.
<i>Halfedge_iterator</i>	<i>P.halfedges_end()</i>	past-the-end iterator.
<i>Facet_iterator</i>	<i>P.facets_begin()</i>	iterator over all facets (excluding holes).
<i>Facet_iterator</i>	<i>P.facets_end()</i>	past-the-end iterator.
<i>Edge_iterator</i>	<i>P.edges_begin()</i>	iterator over all edges.
<i>Edge_iterator</i>	<i>P.edges_end()</i>	past-the-end iterator.

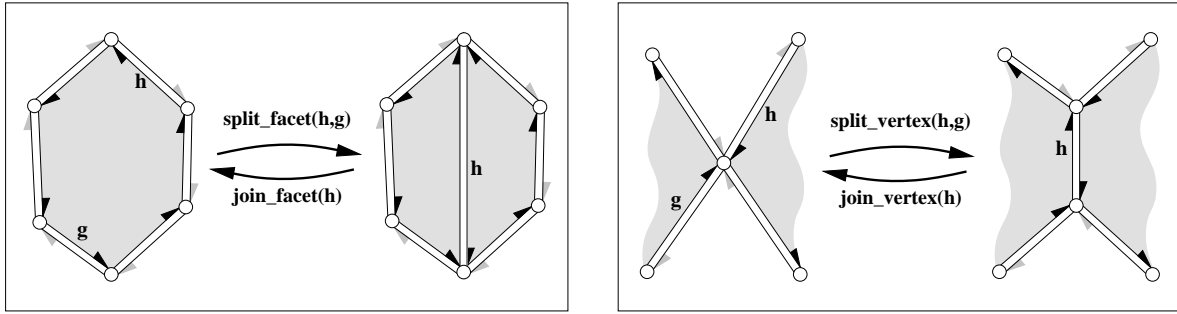
<i>Point_iterator</i>	<i>P.points_begin()</i>	iterator over all points.
<i>Point_iterator</i>	<i>P.points_end()</i>	past-the-end iterator.
<i>Plane_iterator</i>	<i>P.planes_begin()</i>	iterator over all plane equations.
<i>Plane_iterator</i>	<i>P.planes_end()</i>	past-the-end iterator.
<i>Traits</i>	<i>P.traits()</i>	returns the traits class.

Combinatorial Predicates

<i>bool</i>	<i>P.is_closed()</i>	returns <i>true</i> if there are no border edges.
<i>bool</i>	<i>P.is_pure_bivalent()</i>	returns <i>true</i> if all vertices have exactly two incident edges.
<i>bool</i>	<i>P.is_pure_trivalent()</i>	returns <i>true</i> if all vertices have exactly three incident edges.
<i>bool</i>	<i>P.is_pure_triangle()</i>	returns <i>true</i> if all facets are triangles.
<i>bool</i>	<i>P.is_pure_quad()</i>	returns <i>true</i> if all facets are quadrilaterals.
<i>bool</i>	<i>P.is_triangle(Halfedge_const_handle h)</i>	<i>true</i> iff the connected component denoted by <i>h</i> is a triangle.
<i>bool</i>	<i>P.is_tetrahedron(Halfedge_const_handle h)</i>	<i>true</i> iff the connected component denoted by <i>h</i> is a tetrahedron.

Euler Operators (Combinatorial Modifications)

The following Euler operations modify consistently the combinatorial structure of the polyhedral surface. The geometry remains unchanged.



Halfedge_handle $P.\text{split_facet}(\text{Halfedge_handle } h, \text{Halfedge_handle } g)$

splits the facet incident to h and g into two facets with a new diagonal between the two vertices denoted by h and g respectively. The second (new) facet is a copy of the first facet. Returns $h\text{->next}()$ after the operation, i.e., the new diagonal. The new face is to the right of the new diagonal, the old face is to the left. The time is proportional to the distance from h to g around the facet.

Precondition: h and g are incident to the same facet. $h \neq g$ (no loops). $h\text{->next}() \neq g$ and $g\text{->next}() \neq h$ (no multi-edges).

Halfedge_handle $P.\text{join_facet}(\text{Halfedge_handle } h)$

joins the two facets incident to h . The facet incident to $h\text{->opposite}()$ gets removed. Both facets might be holes. Returns the predecessor of h around the facet. The invariant $\text{join_facet}(\text{split_facet}(h, g))$ returns h and keeps the polyhedron unchanged. The time is proportional to the size of the facet removed and the time to compute $h\text{->prev}()$.

Precondition: The degree of both vertices incident to h is at least three (no antennas).

Requirement: $\text{Supports_removal} \equiv \text{CGAL::Tag_true}$.

Halfedge_handle $P.\text{split_vertex}(\text{Halfedge_handle } h, \text{Halfedge_handle } g)$

splits the vertex incident to h and g into two vertices, the old vertex remains and a new copy is created, and connects them with a new edge. Let h_{new} be $h\text{->next}()\text{->opposite}()$ after the split, i.e., a halfedge of the new edge. The split regroupes the halfedges around the two vertices. The halfedge sequence $h_{\text{new}}, g\text{->next}()\text{->opposite}(), \dots, h$ remains around the old vertex, while the halfedge sequence $h_{\text{new}}\text{->opposite}(), h\text{->next}()\text{->opposite}()$ (before the split), \dots, g is regrouped around the new vertex. The split returns h_{new} , i.e., the new halfedge incident to the old vertex. The time is proportional to the distance from h to g around the vertex.

Precondition: h and g are incident to the same vertex. $h \neq g$ (antennas are not allowed).

Note: A special application of the split is $\text{split_vertex}(h, h\text{->next}()\text{->opposite}())$ which is equivalent to an edge split of the halfedge $h\text{->next}()$ that creates a new vertex on the halfedge $h\text{->next}()$. See also $\text{split_edge}(h)$ below.

Halfedge_handle *P.join_vertex(Halfedge_handle h)*

joins the two vertices incident to *h*. The vertex denoted by *h->opposite()* gets removed. Returns the predecessor of *h* around the vertex, i.e., *h->opposite()->prev()*. The invariant *join_vertex(split_vertex(h, g))* returns *h* and keeps the polyhedron unchanged. The time is proportional to the degree of the vertex removed and the time to compute *h->prev()* and *h->opposite()->prev()*.

Precondition: The size of both facets incident to *h* is at least four (no multi-edges).

Requirement: *Supports_removal* \equiv *CGAL::Tag_true*.

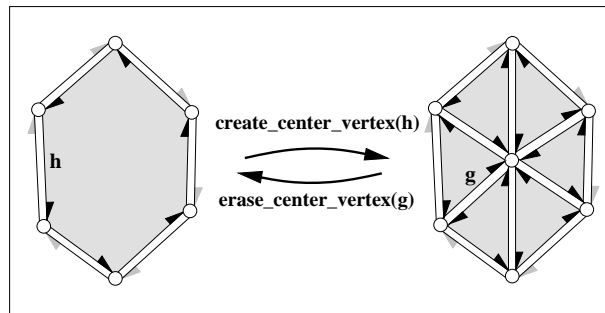
Halfedge_handle *P.split_edge(Halfedge_handle h)*

splits the halfedge *h* into two halfedges inserting a new vertex that is a copy of *h->opposite()->vertex()*. Is equivalent to *split_vertex(h->prev(), h->opposite())*. The call of *prev()* can make this method slower than a direct call of *split_vertex()* if the previous halfedge is already known and computing it would be costly when the halfedge data structure does not support the *prev()* member function. Returns the new halfedge *hnew* pointing to the inserted vertex. The new halfedge is followed by the old halfedge, i.e., *hnew->next() == h*.

Halfedge_handle *P.flip_edge(Halfedge_handle h)*

performs an edge flip. It returns *h* after rotating the edge *h* one vertex in the direction of the face orientation.

Precondition: *h != Halfedge_handle()* and both facets incident to *h* are triangles.



Halfedge_handle *P.create_center_vertex(Halfedge_handle h)*

barycentric triangulation of *h->facet()*. Creates a new vertex, a copy of *h->vertex()*, and connects it to each vertex incident to *h->facet()* splitting *h->facet()* into triangles. *h* remains incident to the original facet, all other triangles are copies of this facet. Returns the halfedge *h->next()* after the operation, i.e., a halfedge pointing to the new vertex. The time is proportional to the size of the facet.

Precondition: *h* is not a border halfedge.

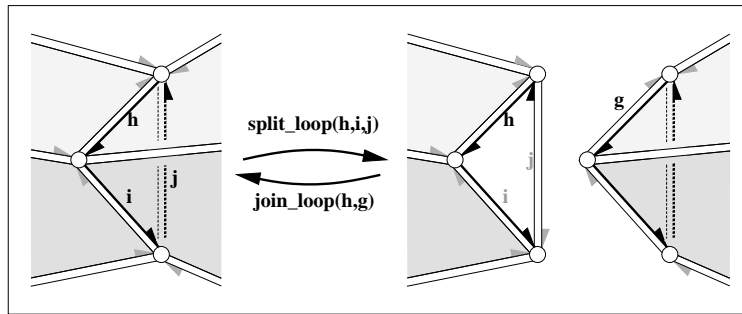
Halfedge_handle *P.erase_center_vertex(Halfedge_handle g)*

reverses *create_center_vertex*. Erases the vertex pointed to by *g* and all incident halfedges thereby merging all incident facets. Only *g->facet()* remains. The neighborhood of *g->vertex()* may not be triangulated, it can have larger facets. Returns the halfedge *g->prev()*. Thus, the invariant *h == erase_center_vertex(create_center_vertex(h))* holds if *h* is not a border halfedge. The time is proportional to the sum of the size of all incident facets.

Precondition: None of the incident facets of *g->vertex()* is a hole. There are at least two distinct facets incident to the facets that are incident to *g->vertex()*. (This prevents the operation from collapsing a volume into two facets glued together with opposite orientations, such as would happen with any vertex of a tetrahedron.)

Requirement: *Supports_removal* \equiv *CGAL::Tag_true*.

Euler Operators Modifying Genus



Halfedge_handle *P.split_loop(Halfedge_handle h, Halfedge_handle i, Halfedge_handle j)*

cuts the polyhedron into two parts along the cycle (h, i, j) (edge j runs on the backside of the three dimensional figure above). Three new vertices (one copy for each vertex in the cycle) and three new halfedges (one copy for each halfedge in the cycle), and two new triangles are created. h, i, j will be incident to the first new triangle. The return value will be the halfedge incident to the second new triangle which is the copy of $h_opposite()$.

Precondition: h, i, j denote distinct, consecutive vertices of the polyhedron and form a cycle: i.e., $h->vertex() == i->opposite()->vertex(), \dots, j->vertex() == h->opposite()->vertex()$. The six facets incident to h, i, j are all distinct.

Halfedge_handle *P.join_loop(Halfedge_handle h, Halfedge_handle g)*

glues the boundary of the two facets denoted by h and g together and returns h . Both facets and the vertices along the facet denoted by g gets removed. Both facets may be holes. The invariant $join_loop(h, split_loop(h, i, j))$ returns h and keeps the polyhedron unchanged.

Precondition: The facets denoted by h and g are different and have equal degree (i.e., number of edges).

Requirement: *Supports_removal* \equiv *CGAL::Tag_true*.

Modifying Facets and Holes

Halfedge_handle *P.make_hole(Halfedge_handle h)*

removes the incident facet of *h* and changes all halfedges incident to the facet into border edges. Returns *h*. See *erase_facet(h)* for a more generalized variant.

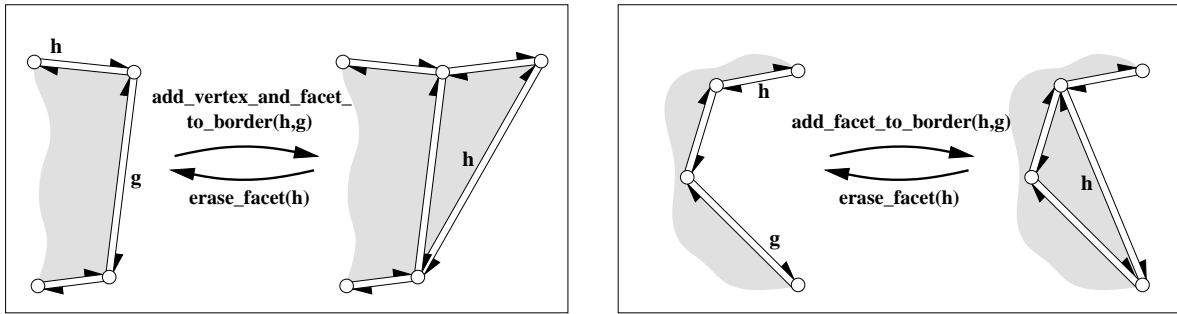
Precondition: None of the incident halfedges of the facet is a border edge.

Requirement: *Supports_removal* \equiv *CGAL::Tag_true*.

Halfedge_handle *P.fill_hole(Halfedge_handle h)*

fills a hole with a newly created facet. Makes all border halfedges of the hole denoted by *h* incident to the new facet. Returns *h*.

Precondition: *h.is_border()*.



Halfedge_handle *P.add_vertex_and_facet_to_border(Halfedge_handle h, Halfedge_handle g)*

creates a new facet within the hole incident to *h* and *g* by connecting the tip of *g* with the tip of *h* with two new halfedges and a new vertex and filling this separated part of the hole with a new facet, such that the new facet is incident to *g*. Returns the halfedge of the new edge that is incident to the new facet and the new vertex.

Precondition: *h->is_border()*, *g->is_border()*, *h != g*, and *g* can be reached along the same hole starting with *h*.

Halfedge_handle *P.add_facet_to_border(Halfedge_handle h, Halfedge_handle g)*

creates a new facet within the hole incident to *h* and *g* by connecting the vertex denoted by *g* with the vertex denoted by *h* with a new halfedge and filling this separated part of the hole with a new facet, such that the new facet is incident to *g*. Returns the halfedge of the new edge that is incident to the new facet.

Precondition: *h->is_border()*, *g->is_border()*, *h != g*, *h->next() != g*, and *g* can be reached along the same hole starting with *h*.

Erasing

<i>void</i>	<i>P.erase_facet(Halfedge_handle h)</i> removes the incident facet of <i>h</i> and changes all halfedges incident to the facet into border edges or removes them from the polyhedral surface if they were already border edges. If this creates isolated vertices they get removed as well. See <i>make_hole(h)</i> for a more specialized variant. <i>Precondition:</i> <i>h->is_border() == false</i> . <i>Requirement:</i> <i>Supports_removal</i> \equiv <i>CGAL::Tag_true</i> .
<i>void</i>	<i>P.erase_connected_component(Halfedge_handle h)</i> removes the vertices, halfedges, and facets that belong to the connected component of <i>h</i> . <i>Requirement:</i> <i>Supports_removal</i> \equiv <i>CGAL::Tag_true</i> .
<i>void</i>	<i>P.clear()</i> removes all vertices, halfedges, and facets.

Operations with Border Halfedges

— *advanced* —

Halfedges incident to a hole are called *border halfedges*. An halfedge is a *border edge* if itself or its opposite halfedge are border halfedges. The only requirement to work with border halfedges is that the *Halfedge* class provides a member function *is_border()* returning a *bool*. Usually, the halfedge data structure supports facets and a *NULL* facet pointer will indicate a border halfedge, but this is not the only possibility. The *is_border()* predicate divides the edges into two classes, the border edges and the non-border edges. The following normalization reorganizes the sequential storage of the edges such that the non-border edges precede the border edges, and that for each border edge the latter one of the two halfedges is a border halfedge (the first one is a non-border halfedge in conformance with the polyhedral surface definition). The normalization stores the number of border halfedges and the halfedge iterator the border edges start at within the data structure. Halfedge insertion or removal and changing the border status of a halfedge invalidate these values. They are not automatically updated.

<i>void</i>	<i>P.normalize_border()</i> sorts halfedges such that the non-border edges precede the border edges. For each border edge the halfedge iterator will reference the halfedge incident to the facet right before the halfedge incident to the hole.
<i>size_type</i>	<i>P.size_of_border_halfedges()</i> number of border halfedges. <i>Precondition:</i> last <i>normalize_border()</i> call still valid, see above.
<i>size_type</i>	<i>P.size_of_border_edges()</i> number of border edges. Since each border edge of a polyhedral surface has exactly one border halfedge, this number is equal to <i>size_of_border_halfedges()</i> . <i>Precondition:</i> last <i>normalize_border()</i> call still valid, see above.

Halfedge_iterator *P.border_halfedges_begin()*

halfedge iterator starting with the border edges. The range [*halfedges_begin()*, *border_halfedges_begin()*) denotes all non-border halfedges. The range [*border_halfedges_begin()*, *halfedges_end()*) denotes all border edges.
Precondition: last *normalize_border()* call still valid, see above.

Edge_iterator *P.border_edges_begin()*

edge iterator starting with the border edges. The range [*edges_begin()*, *border_edges_begin()*) denotes all non-border edges. The range [*border_edges_begin()*, *edges_end()*) denotes all border edges.
Precondition: last *normalize_border()* call still valid, see above.

└────────── *advanced* ─────────┘

Miscellaneous

void *P.inside_out()*

reverses facet orientations (incl. plane equations if supported).

bool *P.is_valid(bool verbose = false, int level = 0)*

returns *true* if the polyhedral surface is combinatorially consistent. If *verbose* is *true*, statistics are printed to *cerr*. For *level == 1* the normalization of the border edges is checked too. This method checks in particular level 3 of *CGAL::Halfedge_data_structure_decorator::is_valid* from page 11.4 and that each facet is at least a triangle and that the two incident facets of a non-border edge are distinct.

bool *P.normalized_border_is_valid(bool verbose = false)*

returns *true* if the border halfedges are in normalized representation, which is when enumerating all halfedges with the iterator: The non-border edges precede the border edges and for border edges, the second halfedge is the border halfedge. The halfedge iterator *border_halfedges_begin()* denotes the first border edge. If *verbose* is *true*, statistics are printed to *cerr*.

advanced

```
void P.delegate( CGAL::Modifier_base<HDS>& m)
```

calls the *operator()* of the modifier *m*. See *CGAL::Modifier_base* in the Support Library Manual for a description of modifier design and its usage.

Precondition: The polyhedral surface must be valid when the modifier returns from execution.

advanced

See Also

CGAL::Polyhedron_3<Traits>::Vertex page [816](#)
CGAL::Polyhedron_3<Traits>::Halfedge page [813](#)
CGAL::Polyhedron_3<Traits>::Facet page [811](#)
PolyhedronTraits_3 page [827](#)
CGAL::Polyhedron_traits_3<Kernel> page [828](#)
PolyhedronItems_3 page [822](#)
CGAL::Polyhedron_items_3 page [824](#)
HalfedgeDS page [847](#)
CGAL::HalfedgeDS_default page [872](#)
CGAL::Polyhedron_incremental_builder_3<HDS> page [818](#)
CGAL::Modifier_base in the Support Library Reference Manual.

Example

This example program instantiates a polyhedron using the default traits class and creates a tetrahedron.

```
// file: examples/Polyhedron/polyhedron_prog_simple.C

#include <CGAL/Simple_cartesian.h>
#include <CGAL/Polyhedron_3.h>

typedef CGAL::Simple_cartesian<double>      Kernel;
typedef CGAL::Polyhedron_3<Kernel>          Polyhedron;
typedef Polyhedron::Halfedge_handle          Halfedge_handle;

int main() {
    Polyhedron P;
    Halfedge_handle h = P.make_tetrahedron();
    if ( P.is_tetrahedron(h) )
        return 0;
    return 1;
}
```

CGAL::Polyhedron_3<Traits>::Facet

Definition

A facet optionally stores a plane equation, and a reference to an incident halfedge that points to the facet. Type tags indicate whether these member functions are supported. Figure 10.1 on page 813 depicts the relationship between a halfedge and its incident halfedges, vertices, and facets. The circulator is assignable to the *Halfedge_handle*. The circulator is bidirectional if the halfedge provided to the polyhedron with the *Items* template argument provides a member function *prev()*, otherwise it is of the forward category.

```
#include <CGAL/Polyhedron_3.h>
```

Types

<i>Facet:: Vertex</i>	type of incident vertices.
<i>Facet:: Halfedge</i>	type of incident halfedges.
<i>Facet:: Plane_3</i>	plane equation type stored in facets.

<i>Facet::Vertex_handle</i>	handle to vertex.
<i>Facet::Halfedge_handle</i>	handle to halfedge.
<i>Facet::Facet_handle</i>	handle to facet.
<i>Facet::Halfedge_around_facet_circulator</i>	circulator of halfedges around a facet.

```
Facet:: Vertex_const_handle
Facet:: Halfedge_const_handle
Facet:: Facet_const_handle
Facet:: Halfedge_around_facet_const_circulator
```

<i>Facet::Supports_facet_halfedge</i>	\equiv <i>CGAL::Tag_true</i> or <i>CGAL::Tag_false</i> .
<i>Facet::Supports_facet_plane</i>	\equiv <i>CGAL::Tag_true</i> or <i>CGAL::Tag_false</i> .

Creation

Facet f ; default constructor.

Operations available if *Supports_facet_plane* \equiv *CGAL::Tag_true*

<i>Plane_3</i> &	<i>f.plane()</i>	
<i>const Plane_3</i> &	<i>f.plane() const</i>	plane equation.

Operations available if `Supports_facet_halfedge` \equiv `CGAL::Tag_true`

<i>Halfedge_handle</i>	<i>f.halfedge()</i>	
<i>Halfedge_const_handle</i>	<i>f.halfedge() const</i>	an incident halfedge that points to <i>f</i> .

Halfedge_around_facet_circulator

f.facet_begin()

Halfedge_around_facet_const_circulator

f.facet_begin() const

circulator of halfedges around the facet (counterclockwise).

void

f.set_halfedge(Halfedge_handle h)

sets incident halfedge to *h*.

Precondition: *h* is incident, i.e., *h->facet() == f*.

std::size_t

f.facet_degree() const

the degree of the facet, i.e., number of edges on the boundary of this facet.

bool

f.is_triangle() const

returns *true* if the facet is a triangle.

bool

f.is_quad() const

returns *true* if the facet is a quadrilateral.

See Also

CGAL::Polyhedron_3<Traits>::Vertex page [816](#)

CGAL::Polyhedron_3<Traits>::Halfedge page [813](#)

CGAL::Polyhedron_3<Traits> page [799](#)

CGAL::Polyhedron_3<Traits>::Halfedge

Definition

Figure 10.1 on page 813 depicts the relationship between a halfedge and its incident halfedges, vertices, and facets. A halfedge is an oriented edge between two vertices. It is always paired with a halfedge pointing in the opposite direction. The *opposite()* member function returns this halfedge of opposite orientation. If a halfedge is incident to a facet the *next()* member function points to the successor halfedge around this facet. For border edges the *next()* member function points to the successor halfedge along the hole. For more than two border edges at a vertex, the next halfedge along a hole is not uniquely defined, but a consistent assignment of the next halfedge will be maintained in the data structure. An invariant is that successive assignments of the form $h = h \rightarrow \text{next}()$ cycle counterclockwise around the facet (or hole) and traverse all halfedges incident to this facet (or hole). A similar invariant is that successive assignments of the form $h = h \rightarrow \text{next}() \rightarrow \text{opposite}()$ cycle clockwise around the vertex and traverse all halfedges incident to this vertex. Two circulators are provided for these circular orders.

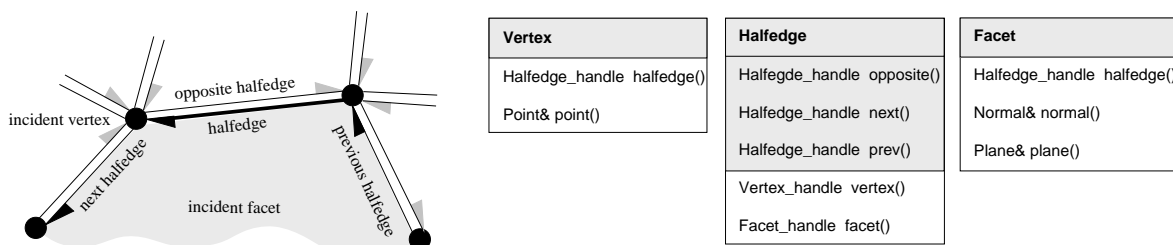


Figure 10.1: The three classes *Vertex*, *Halfedge*, and *Facet* of the polyhedral surface. Member functions with shaded background are mandatory. The others are optionally supported.

The incidences encoded in *opposite()* and *next()* are available for each instantiation of polyhedral surfaces. The other incidences are optionally available as indicated with type tags. The *prev()* member function points to the preceding halfedge around the same facet. It is always available, though it might perform a search around the facet using the *next()* member function to find the previous halfedge if the underlying halfedge data structure does not provide an efficient *prev()* member function for halfedges. Handles to the incident vertex and facet are optionally stored.

The circulators are assignable to the *Halfedge_handle*. The circulators are bidirectional if the halfedge provided to the polyhedron with the *Items* template argument provides a member function *prev()*, otherwise they are of the forward category.

```
#include <CGAL/Polyhedron_3.h>
```

Types

Halfedge::Vertex

Halfedge::Facet

Halfedge::Vertex_handle

Halfedge::Halfedge_handle

Halfedge::Facet_handle

Halfedge::Halfedge_around_vertex_circulator

Halfedge::Halfedge_around_facet_circulator

type of incident vertices.

type of incident facets.

handle to vertex.

handle to halfedge.

handle to facet.

circulator of halfedges around a vertex.

circulator of halfedges around a facet.

Halfedge:: Vertex_const_handle
Halfedge:: Halfedge_const_handle
Halfedge:: Facet_const_handle
Halfedge:: Halfedge_around_vertex_const_circulator
Halfedge:: Halfedge_around_facet_const_circulator

Halfedge:: Supports_halfedge_prev \equiv *CGAL::Tag_true* or *CGAL::Tag_false*.
Halfedge:: Supports_halfedge_vertex \equiv *CGAL::Tag_true* or *CGAL::Tag_false*.
Halfedge:: Supports_halfedge_face \equiv *CGAL::Tag_true* or *CGAL::Tag_false*.

Creation

Halfedge h; default constructor.

Operations

Halfedge_handle *h.opposite()*
Halfedge_const_handle *h.opposite() const* the opposite halfedge.

Halfedge_handle *h.next()*
Halfedge_const_handle *h.next() const* the next halfedge around the facet.

Halfedge_handle *h.prev()*
Halfedge_const_handle *h.prev() const* the previous halfedge around the facet.

Halfedge_handle *h.next_on_vertex()*
Halfedge_const_handle *h.next_on_vertex() const* the next halfedge around the vertex (clockwise). Is equal to *h.next()->opposite()*.

Halfedge_handle *h.prev_on_vertex()*
Halfedge_const_handle *h.prev_on_vertex() const* the previous halfedge around the vertex (counterclockwise). Is equal to *h.opposite()->prev()*.

bool *h.is_border() const* is true if *h* is a border halfedge.
bool *h.is_border_edge() const* is true if *h* or *h.opposite()* is a border halfedge.

Halfedge_around_vertex_circulator

h.vertex_begin()

Halfedge_around_vertex_const_circulator

h.vertex_begin() const circulator of halfedges around the vertex (clockwise).

Halfedge_around_facet_circulator

h.facet_begin()

Halfedge_around_facet_const_circulator

	<i>h.facet_begin() const</i>	circulator of halfedges around the facet (counterclockwise).
<i>std::size_t</i>	<i>h.vertex_degree() const</i>	the degree of the incident vertex, i.e., number of edges emanating from this vertex.
<i>bool</i>	<i>h.is_bivalent() const</i>	returns <i>true</i> if the incident vertex has exactly two incident edges.
<i>bool</i>	<i>h.is_trivalent() const</i>	returns <i>true</i> if the incident vertex has exactly three incident edges.
<i>std::size_t</i>	<i>h.facet_degree() const</i>	the degree of the incident facet, i.e., number of edges on the boundary of this facet.
<i>bool</i>	<i>h.is_triangle() const</i>	returns <i>true</i> if the incident facet is a triangle.
<i>bool</i>	<i>h.is_quad() const</i>	returns <i>true</i> if the incident facet is a quadrilateral.

Operations available if *Supports_halfedge_vertex* \equiv *CGAL::Tag_true*

<i>Vertex_handle</i>	<i>h.vertex()</i>	
<i>Vertex_const_handle</i>	<i>h.vertex() const</i>	the incident vertex of <i>h</i> .

Operations available if *Supports_halfedge_facet* \equiv *CGAL::Tag_true*

<i>Facet_handle</i>	<i>h.facet()</i>	
<i>Facet_const_handle</i>	<i>h.facet() const</i>	the incident facet of <i>h</i> . If <i>h</i> is a border halfedge the result is default construction of the handle.

See Also

<i>CGAL::Polyhedron_3<Traits>::Vertex</i>	page 816
<i>CGAL::Polyhedron_3<Traits>::Facet</i>	page 811
<i>CGAL::Polyhedron_3<Traits></i>	page 799

Implementation

The member functions *prev()* and *prev_on_vertex()* work in constant time if *Supports_halfedge_prev* \equiv *CGAL::Tag_true*. Otherwise both methods search for the previous halfedge around the incident facet.

CGAL::Polyhedron_3<Traits>::Vertex

Definition

A vertex optionally stores a point and a reference to an incident halfedge that points to the vertex. Type tags indicate whether these member functions are supported. Figure 10.1 on page 813 depicts the relationship between a halfedge and its incident halfedges, vertices, and facets. The circulator is assignable to the *Halfedge_handle*. The circulator is bidirectional if the halfedge provided to the polyhedron with the *Items* template argument provides a member function *prev()*, otherwise it is of the forward category.

```
#include <CGAL/Polyhedron_3.h>
```

Types

<i>Vertex:: Halfedge</i>	type of incident halfedges.
<i>Vertex:: Facet</i>	type of incident facets.
<i>Vertex:: Point_3</i>	point type stored in vertices.
<i>Vertex:: Vertex_handle</i>	handle to vertex.
<i>Vertex:: Halfedge_handle</i>	handle to halfedge.
<i>Vertex:: Facet_handle</i>	handle to facet.
<i>Vertex:: Halfedge_around_vertex_circulator</i>	circulator of halfedges around a vertex.
<i>Vertex:: Vertex_const_handle</i>	
<i>Vertex:: Halfedge_const_handle</i>	
<i>Vertex:: Facet_const_handle</i>	
<i>Vertex:: Halfedge_around_vertex_const_circulator</i>	
<i>Vertex:: Supports_vertex_halfedge</i>	\equiv <i>CGAL::Tag_true</i> or <i>CGAL::Tag_false</i> .
<i>Vertex:: Supports_vertex_point</i>	\equiv <i>CGAL::Tag_true</i> or <i>CGAL::Tag_false</i> .

Creation

<i>Vertex v</i> ;	default constructor.
<i>Vertex v</i> (<i>Point p</i>);	vertex initialized with a point.

Operations available if *Supports_vertex_point* \equiv *CGAL::Tag_true*

<i>Point_3&</i>	<i>v.point()</i>	
<i>const Point_3&</i>	<i>v.point() const</i>	the point.

Operations available if *Supports_vertex_halfedge* \equiv *CGAL::Tag_true*

Halfedge_handle *v.halfedge()*
Halfedge_const_handle *v.halfedge() const* an incident halfedge that points to *v*.

Halfedge_around_vertex_circulator
 v.vertex_begin()

Halfedge_around_vertex_const_circulator
 v.vertex_begin() const
 circulator of halfedges around the vertex (clockwise).

void *v.set_halfedge(Halfedge_handle h)*
 sets incident halfedge to *h*.
 Precondition: *h* is incident, i.e., *h->vertex() == v*.

std::size_t *v.vertex_degree() const*
 the degree of the vertex, i.e., number of edges emanating from
 this vertex.

bool *v.is_bivalent() const*
 returns *true* if the vertex has exactly two incident edges.

bool *v.is_trivalent() const*
 returns *true* if the vertex has exactly three incident edges.

See Also

CGAL::Polyhedron_3<Traits>::Halfedge [page 813](#)
CGAL::Polyhedron_3<Traits>::Facet [page 811](#)
CGAL::Polyhedron_3<Traits> [page 799](#)

CGAL::Polyhedron_incremental_builder_3<HDS>

Definition

The auxiliary class *Polyhedron_incremental_builder_3<HDS>* supports the incremental construction of polyhedral surfaces, which is for example convenient when constructing polyhedral surfaces from file formats, such as the Object File Format (OFF) [Phi96], OpenInventor [Wer94] or VRML [BPP95, VRM96]. *Polyhedron_incremental_builder_3<HDS>* needs access to the internal halfedge data structure of type *HDS* of the polyhedral surface. It is intended to be used within a modifier, see *CGAL::Modifier_base* in the Support Library Reference Manual.

The incremental builder might be of broader interest for other uses of the halfedge data structures, but it is specifically bound to the definition of polyhedral surfaces given here. During construction all conditions of polyhedral surfaces are checked and in case of violation an error status is set. A diagnostic message will be issued to *cerr* if the *verbose* flag has been set at construction time.

The incremental construction starts with a list of all point coordinates and concludes with a list of all facet polygons. Edges are not explicitly specified. They are derived from the vertex incidence information provided from the facet polygons. The polygons are given as a sequence of vertex indices. The halfedge data structure *HDS* must support vertices (i.e., *Supports_halfedge_vertex* \equiv *CGAL::Tag_true*). Vertices and facets can be added in arbitrary order as long as a call to *add_vertex_to_facet()* refers only to a vertex index that is already known. Some methods return already handles to vertices, facets, and halfedges newly constructed. They can be used to initialize additional fields, however, the incidences in the halfedge-data structure are not stable and are not allowed to be changed.

The incremental builder can work in two modes: *RELATIVE_INDEXING* (the default), in which a polyhedral surface already contained in the halfedge data structure is ignored and all indices are relative to the newly added surface, or *ABSOLUTE_INDEXING*, in which all indices are absolute indices including an already existing polyhedral surface. The former mode allows to create easily independent connected components, while the latter mode allows to continue the construction of an existing surface, the absolute indexing allows to address existing vertices when creating new facets.

```
#include <CGAL/Polyhedron_incremental_builder_3.h>
```

Types

<i>Polyhedron_incremental_builder_3<HDS>::HalfedgeDS</i>	halfedge data structure <i>HDS</i> .
<i>Polyhedron_incremental_builder_3<HDS>::Point_3</i>	point type of the vertex.
<i>Polyhedron_incremental_builder_3<HDS>::size_type</i>	size type.
<i>typedef typename HalfedgeDS::Vertex_handle</i>	<i>Vertex_handle</i> ;
<i>typedef typename HalfedgeDS::Halfedge_handle</i>	<i>Halfedge_handle</i> ;
<i>typedef typename HalfedgeDS::Face_handle</i>	<i>Facet_handle</i> ;

Constants

```
enum { RELATIVE_INDEXING, ABSOLUTE_INDEXING };
```

two different indexing modes.

Creation

Polyhedron_incremental_builder_3<HDS> *B*(*HDS*& *hds*, *bool verbose* = *false*);

stores a reference to the halfedge data structure *hds* of a polyhedral surface in its internal state. An existing polyhedral surface in *hds* remains unchanged. The incremental builder appends the new polyhedral surface. If *verbose* is *true*, diagnostic messages will be printed to *cerr* in case of malformed input data.

Surface Creation

To build a polyhedral surface, the following regular expression gives the correct and allowed order and nesting of method calls from this section:

begin_surface (*add_vertex* | (*begin_facet add_vertex_to_facet* * *end_facet*)) * *end_surface*

void *B.begin_surface*(*size_type v*,
 size_type f,
 size_type h = 0,
 int mode = *RELATIVE_INDEXING*)

starts the construction. *v* is the number of new vertices to expect, *f* the number of new facets, and *h* the number of new halfedges. If *h* is unspecified (*== 0*) it is estimated using Euler's equation (plus 5% for the so far unknown holes and genus of the object). These values are used to reserve space in the halfedge data structure *hds*. If the representation supports insertion these values do not restrict the class of constructible polyhedra. If the representation does not support insertion the object must fit into the reserved sizes. If *mode* is set to *ABSOLUTE_INDEXING* the incremental builder uses absolute indexing and the vertices of the old polyhedral surface can be used in new facets (needs preprocessing time linear in the size of the old surface). Otherwise relative indexing is used starting with new indices for the new construction.

Vertex_handle *B.add_vertex*(*Point_3 p*)

adds a new vertex for *p* and returns its handle.

Facet_handle *B.begin_facet*() starts a new facet and returns its handle.

void *B.add_vertex_to_facet*(*size_type i*)

adds a vertex with index *i* to the current facet. The first point added with *add_vertex()* has the index 0 if *mode* was set to *RELATIVE_INDEXING*, otherwise the first vertex in the referenced *hds* has the index 0.

<i>Halfedge_handle</i>	<i>B.end_facet()</i>	ends a newly constructed facet. Returns the handle to the halfedge incident to the new facet that points to the vertex added first. The halfedge can be safely used to traverse the halfedge cycle around the new facet.
------------------------	----------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<i>void</i>	<i>B.end_surface()</i>	ends the construction.
-------------	------------------------	------------------------

Additional Operations

template <class InputIterator>

Halfedge_handle B.add_facet(InputIterator first, InputIterator beyond)

is a synonym for *begin_facet()*, a call to *add_facet()* for each value in the range *[first,beyond)*, and a call to *end_facet()*. Returns the return value of *end_facet()*.

Precondition: The value type of *InputIterator* is *std::size_t*. All indices must refer to vertices already added.

template <class InputIterator>

bool B.test_facet(InputIterator first, InputIterator beyond)

returns *true* if a facet described by the vertex indices in the range *[first,beyond)* can be successfully inserted, e.g., with *add_facet(first,beyond)*.

Precondition: The value type of *InputIterator* is *std::size_t*. All indices must refer to vertices already added.

<i>Vertex_handle</i>	<i>B.vertex(std::size_t i)</i>	
----------------------	---------------------------------	--

returns handle for the vertex of index *i*, or *Vertex_handle* if there is no *i*-th vertex.

<i>bool</i>	<i>B.error()</i>	returns error status of the builder.
-------------	------------------	--------------------------------------

<i>void</i>	<i>B.rollback()</i>	undoes all changes made to the halfedge data structure since the last <i>begin_surface()</i> in relative indexing, and deletes the whole surface in absolute indexing. It needs a new call to <i>begin_surface()</i> to start inserting again.
-------------	---------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<i>bool</i>	<i>B.check_unconnected_vertices()</i>	returns <i>true</i> if unconnected vertices are detected. If <i>verbose</i> was set to <i>true</i> (see the constructor above) debug information about the unconnected vertices is printed.
-------------	---------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<i>bool</i>	<i>B.remove_unconnected_vertices()</i>	returns <i>true</i> if all unconnected vertices could be removed successfully. This happens either if no unconnected vertices had appeared or if the halfedge data structure supports the removal of individual elements.
-------------	----------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

See Also

CGAL::Polyhedron_3<Traits> page [799](#)
HalfedgeDS page [847](#)
CGAL::Modifier_base in the Support Library Reference Manual.

Example

A modifier class creates a new triangle in the halfedge data structure using the incremental builder.

```
// file: examples/Polyhedron/polyhedron_prog_incr_builder.C

#include <CGAL/Simple_cartesian.h>
#include <CGAL/Polyhedron_incremental_builder_3.h>
#include <CGAL/Polyhedron_3.h>

// A modifier creating a triangle with the incremental builder.
template <class HDS>
class Build_triangle : public CGAL::Modifier_base<HDS> {
public:
    Build_triangle() {}
    void operator()( HDS& hds) {
        // Postcondition: 'hds' is a valid polyhedral surface.
        CGAL::Polyhedron_incremental_builder_3<HDS> B( hds, true);
        B.begin_surface( 3, 1, 6);
        typedef typename HDS::Vertex    Vertex;
        typedef typename Vertex::Point Point;
        B.add_vertex( Point( 0, 0, 0));
        B.add_vertex( Point( 1, 0, 0));
        B.add_vertex( Point( 0, 1, 0));
        B.begin_facet();
        B.add_vertex_to_facet( 0);
        B.add_vertex_to_facet( 1);
        B.add_vertex_to_facet( 2);
        B.end_facet();
        B.end_surface();
    }
};

typedef CGAL::Simple_cartesian<double>    Kernel;
typedef CGAL::Polyhedron_3<Kernel>        Polyhedron;
typedef Polyhedron::HalfedgeDS            HalfedgeDS;

int main() {
    Polyhedron P;
    Build_triangle<HalfedgeDS> triangle;
    P.delegate( triangle);
    CGAL_assertion( P.is_triangle( P.halfedges_begin()));
    return 0;
}
```

PolyhedronItems_3

Definition

The `PolyhedronItems_3` concept extends the `HalfedgeDSItems` concept on page 879. In addition to the requirements stated there, a model for this concept must fulfill the following requirements for the local `PolyhedronItems_3::Vertex_wrapper<Refs,Traits>::Vertex` type and `PolyhedronItems_3::Face_wrapper<Refs,Traits>::Face` type in order to support the point for vertices and the optional plane equation for facets. Note that the items class uses face instead of facet. Only the polyhedral surface renames faces to facets.

Refines

`HalfedgeDSItems` page 859

Types in `PolyhedronItems_3::Vertex_wrapper<Refs,Traits>::Vertex`

`Vertex::Point` point type stored in vertices. A `HalfedgeDS` has no dimension, so this type is named `Point` and not `Point_3`.

`Vertex::Supports_vertex_point` \equiv `CGAL::Tag_true`. A point is always required.

Operations

<code>Point&</code>	<code>v.point()</code>	
<code>const Point&</code>	<code>v.point() const</code>	point.

Types in `PolyhedronItems_3::Face_wrapper<Refs,Traits>::Face`

Types for (optionally) associated geometry in faces. If it is not supported the respective type has to be defined, although it can be an arbitrary dummy type, such as `void*` or `Tag_false`.

`Face::Plane` plane type stored in faces. A `HalfedgeDS` has no dimension, so this type is named `Plane` and not `Plane_3`.

`Face::Supports_face_plane` either `CGAL::Tag_true` or `CGAL::Tag_false`.

Operations required if `Supports_face_plane \equiv CGAL::Tag_true`

<code>Plane&</code>	<code>f.plane()</code>	
<code>const Plane&</code>	<code>f.plane() const</code>	plane equation.

Has Models

`CGAL::Polyhedron_items_3` page 824

`CGAL::Polyhedron_min_items_3` page 826

See Also

<i>CGAL::Polyhedron_3</i> <Traits>	page 799
<i>HalfedgeDSItems</i>	page 859
<i>CGAL::HalfedgeDS_items_2</i>	page 879
<i>CGAL::HalfedgeDS_vertex_base</i> <Refs>	page 889
<i>CGAL::HalfedgeDS_halfedge_base</i> <Refs>	page 877
<i>CGAL::HalfedgeDS_face_base</i> <Refs>	page 874

Example

We define our own items class based on the available *CGAL::HalfedgeDS_face_base* base class for faces. We derive the *Halfedge_wrapper* without further modifications from the *CGAL::HalfedgeDS_items_2*, replace the *Face_wrapper* definition with our new definition, and also replace the *Vertex_wrapper* with a definition that uses *Point_3* instead of *Point_2* as point type. The result is a model for the *PolyhedronItems_3* concept similar to the available *CGAL::Polyhedron_items_3* class. See also there for another illustrative example.

```
#include <CGAL/HalfedgeDS_bases.h>

struct My_items : public CGAL::HalfedgeDS_items_2 {
    template < class Refs, class Traits>
    struct Vertex_wrapper {
        typedef typename Traits::Point_3 Point;
        typedef CGAL::HalfedgeDS_vertex_base< Refs, CGAL::Tag_true, Point> Vertex;
    };
    template < class Refs, class Traits>
    struct Face_wrapper {
        typedef typename Traits::Plane_3 Plane;
        typedef CGAL::HalfedgeDS_face_base< Refs, CGAL::Tag_true, Plane> Face;
    };
};
```

CGAL::Polyhedron_items_3

Definition

The class *Polyhedron_items_3* is a model of the *PolyhedronItems_3* concept. It provides definitions for vertices with points, halfedges, and faces with plane equations. The polyhedron traits class must provide the respective types for the point and the plane equation. Vertices and facets both contain a halfedge handle to an incident halfedge.

```
#include <CGAL/Polyhedron_items_3.h>
```

Is Model for the Concepts

PolyhedronItems_3 page [822](#)

Types in *Polyhedron_items_3::Vertex_wrapper<Refs,Traits>::Vertex*

```
typedef Traits::Point_3      Point;
typedef CGAL::Tag_true      Supports_vertex_point;
```

Types in *Polyhedron_items_3::Face_wrapper<Refs,Traits>::Face*

```
typedef Traits::Plane_3      Plane;
typedef CGAL::Tag_true      Supports_face_plane;
```

Creation

Polyhedron_items_3 items; default constructor.

Operations

Supported as required by the *PolyhedronItems_3* concept.

See Also

CGAL::Polyhedron_3<Traits> page [799](#)
CGAL::Polyhedron_min_items_3 page [826](#)
CGAL::HalfedgeDS_min_items page [885](#)
CGAL::HalfedgeDS_items_2 page [879](#)

Example

The following example program defines a new face class based on the *CGAL::HalfedgeDS_face_base* and adds a new color member variable. The new face class is used to replace the face definition in the *CGAL::Polyhedron_items_3* class. The main function illustrates the access to the new member variable. See also the *PolyhedronItems_3* concept for another illustrative example.


```

// file: examples/Polyhedron/polyhedron_prog_color.C

#include <CGAL/Simple_cartesian.h>
#include <CGAL/IO/Color.h>
#include <CGAL/Polyhedron_3.h>

// A face type with a color member variable.
template <class Refs>
struct My_face : public CGAL::HalfedgeDS_face_base<Refs> {
    CGAL::Color color;
};

// An items type using my face.
struct My_items : public CGAL::Polyhedron_items_3 {
    template <class Refs, class Traits>
    struct Face_wrapper {
        typedef My_face<Refs> Face;
    };
};

typedef CGAL::Simple_cartesian<double>          Kernel;
typedef CGAL::Polyhedron_3<Kernel, My_items>    Polyhedron;
typedef Polyhedron::Halfedge_handle             Halfedge_handle;

int main() {
    Polyhedron P;
    Halfedge_handle h = P.make_tetrahedron();
    h->facet()->color = CGAL::RED;
    return 0;
}

```

CGAL::Polyhedron_min_items_3

Definition

The class *Polyhedron_min_items_3* is a minimal model of the *PolyhedronItems_3* concept. It provides definitions for vertices containing points, halfedges, and faces. The polyhedron traits class must provide the respective type for the point. Vertices and facets both do *not* contain a halfedge handle to an incident halfedge.

```
#include <CGAL/Polyhedron_min_items_3.h>
```

Is Model for the Concepts

PolyhedronItems_3 page [822](#)

Types in *Polyhedron_min_items_3::Vertex_wrapper<Refs,Traits>::Vertex*

```
typedef Traits::Point_3      Point;
typedef CGAL::Tag_true      Supports_vertex_point;
```

Types in *Polyhedron_min_items_3::Face_wrapper<Refs,Traits>::Face*

```
typedef CGAL::Tag_false      Supports_face_plane;
```

Creation

Polyhedron_min_items_3 items; default constructor.

Operations

Supported as required by the *PolyhedronItems_3* concept.

See Also

CGAL::Polyhedron_3<Traits> page [799](#)
CGAL::Polyhedron_items_3 page [824](#)
CGAL::HalfedgeDS_min_items page [885](#)
CGAL::HalfedgeDS_items_2 page [879](#)

PolyhedronTraits_3

Required types and member functions for the PolyhedronTraits_3 concept. This geometric traits concept is used in the polyhedral surface data structure *CGAL::Polyhedron_3<Traits>*. This concept is a subset of the 3d kernel traits and any CGAL kernel model can be used directly as template argument.

Refines

CopyConstructable, *Assignable*.

Types

PolyhedronTraits_3::Point_3 point type.

PolyhedronTraits_3::Plane_3 plane equation. Even if plane equations are not supported with a particular polyhedral surface this type has to be defined (some dummy type).

PolyhedronTraits_3::Construct_opposite_plane_3

is an unary function object that reverses the plane orientation. Must provide *Plane_3 operator()(Plane_3 plane)* that returns the reversed plane. Required only if plane equations are supported and the *inside_out()* method is used to reverse the polyhedral surface orientation.

Creation

PolyhedronTraits_3 traits(traits2); copy constructor.

PolyhedronTraits_3& traits = traits2 assignment.

Operations

Construct_opposite_plane_3 traits.construct_opposite_plane_3_object()

returns an instance of this function object.

Has Models

CGAL::Polyhedron_traits_3<Kernel> page 828

CGAL::Polyhedron_traits_with_normals_3<Kernel> page 830

All models of the *CGAL::Kernel* concept, e.g., *Simple_cartesian<FieldNumberType>*.

See Also

CGAL::Polyhedron_3<Traits> page 799

CGAL::Polyhedron_traits_3<Kernel>

Definition

The class *Polyhedron_traits_3<Kernel>* is a model of the *PolyhedronTraits_3* concept. It defines the geometric types and primitive operations used in the polyhedral surface data structure *CGAL::Polyhedron_3<PolyhedronTraits_3>* in terms of the CGAL *Kernel*. It keeps a local copy of the kernel which makes it suitable for kernels with local state.

```
#include <CGAL/Polyhedron_traits_3.h>
```

Is Model for the Concepts

PolyhedronTraits_3 page [827](#)

Types

<i>Polyhedron_traits_3<Kernel>::Kernel</i>	the <i>Kernel</i> model.
<i>typedef Kernel::Point_3</i>	<i>Point_3</i> ;
<i>typedef Kernel::Plane_3</i>	<i>Plane_3</i> ;
<i>typedef Kernel::Construct_opposite_plane_3</i>	<i>Construct_opposite_plane_3</i> ;

Creation

<i>Polyhedron_traits_3<Kernel> traits</i> ;	default constructor, uses <i>Kernel()</i> as local reference to the kernel.
---------------------------------------------------	-----------------------------------------------------------------------------

<i>Polyhedron_traits_3<Kernel> traits(Kernel kernel</i>);	stores <i>kernel</i> as local reference.
-------------------------------------------------------------------	------------------------------------------

Operations

<i>Construct_opposite_plane_3 traits.construct_opposite_plane_3_object()</i>	forwarded to <i>kernel</i> .
------------------------------------------------------------------------------	------------------------------

See Also

CGAL::Polyhedron_traits_with_normals_3<Kernel> page [830](#)

Implementation

Since the *PolyhedronTraits_3* concept is a subset of the 3D kernel concept, this class just forwards the relevant types and access member functions from its template argument. However, it is useful for testing sufficiency of requirements.

Example

Instantiation of a polyhedral surface with the Cartesian kernel based on double coordinates.

```

// file: examples/Polyhedron/polyhedron_prog_simple.C

#include <CGAL/Simple_cartesian.h>
#include <CGAL/Polyhedron_3.h>

typedef CGAL::Simple_cartesian<double>      Kernel;
typedef CGAL::Polyhedron_3<Kernel>         Polyhedron;
typedef Polyhedron::Halfedge_handle        Halfedge_handle;

int main() {
    Polyhedron P;
    Halfedge_handle h = P.make_tetrahedron();
    if ( P.is_tetrahedron(h) )
        return 0;
    return 1;
}

```

CGAL::Polyhedron_traits_with_normals_3<Kernel>

Definition

The class *Polyhedron_traits_with_normals_3<Kernel>* is a model of the *PolyhedronTraits_3* concept. It defines the geometric types and primitive operations used in the polyhedral surface data structure *CGAL::Polyhedron_3<PolyhedronTraits_3>*. *Polyhedron_traits_with_normals_3<Kernel>* uses the normal vector from *Kernel* for the plane equation in facets. It keeps a local copy of the kernel which makes it suitable for kernels with local state.

```
#include <CGAL/Polyhedron_traits_with_normals_3.h>
```

Is Model for the Concepts

PolyhedronTraits_3 page 827

Types

```

Polyhedron_traits_with_normals_3<Kernel>:: Kernel           the Kernel model.
typedef Kernel::Point_3           Point_3;
typedef Kernel::Vector_3         Plane_3;

```

```
typedef Kernel::Construct_opposite_vector_3 Construct_opposite_plane_3;
```

Creation

Polyhedron_traits_with_normals_3<*Kernel*> traits; default constructor, uses *Kernel*() as
local reference to the kernel.

Polyhedron_traits_with_normals_3<Kernel> traits(*Kernel kernel*); stores *kernel* as local reference.

Operations

<i>Construct_opposite_plane_3</i>	<i>traits.construct_opposite_plane_3_object()</i>	
		forwarded to <i>kernel</i> .

See Also

CGAL::Polyhedron_traits_3<Kernel> page 828

Example

We use this traits class to instantiate a polyhedral surface with a normal vector and no plane equation for each facet. We compute the normal vector assuming exact arithmetic (integers in this example) and convex planar facets.

```

// file: examples/Polyhedron/polyhedron_prog_normals.C

#include <CGAL/Homogeneous.h>
#include <CGAL/Polyhedron_traits_with_normals_3.h>
#include <CGAL/Polyhedron_3.h>
#include <iostream>
#include <algorithm>

struct Normal_vector {
    template <class Facet>
    typename Facet::Plane_3 operator()( Facet& f) {
        typename Facet::Halfedge_handle h = f.halfedge();
        // Facet::Plane_3 is the normal vector type. We assume the
        // CGAL Kernel here and use its global functions.
        return CGAL::cross_product(
            h->next()->vertex()->point() - h->vertex()->point(),
            h->next()->next()->vertex()->point() - h->next()->vertex()->point());
    }
};

typedef CGAL::Homogeneous<int> Kernel;
typedef Kernel::Point_3 Point_3;
typedef Kernel::Vector_3 Vector_3;
typedef CGAL::Polyhedron_traits_with_normals_3<Kernel> Traits;
typedef CGAL::Polyhedron_3<Traits> Polyhedron;

int main() {
    Point_3 p( 1, 0, 0);
    Point_3 q( 0, 1, 0);
    Point_3 r( 0, 0, 1);
    Point_3 s( 0, 0, 0);
    Polyhedron P;
    P.make_tetrahedron( p, q, r, s);
    std::transform( P.facets_begin(), P.facets_end(), P.planes_begin(),
        Normal_vector());
    CGAL::set_pretty_mode( std::cout);
    std::copy( P.planes_begin(), P.planes_end(),
        std::ostream_iterator<Vector_3>( std::cout, "\n"));
    return 0;
}

```

CGAL::operator<<

Definition

This operator writes the polyhedral surface P to the output stream out using the Object File Format, OFF, with file extension `.off`, which is also understood by GeomView [\[Phi96\]](#). The output is in ASCII format. From the polyhedral surface, only the point coordinates and facets are written. Neither normal vectors nor color attributes are used.

For OFF an ASCII and a binary format exist. The format can be selected with the CGAL modifiers for streams, `set_ascii_mode` and `set_binary_mode` respectively. The modifier `set_pretty_mode` can be used to allow for (a few) structuring comments in the output. Otherwise, the output would be free of comments. The default for writing is ASCII without comments.

```
#include <CGAL/IO/Polyhedron_iostream.h>
```

```
template <class PolyhedronTraits_3>
ostream& ostream& out << CGAL::Polyhedron_3<PolyhedronTraits_3> P
```

See Also

`CGAL::Polyhedron_3<Traits>` [page 799](#)
`operator>>` [page 833](#)

CGAL::operator>>

Definition

This operator reads a polyhedral surface in Object File Format, OFF, with file extension `.off`, which is also understood by GeomView [[Phi96](#)], from the input stream *in* and appends it to the polyhedral surface *P*. Only the point coordinates and facets from the input stream are used to build the polyhedral surface. Neither normal vectors nor color attributes are evaluated. If the stream *in* does not contain a permissible polyhedral surface the `ios::badbit` of the input stream *in* is set and *P* remains unchanged.

For OFF an ASCII and a binary format exist. The stream detects the format automatically and can read both.

```
#include <CGAL/IO/Polyhedron_iostream.h>
```

```
template <class PolyhedronTraits_3>
istream& istream& in >> CGAL::Polyhedron_3<PolyhedronTraits_3>& P
```

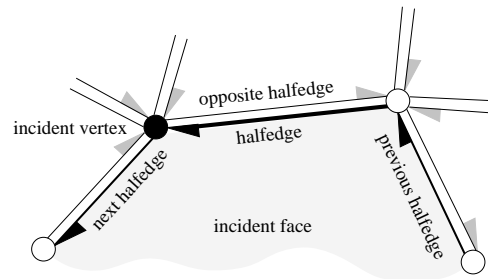
See Also

`CGAL::Polyhedron_3<Traits>` [page 799](#)
`CGAL::Polyhedron_incremental_builder_3<HDS>` [page 818](#)
`operator<<` [page 832](#)

Implementation

This operator is implemented using the modifier mechanism for polyhedral surfaces and the `CGAL::Polyhedron_incremental_builder_3` class, which allows the construction in a single, efficient scan pass of the input and handles also all the possible flexibility of the polyhedral surface.

Chapter 11



Halfedge Data Structures

Lutz Kettner

Contents

11.1 Introduction	835
11.2 Software Design	836
11.3 Example Programs	837
11.3.1 The Default Halfedge Data Structure	837
11.3.2 A Minimal Halfedge Data Structure	838
11.3.3 The Default with a Vector Instead of a List	838
11.3.4 Example Adding Color to Faces	839
11.3.5 Example Defining a More Compact Halfedge	840
11.3.6 Example Using the Halfedge Iterator	842
11.3.7 Example for an Adapter to Build an Edge Iterator	843

11.1 Introduction

A halfedge data structure (abbreviated as *HalfedgeDS*, or *HDS* for template parameters) is an edge-centered data structure capable of maintaining incidence informations of vertices, edges and faces, for example for planar maps, polyhedra, or other orientable, two-dimensional surfaces embedded in arbitrary dimension. Each edge is decomposed into two halfedges with opposite orientations. One incident face and one incident vertex are stored in each halfedge. For each face and each vertex, one incident halfedge is stored. Reduced variants of the halfedge data structure can omit some of these informations, for example the halfedge pointers in faces or the storage of faces at all.

The halfedge data structure is a combinatorial data structure, geometric interpretation is added by classes built on top of the halfedge data structure. These classes might be more convenient to use than the halfedge data structure directly, since the halfedge data structure is meant as an implementation layer. See for example the *CGAL::Polyhedron_3* class in Chapter 10.

The data structure provided here is also known as the FE-structure [Wei85], as halfedges [Män88, BFH95] or as the doubly connected edge list (DCEL) [dBvKOS97], although the original reference for the DCEL [MP78] describes a different data structure. The halfedge data structure can also be seen as one of the variants of the quad-edge data structure [GS85]. In general, the quad-edge data can represent non-orientable 2-manifolds, but

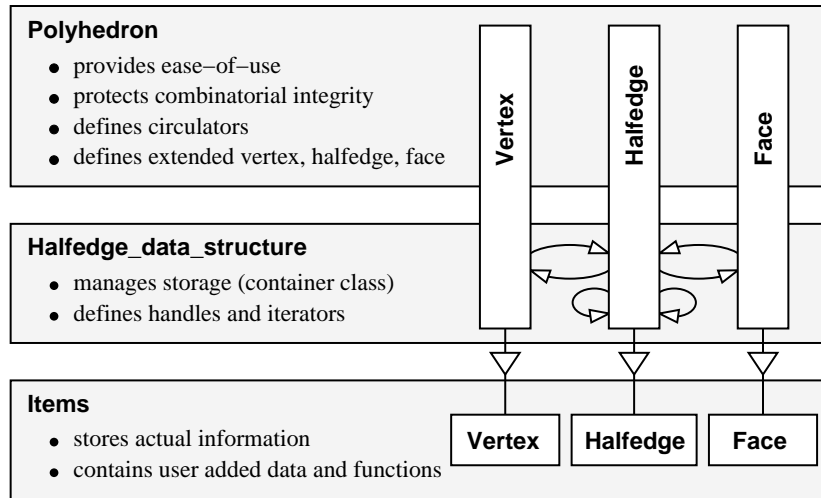


Figure 11.1: Responsibilities of the different layers in the halfedge data-structure design.

the variant here is restricted to orientable 2-manifolds only. An overview and comparison of these different data structures together with a thorough description of the design implemented here can be found in [Ket99].

The design presented here is a revised and incompatible version of the previous design [Ket98] as used in CGAL R2.2 and earlier releases. Files and identifier names are disjoint with the old design which allows for both versions to co-exists. However, classes using a halfedge data structure can only use one design. For example the polyhedral surface *Polyhedron_3* uses by default the new design. See Chapter 10 for how to still select the old implementation.

11.2 Software Design

Figure 11.1 illustrates the responsibilities of the three layers of the software design, with the *CGAL::Polyhedron_3* as an example for the top layer. The items provide the space for the information that is actually stored, i.e., with member variables and access member functions in *Vertex*, *Halfedge*, and *Face* respectively. Halfedges are required to provide a reference to the next halfedge and to the opposite halfedge. Optionally they may provide a reference to the previous halfedge, to the incident vertex, and to the incident face. Vertices and faces may be empty. Optionally they may provide a reference to the incident halfedge. The options mentioned are supported in the halfedge data structure and the polyhedron, for example, Euler operations update the optional references if they are present. Furthermore, the item classes can be extended with arbitrary attributes and member functions, which will be promoted by inheritance to the actual classes used for the polyhedron.

Vertices, halfedges, and faces are passed as local types of the *Items* class to the halfedge data structure and polyhedron. Implementations for vertices, halfedges and faces are provided that fulfill the mandatory part of the requirements. They can be used as base classes for extensions by the user. Richer implementations are also provided to serve as defaults; for polyhedra they provide all optional incidences, a three-dimensional point in the vertex type and a plane equation in the face type.

The *Halfedge_data_structure*, concept *HalfedgeDS*, is responsible for the storage organization of the items. Currently, implementations using internally a bidirectional list or a vector are provided. The *HalfedgeDS* defines the handles and iterators belonging to the items. These types are promoted to the declaration of the items themselves and are used there to provide the references to the incident items. This promotion of types is done

with a template parameter *Refs* of the item types. The halfedge data structure provides member functions to insert and delete items, to traverse all items, and it gives access to the items.

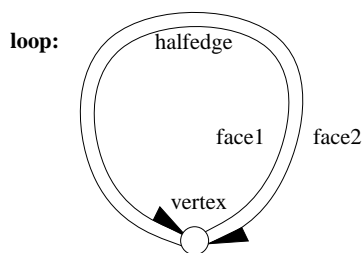
There are two different models for the *HalfedgeDS* concept available, *HalfedgeDS_list* and *HalfedgeDS_vector*, and more might come. Therefore we have kept their interface small and factored out common functionality into separate helper classes, *HalfedgeDS_decorator*, *HalfedgeDS_const_decorator*, and *HalfedgeDS_items_decorator*, which are not shown in Figure 11.1, but would be placed at the side of the *HalfedgeDS* since they broaden that interface but do not hide it. These helper classes contain operations that are useful to implement the operations in the next layer, for example, the polyhedron. They add, for example, the Euler operations and partial operations from which further Euler operations can be built, such as inserting an edge into the ring of edges at a vertex. Furthermore, the helper classes contain adaptive functionality. For example, if the *prev()* member function is not provided for halfedges, the *find_prev()* member function of a helper class searches in the positive direction along the face for the previous halfedge. But if the *prev()* member function is provided, the *find_prev()* member function simply calls it. This distinction is resolved at compile time with a technique called *compile-time tags*, similar to iterator tags in [SL95].

The *Polyhedron_3* as an example for the third layer adds the geometric interpretation, provides an easy-to-use interface of high-level functions, and unifies the access to the flexibility provided underneath. It renames face to facet, which is more common for three-dimensional surfaces. The interface is designed to protect the integrity of the internal representation, the handles stored in the items can no longer directly be written by the user. The polyhedron adds the convenient and efficient circulators, see the Support Library Manuals, for accessing the circular sequence of edges around a vertex or around a facet. To achieve this, the *Polyhedron_3* derives new vertices, halfedges and facets from those provided in *Items*. These new items are those actually used in the *HalfedgeDS*, which gives us the coherent type structure in this design, especially if compared to our previous design.

11.3 Example Programs

11.3.1 The Default Halfedge Data Structure

The following example program uses the default halfedge data structure and the decorator class. The default halfedge data structure uses a list-based representation. All incidences of the items and a point type for vertices are defined. The trivial traits class provides the type used for the point. The program creates a loop, consisting of two halfedges, one vertex and two faces, and checks its validity.



```
// file: examples/HalfedgeDS/hds_prog_default.C

#include <CGAL/HalfedgeDS_default.h>
#include <CGAL/HalfedgeDS_decorator.h>

struct Traits { typedef int Point_2; };
typedef CGAL_HALFEDGEDS_DEFAULT<Traits> HDS;
```

```

typedef CGAL::HalfedgeDS_decorator<HDS> Decorator;

int main() {
    HDS hds;
    Decorator decorator(hds);
    decorator.create_loop();
    CGAL_assertion( decorator.is_valid());
    return 0;
}

```

11.3.2 A Minimal Halfedge Data Structure

The following program defines a minimal halfedge data structure using the minimal items class *CGAL::HalfedgeDS_min_items* and a list-based halfedge data structure. The result is a data structure maintaining only halfedges with next and opposite pointers. No vertices or faces are stored. The data structure represents an *undirected graph*.

```

// file: examples/HalfedgeDS/hds_prog_graph.C

#include <CGAL/HalfedgeDS_min_items.h>
#include <CGAL/HalfedgeDS_default.h>
#include <CGAL/HalfedgeDS_decorator.h>

// no traits needed, argument can be arbitrary dummy.
typedef CGAL_HALFEDGE_DS_DEFAULT<int, CGAL::HalfedgeDS_min_items> HDS;
typedef CGAL::HalfedgeDS_decorator<HDS> Decorator;

int main() {
    HDS hds;
    Decorator decorator(hds);
    decorator.create_loop();
    CGAL_assertion( decorator.is_valid());
    return 0;
}

```

11.3.3 The Default with a Vector Instead of a List

The default halfedge data structure uses a list internally and the maximal base classes. We change the list to a vector representation here. Again, a trivial traits class provides the type used for the point. Note that for the vector storage the size of the halfedge data structure should be reserved beforehand, either with the constructor as shown in the example or with the *reserve()* member function. One can later resize the data structure with further calls to the *reserve()* member function, but only if the data structure is in a consistent, i.e., *valid*, state.

Unfortunately this example has also to expose the workaround necessary for compilers that do not support templates as template parameters. The workaround is necessary if the symbolic constant *CGAL_CFG_NO_TMPL_IN_TMPL_PARAM* is set. It uses a member template instead of the class template.

```

// file: examples/HalfedgeDS/hds_prog_vector.C

#include <CGAL/HalfedgeDS_items_2.h>

```

```

#include <CGAL/HalfedgeDS_vector.h>
#include <CGAL/HalfedgeDS_decorator.h>

struct Traits { typedef int Point_2; };
#ifndef CGAL_CFG_NO_TMPL_IN_TMPL_PARAM
typedef CGAL::HalfedgeDS_vector      < Traits, CGAL::HalfedgeDS_items_2> HDS;
#else
typedef CGAL::HalfedgeDS_vector::HDS< Traits, CGAL::HalfedgeDS_items_2> HDS;
#endif
typedef CGAL::HalfedgeDS_decorator<HDS>  Decorator;

int main() {
    HDS hds(1,2,2);
    Decorator decorator(hds);
    decorator.create_loop();
    CGAL_assertion( decorator.is_valid());
    return 0;
}

```

11.3.4 Example Adding Color to Faces

This example re-uses the base class available for faces and adds a member variable *color*.

```

// file: examples/HalfedgeDS/hds_prog_color.C

#include <CGAL/HalfedgeDS_items_2.h>
#include <CGAL/HalfedgeDS_default.h>
#include <CGAL/IO/Color.h>

// A face type with a color member variable.
template <class Refs>
struct My_face : public CGAL::HalfedgeDS_face_base<Refs> {
    CGAL::Color color;
    My_face() {}
    My_face( CGAL::Color c) : color(c) {}
};

// An items type using my face.
struct My_items : public CGAL::HalfedgeDS_items_2 {
    template <class Refs, class Traits>
    struct Face_wrapper {
        typedef My_face<Refs> Face;
    };
};

struct My_traits { // arbitrary point type, not used here.
    typedef int Point_2;
};

typedef CGAL_HALFEDGEDS_DEFAULT <My_traits, My_items> HDS;
typedef HDS::Face Face;
typedef HDS::Face_handle Face_handle;

```

```

int main() {
    HDS hds;
    Face_handle f = hds.faces_push_back( Face( CGAL::RED));
    f->color = CGAL::BLUE;
    CGAL_assertion( f->color == CGAL::BLUE);
    return 0;
}

```

11.3.5 Example Defining a More Compact Halfedge

— *advanced* —

The halfedge data structure as presented here is slightly less space efficient as, for example, the winged-edge data structure [Bau75], the DCEL [MP78] or variants of the quad-edge data structure [GS85]. On the other hand, it does not require any search operations during traversals. A comparison can be found in [Ket99].

The following example trades traversal time for a compact storage representation using traditional C techniques (i.e., type casting and the assumption that pointers, especially those from `malloc` or `new`, point to even addresses). The idea goes as follows: The halfedge data structure allocates halfedges pairwise. Concerning the vector-based data structure this implies that the absolute value of the difference between a halfedge and its opposite halfedge is always one with respect to C pointer arithmetic. We can replace the opposite pointer by a single bit encoding the sign of this difference. We will store this bit as the least significant bit in the next halfedge handle. Furthermore, we do not implement a pointer to the previous halfedge. What remains are three pointers per halfedge.

We use the static member function `halfedge_handle()` to convert from pointers to halfedge handles. The same solution can be applied to the list-based halfedge data structure `CGAL::HalfedgeDS_list`, see `examples/HalfedgeDS/hds_prog_compact2.C`. Here is the example for the vector-based data structure.

```

// file: examples/HalfedgeDS/hds_prog_compact.C

#include <CGAL/HalfedgeDS_items_2.h>
#include <CGAL/HalfedgeDS_vector.h>
#include <CGAL/HalfedgeDS_decorator.h>
#include <cstdint>

// Define a new halfedge class. We assume that the Halfedge_handle can
// be created from a pointer (e.g. the HalfedgeDS is based here on the
// In_place_list or a std::vector with such property) and that halfedges
// are allocated in pairs. We encode the opposite pointer in a single bit,
// which is stored in the lower bit of the next-pointer. We use the
// static member function HDS::halfedge_handle to translate pointer to
// handles.
template <class Refs>
class My_halfedge {
public:
    typedef Refs                                HDS;
    typedef My_halfedge<Refs>                   Base_base;
    typedef My_halfedge<Refs>                   Base;
    typedef My_halfedge<Refs>                   Self;
    typedef CGAL::Tag_false                     Supports_halfedge_prev;

```



```

typedef CGAL::Tag_true Supports_halfedge_vertex;
typedef CGAL::Tag_true Supports_halfedge_face;
typedef typename Refs::Vertex_handle Vertex_handle;
typedef typename Refs::Vertex_const_handle Vertex_const_handle;
typedef typename Refs::Halfedge Halfedge;
typedef typename Refs::Halfedge_handle Halfedge_handle;
typedef typename Refs::Halfedge_const_handle Halfedge_const_handle;
typedef typename Refs::Face_handle Face_handle;
typedef typename Refs::Face_const_handle Face_const_handle;
private:
    std::ptrdiff_t nxt;
public:
    My_halfedge() : nxt(0), f( Face_handle() ) {}

    Halfedge_handle opposite() {
        // Halfedge could be different from My_halfedge (e.g. pointer for
        // linked list). Get proper handle from 'this' pointer first, do
        // pointer arithmetic, then convert pointer back to handle again.
        Halfedge_handle h = HDS::halfedge_handle(this); // proper handle
        if ( nxt & 1)
            return HDS::halfedge_handle( &* h + 1);
        return HDS::halfedge_handle( &* h - 1);
    }
    Halfedge_const_handle opposite() const { // same as above
        Halfedge_const_handle h = HDS::halfedge_handle(this); // proper handle
        if ( nxt & 1)
            return HDS::halfedge_handle( &* h + 1);
        return HDS::halfedge_handle( &* h - 1);
    }
    Halfedge_handle next() {
        return HDS::halfedge_handle((Halfedge*)(nxt & (~ std::ptrdiff_t(1))));
    }
    Halfedge_const_handle next() const {
        return HDS::halfedge_handle((const Halfedge*)
                                     (nxt & (~ std::ptrdiff_t(1))));
    }
    void set_opposite( Halfedge_handle h) {
        CGAL_precondition(( &* h - 1 == &* HDS::halfedge_handle(this)) ||
                          ( &* h + 1 == &* HDS::halfedge_handle(this)));
        if ( &* h - 1 == &* HDS::halfedge_handle(this))
            nxt |= 1;
        else
            nxt &= (~ std::ptrdiff_t(1));
    }
    void set_next( Halfedge_handle h) {
        CGAL_precondition( ((std::ptrdiff_t)(&*h) & 1) == 0);
        nxt = ((std::ptrdiff_t)(&*h)) | (nxt & 1);
    }
private:    // Support for the Vertex_handle.
    Vertex_handle v;
public:
    // the incident vertex.
    Vertex_handle vertex() { return v; }
    Vertex_const_handle vertex() const { return v; }

```

```

        void                set_vertex( Vertex_handle w) { v = w; }

private:
    Face_handle            f;
public:
    Face_handle            face()                { return f; }
    Face_const_handle      face() const          { return f; }
    void                  set_face( Face_handle g)    { f = g; }
    bool                  is_border() const { return f == Face_handle(); }
};

// Replace halfedge in the default items type.
struct My_items : public CGAL::HalfedgeDS_items_2 {
    template <class Refs, class Traits>
    struct Halfedge_wrapper {
        typedef My_halfedge<Refs> Halfedge;
    };
};

struct Traits { typedef int Point_2; };
#ifndef CGAL_CFG_NO_TMPL_IN_TMPL_PARAM
    typedef CGAL::HalfedgeDS_vector    <Traits, My_items> HDS;
#else
    typedef CGAL::HalfedgeDS_vector::HDS<Traits, My_items> HDS;
#endif
typedef CGAL::HalfedgeDS_decorator<HDS>  Decorator;

int main() {
    HDS hds(1,2,2);
    Decorator decorator(hds);
    decorator.create_loop();
    CGAL_assertion( decorator.is_valid());
    return 0;
}

```

└────────── *advanced* ─────────┘

11.3.6 Example Using the Halfedge Iterator

Two edges are created in the default halfedge data structure. The halfedge iterator is used to count the halfedges.

```

// file: examples/HalfedgeDS/hds_prog_halfedge_iterator.C

#include <CGAL/HalfedgeDS_default.h>
#include <CGAL/HalfedgeDS_decorator.h>

struct Traits { typedef int Point_2; };
typedef CGAL_HALFEDGEDS_DEFAULT<Traits> HDS;
typedef CGAL::HalfedgeDS_decorator<HDS> Decorator;
typedef HDS::Halfedge_iterator          Iterator;

int main() {

```

```

HDS hds;
Decorator decorator(hds);
decorator.create_loop();
decorator.create_segment();
CGAL_assertion( decorator.is_valid());
int n = 0;
for ( Iterator i = hds.halfedges_begin(); i != hds.halfedges_end(); ++i )
    ++n;
CGAL_assertion( n == 4); // == 2 edges
return 0;
}

```

11.3.7 Example for an Adapter to Build an Edge Iterator

Three edges are created in the default halfedge data structure. The adapter *N_step_adaptor* is used to declare the edge iterator used in counting the edges.

```

// file: examples/HalfedgeDS/hds_prog_edge_iterator.C

#include <CGAL/HalfedgeDS_default.h>
#include <CGAL/HalfedgeDS_decorator.h>
#include <CGAL/N_step_adaptor.h>

struct Traits { typedef int Point_2; };
typedef CGAL_HALFEDGEDS_DEFAULT<Traits>          HDS;
typedef CGAL::HalfedgeDS_decorator<HDS>          Decorator;
typedef HDS::Halfedge_iterator                    Halfedge_iterator;
typedef CGAL::N_step_adaptor< Halfedge_iterator, 2> Iterator;

int main() {
    HDS hds;
    Decorator decorator(hds);
    decorator.create_loop();
    decorator.create_segment();
    CGAL_assertion( decorator.is_valid());
    int n = 0;
    for ( Iterator e = hds.halfedges_begin(); e != hds.halfedges_end(); ++e )
        ++n;
    CGAL_assertion( n == 2); // == 2 edges
    return 0;
}

```


Halfedge Data Structure

Reference Manual

Lutz Kettner

A halfedge data structure (abbreviated as *HalfedgeDS*, or *HDS* for template parameters) is an edge-centered data structure capable of maintaining incidence informations of vertices, edges and faces, for example for planar maps or polyhedral surfaces. It is a combinatorial data structure, geometric interpretation is added by classes built on top of the halfedge data structure. These classes might be more convenient to use than the halfedge data structure directly, since the halfedge data structure is meant as an implementation layer. See for example the *CGAL::Polyhedron_3* class in Chapter 10.

The data structure provided here is known as the FE-structure [Wei85], as halfedges [Män88, BFH95] or as the doubly connected edge list (DCEL) [dBvKOS97], although the original reference for the DCEL [MP78] describes a related but different data structure. The halfedge data structure can also be seen as one of the variants of the quad-edge data structure [GS85]. In general, the quad-edge data can represent non-orientable 2-manifolds, but the variant here is restricted to orientable 2-manifolds only. An overview and comparison of these different data structures together with a thorough description of the design implemented here can be found in [Ket99].

11.4 Classified Reference Pages

Concepts

<i>HalfedgeDS</i> <Traits,Items,Alloc>	page 847
<i>HalfedgeDSItems</i>	page 859
<i>HalfedgeDSVertex</i>	page 861
<i>HalfedgeDSHalfedge</i>	page 857
<i>HalfedgeDSFace</i>	page 855

Classes

<i>CGAL::HalfedgeDS_default</i> <Traits,HalfedgeDSItems,Alloc>	page 872
<i>CGAL::HalfedgeDS_list</i> <Traits,HalfedgeDSItems,Alloc>	page 886
<i>CGAL::HalfedgeDS_vector</i> <Traits,HalfedgeDSItems,Alloc>	page 888
<i>CGAL::HalfedgeDS_min_items</i>	page 885
<i>CGAL::HalfedgeDS_items_2</i>	page 879

<i>CGAL::HalfedgeDS_vertex_base<Refs></i>	page 889
<i>CGAL::HalfedgeDS_halfedge_base<Refs></i>	page 877
<i>CGAL::HalfedgeDS_face_base<Refs></i>	page 874
<i>CGAL::HalfedgeDS_vertex_min_base<Refs></i>	page 891
<i>CGAL::HalfedgeDS_halfedge_min_base<Refs></i>	page 878
<i>CGAL::HalfedgeDS_face_min_base<Refs></i>	page 876
<i>CGAL::HalfedgeDS_items_decorator<HDS></i>	page 881
<i>CGAL::HalfedgeDS_decorator<HDS></i>	page 865
<i>CGAL::HalfedgeDS_const_decorator<HDS></i>	page 863

11.5 Alphabetical List of Reference Pages

<i>HalfedgeDS<Traits,Items,Alloc></i>	page 847
<i>HalfedgeDSFace</i>	page 855
<i>HalfedgeDSHalfedge</i>	page 857
<i>HalfedgeDSItems</i>	page 859
<i>HalfedgeDSVertex</i>	page 861
<i>HalfedgeDS_const_decorator<HDS></i>	page 863
<i>HalfedgeDS_decorator<HDS></i>	page 865
<i>HalfedgeDS_default<Traits,HalfedgeDSItems,Alloc></i>	page 872
<i>HalfedgeDS_face_base<Refs></i>	page 874
<i>HalfedgeDS_face_min_base<Refs></i>	page 876
<i>HalfedgeDS_halfedge_base<Refs></i>	page 877
<i>HalfedgeDS_halfedge_min_base<Refs></i>	page 878
<i>HalfedgeDS_items_2</i>	page 879
<i>HalfedgeDS_items_decorator<HDS></i>	page 881
<i>HalfedgeDS_list<Traits,HalfedgeDSItems,Alloc></i>	page 886
<i>HalfedgeDS_min_items</i>	page 885
<i>HalfedgeDS_vector<Traits,HalfedgeDSItems,Alloc></i>	page 888
<i>HalfedgeDS_vertex_base<Refs></i>	page 889
<i>HalfedgeDS_vertex_min_base<Refs></i>	page 891

HalfedgeDS<Traits,Items,Alloc>

Release Note

Beginning with CGAL R2.3, this package has a new design. The old design is still available for backwards compatibility and to support older compiler, such as MSVC++6.0. However its use is deprecated and the manual pages are not converted into this new manual format. Instead, see its old documentation in the manual of deprecated packages. The two designs are not interchangeable.

Definition

The concept of a halfedge data structure (abbreviated as *HalfedgeDS*, or *HDS* for template parameters) defines an edge-centered data structure capable of maintaining incidence informations of vertices, edges, and faces, for example for planar maps or polyhedral surfaces. It is a combinatorial data structure, geometric interpretation is added by classes built on top of the halfedge data structure.

The data structure defined here is known as the FE-structure [Wei85], as halfedges [Män88, BFH95] or as the doubly connected edge list (DCEL) [dBvKOS97], although the original reference for the DCEL [MP78] describes a different data structure. The halfedge data structure can also be seen as one of the variants of the quad-edge data structure [GS85]. In general, the quad-edge data can represent non-orientable 2-manifolds, but the variant here is restricted to orientable 2-manifolds only. An overview and comparison of these different data structures together with a thorough description of the design implemented here can be found in [Ket99].

Each edge is represented by two halfedges with opposite orientations. Each halfedge can store a reference to an incident face and an incident vertex. For each face and each vertex an incident halfedge is stored. Reduced variants of the halfedge data structure can omit some of these incidences, for example the reference to halfedges in vertices or the storage of vertices at all. See Figure 11.2 for the incidences, the mandatory and optional member functions possible for vertices, halfedges, and faces.

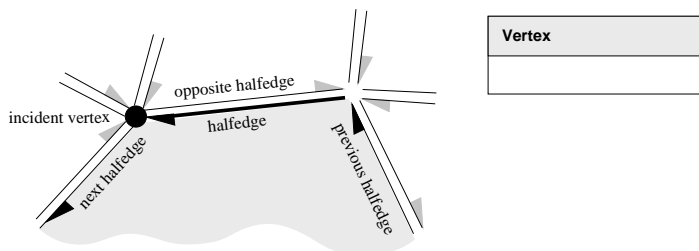


Figure 11.2: The three classes *Vertex*, *Halfedge*, and *Face* of the halfedge data structure. Member functions with shaded background are mandatory. The others are optionally supported.

A `HalfedgeDS<Traits,Items,Alloc>` organizes the internal storage of its items. Examples are a list-based or a vector-based storage. The `HalfedgeDS<Traits,Items,Alloc>` exhibits most of the characteristics of the container class used internally, for example the iterator category. A vector resizes automatically when a new item exceeds the reserved space. Since resizing is an expensive operation for a `HalfedgeDS<Traits,Items,Alloc>` in general and only possible in a well defined state of the data structure (no dangling handles), it must be called explicitly in advance for a `HalfedgeDS<Traits,Items,Alloc>` before inserting new items beyond the current capacity. Classes built on top of a `HalfedgeDS<Traits,Items,Alloc>` are advised to call the `reserve()` member function before creating new items.

Parameters

A `HalfedgeDS<Traits,Items,Alloc>` is a class template and will be used as argument for other class templates, for example `CGAL::Polyhedron_3`. The template parameters to instantiate the `HalfedgeDS<Traits,Items,Alloc>` will be provided by this other class template. Therefore, the three template parameters and their meaning are mandatory. We distinguish between the template `HalfedgeDS<Traits,Items,Alloc>` and an instantiation of it.

Traits is a traits class that will be passed to the item types in *Items*. It will not be used in `HalfedgeDS<Traits,Items,Alloc>` itself. *Items* is a model of the *HalfedgeDSItems* concept. *Alloc* is a standard allocator that fulfills all requirements of allocators for STL container classes. The *rebind* mechanism from *Alloc* will be used to create appropriate allocators internally. A default argument is mandatory for *Alloc*, for example, the macro `CGAL_ALLOCATOR(int)` from the `<CGAL/memory.h>` header file can be used as default allocator.

Types

<code>HalfedgeDS<Traits,Items,Alloc>:: Traits</code>	traits class.
<code>HalfedgeDS<Traits,Items,Alloc>:: Items</code>	model of <i>HalfedgeDSItems</i> concept.

<code>HalfedgeDS<Traits,Items,Alloc>:: size_type</code>	size type.
<code>HalfedgeDS<Traits,Items,Alloc>:: difference_type</code>	difference type.
<code>HalfedgeDS<Traits,Items,Alloc>:: iterator_category</code>	iterator category for all iterators.
<code>HalfedgeDS<Traits,Items,Alloc>:: allocator_type</code>	allocator type <i>Alloc</i> .

<code>HalfedgeDS<Traits,Items,Alloc>:: Vertex</code>	model of <i>HalfedgeDSVertex</i> concept.
<code>HalfedgeDS<Traits,Items,Alloc>:: Halfedge</code>	model of <i>HalfedgeDSHalfedge</i> concept.
<code>HalfedgeDS<Traits,Items,Alloc>:: Face</code>	model of <i>HalfedgeDSFace</i> concept.

The following handles and iterators have appropriate non-mutable counterparts, i.e., *const_handle* and *const_iterator*. The mutable types are assignable to their non-mutable counterparts. The iterators are assignable to the respective handle types. Wherever the handles appear in function parameter lists, the corresponding iterators can be used as well. **Note:** The handle types must have a default constructor that creates a unique and always the same handle value. It will be used in analogy to `NULL` for pointers.

<code>HalfedgeDS<Traits,Items,Alloc>:: Vertex_handle</code>	handle to vertex.
<code>HalfedgeDS<Traits,Items,Alloc>:: Halfedge_handle</code>	handle to halfedge.
<code>HalfedgeDS<Traits,Items,Alloc>:: Face_handle</code>	handle to face.

<code>HalfedgeDS<Traits,Items,Alloc>:: Vertex_iterator</code>	iterator over all vertices.
<code>HalfedgeDS<Traits,Items,Alloc>:: Halfedge_iterator</code>	iterator over all halfedges.
<code>HalfedgeDS<Traits,Items,Alloc>:: Face_iterator</code>	iterator over all faces.

— advanced —

Types for Tagging Optional Features

The following types are equal to either `CGAL::Tag_true` or `CGAL::Tag_false`, depending on whether the named feature is supported or not.

<i>HalfedgeDS</i> < <i>Traits</i> , <i>Items</i> , <i>Alloc</i> >:: <i>Supports_vertex_halfedge</i>	<i>Vertex</i> :: <i>halfedge</i> ()
<i>HalfedgeDS</i> < <i>Traits</i> , <i>Items</i> , <i>Alloc</i> >:: <i>Supports_halfedge_prev</i>	<i>Halfedge</i> :: <i>prev</i> ()
<i>HalfedgeDS</i> < <i>Traits</i> , <i>Items</i> , <i>Alloc</i> >:: <i>Supports_halfedge_vertex</i>	<i>Halfedge</i> :: <i>vertex</i> ()
<i>HalfedgeDS</i> < <i>Traits</i> , <i>Items</i> , <i>Alloc</i> >:: <i>Supports_halfedge_face</i>	<i>Halfedge</i> :: <i>face</i> ()
<i>HalfedgeDS</i> < <i>Traits</i> , <i>Items</i> , <i>Alloc</i> >:: <i>Supports_face_halfedge</i>	<i>Face</i> :: <i>halfedge</i> ()
<i>HalfedgeDS</i> < <i>Traits</i> , <i>Items</i> , <i>Alloc</i> >:: <i>Supports_removal</i>	removal of individual elements.

The following dependencies among these options must be regarded:

Vertices are supported \iff *Supports_halfedge_vertex* \equiv *CGAL::Tag_true*.
 Faces are supported \iff *Supports_halfedge_face* \equiv *CGAL::Tag_true*.
Supports_halfedge \equiv *CGAL::Tag_true* \implies *Supports_halfedge_vertex* \equiv *CGAL::Tag_true*.
Supports_vertex_point \equiv *CGAL::Tag_true* \implies *Supports_halfedge_vertex* \equiv *CGAL::Tag_true*.
Supports_face_halfedge \equiv *CGAL::Tag_true* \implies *Supports_halfedge_face* \equiv *CGAL::Tag_true*.

— advanced —

— advanced —

Static Member Functions

When writing an items type, such as a user defined vertex, certain functions need to create a handle but knowing only a pointer, for example, the *this*-pointer. The following static member functions of *HalfedgeDS*<*Traits*,*Items*,*Alloc*> create such a corresponding handle for an item type from a pointer. This conversion encapsulates possible adjustments for hidden data members in the true item type, such as linked-list pointers. Note that the user provides item types with the *Items* template argument, which may differ from the *Vertex*, *Halfedge*, and *Face* types defined in *HalfedgeDS*<*Traits*,*Items*,*Alloc*>. If they differ, they are derived from the user provided item types. We denote the user item types with *Vertex_base*, *Halfedge_base*, and *Face_base* in the following. The fully qualified name for *Vertex_base* would be for example – assuming that the type *Self* refers to the instantiated *HalfedgeDS* –

```
typedef typename Items::template Vertex_wrapper<Self,Traits> Vertex_wrapper;
typedef typename Vertex_wrapper::Vertex Vertex_base;
```

Implementing these functions relies on the fundamental assumption that an iterator (or handle) of the internally used container class can be constructed from a pointer of a contained item only. This is true and controlled by us for *CGAL::In_place_list*. It is true for the *std::vector* of major STL distributions, but not necessarily guaranteed. We might switch to an internal implementation if need arises.

<i>static Vertex_handle</i>	<i>HalfedgeDS</i> :: <i>vertex_handle</i> (<i>Vertex_base</i> * <i>v</i>)
<i>static Vertex_const_handle</i>	<i>HalfedgeDS</i> :: <i>vertex_handle</i> (<i>const Vertex_base</i> * <i>v</i>)
<i>static Halfedge_handle</i>	<i>HalfedgeDS</i> :: <i>halfedge_handle</i> (<i>Halfedge_base</i> * <i>h</i>)
<i>static Halfedge_const_handle</i>	<i>HalfedgeDS</i> :: <i>halfedge_handle</i> (<i>const Halfedge_base</i> * <i>h</i>)
<i>static Face_handle</i>	<i>HalfedgeDS</i> :: <i>face_handle</i> (<i>Face_base</i> * <i>f</i>)
<i>static Face_const_handle</i>	<i>HalfedgeDS</i> :: <i>face_handle</i> (<i>const Face_items</i> * <i>f</i>)

— advanced —

Creation

<i>HalfedgeDS</i> < <i>Traits</i> , <i>Items</i> , <i>Alloc</i> > <i>hds</i> ;	empty halfedge data structure.
<i>HalfedgeDS</i> < <i>Traits</i> , <i>Items</i> , <i>Alloc</i> > <i>hds</i> (<i>size_type</i> <i>v</i> , <i>size_type</i> <i>h</i> , <i>size_type</i> <i>f</i>);	storage reserved for <i>v</i> vertices, <i>h</i> halfedges, and <i>f</i> faces.
<i>HalfedgeDS</i> < <i>Traits</i> , <i>Items</i> , <i>Alloc</i> > <i>hds</i> (<i>hds2</i>);	copy constructor. <i>Precondition</i> : <i>hds2</i> contains no dangling handles.
<i>HalfedgeDS</i> < <i>Traits</i> , <i>Items</i> , <i>Alloc</i> >& <i>hds</i> = <i>hds2</i>	assignment operator. <i>Precondition</i> : <i>hds2</i> contains no dangling handles.
<i>void</i> <i>hds.reserve</i> (<i>size_type</i> <i>v</i> , <i>size_type</i> <i>h</i> , <i>size_type</i> <i>f</i>)	reserves storage for <i>v</i> vertices, <i>h</i> halfedges, and <i>f</i> faces. If all capacities are already greater or equal than the requested sizes nothing happens. Otherwise, <i>hds</i> will be resized and all handles, iterators and circulators invalidate. <i>Precondition</i> : If resizing is necessary <i>hds</i> contains no dangling handles.

Access Member Functions

<i>Size</i>	<i>hds.size_of_vertices</i> ()	number of vertices.
<i>Size</i>	<i>hds.size_of_halfedges</i> ()	number of halfedges.
<i>Size</i>	<i>hds.size_of_faces</i> ()	number of faces.
<i>Size</i>	<i>hds.capacity_of_vertices</i> ()	space reserved for vertices.
<i>Size</i>	<i>hds.capacity_of_halfedges</i> ()	space reserved for halfedges.
<i>Size</i>	<i>hds.capacity_of_faces</i> ()	space reserved for faces.
<i>size_t</i>	<i>hds.bytes</i> ()	bytes used for <i>hds</i> .
<i>size_t</i>	<i>hds.bytes_reserved</i> ()	bytes reserved for <i>hds</i> .
<i>allocator_type</i>	<i>hds.get_allocator</i> ()	allocator object.

The following member functions return the non-mutable iterator if *hds* is declared const.

<i>Vertex_iterator</i>	<i>hds.vertices_begin</i> ()	iterator over all vertices.
<i>Vertex_iterator</i>	<i>hds.vertices_end</i> ()	
<i>Halfedge_iterator</i>	<i>hds.halfedges_begin</i> ()	iterator over all halfedges
<i>Halfedge_iterator</i>	<i>hds.halfedges_end</i> ()	
<i>Face_iterator</i>	<i>hds.faces_begin</i> ()	iterator over all faces.
<i>Face_iterator</i>	<i>hds.faces_end</i> ()	

Insertion

Note that the vertex-related and the face-related member functions may not be provided for a *HalfedgeDS*<*Traits*,*Items*,*Alloc*> that does not support vertices or faces respectively.

Vertex_handle *hds.vertices_push_back(const Vertex& v)*

appends a copy of *v* to *hds*. Returns a handle of the new vertex.

Halfedge_handle *hds.edges_push_back(const Halfedge& h, const Halfedge& g)*

appends a copy of *h* and a copy of *g* to *hds* and makes them opposite to each other. Returns a handle of the copy of *h*.

Halfedge_handle *hds.edges_push_back(const Halfedge& h)*

appends a copy of *h* and a copy of *h*→*opposite()* to *hds* and makes them opposite to each other. Returns a handle of the copy of *h*.
Precondition: h→*opposite()* denotes a halfedge.

Face_handle *hds.faces_push_back(const Face& f)*

appends a copy of *f* to *hds*. Returns a handle of the new face.

Removal

Erasing single elements is optional and indicated with the type tag *Supports_removal*. The *pop_back* and the *clear* member functions are mandatory. If vertices or faces are not supported for a *HalfedgeDS*< Traits, Items, Alloc > the *pop_back* and the *clear* member functions must be provided as null operations.

void *hds.vertices_pop_front()* removes the first vertex if vertices are supported and *Supports_removal* ≡ *CGAL::Tag_true*.

void *hds.vertices_pop_back()* removes the last vertex.

void *hds.vertices_erase(Vertex_handle v)* removes the vertex *v* if vertices are supported and *Supports_removal* ≡ *CGAL::Tag_true*.

void *hds.vertices_erase(Vertex_handle first, Vertex_handle last)*

removes the range of vertices [*first*, *last*) if vertices are supported and *Supports_removal* ≡ *CGAL::Tag_true*.

void *hds.edges_pop_front()* removes the first two halfedges if *Supports_removal* ≡ *CGAL::Tag_true*.

void *hds.edges_pop_back()* removes the last two halfedges.

void *hds.edges_erase(Halfedge_handle h)* removes the pair of halfedges *h* and *h*→*opposite()* if *Supports_removal* ≡ *CGAL::Tag_true*.

<code>void</code>	<code>hds.edges_erase(Halfedge_handle first, Halfedge_handle last)</code>	removes the range of edges $[first, last)$ if <i>Supports_removal</i> \equiv <i>CGAL::Tag_true</i> .
<code>void</code>	<code>hds.faces_pop_front()</code>	removes the first face if faces are supported and <i>Supports_removal</i> \equiv <i>CGAL::Tag_true</i> .
<code>void</code>	<code>hds.faces_pop_back()</code>	removes the last face.
<code>void</code>	<code>hds.faces_erase(Face_handle f)</code>	removes the face <i>f</i> if faces are supported and <i>Supports_removal</i> \equiv <i>CGAL::Tag_true</i> .
<code>void</code>	<code>hds.faces_erase(Face_handle first, Face_handle last)</code>	removes the range of faces $[first, last)$ if faces are supported and <i>Supports_removal</i> \equiv <i>CGAL::Tag_true</i> .
<code>void</code>	<code>hds.vertices_clear()</code>	removes all vertices.
<code>void</code>	<code>hds.edges_clear()</code>	removes all halfedges.
<code>void</code>	<code>hds.faces_clear()</code>	removes all faces.
<code>void</code>	<code>hds.clear()</code>	removes all elements.

— advanced —

Operations with Border Halfedges

The following notion of *border halfedges* is particular useful where the halfedge data structure is used to model surfaces with boundary, i.e., surfaces with missing faces or open regions. Halfedges incident to an open region are called *border halfedges*. A halfedge is a *border edge* if the halfedge itself or its opposite halfedge is a border halfedge. The only requirement to work with border halfedges is that the *Halfedge* class provides a member function *is_border()* returning a *bool*. Usually, the halfedge data structure supports faces and the value of the default constructor of the face handle will indicate a border halfedge, but this may not be the only possibility. The *is_border()* predicate divides the edges into two classes, the border edges and the non-border edges. The following normalization reorganizes the sequential storage of the edges such that the non-border edges precede the border edges, and that for each border edge the latter of the two halfedges is a border halfedge (the first one might be a border halfedge too). The normalization stores the number of border halfedges, as well as the halfedge iterator where the border edges start at, within the halfedge data structure. These values will be invalid after further halfedge insertions or removals and changes in the border status of a halfedge. There is no automatic update required.

<code>void</code>	<code>hds.normalize_border()</code>	sorts halfedges such that the non-border edges precede the border edges. For each border edge that is incident to a face, the halfedge iterator will reference the halfedge incident to the face right before the halfedge incident to the open region.
-------------------	-------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Size *hds.size_of_border_halfedges() const*

number of border halfedges. An edge with no incident face counts as two border halfedges.

Precondition: *normalize_border()* has been called and no halfedge insertion or removal and no change in border status of the halfedges have occurred since then.

Size *hds.size_of_border_edges() const*

number of border edges. If *size_of_border_edges()* is equal to *size_of_border_halfedges()* all border edges are incident to a face on one side and to an open region on the other side.

Precondition: *normalize_border()* has been called and no halfedge insertion or removal and no change in border status of the halfedges have occurred since then.

Halfedge_iterator *hds.border_halfedges_begin()*

halfedge iterator starting with the border edges. The range [*halfedges_begin()*, *border_halfedges_begin()*) denotes all non-border edges. The range [*border_halfedges_begin()*, *halfedges_end()*) denotes all border edges.

Precondition: *normalize_border()* has been called and no halfedge insertion or removal and no change in border status of the halfedges have occurred since then.

————— advanced —————

Has Models

CGAL::HalfedgeDS_default page 872
CGAL::HalfedgeDS_list page 886
CGAL::HalfedgeDS_vector page 888

See Also

HalfedgeDSItems page 859
CGAL::Polyhedron_3<Traits> page 799
CGAL::HalfedgeDS_vertex_base<Refs> page 889
CGAL::HalfedgeDS_halfedge_base<Refs> page 877
CGAL::HalfedgeDS_face_base<Refs> page 874
CGAL::HalfedgeDS_items_decorator<HDS> page 881
CGAL::HalfedgeDS_decorator<HDS> page 865
CGAL::HalfedgeDS_const_decorator<HDS> page 863

Implementation

Classes parameterized with a halfedge data structure, such as *CGAL::Polyhedron_3*, need to declare a class template as one of its template parameters for the *HalfedgeDS<Traits,Items,Alloc>*. For compilers not supporting this (i.e. the flag *CGAL_CFG_NO_TMPL_IN_TMPL_PARAM* is set), the following workaround is required,

which defines a `HalfedgeDS<Traits,Items,Alloc>` as a normal class that contains a member class template named *HDS*, which is the actual halfedge data structure as defined here. The following program fragment illustrates this workaround:

```
#ifndef CGAL_CFG_NO_TMPL_IN_TMPL_PARAM
    template <class Traits, class Items, class Alloc>
    class HalfedgeDS {
    public:
        typedef HalfedgeDS<Traits,Items,Alloc> Self;
        HalfedgeDS_vector(); // constructors
#else
    struct HalfedgeDS {
        template <class Traits, class Items, class Alloc>
        class HDS {
        public:
            typedef HDS<Traits,Items,Alloc> Self;
            HDS(); // constructors
#endif
        // ... further member functions. Self denotes the HalfedgeDS.
    };
#endif
#ifdef CGAL_CFG_NO_TMPL_IN_TMPL_PARAM
    };
#endif
```

HalfedgeDSFace

Definition

The concept `HalfedgeDSFace` defines the requirements for the local *Face* type in the *HalfedgeDS* concept. It is also required in the *Face_wrapper*<*Refs*,*Traits*> member class template of an items class, see the *HalfedgeDSItems* concept.

A face optionally stores a reference to an incident halfedge that points to the face. A type tag indicates whether the related member functions are supported. Figure 11.3 on page 857 depicts the relationship between a halfedge and its incident halfedges, vertices, and faces.

For the protection of the integrity of the data structure classes such as *CGAL::Polyhedron_3* are allowed to redefine the modifying member functions to be private. In order to make them accessible for the halfedge data structure they must be derived from a base class *Base* where the modifying member functions are still public. (The protection can be bypassed by the user, but not by accident.)

Types

<i>HalfedgeDSFace::HalfedgeDS</i>	instantiated <i>HalfedgeDS</i> (\equiv <i>Refs</i>).
<i>HalfedgeDSFace::Base</i>	base class that allows modifications.
<i>HalfedgeDSFace::Vertex</i>	model of <i>HalfedgeDSVertex</i> .
<i>HalfedgeDSFace::Halfedge</i>	model of <i>HalfedgeDSHalfedge</i> .
<i>HalfedgeDSFace::Vertex_handle</i>	handle to vertex.
<i>HalfedgeDSFace::Halfedge_handle</i>	handle to halfedge.
<i>HalfedgeDSFace::Face_handle</i>	handle to face.
<i>HalfedgeDSFace::Vertex_const_handle</i>	
<i>HalfedgeDSFace::Halfedge_const_handle</i>	
<i>HalfedgeDSFace::Face_const_handle</i>	
<i>HalfedgeDSFace::Supports_face_halfedge</i>	<i>CGAL::Tag_true</i> or <i>CGAL::Tag_false</i> .

Creation

<i>HalfedgeDSFace f</i> ;	default constructor.
---------------------------	----------------------

Operations available if *Supports_face_halfedge* \equiv *CGAL::Tag_true*

<i>Halfedge_handle</i>	<i>f.halfedge()</i>	
<i>Halfedge_const_handle</i>	<i>f.halfedge() const</i>	incident halfedge that points to <i>f</i> .
<i>void</i>	<i>f.set_halfedge(Halfedge_handle h)</i>	sets incident halfedge to <i>h</i> .

Has Models

<i>CGAL::HalfedgeDS_face_base</i> < <i>Refs</i> >	page 874
<i>CGAL::HalfedgeDS_face_min_base</i> < <i>Refs</i> >	page 876

See Also

HalfedgeDS<Traits,Items,Alloc>	page 847
HalfedgeDSItems	page 859
HalfedgeDSVertex	page 861
HalfedgeDSHalfedge	page 857

HalfedgeDSHalfedge

Definition

The concept `HalfedgeDSHalfedge` defines the requirements for the local *Halfedge* type in the *HalfedgeDS* concept. It is also required in the *Halfedge_wrapper*<*Refs*,*Traits*> member class template of an items class, see the *HalfedgeDSItems* concept.

A halfedge is an oriented edge between two vertices. It is always paired with a halfedge pointing in the opposite direction. The *opposite()* member function returns this halfedge of opposite orientation. The *next()* member function points to the successor halfedge along the face – or if the halfedge is a border halfedge – along the open region. A halfedge optionally stores a reference to the previous halfedge along the face, a reference to an incident vertex, and a reference to an incident face. Type tags indicate whether the related member functions are supported. Figure 11.3 depicts the relationship between a halfedge and its incident halfedges, vertices, and faces.

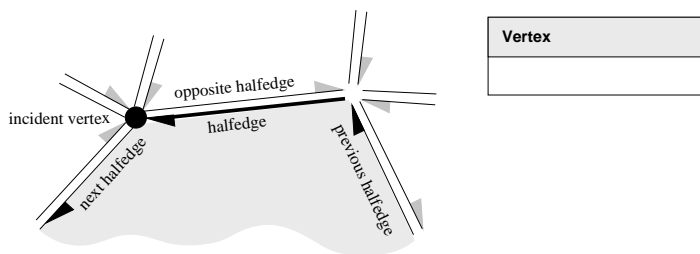


Figure 11.3: The three classes *Vertex*, *Halfedge*, and *Face* of the halfedge data structure. Member functions with shaded background are mandatory. The others are optionally supported.

For the protection of the integrity of the data structure classes such as `CGAL::Polyhedron_3` are allowed to redefine the modifying member functions to be private. In order to make them accessible for the halfedge data structure they must be derived from a base class *Base* where the modifying member functions are still public. Even more protection is provided for the *set_opposite()* member function. The base class *Base_base* provides access to it. (The protection could be bypassed also by an user, but not by accident.)

Types

<code>HalfedgeDSHalfedge:: HalfedgeDS</code>	instantiated <i>HalfedgeDS</i> (\equiv <i>Refs</i>).
<code>HalfedgeDSHalfedge:: Base</code>	base class that allows modifications.
<code>HalfedgeDSHalfedge:: Base_base</code>	base class to access <i>set_opposite()</i> .
<code>HalfedgeDSHalfedge:: Vertex</code>	model of <i>HalfedgeDSVertex</i> .
<code>HalfedgeDSHalfedge:: Face</code>	model of <i>HalfedgeDSFace</i> .
<code>HalfedgeDSHalfedge:: Vertex_handle</code>	handle to vertex.
<code>HalfedgeDSHalfedge:: Halfedge_handle</code>	handle to halfedge.
<code>HalfedgeDSHalfedge:: Face_handle</code>	handle to face.
<code>HalfedgeDSHalfedge:: Vertex_const_handle</code>	
<code>HalfedgeDSHalfedge:: Halfedge_const_handle</code>	
<code>HalfedgeDSHalfedge:: Face_const_handle</code>	
<code>HalfedgeDSHalfedge:: Supports_halfedge_prev</code>	<code>CGAL::Tag_true</code> or <code>CGAL::Tag_false</code> .

<i>HalfedgeDSHalfedge:: Supports_halfedge_vertex</i>	~
<i>HalfedgeDSHalfedge:: Supports_halfedge_face</i>	~

Creation

<i>HalfedgeDSHalfedge h;</i>	default constructor.
------------------------------	----------------------

Operations

<i>Halfedge_handle</i>	<i>h.opposite()</i>	
<i>Halfedge_const_handle</i>	<i>h.opposite() const</i>	the opposite halfedge.
<i>void</i>	<i>h.set_opposite(Halfedge_handle h)</i>	sets opposite halfedge to <i>h</i> .
<i>Halfedge_handle</i>	<i>h.next()</i>	
<i>Halfedge_const_handle</i>	<i>h.next() const</i>	the next halfedge around the face.
<i>void</i>	<i>h.set_next(Halfedge_handle h)</i>	sets next halfedge to <i>h</i> .
<i>bool</i>	<i>h.is_border() const</i>	is true if <i>h</i> is a border halfedge.

Operations available if *Supports_halfedge_prev* \equiv *CGAL::Tag_true*

<i>Halfedge_handle</i>	<i>h.prev()</i>	
<i>Halfedge_const_handle</i>	<i>h.prev() const</i>	the previous halfedge around the face.
<i>void</i>	<i>h.set_prev(Halfedge_handle h)</i>	sets prev halfedge to <i>h</i> .

Operations available if *Supports_halfedge_vertex* \equiv *CGAL::Tag_true*

<i>Vertex_handle</i>	<i>h.vertex()</i>	
<i>Vertex_const_handle</i>	<i>h.vertex() const</i>	the incident vertex of <i>h</i> .
<i>void</i>	<i>h.set_vertex(Vertex_handle v)</i>	sets incident vertex to <i>v</i> .

Operations available if *Supports_halfedge_face* \equiv *CGAL::Tag_true*

<i>Face_handle</i>	<i>h.face()</i>	
<i>Face_const_handle</i>	<i>h.face() const</i>	the incident face of <i>h</i> . If <i>h</i> is a border halfedge the result is default construction of the handle.
<i>void</i>	<i>h.set_face(Face_handle f)</i>	sets incident face to <i>f</i> .

Has Models

<i>CGAL::HalfedgeDS_halfedge_base<Refs></i>	page 877
<i>CGAL::HalfedgeDS_halfedge_min_base<Refs></i>	page 878

See Also

<i>HalfedgeDS<Traits,Items,Alloc></i>	page 847
<i>HalfedgeDSItems</i>	page 859
<i>HalfedgeDSVertex</i>	page 861
<i>HalfedgeDSFace</i>	page 855

HalfedgeDSItems

Definition

The concept `HalfedgeDSItems` wraps the three item types – vertex, halfedge, and face – for a halfedge data structure. A `HalfedgeDSItems` contains three member class templates named `Vertex_wrapper`, `Halfedge_wrapper`, and `Face_wrapper`, each with two template parameters, `Refs` and `Traits`. `Refs` requires an instantiated halfedge data structure `HalfedgeDS` as argument, `Traits` is a geometric traits class supplied by the class that uses the halfedge data structure as internal representation. `Traits` is not used by the halfedge data structure itself. These three member class templates provide a local type named `Vertex`, `Halfedge`, and `Face` respectively. The requirements on these types are described on page 861, page 857, and page 855 respectively.

Types

<code>HalfedgeDSItems::Vertex_wrapper<Refs,Traits>::Vertex</code>	model of <code>HalfedgeDSVertex</code> .
<code>HalfedgeDSItems::Halfedge_wrapper<Refs,Traits>::Halfedge</code>	model of <code>HalfedgeDSHalfedge</code> .
<code>HalfedgeDSItems::Face_wrapper<Refs,Traits>::Face</code>	model of <code>HalfedgeDSFace</code> .

Has Models

<code>CGAL::HalfedgeDS_min_items</code>	page 885
<code>CGAL::HalfedgeDS_items_2</code>	page 879
<code>CGAL::Polyhedron_items_3</code>	page 824

See Also

<code>HalfedgeDS<Traits,Items,Alloc></code>	page 847
<code>HalfedgeDSVertex</code>	page 861
<code>HalfedgeDSHalfedge</code>	page 857
<code>HalfedgeDSFace</code>	page 855
<code>PolyhedronItems_3</code>	page 822
<code>CGAL::HalfedgeDS_vertex_base<Refs></code>	page 889
<code>CGAL::HalfedgeDS_halfedge_base<Refs></code>	page 877
<code>CGAL::HalfedgeDS_face_base<Refs></code>	page 874

Example

The following example shows the canonical implementation of the `CGAL::HalfedgeDS_min_items` class. It uses the base classes for the item types that are provided in the library.

```
struct HalfedgeDS_min_items {
    template < class Refs, class Traits>
    struct Vertex_wrapper {
        typedef CGAL::HalfedgeDS_vertex_min_base< Refs>    Vertex;
    };
    template < class Refs, class Traits>
    struct Halfedge_wrapper {
```

```

        typedef CGAL::HalfedgeDS_halfedge_min_base< Refs> Halfedge;
    };
    template < class Refs, class Traits>
    struct Face_wrapper {
        typedef CGAL::HalfedgeDS_face_min_base< Refs>      Face;
    };
};

```

See page [879](#) for an example implementation of the *CGAL::HalfedgeDS_items_2* class.

See Also

HalfedgeDS<Traits,Items,Alloc>	page 847
HalfedgeDSItems	page 859
HalfedgeDSHalfedge	page 857
HalfedgeDSFace	page 855

CGAL::HalfedgeDS_const_decorator<HDS>

Definition

The classes *CGAL::HalfedgeDS_items_decorator<HDS>*, *CGAL::HalfedgeDS_decorator<HDS>*, and *CGAL::HalfedgeDS_const_decorator<HDS>* provide additional functions to examine and to modify a halfedge data structure *HDS*. The class *CGAL::HalfedgeDS_items_decorator<HDS>* provides additional functions for vertices, halfedges, and faces of a halfedge data structure without knowing the containing halfedge data structure. The class *CGAL::HalfedgeDS_decorator<HDS>* stores a reference to the halfedge data structure and provides functions that modify the halfedge data structure, for example Euler-operators. The class *CGAL::HalfedgeDS_const_decorator<HDS>* stores a const reference to the halfedge data structure. It contains non-modifying functions, for example the test for validness of the data structure.

All these additional functions take care of the different capabilities a halfedge data structure may have or may not have. The functions evaluate the type tags of the halfedge data structure to decide on the actions. If a particular feature is not supported nothing is done. Note that for example the creation of new halfedges is mandatory for all halfedge data structures and will not appear here again.

```
#include <CGAL/HalfedgeDS_const_decorator.h>
```

Inherits From

CGAL::HalfedgeDS_items_decorator<HDS> page 881

Creation

```
HalfedgeDS_const_decorator<HDS> D( const HDS& hds);
```

keeps internally a const reference to *hds*.

Validness Checks

A halfedge data structure has no definition of validness of its own, but a useful set of tests is defined with the following levels:

Level 0 The number of halfedges is even. All pointers except the vertex pointer and the face pointer for border halfedges are unequal to their respective default construction value. For all halfedges *h*: The opposite halfedge is different from *h* and the opposite of the opposite is equal to *h*. The next of the previous halfedge is equal to *h*. For all vertices *v*: the incident vertex of the incident halfedge of *v* is equal to *v*. The halfedges around *v* starting with the incident halfedge of *v* form a cycle. For all faces *f*: the incident face of the incident halfedge of *f* is equal to *f*. The halfedges around *f* starting with the incident halfedge of *f* form a cycle. Redundancies among internal variables are tested, e.g., that iterators enumerate as many items as the related size value indicates.

Level 1 All tests of level 0. For all halfedges *h*: The incident vertex of *h* exists and is equal to the incident vertex of the opposite of the next halfedge. The incident face (or hole) of *h* is equal to the incident face (or hole) of the next halfedge.

Level 2 All tests of level 1. The sum of all halfedges that can be reached through the vertices must be equal to the number of all halfedges, i.e., all halfedges incident to a vertex must form a single cycle.

Level 3 All tests of level 2. The sum of all halfedges that can be reached through the faces must be equal to the number of all halfedges, i.e., all halfedges surrounding a face must form a single cycle (no holes in faces).

Level 4 All tests of level 3 and *normalized_border_is_valid*.

bool

D.is_valid(bool verbose = false, int level = 0)

returns *true* if the halfedge data structure *hds* is valid with respect to the *level* value as defined above. If *verbose* is *true*, statistics are written to *cerr*.

bool

D.normalized_border_is_valid(bool verbose = false)

returns *true* if the border halfedges are in normalized representation, which is when enumerating all halfedges with the halfedge iterator the following holds: The non-border edges precede the border edges. For border edges, the second halfedge is a border halfedge. (The first halfedge may or may not be a border halfedge.) The halfedge iterator *border_halfedges_begin()* denotes the first border edge. If *verbose* is *true*, statistics are written to *cerr*.

See Also

CGAL::HalfedgeDS_items_decorator<HDS> [page 881](#)
CGAL::HalfedgeDS_decorator<HDS> [page 865](#)

Example

The following program fragment illustrates the implementation of a *is_valid()* member function for a simplified polyhedron class. We assume here that the level three check is the appropriate default for polyhedral surfaces.

```
namespace CGAL {
    template <class Traits>
    class Polyhedron {
        typedef HalfedgeDS_default<Traits> HDS;
        HDS hds;
    public:
        // ...
        bool is_valid( bool verb = false, int level = 0) const {
            Verbose_ostream verr(verb);
            verr << "begin Polyhedron::is_valid( verb=true, level = " << level
                << "):" << std::endl;
            HalfedgeDS_const_decorator<HDS> decorator(hds);
            bool valid = decorator.is_valid( verb, level + 3);
            // further checks ...
        }
    };
}
```


CGAL::HalfedgeDS_decorator<HDS>

Definition

The classes *CGAL::HalfedgeDS_items_decorator<HDS>*, *CGAL::HalfedgeDS_decorator<HDS>*, and *CGAL::HalfedgeDS_const_decorator<HDS>* provide additional functions to examine and to modify a halfedge data structure *HDS*. The class *CGAL::HalfedgeDS_items_decorator<HDS>* provides additional functions for vertices, halfedges, and faces of a halfedge data structure without knowing the containing halfedge data structure. The class *CGAL::HalfedgeDS_decorator<HDS>* stores a reference to the halfedge data structure and provides functions that modify the halfedge data structure, for example Euler-operators. The class *CGAL::HalfedgeDS_const_decorator<HDS>* stores a const reference to the halfedge data structure. It contains non-modifying functions, for example the test for validness of the data structure.

All these additional functions take care of the different capabilities a halfedge data structure may have or may not have. The functions evaluate the type tags of the halfedge data structure to decide on the actions. If a particular feature is not supported nothing is done. Note that for example the creation of new halfedges is mandatory for all halfedge data structures and will not appear here again.

```
#include <CGAL/HalfedgeDS_decorator.h>
```

Inherits From

CGAL::HalfedgeDS_items_decorator<HDS> page 881

Creation

HalfedgeDS_decorator<HDS> *D*(*HDS& hds*); keeps internally a reference to *hds*.

Creation of New Items

Vertex_handle *D.vertices_push_back*(*Vertex v*) appends a copy of *v* to *hds* if vertices are supported. Returns a handle of the new vertex, or *Vertex_handle()* otherwise.

Face_handle *D.faces_push_back*(*Face f*) appends a copy of *f* to *hds* if faces are supported. Returns a handle of the new face, or *Face_handle()* otherwise.

Creation of New Composed Items

Halfedge_handle *D.create_loop*() returns handle of a halfedge from a newly created loop in *hds* consisting of a single closed edge, one vertex and two faces (if supported respectively).

Halfedge_handle *D.create_segment*() returns a halfedge from a newly created segment in *hds* consisting of a single open edge, two vertices and one face (if supported respectively).

Removal of Elements

The following member functions do *not* update affected incidence relations except if mentioned otherwise.

- void D.vertices_pop_front()* removes the first vertex if vertices are supported.
Requirement: Supports_removal \equiv CGAL::Tag_true.
- void D.vertices_pop_back()* removes the last vertex if vertices are supported.
- void D.vertices_erase(Vertex_handle v)* removes the vertex *v* if vertices are supported.
Requirement: Supports_removal \equiv CGAL::Tag_true.
- void D.vertices_erase(Vertex_handle first, Vertex_handle last)*
removes the range $[first, last)$ if vertices are supported.
Requirement: Supports_removal \equiv CGAL::Tag_true.
- void D.faces_pop_front()* removes the first face if faces are supported.
Requirement: Supports_removal \equiv CGAL::Tag_true.
- void D.faces_pop_back()* removes the last face if faces are supported.
- void D.faces_erase(Face_handle f)* removes the face *f* if faces are supported.
Requirement: Supports_removal \equiv CGAL::Tag_true.
- void D.faces_erase(Face_handle first, Face_handle last)*
removes the range $[first, last)$ if faces are supported.
Requirement: Supports_removal \equiv CGAL::Tag_true.
- void D.erase_face(Halfedge_handle h)* removes the face incident to *h* from *hds* and changes all halfedges incident to the face into border edges or removes them from the halfedge data structure if they were already border edges. If this creates isolated vertices they get removed as well. See *make_hole(h)* for a more specialized variant.
Precondition: h->is_border() == false.
Requirement: If faces are supported, Supports_removal \equiv CGAL::Tag_true.
- void D.erase_connected_component(Halfedge_handle h)*
removes the vertices, halfedges, and faces that belong to the connected component of *h*.
Precondition: For all halfedges g in the connected component g.next() != Halfedge_handle().
Requirement: Supports_removal \equiv CGAL::Tag_true.

Modifying Functions (For Border Halfedges)

- Halfedge_handle D.make_hole(Halfedge_handle h)*
removes the face incident to *h* from *hds* and creates a hole.
Precondition: h != Halfedge_handle() and !(h->is_border()).
Requirement: If faces are supported, Supports_removal \equiv CGAL::Tag_true.

Halfedge_handle *D.fill_hole(Halfedge_handle h)*

fills the hole incident to *h* with a new face from *hds*. Returns *h*.
Precondition: *h* \neq *Halfedge_handle()* and *h->is_border()*.

Halfedge_handle *D.fill_hole(Halfedge_handle h, Face f)*

fills the hole incident to *h* with a copy of face *f*. Returns *h*.
Precondition: *h* \neq *Halfedge_handle()* and *h->is_border()*.

Halfedge_handle *D.add_face_to_border(Halfedge_handle h, Halfedge_handle g)*

extends the surface with a new face from *hds* into the hole incident to *h* and *g*. It creates a new edge connecting the vertex denoted by *g* with the vertex denoted by *h* and fills this separated part of the hole with a new face, such that the new face is incident to *g*. Returns the new halfedge that is incident to the new face.
Precondition: *h* \neq *Halfedge_handle()*, *g* \neq *Halfedge_handle()*, *h->is_border()*, *g->is_border()* and *g* can be reached along the hole starting with *h*.

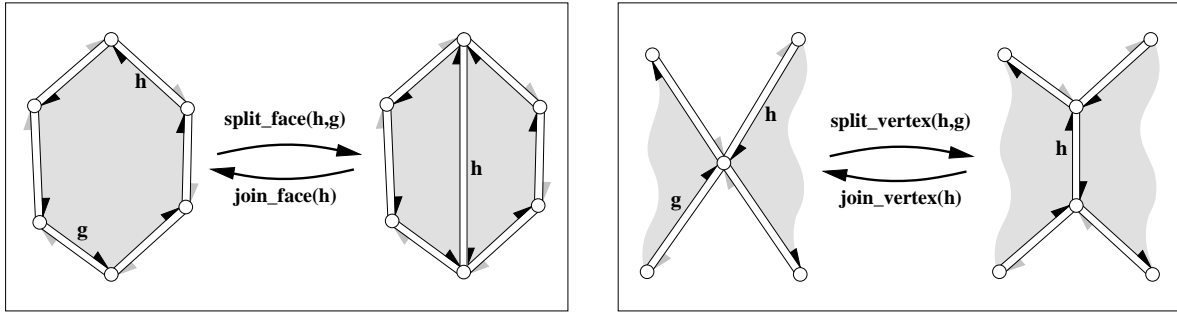
Halfedge_handle *D.add_face_to_border(Halfedge_handle h, Halfedge_handle g, Face f)*

extends the surface with a copy of face *f* into the hole incident to *h* and *g*. It creates a new edge connecting the tip of *g* with the tip of *h* and fills this separated part of the hole with a copy of face *f*, such that the new face is incident to *g*. Returns the new halfedge that is incident to the new face.
Precondition: *h* \neq *Halfedge_handle()*, *g* \neq *Halfedge_handle()*, *h->is_border()*, *g->is_border()* and *g* can be reached along the hole starting with *h*.

Modifying Functions (Euler Operators)

The following Euler operations modify consistently the combinatorial structure of the halfedge data structure. The geometry remains unchanged. Note that well known graph operations are also captured with these Euler operators, for example an edge contraction is equal to a *join_vertex()* operation, or an edge removal to *join_face()*.

Given a halfedge data structure *hds* and a halfedge handle *h* four special applications of the Euler operators are worth mentioning: *split_vertex(h,h)* results in an antenna emanating from the tip of *h*; *split_vertex(h,h->next()->opposite())* results in an edge split of the halfedge *h->next* with a new vertex in-between; *split_face(h,h)* results in a loop directly following *h*; and *split_face(h,h->next())* results in a bridge parallel to the halfedge *h->next* with a new face in-between.



Halfedge_handle *D.split_face(Halfedge_handle h, Halfedge_handle g)*

splits the face incident to *h* and *g* into two faces with a new diagonal between the two vertices denoted by *h* and *g* respectively. The second (new) face obtained from *hds* is a copy of the first face. Returns *h->next()* after the operation, i.e., the new diagonal. The new face is to the right of the new diagonal, the old face is to the left. The time is proportional to the distance from *h* to *g* around the face.

Halfedge_handle *D.join_face(Halfedge_handle h)*

joins the two faces incident to *h*. The face incident to *h->opposite()* gets removed from *hds*. Both faces might be holes. Returns the predecessor of *h* around the face. The invariant *join_face(split_face(h, g))* returns *h* and keeps the data structure unchanged. The time is proportional to the size of the face removed and the time to compute *h->prev()*.

Requirement: Supports_removal \equiv *CGAL::Tag_true*.

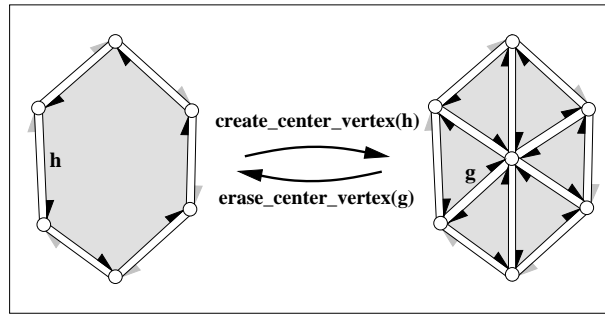
Halfedge_handle *D.split_vertex(Halfedge_handle h, Halfedge_handle g)*

splits the vertex incident to *h* and *g* into two vertices and connects them with a new edge. The second (new) vertex obtained from *hds* is a copy of the first vertex. Returns *h->next()->opposite()* after the operation, i.e., the new edge in the orientation towards the new vertex. The time is proportional to the distance from *h* to *g* around the vertex.

Halfedge_handle *D.join_vertex(Halfedge_handle h)*

joins the two vertices incident to *h*. The vertex denoted by *h->opposite()* gets removed by *hds*. Returns the predecessor of *h* around the vertex, i.e., *h->opposite()->prev()*. The invariant *join_vertex(split_vertex(h, g))* returns *h* and keeps the polyhedron unchanged. The time is proportional to the degree of the vertex removed and the time to compute *h->prev()* and *h->opposite()->prev()*.

Requirement: Supports_removal \equiv *CGAL::Tag_true*.



Halfedge_handle *D.create_center_vertex(Halfedge_handle h)*

barycentric triangulation of $h \rightarrow \text{face}()$. Creates a new vertex, a copy of $h \rightarrow \text{vertex}()$, and connects it to each vertex incident to $h \rightarrow \text{face}()$ splitting $h \rightarrow \text{face}()$ into triangles. h remains incident to the original face, all other triangles are copies of this face. Returns the halfedge $h \rightarrow \text{next}()$ after the operation, i.e., a halfedge pointing to the new vertex. The time is proportional to the size of the face.

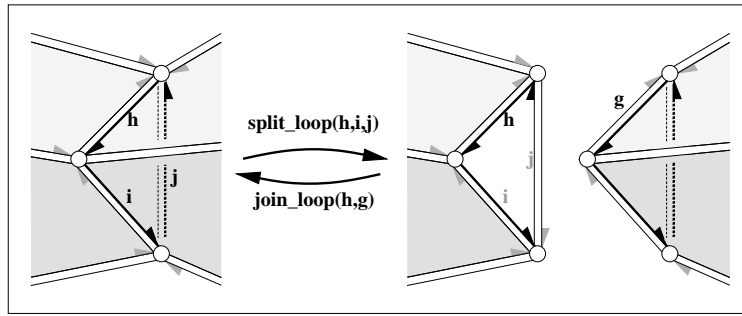
Precondition: h is not a border halfedge.

Halfedge_handle *D.erase_center_vertex(Halfedge_handle g)*

reverses *create_center_vertex*. Erases the vertex pointed to by g and all incident halfedges thereby merging all incident faces. Only $g \rightarrow \text{face}()$ remains. The neighborhood of $g \rightarrow \text{vertex}()$ may not be triangulated, it can have larger faces. Returns the halfedge $g \rightarrow \text{prev}()$. Thus, the invariant $h == \text{erase_center_vertex}(\text{create_center_vertex}(h))$ holds if h is not a border halfedge. The time is proportional to the sum of the size of all incident faces.

Precondition: None of the incident faces of $g \rightarrow \text{vertex}()$ is a hole. There are at least two distinct faces incident to the faces that are incident to $g \rightarrow \text{vertex}()$. (This prevents the operation from collapsing a volume into two faces glued together with opposite orientations, such as would happen with any vertex of a tetrahedron.)

Requirement: *Supports_removal* \equiv *CGAL::Tag_true*.



Halfedge_handle *D.split_loop(Halfedge_handle h, Halfedge_handle i, Halfedge_handle j)*

cuts the halfedge data structure into two parts along the cycle (h, i, j) . Three new vertices (one copy for each vertex in the cycle) and three new halfedges (one copy for each halfedge in the cycle), and two new triangles are created. h, i, j will be incident to the first new triangle. The return value will be the halfedge incident to the second new triangle which is the copy of h -*opposite()*.

Precondition: h, i, j denote distinct, consecutive vertices of the halfedge data structure and form a cycle: i.e., $h \rightarrow \text{vertex}() == i \rightarrow \text{opposite}() \rightarrow \text{vertex}(), \dots, j \rightarrow \text{vertex}() == h \rightarrow \text{opposite}() \rightarrow \text{vertex}()$.

Halfedge_handle *D.join_loop(Halfedge_handle h, Halfedge_handle g)*

glues the boundary of the two faces denoted by h and g together and returns h . Both faces and the vertices along the face denoted by g gets removed. Both faces may be holes. The invariant $\text{join_loop}(h, \text{split_loop}(h, i, j))$ returns h and keeps the data structure unchanged.

Precondition: The faces denoted by h and g are different and have equal degree (i.e., number of edges).

Requirement: *Supports_removal* \equiv *CGAL::Tag_true*.

Validness Checks

These operations are the same as for *CGAL::HalfedgeDS_const_decorator<HDS>*. See their documentation on page 863.

bool *D.is_valid(bool verbose = false, int level = 0)*

bool *D.normalized_border_is_valid(bool verbose = false)*

Miscellaneous

void *D.inside_out()* reverses face orientations.
Precondition: *is_valid()* of level three.

See Also

CGAL::HalfedgeDS_items_decorator<HDS> page [881](#)
CGAL::HalfedgeDS_const_decorator<HDS> page [863](#)

Example

The following program fragment illustrates the implementation of the Euler operator *split_vertex()* for a simplified polyhedron class.

```
template <class Traits>
namespace CGAL {
    class Polyhedron {
        typedef HalfedgeDS_default<Traits> HDS;
        HDS hds;
    public:
        // ...
        Halfedge_handle split_vertex( Halfedge_handle h, Halfedge_handle g) {
            HalfedgeDS_decorator<HDS> D(hds);
            // Stricter preconditions than for HalfedgeDS only.
            CGAL_precondition( D.get_vertex(h) == D.get_vertex(g));
            CGAL_precondition( h != g);
            return D.split_vertex( h, g);
        }
    };
}
```

CGAL::HalfedgeDS_default<Traits,HalfedgeDSItems,Alloc>

Definition

```
template < class Traits,
           class HalfedgeDSItems = CGAL::HalfedgeDS_items_2,
           class Alloc = CGAL_ALLOCATOR(int)>
class HalfedgeDS_default;
```

The class *HalfedgeDS_default*<Traits,HalfedgeDSItems,Alloc> is a model for the *HalfedgeDS* concept. The second template parameter *HalfedgeDSItems* has a default argument *CGAL::HalfedgeDS_items_2*. The third template parameter *Alloc* uses the CGAL default allocator as default setting. *HalfedgeDS_default*<Traits,HalfedgeDSItems,Alloc> is a list-based representation with bidirectional iterators that supports removal.

```
#include <CGAL/HalfedgeDS_default.h>
```

Is Model for the Concepts

HalfedgeDS<Traits,Items,Alloc> page [847](#)

Types

```
typedef bidirectional_iterator_tag    iterator_category;
typedef CGAL::Tag_true                Supports_removal;
```

See Also

CGAL::HalfedgeDS_list page [886](#)
CGAL::HalfedgeDS_vector page [888](#)
HalfedgeDSItems page [859](#)
CGAL::HalfedgeDS_items_2 page [879](#)
CGAL::Polyhedron_3<Traits> page [799](#)
CGAL::HalfedgeDS_items_decorator<HDS> page [881](#)
CGAL::HalfedgeDS_decorator<HDS> page [865](#)
CGAL::HalfedgeDS_const_decorator<HDS> page [863](#)

Implementation

Currently, *HalfedgeDS_default*<Traits,HalfedgeDSItems,Alloc> is derived from *CGAL::HalfedgeDS_list*<Traits>. The copy constructor and the assignment operator need $O(n)$ time with n the total number of vertices, halfedges, and faces. The former suboptimal implementation with an $O(n \log n)$ runtime has been replaced with a faster implementation based on hashing for the pointer lookup.

Due to a workaround for the flag *CGAL_CFG_NO_TMPL_IN_TMPL_PARAM*, a halfedge data structure cannot be instantiated directly. For the *HalfedgeDS_default*<Traits,HalfedgeDSItems,Alloc> a macro simplifies its direct use. However, when using a halfedge data structure as an argument for another class template, the class template name *HalfedgeDS_default* must be used, not the macro.


```

// The macro definition.
#ifndef CGAL_CFG_NO_TMPL_IN_TMPL_PARAM
    #define CGAL_HALFEDGE_DS_DEFAULT ::CGAL::HalfedgeDS_default
#else
    #define CGAL_HALFEDGE_DS_DEFAULT ::CGAL::HalfedgeDS_default::HDS
#endif

// The direct instantiation of the default HalfedgeDS given a Traits class.
typedef CGAL_HALFEDGE_DS_DEFAULT<Traits> HDS;

```

CGAL::HalfedgeDS_face_base<Refs>

Definition

The class *HalfedgeDS_face_base<Refs>* is a model of the *HalfedgeDSFace* concept. *Refs* is an instantiation of a *HalfedgeDS*. The template declaration of *HalfedgeDS_face_base<Refs>* has three parameters with some defaults that allow to select various flavors of faces. The declaration is best explained with the two following declarations, essentially hiding an implementation dependent default setting:

```
template <class Refs, class T = CGAL::Tag_true>
class HalfedgeDS_face_base;
```

```
template <class Refs, class T, class Plane>
class HalfedgeDS_face_base;
```

HalfedgeDS_face_base<Refs> defines a face including a reference to an incident halfedge.

CGAL::HalfedgeDS_face_base<Refs,CGAL::Tag_false> is a face without a reference to an incident halfedge. It is empty besides the required type definitions. It can be used for deriving own faces. See also *CGAL::HalfedgeDS_face_min_base<Refs>*.

CGAL::HalfedgeDS_face_base<Refs,CGAL::Tag_true,Plane> is a face with a reference to an incident halfedge and it stores a plane equation of type *Plane*. It can be used as a face for a model of the *PolyhedronItems_3* concept.

CGAL::HalfedgeDS_face_base<Refs,CGAL::Tag_false,Plane> is a face without a reference to an incident halfedge and it stores a plane equation of type *Plane*. It can be used as a face for a model of the *PolyhedronItems_3* concept.

```
#include <CGAL/HalfedgeDS_face_base.h>
```

Is Model for the Concepts

HalfedgeDSFacepage [855](#)

Types

HalfedgeDS_face_base<Refs>::Plane plane type for three argument version.

Creation

HalfedgeDS_face_base<Refs> f; default constructor.
HalfedgeDS_face_base<Refs> f(Plane pln); initialized with plane *pln*.

Operations

Plane& *f.plane()*
const Plane& *f.plane() const*

See Also

HalfedgeDS<Traits,Items,Alloc>	page 847
HalfedgeDSItems	page 859
PolyhedronItems_3	page 822
<i>CGAL::HalfedgeDS_items_2</i>	page 879
<i>CGAL::HalfedgeDS_vertex_base<Refs></i>	page 889
<i>CGAL::HalfedgeDS_halfedge_base<Refs></i>	page 877
<i>CGAL::HalfedgeDS_face_min_base<Refs></i>	page 876

CGAL::HalfedgeDS_face_min_base<Refs>

The class *HalfedgeDS_face_min_base<Refs>* is a model of the *HalfedgeDSFace* concept. *Refs* is an instantiation of a *HalfedgeDS*. It is equivalent to *CGAL::HalfedgeDS_face_base<Refs, CGAL::Tag_false>*. It is empty besides the required type definitions. It can be used for deriving own faces.

```
#include <CGAL/HalfedgeDS_face_min_base.h>
```

Is Model for the Concepts

HalfedgeDSFace page [855](#)

Creation

HalfedgeDS_face_min_base<Refs> *f*; default constructor.

See Also

HalfedgeDS<Traits,Items,Alloc> page [847](#)
 HalfedgeDSItems page [859](#)
 PolyhedronItems_3 page [822](#)
 CGAL::HalfedgeDS_min_items page [885](#)
 CGAL::HalfedgeDS_vertex_min_base<Refs> page [891](#)
 CGAL::HalfedgeDS_halfedge_min_base<Refs> page [878](#)
 CGAL::HalfedgeDS_face_base<Refs> page [874](#)

CGAL::HalfedgeDS_halfedge_base<Refs>

Definition

The class *HalfedgeDS_halfedge_base*<Refs> is a model of the *HalfedgeDSHalfedge* concept. *Refs* is an instantiation of a *HalfedgeDS*. The full declaration states four template parameters:

```
template < class Refs,
           class Tag_prev = CGAL::Tag_true,
           class Tag_vertex = CGAL::Tag_true,
           class Tag_face = CGAL::Tag_true>
class HalfedgeDS_halfedge_base;
```

If *Tag_prev* \equiv *CGAL::Tag_true* a reference to the previous halfedge is supported.

If *Tag_vertex* \equiv *CGAL::Tag_true* an incident vertex is supported.

If *Tag_face* \equiv *CGAL::Tag_true* an incident face is supported.

In all cases, a reference to the next halfedge and to the opposite halfedge is supported.

```
#include <CGAL/HalfedgeDS_halfedge_base.h>
```

Is Model for the Concepts

HalfedgeDSHalfedge page 857

Creation

HalfedgeDS_halfedge_base<Refs> *h*; default constructor.

See Also

HalfedgeDS<Traits,Items,Alloc> page 847
HalfedgeDSItems page 859
PolyhedronItems_3 page 822
CGAL::HalfedgeDS_items_2 page 879
CGAL::HalfedgeDS_vertex_base<Refs> page 889
CGAL::HalfedgeDS_face_base<Refs> page 874
CGAL::HalfedgeDS_halfedge_min_base<Refs> page 878

CGAL::HalfedgeDS_halfedge_min_base<Refs>

Definition

The class *HalfedgeDS_halfedge_min_base<Refs>* is a model of the *HalfedgeDSHalfedge* concept. *Refs* is an instantiation of a *HalfedgeDS*. It is equivalent to *CGAL::HalfedgeDS_halfedge_base<Refs, CGAL::Tag_false, CGAL::Tag_false, CGAL::Tag_false>*. The class contains support for the next and the opposite pointer and the required type definitions. It can be used for deriving own halfedges.

```
#include <CGAL/HalfedgeDS_halfedge_min_base.h>
```

Is Model for the Concepts

HalfedgeDSHalfedge page [857](#)

Creation

HalfedgeDS_halfedge_min_base<Refs> *h*; default constructor.

See Also

HalfedgeDS<Traits,Items,Alloc> page [847](#)
HalfedgeDSItems page [859](#)
PolyhedronItems_3 page [822](#)
CGAL::HalfedgeDS_min_items page [885](#)
CGAL::HalfedgeDS_vertex_min_base<Refs> page [891](#)
CGAL::HalfedgeDS_face_min_base<Refs> page [876](#)
CGAL::HalfedgeDS_halfedge_base<Refs> page [877](#)

CGAL::HalfedgeDS_items_2

Definition

The class *HalfedgeDS_items_2* is a model of the *HalfedgeDSItems* concept. It uses the default types for vertices, halfedges, and faces that declare all incidences supported by a *HalfedgeDS*. The vertex also contains a point of type *Traits::Point_2*, where *Traits* is the template argument of the corresponding *HalfedgeDS*.

```
#include <CGAL/HalfedgeDS_items_2.h>
```

Is Model for the Concepts

HalfedgeDSItems page [859](#)

See Also

CGAL::HalfedgeDS_min_items page [885](#)
CGAL::Polyhedron_items_3 page [824](#)
HalfedgeDS<*Traits*,*Items*,*Alloc*> page [847](#)
PolyhedronItems_3 page [822](#)
CGAL::HalfedgeDS_vertex_base<*Refs*> page [889](#)
CGAL::HalfedgeDS_halfedge_base<*Refs*> page [877](#)
CGAL::HalfedgeDS_face_base<*Refs*> page [874](#)

Example

The following example shows the canonical implementation of the *HalfedgeDS_items_2* class. It uses the base classes for the item types that are provided in the library.

```
struct HalfedgeDS_items_2 {
    template < class Refs, class Traits>
    struct Vertex_wrapper {
        typedef typename Traits::Point_2 Point;
        typedef CGAL::HalfedgeDS_vertex_base< Refs, Tag_true, Point> Vertex;
    };
    template < class Refs, class Traits>
    struct Halfedge_wrapper {
        typedef CGAL::HalfedgeDS_halfedge_base< Refs> Halfedge;
    };
    template < class Refs, class Traits>
    struct Face_wrapper {
        typedef CGAL::HalfedgeDS_face_base< Refs> Face;
    };
};
```

The following example shows a class definition for a new items class derived from the *HalfedgeDS_items_2* class. It replaces the *Face_wrapper* with a new definition of a face that contains a member variable for color. The new face makes use of the face base class provided in the library.

```

// A face type with a color member variable.
template <class Refs>
struct My_face : public CGAL::HalfedgeDS_face_base<Refs> {
    CGAL::Color color;
    My_face() {}
    My_face( CGAL::Color c) : color(c) {}
};

// An items type using my face.
struct My_items : public CGAL::HalfedgeDS_items_2 {
    template <class Refs, class Traits>
    struct Face_wrapper {
        typedef My_face<Refs> Face;
    };
};

```


CGAL::HalfedgeDS_items_decorator<HDS>

Definition

The classes *CGAL::HalfedgeDS_items_decorator<HDS>*, *CGAL::HalfedgeDS_decorator<HDS>*, and *CGAL::HalfedgeDS_const_decorator<HDS>* provide additional functions to examine and to modify a halfedge data structure *HDS*. The class *CGAL::HalfedgeDS_items_decorator<HDS>* provides additional functions for vertices, halfedges, and faces of a halfedge data structure without knowing the containing halfedge data structure. The class *CGAL::HalfedgeDS_decorator<HDS>* stores a reference to the halfedge data structure and provides functions that modify the halfedge data structure, for example Euler-operators. The class *CGAL::HalfedgeDS_const_decorator<HDS>* stores a const reference to the halfedge data structure. It contains non-modifying functions, for example the test for validness of the data structure.

All these additional functions take care of the different capabilities a halfedge data structure may have or may not have. The functions evaluate the type tags of the halfedge data structure to decide on the actions. If a particular feature is not supported nothing is done. Note that for example the creation of new halfedges is mandatory for all halfedge data structures and will not appear here again.

```
#include <CGAL/HalfedgeDS_items_decorator.h>
```

Types

<i>HalfedgeDS_items_decorator<HDS>:: HalfedgeDS</i>	halfedge data structure.
<i>HalfedgeDS_items_decorator<HDS>:: Traits</i>	traits class.
<i>HalfedgeDS_items_decorator<HDS>:: Vertex</i>	vertex type of <i>HalfedgeDS</i> .
<i>HalfedgeDS_items_decorator<HDS>:: Halfedge</i>	halfedge type of <i>HalfedgeDS</i> .
<i>HalfedgeDS_items_decorator<HDS>:: Face</i>	face type of <i>HalfedgeDS</i> .

```
HalfedgeDS_items_decorator<HDS>:: Vertex_handle
HalfedgeDS_items_decorator<HDS>:: Halfedge_handle
HalfedgeDS_items_decorator<HDS>:: Face_handle
HalfedgeDS_items_decorator<HDS>:: Vertex_iterator
HalfedgeDS_items_decorator<HDS>:: Halfedge_iterator
HalfedgeDS_items_decorator<HDS>:: Face_iterator
```

The respective *const_handle*'s and *const_iterator*'s are available as well.

```
HalfedgeDS_items_decorator<HDS>:: size_type
HalfedgeDS_items_decorator<HDS>:: difference_type
HalfedgeDS_items_decorator<HDS>:: iterator_category
```

```
HalfedgeDS_items_decorator<HDS>:: Supports_vertex_halfedge
HalfedgeDS_items_decorator<HDS>:: Supports_halfedge_prev
HalfedgeDS_items_decorator<HDS>:: Supports_halfedge_vertex
HalfedgeDS_items_decorator<HDS>:: Supports_halfedge_face
HalfedgeDS_items_decorator<HDS>:: Supports_face_halfedge
HalfedgeDS_items_decorator<HDS>:: Supports_removal
```

Creation

```
HalfedgeDS_items_decorator<HDS> D;
```

default constructor.

Access Functions

<i>Halfedge_handle</i>	<i>D.get_vertex_halfedge(Vertex_handle v)</i>	returns the incident halfedge of <i>v</i> if supported, <i>Halfedge_handle()</i> otherwise.
------------------------	-------------------------------------------------	------------------------------------------------------------------------------------------------

<i>Vertex_handle</i>	<i>D.get_vertex(Halfedge_handle h)</i>	returns the incident vertex of <i>h</i> if supported, <i>Vertex_handle()</i> otherwise.
----------------------	-----------------------------------------	--------------------------------------------------------------------------------------------

<i>Halfedge_handle</i>	<i>D.get_prev(Halfedge_handle h)</i>	returns the previous halfedge of <i>h</i> if supported, <i>Halfedge_handle()</i> otherwise.
------------------------	----------------------------------------	---------------------------------------------------------------------------------------------

<i>Halfedge_handle</i> <i>D.find_prev(Halfedge_handle h)</i>	returns the previous halfedge of <i>h</i> . Uses the <i>prev()</i> method if supported or performs a search around the face using <i>next()</i> .
---------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------

<i>Halfedge_handle</i>	<i>D.find_prev_around_vertex(Halfedge_handle h)</i> returns the previous halfedge of <i>h</i> . Uses the <i>prev()</i> method if supported or performs a search around the vertex using <i>next()</i> .
------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<i>Face_handle</i>	<i>D.get_face(Halfedge_handle h)</i>	returns the incident face of <i>h</i> if supported, <i>Face_handle()</i> otherwise.
--------------------	----------------------------------------	----------------------------------------------------------------------------------------

<i>Halfedge_handle</i>	<i>D.get_face_halfedge(Face_handle f)</i>	returns the incident halfedge of <i>f</i> if supported, <i>Halfedge_handle()</i> otherwise.
------------------------	--------------------------------------------	---------------------------------------------------------------------------------------------

Corresponding member functions for *const_handle*'s are provided as well.

Modifying Functions (Composed)

<i>void</i>	<i>D.close_tip(Halfedge_handle h)</i>	makes <i>h->opposite()</i> the successor of <i>h</i> .
-------------	----------------------------------------	-----------------------------------------------------------

<i>void</i>	<i>D.close_tip(Halfedge_handle h, Vertex_handle v)</i> makes <i>h->opposite()</i> the successor of <i>h</i> and sets the incident vertex of <i>h</i> to <i>v</i> .
-------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<code>void</code>	<code>D.insert_tip(Halfedge_handle h, Halfedge_handle v)</code>
	<p>inserts the tip of the edge h into the halfedges around the vertex pointed to by v. Halfedge $h \rightarrow \text{opposite}()$ is the new successor of v and $h \rightarrow \text{next}()$ will be set to $v \rightarrow \text{next}()$. The vertex of h will be set to the vertex v refers to if vertices are supported.</p>

<i>void</i>	<i>D.remove_tip(Halfedge_handle h)</i>	removes the edge <i>h->next()->opposite()</i> from the halfedge circle around the vertex referred to by <i>h</i> . The new successor halfedge of <i>h</i> will be <i>h->next()->opposite()->next()</i> .
-------------	------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<i>void</i>	<i>D.insert_halfedge(Halfedge_handle h, Halfedge_handle f)</i>	inserts the halfedge <i>h</i> between <i>f</i> and <i>f->next()</i> . The face of <i>h</i> will be the one <i>f</i> refers to if faces are supported.
<i>void</i>	<i>D.remove_halfedge(Halfedge_handle h)</i>	removes edge <i>h->next()</i> from the halfedge circle around the face referred to by <i>h</i> . The new successor of <i>h</i> will be <i>h->next()->next()</i> .
<i>void</i>	<i>D.set_vertex_in_vertex_loop(Halfedge_handle h, Vertex_handle v)</i>	loops around the vertex incident to <i>h</i> and sets all vertex pointers to <i>v</i> . <i>Precondition: h != Halfedge_handle().</i>
<i>void</i>	<i>D.set_face_in_face_loop(Halfedge_handle h, Face_handle f)</i>	loops around the face incident to <i>h</i> and sets all face pointers to <i>f</i> . <i>Precondition: h != Halfedge_handle().</i>
<i>Halfedge_handle</i>	<i>D.flip_edge(Halfedge_handle h)</i>	performs an edge flip. It returns <i>h</i> after rotating the edge <i>h</i> one vertex in the direction of the face orientation. <i>Precondition: h != Halfedge_handle() and both incident faces of h are triangles.</i>

Modifying Functions (Primitives)

<i>void</i>	<i>D.set_vertex_halfedge(Vertex_handle v, Halfedge_handle g)</i>	sets the incident halfedge of <i>v</i> to <i>g</i> .
<i>void</i>	<i>D.set_vertex_halfedge(Halfedge_handle h)</i>	sets the incident halfedge of the vertex incident to <i>h</i> to <i>h</i> .
<i>void</i>	<i>D.set_vertex(Halfedge_handle h, Vertex_handle v)</i>	sets the incident vertex of <i>h</i> to <i>v</i> .
<i>void</i>	<i>D.set_prev(Halfedge_handle h, Halfedge_handle g)</i>	sets the previous link of <i>h</i> to <i>g</i> .
<i>void</i>	<i>D.set_face(Halfedge_handle h, Face_handle f)</i>	sets the incident face of <i>h</i> to <i>f</i> .
<i>void</i>	<i>D.set_face_halfedge(Face_handle f, Halfedge_handle g)</i>	sets the incident halfedge of <i>f</i> to <i>g</i> .
<i>void</i>	<i>D.set_face_halfedge(Halfedge_handle h)</i>	sets the incident halfedge of the face incident to <i>h</i> to <i>h</i> .

See Also

CGAL::HalfedgeDS_decorator<HDS> page [865](#)
CGAL::HalfedgeDS_const_decorator<HDS> page [863](#)

Example

The following program fragment illustrates how a refined halfedge class for a polyhedron can make use of the *find_prev()* member function to implement a *prev()* member function that works regardless of whether the halfedge data structure *HDS* provides a *prev()* member function for its halfedges or not. In the case that not, the implementation given here runs in time proportional to the size of the incident face. For const-correctness a second implementation with signature *Halfedge_const_handle prev() const;* is needed.

Note also the use of the static member function *halfedge_handle()* of the halfedge data structure. It converts a pointer to the halfedge into a halfedge handle. This conversion encapsulates possible adjustments for hidden data members in the true halfedge type, such as linked-list pointers.

```
struct Polyhedron_halfedge {
    // ...
    Halfedge_handle prev() {
        CGAL::HalfedgeDS_items_decorator<HDS> decorator;
        return decorator.find_prev( HDS::halfedge_handle(this));
    }
};
```

CGAL::HalfedgeDS_min_items

Definition

The class *HalfedgeDS_min_items* is a model of the *HalfedgeDSItems* concept. It defines types for vertices, halfedges, and faces that declare the minimal required incidences for a *HalfedgeDS*, which are the *next()* and the *opposite()* member function for halfedges.

```
#include <CGAL/HalfedgeDS_min_items.h>
```

Is Model for the Concepts

HalfedgeDSItems page 859

See Also

CGAL::HalfedgeDS_items_2 page 879
 CGAL::Polyhedron_items_3 page 824
 HalfedgeDS<Traits,Items,Alloc> page 847
 PolyhedronItems_3 page 822
 CGAL::HalfedgeDS_vertex_min_base<Refs> page 891
 CGAL::HalfedgeDS_halfedge_min_base<Refs> page 878
 CGAL::HalfedgeDS_face_min_base<Refs> page 876

Example

The following example shows the canonical implementation of the *CGAL::HalfedgeDS_min_items* class. It uses the base classes for the item types that are provided in the library.

```
struct HalfedgeDS_min_items {
    template < class Refs, class Traits>
    struct Vertex_wrapper {
        typedef CGAL::HalfedgeDS_vertex_min_base< Refs>    Vertex;
    };
    template < class Refs, class Traits>
    struct Halfedge_wrapper {
        typedef CGAL::HalfedgeDS_halfedge_min_base< Refs> Halfedge;
    };
    template < class Refs, class Traits>
    struct Face_wrapper {
        typedef CGAL::HalfedgeDS_face_min_base< Refs>      Face;
    };
};
```

CGAL::HalfedgeDS_list<Traits,HalfedgeDSItems,Alloc>

Definition

The class *HalfedgeDS_list*<Traits,HalfedgeDSItems,Alloc> is a model for the *HalfedgeDS* concept. *HalfedgeDS_list*<Traits,HalfedgeDSItems,Alloc> is a list-based representation with bidirectional iterators that supports removal.

```
#include <CGAL/HalfedgeDS_list.h>
```

Is Model for the Concepts

HalfedgeDS<Traits,Items,Alloc> page [847](#)

Types

```
typedef bidirectional_iterator_tag    iterator_category;
typedef CGAL::Tag_true               Supports_removal;
```

Operations

Besides operations required from the concept *HalfedgeDS*<Traits,Items,Alloc>, this class supports additionally:

```
void    hds.vertices_splice( Vertex_iterator target, Self &source, Vertex_iterator first, Vertex_iterator last)
```

inserts elements in the range *[first, last)* before position *target* and removes the elements from *source*. It takes constant time if *&source == &hds*; otherwise, it takes linear time in the size of the range.

Precondition: *[first, last)* is a valid range in *source*. *target* is not in the range *[first, last)*.

```
void    hds.halfedges_splice( Halfedge_iterator target,
                             Self &source,
                             Halfedge_iterator first,
                             Halfedge_iterator last)
```

inserts elements in the range *[first, last)* before position *target* and removes the elements from *source*. It takes constant time if *&source == &hds*; otherwise, it takes linear time in the size of the range.

Precondition: *[first, last)* is a valid range in *source*. *target* is not in the range *[first, last)*.

```
void    hds.faces_splice( Face_iterator target, Self &source, Face_iterator first, Face_iterator last)
```

inserts elements in the range *[first, last)* before position *target* and removes the elements from *source*. It takes constant time if *&source == &hds*; otherwise, it takes linear time in the size of the range.

Precondition: *[first, last)* is a valid range in *source*. *target* is not in the range *[first, last)*.

See Also

<i>CGAL::HalfedgeDS_default</i>	page 872
<i>CGAL::HalfedgeDS_vector</i>	page 888
<i>HalfedgeDSItems</i>	page 859
<i>CGAL::Polyhedron_3<Traits></i>	page 799
<i>CGAL::HalfedgeDS_items_decorator<HDS></i>	page 881
<i>CGAL::HalfedgeDS_decorator<HDS></i>	page 865
<i>CGAL::HalfedgeDS_const_decorator<HDS></i>	page 863

Implementation

HalfedgeDS_list<Traits,HalfedgeDSItems,Alloc> uses internally the *CGAL::In_place_list* container class. The copy constructor and the assignment operator need $O(n)$ time with n the total number of vertices, halfedges, and faces. The former suboptimal implementation with an $O(n \log n)$ runtime has been replaced with a faster implementation based on hashing for the pointer lookup.

CGAL_ALLOCATOR(int) is used as default argument for the *Alloc* template parameter.

CGAL::HalfedgeDS_vector<Traits,HalfedgeDSItems,Alloc>

Definition

The class *HalfedgeDS_vector*<Traits,HalfedgeDSItems,Alloc> is a model for the *HalfedgeDS* concept. *HalfedgeDS_vector*<Traits,HalfedgeDSItems,Alloc> is a vector-based representation with random access iterators that does not support removal.

```
#include <CGAL/HalfedgeDS_vector.h>
```

Is Model for the Concepts

HalfedgeDS<Traits,Items,Alloc> page [847](#)

Types

```
typedef random_access_iterator_tag    iterator_category;
typedef CGAL::Tag_false               Supports_removal;
```

See Also

CGAL::HalfedgeDS_default page [872](#)
CGAL::HalfedgeDS_list page [886](#)
HalfedgeDSItems page [859](#)
CGAL::Polyhedron_3<Traits> page [799](#)
CGAL::HalfedgeDS_items_decorator<HDS> page [881](#)
CGAL::HalfedgeDS_decorator<HDS> page [865](#)
CGAL::HalfedgeDS_const_decorator<HDS> page [863](#)

Implementation

HalfedgeDS_vector<Traits,HalfedgeDSItems,Alloc> uses internally the STL *std::vector* container class. We require that we can create a *std::vector::iterator* from a pointer. If this will not be true any longer for any major STL distribution we might switch to an internal implementation of a vector.

The capacity is restricted to the reserved size. Allocations are not possible beyond the capacity without calling reserve again. All handles and iterators are invalidated upon a reserve call that increases the capacity.

CGAL_ALLOCATOR(int) is used as default argument for the *Alloc* template parameter.

CGAL::HalfedgeDS_vertex_base<Refs>

Definition

The class *HalfedgeDS_vertex_base<Refs>* is a model of the *HalfedgeDSVertex* concept. *Refs* is an instantiation of a *HalfedgeDS*. The template declaration of *HalfedgeDS_vertex_base<Refs>* has three parameters with some defaults that allow to select various flavors of vertices. The declaration is best explained with the two following declarations, essentially hiding an implementation dependent default setting:

```
template <class Refs, class T = CGAL::Tag_true>
class HalfedgeDS_vertex_base;
```

```
template <class Refs, class T, class Point>
class HalfedgeDS_vertex_base;
```

HalfedgeDS_vertex_base<Refs> defines a vertex including a reference to an incident halfedge.

CGAL::HalfedgeDS_vertex_base<Refs,CGAL::Tag_false> is a vertex without a reference to an incident halfedge. It is empty besides the required type definitions. It can be used for deriving own vertex implementations. See also *CGAL::HalfedgeDS_vertex_min_base<Refs>*.

CGAL::HalfedgeDS_vertex_base<Refs,CGAL::Tag_true,Point> is a vertex with a reference to an incident halfedge and it stores a point of type *Point*. It can be used as a vertex for a model of the *PolyhedronItems_3* concept.

CGAL::HalfedgeDS_vertex_base<Refs,CGAL::Tag_false,Point> is a vertex without a reference to an incident halfedge and it stores a point of type *Point*. It can be used as a vertex for a model of the *PolyhedronItems_3* concept.

```
#include <CGAL/HalfedgeDS_vertex_base.h>
```

Is Model for the Concepts

HalfedgeDSVertex page [861](#)

Types

HalfedgeDS_vertex_base<Refs>::Point point type for three argument version.

Creation

<i>HalfedgeDS_vertex_base<Refs> v;</i>	default constructor.
<i>HalfedgeDS_vertex_base<Refs> v(Point p);</i>	initialized with point <i>p</i> .

Operations

<i>Point&</i>	<i>v.point()</i>
<i>const Point&</i>	<i>v.point() const</i>

See Also

HalfedgeDS<Traits,Items,Alloc>	page 847
HalfedgeDSItems	page 859
PolyhedronItems_3	page 822
<i>CGAL::HalfedgeDS_items_2</i>	page 879
<i>CGAL::HalfedgeDS_halfedge_base<Refs></i>	page 877
<i>CGAL::HalfedgeDS_face_base<Refs></i>	page 874
<i>CGAL::HalfedgeDS_vertex_min_base<Refs></i>	page 891

CGAL::HalfedgeDS_vertex_min_base<Refs>

Definition

The class *HalfedgeDS_vertex_min_base<Refs>* is a model of the *HalfedgeDSVertex* concept. *Refs* is an instantiation of a *HalfedgeDS*. It is equivalent to *CGAL::HalfedgeDS_vertex_base<Refs, CGAL::Tag_false>*. It is empty besides the required type definitions. It can be used for deriving own vertices.

```
#include <CGAL/HalfedgeDS_vertex_min_base.h>
```

Is Model for the Concepts

HalfedgeDSVertex page [861](#)

Creation

HalfedgeDS_vertex_min_base<Refs> *v*; default constructor.

See Also

HalfedgeDS<Traits,Items,Alloc> page [847](#)
 HalfedgeDSItems page [859](#)
 PolyhedronItems_3 page [822](#)
 CGAL::HalfedgeDS_min_items page [885](#)
 CGAL::HalfedgeDS_halfedge_min_base<Refs> page [878](#)
 CGAL::HalfedgeDS_face_min_base<Refs> page [876](#)
 CGAL::HalfedgeDS_vertex_base<Refs> page [889](#)

Part V

Polygon and Polyhedron Operations

Chapter 12

2D Regularized Boolean Set-Operations

Efi Fogel, Ron Wein, Baruch Zukerman, and Dan Halperin

12.1 Introduction

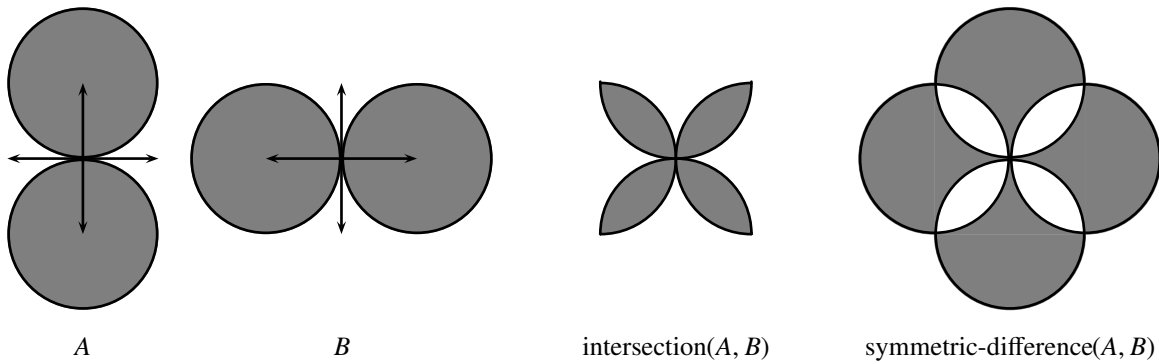


Figure 12.1: Examples of Boolean set-operations on general polygons.

This package consists of the implementation of Boolean set-operations on point sets bounded by x -monotone curves¹ in 2-dimensional Euclidean space. In particular, it contains the implementation of *regularized* Boolean set-operations, intersection predicates, and point containment predicates. Figure 12.1 shows simple examples of such operations.

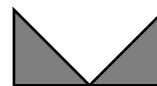
A regularized Boolean set-operation op^* can be obtained by first taking the interior of the resultant point set of an *ordinary* Boolean set-operation ($P \text{ op } Q$) and then by taking the closure [Hof04]. That is, $P \text{ op}^* Q = \text{closure}(\text{interior}(P \text{ op } Q))$. Regularized Boolean set-operations appear in Constructive Solid Geometry (CSG), because regular sets are closed under regularized Boolean set-operations, and because regularization eliminates lower dimensional features, namely isolated vertices and antennas, thus simplifying and restricting the representation to physically meaningful solids. Our package provides regularized operations on polygons and general polygons, where the edges of a general polygon may be general x -monotone curves, rather than being simple line segments. Ordinary Boolean set-operations, which distinguish between the interior and the boundary of a polygon, are not implemented within this package. The *Nef_2* package supports these operations for (linear) polygons; see Chapter 13.

¹A continuous planar curve C is *x-monotone* if every vertical line intersects it at most once. We also allow vertical line segments, which are considered *weakly x-monotone*.

In the rest of this chapter we use — unless otherwise stated — the traditional notation to designate regularized operations; e.g., $P \cap Q$ means the *regularized* intersection of P and Q .

A polygon P is said to be *simple* (or Jordan) if the only points of the plane belonging to two polygon edges of P are the polygon vertices of P . Namely, the polygon edges are pairwise disjoint in their interior. Such a polygon has a well-defined interior and exterior and is topologically equivalent to a disk. A polygon in our context must be simple and its vertices must be ordered in a counterclockwise direction around the interior of the polygon.

The counterclockwise cyclic sequence of alternating polygon edges and polygon vertices is referred to as the polygon *boundary*. A polygon whose boundary contains the same vertex twice or more is connected and simple but not necessarily strictly simple, such as the polygon depicted on the right. We extend the notion of a polygon to a point set in \mathbb{R}^2 that has a topology of a polygon and its boundary edges must map to x -monotone curves, and refer to it as a *general polygon*. We sometimes use the term *polygon* instead of general polygon for simplicity hereafter.



Our package supports the following Boolean set-operations on two point sets P and Q that are comprised of general polygons:

Intersection computes the intersection $R = P \cap Q$.

Join computes the union $R = P \cup Q$.

Difference computes the difference $R = P \setminus Q$.

Symmetric Difference computes the symmetric difference $P = P \oplus Q = (P \setminus Q) \cup (Q \setminus P)$.

Complement computes the complement $R = \overline{P}$.

Intersection predicate tests whether the two sets P and Q overlap, distinguishing three possible scenarios: (i) the two sets intersect on their interior (that is, their regularized intersection is not empty $P \cap Q \neq \emptyset$); (ii) the boundaries of two sets intersect but their interiors are disjoint; namely they have a finite number of common points or even share a boundary curve (still in this case $P \cap Q = \emptyset$); and (iii) the two sets are disjoint.

In general, the set R , resulting from a regularized Boolean set-operation, is considered as being a closed point-set.

In the rest of this chapter we review the Boolean set-operations package in more depth. In Section 12.2 we focus on Boolean set-operations on linear polygons, introducing the notion of polygons with holes and of a general polygon set. Section 12.3 introduces general polygons. We first discuss polygons whose edges are either line segments or circular arcs and then explain how to construct and use general polygons whose edges can be arbitrary x -monotone curves.

12.2 Boolean Set-Operations on Linear Polygons

The basic library of CGAL includes the `Polygon_2<Kernel, Container>` class-template that represents a linear polygon in the plane. The polygon is represented by its vertices stored in a container of objects of type `Kernel::Point_2`. The polygon edges are line segments (`Kernel::Segment_2` objects) between adjacent points in the container. By default, the `Container` is a vector of `Kernel::Point_2` objects.

The following function demonstrates how to use the basic access functions of the `Polygon_2` class. It accepts a polygon P and prints it in a readable format:


```

template<class Kernel, class Container>
void print_polygon (const CGAL::Polygon_2<Kernel, Container>& P)
{
    typename CGAL::Polygon_2<Kernel, Container>::Vertex_const_iterator vit;

    std::cout << "[ " << P.size() << " vertices:";
    for (vit = P.vertices_begin(); vit != P.vertices_end(); ++vit)
        std::cout << " (" << *vit << ')';
    std::cout << " ]" << std::endl;
}

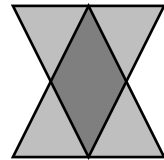
```

In this section we use the term *polygon* to indicate a *Polygon_2* instance, namely, a polygon having linear edges. General polygons are only discussed in Section 12.3.

The basic components of our package are the free (global) functions *complement()* that accepts a single *Polygon_2* object, and *intersection()*, *join()*,² *difference()*, *symmetric_difference()* and the predicate *do_intersect()* that accept two *Polygon_2* objects as their input. We explain how these functions should be used through several examples in the following sections.

A Simple Example

Testing whether two polygons intersect results with a Boolean value, and does not require any additional data beyond the provision of the two input polygons. The example listed below tests whether the two triangles depicted on the right intersect. It uses, as do the other example programs in this chapter, the auxiliary header file *bso_rational_nt.h*, which defines the type *Number_type* as GMP's rational number-type (*CGAL::Gmpq*), or as the number type *Quotient<MP_Float>* that is included in the support library of CGAL, based on whether the GMP library is installed or not. It also uses the function *print_polygon.h* listed above, which is located in the header file *print_utils.h*.



```

//! \file examples/Boolean_set_operations_2/ex_do_intersect.C
// Determining whether two triangles intersect.

```

```

#include "bso_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Boolean_set_operations_2.h>

typedef CGAL::Cartesian<Number_type>          Kernel;
typedef Kernel::Point_2                       Point_2;
typedef CGAL::Polygon_2<Kernel>               Polygon_2;

#include "print_utils.h"

int main ()
{
    Polygon_2 P;
    P.push_back (Point_2 (-1,1));
    P.push_back (Point_2 (0,-1));
    P.push_back (Point_2 (1,1));
    std::cout << "P = "; print_polygon (P);

    Polygon_2 Q;
    Q.push_back (Point_2 (-1,-1));
    Q.push_back (Point_2 (1,-1));

```

²The function that computes the union of two polygons is called *join()*, since the word *union* is reserved in C++.

```

Q.push_back(Point_2 (0,1));
std::cout << "Q = "; print_polygon (Q);

if ((CGAL::do_intersect (P, Q)))
    std::cout << "The two polygons intersect in their interior." << std::endl;
else
    std::cout << "The two polygons do not intersect." << std::endl;

return 0;
}

```

12.2.1 Polygons with Holes

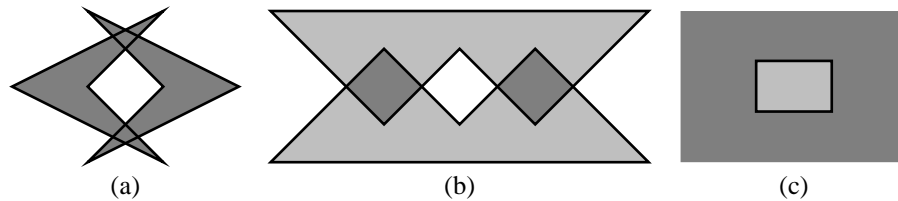


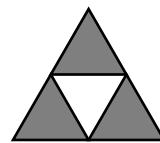
Figure 12.2: Operations on strictly simple polygons. (a) The union of two polygons, resulting in a point set whose outer boundary is defined by a simple polygon and contains a polygonal hole in its interior. (b) The intersection (darkly shaded) of two polygons (lightly shaded), resulting in two disjoint polygons. (c) The complement (darkly shaded) of a strictly simple polygon (lightly shaded).

In many cases a binary operation that operates on two strictly simple polygons that have no holes may result in a set of polygon that contains holes in their interior (see Figure 12.2.1 (a)), or a set of disjoint polygons (see Figure 12.2.1 (b)); a similar set may result from the union, or the symmetric difference, of two disjoint polygons). Moreover, the complement of a simple polygon is an unbounded set that contains a hole; see Figure 12.2.1 (c).

Regular sets are closed under regularized Boolean set-operations. These operations accept as input, and may result as output, polygons with holes. A *polygon with holes* represents a point set that may be bounded or unbounded. In case of a bounded set, its *outer boundary* is represented as a simple (but not necessarily strictly simple) polygon, whose vertices are oriented in a counterclockwise order around the interior of the set. In addition, the set may contain *holes*, where each hole is represented as a strictly simple polygon, whose vertices are oriented in a clockwise order around the interior of the hole. Note that an unbounded polygon without holes spans the entire plane. Vertices of holes may coincide with vertices of the boundary; see below for an example.

A point set represented by a polygon with holes is considered to be closed. Therefore, the boundaries of the holes are parts of the set (and not part of the holes). The exact definition of the obtained polygon with holes as a result of a Boolean set-operation or a sequence of such operations is closely related to the definition of regularized Boolean set-operations, being the closure of the interior of the corresponding ordinary operation as explained next.

Consider, for example, the regular set depicted on the right, which is the result of the union of three small triangles translated appropriately. Alternatively, the same set can be reached by taking the difference between a large triangle and a small upside-down triangle. In general, there are many ways to arrive at a particular point set. However, the set of polygons with holes obtained through the application of any sequence of operations is unique. The set depicted on the right is represented as a single polygon having a triangular outer boundary with a single triangular hole in its interior — and not as three triangles that have no holes at all. As a general rule, if two point sets are connected, then they belong to the same polygon with holes.



The class template *Polygon_with_holes_2*<Kernel,Container> represents polygons with holes as described above, where the outer boundary and the hole boundaries are realized as *Polygon_2*<Kernel,Container> objects. Given an instance *P* of the *Polygon_with_holes_2* class, you can use the predicate *is_unbounded*() to check whether *P* is a unbounded set. If it is bounded, you can obtain the counterclockwise-oriented polygon that represents its outer boundary through the member function *outer_boundary*(). You can also traverse the holes of *P* through *holes_begin*() and *holes_end*(). The two functions return iterators of the nested type *Polygon_with_holes_2::Hole_const_iterator* that defines the valid range of *P*'s holes. The value type of this iterator is *Polygon_2*.

The following function demonstrates how to traverse a polygon with holes. It accepts a *Polygon_with_holes_2* object and uses the auxiliary function *print_polygon*() to prints all its components in a readable format:

```
template<class Kernel, class Container>
void print_polygon_with_holes(const CGAL::Polygon_with_holes_2<Kernel, Container> & pwh)
{
    if (! pwh.is_unbounded()) {
        std::cout << "{ Outer boundary = ";
        print_polygon (pwh.outer_boundary());
    }
    else
        std::cout << "{ Unbounded polygon." << std::endl;

    typename CGAL::Polygon_with_holes_2<Kernel,Container>::Holes_const_iterator hit;
    unsigned int k = 1;

    std::cout << " " << pwh.number_of_holes() << " holes:" << std::endl;
    for (hit = pwh.holes_begin(); hit != pwh.holes_end(); ++hit, ++k) {
        std::cout << "    Hole #" << k << " = ";
        print_polygon (*hit);
    }
    std::cout << " }" << std::endl;
}
```

The simple versions of the free functions mentioned therefore accept two *Polygon_2* object *P* and *Q* as their input, while their output is given using polygon with holes instances:

- The complement of a simple polygon *P* is always an unbounded set with a single polygonal hole. The function *complement(P)* therefore returns a polygon-with-holes object that represents the complement of *P*.
- The union of two polygons *P* and *Q* is always a single connected set, unless of course the two input polygons are completely disjoint. In the latter case $P \cup Q$ trivially consists of the two input polygons. The free function *join(P, Q, R)* therefore returns a Boolean value indicating whether $P \cap Q \neq \emptyset$. If the two polygons are not disjoint, it assigns the a polygon with holes object *R* (which it accepts by reference) with the union of the regularized union operation $P \cup Q$.

- The other three functions, namely *intersection(P, Q, oi)*, *difference(P, Q, oi)* and *symmetric_difference(P, Q, oi)*, all have a similar interface. As the result of these operation may consist of several disconnected components, they all accept an output iterator *oi*, whose value type is *Polygon_with_holes_2*, and add the output polygons to its associated container.

Example — Joining and Intersecting Simple Polygons

The following example demonstrates the usage of the free functions *join()* and *intersect()* for computing the union and the intersection of the two simple polygons depicted in Figure 12.2.1 (b). The example uses the auxiliary function *print_polygon_with_holes()* listed above, which is located in the header file *print_utils.h* under the examples folder.

```

//! \file examples/Boolean_set_operations_2/ex_simple_join_intersect.C
// Computing the union and the intersection of two simple polygons.

#include "bso_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Boolean_set_operations_2.h>
#include <list>

typedef CGAL::Cartesian<Number_type>          Kernel;
typedef Kernel::Point_2                       Point_2;
typedef CGAL::Polygon_2<Kernel>               Polygon_2;
typedef CGAL::Polygon_with_holes_2<Kernel>    Polygon_with_holes_2;
typedef std::list<Polygon_with_holes_2>       Pwh_list_2;

#include "print_utils.h"

int main ()
{
    // Construct the two input polygons.
    Polygon_2 P;
    P.push_back (Point_2 (0, 0));
    P.push_back (Point_2 (5, 0));
    P.push_back (Point_2 (3.5, 1.5));
    P.push_back (Point_2 (2.5, 0.5));
    P.push_back (Point_2 (1.5, 1.5));

    std::cout << "P = "; print_polygon (P);

    Polygon_2 Q;
    Q.push_back (Point_2 (0, 2));
    Q.push_back (Point_2 (1.5, 0.5));
    Q.push_back (Point_2 (2.5, 1.5));
    Q.push_back (Point_2 (3.5, 0.5));
    Q.push_back (Point_2 (5, 2));

    std::cout << "Q = "; print_polygon (Q);

    // Compute the union of P and Q.
    Polygon_with_holes_2 unionR;

```

```

if (CGAL::join (P, Q, unionR)) {
    std::cout << "The union: ";
    print_polygon_with_holes (unionR);
} else
    std::cout << "P and Q are disjoint and their union is trivial."
                << std::endl;
std::cout << std::endl;

// Compute the intersection of P and Q.
Pwh_list_2          intR;
Pwh_list_2::const_iterator it;

CGAL::intersection (P, Q, std::back_inserter(intR));

std::cout << "The intersection:" << std::endl;
for (it = intR.begin(); it != intR.end(); ++it) {
    std::cout << "--> ";
    print_polygon_with_holes (*it);
}

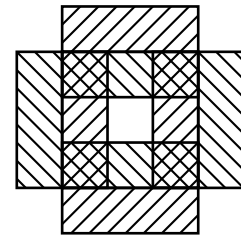
return 0;
}

```

Operations on Polygons with Holes

Having introduced polygons with holes and explained how the free functions output such objects, it is only natural to perform operations on sets that are represented as polygon with holes, rather than simple polygons. Indeed, the Boolean set-operations package provides overridden free functions *complement()*, *intersection()*, *join()*, *difference()*, *symmetric_difference()* and *do_intersect()* that accept *General_polygon_with_holes_2* objects as their input. The prototypes of most functions is the same as of their simpler counterparts that operate on simple polygons. The only exception is *complement(P, oi)*, which outputs a range of polygons with holes that represents the complement of the polygon with holes *P*.

The following example demonstrates how to compute the symmetric difference between two sets that contain holes. Each set is a rectangle that contains a rectangular hole in its interior, such that the symmetric difference between the two sets is a single polygon that contains of five holes:



```

//! \file examples/Boolean_set_operations_2/ex_symmetric_difference.C
// Computing the symmetric difference of two polygons with holes.

#include "bso_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Boolean_set_operations_2.h>
#include <list>

typedef CGAL::Cartesian<Number_type>          Kernel;
typedef Kernel::Point_2                       Point_2;
typedef CGAL::Polygon_2<Kernel>              Polygon_2;

```

```

typedef CGAL::Polygon_with_holes_2<Kernel>          Polygon_with_holes_2;
typedef std::list<Polygon_with_holes_2>             Pwh_list_2;

#include "print_utils.h"

int main ()
{
    // Construct P - a bounded rectangle that contains a rectangular hole.
    Polygon_2 outP;
    Polygon_2 holesP[1];

    outP.push_back (Point_2 (-3, -5)); outP.push_back (Point_2 (3, -5));
    outP.push_back (Point_2 (3, 5));   outP.push_back (Point_2 (-3, 5));
    holesP[0].push_back (Point_2 (-1, -3));
    holesP[0].push_back (Point_2 (-1, 3));
    holesP[0].push_back (Point_2 (1, 3));
    holesP[0].push_back (Point_2 (1, -3));

    Polygon_with_holes_2 P (outP, holesP, holesP + 1);
    std::cout << "P = "; print_polygon_with_holes (P);

    // Construct Q - a bounded rectangle that contains a rectangular hole.
    Polygon_2 outQ;
    Polygon_2 holesQ[1];

    outQ.push_back (Point_2 (-5, -3)); outQ.push_back (Point_2 (5, -3));
    outQ.push_back (Point_2 (5, 3));   outQ.push_back (Point_2 (-5, 3));
    holesQ[0].push_back (Point_2 (-3, -1));
    holesQ[0].push_back (Point_2 (-3, 1));
    holesQ[0].push_back (Point_2 (3, 1));
    holesQ[0].push_back (Point_2 (3, -1));

    Polygon_with_holes_2 Q (outQ, holesQ, holesQ + 1);
    std::cout << "Q = "; print_polygon_with_holes (Q);

    // Compute the symmetric difference of P and Q.
    Pwh_list_2 symmR;
    Pwh_list_2::const_iterator it;

    CGAL::symmetric_difference (P, Q, std::back_inserter(symmR));

    std::cout << "The symmetric difference:" << std::endl;
    for (it = symmR.begin(); it != symmR.end(); ++it) {
        std::cout << "--> ";
        print_polygon_with_holes (*it);
    }

    return 0;
}

```

12.2.2 Operating on Polygon Sets

We argue that the result of a regularized operations on two polygons (or polygons with holes) P and Q is typically a collection of several disconnected polygons with holes. It is only natural to represent such a collection in terms of a class, making it possible to operate on the set resulting from computing, for example, $P \setminus Q$.

A central component in the Boolean set-operations package is the *Polygon_set_2*<*Kernel*, *Container*> class-template. An instance of this class represents a point set formed by the collection of several disconnected polygons with holes. It employs the *Arrangement_2* class to represent this point set in the plane as a planar arrangement; see Chapter 17.

An instance S of a *Polygon_set_2* class usually represents the result of a sequence of operations that were applied on some input polygons. The representation of S is unique, regardless of the particular sequence of operations that were applied in order to arrive at it.

In addition, a polygon-set object can be constructed from a single polygon object or from a polygon-with-holes object. Once constructed, it is possible to insert new polygons (or polygons with holes) into the set using the *insert()* method, as long as the inserted polygons and the existing polygons in the set are disjoint. The *Polygon_set_2* class also provides access functions for accessing the polygons with holes it contains, and a few queries. The most important query is *S.oriented_side(q)*, which determined whether the query point q is contained in the interior of the set S , lies on the boundary of the set, or neither.

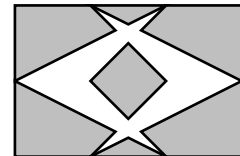
The *General_polygon_set_2* class defines the predicate *do_intersect()* and the methods *complement()*, *intersection()*, *join()*, *difference()* and *symmetric_difference()* as member functions. The operands to these functions may be simple polygons (*Polygon_2* object), polygons with holes (*General_polygon_with_holes_2* objects), or polygon sets (*General_polygon_set_2* objects). Thus, each of the function mentioned above is actually realized by a set overriding member functions.

Member functions of the *General_polygon_set_2* that perform Boolean set-operations come in two flavors: for example, *S.join(P, Q)* computes the union of P and Q and assigned the result to S , while *S.join(P)* performs the operation $S \leftarrow S \cup P$. Similarly, *S.complement(P)* sets S to be the complement of P , while *S.complement()* simply negates the set S .

A Sequence of Set Operations

The free functions reviewed in Section 12.2.1 serve as a wrapper for the polygon-set class, and are only provided for convenience. A typical such function constructs a pair of *General_polygon_set_2* objects, invokes the appropriate method to apply the desired Boolean operation, and transforms the resulting polygon set to the required output format. Thus, when several operations are performed in a sequence, it is much more efficient to use the member functions of the *General_polygon_set_2* type directly, as the extraction of the polygons from the internal representation for some operation, and the reconstruction of the internal representation for the succeeding operation could be time consuming.

The next example performs a sequence of three Boolean set-operations. First, it computes the union of two simple polygons depicted in Figure 12.2.1 (a). It then computes the complement of the result of the union operation. Finally, it computes the intersection of the result of the complement operation with a rectangle, confining the final result to the area of the rectangle. The resulting set S is comprised of two components: a polygon with a hole, and a simple polygon contained in the interior of this hole.



```
/// \file examples/Boolean_set_operations_2/ex_sequence.C
// Performing a sequence of Boolean set-operations.
```

```

#include "bso_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Polygon_2.h>
#include <CGAL/Polygon_with_holes_2.h>
#include <CGAL/Polygon_set_2.h>

#include <list>

typedef CGAL::Cartesian<Number_type>      Kernel;
typedef Kernel::Point_2                   Point_2;
typedef CGAL::Polygon_2<Kernel>           Polygon_2;
typedef CGAL::Polygon_with_holes_2<Kernel> Polygon_with_holes_2;
typedef CGAL::Polygon_set_2<Kernel>       Polygon_set_2;

#include "print_utils.h"

int main ()
{
    // Construct the two initial polygons and the clipping rectangle.
    Polygon_2 P;
    P.push_back (Point_2 (0, 1));
    P.push_back (Point_2 (2, 0));
    P.push_back (Point_2 (1, 1));
    P.push_back (Point_2 (2, 2));

    Polygon_2 Q;
    Q.push_back (Point_2 (3, 1));
    Q.push_back (Point_2 (1, 2));
    Q.push_back (Point_2 (2, 1));
    Q.push_back (Point_2 (1, 0));

    Polygon_2 rect;
    rect.push_back (Point_2 (0, 0));
    rect.push_back (Point_2 (3, 0));
    rect.push_back (Point_2 (3, 2));
    rect.push_back (Point_2 (0, 2));

    // Perform a sequence of operations.
    Polygon_set_2 S;
    S.insert (P);
    S.join (Q);           // Compute the union of P and Q.
    S.complement();       // Compute the complement.
    S.intersection (rect); // Intersect with the clipping rectangle.

    // Print the result.
    std::list<Polygon_with_holes_2> res;
    std::list<Polygon_with_holes_2>::const_iterator it;

    std::cout << "The result contains " << S.number_of_polygons_with_holes()
               << " components:" << std::endl;

    S.polygons_with_holes (std::back_inserter (res));
    for (it = res.begin(); it != res.end(); ++it) {
        std::cout << "--> ";
    }
}

```



```

    print_polygon_with_holes (*it);
}

return 0;
}

```

Inserting Non Intersecting Polygons

If you want to compute the union of a polygon P (P may be a simple polygon or a polygon-with-holes object) with a general-polygon set R , and store the result in R , you can construct a polygon set $S(P)$, and apply the *union* operation as follows:

```

General_polygon_2 S (P);
R.join (S);

```

As a matter of fact, you can apply the union operation directly:

```

R.join (P);

```

However, if you know that the polygon does not intersect any one of the polygons represented by R , you can use the more efficient method *insert()*:

```

R.insert (P);

```

As *insert()* assumes that $P \cap R = \emptyset$, it does not try to compute intersections between the boundaries of P and of R . This fact significantly speeds up the insertion process in comparison with the insertion of a non-disjoint polygon that intersects R .

The *insert()* function is also overloaded, so it can also accept a range of polygons. When a range of polygons are inserted into a polygon set S , all the polygons in the range and the polygons represented by S must be pairwise disjoint in their interiors.

12.2.3 Performing Aggregated Operations

There are a few options to compute the union of a set of polygons P_1, \dots, P_m . You can do it incrementally as follows. At each step compute the union of $S_{k-1} = \bigcup_{i=1}^{k-1} P_i$ with P_k and obtain S_k . Namely, if the polygon set is given as a pair of iterator $[begin, end)$, the following loop computes their union in S .

```

InputIterator iter = begin;
Polygon_set_2 S (*iter);

while (++iter != end) {
    S.join (*iter);
    ++iter;
}

```

A second option is to use a divide-and-conquer approach. You bisect the set of polygons into two sets. Compute the union of each set recursively and obtain the partial results in S_1 and S_2 , and finally, you compute the union $S_1 \cup S_2$. However, the union operation can be done more efficiently for sparse polygons, having a relatively small number of intersections, using a third option that simultaneously computes the union of all polygons. This is done by constructing a planar arrangement of all input polygons, utilizing the sweep-line algorithm, then extracting the result from the arrangement. Similarly, it is also possible to aggregately compute the intersection $\bigcap_{i=1}^m P_i$ of a set of input polygons.

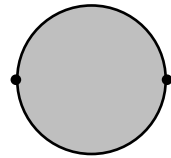
Our package provides the free overloaded functions *join()* and *intersect()* that aggregately compute the union and the intersection of a range of input polygons. There is no restriction on the polygons in the range — naturally, they may intersect each other. The package provides the overloaded free function *do_intersect(begin, end)* that determines whether the polygons in the range defined by the two input iterators *[begin, end)* intersect.

The class *General_polygon_set_2* also provides equivalent member functions that aggregately operate on a range of input polygons or polygons with holes. When such a member function is called, the general polygons represented by the current object are considered operands as well. Thus, you can easily compute the union of our polygon range as follows:

```
Polygon_set_2 S;  
S.join (begin, end);
```

12.3 Boolean Set-Operations on General Polygons

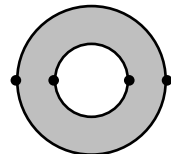
In previous sections ordinary polygons were dealt with. Namely, closed point sets bounded by piecewise linear curves. The Boolean set-operations package allows a more general geometric mapping of the polygon edges. The operations provided by the package operate on point sets bounded by x -monotone segments of general curves (e.g., conic arcs and segments of polynomial functions). For example, the point set depicted on the right is general polygon bounded by two x -monotone circular arcs that correspond to the lower half and the upper half of a circle.



Using the topological terminology, a general polygon can represent any simply-connected point set whose boundary is a strictly simple curve. Such a polygon is a model of the *GeneralPolygon_2* concept. A model of this concept must fulfill the following requirements:

- A general polygon is constructible from a range of pairwise interior disjoint x -monotone curves c_1, \dots, c_n . The target point of the k th curve c_k and the source point of the next curve in the range (in a cyclic order) must coincide, such that this point defines the k th polygon vertex.
- It is possible to traverse the x -monotone curves that form the edges of a general polygon.

The concept *GeneralPolygonWithHoles_2* is defined in an analogous way to the definition of linear polygons with holes. A model of this concept represent a bounded or an unbounded set that may not be simply connected, and must provide the following operations:



- Construction for a general polygon that represent the outer boundary and a range of general polygons that represent the holes.
- Accessing the general polygons that represents the outer boundary (in case of a bounded set).

- Traversing the holes.

In Section 12.2 we introduce the classes *Polygon_2* and *Polygon_with_holes_2* that model the concepts *GeneralPolygon_2* and *GeneralPolygonWithHoles_2* respectively. In this section we introduce other models of these two concepts.

The central class-template *General_polygon_set_2<Traits>* is used to represent point sets that are comprised of a finite number of general polygons with holes that are pairwise disjoint, and provides various Boolean set-operations on such sets. It is parameterized by a *traits* class that defines the type of points used to represent polygon vertices and the type of *x*-monotone curves that represent the polygon edges. The traits class also provides primitive geometric operations that operate on objects of these types. An instantiated *General_polygon_set_2* class defines the nested types *General_polygon_set_2<Traits>::Polygon_2* and *General_polygon_set_2<Traits>::Polygon_with_holes_2*, which model the concepts *GeneralPolygon_2* and *GeneralPolygonWithHoles_2* respectively.

12.3.1 The Traits-Class Concepts

The traits class used to instantiate the *General_polygon_set_2* class template must model the concept *GeneralPolygonSetTraits_2*, and is tailored to handle a specific family of curves. The concept *GeneralPolygonSetTraits_2* refines the concept *ArrangementDirectionalXMonotoneTraits_2* specified next.

The concept *ArrangementDirectionalXMonotoneTraits_2* refines the concept *ArrangementXMonotoneTraits_2* (see Section 17.4.1 in the 2D Arrangements chapter). Thus, a model of this concept must define the type *X_monotone_curve_2*, which represents an *x*-monotone curve, and the type *Point_2*, which represents a planar point. Such a point may be an endpoint of an *x*-monotone curve or an intersection point between two curves. It must also provide various geometric predicates and operations on these types listed by the base concept, such as determining whether a point lies above or below an *x*-monotone curve, computing the intersections between two curves, etc. Note that the base concept does not assume that *x*-monotone curves are directed: an *x*-monotone curve is not required to have a designated *source* and *target*, it is only required to determine the left (lexicographically smaller) and the right (lexicographically larger) endpoints of a given curve.

The *ArrangementDirectionalXMonotoneTraits_2* concept treats its *x*-monotone curves as directed objects. It thus requires two additional operations on *x*-monotone curves:

- Given an *x*-monotone curve, compare its source and target points lexicographically.
- Given an *x*-monotone curve, construct its opposite curve (namely, swap its source and target points).

The traits classes *Arr_segment_traits_2*, *Arr_non_caching_segment_traits*, *Arr_circle_segment_traits_2*, *Arr_conic_traits_2* and *Arr_rational_arc_traits_2*, which are bundled in the *Arrangement_2* package and distributed with CGAL, are all models of the refined concept *ArrangementDirectionalXMonotoneTraits_2*.³

Just as with the case of computations using models of the *ArrangementXMonotoneTraits_2* concept, operations are robust only when exact arithmetic is used. When inexact arithmetic is used, (nearly) degenerate configurations may result in abnormal termination of the program or even incorrect results.

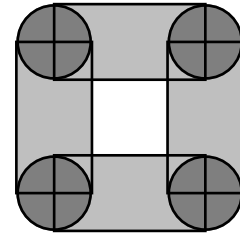
³The *Arr_polyline_traits_2* class is *not* a model of the, *ArrangementDirectionalXMonotoneTraits_2* concept, as the *x*-monotone curve it defines is always directed from left to right. Thus, an opposite curve cannot be constructed. However, it is not very useful to construct a polygon whose edges are polylines, as an ordinary polygon with linear edges can represent the same entity.

12.3.2 Operating on Polygons with Circular Arcs

Two traits classes are distributed with CGAL. The first one is named *Gps_segment_traits_2*, and it is used to perform Boolean set-operations on ordinary polygons and polygons with holes. In fact, the class *Polygon_set_2* introduced in Section 12.2.2 is a specialization of *General_polygon_set_2<Gps_segment_traits_2>*. This class defined its polygon and polygon with holes types, such that the usage of this traits class is encapsulated in the polygon-set class.

The other predefined traits class is named *Gps_circle_segment_traits_2<Kernel>* and is parameterized by a geometric CGAL kernel. By instantiating the *General_polygon_set_2* with this traits class, we obtain the representation of a polygon whose boundary may be comprised of line segments and circular arcs. The circle-segment traits class provides predicates and constructions on non-linear objects; yet, it uses only rational arithmetic and is very efficient as a consequence.

The following example uses the *Gps_circle_segment_traits_2* class to compute the union of four rectangles and four circles. Each circle is represented as a general polygon having two *x*-monotone circular arcs. The union is computed incrementally, resulting with a single polygon with a single hole, as depicted on the right. Note that as the four circles are disjoint, their union is computed with the *insert* method, while the union with the rectangles is computed with the *join* operator.



```
#!/ \file examples/Boolean_set_operations_2/ex_circle_segment.C
// Handling circles and linear segments concurrently.

#include "bso_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Gps_circle_segment_traits_2.h>
#include <CGAL/General_polygon_set_2.h>
#include <CGAL/General_polygon_with_holes_2.h>
#include <CGAL/Lazy_exact_nt.h>

#include <list>

typedef CGAL::Lazy_exact_nt<Number_type>          Lazy_exact_nt;
typedef CGAL::Cartesian<Lazy_exact_nt>           Kernel;
typedef Kernel::Point_2                           Point_2;
typedef Kernel::Circle_2                         Circle_2;
typedef CGAL::Gps_circle_segment_traits_2<Kernel> Traits_2;

typedef CGAL::General_polygon_set_2<Traits_2>      Polygon_set_2;
typedef Traits_2::Polygon_2                       Polygon_2;
typedef Traits_2::Polygon_with_holes_2            Polygon_with_holes_2;
typedef Traits_2::Curve_2                        Curve_2;
typedef Traits_2::X_monotone_curve_2              X_monotone_curve_2;

// Construct a polygon from a circle.
Polygon_2 construct_polygon (const Circle_2& circle)
{
    // Subdivide the circle into two x-monotone arcs.
    Traits_2 traits;
    Curve_2 curve (circle);
    std::list<CGAL::Object> objects;
```

```

traits.make_x_monotone_2_object() (curve, std::back_inserter(objects));
CGAL_assertion (objects.size() == 2);

// Construct the polygon.
Polygon_2 pgn;
X_monotone_curve_2 arc;
std::list<CGAL::Object>::iterator iter;

for (iter = objects.begin(); iter != objects.end(); ++iter) {
    CGAL::assign (arc, *iter);
    pgn.push_back (arc);
}

return pgn;
}

// Construct a point from a rectangle.
Polygon_2 construct_polygon (const Point_2& p1, const Point_2& p2,
                             const Point_2& p3, const Point_2& p4)
{
    Polygon_2 pgn;
    X_monotone_curve_2 s1(p1, p2);    pgn.push_back(s1);
    X_monotone_curve_2 s2(p2, p3);    pgn.push_back(s2);
    X_monotone_curve_2 s3(p3, p4);    pgn.push_back(s3);
    X_monotone_curve_2 s4(p4, p1);    pgn.push_back(s4);
    return pgn;
}

// The main program:
int main ()
{
    // Insert four non-intersecting circles.
    Polygon_set_2 S;
    Polygon_2 circ1, circ2, circ3, circ4;

    circ1 = construct_polygon(Circle_2(Point_2(1, 1), 1)); S.insert(circ1);
    circ2 = construct_polygon(Circle_2(Point_2(5, 1), 1)); S.insert(circ2);
    circ3 = construct_polygon(Circle_2(Point_2(5, 5), 1)); S.insert(circ3);
    circ4 = construct_polygon(Circle_2(Point_2(1, 5), 1)); S.insert(circ4);

    // Compute the union with four rectangles incrementally.
    Polygon_2 rect1, rect2, rect3, rect4;

    rect1 = construct_polygon(Point_2(1, 0), Point_2(5, 0),
                              Point_2(5, 2), Point_2(1, 2));
    S.join (rect1);

    rect2 = construct_polygon(Point_2(1, 4), Point_2(5, 4),
                              Point_2(5, 6), Point_2(1, 6));
    S.join (rect2);

    rect3 = construct_polygon(Point_2(0, 1), Point_2(2, 1),
                              Point_2(2, 5), Point_2(0, 5));
    S.join (rect3);

```

```

rect4 = construct_polygon(Point_2(4, 1), Point_2(6, 1),
                          Point_2(6, 5), Point_2(4, 5));
S.join (rect4);

// Print the output.
std::list<Polygon_with_holes_2> res;
S.polygons_with_holes (std::back_inserter (res));

std::copy (res.begin(), res.end(),
           std::ostream_iterator<Polygon_with_holes_2>(std::cout, "\n"));
std::cout << std::endl;

return 0;
}

```

12.3.3 General Polygon Set Traits Adapter

The concept *GeneralPolygon_2* and its generic model *General_polygon_2<ArrDirectionalXMonotoneTraits>* facilitate the production of general-polygon set traits classes. A model of the concept *GeneralPolygon_2* represents a simple point-set in the plane bounded by *x*-monotone curves. As opposed to the plain *Traits::Polygon_2* type defined by any traits class, it must define the type *X_monotone_curve_2*, which represents an *x*-monotone curve of the point-set boundary. It must provide a constructor from a range of such curves, and a pair of methods, namely *curves_begin()* and *curves_end()*, that can be used to iterate over the point-set boundary curves.

The class-template *General_polygon_2<ArrDirectionalXMonotoneTraits>* models the concept *GeneralPolygon_2*. Its template parameter must be instantiated with a model of the concept *ArrangementDirectionalXMonotoneTraits_2* from which it obtains the *X_monotone_curve_2* type, and it uses the necessary operations on this type provided by such a model to maintain a container of directed curves of type *X_monotone_curve_2*, which represents the boundary of the general polygon.

The class-template *Gps_traits_2<ArrDirectionalXMonotoneTraits,GeneralPolygon>* models the concept *GeneralPolygonSetTraits_2*. Its implementation is rather simple, as it is derived from the instantiated template-parameter *ArrXMonotoneTraits_2* inheriting its necessary types and methods, and it exploits the methods provided by the instantiated parameter *GeneralPolygon* — a model of the concept *GeneralPolygon_2*. By default, the *GeneralPolygon* parameter is defined as *General_polygon_2<ArrangementDirectionalXMonotoneTraits_2>*

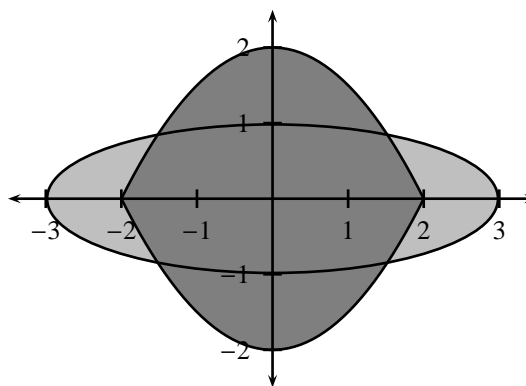
The code excerpt listed below defines a general-polygon set type that can be used to perform Boolean set-operations on point sets bounded by linear segments used by the *Arrangement_2* class by default. A model of the *GeneralPolygon_2* concept that represents a (linear) polygon bounded by curves of type *Arr_segment_2* is generated. The later is obtained from the instantiated parameter *Arr_segment_traits_2*, which defines *Arr_segment_2* to be its exposed type *X_monotone_curve_2*.

```

typedef CGAL::Gmpq                                     Number_type;
typedef CGAL::Cartesian<Number_type>                   Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>              Arr_traits_2;
typedef CGAL::General_polygon_2<Arr_traits_2>           General_polygon_2;
typedef CGAL::Gps_traits_2<Arr_traits_2,General_polygon_2> Traits_2;
typedef CGAL::General_polygon_set_2<Traits_2>          General_polygon_set_2;

```

Swapping the linear arrangement-traits *Arr_segment_traits_2* above with a traits class that handle conic arcs, such as *Arr_conic_traits_2*, results with the definition of a general-polygon set type that can be used to perform Boolean set-operations on point sets bounded by conic arcs of type *Arr_conic_2*. The next example computes the intersection of the two general polygons depicted on the right. One is an ellipse given by $x^2 + 9y^2 - 9 = 0$, and the other is bounded by the two parabolic arcs whose underlying parabola are given by $x^2 + 2y - 4 = 0$, and $x^2 - 2y - 4 = 0$. The code in the example adapts the traits model that handles conics included with the *Arrangement_2* package.



```

//! \file examples/Boolean_set_operations_2/ex_traits_adaptor.C
// Using the traits adaptor to generate a traits of conics.
#include <CGAL/basic.h>

#ifndef CGAL_USE_CORE
#include <iostream>
int main ()
{
    std::cout << "Sorry, this example needs CORE ..." << std::endl;
    return (0);
}
#else

#include <CGAL/Cartesian.h>
#include <CGAL/CORE_algebraic_number_traits.h>
#include <CGAL/Arr_conic_traits_2.h>
#include <CGAL/General_polygon_2.h>
#include <CGAL/General_polygon_with_holes_2.h>
#include <CGAL/Gps_traits_2.h>
#include <CGAL/Boolean_set_operations_2.h>

#include <list>

typedef CGAL::CORE_algebraic_number_traits          Nt_traits;
typedef Nt_traits::Rational                        Rational;
typedef Nt_traits::Algebraic                       Algebraic;
typedef CGAL::Cartesian<Rational>                  Rat_kernel;
typedef CGAL::Cartesian<Algebraic>                  Alg_kernel;
typedef CGAL::Arr_conic_traits_2<Rat_kernel,
                                Alg_kernel,Nt_traits> Conic_traits_2;
typedef CGAL::General_polygon_2<Conic_traits_2>    Polygon_2;
typedef CGAL::Gps_traits_2<Conic_traits_2, Polygon_2> Traits_2;
typedef Traits_2::Polygon_with_holes_2             Polygon_with_holes_2;
typedef Traits_2::Curve_2                           Curve_2;
typedef Traits_2::X_monotone_curve_2                X_monotone_curve_2;
typedef Traits_2::Point_2                           Point_2;

// Insert a conic arc as a polygon edge: Subdivide the arc into x-monotone
// sub-arcs and append these sub-arcs as polygon edges.
void append_conic_arc (Polygon_2& polygon, const Curve_2& arc)
{

```

```

Conic_traits_2          traits;
std::list<CGAL::Object> objects;
std::list<CGAL::Object>::iterator it;
X_monotone_curve_2      xarc;

traits.make_x_monotone_2_object() (arc, std::back_inserter(objects));
for (it = objects.begin(); it != objects.end(); ++it)
{
    if (CGAL::assign (xarc, *it))
        polygon.push_back (xarc);
}

int main ()
{
    // Construct a parabolic arc supported by a parabola:  $x^2 + 2y - 4 = 0$ ,
    // and whose endpoints lie on the line  $y = 0$ :
    Curve_2 parabola1 = Curve_2 (1, 0, 0, 0, 2, -4, CGAL::COUNTERCLOCKWISE,
                                Point_2(2, 0), Point_2(-2, 0));

    // Construct a parabolic arc supported by a parabola:  $x^2 - 2y - 4 = 0$ ,
    // and whose endpoints lie on the line  $y = 0$ :
    Curve_2 parabola2 = Curve_2 (1, 0, 0, 0, -2, -4, CGAL::COUNTERCLOCKWISE,
                                Point_2(-2, 0), Point_2(2, 0));

    // Construct a polygon from these two parabolic arcs.
    Polygon_2 P;
    append_conic_arc (P, parabola1);
    append_conic_arc (P, parabola2);

    // Construct a polygon that corresponds to the ellipse:  $x^2 + 9y^2 - 9 = 0$ :
    Polygon_2 Q;
    append_conic_arc (Q, Curve_2 (-1, -9, 0, 0, 0, 9));

    // Compute the intersection of the two polygons.
    std::list<Polygon_with_holes_2> res;
    CGAL::intersection (P, Q, std::back_inserter(res));

    std::copy (res.begin(), res.end(),          // export to standard output
               std::ostream_iterator<Polygon_with_holes_2>(std::cout, "\n"));
    std::cout << std::endl;

    return (0);
}

#endif

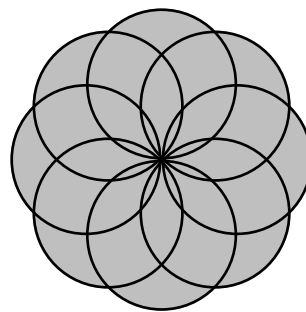
```

12.3.4 Example - Aggregated Operations

In Section [12.2.3](#) we describe how aggregated union and intersection operations can be applied to a collection of ordinary polygons or polygons with holes. Naturally, the aggregated operations can be applied also to collections of general polygons. As was the case with ordinary polygons, using aggregated operations is recommended when the number of intersections of the input polygons is of the same order of magnitude as the complexity of

the result. If this is not the case, computing the result incrementally may prove faster.

The next example computes the union of eight unit discs whose centers are placed a unit distance from the origin, as depicted to the right. The example also allows users to provide a different number of discs through the command line.



```

//! \file examples/Boolean_set_operations_2/ex_set_union.C
// Computing the union of a set of circles.

#include "bso_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Gps_circle_segment_traits_2.h>
#include <CGAL/Boolean_set_operations_2.h>
#include <CGAL/Lazy_exact_nt.h>

#include <list>
#include <stdlib.h>
#include <math.h>

typedef CGAL::Lazy_exact_nt<Number_type>           Lazy_exact_nt;
typedef CGAL::Cartesian<Lazy_exact_nt>             Kernel;
typedef Kernel::Point_2                             Point_2;
typedef Kernel::Circle_2                             Circle_2;
typedef CGAL::Gps_circle_segment_traits_2<Kernel>   Traits_2;

typedef CGAL::General_polygon_set_2<Traits_2>       Polygon_set_2;
typedef Traits_2::Polygon_2                         Polygon_2;
typedef Traits_2::Polygon_with_holes_2              Polygon_with_holes_2;
typedef Traits_2::Curve_2                           Curve_2;
typedef Traits_2::X_monotone_curve_2                X_monotone_curve_2;

// Construct a polygon from a circle.
Polygon_2 construct_polygon (const Circle_2& circle)
{
    // Subdivide the circle into two x-monotone arcs.
    Traits_2 traits;
    Curve_2 curve (circle);
    std::list<CGAL::Object> objects;
    traits.make_x_monotone_2_object() (curve, std::back_inserter(objects));
    CGAL_assertion (objects.size() == 2);

    // Construct the polygon.
    Polygon_2 pgn;
    X_monotone_curve_2 arc;
    std::list<CGAL::Object>::iterator iter;

    for (iter = objects.begin(); iter != objects.end(); ++iter) {

```

```

        CGAL::assign (arc, *iter);
        pgn.push_back (arc);
    }

    return pgn;
}

// The main program:
int main (int argc, char * argv[])
{
    // Read the number of circles from the command line.
    unsigned int n_circles = 8;
    if (argc > 1) n_circles = atoi(argv[1]);

    // Create the circles, equally spaced of the circle  $x^2 + y^2 = 1$ .
    const double pi = std::atan(1.0) * 4;
    const double n_circles_reciep = 1.0 / n_circles;
    const double radius = 1;
    const double frac = 2 * pi * n_circles_reciep;
    std::list<Polygon_2> circles;
    unsigned int k;
    for (k = 0; k < n_circles; k++) {
        const double angle = frac * k;
        const double x = radius * std::sin(angle);
        const double y = radius * std::cos(angle);
        Point_2 center = Point_2(x, y);
        Circle_2 circle(center, radius);

        circles.push_back (construct_polygon (circle));
    }

    // Compute the union aggragately.
    std::list<Polygon_with_holes_2> res;
    CGAL::join (circles.begin(), circles.end(), std::back_inserter (res));

    // Print the result.
    std::copy (res.begin(), res.end(),
               std::ostream_iterator<Polygon_with_holes_2>(std::cout, "\n"));
    std::cout << std::endl;

    return 0;
}

```

2D Regularized Boolean Set-Operations Reference Manual

Efi Fogel, Ron Wein, Baruch Zukerman, and Dan Halperin

Introduction

This package consists of the implementation of Boolean set-operations on point sets bounded by x -monotone curves in 2-dimensional Euclidean space. In particular, it contains the implementation of regularized Boolean set-operations, intersection predicates, and point containment predicates.

12.4 Classified Reference Pages

Concepts

GeneralPolygon_2	page 924
GeneralPolygonWithHoles_2	page 926
ArrangementDirectionalXMonotoneTraits_2	page 928
GeneralPolygonSetTraits_2	page 930

Classes

CGAL::Polygon_with_holes_2<Kernel, Container>	page 933
CGAL::Polygon_set_2<Kernel, Container>	page 934
CGAL::General_polygon_set_2<Traits>	page 917
CGAL::General_polygon_2<ArrTraits>	page 932
CGAL::General_polygon_with_holes_2<Polygon>	page 935
CGAL::Gps_segment_traits_2<Kernel, Container, ArrSegmentTraits>	page 936
CGAL::Gps_circle_segment_traits_2<Kernel>	page 937
CGAL::Gps_traits_2<ArrTraits, GeneralPolygon>	page 938

Functions

CGAL::complement	page 939
CGAL::do_intersect	page 943

<i>CGAL::intersection</i>	page 945
<i>CGAL::join</i>	page 947
<i>CGAL::difference</i>	page 941
<i>CGAL::symmetric_difference</i>	page 949
<i>CGAL::operator<<</i>	page 951
<i>CGAL::operator>></i>	page 952

12.5 Alphabetical List of Reference Pages

<i>ArrangementDirectionalXMonotoneTraits_2</i>	page 928
<i>complement</i>	page 939
<i>difference</i>	page 941
<i>do_intersect</i>	page 943
<i>GeneralPolygonSetTraits_2</i>	page 930
<i>GeneralPolygonWithHoles_2</i>	page 926
<i>GeneralPolygon_2</i>	page 924
<i>General_polygon_2<ArrTraits></i>	page 932
<i>General_polygon_set_2<Traits></i>	page 917
<i>General_polygon_with_holes_2<Polygon></i>	page 935
<i>Gps_circle_segment_traits_2<Kernel></i>	page 937
<i>Gps_segment_traits_2<Kernel,Container,ArrSegmentTraits></i>	page 936
<i>Gps_traits_2<ArrTraits,GeneralPolygon></i>	page 938
<i>intersection</i>	page 945
<i>join</i>	page 947
<i>operator<<</i>	page 2800
<i>operator>></i>	page 2792
<i>Polygon_set_2<Kernel,Container></i>	page 934
<i>Polygon_with_holes_2<Kernel,Container></i>	page 933
<i>symmetric_difference</i>	page 949

CGAL::General_polygon_set_2<Traits>

Definition

An object of the *General_polygon_set_2<Traits>* class-template represents a point set in the plane bounded by x monotone curves. Points in the set lie on the boundary or on the positive side of the curves. This class template provides methods to apply regularized Boolean set-operations and few other utility methods. An *Arrangement_2* data structure is internally used to represent the point set.

The *Traits* template-parameter should be instantiated with a model of the concept *GeneralPolygonSetTraits_2*. The traits class defines the types of points, x -monotone curves, general polygons, and general polygons with holes, that is *Traits::Point_2*, *Traits::X_monotone_curve_2*, *Traits::Polygon_2*, and *Traits::Polygon_with_holes_2* respectively. *Traits::Point_2* must be the type of the endpoints of *Traits::X_monotone_curve_2*, and *Traits::X_monotone_curve_2* must be the type of the curves that comprise the boundaries of the general polygons. The traits class supports geometric operations on the types above. We sometimes use the term *polygon* instead of general polygon for simplicity hereafter.

The input and output of the Boolean set-operations methods consist of one or more general polygons, some of which may have holes. In particular, these methods operate on pairs of objects of type *General_polygon_set_2<Traits>*, or directly on objects of type *Traits::Polygon_2* or *Traits::Polygon_with_holes_2*. An object of type *Traits::Polygon_2* is a valid operand, only if it is strictly simple and its boundary is oriented counterclockwise. An object of type *Traits::Polygon_with_holes_2* is valid, only if its outer boundary is simple and oriented counterclockwise, and each one of its holes is a strictly simple polygon that oriented clockwise, contained inside its outer boundary, and they are all together pairwise disjoint, except perhaps at the vertices.

Types

<i>General_polygon_set_2<Traits>:: Traits_2</i>	the traits class in use.
<i>General_polygon_set_2<Traits>:: Polygon_2</i>	the general polygon type.
<i>General_polygon_set_2<Traits>:: Polygon_with_holes_2</i>	the general polygon with holes type.
<i>General_polygon_set_2<Traits>:: Size</i>	number of polygons with holes size type.
<i>General_polygon_set_2<Traits>:: Arrangement_2</i>	the arrangement type used internally.

Creation

<i>General_polygon_set_2<Traits> gps;</i>	constructs an empty set of polygons represented by an empty arrangement.
<i>General_polygon_set_2<Traits> gps(Self other);</i>	copy constructor.

<i>General_polygon_set_2</i> < <i>Traits</i> > <i>gps</i> (<i>Traits</i> & <i>traits</i>);	constructs an empty set of polygons that uses the given <i>traits</i> instance for performing the geometric operations.
<i>General_polygon_set_2</i> < <i>Traits</i> > <i>gps</i> (<i>Polygon_2</i> <i>pgn</i>);	constructs a set of polygons that consists of the single polygon <i>pgn</i> .
<i>General_polygon_set_2</i> < <i>Traits</i> > <i>gps</i> (<i>Polygon_with_holes_2</i> <i>pgn_with_holes</i>);	constructs a set of polygons that consists of the single polygon with holes <i>pgn_with_holes</i> .

Access Functions

<pre>template <class OutputIterator> OutputIterator gps.polygons_with_holes(OutputIterator out)</pre>		obtains the polygons with holes represented by <i>gps</i> .
<i>Size</i>	<i>gps.number_of_polygons_with_holes()</i>	returns the total number of general polygons represented by <i>gps</i> .
<i>bool</i>	<i>gps.is_empty()</i>	returns <i>true</i> if <i>gps</i> represents an empty set.
<i>bool</i>	<i>gps.is_plane()</i>	returns <i>true</i> if <i>gps</i> represents the entire plane.
<i>Traits</i> &	<i>gps.traits()</i>	obtains an instance of the traits. If the traits was passed as a parameter to the constructor of <i>gps</i> , it is returned. Otherwise, a newly created instance is returned.
<i>Arrangement_2</i>	<i>gps.arrangement()</i>	obtains the arrangement data structure that internally represents the general-polygon set.

Modifiers

<i>void</i>	<i>gps.clear()</i>	clears <i>gps</i> .
<i>void</i>	<i>gps.insert(Polygon_2 & pgn)</i>	<p>inserts <i>pgn</i> into <i>gps</i>.</p> <p><i>Precondition:</i> <i>pgn</i> and the point set represented by <i>gps</i> are disjoint. This precondition enables the use of very efficient insertion methods. Use the respective overloaded method that inserts a polygon of type <i>Polygon_with_holes_2</i>, if only a relaxed precondition can be guaranteed. If even the relaxed precondition cannot be guaranteed, use the <i>join</i> method.</p>

`void gps.insert(Polygon_with_holes_2 & pgn_with_holes)`

inserts *pgn_with_holes* into *gps*.

Precondition: *pgn_with_holes* does not intersect with the point set represented by *gps*, except maybe at the vertices. If this relaxed precondition cannot be guaranteed, use the *join* method.

`template <class InputIterator>`

`void gps.insert(InputIterator begin, InputIterator end)`

inserts the range of polygons (or polygons with holes) into *gps*. (The value type of the input iterator is used to distinguish between the two.)

Precondition: If the given range contains objects of type *Polygon_with_holes_2*, then these polygons and the point set represented by *gps* are pairwise disjoint, except maybe at the vertices. If the given range contains objects of type *Polygon_2*, then these polygons and the point set represented by *gps* are pairwise disjoint without any exceptions.

`template <class InputIterator1, class InputIterator2>`

`void gps.insert(InputIterator1 pgn_begin,
 InputIterator1 pgn_end,
 InputIterator2 pgn_with_holes_begin,
 InputIterator2 pgn_with_holes_end)`

inserts the two ranges of polygons and polygons with holes into *gps*.

Precondition: All polygons in the first range, all polygon with holes in the second range, and the point set represented by *gps* are pairwise disjoint, except maybe at the vertices

`void gps.complement()`

computes the complement of *gps*.

`void gps.complement(Polygon_set_2 other)`

computes the complement of *other*. *gps* is overridden by the result.

Univariate Operations

In the following univariate and bivariate methods the result is placed in *gps* after it is cleared.

`void gps.intersection(General_polygon_set_2 other)`

computes the intersection of *gps* and *other*.

`void gps.intersection(Polygon_2 pgn)`

computes the intersection of *gps* and *pgn*.

`void gps.intersection(Polygon_with_holes_2 pgn)`

computes the intersection of *gps* and *pgn*.

```
template <class InputIterator>
void    gps.intersection( InputIterator begin, InputIterator end)
```

computes the intersection of a collection of point sets. The collection consists of the polygons (or polygons with holes) in the given range and the point set represented by *gps*. (The value type of the input iterator is used to distinguish between the two options.)

```
template <class InputIterator1, class InputIterator2>
void    gps.intersection( InputIterator1 pgn_begin,
                          InputIterator1 pgn_end,
                          InputIterator2 pgn_with_holes_begin,
                          InputIterator2 pgn_with_holes_end)
```

computes the intersection of a collection of point sets. The collection consists of the polygons and polygons with holes in the given two ranges and the point set represented by *gps*.

```
void    gps.join( General_polygon_set_2 other)
void    gps.join( Polygon_2 pgn)
void    gps.join( Polygon_with_holes_2 pgn)
```

computes the union of *gps* and *other*.
computes the union of *gps* and *pgn*.
computes the union of *gps* and *pgn*.

```
template <class InputIterator>
void    gps.join( InputIterator begin, InputIterator end)
```

computes the union of the polygons (or polygons with holes) in the given range and the point set represented by *gps*. (The value type of the input iterator is used to distinguish between the two options.)

```
template <class InputIterator1, class InputIterator2>
void    gps.join( InputIterator1 pgn_begin,
                  InputIterator1 pgn_end,
                  InputIterator2 pgn_with_holes_begin,
                  InputIterator2 pgn_with_holes_end)
```

computes the union of the polygons and polygons with holes in the given two ranges and the point set represented by *gps*.

```
void    gps.difference( General_polygon_set_2 other)
```

computes the difference between *gps* and *other*.

```
void    gps.difference( Polygon_2 pgn)
void    gps.difference( Polygon_with_holes_2 pgn)
```

computes the difference between *gps* and *pgn*.
computes the difference between *gps* and *pgn*.

```
void    gps.symmetric_difference( General_polygon_set_2 other)
```

computes the symmetric difference between *gps* and *other*.

```
void    gps.symmetric_difference( Polygon_2 pgn)
```

computes the symmetric difference between *gps* and *pgn*.


```
void    gps.symmetric_difference( Polygon_with_holes_2 pgn)
```

computes the symmetric difference between *gps* and *pgn*.

```
template <class InputIterator>
```

```
void    gps.symmetric_difference( InputIterator begin, InputIterator end)
```

computes the symmetric difference (xor) of a collection of point sets. The collection consists of the polygons (or polygons with holes) in the given range and the point set represented by *gps*. (The value type of the input iterator is used to distinguish between the two options.)

```
template <class InputIterator1, class InputIterator2>
```

```
void    gps.symmetric_difference( InputIterator1 pgn_begin,
                                InputIterator1 pgn_end,
                                InputIterator2 pgn_with_holes_begin,
                                InputIterator2 pgn_with_holes_end)
```

computes the symmetric difference (xor) of a collection of point sets. The collection consists of the polygons and polygons with holes in the given two ranges and the point set represented by *gps*.

Bivariate Operations

The following bivariate function replace *gps* with the result.

```
void    gps.intersection( General_polygon_set_2 gps1, General_polygon_set_2 gps2)
```

computes the intersection of *gps1* and *gps2*.

```
void    gps.join( General_polygon_set_2 gps1, General_polygon_set_2 gps2)
```

computes the union of *gps1* and *gps2*.

```
void    gps.difference( General_polygon_set_2 gps1, General_polygon_set_2 gps2)
```

computes the difference between *gps1* and *gps2*.

```
void    gps.symmetric_difference( General_polygon_set_2 gps1, General_polygon_set_2 gps2)
```

computes the symmetric difference between *gps1* and *gps2*.

Query Functions

<i>bool</i>	<i>gps.do_intersect(General_polygon_set_2 other)</i>	returns <i>true</i> if <i>gps</i> and <i>other</i> intersect in their interior, and <i>false</i> otherwise.
<i>bool</i>	<i>gps.do_intersect(Polygon_2 pgn)</i>	returns <i>true</i> if <i>gps</i> and <i>pgn</i> intersect in their interior, and <i>false</i> otherwise.
<i>bool</i>	<i>gps.do_intersect(Polygon_with_holes_2 pgn)</i>	returns <i>true</i> if <i>gps</i> and <i>pgn</i> intersect in their interior, and <i>false</i> otherwise.

```
template <class InputIterator>
void      gps.do_intersect( InputIterator begin, InputIterator end)
```

returns *true* if the interior of the point sets in a collection intersect, and *false* otherwise. The collection consists of the polygons (or polygons with holes) in the given range and the point set represented by *gps*. (The value type of the input iterator is used to distinguish between the two options.)

```
template <class InputIterator1, class InputIterator2>
void      gps.do_intersect( InputIterator1 pgn_begin,
                           InputIterator1 pgn_end,
                           InputIterator2 pgn_with_holes_begin,
                           InputIterator2 pgn_with_holes_end)
```

returns *true* if the interior of the point sets in a collection intersect, and *false* otherwise. The collection consists of the polygons and polygons with holes in the given two ranges and the point set represented by *gps*.

<i>bool</i>	<i>gps.locate(Point_2 p, Polygon_with_holes_2 & pgn)</i>
-------------	---------------------------------------------------------------

obtains a polygon with holes that contains the query point *p*, if exists, through *pgn*, and returns *true*. Otherwise, returns *false*.

<i>Oriented_side</i>	<i>gps.oriented_side(Point_2 q)</i>
----------------------	--------------------------------------

returns either the constant *ON_ORIENTED_BOUNDARY*, *ON_POSITIVE_SIDE*, or *ON_NEGATIVE_SIDE*, iff *p* lies on the boundary, properly on the positive side, or properly on the negative side of *gps* respectively.

Oriented_side *gps.oriented_side(General_polygon_set_2 other)*

returns either the constant *ON_NEGATIVE_SIDE*, *ON_ORIENTED_BOUNDARY*, or *ON_POSITIVE_SIDE*, iff *other* and *gps* are completely disjoint, in contact, or intersect in their interior, respectively.

Oriented_side *gps.oriented_side(Polygon_2 pgn)*

returns either the constant *ON_NEGATIVE_SIDE*, *ON_ORIENTED_BOUNDARY*, or *ON_POSITIVE_SIDE*, iff *pgn* and *gps* are completely disjoint, in contact, or intersect in their interior, respectively.

Oriented_side *gps.oriented_side(Polygon_with_holes_2 pgn)*

returns either the constant *ON_NEGATIVE_SIDE*, *ON_ORIENTED_BOUNDARY*, or *ON_POSITIVE_SIDE*, iff *pgn* and *gps* are completely disjoint, in contact, or intersect in their interior, respectively.

Miscellaneous

bool *gps.is_valid()*

returns *true* if *gps* represents a valid point set.

See Also

Arrangement_2(page ??)

ArrangementXMonotoneTraits_2(page [1261](#))

Nef_2(page ??)

GeneralPolygon_2

Types

<i>GeneralPolygon_2:: X_monotone_curve_2</i>	represents a planar (weakly) <i>x</i> -monotone curve. The type of the geometric mapping of the polygonal edges.
<i>GeneralPolygon_2:: Curve_iterator</i>	an iterator over the geometric mapping of the polygon edges. Its value type is <i>X_monotone_curve_2</i> .
<i>GeneralPolygon_2:: Curve_const_iterator</i>	a const iterator over the geometric mapping of the polygon edges. Its value type is <i>X_monotone_curve_2</i> .

Definition

A model of this concept represents a simple general-polygon. The geometric mapping of the edges of the polygon must be *x*-monotone curves. The concept requires the ability to access these curves. The general polygon represented must be simple. That is, the only points of the plane belonging to two curves are the geometric mapping of the polygon vertices. In addition, the vertices of the represented polygon must be ordered consistently, and the curved must be directed accordingly. Only counterclockwise orientated polygons are valid operands of Boolean set-operations. General polygon that represent holes must be clockwise orientated.

Creation

<i>GeneralPolygon_2 polygon;</i>	default constructor.
<i>GeneralPolygon_2 polygon(other);</i>	copy constructor.
<i>GeneralPolygon_2 polygon = other</i>	assignment operator.
<i>template <class InputIterator></i> <i>GeneralPolygon_2 polygon(InputIterator begin, InputIterator end);</i>	constructs a general polygon from a given range of curves.

Access Functions

<i>Curve_iterator</i>	<i>polygon.curves_begin()</i>	returns the begin iterator of the curves.
<i>Curve_iterator</i>	<i>polygon.curves_end()</i>	returns the past-the-end iterator of the curves.
<i>Curve_const_iterator</i>	<i>polygon.curves_begin()</i>	returns the begin const iterator of the curves.
<i>Curve_const_iterator</i>	<i>polygon.curves_end()</i>	returns the past-the-end const iterator of the curves.

Modifiers

template <class Iterator>

void

polygon.init(Iterator begin, Iterator end)

initializes the polygon with the polygonal chain given by the range. The value type of *Iterator* must be *X_monotone_curve_2*.

Precondition: The curves in the range must define a simple polygon.

Has Models

CGAL::General_polygon_2<ArrTraits>

GeneralPolygonWithHoles_2

Types

<i>GeneralPolygonWithHoles_2::General_polygon_2</i>	the general-polygon type used to represent the outer boundary and each hole.
<i>GeneralPolygonWithHoles_2::Hole_const_iterator</i>	a bidirectional iterator over the polygonal holes. Its value type is <i>General_polygon_2</i> .

Definition

A model of this concept represents a general polygon with holes. The concept requires the ability to access the general polygon that represents the outer boundary and the general polygons that represent the holes.

Creation

<i>GeneralPolygonWithHoles_2 polygon;</i>	default constructor.
<i>GeneralPolygonWithHoles_2 polygon(other);</i>	copy constructor.
<i>GeneralPolygonWithHoles_2 polygon = other</i>	assignment operator.

template <class InputIterator>

GeneralPolygonWithHoles_2 polygon(General_polygon_2 & outer, InputIterator begin, InputIterator end);

constructs a general polygon with holes using a given general polygon *outer* as the outer boundary and a given range of holes. If *outer* is an empty general polygon, then an unbounded polygon with holes will be created. The holes must be contained inside the outer boundary, and the polygons representing the holes must be strictly simple and pairwise disjoint, except perhaps at the vertices.

Predicates

<i>bool</i>	<i>polygon.is_unbounded()</i>	returns <i>true</i> if the outer boundary is empty, and <i>false</i> otherwise.
-------------	-------------------------------	---------------------------------------------------------------------------------

Access Functions

<i>General_polygon_2</i>	<i>polygon.outer_boundary()</i>	returns the general polygon that represents the outer boundary.
<i>Hole_const_iterator</i>	<i>polygon.holes_begin()</i>	returns the begin iterator of the holes.
<i>Hole_const_iterator</i>	<i>polygon.holes_end()</i>	returns the past-the-end iterator of the holes.

Has Models

CGAL::General_polygon_with_holes_2<*General_polygon*>

CGAL::Polygon_with_holes_2<*Kernel*,*Container*>

CGAL::Gps_circle_segment_traits_2<*Kernel*>::*Polygon_with_holes_2*

CGAL::Gps_traits_2<*ArrTraits*,*GeneralPolygon*>::*Polygon_with_holes_2*

ArrangementDirectionalXMonotoneTraits_2

Definition

This concept refines the basic arrangement x -monotone traits concept. A model of this concept is able to handle *directed* x -monotone curves that intersect in their interior. Namely, an instance of the *X_monotone_curve_2* type defined by a model of the concept *ArrangementXMonotoneTraits_2* is only required to have a *left* (lexicographically smaller) endpoint and a *right* endpoint. If the traits class is also a model of *ArrangementDirectionalXMonotoneTraits_2*, the x -monotone curve is also required to have a direction, namely one of these two endpoint is viewed as its *source* and the other as its *target*.

Refines

ArrangementXMonotoneTraits_2

Functor Types

ArrangementDirectionalXMonotoneTraits_2:: Compare_endpoints_xy_2

provides the operator :

Comparison_result operator() (X_monotone_curve_2 c)

which accepts an input curve c and compare its source and target point. It returns *SMALLER* if the curve is directed from left to right (lexicographically — i.e., in case of a vertical line segment, this means it is directed upward), and *LARGER* if it is directed from right to left.

ArrangementDirectionalXMonotoneTraits_2:: Construct_opposite_2

provides the operator :

X_monotone_curve_2 operator() (X_monotone_curve_2 c)

which accepts an x -monotone curve c and returns its oppsite curve, namely a curve whose graph is the same as c 's, and whose source and target are swapped with respect to c 's source and target.

In addition, the two following functors, required by the concept *ArrangementXMonotoneTraits_2* should operate as follows:

ArrangementDirectionalXMonotoneTraits_2:: Intersect_2

provides the operator (templated by the *OutputIterator* type) :

OutputIterator operator() (X_monotone_curve_2 c1, X_monotone_curve_2 c2, OutputIterator oi)

which computes the intersections of $c1$ and $c2$ and inserts them *in an ascending lexicographic xy -order* into the output iterator. The value-type of *OutputIterator* is *CGAL::Object*, where each *Object* either wraps a *pair<Point_2,Multiplicity>* instance, which represents an intersection point with its multiplicity (in case the multiplicity is undefined or not known, it should be set to 0) or an *X_monotone_curve_2* instance, representing an overlapping subcurve of $c1$ and $c2$. In the latter case, if $c1$ and $c2$ have the same direction, then the overlapping subcurves should also be directed the same way; otherwise, they can be associated with an arbitrary direction. The operator returns a past-the-end iterator for the output sequence.

ArrangementDirectionalXMonotoneTraits_2:: Split_2

provides the operator :

```
void operator() (X_monotone_curve_2 c, Point_2 p, X_monotone_curve_2& c1, X_monotone_curve_2& c2)
```

which accepts an input curve *c* and a split point *p* in its interior. It splits *c* at the split point into two subcurves *c1* and *c2*, such that *p* is *c1*'s *right* endpoint and *c2*'s *left* endpoint. The direction of *c* should be preserved: in case *c* is directed from left to right then *p* becomes *c1*'s target and *c2*'s source; otherwise, *p* becomes *c2*'s target and *c1*'s source.

Creation

ArrangementDirectionalXMonotoneTraits_2 traits; default constructor.

ArrangementDirectionalXMonotoneTraits_2 traits(other);

copy constructor

ArrangementDirectionalXMonotoneTraits_2

traits = other assignment operator.

Accessing Functor Objects

Compare_endpoints_xy_2 *traits.compare_endpoints_xy_2_object()*

Construct_opposite_2 *traits.construct_opposite_2_object()*

Has Models

CGAL::Arr_segment_traits_2<Kernel>

CGAL::Arr_non_caching_segment_traits_2<Kernel>

CGAL::Arr_circle_segment_traits_2<Kernel>

CGAL::Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>

CGAL::Arr_rational_arc_traits_2<AlgKernel,NtTraits>

See Also

ArrangementXMonotoneTraits_2 (page [1261](#))

GeneralPolygonSetTraits_2

Definition

This concept defines the minimal set of geometric predicates needed to perform the Boolean-set operations. It refines the directional x -monotone arrangement-traits concept. In addition to the *Point_2* and *X_monotone_curve_2* types defined in the generalized concept, it defines a type that represents a general polygon and another one that represents general polygon with holes. It also requires operations that operate on these types.

Refines

ArrangementDirectionalXMonotoneTraits_2

Types

<i>GeneralPolygonSetTraits_2:: Polygon_2</i>	represents a general polygon.
<i>GeneralPolygonSetTraits_2:: Polygon_with_holes_2</i>	represents a general polygon with holes.
<i>GeneralPolygonSetTraits_2:: Curve_const_iterator</i>	A const iterator of curves. Its value type is const <i>X_monotone_curve_2</i> .

Functor Types

<i>GeneralPolygonSetTraits_2:: Construct_polygon_2</i>	a functor that constructs a general polygon from a range of x -monotone curves. It uses the operator <i>void operator() (InputIterator begin, Input iterator end, Polygon_2 & pgn)</i> , parameterized by the <i>InputIterator</i> type.
<i>GeneralPolygonSetTraits_2:: Construct_curves_2</i>	a functor that returns a pair that consists of the begin and past-the-end iterators of the x monotone curves of the boundary of a given general polygon. It uses the operator <i>std::pair<Curve_const_iterator, Curve_const_iterator> operator() (const Polygon_2 & pgn)</i> .

GeneralPolygonSetTraits_2::Is_valid_2

provides the operators :

bool operator() (Polygon_2 & pgn)

which returns *true* if the *pgn* is valid, and *false* otherwise;

and :

bool operator() (Polygon_with_holes_2 & pgn_with_holes)

which returns *true* if *pgn_with_holes* is valid, and *false* otherwise. A polygon of type *Polygon_2* is valid, if it is strictly simple and oriented counterclockwise. A polygon of type *Polygon_with_holes_2* is valid, if its outer boundary is simple and oriented counterclockwise, and each one of its holes is a strictly simple polygon that oriented clockwise, contained inside its outer boundary, and they are all together pairwise disjoint, except perhaps at the vertices.

This functionality is used to verify precondition of some of the operations.

Creation

GeneralPolygonSetTraits_2 traits;

default constructor.

GeneralPolygonSetTraits_2 traits(other);

copy constructor

GeneralPolygonSetTraits_2

traits = other

assignment operator.

Accessing Functor Objects

Construct_polygon_2 traits.construct_polygon_2_object()

returns a functor that constructs a polygon.

Construct_curves_2 traits.construct_curves_2_object()

returns a functor that obtains the curves of a polygon.

Is_valid_2 traits.is_valid_2_object()

returns a functor that checks the validity of a polygon.

Has Models

CGAL::Gps_segment_traits_2<Kernel, Container, ArrSegmentTraits>

CGAL::Gps_circle_segment_traits_2<Kernel>

CGAL::Gps_traits_2<ArrTraits, GeneralPolygon>

See Also

ArrangementDirectionalXMonotoneTraits_2(page 928)

CGAL::General_polygon_2<ArrTraits>

Definition

The class *General_polygon_2<ArrTraits>* models the concept *GeneralPolygon_2*. It represents a simple general-polygon. It is parameterized with the type *ArrTraits* that models the concept *ArrangementDirectionalXMonotoneTraits*. The latter is a refinement of the concept *ArrangementXMonotoneTraits_2*. In addition to the requirements of the concept *ArrangementXMonotoneTraits_2*, a model of the concept *ArrangementDirectionalXMonotoneTraits* must support the following functions:

- Given an *x*-monotone curve, construct its opposite curve.
- Given an *x*-monotone curve, compare its two endpoints lexicographically.

This class supports a few convenient operations in addition to the requirements that the concept *GeneralPolygon_2* lists.

```
#include <CGAL/General_polygon_2.h>
```

Types

General_polygon_2<ArrTraits>::Size number of edges size type.

Creation

Operations

Size *polygon.size()* returns the number of edges of *polygon*.

Modifiers

void *polygon.clear()* clears *polygon*.

void *polygon.reverse_orientation()* reverses the orientation of the polygon.
Precondition: is_simple().

Predicates

bool *polygon.is_empty()* returns *true* if *polygon* is empty, and *false* otherwise.

Orientation *polygon.orientation()* returns the orientation of *polygon*.
Precondition: is_simple().

Is Model for the Concepts

GeneralPolygon_2

CGAL::Polygon_with_holes_2<Kernel,Container>

Definition

The class *Polygon_with_holes_2<Kernel,Container>* models the concept *GeneralPolygonWithHoles_2*. It represents a linear polygon with holes. It is parameterized with two types (*Kernel* and *Container*) that are used to instantiate the type *CGAL::Polygon_2<Kernel,Container>*. The latter is used to represent the outer boundary and the boundary of the holes (if any exist).

```
#include <CGAL/Polygon_with_holes_2.h>
```

Is Model for the Concepts

GeneralPolygonWithHoles_2

CGAL::Polygon_set_2<Kernel,Container>

Definition

The class *Polygon_set_2*<*Kernel*,*Container*> represents sets of linear polygons with holes. It is parameterized with two types (*Kernel* and *Container*) that are used to instantiate the type *CGAL::Polygon_2*<*Kernel*,*Container*>. The latter is used to represent the outer boundary and the boundary of the holes of the set members.

```
#include <CGAL/Polygon_set_2.h>
```

Inherits From

General_polygon_set_2<*Gps_segment_traits_2*<*Kernel*,*Container*> >

See Also

General_polygon_set_2(page [917](#))

Gps_segment_traits_2(page [936](#))

CGAL::General_polygon_with_holes_2<Polygon>

Definition

The class *General_polygon_with_holes_2<Polygon>* models the concept *GeneralPolygonWithHoles_2*. It represents a general polygon with holes. It is parameterized with a type *Polygon* used to define the exposed type *General_polygon_2*. This type represents the outer boundary of the general polygon and the outer boundaries of each hole.

```
#include <CGAL/General_polygon_with_holes_2.h>
```

```
typedef Polygon          General_polygon_2;
```

Is Model for the Concepts

GeneralPolygonWithHoles_2

CGAL::Gps_segment_traits_2<Kernel,Container,ArrSegmentTraits>

Definition

The traits class *Gps_segment_traits_2<Kernel,Container,ArrSegmentTraits>* models the concept *GeneralPolygonSetTraits_2*. It enables Boolean set-operations on (linear) polygons. It defines the exposed type *Polygon_2* to be *CGAL::Polygon_2<Kernel,Container>*. By default, the template parameter *Container* is instantiated by *std::vector<Kernel::Point_2>* and the template parameter *ArrSegmentTraits* is instantiated by *CGAL::Arr_segment_traits_2<Kernel>*.

```
#include <CGAL/Gps_segment_traits_2.h>
```

```
typedef CGAL::Polygon_2<Kernel,Container>
```

```
Polygon_2;
```

Is Model for the Concepts

GeneralPolygonSetTraits_2

See Also

CGAL::Arr_segment_traits_2<Kernel>(page ??)

CGAL::Gps_circle_segment_traits_2<Kernel>

Definition

The traits class *Gps_circle_segment_traits_2<Kernel>* models the *GeneralPolygonSetTraits_2* concept. It enables Boolean set-operations on general polygons bounded by linear segments or circular arcs. It should be parameterized with a kernel.

#include <CGAL/Gps_circle_segment_traits_2.h>

Is Model for the Concepts

GeneralPolygonSetTraits_2

See Also

CGAL::Arr_circle_segment_traits_2<Kernel>(page ??)

CGAL::Gps_traits_2<ArrTraits,GeneralPolygon>

Definition

The traits class *Gps_traits_2<ArrTraits,GeneralPolygon>* models the concept *GeneralPolygonSetTraits_2*. It inherits from the instantiated type of the template parameter *ArrTraits*, which must model the concept *ArrangementDirectionalXMonotoneTraits*, (which in turn refines the concept *ArrangementXMonotoneTraits*). The template parameter *GeneralPolygon* must be instantiated with a model of the concept of *GeneralPolygon_2*. By default, the latter is instantiated by *CGAL::General_polygon_2<ArrTraits>*.

```
#include <CGAL/Gps_traits_2.h>
```

Is Model for the Concepts

GeneralPolygonSetTraits_2

CGAL::complement

Definition

```
#include <CGAL/Boolean_set_operations_2.h>
```

```
void                complement( Type pgn, Type & res)
```

Each one of these functions computes the complement of a given polygon *pgn*, and stores the resulting polygon with holes in *res*.

Arg Type
<i>Polygon_2</i>
<i>General_polygon_2</i>

```
OutputIterator      complement( Type pgn, OutputIterator oi)
```

Each one of these functions computes the complement of a given polygon *pgn*, inserts the resulting polygons with holes into an output container through a given output iterator *oi*, and returns the output iterator. The value type of the *OutputIterator* is either *Polygon_with_holes_2* or *General_polygon_with_holes_2*.

Arg Type
<i>Polygon_with_holes_2</i>
<i>General_polygon_with_holes_2</i>

```
template <class Kernel, class Container>
void                complement( Polygon_2<Kernel, Container> pgn,
                               Polygon_with_holes_2<Kernel, Container> & res)
```

```
template <class Traits>
void                complement( General_polygon_2<Traits> pgn,
                               General_polygon_with_holes_2<Traits> & res)
```

```
template <class Traits, class OutputIterator>
OutputIterator      complement( Polygon_with_holes_2<Kernel, Container> pgn, OutputIterator oi)
template <class Traits, class OutputIterator>
OutputIterator      complement( General_polygon_with_holes_2<General_polygon_2<Traits> > pgn,
                               OutputIterator oi)
```

See Also

CGAL::do_intersect page [943](#)
CGAL::intersection page [945](#)
CGAL::join page [947](#)

<i>CGAL::difference</i>	page 941
<i>CGAL::symmetric_difference</i>	page 949

CGAL::difference

Definition

```
#include <CGAL/Boolean_set_operations_2.h>
```

```
OutputIterator    difference( Type1 p1, Type2 p2, OutputIterator oi)
```

Each one of these functions computes the difference between two given polygons $p1$ and $p2$, and inserts the resulting polygons with holes into an output container through the output iterator oi . The value type of the *OutputIterator* is either *Polygon_with_holes_2* or *General_polygon_with_holes_2*.

Arg 1 Type	Arg 2 Type
<i>Polygon_2</i>	<i>Polygon_2</i>
<i>Polygon_2</i>	<i>Polygon_with_holes_2</i>
<i>Polygon_with_holes_2</i>	<i>Polygon_2</i>
<i>Polygon_with_holes_2</i>	<i>Polygon_with_holes_2</i>
<i>General_polygon_2</i>	<i>General_polygon_2</i>
<i>General_polygon_2</i>	<i>General_polygon_with_holes_2</i>
<i>General_polygon_with_holes_2</i>	<i>General_polygon_2</i>
<i>General_polygon_with_holes_2</i>	<i>General_polygon_with_holes_2</i>

```
template <class Kernel, class Container, class OutputIterator>
OutputIterator    difference( Polygon_2<Kernel, Container> p1,
                             Polygon_2<Kernel, Container> p2,
                             OutputIterator oi)
```

```
template <class Kernel, class Container, class OutputIterator>
OutputIterator    difference( Polygon_2<Kernel, Container> p1,
                             Polygon_with_holes_2<Kernel, Container> p2,
                             OutputIterator oi)
```

```
template <class Kernel, class Container, class OutputIterator>
OutputIterator    difference( Polygon_with_holes_2<Kernel, Container> p1,
                             Polygon_2<Kernel, Container> p2,
                             OutputIterator oi)
```

```
template <class Kernel, class Container, class OutputIterator>
OutputIterator    difference( Polygon_with_holes_2<Kernel, Container> p1,
                             Polygon_with_holes_2<Kernel, Container> p2,
                             OutputIterator oi)
```

```
template <class Traits, class OutputIterator>
OutputIterator    difference( General_polygon_2<Traits> p1,
                             General_polygon_2<Traits> p2,
                             OutputIterator oi)
```

```
template <class Traits, class OutputIterator>
OutputIterator    difference( General_polygon_with_holes_2<General_polygon_2<Traits> > p1,
                             General_polygon_2<Traits> p2,
                             OutputIterator oi)
```

```

template <class Traits, class OutputIterator>
OutputIterator    difference( General_polygon_2<Traits> p1,
                             General_polygon_with_holes_2<General_polygon_2<Traits> > p2,
                             OutputIterator oi)

template <class Polygon, class OutputIterator>
OutputIterator    difference( General_polygon_with_holes_2<Polygon> p1,
                             General_polygon_with_holes_2<Polygon> p2)

```

See Also

CGAL::do_intersect page [943](#)
CGAL::intersection page [945](#)
CGAL::join page [947](#)
CGAL::symmetric_difference page [949](#)

CGAL::do_intersect

Definition

```
#include <CGAL/Boolean_set_operations_2.h>
```

```
bool do_intersect( Type1 p1, Type2 p2)
```

Each one of these functions return *true*, if the two given polygons *p1* and *p2* intersect in their interior, and *false* otherwise.

Arg 1 Type	Arg 2 Type
<i>Polygon_2</i>	<i>Polygon_2</i>
<i>Polygon_2</i>	<i>Polygon_with_holes_2</i>
<i>Polygon_with_holes_2</i>	<i>Polygon_2</i>
<i>Polygon_with_holes_2</i>	<i>Polygon_with_holes_2</i>
<i>General_polygon_2</i>	<i>General_polygon_2</i>
<i>General_polygon_2</i>	<i>General_polygon_with_holes_2</i>
<i>General_polygon_with_holes_2</i>	<i>General_polygon_2</i>
<i>General_polygon_with_holes_2</i>	<i>General_polygon_with_holes_2</i>

```
template <class Kernel, class Container>
bool do_intersect( Polygon_2<Kernel, Container> p1, Polygon_2<Kernel, Container> p2)
template <class Kernel, class Container>
bool do_intersect( Polygon_2<Kernel, Container> p1, Polygon_with_holes_2<Kernel, Container> p2)
template <class Kernel, class Container>
bool do_intersect( Polygon_with_holes_2<Kernel, Container> p1, Polygon_2<Kernel, Container> p2)
template <class Kernel, class Container>
bool do_intersect( Polygon_with_holes_2<Kernel, Container> p1,
                  Polygon_with_holes_2<Kernel, Container> p2)
```

```
template <class Traits>
bool do_intersect( General_polygon_2<Traits> p1, General_polygon_2<Traits> p2)
template <class Traits>
bool do_intersect( General_polygon_2<Traits> p1,
                  General_polygon_with_holes_2<General_polygon_2<Traits> > p2)
```

```
template <class Traits>
bool do_intersect( General_polygon_with_holes_2<General_polygon_2<Traits> > p1,
                  General_polygon_2<Traits> p2)
```

```
template <class Polygon>
bool do_intersect( General_polygon_with_holes_2<Polygon> p1,
                  General_polygon_with_holes_2<Polygon> p2)
```

```
template <class InputIterator>
bool do_intersect( InputIterator begin, InputIterator end)
```

Returns *true*, if the set of general polygons (or general polygons with holes) in the given range intersect in their interior, and *false* otherwise. (The value type of the input iterator is used to distinguish between the two).

```
template <class InputIterator1, class InputIterator2>
bool    do_intersect( InputIterator1 pgn_begin1,
                     InputIterator1 pgn_end1,
                     InputIterator2 pgn_begin2,
                     InputIterator2 pgn_end2)
```

Returns *true*, if the set of general polygons and general polygons with holes in the given two ranges respectively intersect in their interior, and *false* otherwise.

See Also

CGAL::intersection page [945](#)
CGAL::join page [947](#)
CGAL::difference page [941](#)
CGAL::symmetric_difference page [949](#)

CGAL::intersection

Definition

```
#include <CGAL/Boolean_set_operations_2.h>
```

```
OutputIterator intersection( Type1 p1, Type2 p2, OutputIterator oi)
```

Each one of these functions computes the intersection of two given polygons $p1$ and $p2$, inserts the resulting polygons with holes into an output container through a given output iterator oi , and returns the output iterator. The value type of the *OutputIterator* is either *Polygon_with_holes_2* or *General_polygon_with_holes_2*.

Arg 1 Type	Arg 2 Type
<i>Polygon_2</i>	<i>Polygon_2</i>
<i>Polygon_2</i>	<i>Polygon_with_holes_2</i>
<i>Polygon_with_holes_2</i>	<i>Polygon_2</i>
<i>Polygon_with_holes_2</i>	<i>Polygon_with_holes_2</i>
<i>General_polygon_2</i>	<i>General_polygon_2</i>
<i>General_polygon_2</i>	<i>General_polygon_with_holes_2</i>
<i>General_polygon_with_holes_2</i>	<i>General_polygon_2</i>
<i>General_polygon_with_holes_2</i>	<i>General_polygon_with_holes_2</i>

```
template <class Kernel, class Container, class OutputIterator>
OutputIterator intersection( Polygon_2<Kernel, Container> p1,
                           Polygon_2<Kernel, Container> p2,
                           OutputIterator oi)
```

```
template <class Kernel, class Container, class OutputIterator>
OutputIterator intersection( Polygon_2<Kernel, Container> p1,
                           Polygon_with_holes_2<Kernel, Container> p2,
                           OutputIterator oi)
```

```
template <class Kernel, class Container, class OutputIterator>
OutputIterator intersection( Polygon_with_holes_2<Kernel, Container> p1,
                           Polygon_2<Kernel, Container> p2,
                           OutputIterator oi)
```

```
template <class Kernel, class Container, class OutputIterator>
OutputIterator intersection( Polygon_with_holes_2<Kernel, Container> p1,
                           Polygon_with_holes_2<Kernel, Container> p2,
                           OutputIterator oi)
```

```
template <class Traits, class OutputIterator>
OutputIterator intersection( General_polygon_2<Traits> p1,
                           General_polygon_2<Traits> p2,
                           OutputIterator oi)
```

```
template <class Traits, class OutputIterator>
OutputIterator intersection( General_polygon_with_holes_2<General_polygon_2<Traits> > p1,
                           General_polygon_2<Traits> p2,
                           OutputIterator oi)
```

```
template <class Traits, class OutputIterator>
OutputIterator    intersection( General_polygon_2<Traits> p1,
                               General_polygon_with_holes_2<General_polygon_2<Traits> > p2,
                               OutputIterator oi)
```

```
template <class Polygon, class OutputIterator>
OutputIterator    intersection( General_polygon_with_holes_2<Polygon> p1,
                               General_polygon_with_holes_2<Polygon> p2,
                               OutputIterator oi)
```

```
template <class InputIterator, class OutputIterator>
OutputIterator    intersection( InputIterator begin, InputIterator end, OutputIterator oi)
```

Computes the intersection of the general polygons (or general polygons with holes) in the given range. (The value type of the input iterator is used to distinguish between the two.) The result, represented by a set of general polygon with holes, is inserted into an output container through a given output iterator *oi*. The output iterator is returned. The value type of the *OutputIterator* is *Traits::Polygon_with_holes_2*.

```
template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator    intersection( InputIterator1 pgn_begin1,
                               InputIterator1 pgn_end1,
                               InputIterator2 pgn_begin2,
                               InputIterator2 pgn_end2,
                               OutputIterator oi)
```

Computes the intersection of the general polygons and general polygons with holes in the given two ranges. The result, represented by a set of general polygon with holes, is inserted into an output container through a given output iterator *oi*. The output iterator is returned. The value type of the *OutputIterator* is *Traits::Polygon_with_holes_2*.

See Also

CGAL::do_intersect page [943](#)
CGAL::join page [947](#)
CGAL::difference page [941](#)
CGAL::symmetric_difference page [949](#)

CGAL::join

Definition

```
#include <CGAL/Boolean_set_operations_2.h>
```

```
bool join( Type1 p1, Type2 p2, General_polygon_with_holes_2 & p)
```

Each one of these functions computes the union of two given polygons *p1* and *p2*. If the two given polygons overlap, it returns *true*, and places the resulting polygon in *p*. Otherwise, it returns *false*.

Arg 1 Type	Arg 2 Type
<i>Polygon_2</i>	<i>Polygon_2</i>
<i>Polygon_2</i>	<i>Polygon_with_holes_2</i>
<i>Polygon_with_holes_2</i>	<i>Polygon_2</i>
<i>Polygon_with_holes_2</i>	<i>Polygon_with_holes_2</i>
<i>General_polygon_2</i>	<i>General_polygon_2</i>
<i>General_polygon_2</i>	<i>General_polygon_with_holes_2</i>
<i>General_polygon_with_holes_2</i>	<i>General_polygon_2</i>
<i>General_polygon_with_holes_2</i>	<i>General_polygon_with_holes_2</i>

```
template <class Kernel, class Container>
bool join( Polygon_2<Kernel, Container> p1,
           Polygon_2<Kernel, Container> p2,
           General_polygon_with_holes_2<Polygon_2<Kernel, Container> > & p)
```

```
template <class Kernel, class Container>
bool join( Polygon_2<Kernel, Container> p1,
           Polygon_with_holes_2<Kernel, Container> p2,
           General_polygon_with_holes_2<Polygon_2<Kernel, Container> > & p)
```

```
template <class Kernel, class Container>
bool join( Polygon_with_holes_2<Kernel, Container> p2,
           Polygon_2<Kernel, Container> p1,
           General_polygon_with_holes_2<Polygon_2<Kernel, Container> > & p)
```

```
template <class Kernel, class Container>
bool join( Polygon_with_holes_2<Kernel, Container> p2,
           Polygon_with_holes_2<Kernel, Container> p1,
           General_polygon_with_holes_2<Polygon_2<Kernel, Container> > & p)
```

```
template <class Traits>
bool join( General_polygon_2<Traits> p1,
           General_polygon_2<Traits> p2,
           General_polygon_with_holes_2<General_polygon_2<Traits> > & p)
```

```
template <class Traits>
bool join( General_polygon_2<Traits> p1,
           General_polygon_with_holes_2<General_polygon_2<Traits> > p2,
           General_polygon_with_holes_2<General_polygon_2<Traits> > & p)
```

```
template <class Traits>
bool      join( General_polygon_with_holes_2<General_polygon_2<Traits> > p2,
                General_polygon_2<Traits> p1,
                General_polygon_with_holes_2<General_polygon_2<Traits> > & p)
```

```
template <class Polygon>
bool      join( General_polygon_with_holes_2<Polygon> p1,
                General_polygon_with_holes_2<Polygon> p2,
                Traits::Polygon_with_holes_2 & p)
```

```
template <class InputIterator, class OutputIterator>
OutputIterator  join( InputIterator begin, InputIterator end, OutputIterator oi)
```

Computes the union of the general polygons (or general polygons with holes) in the given range. (The value type of the input iterator is used to distinguish between the two.) The result, represented by a set of general polygon with holes, is inserted into an output container through a given output iterator *oi*. The output iterator is returned. The value type of the *OutputIterator* is *Traits::Polygon_with_holes_2*.

```
template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator  join( InputIterator1 pgn_begin1,
                    InputIterator1 pgn_end1,
                    InputIterator2 pgn_begin2,
                    InputIterator2 pgn_end2,
                    OutputIterator oi)
```

Computes the union of the general polygons and general polygons with holes in the given two ranges. The result, represented by a set of general polygon with holes, is inserted into an output container through a given output iterator *oi*. The output iterator is returned. The value type of the *OutputIterator* is *Traits::Polygon_with_holes_2*.

See Also

CGAL::do_intersect page [943](#)
CGAL::intersection page [945](#)
CGAL::difference page [941](#)
CGAL::symmetric_difference page [949](#)

CGAL::symmetric_difference

Definition

```
#include <CGAL/Boolean_set_operations_2.h>
```

```
OutputIterator    intersection( Type1 p1, Type2 p2, OutputIterator oi)
```

Each one of these functions computes the symmetric difference between two given polygons *p1* and *p2*, and inserts the resulting polygons with holes into an output container through the output iterator *oi*. The value type of the *OutputIterator* is either *Polygon_with_holes_2* or *General_polygon_with_holes_2*.

Arg 1 Type	Arg 2 Type
<i>Polygon_2</i>	<i>Polygon_2</i>
<i>Polygon_2</i>	<i>Polygon_with_holes_2</i>
<i>Polygon_with_holes_2</i>	<i>Polygon_2</i>
<i>Polygon_with_holes_2</i>	<i>Polygon_with_holes_2</i>
<i>General_polygon_2</i>	<i>General_polygon_2</i>
<i>General_polygon_2</i>	<i>General_polygon_with_holes_2</i>
<i>General_polygon_with_holes_2</i>	<i>General_polygon_2</i>
<i>General_polygon_with_holes_2</i>	<i>General_polygon_with_holes_2</i>

```
template <class Kernel, class Container, class OutputIterator>
OutputIterator    symmetric_difference( Polygon_2<Kernel, Container> p1,
                                       Polygon_2<Kernel, Container> p2,
                                       OutputIterator oi)
```

```
template <class Kernel, class Container, class OutputIterator>
OutputIterator    symmetric_difference( Polygon_2<Kernel, Container> p1,
                                       Polygon_with_holes_2<Kernel, Container> p2,
                                       OutputIterator oi)
```

```
template <class Kernel, class Container, class OutputIterator>
OutputIterator    symmetric_difference( Polygon_with_holes_2<Kernel, Container> p1,
                                       Polygon_2<Kernel, Container> p2,
                                       OutputIterator oi)
```

```
template <class Kernel, class Container, class OutputIterator>
OutputIterator    symmetric_difference( Polygon_with_holes_2<Kernel, Container> p1,
                                       Polygon_with_holes_2<Kernel, Container> p2,
                                       OutputIterator oi)
```

```
template <class Traits, class OutputIterator>
OutputIterator    symmetric_difference( General_polygon_2<Traits> p1,
                                       General_polygon_2<Traits> p2,
                                       OutputIterator oi)
```

```
template <class Traits, class OutputIterator>
OutputIterator    symmetric_difference( General_polygon_with_holes_2<General_polygon_2<Traits> > p1,
                                       General_polygon_2<Traits> p2,
                                       OutputIterator oi)
```

```
template <class Traits, class OutputIterator>
OutputIterator    symmetric_difference( General_polygon_2<Traits> p1,
                                     General_polygon_with_holes_2<General_polygon_2<Traits> > p2,
                                     OutputIterator oi)
```

```
template <class Polygon, class OutputIterator>
OutputIterator    symmetric_difference( General_polygon_with_holes_2<Polygon> p1,
                                     General_polygon_with_holes_2<Polygon> p2,
                                     OutputIterator oi)
```

```
template <class InputIterator, class OutputIterator>
OutputIterator    symmetric_difference( InputIterator begin, InputIterator end, OutputIterator oi)
```

Computes the symmetric difference of the general polygons (or general polygons with holes) in the given range. A point is contained in the symmetric difference, if and only if it is contained in odd number of input polygons. (The value type of the input iterator is used to distinguish between the two.) The result, represented by a set of general polygon with holes, is inserted into an output container through a given output iterator *oi*. The output iterator is returned. The value type of the *OutputIterator* is *Traits::Polygon_with_holes_2*.

```
template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator    symmetric_difference( InputIterator1 pgn_begin1,
                                     InputIterator1 pgn_end1,
                                     InputIterator2 pgn_begin2,
                                     InputIterator2 pgn_end2,
                                     OutputIterator oi)
```

Computes the union of the general polygons and general polygons with holes in the given two ranges. A point is contained in the symmetric difference, if and only if it is contained in odd number of input polygons. The result, represented by a set of general polygon with holes, is inserted into an output container through a given output iterator *oi*. The output iterator is returned. The value type of the *OutputIterator* is *Traits::Polygon_with_holes_2*.

See Also

CGAL::do_intersect page [943](#)
CGAL::intersection page [945](#)
CGAL::join page [947](#)
CGAL::difference page [941](#)

CGAL::operator<<

Definition

This operator exports a polygon with holes, a general polygon, or a general polygon with holes P to the output stream *out*. The output is in ASCII format.

An ASCII and a binary format exist. The format can be selected with the CGAL modifiers for streams, *set_ascii_mode* and *set_binary_mode* respectively. The modifier *set_pretty_mode* can be used to allow for (a few) structuring comments in the output. Otherwise, the output would be free of comments. The default for writing is ASCII without comments.

```
#include <CGAL/Polygon_with_holes_2.h>          template <class Kernel, class Container>
ostream& out << CGAL::Polygon_with_holes_2<Kernel, Container> P
```

The number of points of the outer boundary is exported followed by the points themselves in counterclockwise order. Then, the number of holes is exported, and for each hole, the number of points on its outer boundary is exported followed by the points themselves in clockwise order.

```
#include <CGAL/General_polygon_2.h>             template <class ArrTraits>
ostream& out << CGAL::General_polygon_2<ArrTraits> P
```

The number of curves of the outer boundary is exported followed by the curves themselves in counterclockwise order.

```
#include <CGAL/General_polygon_with_holes_2.h>   template <class Polygon>
ostream& out << CGAL::General_polygon_with_holes_2<Polygon> P
```

The number of curves of the outer boundary is exported followed by the curves themselves in counterclockwise order. Then, the number of holes is exported, and for each hole, the number of curves on its outer boundary is exported followed by the curves themselves in clockwise order.

See Also

CGAL::Polygon_2<Kernel, Container> page ??
 CGAL::General_polygon_2<ArrTraits> page [932](#)
 CGAL::General_polygon_with_holes_2<Polygon> page [935](#)
 operator>> page [952](#)

CGAL::operator>>

Definition

This operator imports a polygon with holes, a general polygon, or a general polygon with holes from the input stream *in*.

An ASCII and a binary format exist. The stream detects the format automatically and can read both.

```
#include <CGAL/Polygon_with_holes_2.h>          template <class Kernel, Class Container>
istream&                                         istream& in >> CGAL::Polygon_with_holes_2<Kernel, Container>& P
```

The format consists of the number of points of the outer boundary followed by the points themselves in counter-clockwise order, followed by the number of holes, and for each hole, the number of points of the outer boundary is followed by the points themselves in clockwise order.

```
#include <CGAL/General_polygon_2.h>             template <class ArrTraits>
istream&                                         istream& in >> CGAL::General_polygon_2<ArrTraits>& P
```

The format consists of the number of curves of the outer boundary followed by the curves themselves in counterclockwise order.

```
#include <CGAL/General_polygon_with_holes_2.h>  template <class Polygon>
istream&                                         istream& in >> CGAL::General_polygon_with_holes_2<Polygon>& P
```

The format consists of the number of curves of the outer boundary followed by the curves themselves in counter-clockwise order, followed by the number of holes, and for each hole, the number of curves on its outer boundary is followed by the curves themselves in clockwise order.

See Also

CGAL::Polygon_2<Kernel,Container> page ??
 CGAL::General_polygon_2<ArrTraits> page [932](#)
 CGAL::General_polygon_with_holes_2<Polygon> page [935](#)
 operator<< page [951](#)

Chapter 13

2D Boolean Operations on Nef Polygons

Michael Seel

Contents

13.1 Introduction	953
13.2 Construction and Composition	954
13.3 Exploration	955
13.4 Traits Classes	956
13.5 Implementation	956

13.1 Introduction

When working with polygonal and polyhedral sets, the mathematical model determines the kind of point set that can be represented. Nef polyhedra are the most general rectilinear polyhedral model.

Topological simpler models that are contained in the domain of Nef polyhedra are:

- *convex polytopes* normally defined as the convex hull of a nonempty finite set of points. Convex polytopes are compact closed and manifold sets.
- *elementary polyhedra* normally defined as the union of a finite number of convex polytopes.
- *polyhedral sets* normally defined as the intersection of a finite number of closed halfspaces. Such sets are closed and convex but need not to be compact.
- *linear polyhedra* normally defined as the set of all points belonging to the simplices of a *simplicial complex*.

A planar *Nef polyhedron* is any set that can be obtained from a finite set of open halfspaces by set complement and set intersection operations. Due to the fact that all other binary set operations like union, difference and symmetric difference can be reduced to intersection and complement calculations, Nef polyhedra are also closed under those operations. Apart from the set complement operation there are more topological unary set operations that are closed in the domain of Nef polyhedra. Given a Nef polyhedron one can determine its interior, its boundary, and its closure, and also composed operations like regularization (defined to be the closure of the interior or a point set).

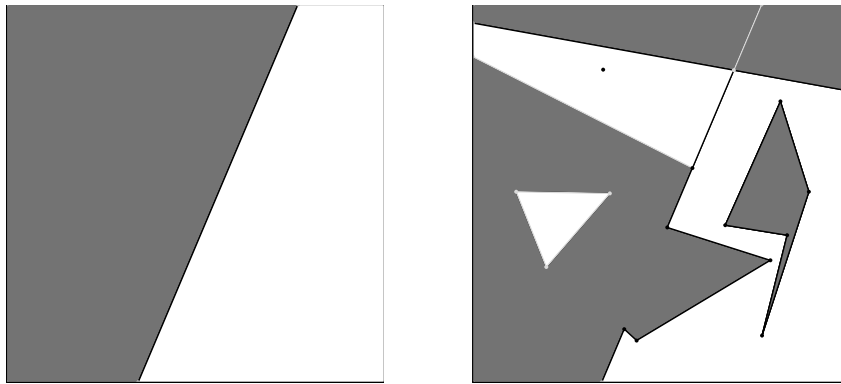


Figure 13.1: Two Nef polyhedra in the plane. A closed halfspace on the left and a complex polyhedron on the right. Note that the points on the squared boundary are at infinity.

13.2 Construction and Composition

Following the above definition, the data type *Nef_polyhedron_2<T>* allows construction of elementary Nef polyhedra and the binary and unary composition by the mentioned set operations.

In the following examples skip the typedefs at the beginning at first and take the types *Point* and *Line* to be models of the standard two-dimensional CGAL kernel (*CGAL::Point_2<K>* and *CGAL::Line_2<K>*). Their user interface is thus defined in the corresponding reference pages.

```
// file : examples/Nef_2/construction.C

#include <CGAL/Gmpz.h>
#include <CGAL/Filtered_extended_homogeneous.h>
#include <CGAL/Nef_polyhedron_2.h>

typedef CGAL::Gmpz RT;
typedef CGAL::Filtered_extended_homogeneous<RT> Extended_kernel;
typedef CGAL::Nef_polyhedron_2<Extended_kernel> Nef_polyhedron;
typedef Nef_polyhedron::Point Point;
typedef Nef_polyhedron::Line Line;

int main() {

    Nef_polyhedron N1(Nef_polyhedron::COMPLETE);

    Line l(2,4,2); // l : 2x + 4y + 2 = 0
    Nef_polyhedron N2(l,Nef_polyhedron::INCLUDED);
    Nef_polyhedron N3 = N2.complement();
    CGAL_assertion(N1 == N2.join(N3));

    Point p1(0,0), p2(10,10), p3(-20,15);
    Point triangle[3] = { p1, p2, p3 };
    Nef_polyhedron N4(triangle, triangle+3);
    Nef_polyhedron N5 = N2.intersection(N4);
    CGAL_assertion(N5 <= N2 && N5 <= N4);
```

```

    return 0;
}

```

Planar halfspaces (as used in the definition) are modelled by oriented lines. In the previous example $N1$ is the Nef polyhedron representing the full plane, $N2$ is the closed halfspace left of the oriented line with equation $2x + 4y + 2 = 0$ including the line, $N3$ is the complement of $N2$ and therefore it must hold that $N2 \cup N3 = N1$.

Additionally one can construct Nef polyhedra from iterator ranges that hold simple polygonal chains. In the example $N4$ is the triangle spanned by the vertices $(0,0)$, $(10,10)$, $(-20,15)$. Note that the construction from a simple polygonal chain has several cases and preconditions that are described in the reference manual page of *Nef_polyhedron_2<T>*. The *operator<=* in the last assertion is a subset-or-equal comparison of two polyhedra.

Nef polyhedra have input and output operators that allows one to output them via streams and read them from streams. Graphical output is currently possible to a *CGAL::Window_stream*. The output operation is defined in *CGAL/IO/Nef_polyhedron_2_Window_stream.h*. For an elaborate example see the demo programs in the directory *demo/Nef_2*.

13.3 Exploration

By recursively composing binary and unary operations one can end with a very complex rectilinear structure. To explore that structure there is a data type *Nef_polyhedron_2<T>::Explorer* that allows read-only exploration of the rectilinear structure. To understand its usability we need more knowledge about the representation of Nef polyhedra.

The rectilinear structure underlying a Nef polyhedron is stored in a selective plane map. Plane map here means a straightline embedded bidirected graph with face objects such that each point in the plane can be uniquely assigned to an object (vertex, edge, face) of the planar subdivision defined by the graph. Selective means that each object (vertex, edge, face) has a Boolean value associated with it to indicate set inclusion or exclusion.

The plane map is defined by the interface data type *Nef_polyhedron_2<T>::Topological_explorer*. Embedding the vertices by standard affine points does not suffice to model the unboundedness of halfspaces and ray-like structures. Therefore the planar subdivision is bounded symbolically by an axis-parallel square box of infimaximal size centered at the origin of our coordinate system. All structures extending to infinity are pruned by the box. Lines and rays have symbolic endpoints on the box. Faces are circularly closed. Infimaximal here means that its geometric extend is always large enough (but finite for our intuition). Assume you approach the box with an affine point, then this point is always inside the box. The same holds for straight lines; they always intersect the box. There are more accurate notions of “large enough”, but the previous propositions are enough at this point. Due to the fact that the infimaximal box is included in the plane map, the vertices and edges are partitioned with respect to this box.

Vertices inside the box are called standard vertices and they are embedded by affine points of type *Explorer::Point*. Vertices on the box are called non-standard vertices and they get their embedding where a ray intersects the box (their embedding is defined by an object of type *Explorer::Ray*). By their straightline embedding, edges represent either segments, rays, lines, or box segments depending on the character of their source and target vertices.

During exploration, box objects can be tracked down by the interface of *Nef_polyhedron_2<T>::Explorer* that is derived from *Nef_polyhedron_2<T>::Topological_explorer* and adds just the box exploration functionality to the interface of the latter. In the following code fragment we iterate over all vertices of a Nef polyhedron and check whether their embedding is an affine point or a point on the infimaximal frame.

```

typedef Nef_polyhedron::Explorer Explorer;
Explorer E = N4.explorer();
Explorer::Vertex_const_iterator v;
for (v = E.vertices_begin(); v != E.vertices_end(); ++v)
    if ( E.is_standard(v) )
        Explorer::Point p = E.point(v) // affine embedding of v
    else /* non-standard */
        Explorer::Ray r = E.ray(v) // extended embedding of v

```

Note that box edges only serve as boundary edges (combinatorically) to close the faces that extend to infinity (geometrically). Their status can be queried by the following operation:

```

typedef Nef_polyhedron::Explorer Explorer;
Explorer E = N4.explorer();
Explorer::Halfedge_const_iterator e;
for (e = E.halfedges_begin(); e != E.halfedges_end(); ++e)
    if ( E.is_frame_edge(e) ) // e is part of square box.

```

13.4 Traits Classes

Now finally we clarify what the template parameter of class *Nef_polyhedron_2*<*T*> actually models. *T* carries the implementation of a so-called extended geometric kernel.

Currently there are three kernel models: *CGAL::Extended_cartesian*<*FT*>, *CGAL::Extended_homogeneous*<*RT*>, and *CGAL::Filtered_extended_homogeneous*<*RT*>. The latter is the most optimized one. The former two are simpler versions corresponding to the simple planar affine kernels. Actually, it holds that (type equality in pseudo-code notation):

```

CGAL::Nef_polyhedron_2< CGAL::Extended_cartesian<FT> >::Point
== CGAL::Cartesian<FT>::Point_2

CGAL::Nef_polyhedron_2< CGAL::Extended_homogeneous<RT> >::Point
== CGAL::Homogeneous<RT>::Point_2

CGAL::Nef_polyhedron_2< CGAL::Filtered_extended_homogeneous<RT> >::Point
== CGAL::Homogeneous<RT>::Point_2

```

Similar equations hold for the types *Line* and *Direction* in the local scope of *Nef_polyhedron_2*<...>.

— advanced —

For its notions and requirements see the description of the concept *ExtendedKernelTraits_2* in the reference manual.

— advanced —

13.5 Implementation

The underlying set operations are realized by an efficient and complete algorithm for the overlay of two plane maps. The algorithm is efficient in the sense that its running time is bounded by the size of the inputs plus the

size of the output times a logarithmic factor. The algorithm is complete in the sense that it can handle all inputs and requires no general position assumption.

2D Boolean Operations on Nef Polygons

Reference Manual

Michael Seel

Planar Nef Polyhedra are whatever can be constructed with a finite number of Boolean operations on halfspaces. See the user manual pages for a detailed introduction.

13.6 Classified Reference Pages

Concepts

ExtendedKernelTraits_2 page [972](#)

Classes

CGAL::Extended_cartesian<FT> page [977](#)
CGAL::Extended_homogeneous<RT> page [978](#)
CGAL::Filtered_extended_homogeneous<RT> page [979](#)
CGAL::Nef_polyhedron_2<T> page [960](#)
CGAL::Explorer page [970](#)
CGAL::Topological_explorer page [965](#)

13.7 Alphabetical List of Reference Pages

Explorer page [970](#)
ExtendedKernelTraits_2 page [972](#)
Extended_cartesian<FT> page [977](#)
Extended_homogeneous<RT> page [978](#)
Filtered_extended_homogeneous<RT> page [979](#)
Nef_polyhedron_2<T> page [960](#)
Topological_explorer page [965](#)

CGAL::Nef_polyhedron_2<T>

Definition

An instance of data type *Nef_polyhedron_2<T>* is a subset of the plane that is the result of forming complements and intersections starting from a finite set *H* of halfspaces. *Nef_polyhedron_2* is closed under all binary set operations *intersection*, *union*, *difference*, *complement* and under the topological operations *boundary*, *closure*, and *interior*.

The template parameter *T* is specified via an extended kernel concept. *T* must be a model of the concept *ExtendedKernelTraits_2*.

```
#include <CGAL/Nef_polyhedron_2.h>
```

Types

Nef_polyhedron_2<T>::Line the oriented lines modeling halfplanes

Nef_polyhedron_2<T>::Point the affine points of the plane.

Nef_polyhedron_2<T>::Direction directions in our plane.

enum Boundary { *EXCLUDED*, *INCLUDED* }; construction selection.

enum Content { *EMPTY*, *COMPLETE* }; construction selection

Creation

```
Nef_polyhedron_2<T> N( Content plane = EMPTY );
```

creates an instance *N* of type *Nef_polyhedron_2<T>* and initializes it to the empty set if *plane* == *EMPTY* and to the whole plane if *plane* == *COMPLETE*.

```
Nef_polyhedron_2<T> N( Line l, Boundary line = INCLUDED );
```

creates a Nef polyhedron *N* containing the halfplane left of *l* including *l* if *line*==*INCLUDED*, excluding *l* if *line*==*EXCLUDED*.

```
template <class Forward_iterator>
```

```
Nef_polyhedron_2<T> N( Forward_iterator it, Forward_iterator end, Boundary b = INCLUDED );
```

creates a Nef polyhedron *N* from the simple polygon *P* spanned by the list of points in the iterator range *[it,end)* and including its boundary if *b* = *INCLUDED* excluding the boundary otherwise. *Forward_iterator* has to be an iterator with value type *Point*. This construction expects that *P* is simple. The degenerate cases where *P* contains no point, one point or spans just one segment (two points) are correctly handled. In all degenerate cases there's only one unbounded face adjacent to the degenerate polygon. If *b* == *INCLUDED* then *N* is just the boundary. If *b* == *EXCLUDED* then *N* is the whole plane without the boundary.

Operations

<i>void</i>	<i>N.clear(Content plane = EMPTY)</i>	makes <i>N</i> the empty set if <i>plane</i> == <i>EMPTY</i> and the full plane if <i>plane</i> == <i>COMPLETE</i> .
<i>bool</i>	<i>N.is_empty()</i>	returns true if <i>N</i> is empty, false otherwise.
<i>bool</i>	<i>N.is_plane()</i>	returns true if <i>N</i> is the whole plane, false otherwise.

Constructive Operations

<i>Nef_polyhedron_2<T></i>	<i>N.complement()</i>	returns the complement of <i>N</i> in the plane.
<i>Nef_polyhedron_2<T></i>	<i>N.interior()</i>	returns the interior of <i>N</i> .
<i>Nef_polyhedron_2<T></i>	<i>N.closure()</i>	returns the closure of <i>N</i> .
<i>Nef_polyhedron_2<T></i>	<i>N.boundary()</i>	returns the boundary of <i>N</i> .
<i>Nef_polyhedron_2<T></i>	<i>N.regularization()</i>	returns the regularized polyhedron (closure of interior).
<i>Nef_polyhedron_2<T></i>	<i>N.intersection(N1)</i>	returns $N \cap N1$.
<i>Nef_polyhedron_2<T></i>	<i>N.join(N1)</i>	returns $N \cup N1$. Note that “union” is a keyword of C++ and cannot be used for this operation.
<i>Nef_polyhedron_2<T></i>	<i>N.difference(N1)</i>	returns $N - N1$.
<i>Nef_polyhedron_2<T></i>	<i>N.symmetric_difference(N1)</i>	returns the symmetric difference $N - T \cup T - N$.

Additionally there are operators $*, +, -, \wedge, !$ which implement the binary operations *intersection*, *join*, *difference*, *symmetric difference*, and the unary operation *complement*, respectively. There are also the corresponding modification operations $*=, +=, -=, \wedge=$.

There are also comparison operations like $<, <=, >, >=, ==, !=$ which implement the relations subset, subset or equal, superset, superset or equal, equality, inequality, respectively.

Exploration - Point location - Ray shooting

As Nef polyhedra are the result of forming complements and intersections starting from a set *H* of halfspaces that are defined by oriented lines in the plane, they can be represented by an attributed plane map $M = (V, E, F)$. For topological queries within *M* the following types and operations allow exploration access to this structure.

Types

Nef_polyhedron_2<T>:: Explorer

a decorator to examine the underlying plane map. See the manual page of *Explorer*.

Nef_polyhedron_2<T>:: Object_handle

a generic handle to an object of the underlying plane map. The kind of object (*vertex*, *halfedge*, *face*) can be determined and the object can be assigned to a corresponding handle by the three functions:

bool assign(Vertex_const_handle& h, Object_handle)

bool assign(Halfedge_const_handle& h, Object_handle)

bool assign(Face_const_handle& h, Object_handle)

where each function returns *true* iff the assignment to *h* was done.

enum Location_mode { DEFAULT, NAIVE, LMWT};

selection flag¹ for the point location mode.

Operations

bool N.contains(Object_handle h)

returns true iff the object *h* is contained in the set represented by *N*.

bool N.contained_in_boundary(Object_handle h)

returns true iff the object *h* is contained in the 1-skeleton of *N*.

Object_handle N.locate(Point p, Location_mode m = DEFAULT)

returns a generic handle *h* to an object (face, halfedge, vertex) of the underlying plane map that contains the point *p* in its relative interior. The point *p* is contained in the set represented by *N* if *N.contains(h)* is true. The location mode flag *m* allows one to choose between different point location strategies.

Object_handle N.ray_shoot(Point p, Direction d, Location_mode m = DEFAULT)

returns a handle *h* with *N.contains(h)*, that can be converted to a *Vertex_*/*Halfedge_*/*Face_const_handle* as described above. The object returned is intersected by the ray starting in *p* with direction *d* and has minimal distance to *p*. The operation returns the null handle *NULL* if the ray shoot along *d* does not hit any object *h* of *N* with *N.contains(h)*. The location mode flag *m* allows one to choose between different point location strategies.

Object_handle *N.ray_shoot_to_boundary(Point p, Direction d, Location_mode m = DEFAULT)*

returns a handle *h*, that can be converted to a *Vertex/Halfedge_const_handle* as described above. The object returned is part of the 1-skeleton of *N*, intersected by the ray starting in *p* with direction *d* and has minimal distance to *p*. The operation returns the null handle *NULL* if the ray shoot along *d* does not hit any 1-skeleton object *h* of *N*. The location mode flag *m* allows one to choose between different point location strategies.

Explorer *N.explorer()*

returns a decorator object that allows read-only access of the underlying plane map. See the manual page *Explorer* for its usage.

Input and Output

A Nef polyhedron *N* can be visualized in a *Window_stream W*. The output operator is defined in the file *CGAL/IO/Nef_polyhedron_2_Window_stream.h*.

Implementation

Nef polyhedra are implemented on top of a halfedge data structure and use linear space in the number of vertices, edges and facets. Operations like *empty* take constant time. The operations *clear*, *complement*, *interior*, *closure*, *boundary*, *regularization*, input and output take linear time. All binary set operations and comparison operations take time $O(n \log n)$ where *n* is the size of the output plus the size of the input.

The point location and ray shooting operations are implemented in two flavors. The *NAIVE* operations run in linear query time without any preprocessing, the *DEFAULT* operations (equals *LMWT*) run in sub-linear query time, but preprocessing is triggered with the first operation. Preprocessing takes time $O(N^2)$, the sub-linear point location time is either logarithmic when LEDA's persistent dictionaries are present or if not then the point location time is worst-case linear, but experiments show often sublinear runtimes. Ray shooting equals point location plus a walk in the constrained triangulation overlayed on the plane map representation. The cost of the walk is proportional to the number of triangles passed in direction *d* until an obstacle is met. In a minimum weight triangulation of the obstacles (the plane map representing the polyhedron) the theory provides a $O(\sqrt{n})$ bound for the number of steps. Our locally minimum weight triangulation approximates the minimum weight triangulation only heuristically (the calculation of the minimum weight triangulation is conjectured to be NP hard). Thus we have no runtime guarantee but a strong experimental motivation for its approximation.

Example

Nef polyhedra are parameterized by a so-called extended geometric kernel. There are three kernels, one based on a homogeneous representation of extended points called *Extended_homogeneous<RT>* where *RT* is a ring type providing additionally a *gcd* operation, one based on a Cartesian representation of extended points called *Extended_cartesian<NT>* where *NT* is a field type, and finally *Filtered_extended_homogeneous<RT>* (an optimized version of the first). The following example uses the filtered homogeneous kernel to construct the intersection of two halfspaces.

```
// file : examples/Nef_2/simple_intersection.C

#include <CGAL/Gmpz.h>
```

```

#include <CGAL/Filtered_extended_homogeneous.h>
#include <CGAL/Nef_polyhedron_2.h>

typedef CGAL::Gmpz RT;
typedef CGAL::Filtered_extended_homogeneous<RT> Extended_kernel;
typedef CGAL::Nef_polyhedron_2<Extended_kernel> Nef_polyhedron;
typedef Nef_polyhedron::Line Line;

int main()
{
    Nef_polyhedron N1(Line(1,0,0));
    Nef_polyhedron N2(Line(0,1,0), Nef_polyhedron::EXCLUDED);
    Nef_polyhedron N3 = N1 * N2; // line (*)
    return 0;
}

```

After line (*) *N3* is the intersection of *N1* and *N2*. The member types of *Nef_polyhedron_2< Extended_homogeneous<NT> >* map to corresponding types of the standard CGAL geometry kernel (type equality in pseudo-code notation):

```

CGAL::Nef_polyhedron_2< CGAL::Extended_cartesian<FT> >::Point
    == CGAL::Cartesian<FT>::Point_2

CGAL::Nef_polyhedron_2< CGAL::Extended_homogeneous<RT> >::Point
    == CGAL::Homogeneous<RT>::Point_2

CGAL::Nef_polyhedron_2< CGAL::Filtered_extended_homogeneous<RT> >::Point
    == CGAL::Homogeneous<RT>::Point_2

```

The same holds for the types *Line* and *Direction* in the local scope of *Nef_polyhedron_2<...>*.

CGAL::Topological_explorer

Definition

An instance D of the data type *Topological_explorer* is a decorator for interfacing the topological structure of a plane map P (read-only).

A plane map P consists of a triple (V, E, F) of vertices, edges, and faces. We collectively call them objects. An edge e is a pair of vertices (v, w) with incidence operations $v = \text{source}(e)$, $w = \text{target}(e)$. The list of all edges with source v is called the adjacency list $A(v)$.

Edges are paired into twins. For each edge $e = (v, w)$ there's an edge $\text{twin}(e) = (w, v)$ and $\text{twin}(\text{twin}(e)) = e$ ².

An edge $e = (v, w)$ knows two adjacent edges $en = \text{next}(e)$ and $ep = \text{previous}(e)$ where $\text{source}(en) = w$, $\text{previous}(en) = e$ and $\text{target}(ep) = v$ and $\text{next}(ep) = e$. By this symmetric *previous-next* relationship all edges are partitioned into face cycles. Two edges e and e' are in the same face cycle if $e = \text{next}^*(e')$. All edges e in the same face cycle have the same incident face $f = \text{face}(e)$. The cyclic order on the adjacency list of a vertex $v = \text{source}(e)$ is given by $\text{cyclic_adj_succ}(e) = \text{twin}(\text{previous}(e))$ and $\text{cyclic_adj_pred}(e) = \text{next}(\text{twin}(e))$.

A vertex v is embedded via coordinates $\text{point}(v)$. By the embedding of its source and target an edge corresponds to a segment. P has the property that the embedding is always *order-preserving*. This means a ray fixed in $\text{point}(v)$ of a vertex v and swept around counterclockwise meets the embeddings of $\text{target}(e)$ ($e \in A(v)$) in the cyclic order defined by the list order of A .

The embedded face cycles partition the plane into maximal connected subsets of points. Each such set corresponds to a face. A face is bounded by its incident face cycles. For all the edges in the non-trivial face cycles it holds that the face is left of the edges. There can also be trivial face cycles in form of isolated vertices in the interior of a face. Each such vertex v knows its surrounding face $f = \text{face}(v)$.

Plane maps are attributed, for each object $u \in V \cup E \cup F$ we attribute an information $\text{mark}(u)$ of type *Mark*. *Mark* fits the concepts assignable, default-constructible, and equal-comparable.

Types

<i>Topological_explorer::Plane_map</i>	The underlying plane map type
<i>Topological_explorer::Point</i>	The point type of vertices.
<i>Topological_explorer::Mark</i>	All objects (vertices, edges, faces) are attributed by a <i>Mark</i> object.
<i>Topological_explorer::Size_type</i>	The size type.

Local types are handles, iterators and circulators of the following kind: *Vertex_const_handle*, *Vertex_const_iterator*, *Halfedge_const_handle*, *Halfedge_const_iterator*, *Face_const_handle*, *Face_const_iterator*. Additionally the following circulators are defined.

Topological_explorer::Halfedge_around_vertex_const_circulator

circulating the outgoing halfedges in $A(v)$.

²The existence of the edge pairs makes P a bidirected graph, the *twin* links make P a map.

Topological_explorer:: Halfedge_around_face_const_circulator

circulating the halfedges in the face cycle of a face f .

Topological_explorer:: Hole_const_iterator

iterating all holes of a face f . The type is convertible to *Halfedge_const_handle*.

Topological_explorer:: Isolated_vertex_const_iterator

iterating all isolated vertices of a face f . The type generalizes *Vertex_const_handle*.

Operations

Vertex_const_handle

D.source(Halfedge_const_handle e)

returns the source of e .

Vertex_const_handle

D.target(Halfedge_const_handle e)

returns the target of e .

Halfedge_const_handle

D.twin(Halfedge_const_handle e)

returns the twin of e .

bool

D.is_isolated(Vertex_const_handle v)

returns *true* iff $A(v) = \emptyset$.

Halfedge_const_handle

D.first_out_edge(Vertex_const_handle v)

returns one halfedge with source v . It's the starting point for the circular iteration over the halfedges with source v .

Precondition: !is_isolated(v).

Halfedge_const_handle

D.last_out_edge(Vertex_const_handle v)

returns the halfedge with source v that is the last in the circular iteration before encountering *first_out_edge(v)* again.

Precondition: !is_isolated(v).

Halfedge_const_handle

D.cyclic_adj_succ(Halfedge_const_handle e)

returns the edge after e in the cyclic ordered adjacency list of *source(e)*.

<i>Halfedge_const_handle</i>	<i>D.cyclic_adj_pred(Halfedge_const_handle e)</i>	returns the edge before <i>e</i> in the cyclic ordered adjacency list of <i>source(e)</i> .
<i>Halfedge_const_handle</i>	<i>D.next(Halfedge_const_handle e)</i>	returns the next edge in the face cycle containing <i>e</i> .
<i>Halfedge_const_handle</i>	<i>D.previous(Halfedge_const_handle e)</i>	returns the previous edge in the face cycle containing <i>e</i> .
<i>Face_const_handle</i>	<i>D.face(Halfedge_const_handle e)</i>	returns the face incident to <i>e</i> .
<i>Face_const_handle</i>	<i>D.face(Vertex_const_handle v)</i>	returns the face incident to <i>v</i> . <i>Precondition: is_isolated(v)</i> .
<i>Halfedge_const_handle</i>	<i>D.halfedge(Face_const_handle f)</i>	returns a halfedge in the bounding face cycle of <i>f</i> (<i>Halfedge_const_handle()</i>) if there is no bounding face cycle).

Iteration

<i>Vertex_const_iterator</i>	<i>D.vertices_begin()</i>	iterator over vertices of the map.
<i>Vertex_const_iterator</i>	<i>D.vertices_end()</i>	past-the-end iterator for vertices.
<i>Halfedge_const_iterator</i>	<i>D.halfedges_begin()</i>	iterator over halfedges of the map.
<i>Halfedge_const_iterator</i>	<i>D.halfedges_end()</i>	past-the-end iterator for halfedges.
<i>Face_const_iterator</i>	<i>D.faces_begin()</i>	iterator over faces of the map.
<i>Face_const_iterator</i>	<i>D.faces_end()</i>	past-the-end iterator for faces

Halfedge_around_vertex_const_circulator

D.out_edges(Vertex_const_handle v)

returns a circulator for the cyclic adjacency list of *v*.

Halfedge_around_face_const_circulator

D.face_cycle(Face_const_handle f)

returns a circulator for the outer face cycle of *f*.

Hole_const_iterator

D.holes_begin(Face_const_handle f)

returns an iterator for all holes in the interior of *f*. A *Hole_iterator* can be assigned to a *Halfedge_around_face_const_circulator*.

Hole_const_iterator

D.holes_end(Face_const_handle f)

returns the past-the-end iterator of *f*.

Isolated_vertex_const_iterator

D.isolated_vertices_begin(Face_const_handle f)

returns an iterator for all isolated vertices in the interior of *f*.

Isolated_vertex_const_iterator

D.isolated_vertices_end(Face_const_handle f)

returns the past the end iterator of *f*.

Associated Information

The type *Mark* is the general attribute of an object.

Point

D.point(Vertex_const_handle v)

returns the embedding of *v*.

Mark

D.mark(Vertex_const_handle v)

returns the mark of *v*.

<i>Mark</i>	<i>D.mark(Halfedge_const_handle e)</i>	returns the mark of <i>e</i> .
<i>Mark</i>	<i>D.mark(Face_const_handle f)</i>	returns the mark of <i>f</i> .
Statistics and Integrity		
<i>Size_type</i>	<i>D.number_of_vertices()</i>	returns the number of vertices.
<i>Size_type</i>	<i>D.number_of_halfedges()</i>	returns the number of halfedges.
<i>Size_type</i>	<i>D.number_of_edges()</i>	returns the number of halfedge pairs.
<i>Size_type</i>	<i>D.number_of_faces()</i>	returns the number of faces.
<i>Size_type</i>	<i>D.number_of_face_cycles()</i>	returns the number of face cycles.
<i>Size_type</i>	<i>D.number_of_connected_components()</i>	calculates the number of connected components of <i>P</i> .
<i>void</i>	<i>D.print_statistics(std::ostream& os = std::cout)</i>	print the statistics of <i>P</i> : the number of vertices, edges, and faces.
<i>void</i>	<i>D.check_integrity_and_topological_planarity(bool faces=true)</i>	checks the link structure and the genus of <i>P</i> .

CGAL::Explorer

Definition

An instance E of the data type *Explorer* is a decorator to explore the structure of the plane map underlying the Nef polyhedron. It inherits all topological adjacency exploration operations from *Topological_explorer*. *Explorer* additionally allows one to explore the geometric embedding.

The position of each vertex is given by a so-called extended point, which is either a standard affine point or the tip of a ray touching an infinimaximal square frame centered at the origin. A vertex v is called a *standard* vertex if its embedding is a *standard* point and *non-standard* if its embedding is a *non-standard* point. By the straightline embedding of their source and target vertices, edges correspond to either affine segments, rays or lines or are part of the bounding frame.

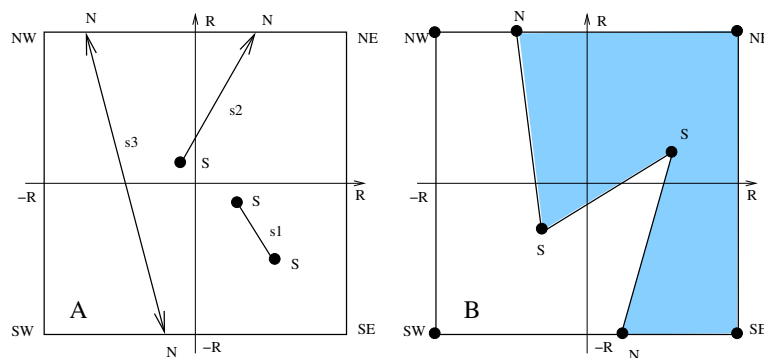


Figure 13.2: Extended geometry: standard vertices are marked by S, non-standard vertices are marked by N. **A:** The possible embeddings of edges: an affine segment $s1$, an affine ray $s2$, an affine line $s3$. **B:** A plane map embedded by extended geometry: note that the frame is arbitrarily large, the 6 vertices on the frame are at infinity, the two faces represent a geometrically unbounded area, however they are topologically closed by the frame edges. No standard point can be placed outside the frame.

Inherits From

Topological_explorer

Types

Explorer::Point the point type of finite vertices.

Explorer::Ray the ray type of vertices on the frame.

Iterators, handles, and circulators are inherited from *Topological_explorer*.

Creation

Explorer is copy constructable and assignable. An object can be obtained via the *Nef_polyhedron_2::explorer()* method of *Nef_polyhedron_2*.

Operations

bool *E.is_standard(Vertex_const_handle v)*

returns true iff *v*'s position is a standard point.

Point *E.point(Vertex_const_handle v)*

returns the standard point that is the embedding of *v*.

Precondition: E.is_standard(v).

Ray *E.ray(Vertex_const_handle v)*

returns the ray defining the non-standard point on the frame.

Precondition: !E.is_standard(v).

bool *E.is_frame_edge(Halfedge_const_handle e)*

returns true iff *e* is part of the infinimaximal frame.

ExtendedKernelTraits_2

Definition

ExtendedKernelTraits_2 is a kernel concept providing extended geometry³. Let K be an instance of the data type *ExtendedKernelTraits_2*. The central notion of extended geomtry are extended points. An extended point represents either a standard affine point of the Cartesian plane or a non-standard point representing the equivalence class of rays where two rays are equivalent if one is contained in the other.

Let R be an infinimaximal number⁴, F be the square box with corners $NW(-R, R)$, $NE(R, R)$, $SE(R, -R)$, and $SW(-R, -R)$. Let p be a non-standard point and let r be a ray defining it. If the frame F contains the source point of r then let $p(R)$ be the intersection of r with the frame F , if F does not contain the source of r then $p(R)$ is undefined. For a standard point let $p(R)$ be equal to p if p is contained in the frame F and let $p(R)$ be undefined otherwise. Clearly, for any standard or non-standard point p , $p(R)$ is defined for any sufficiently large R . Let f be any function on standard points, say with k arguments. We call f *extensible* if for any k points p_1, \dots, p_k the function value $f(p_1(R), \dots, p_k(R))$ is constant for all sufficiently large R . We define this value as $f(p_1, \dots, p_k)$. Predicates like lexicographic order of points, orientation, and incircle tests are extensible.

An extended segment is defined by two extended points such that it is either an affine segment, an affine ray, an affine line, or a segment that is part of the square box. Extended directions extend the affine notion of direction to extended objects.

This extended geometry concept serves two purposes. It offers functionality for changing between standard affine and extended geometry. At the same time it provides extensible geometric primitives on the extended geometric objects.

Types

Affine kernel types

<i>ExtendedKernelTraits_2::Standard_kernel</i>	the standard affine kernel.
<i>ExtendedKernelTraits_2::Standard_RT</i>	the standard ring type.
<i>ExtendedKernelTraits_2::Standard_point_2</i>	standard points.
<i>ExtendedKernelTraits_2::Standard_segment_2</i>	standard segments.
<i>ExtendedKernelTraits_2::Standard_line_2</i>	standard oriented lines.
<i>ExtendedKernelTraits_2::Standard_direction_2</i>	standard directions.
<i>ExtendedKernelTraits_2::Standard_ray_2</i>	standard rays.

³It is called extended geometry for simplicity, though it is not a real geometry in the classical sense.

⁴A finite but very large number.

ExtendedKernelTraits_2:: Standard_aff_transformation_2

standard affine transformations.

Extended kernel types

ExtendedKernelTraits_2:: RT

the ring type of our extended kernel.

ExtendedKernelTraits_2:: Point_2

extended points.

ExtendedKernelTraits_2:: Segment_2

extended segments.

ExtendedKernelTraits_2:: Direction_2

extended directions.

```
enum Point_type { SWCORNER,
                  LEFTFRAME,
                  NWCORNER,
                  BOTTOMFRAME,
                  STANDARD,
                  TOPFRAME,
                  SECORNER,
                  RIGHTFRAME,
                  NECORNER }
```

a type descriptor for extended points.

Operations

Interfacing the affine kernel types

Point_2 *K.construct_point(Standard_point_2 p)*

creates an extended point and initializes it to the standard point *p*.

Point_2 *K.construct_point(Standard_line_2 l)*

creates an extended point and initializes it to the equivalence class of all the rays underlying the oriented line *l*.

Point_2 *K.construct_point(Standard_point_2 p1, Standard_point_2 p2)*

creates an extended point and initializes it to the equivalence class of all the rays underlying the oriented line *l(p1,p2)*.

<i>Point_2</i>	<i>K.construct_point(Standard_point_2 p, Standard_direction_2 d)</i>	creates an extended point and initializes it to the equivalence class of all the rays underlying the ray starting in <i>p</i> in direction <i>d</i> .
<i>Point_2</i>	<i>K.construct_opposite_point(Standard_line_2 l)</i>	creates an extended point and initializes it to the equivalence class of all the rays underlying the oriented line opposite to <i>l</i> .
<i>Point_type</i>	<i>K.type(Point_2 p)</i>	determines the type of <i>p</i> and returns it.
<i>bool</i>	<i>K.is_standard(Point_2 p)</i>	returns <i>true</i> iff <i>p</i> is a standard point.
<i>Standard_point_2</i>	<i>K.standard_point(Point_2 p)</i>	returns the standard point represented by <i>p</i> . <i>Precondition: K.is_standard(p).</i>
<i>Standard_line_2</i>	<i>K.standard_line(Point_2 p)</i>	returns the oriented line representing the bundle of rays defining <i>p</i> . <i>Precondition: !K.is_standard(p).</i>
<i>Standard_ray_2</i>	<i>K.standard_ray(Point_2 p)</i>	a ray defining <i>p</i> . <i>Precondition: !K.is_standard(p).</i>
<i>Point_2</i>	<i>K.NE()</i>	returns the point on the northeast frame corner.
<i>Point_2</i>	<i>K.SE()</i>	returns the point on the southeast frame corner.
<i>Point_2</i>	<i>K.NW()</i>	returns the point on the northwest frame corner.
<i>Point_2</i>	<i>K.SW()</i>	returns the point on the southwest frame corner.

Geometric kernel calls

<i>Point_2</i>	<i>K.source(Segment_2 s)</i>	returns the source point of <i>s</i> .
----------------	-------------------------------	----------------------------------------

<i>Point_2</i>	<i>K.target(Segment_2 s)</i>	returns the target point of <i>s</i> .
<i>Segment_2</i>	<i>K.construct_segment(Point_2 p, Point_2 q)</i>	constructs a segment <i>pq</i> .
<i>int</i>	<i>K.orientation(Segment_2 s, Point_2 p)</i>	returns the orientation of <i>p</i> with respect to the line through <i>s</i> .
<i>int</i>	<i>K.orientation(Point_2 p1, Point_2 p2, Point_2 p3)</i>	returns the orientation of <i>p3</i> with respect to the line through <i>p1p2</i> .
<i>bool</i>	<i>K.left_turn(Point_2 p1, Point_2 p2, Point_2 p3)</i>	return true iff the <i>p3</i> is left of the line through <i>p1p2</i> .
<i>bool</i>	<i>K.is_degenerate(Segment_2 s)</i>	return true iff <i>s</i> is degenerate.
<i>int</i>	<i>K.compare_xy(Point_2 p1, Point_2 p2)</i>	returns the lexicographic order of <i>p1</i> and <i>p2</i> .
<i>int</i>	<i>K.compare_x(Point_2 p1, Point_2 p2)</i>	returns the order on the <i>x</i> -coordinates of <i>p1</i> and <i>p2</i> .
<i>int</i>	<i>K.compare_y(Point_2 p1, Point_2 p2)</i>	returns the order on the <i>y</i> -coordinates of <i>p1</i> and <i>p2</i> .
<i>Point_2</i>	<i>K.intersection(Segment_2 s1, Segment_2 s2)</i>	returns the point of intersection of the lines supported by <i>s1</i> and <i>s2</i> . <i>Precondition:</i> the intersection point exists.
<i>Direction_2</i>	<i>K.construct_direction(Point_2 p1, Point_2 p2)</i>	returns the direction of the vector <i>p2 - p1</i> .

<i>bool</i>	<i>K.strictly_ordered_ccw(Direction_2 d1, Direction_2 d2, Direction_2 d3)</i>	returns <i>true</i> iff <i>d2</i> is in the interior of the counterclockwise angular sector between <i>d1</i> and <i>d3</i> .
<i>bool</i>	<i>K.strictly_ordered_along_line(Point_2 p1, Point_2 p2, Point_2 p3)</i>	returns <i>true</i> iff <i>p2</i> is in the relative interior of the segment <i>p1p3</i> .
<i>bool</i>	<i>K.contains(Segment_2 s, Point_2 p)</i>	returns <i>true</i> iff <i>s</i> contains <i>p</i> .
<i>bool</i>	<i>K.first_pair_closer_than_second(Point_2 p1, Point_2 p2, Point_2 p3, Point_2 p4)</i>	returns <i>true</i> iff $\ p1 - p2\ < \ p3 - p4\ $.
<i>const char*</i>	<i>K.output_identifier()</i>	returns a unique identifier for kernel object Input/Output. Usually this should be the name of the model.

Has Models

<i>CGAL::Extended_cartesian<FT></i>	page 977
<i>CGAL::Extended_homogeneous<RT></i>	page 978
<i>CGAL::Filtered_extended_homogeneous<RT></i>	page 979

CGAL::Extended_cartesian<FT>

Definition

The class *Extended_cartesian*<FT> serves as a traits class for the class *CGAL::Nef_polyhedron_2*<T>. It uses a polynomial component representation based on a field number type *FT*.

```
#include <CGAL/Extended_cartesian.h>
```

Is Model for the Concepts

ExtendedKernelTraits_2 page [972](#)

Creation

Extended_cartesian<FT> traits; default constructor.

Requirements

To make a field number type *FT_model* work with this class, you must provide a traits class for this number type: *CGAL::Number_type_traits*<FT_model> (See the support library manual.)

Operations

Fits all operation requirements of the concept.

See Also

CGAL::Extended_homogeneous<RT> page [978](#)
CGAL::Filtered_extended_homogeneous<RT> page [979](#)

CGAL::Extended_homogeneous<RT>

Definition

The class *Extended_homogeneous<RT>* serves as a traits class for the class *CGAL::Nef_polyhedron_2<T>*. It uses a polynomial component representation based on a Euclidean ring number type *RT*.

`#include <CGAL/Extended_homogeneous.h>`

Is Model for the Concepts

ExtendedKernelTraits_2 page [972](#)

Creation

Extended_homogeneous<RT> traits; default constructor.

Requirements

To make an Euclidean ring number type *RT_model* work with this class the number type must support a gcd computation in namespace *CGAL::NTS*. CGAL provides a function template for this, which will be used by default when your number type is not one of the built-in number types, one of the number types distributed with CGAL or one of the LEDA number types.

Operations

Fits all operation requirements of the concept.

See Also

CGAL::Extended_cartesian<FT> page [977](#)
CGAL::Filtered_extended_homogeneous<RT> page [979](#)

CGAL::Filtered_extended_homogeneous<RT>

Definition

The class *Filtered_extended_homogeneous*<RT> serves as a traits class for the class *CGAL::Nef_polyhedron_2*<T>. It uses a polynomial component representation based on a ring number type *RT*.

`#include <CGAL/Filtered_extended_homogeneous.h>`

Is Model for the Concepts

ExtendedKernelTraits_2 page [972](#)

Creation

Filtered_extended_homogeneous<RT> traits; default constructor.

Operations

Fits all operation requirements of the concept.

See Also

CGAL::Extended_cartesian<FT> page [977](#)

CGAL::Extended_homogeneous<RT> page [978](#)

Chapter 14

2D Boolean Operations on Nef Polygons Embedded on the Sphere

Peter Hachenberger and Lutz Kettner

Contents

14.1 Introduction	981
14.2 Restricted Spherical Geometry	982
14.3 Example Programs	983
14.3.1 First Example	983
14.3.2 Construction and Combinations	983
14.3.3 Exploration	984
14.3.4 Point Location	985
14.3.5 Visualization	986

14.1 Introduction

Nef polyhedra are defined as a subset of the d -dimensional space obtained by a finite number of set complement and set intersection operations on halfspaces.

Due to the fact that all other binary set operations like union, difference and symmetric difference can be reduced to intersection and complement calculations, Nef polyhedra are also closed under those operations. Also, Nef polyhedra are closed under topological unary set operations. Given a Nef polyhedron one can determine its interior, its boundary, and its closure.

Additionally, a d -dimensional Nef polyhedron has the property, that its boundary is a $(d-1)$ -dimensional Nef polyhedron. This property can be used as a way to represent 3-dimensional Nef polyhedra by means of planar Nef polyhedra. This is done by intersecting the neighborhood of a vertex in a 3D Nef polyhedron with an ε -sphere. The result is a planar Nef polyhedron embedded on the sphere.

The intersection of a halfspace going through the center of the ε -sphere, with the ε -sphere, results in a halfsphere which is bounded by a great circle. A binary operation of two halfspheres cuts the great circles into great arcs.

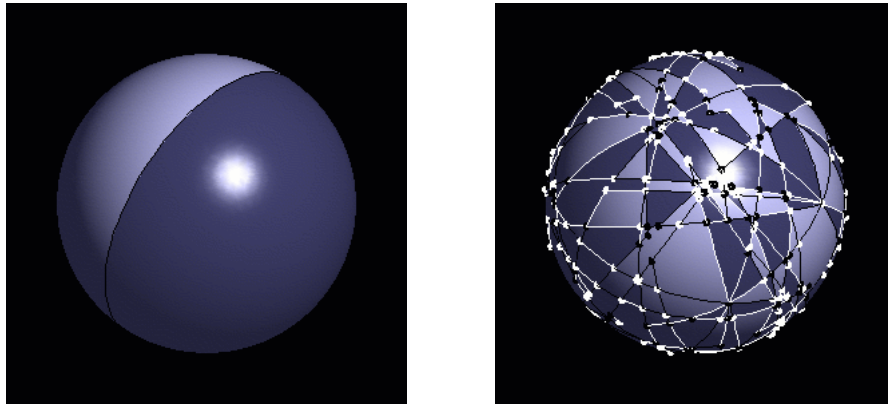
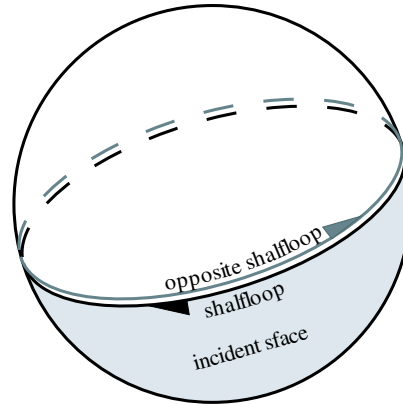


Figure 14.1: Two spherical Nef polyhedra. A closed halfspace on the left and a complex polyhedron on the right. The different colors indicate selected and unselected regions, lines and points.



The incidence structure of planar Nef polyhedra can be reused. The items are denoted as *svertex*, *shalfedge* and *sface*, analogous to their counterparts in *Nef_polyhedron_S2*. Additionally, there is the *shalfloop* representing the great circles. The incidences are illustrated in the figure above.

14.2 Restricted Spherical Geometry

We introduce geometric objects that are part of the spherical surface S_2 and operations on them. We define types *Sphere_point*, *Sphere_circle*, *Sphere_segment*, and *Sphere_direction*. *Sphere_points* are points on S_2 , *Sphere_circles* are oriented great circles of S_2 , *Sphere_segments* are oriented parts of *Sphere_circles* bounded by a pair of *Sphere_points*, and *Sphere_directions* are directions that are part of great circles. (a direction is usually defined to be a vector without length, that floats around in its underlying space and can be used to specify a movement at any point of the underlying space; in our case we use directions only at points that are part of the great circle that underlies also the direction.)

Note that we have to consider special geometric properties of the objects. For example two points that are part of a great circle define two *Sphere_segments*, and two arbitrary *Sphere_segments* can intersect in two points.

If we restrict our geometric objects to a so-called perfect hemisphere of S_2 ¹ then the restricted objects behave like in classical geometry, e.g., two points define exactly one segment, two segments intersect in at most one

¹A perfect hemisphere of S_2 is an open half-sphere plus an open half-circle in the boundary of the open half-sphere plus one endpoint of the half-circle.

interior point (non-degenerately), or three non-cocircular sphere points can be qualified as being positively or negatively oriented.

14.3 Example Programs

14.3.1 First Example

In this first example *Nef_polyhedron_S2* is parametrized with a CGAL Kernel as traits class. The types comprising the spherical geometry can be retrieved from the type *Nef_polyhedron_S2<Traits>* as is done in the example with the type *Sphere_circle*. Then three Nef polyhedra are created: *N1* is a halfsphere including the boundary, *N2* is another halfsphere without the boundary, and *N3* is the intersection of *N1* and *N2*.

```
// examples/Nef_S2/simple.C

#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_S2.h>

typedef CGAL::Gmpz RT;
typedef CGAL::Homogeneous<RT> Kernel;
typedef CGAL::Nef_polyhedron_S2<Kernel> Nef_polyhedron;
typedef Nef_polyhedron::Sphere_circle Sphere_circle;

int main()
{
    Nef_polyhedron N1(Sphere_circle(1,0,0));
    Nef_polyhedron N2(Sphere_circle(0,1,0), Nef_polyhedron::EXCLUDED);
    Nef_polyhedron N3 = N1 * N2;
    return 0;
}
```

14.3.2 Construction and Combinations

The example shows the different types of constructors: *N1* is the complete sphere, *N2* is a halfsphere which includes the boundary, *N3* is created with the copy constructor, *N4* is created as an arrangement of a set of *Sphere_segments*, and *N5* is created as the empty set.

The example also shows the use of unary set operations, binary operations, and binary predicates: *N3* is defined as the complement of *N2*, *N1* is compared with the union of *N2* and *N3*, *N5* is united with *N2* and then intersected with *N4*. At last, it is tested if *N5* is a subset of *N2* and if *N5* is not equal to *N4*.

```
// examples/Nef_S2/construction.C

#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_S2.h>

typedef CGAL::Gmpz RT;
```

```

typedef CGAL::Homogeneous<RT> Kernel;
typedef CGAL::Nef_polyhedron_S2<Kernel> Nef_polyhedron;
typedef Nef_polyhedron::Sphere_point Sphere_point;
typedef Nef_polyhedron::Sphere_segment Sphere_segment;
typedef Nef_polyhedron::Sphere_circle Sphere_circle;

int main() {

    Nef_polyhedron N1(Nef_polyhedron::COMPLETE);

    Sphere_circle c(1,1,1); // c : x + y + z = 0
    Nef_polyhedron N2(c, Nef_polyhedron::INCLUDED);
    Nef_polyhedron N3(N2.complement());
    CGAL_assertion(N1 == N2.join(N3));

    Sphere_point p1(1,0,0), p2(0,1,0), p3(0,0,1);
    Sphere_segment s1(p1,p2), s2(p2,p3), s3(p3,p1);
    Sphere_segment triangle[3] = { s1, s2, s3 };
    Nef_polyhedron N4(triangle, triangle+3);
    Nef_polyhedron N5;
    N5 += N2;
    N5 = N5.intersection(N4);
    CGAL_assertion(N5 <= N2 && N5 != N4);

    return 0;
}

```

14.3.3 Exploration

By recursively composing binary and unary operations one can end with a very complex rectilinear structure. *Nef_polyhedron_S2* allows read-only exploration of the structure.

In the following example, a random *Nef_polyhedron_S2* *S* created from *n* halfspheres is explored. Each sface is composed of one outer sface cycles and an arbitrary number of inner sfaces cycles. The outer cycle is either an shalfloop or a cycle of shalfedges. An inner cycles additionally can be an isolated vertex. The example shows how to get the entry item *it* to all sface cycles of an sface *sf* and how to find out what type of item it is.

The macro *CGAL_forall_sface_cycles_of* is equivalent to a for-loop on the range [*sf*->*sface_cycles_begin()*, *sf*->*sface_cycles_end()*). An *SFace_cycle_const_iterator* either represents a *SVertex_const_handle*, a *SHalfedge_const_handle* or a *SHalfloop_const_handle*. In order to find out which handle type is represented, the functions *is_svertex()*, *is_shalfedge()* and *is_shalfloop()* are provided. Afterwards the iterator can be casted to the proper handle type.

```

// examples/Nef_S2/exploration.C

#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_S2.h>
#include <CGAL/Nef_S2/create_random_Nef_S2.h>

```



```

typedef CGAL::Gmpz RT;
typedef CGAL::Homogeneous<RT> Kernel;
typedef CGAL::Nef_polyhedron_S2<Kernel> Nef_polyhedron_S2;
typedef Nef_polyhedron_S2::SVertex_const_handle SVertex_const_handle;
typedef Nef_polyhedron_S2::SHalfedge_const_handle SHalfedge_const_handle;
typedef Nef_polyhedron_S2::SHalfloop_const_handle SHalfloop_const_handle;
typedef Nef_polyhedron_S2::SFace_const_iterator SFace_const_iterator;
typedef Nef_polyhedron_S2::SFace_cycle_const_iterator
        SFace_cycle_const_iterator;

int main() {

    Nef_polyhedron_S2 S;
    CGAL::create_random_Nef_S2(S,5);

    int i=0;
    SFace_const_iterator sf;
    CGAL_forall_sfaced(S) {
        SFace_cycle_const_iterator it;
        std::cout << "the sfaced cycles of sfaced " << i++;
        std::cout << " start with an " << std::endl;
        CGAL_forall_sfaced_cycles_of(it,sf) {
            if (it.is_svertex()) {
std::cout << " svertex at position ";
std::cout << SVertex_const_handle(it)->point() << std::endl;
            }
            else if (it.is_shalfedge()) {
std::cout << " shalfedge from ";
std::cout << SHalfedge_const_handle(it)->source()->point() << " to ";
std::cout << SHalfedge_const_handle(it)->target()->point() << std::endl;
            }
            else if (it.is_shalfloop()) {
std::cout << " shalfloop lying in the plane ";
std::cout << SHalfloop_const_handle(it)->circle() << std::endl;
            }
            else
std::cout << "something is wrong" << std::endl;
        }
    }
    return 0;
}

```

14.3.4 Point Location

Using the *locate* function, it is possible to retrieve an item at a certain location on the sphere. In the following example, the item at location *Sphere_point(1,0,0)* in a random *Nef_polyhedron_S2* is retrieved. *locate* returns an instance of type *Object_handle*, which is a container for any handle type. Here, it either a *SVertex_const_handle*, a *SHalfedge_const_handle*, a *SHalfloop_const_handle* or a *SFace_const_handle*. The function *CGAL::assign* performs the cast operation and returns a boolean which indicates whether the cast was successful or not.

```

// examples/Nef_S2/point_location.C

#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_S2.h>
#include <CGAL/Nef_S2/create_random_Nef_S2.h>

typedef CGAL::Gmpz RT;
typedef CGAL::Homogeneous<RT> Kernel;
typedef CGAL::Nef_polyhedron_S2<Kernel> Nef_polyhedron_S2;
typedef Nef_polyhedron_S2::SVertex_const_handle SVertex_const_handle;
typedef Nef_polyhedron_S2::SHalfedge_const_handle SHalfedge_const_handle;
typedef Nef_polyhedron_S2::SHalfloop_const_handle SHalfloop_const_handle;
typedef Nef_polyhedron_S2::SFace_const_handle SFace_const_handle;
typedef Nef_polyhedron_S2::Object_handle Object_handle;
typedef Nef_polyhedron_S2::Sphere_point Sphere_point;

int main() {

    Nef_polyhedron_S2 S;
    CGAL::create_random_Nef_S2(S,5);

    SVertex_const_handle sv;
    SHalfedge_const_handle se;
    SHalfloop_const_handle sl;
    SFace_const_handle sf;
    Object_handle o = S.locate(Sphere_point(1,0,0));
    if(CGAL::assign(sv,o))
        std::cout << "Locating svertex" << std::endl;
    else if(CGAL::assign(se,o))
        std::cout << "Locating shalfedge" << std::endl;
    else if(CGAL::assign(sl,o))
        std::cout << "Locating shalfloop" << std::endl;
    else if(CGAL::assign(sf,o))
        std::cout << "Locating sface" << std::endl;
    else {
        std::cout << "something wrong" << std::endl;
        return 1;
    }
    return 0;
}

```

14.3.5 Visualization

Nef_polyhedron_S2 provides an interface for OpenGL visualization via a Qt widget. The usage is shown in the following example:

```

// Copyright (c) 2004 Max-Planck-Institute Saarbruecken (Germany).
// All rights reserved.
//

```

```

// This file is part of CGAL (www.cgal.org); you may redistribute it under
// the terms of the Q Public License version 1.0.
// See the file LICENSE.QPL distributed with CGAL.
//
// Licensees holding a valid commercial license may use this file in
// accordance with the commercial license agreement provided with the software.
//
// This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
// WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
//
// $URL: svn+ssh://scm.gforge.inria.fr/svn/cgal/branches/CGAL-3.2-branch/Nef_S2/demo/Nef_S2/visualization
// $Id: visualization.C 29613 2006-03-19 19:35:17Z spion $
//
//
// Author(s)      : Peter Hachenberger <hachenberger@mpi-sb.mpg.de>

#include <CGAL/basic.h>

#ifndef CGAL_USE_QT
#include <iostream>
int main(int, char*){
    std::cout << "Sorry, this demo needs QT..." << std::endl; return 0;}
#else
#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_S2.h>
#include <CGAL/Nef_S2/create_random_Nef_S2.h>
#include <CGAL/IO/Qt_widget_Nef_S2.h>
#include <qapplication.h>

typedef CGAL::Gmpz RT;
typedef CGAL::Homogeneous<RT> Kernel;
typedef CGAL::Nef_polyhedron_S2<Kernel> Nef_polyhedron_S2;

int main(int argc, char* argv[]) {

    Nef_polyhedron_S2 S;
    create_random_Nef_S2(S,5);

    QApplication a(argc, argv);
    CGAL::Qt_widget_Nef_S2<Nef_polyhedron_S2>* w =
        new CGAL::Qt_widget_Nef_S2<Nef_polyhedron_S2>(S);
    a.setMainWidget(w);
    w->show();
    return a.exec();
}
#endif

```


2D Boolean Operations on Nef Polygons Embedded on the Sphere Reference Manual

Peter Hachenberger, Lutz Kettner, and Michael Seel

Nef polyhedra are defined as a subset of the d -dimensional space obtained by a finite number of set complement and set intersection operations on halfspaces.

Due to the fact that all other binary set operations like union, difference and symmetric difference can be reduced to intersection and complement calculations, Nef polyhedra are also closed under those operations. Also, Nef polyhedra are closed under topological unary set operations. Given a Nef polyhedron one can determine its interior, its boundary, and its closure.

Additionally, a d -dimensional Nef polyhedron has the property, that its boundary is a $(d-1)$ -dimensional Nef polyhedron. This property can be used as a way to represent 3-dimensional Nef polyhedra by means of planar Nef polyhedra. This is done by intersecting the neighborhood of a vertex in a 3D Nef polyhedron with an ε -sphere. The result is a planar Nef polyhedron embedded on the sphere.

The intersection of a halfspace going through the center of the ε -sphere, with the ε -sphere, results in a halfsphere which is bounded by a great circle. A binary operation of two halfspheres cuts the great circles into great arcs.

The incidence structure of planar Nef polyhedra can be reused. The items are denoted as *svertex*, *shalfedge* and *sface*, analogous to their counterparts in *Nef_polyhedron_2*. Additionally, there is the *shalfloop* representing the great circles.

14.4 Classified Reference Pages

14.5 Alphabetical List of Reference Pages

<i>Nef_polyhedron_S2<Traits></i>	page 991
<i>Qt_widget_Nef_S2<Nef_polyhedron_S2></i>	page 1009
<i>SFace_cycle_iterator</i>	page 1008
<i>SFace</i>	page 1007
<i>SHalfedge</i>	page 1003
<i>SHalfloop</i>	page 1005
<i>Sphere_circle</i>	page 1000
<i>Sphere_point</i>	page 997

<i>Sphere_segment</i>	page 998
<i>SVertex</i>	page 1002

CGAL::Nef_polyhedron_S2<Traits>

Definition

An instance of data type *Nef_polyhedron_S2<Traits>* is a subset of the sphere S_2 that is the result of forming complements and intersections starting from a finite set H of halfspaces bounded by a plane containing the origin. Halfspaces correspond to hemispheres of S_2 and are therefore modeled by oriented great circles of type *Sphere_circle*. *Nef_polyhedron_S2* is closed under all binary set operations *intersection*, *union*, *difference*, *complement* and under the topological operations *boundary*, *closure*, and *interior*.

```
#include <CGAL/Nef_polyhedron_S2.h>
```

Parameters

```
template < class Nef_polyhedronTraits_S2,
            class Nef_polyhedronItems_S2 = CGAL::SM_items,
            class Nef_polyhedronMarks = bool
class Nef_polyhedron_S2;
```

The first parameter requires one of the following exact kernels: *Homogeneous*, *Simple_homogeneous* parametrized with *Gmpz*, *leda_integer* or any other number type modeling \mathbb{Z} , or *Cartesian*, *Simple_cartesian* parametrized with *Gmpq*, *leda_rational*, *Quotient<Gmpz>* or any other number type modeling \mathbb{Q} .

The second parameter and the third parameter are for future considerations. Neither *Nef_polyhedronItems_S2* nor *Nef_polyhedronMarks* is specified, yet. Do not use other than the default types for these two template parameters.

Types

<i>Nef_polyhedron_S2<Traits>::Sphere_point</i>	points in the sphere surface.
<i>Nef_polyhedron_S2<Traits>::Sphere_segment</i>	segments in the sphere surface.
<i>Nef_polyhedron_S2<Traits>::Sphere_circle</i>	oriented great circles modeling spatial half-spaces.

<i>Nef_polyhedron_S2<Traits>::SVertex_const_handle</i>	non-mutable handle to svertex.
<i>Nef_polyhedron_S2<Traits>::SHalfedge_const_handle</i>	non-mutable handle to shalfedge.
<i>Nef_polyhedron_S2<Traits>::SHalfloop_const_handle</i>	non-mutable handle to shalfloop.
<i>Nef_polyhedron_S2<Traits>::SFace_const_handle</i>	non-mutable handle to sface.

<i>Nef_polyhedron_S2<Traits>::SVertex_const_iterator</i>	non-mutable iterator over all svertices.
<i>Nef_polyhedron_S2<Traits>::SHalfedge_const_iterator</i>	non-mutable iterator over all shalfedges.
<i>Nef_polyhedron_S2<Traits>::SHalfloop_const_iterator</i>	non-mutable iterator over all shalfloops.
<i>Nef_polyhedron_S2<Traits>::SFace_const_iterator</i>	non-mutable iterator over all sfaces.

<i>Nef_polyhedron_S2<Traits>::SHalfedge_around_svertex_const_circulator</i>	circulating the adjacency list of an svertex v .
-----------------------------------------------------------------------------------	----------------------------------------------------

Nef_polyhedron_S2<Traits>:: SHalfedge_around_sface_const_circulator

circulating the sface cycle of an sface *f*.

Nef_polyhedron_S2<Traits>:: SFace_cycle_const_iterator

iterating all sface cycles of an sface *f*. The iterator has method *bool is_svertex()*, *bool is_shalfedge()*, *bool is_shalfloop()*, and can be converted to the corresponding handles *SVertex_const_handle*, *SHalfedge_const_handle*, or *SHalfloop_const_handle*.

Nef_polyhedron_S2<Traits>:: Mark

attributes of objects (vertices, edges, faces).

Nef_polyhedron_S2<Traits>:: size_type

size type

enum Boundary { EXCLUDED, INCLUDED};

construction selection.

enum Content { EMPTY, COMPLETE};

construction selection.

Creation

Nef_polyhedron_S2<Traits> N(Content sphere = EMPTY);

creates an instance *N* of type *Nef_polyhedron_S2<K>* and initializes it to the empty set if *sphere == EMPTY* and to the whole sphere if *sphere == COMPLETE*.

Nef_polyhedron_S2<Traits> N(Sphere_circle c, Boundary circle = INCLUDED);

creates a Nef polyhedron *N* containing the half-sphere left of *c* including *c* if *circle == INCLUDED*, excluding *c* if *circle == EXCLUDED*.

template <class Forward_iterator>

Nef_polyhedron_S2<Traits> N(Forward_iterator first,
Forward_iterator beyond,
Boundary b = INCLUDED)

creates a Nef polyhedron *N* from the set of sphere segments in the iterator range *[first,beyond)*. If the set of sphere segments is a simple polygon that separates the sphere surface into two regions, then the polygonal region that is left of the segment **first* is selected. The polygonal region includes its boundary if *b = INCLUDED* and excludes the boundary otherwise. *Forward_iterator* has to be an iterator with value type *Sphere_segment*.

Operations

void N.clear(Content plane = EMPTY)

makes *N* the empty set if *plane == EMPTY* and the full plane if *plane == COMPLETE*.

<i>bool</i>	<i>N.is_empty()</i>	returns true if <i>N</i> is empty, false otherwise.
<i>bool</i>	<i>N.is_sphere()</i>	returns true if <i>N</i> is the whole sphere, false otherwise.

Constructive Operations

<i>Nef_polyhedron_S2<K></i>	<i>N.complement()</i>	returns the complement of <i>N</i> in the plane.
<i>Nef_polyhedron_S2<K></i>	<i>N.interior()</i>	returns the interior of <i>N</i> .
<i>Nef_polyhedron_S2<K></i>	<i>N.closure()</i>	returns the closure of <i>N</i> .
<i>Nef_polyhedron_S2<K></i>	<i>N.boundary()</i>	returns the boundary of <i>N</i> .
<i>Nef_polyhedron_S2<K></i>	<i>N.regularization()</i>	returns the regularized polyhedron (closure of interior).
<i>Nef_polyhedron_S2<K></i>	<i>N.intersection(Nef_polyhedron_S2<K> N1)</i>	returns $N \cap N1$.
<i>Nef_polyhedron_S2<K></i>	<i>N.join(Nef_polyhedron_S2<K> N1)</i>	returns $N \cup N1$.
<i>Nef_polyhedron_S2<K></i>	<i>N.difference(Nef_polyhedron_S2<K> N1)</i>	returns $N - N1$.
<i>Nef_polyhedron_S2<K></i>	<i>N.symmetric_difference(Nef_polyhedron_S2<K> N1)</i>	returns the symmetric difference $N - T \cup T - N$.

Additionally there are operators $*, +, -, \wedge, !$ which implement the binary operations *intersection*, *union*, *difference*, *symmetric difference*, and the unary operation *complement* respectively. There are also the corresponding modification operations $*=, +=, -=, \wedge=$.

There are also comparison operations like $<, <=, >, >=, ==, !=$ which implement the relations subset, subset or equal, superset, superset or equal, equality, inequality, respectively.

Statistics and Integrity

<i>Size_type</i>	<i>N.number_of_svertices()</i>	returns the number of svertices.
<i>Size_type</i>	<i>N.number_of_shalfedges()</i>	returns the number of shalfedges.
<i>Size_type</i>	<i>N.number_of_sedges()</i>	returns the number of sedges.
<i>Size_type</i>	<i>N.number_of_shalfloops()</i>	returns the number of shalfloops.
<i>Size_type</i>	<i>N.number_of_sloops()</i>	returns the number of sloops.

<i>Size_type</i>	<i>N.number_of_sfaced()</i>	returns the number of sfaced.
<i>Size_type</i>	<i>N.number_of_sfaced_cycles()</i>	returns the number of sfaced cycles.
<i>Size_type</i>	<i>N.number_of_connected_components()</i>	
		calculates the number of connected components of <i>P</i> .
<i>void</i>	<i>N.print_statistics(std::ostream& os = std::cout)</i>	
		print the statistics of <i>P</i> : the number of vertices, edges, and faces.
<i>void</i>	<i>N.check_integrity_and_topological_planarity(bool faces=true)</i>	
		checks the link structure and the genus of <i>P</i> .

Exploration - Point location - Ray shooting

As Nef polyhedra are the result of forming complements and intersections starting from a set H of half-spaces that are defined by oriented lines in the plane, they can be represented by an attributed plane map $M = (V, E, F)$. For topological queries within M the following types and operations allow exploration access to this structure.

Types

Nef_polyhedron_S2<Traits>:: Object_handle

a generic handle to an object of the underlying plane map. The kind of object (*vertex*, *halfedge*, *face*) can be determined and the object can be assigned to a corresponding handle by the three functions:

bool assign(Vertex_const_handle& h, Object_handle)

bool assign(Halfedge_const_handle& h, Object_handle)

bool assign(Face_const_handle& h, Object_handle)

where each function returns *true* iff the assignment to *h* was done.

Operations

<i>bool</i>	<i>N.contains(Object_handle h)</i>	
		returns true iff the object <i>h</i> is contained in the set represented by <i>N</i> .

<i>bool</i>	<i>N.contained_in_boundary(Object_handle h)</i>	
		returns true iff the object <i>h</i> is contained in the 1-skeleton of <i>N</i> .

<i>Object_handle</i>	<i>N.locate(Sphere_point p)</i>	
		returns a generic handle <i>h</i> to an object (face, halfedge, vertex) of the underlying plane map that contains the point <i>p</i> in its relative interior. The point <i>p</i> is contained in the set represented by <i>N</i> if <i>N.contains(h)</i> is true. The location mode flag <i>m</i> allows one to choose between different point location strategies.

Object_handle *N.ray_shoot(Sphere_point p, Sphere_direction d)*

returns a handle *h* with *N.contains(h)* that can be converted to a *Vertex_*/*Halfedge_*/*Face_const_handle* as described above. The object returned is intersected by the ray starting in *p* with direction *d* and has minimal distance to *p*. The operation returns the null handle *NULL* if the ray shoot along *d* does not hit any object *h* of *N* with *N.contains(h)*.

Object_handle *N.ray_shoot_to_boundary(Sphere_point p, Sphere_direction d)*

returns a handle *h* that can be converted to a *Vertex_*/*Halfedge_const_handle* as described above. The object returned is part of the 1-skeleton of *N*, intersected by the ray starting in *p* with direction *d* and has minimal distance to *p*. The operation returns the null handle *NULL* if the ray shoot along *d* does not hit any 1-skeleton object *h* of *N*. The location mode flag *m* allows one to choose between different point location strategies.

Iteration

bool *N.has_shalfloop()* returns true iff there is a shalfloop.

SHalfloop_const_handle *N.shalfloop()* returns access to the sloop.

The list of all objects can be accessed via iterator ranges. For comfortable iteration we also provide iterations macros. The iterator range access operations are of the following kind:

SVertex_iterator svertices_begin()/svertices_end()
SHalfedge_iterator shalfedges_begin()/shalfedges_end()
SHalfloop_iterator shalfloops_begin()/shalfloops_end()
SFace_iterator sfaces_begin()/sfaces_end()

The macros are then *CGAL_forall_svertices(v,M)*, *CGAL_forall_shalfedges(e,M)*, *CGAL_forall_sfases(f,M)*, *CGAL_forall_sface_cycles_of(fc,F)* where *M* is a sphere map and *F* is a sface.

Input and Output

A Nef polyhedron *N* can be visualized in an open GL window. The output operator is defined in the file *CGAL/IO/Nef_polyhedron_2_Window_stream.h*.

Implementation

Nef polyhedra are implemented on top of a halfedge data structure and use linear space in the number of vertices, edges and facets. Operations like *empty* take constant time. The operations *clear*, *complement*, *interior*, *closure*, *boundary*, *regularization*, input and output take linear time. All binary set operations and comparison operations take time $O(n \log n)$ where *n* is the size of the output plus the size of the input.

The point location and ray shooting operations are implemented in the naive way. The operations run in linear query time without any preprocessing.

Example

Nef polyhedra are parameterized by a standard CGAL kernel. The example computes the intersection of two Nef polyhedra *N1* and *N2*.

```
// examples/Nef_S2/simple.C

#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_S2.h>

typedef CGAL::Gmpz RT;
typedef CGAL::Homogeneous<RT> Kernel;
typedef CGAL::Nef_polyhedron_S2<Kernel> Nef_polyhedron;
typedef Nef_polyhedron::Sphere_circle Sphere_circle;

int main()
{
    Nef_polyhedron N1(Sphere_circle(1,0,0));
    Nef_polyhedron N2(Sphere_circle(0,1,0), Nef_polyhedron::EXCLUDED);
    Nef_polyhedron N3 = N1 * N2;
    return 0;
}
```

CGAL::Nef_polyhedron_S2<Traits>::Sphere_point

Definition

An object p of type *Sphere_point*< R > is a point on the surface of a unit sphere. Such points correspond to the nontrivial directions in space and similarly to the equivalence classes of all nontrivial vectors under normalization.

Types

Sphere_point:: RT ring number type.

Creation

Sphere_point p ; creates some sphere point.

Sphere_point $p(RT\ x, RT\ y, RT\ z)$; creates a sphere point corresponding to the point of intersection of the ray starting at the origin in direction (x, y, z) and the surface of S_2 .

Operations

Access to the coordinates is provided by the following operations. Note that the vector (x, y, z) is not normalized.

RT $p.x()$ the x -coordinate.

RT $p.y()$ the y -coordinate.

RT $p.z()$ the z -coordinate.

$bool$ $p == q$ Equality.

$bool$ $p != q$ Inequality.

Sphere_point $p.antipode()$ returns the antipode of p .

CGAL::Nef_polyhedron_S2<Traits>::Sphere_segment

Definition

An object s of type *Sphere_segment* is a segment in the surface of a unit sphere that is part of a great circle through the origin. Sphere segments are represented by two sphere points p and q plus an oriented plane h that contains p and q . The plane determines the sphere segment as follows. Let c be the circle in the intersection of h and S_2 . Then s is that part of c that is swept, when we rotate p into q in counterclockwise rotation around the normal vector of h as seen from the positive halfspace.

Creation

Sphere_segment s ; creates some sphere segment.

Sphere_segment s (*Sphere_point* $p1$, *Sphere_point* $p2$, *bool* *shorter_arc*=*true*);

creates a spherical segment spanning the shorter arc from $p1$ to $p2$ if *shorter_arc* == *true*. Otherwise the longer arc is created.

Precondition: $p1 \neq p2$ and $p1 \neq p2.opposite()$.

Sphere_segment s (*Sphere_point* $p1$, *Sphere_point* $p2$, *Sphere_circle* c);

creates a spherical segment spanning the arc from $p1$ to $p2$ as part of the oriented circle c ($p1 == p2$ or $p1 == p2.opposite()$ are possible.)

Precondition: $p1$ and $p2$ are contained in c .

Sphere_segment s (*Sphere_circle* $c1$, *Sphere_circle* $c2$);

creates the spherical segment as part of $c1$ that is part of the halfsphere left of the oriented circle $c2$.

Precondition: $c1 \neq c2$ as unoriented circles.

Operations

Sphere_point $s.source()$ the source point of s .

Sphere_point $s.target()$ the target point of s .

Sphere_circle $s.sphere_circle()$ the great circle supporting s .

Sphere_segment $s.opposite()$ returns the spherical segment oriented from $target()$ to $source()$ with the same point set as s .

Sphere_segment $s.complement()$ returns the spherical segment oriented from $target()$ to $source()$ with the point set completing s to a full circle.

<i>bool</i>	<i>s.is_short()</i>	a segment is short iff it is shorter than a half-circle.
<i>bool</i>	<i>s.is_long()</i>	a segment is long iff it is longer than a half-circle.
<i>bool</i>	<i>s.is_degenerate()</i>	return true iff <i>s</i> is degenerate, i.e. source and target are the same.
<i>bool</i>	<i>s.is_halfcircle()</i>	return true iff <i>s</i> is a perfect half-circle, i.e. <i>source().antipode</i> == <i>target()</i> .
<i>bool</i>	<i>s.has_on(Sphere_point p)</i>	 return true iff <i>s</i> contains <i>p</i> .
<i>bool</i>	<i>s.has_in_relative_interior(Sphere_point p)</i>	 return true iff <i>s</i> contains <i>p</i> in its relative interior.

CGAL::Nef_polyhedron_S2<Traits>::Sphere_circle

Definition

An object c of type *Sphere_circle* is an oriented great circle on the surface of a unit sphere. Such circles correspond to the intersection of an oriented plane (that contains the origin) and the surface of S_2 . The orientation of the great circle is that of a counterclockwise walk along the circle as seen from the positive halfspace of the oriented plane.

Types

Sphere_circle::RT ring type.

Sphere_circle::Plane_3 plane a *Sphere_circle* lies in.

Creation

Sphere_circle c ; creates some great circle.

Sphere_circle c (*Sphere_point* p , *Sphere_point* q);

If p and q are opposite of each other, then we create the unique great circle on S_2 which contains p and q . This circle is oriented such that a walk along c meets p just before the shorter segment between p and q . If p and q are opposite of each other then we create any great circle that contains p and q .

Sphere_circle c (*Plane_3* h); creates the circle corresponding to the plane h .
Precondition: h contains the origin.

Sphere_circle c (*RT* x , *RT* y , *RT* z); creates the circle orthogonal to the vector (x, y, z) .

Sphere_circle c (*Sphere_circle* c , *Sphere_point* p);

creates a great circle orthogonal to c that contains p .
Precondition: p is not part of c .

Operations

Sphere_circle c .*opposite()* Returns a sphere circle in the opposite direction of c .

bool c .*has_on*(*Sphere_point* p) returns true iff c contains p .

Plane_3 c .*plane*() returns the plane supporting c .

Sphere_point c .*orthogonal_pole*() returns the point that is the pole of the hemisphere left of c .

Global functions

bool *equal_as_sets(const c1, const c2)*

returns true iff *c1* and *c2* are equal as unoriented circles.

CGAL::Nef_polyhedron_S2<Traits>::SVertex

Definition

The figure on page [1003](#) illustrate the incidence of a svertex on a sphere map.

The member function *out_sedge* returns the first outgoing shalfedge, and *incident_sface* returns the incident sface.

```
#include <CGAL/Nef_polyhedron_S2.h>
```

Types

The following types are the same as in *Nef_polyhedron_S2<Traits>*.

<i>SVertex:: Mark</i>	type of mark.
<i>SVertex:: Sphere_point</i>	sphere point type stored in SVertex.
<i>SVertex:: SVertex_const_handle</i>	const handle to SVertex.
<i>SVertex:: SHalfedge_const_handle</i>	const handle to SHalfedge.
<i>SVertex:: SFace_const_handle</i>	const handle to SFace.

Creation

There is no need for a user to create a *SVertex* explicitly. The class *Nef_polyhedron_S2<Traits>* manages the needed svertices internally.

Operations

<i>Mark</i>	<i>e.mark()</i>	the mark of <i>e</i> .
<i>Sphere_point</i>	<i>e.point()</i>	the sphere point of <i>e</i> .
<i>bool</i>	<i>e.is_isolated()</i>	returns —true— if <i>e</i> has no adjacent sedges.
<i>SVertex_const_handle</i>	<i>e.twin()</i>	the twin of <i>e</i> .
<i>SHalfedge_const_handle</i>	<i>e.out_sedge()</i>	the first out sedge of <i>e</i> .
<i>SFace_const_handle</i>	<i>e.incident_sface()</i>	the incident sface of <i>e</i> .

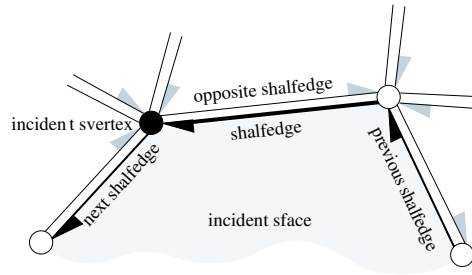
See Also

CGAL::Nef_polyhedron_S2<Traits>::SHalfedge page [1003](#)
CGAL::Nef_polyhedron_S2<Traits>::SFace page [1007](#)
CGAL::Nef_polyhedron_S2<Traits>::Sphere_point page [997](#)

CGAL::Nef_polyhedron_S2<Traits>::SHalfedge

Definition

A shalfedge is a great arc on a sphere map. The figure below depicts the relationship between a shalfedge and its incident shalfedges, svertices, and sfaces on a sphere map. A shalfedge is an oriented sedge between two svertices. It is always paired with a shalfedge pointing in the opposite direction. The *twin()* member function returns this shalfedge of opposite orientation.



The *snext()* member function points to the successor shalfedge around this sface while the *sprev()* member function points to the preceding shalfedge. An successive assignments of the form *se = se->snext()* cycles counterclockwise around the sface (or hole).

Similarly, the successive assignments of the form *se = se->snext()->twin()* cycle clockwise around the svertex and traverse all halfedges incident to this svertex. The assignment *se = se->cyclic_adj_succ()* can be used as a shortcut.

A const circulator is provided for each of the two circular orders. The circulators are bidirectional and assignable to *SHalfedge_const_handle*.

```
#include <CGAL/Nef_polyhedron_S2.h>
```

Types

The following types are the same as in *Nef_polyhedron_S2<Traits>*.

<i>SHalfedge:: Mark</i>	type of mark.
<i>SHalfedge:: Sphere_circle</i>	sphere circle type stored in SHalfedge.
<i>SHalfedge:: SVertex_const_handle</i>	const handle to SVertex.
<i>SHalfedge:: SHalfedge_const_handle</i>	const handle to SHalfedge.
<i>SHalfedge:: SFace_const_handle</i>	const handle to SFace.

Creation

There is no need for a user to create a *SHalfedge* explicitly. The class *Nef_polyhedron_S2<Traits>* manages the needed shalfedges internally.

Operations

<i>Mark</i>	<i>se.mark()</i>	the mark of <i>se</i> .
<i>Sphere_circle</i>	<i>se.circle()</i>	the sphere circle of <i>se</i> .
<i>SHalfedge_const_handle</i>	<i>se.twin()</i>	the twin of <i>se</i> .
<i>SVertex_const_handle</i>	<i>se.source()</i>	the source svertex of <i>se</i> .
<i>SVertex_const_handle</i>	<i>se.target()</i>	equals <i>twin()</i> -> <i>source()</i> .
<i>SHalfedge_const_handle</i>	<i>se.sprev()</i>	the SHalfedge previous to <i>se</i> in a sface cycle.
<i>SHalfedge_const_handle</i>	<i>se.snext()</i>	the next SHalfedge of <i>se</i> in a sface cycle.
<i>SHalfedge_const_handle</i>	<i>se.cyclic_adj_pred()</i>	the edge before <i>se</i> in the cyclic ordered adjacency list of <i>source()</i> .
<i>SHalfedge_const_handle</i>	<i>se.cyclic_adj_succ()</i>	the edge after <i>se</i> in the cyclic ordered adjacency list of <i>source()</i> .
<i>SFace_const_handle</i>	<i>se.incident_sface()</i>	the incident sface of <i>se</i> .

See Also

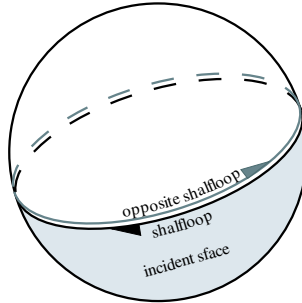
CGAL::Nef_polyhedron_S2<Traits>::SVertex page [1002](#)
CGAL::Nef_polyhedron_S2<Traits>::SFace page [1007](#)
CGAL::Nef_polyhedron_S2<Traits>::Sphere_circle page [1000](#)

CGAL::Nef_polyhedron_S2<Traits>::SHalfloop

Definition

A sloop is a great circle on a sphere. A shalfloop is an oriented sloop. It is always paired with a shalfloop whose supporting *Sphere_circle* is pointing in the opposite direction. The *twin()* member function returns this shalfloop of opposite orientation. Each *Nef_polyhedron_S2* can only have one sloop (resp. two shalfloops).

The figure below depicts the relationship between a shalfloop and sfaces on a sphere map.



```
#include <CGAL/Nef_polyhedron_S2.h>
```

Types

The following types are the same as in *Nef_polyhedron_S2<Traits>*.

<i>SHalfloop:: Mark</i>	type of mark.
<i>SHalfloop:: Sphere_circle</i>	sphere circle type stored in SHalfloop.
<i>SHalfloop:: SHalfloop_const_handle</i>	const handle to SHalfloop.
<i>SHalfloop:: SFace_const_handle</i>	const handle to SFace.

Creation

There is no need for a user to create a *SHalfloop* explicitly. The class *Nef_polyhedron_S2<Traits>* manages the needed shalfloops internally.

Operations

<i>Mark</i>	<i>se.mark()</i>	the mark of <i>se</i> .
<i>Sphere_circle</i>	<i>se.circle()</i>	the sphere circle of <i>se</i> .
<i>SHalfloop_const_handle</i>	<i>se.twin()</i>	the twin of <i>se</i> .
<i>SFace_const_handle</i>	<i>se.incident_sface()</i>	the incident sface of <i>se</i> .

See Also

CGAL::Nef_polyhedron_S2<Traits>::SFace page [1007](#)
CGAL::Nef_polyhedron_S2<Traits>::Sphere_circle page [1000](#)

CGAL::Nef_polyhedron_S2<Traits>::SFace

Definition

Figure 14.4 on page 1003 and figure 14.4 on page 1005 illustrate the incidences of an sface. An sface is described by its boundaries. An entry item to each boundary cycle can be accessed using the iterator range (*sface_cycles_begin()/sface_cycles_end()*). Additionally, *Nef_polyhedron_S2* provides the macro *CGAL_forall_sface_cycles_of*. The iterators are of type *SFace_cycle_const_iterator* and represent either a shalfedge, a shalfloop, or a svertex.

```
#include <CGAL/Nef_polyhedron_S2.h>
```

Types

The following types are the same as in *Nef_polyhedron_S2<Traits>*.

<i>SFace:: Mark</i>	type of mark.
<i>SFace:: Object_list</i>	list of Object handles.
<i>SFace:: Vertex_const_handle</i>	const handle to Vertex.
<i>SFace:: Volume_const_handle</i>	const handle to Volume.
<i>SFace:: SFace_const_handle</i>	const handle to SFace.
<i>SFace:: SFace_cycle_const_iterator</i>	const iterator over the entries to all sface cycles of a sface.

Creation

There is no need for a user to create a *SFace* explicitly. The class *Nef_polyhedron_S2<Traits>* manages the needed sfaces internally.

Operations

<i>Mark</i>	<i>sf.mark()</i>	the mark of <i>sf</i> .
<i>SFace_cycle_const_iterator</i>	<i>sf.sface_cycle_begin()</i>	iterator over the entries to all sface cycles of <i>sf</i> .
<i>SFace_cycle_const_iterator</i>	<i>sf.sface_cycle_end()</i>	past-the-end iterator.

See Also

CGAL::Nef_polyhedron_S2<Traits> page 991
CGAL::Nef_polyhedron_S2<Traits>::SVertex page 1002

CGAL::Nef_polyhedron_S2<Traits>::SFace_cycle_iterator

Definition

The type *SFace_cycle_iterator* iterates over a list of *Object_handles*. Each item of that list can either be assigned to *SVertex_handle*, *SHalfedge_handle* or *SHalfloop_handle*. To find out which of these assignment works out, the member functions *is_svertex()*, *is_shalfedge()* and *is_shalfloop()* are provided.

```
#include <CGAL/Nef_polyhedron_S2.h>
```

Types

<i>SFace_cycle_iterator::SVertex_handle</i>	const handle to <i>SVertex</i> .
<i>SFace_cycle_iterator::SHalfedge_handle</i>	const handle to <i>SHalfedge</i> .
<i>SFace_cycle_iterator::SHalfloop_handle</i>	const handle to <i>SHalfloop</i> .

Creation

<i>SFace_cycle_iterator sfc;</i>	default constructor.
----------------------------------	----------------------

Operations

<i>bool</i>	<i>sfc.is_svertex()</i>	returns true if <i>sfc</i> represents a <i>SVertex_handle</i> .
<i>bool</i>	<i>sfc.is_shalfedge()</i>	returns true if <i>sfc</i> represents a <i>SHalfedge_handle</i> .
<i>bool</i>	<i>sfc.is_shalfloop()</i>	returns true if <i>sfc</i> represents a <i>SHalfloop_handle</i> .
<i>SVertex_handle</i>	<i>SVertex_handle(sfc)</i>	casts <i>sfc</i> to <i>SVertex_handle</i> .
<i>SHalfedge_handle</i>	<i>SHalfedge_handle(sfc)</i>	casts <i>sfc</i> to <i>SHalfedge_handle</i> .
<i>SHalfloop_handle</i>	<i>SHalfloop_handle(sfc)</i>	casts <i>sfc</i> to <i>SHalfloop_handle</i> .

See Also

<i>CGAL::Nef_polyhedron_S2<Traits>::SVertex</i>	page 1002
<i>CGAL::Nef_polyhedron_S2<Traits>::SHalfedge</i>	page 1003
<i>CGAL::Nef_polyhedron_S2<Traits>::SHalfloop</i>	page 1005

CGAL::Qt_widget_Nef_S2<Nef_polyhedron_S2>

Definition

The class *Qt_widget_Nef_S2* uses the OpenGL interface of Qt in order to display a *Nef_polyhedron_S2*. Its purpose is to provide an easy to use viewer for *Nef_polyhedron_S2*. There are no means provided to enhance the functionality of the viewer.

In addition to the functions inherited from the Qt class *QGLWidget*, *Qt_widget_Nef_S2* only has a single public constructor. For the usage of *Qt_widget_Nef_S2* see the example below.

```
#include <CGAL/IO/Qt_widget_Nef_S2.h>
```

Parameters

The template parameter expects an instantiation of *Nef_polyhedron_S2<Traits>*.

Creation

```
Qt_widget_Nef_S2<Nef_polyhedron_S2> W( Nef_polyhedron_S2 N);
```

Creates a widget *W* for displaying the *Nef_polyhedron_S2* *N*.

See Also

CGAL::Nef_polyhedron_S2<Traits> page [991](#)

Example

This example create some random *Sphere_segments* and distributes them on two *Nef_polyhedron_2*. The two Nef polyhedra are combined by a symmetric difference and the result is displayed in a Qt widget.

```
// Copyright (c) 2004 Max-Planck-Institute Saarbruecken (Germany).
// All rights reserved.
//
// This file is part of CGAL (www.cgal.org); you may redistribute it under
// the terms of the Q Public License version 1.0.
// See the file LICENSE.QPL distributed with CGAL.
//
// Licensees holding a valid commercial license may use this file in
// accordance with the commercial license agreement provided with the software.
//
// This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
// WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
//
// $URL: svn+ssh://scm.gforge.inria.fr/svn/cgal/branches/CGAL-3.2-branch/Nef_S2/demo/Nef_S2/visualization
// $Id: visualization.C 29613 2006-03-19 19:35:17Z spion $
```

```

//
//
// Author(s)      : Peter Hachenberger <hachenberger@mpi-sb.mpg.de>

#include <CGAL/basic.h>

#ifdef CGAL_USE_QT
#include <iostream>
int main(int, char*){
    std::cout << "Sorry, this demo needs QT..." << std::endl; return 0;}
#else
#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_S2.h>
#include <CGAL/Nef_S2/create_random_Nef_S2.h>
#include <CGAL/IO/Qt_widget_Nef_S2.h>
#include <qapplication.h>

typedef CGAL::Gmpz RT;
typedef CGAL::Homogeneous<RT> Kernel;
typedef CGAL::Nef_polyhedron_S2<Kernel> Nef_polyhedron_S2;

int main(int argc, char* argv[]) {

    Nef_polyhedron_S2 S;
    create_random_Nef_S2(S,5);

    QApplication a(argc, argv);
    CGAL::Qt_widget_Nef_S2<Nef_polyhedron_S2>* w =
        new CGAL::Qt_widget_Nef_S2<Nef_polyhedron_S2>(S);
    a.setMainWidget(w);
    w->show();
    return a.exec();
}
#endif

```

Chapter 15

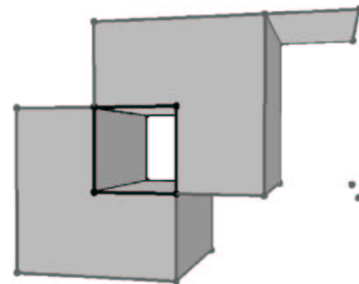
3D Boolean Operations on Nef Polyhedra

Peter Hachenberger and Lutz Kettner

Contents

15.1 Introduction	1011
15.2 Definition	1012
15.3 Infimal Box	1015
15.4 Regularized Set Operations	1015
15.5 Example Programs	1016
15.5.1 Construction and Comparison	1016
15.5.2 Point Set Operations	1017
15.5.3 Transformation	1018
15.5.4 The Interface between <i>Polyhedron_3</i> and <i>Nef_polyhedron_3</i>	1019
15.5.5 Using an Extended Kernel	1020
15.6 File I/O	1021
15.7 Further Example Programs	1022
15.7.1 Exploring a Sphere Map	1022
15.7.2 Exploring Shells	1023
15.7.3 Point Location	1025
15.8 Visualization	1026
15.8.1 Visualizing a 3D Nef polyhedron	1027
15.8.2 Visualizing a Sphere Map	1028

15.1 Introduction



In solid modeling, two major representation schemes are used: *constructive solid geometry* (CSG) and *boundary representations* (B-rep). Both have inherent strengths and weaknesses, see [Hof89c] for a discussion.

In CSG a solid is represented as a set-theoretic boolean combination of primitive solid objects, such as blocks, prisms, cylinders, or toruses. The boolean operations are not evaluated, instead, objects are represented implicitly with a tree structure; leaves represent primitive objects and interior nodes represent boolean operations or rigid motions, e.g., translation and rotation. Algorithms on such a CSG-tree first evaluate properties on the primitive objects and propagate the results using the tree structure.

A B-rep describes the incidence structure and the geometric properties of all lower-dimensional features of the boundary of a solid. Surfaces are oriented to decide between the interior and exterior of a solid.

The class of representable objects in a CSG is usually limited by the choice of the primitive solids. A B-rep is usually limited by the choice for the geometry of the supporting curves for edges and the supporting surfaces for surface patches, and, in addition, the connectivity structure that is allowed. In particular, a B-rep is not always closed under boolean set operations. As an example, the class of orientable 2-manifold objects is a popular and well understood class of surfaces commonly used for B-reps. They can be represented and manipulated efficiently, the data structures are compact in storage size, and many algorithms are simple. On the other side, this object class is not closed under boolean set operations, as many examples can illustrate, such as the Figure shown above that can be generated using boolean set operations on cubes. The vertices bounding the tunnel, or the edge connecting the “roof” with the cube are non-manifold situations.

In our implementation of Nef polyhedra in 3D, we offer a B-rep data structure that is closed under boolean operations and with all their generality. Starting from halfspaces (and also directly from oriented 2-manifolds), we can work with set union, set intersection, set difference, set complement, interior, exterior, boundary, closure, and regularization operations (see Section 15.4 for an explanation of regularized set operations). In essence, we can evaluate a CSG-tree with halfspaces as primitives and convert it into a B-rep representation.

In fact, we work with two data structures; one that represents the local neighborhoods of vertices, which is in itself already a complete description, and a data structure that connects these neighborhoods up to a global data structure with edges, facets, and volumes. We offer a rich interface to investigate these data structures, their different elements and their connectivity. We provide affine (rigid) transformations and a point location query operation. We have a custom file format for storing and reading Nef polyhedra from files. We offer a simple OpenGL-based visualizer for debugging and illustrations.

15.2 Definition

The theory of Nef polyhedra has been developed for arbitrary dimensions. The class `CGAL::Nef_polyhedron_3` implements a boundary representation for the 3-dimensional case.

Definition: A *Nef-polyhedron* in dimension d is a point set $P \subseteq \mathbb{R}^d$ generated from a finite number of open halfspaces by set complement and set intersection operations.

Set union, difference and symmetric difference can be reduced to intersection and complement. Set complement changes between open and closed halfspaces, thus the topological operations *boundary*, *interior*, *exterior*, *closure* and *regularization* are also in the modeling space of Nef polyhedra.

A face of a Nef polyhedron is defined as an equivalence class of *local pyramids* that are a characterization of the local space around a point.

Definition: A point set $K \subseteq \mathbb{R}^d$ is called a *cone with apex 0*, if $K = \mathbb{R}^+ K$ (i.e., $\forall p \in K, \forall \lambda > 0 : \lambda p \in K$) and it is called a *cone with apex x* , $x \in \mathbb{R}^d$, if $K = x + \mathbb{R}^+(K - x)$. A cone K is called a *pyramid* if K is a polyhedron.

Now let $P \in \mathbb{R}^d$ be a polyhedron and $x \in \mathbb{R}^d$. There is a neighborhood $U_0(x)$ of x such that the pyramid

$Q := x + \mathbb{R}^+((P \cap U(x)) - x)$ is the same for all neighborhoods $U(x) \subseteq U_0(x)$. Q is called the *local pyramid* of P in x and denoted $\text{Pyr}_P(x)$.

Definition: Let $P \in \mathbb{R}^d$ be a polyhedron and $x, y \in \mathbb{R}^d$ be two points. We define an equivalence relation $x \sim y$ iff $\text{Pyr}_P(x) = \text{Pyr}_P(y)$. The equivalence classes of \sim are the *faces* of P . The dimension of a face s is the dimension of its affine hull, $\dim s := \dim \text{aff } s$.

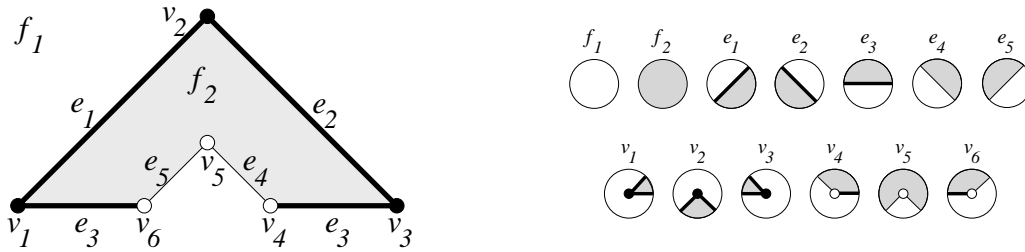
In other words, a *face* s of P is a maximal non-empty subset of \mathbb{R}^d such that all of its points have the same local pyramid Q denoted $\text{Pyr}_P(s)$. This definition of a face partitions \mathbb{R}^d into faces of different dimension. A face s is either a subset of P , or disjoint from P . We use this later in our data structure and store a selection mark in each face indicating its set membership.

Faces do not have to be connected. There are only two full-dimensional faces possible, one whose local pyramid is the space \mathbb{R}^d itself and the other with the empty set as a local pyramid. All lower-dimensional faces form the *boundary* of the polyhedron. As usual, we call zero-dimensional faces *vertices* and one-dimensional faces *edges*. In the case of polyhedra in space we call two-dimensional faces *facets* and the full-dimensional faces *volumes*. Faces are *relative open* sets, e.g., an edge does not contain its end-vertices.

We illustrate the definitions with an example in the plane. Given the closed halfspaces

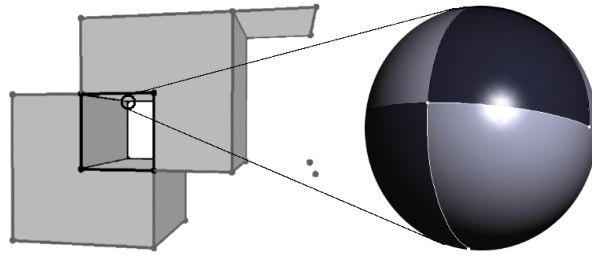
$$h_1 : y \geq 0, \quad h_2 : x - y \geq 0, \quad h_3 : x + y \leq 3, \quad h_4 : x - y \geq 1, \quad h_5 : x + y \leq 2,$$

we define our polyhedron $P := (h_1 \cap h_2 \cap h_3) - (h_4 \cap h_5)$.

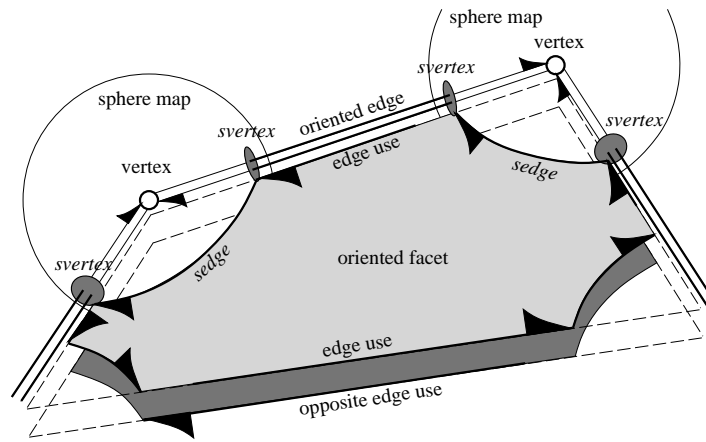


The left figure illustrates the polyhedron with its partially closed and partially open boundary, i.e., vertex v_4, v_5, v_6 , and edges e_4 and e_5 are not part of P . The local pyramids for the faces are $\text{Pyr}_P(f_1) = \emptyset$ and $\text{Pyr}_P(f_2) = \mathbb{R}^2$. Examples for the local pyramids of edges are the closed halfspace h_2 for the edge e_1 , $\text{Pyr}_P(e_1) = h_2$, and the open halfspace that is the complement of h_4 for the edge e_5 , $\text{Pyr}_P(e_5) = \{(x, y) | x - y < 1\}$. The edge e_3 consists actually of two disconnected parts, both with the same local pyramid $\text{Pyr}_P(e_3) = h_1$. In our data structure, we will represent the two connected components of the edge e_3 separately. The figure on the right lists all local pyramids for this example.

The local pyramids of each vertex are represented by conceptually intersecting the local neighborhood with a small ε -sphere. This intersection forms a planar map on the sphere (see next two figures), which together with the set-selection mark for each item (i.e. vertices, edges, loops and faces) forms a two-dimensional Nef polyhedron embedded in the sphere. We add the set-selection mark for the vertex and call the resulting structure the *sphere map* of the vertex. We use the prefix s to distinguish the elements of the sphere map from the three-dimensional elements. See Chapter 14 for further details.



Having sphere maps for all vertices of our polyhedron is a sufficient but not easily accessible representation of the polyhedron. We enrich the data structure with more explicit representations of all the faces and incidences between them.



We depart slightly from the definition of faces in a Nef polyhedron; we represent the connected components of a face individually and do not implement additional bookkeeping to recover the original faces (e.g., all edges on a common supporting line with the same local pyramid) as this is not needed in our algorithms. We discuss features in the increasing order of dimension.

edges: We store two oppositely oriented edges for each edge and have a pointer from one oriented edge to its opposite edge. Such an oriented edge can be identified with an *svertex* in a sphere map; it remains to link one *svertex* with the corresponding opposite *svertex* in the other sphere map.

edge uses: An edge can have many incident facets (non-manifold situation). We introduce two oppositely oriented edge-uses for each incident facet; one for each orientation of the facet. An edge-use points to its corresponding oriented edge and to its oriented facet. We can identify an edge-use with an oriented *sedge* in the sphere map, or, in the special case also with an *sloop*. Without mentioning it explicitly in the remainder, all references to *sedge* can also refer to *sloop*.

facets: We store oriented facets as boundary cycles of oriented edge-uses. We have a distinguished outer boundary cycle and several (or maybe none) inner boundary cycles representing holes in the facet. Boundary cycles are linked in one direction. We can access the other traversal direction when we switch to the oppositely oriented facet, i.e., by using the opposite edge-use.

shells: The volume boundary decomposes into different connected components, the *shells*. A shell consists of a connected set of facets, edges, and vertices incident to this volume. Facets around an edge form a radial order that is captured in the radial order of *sedges* around an *svertex* in the sphere map. Using this information, we can trace a shell from one entry element with a graph search. We offer this graph traversal (to the user) in a visitor design pattern.

volumes: A volume is defined by a set of shells, one outer shell containing the volume and several (or maybe none) inner shells separating voids which are excluded from the volume.

For each face we store a label, e.g., a set-selection mark, which indicates whether the face is part of the solid or if it is excluded. We call the resulting data structure *Selective Nef Complex*, *SNC* for short [GHH⁺03]. However, in CGAL we identify the names and call the *SNC* data structure *CGAL::Nef_polyhedron_3*.

15.3 Infimaximal Box

We call a Nef polyhedron *bounded* if its boundary is bounded, i.e., finite, and *unbounded* otherwise. Note that unbounded point sets can have a bounded boundary, for example, the complement of a cube has an unbounded outer volume, but its boundary remains bounded.

Using a boundary representation, it is convenient (conceptually and in our implementation) to consider bounded Nef polyhedra only. Bounded Nef polyhedra are also closed under boolean set operations. However, one needs to start with bounded primitives; the conceptually nice halfspaces cannot be used. Instead, we offer a construction from oriented 2-manifolds represented in a *CGAL::Polyhedron*, see Section 15.5.4 below.

In order to handle unbounded Nef polyhedra conceptually in the same way as we handle bounded Nef polyhedra, we intersect them with a bounding cubical volume of size $[-R, R]^3$, where R is a symbolical unspecified value, which is finite but larger than all coordinate values that may occur in the bounded part of the polyhedron. As a result, each Nef polyhedron becomes bounded. We call the boundary of the bounding volume the *infimaximal box* [SM00].

We clip lines and rays at the infimaximal box. The intersection points with the infimaximal box are called *non-standard points*, which are points whose coordinates are $-R$ or R in at least one dimension, and linear functions $f(R)$ for the other dimensions. Such extended points (and developed from there also extended segments etc) are provided in CGAL with extended kernels—*CGAL::Extended_cartesian* and *CGAL::Extended_homogeneous*. They are regular CGAL kernels with a polynomial type as coordinate number type.

As long as an extended kernel is used, the full functionality provided by the *CGAL::Nef_polyhedron_3* class is available. If a kernel that does not use polynomials to represent coordinates is used, it is not possible to create or load unbounded Nef polyhedra, but all other operations work as expected. We provided both possibilities, since the restriction to bounded Nef polyhedra improves considerably space requirements (plain number type instead of polynomial), and runtime performance.

15.4 Regularized Set Operations

Since manifolds are not closed under boolean operations, Requicha proposes to use *regularized set operations* [KM76, Req80]. A set is *regular*, if it equals the closure of its interior. A regularized set operation is defined as the standard set operation followed by a regularization of the result. Regularized sets are closed under regularized set operations.

Regularized set operations are important since they simplify the class of solids to exclude lower dimensional features and the boundary belongs to the point set. These properties are considered to reflect the nature of physical solids more closely.

Regularized polyhedral sets are a subclass of Nef polyhedra. We provide the *regularization* operation as a shortcut for the consecutive execution of the *interior* and the *closure* operations.

15.5 Example Programs

The following example gives a first impression of how to instantiate and use *Nef_polyhedron_3*. We use the *CGAL::Cartesian* kernel. All Cartesian and homogeneous kernels of CGAL are suitable if the number type parameter follows the usual requirements of being a model of the *CGAL::FieldNumberType* concept for the Cartesian kernels, or the *CGAL::RingNumberType* concept for the homogeneous kernels, respectively. Note however, that in the current state, the Nef polyhedron works only with CGAL kernels. The implementation makes use of CGAL specific functions in kernel objects, and does not yet offer a designed interface to a clean kernel concept that could be offered by an external kernel as well.

The example creates two Nef polyhedra—*N0* is the empty set, while *N1* represents the full space, i.e., the set of all points in the 3-dimensional space. The assertion assures that the empty set is the complement of the full space.

```
// file: examples/Nef_3/simple.C

#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>

typedef CGAL::Homogeneous<CGAL::Gmpz> Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron;

int main() {
    Nef_polyhedron N0(Nef_polyhedron::EMPTY);
    Nef_polyhedron N1(Nef_polyhedron::COMPLETE);

    CGAL_assertion (N0 == N1.complement());
    return 0;
}
```

15.5.1 Construction and Comparison

This example shows the various constructors. We can create the empty set, which is also the default constructor, and the full space, i.e. all points of \mathbb{R}^3 belong to the polyhedron. We can create a halfspace defined by a plane bounding it. It is only available if an extended kernel is used. The halfspace constructor has a second parameter that specifies whether the defining plane belongs to the point set (*Nef_polyhedron::INCLUDED*) or not (*Nef_polyhedron::EXCLUDED*). The default value is *Nef_polyhedron::INCLUDED*. Additionally, we can create a *Nef_polyhedron_3* from a *Polyhedron_3*, see the Section 15.5.4 below.

We can compute the point sets of two Nef polyhedra for equality and proper subset relationships. We offer the usual comparison operators `==`, `!=`, `<=`, `>=`, `<` and `>`.

Nef polyhedra have the important feature that a representation that is called the *reduced Würzburg structure* is unique, i.e., two point sets of Nef polyhedra are equal if and only if the representations are equal. The proof for the reduced Würzburg structure carries over to our representation and the comparison operators are therefore trivial to implement.

```
// file: examples/Nef_3/construction.C

#include <CGAL/Gmpz.h>
```



```

#include <CGAL/Extended_homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>

typedef CGAL::Extended_homogeneous<CGAL::Gmpz> Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron;
typedef Nef_polyhedron::Plane_3 Plane_3;

int main() {
    Nef_polyhedron N0;
    Nef_polyhedron N1(Nef_polyhedron::EMPTY);
    Nef_polyhedron N2(Nef_polyhedron::COMPLETE);
    Nef_polyhedron N3(Plane_3( 1, 2, 5,-1));
    Nef_polyhedron N4(Plane_3( 1, 2, 5,-1), Nef_polyhedron::INCLUDED);
    Nef_polyhedron N5(Plane_3( 1, 2, 5,-1), Nef_polyhedron::EXCLUDED);

    CGAL_assertion(N0 == N1);
    CGAL_assertion(N3 == N4);
    CGAL_assertion(N0 != N2);
    CGAL_assertion(N3 != N5);

    CGAL_assertion(N4 >= N5);
    CGAL_assertion(N5 <= N4);
    CGAL_assertion(N4 > N5);
    CGAL_assertion(N5 < N4);

    N5 = N5.closure();
    CGAL_assertion(N4 >= N5);
    CGAL_assertion(N4 <= N5);

    return 0;
}

```

15.5.2 Point Set Operations

As explained in the introduction, Nef polyhedra are closed under all boolean set operations. The class *Nef_polyhedron_3* provides functions and operators for the most common ones: complement (*operator!*), union (*operator+*), difference (*operator-*), intersection (*operator**) and symmetric difference (*operator^*). Additionally, the operators **=*, *-=*, **=* and *^=* are defined.

Nef_polyhedron_3 also provides the topological operations *interior()*, *closure()* and *boundary()*. With *interior()* one deselects all boundary items, with *boundary()* one deselects all volumes, and with *closure()* one selects all boundary items.

```

// file: examples/Nef_3/point_set_operations.C

#include <CGAL/Gmpz.h>
#include <CGAL/Extended_homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>

typedef CGAL::Extended_homogeneous<CGAL::Gmpz> Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron;
typedef Kernel::Plane_3 Plane_3;

```

```

int main() {

    Nef_polyhedron N1(Plane_3( 1, 0, 0,-1));
    Nef_polyhedron N2(Plane_3(-1, 0, 0,-1));
    Nef_polyhedron N3(Plane_3( 0, 1, 0,-1));
    Nef_polyhedron N4(Plane_3( 0,-1, 0,-1));
    Nef_polyhedron N5(Plane_3( 0, 0, 1,-1));
    Nef_polyhedron N6(Plane_3( 0, 0,-1,-1));

    Nef_polyhedron I1(!N1 + !N2); // open slice in yz-plane
    Nef_polyhedron I2(N3 - !N4); // closed slice in xz-plane
    Nef_polyhedron I3(N5 ^ N6); // open slice in yz-plane
    Nef_polyhedron Cube1(I2 * !I1);
    Cube1 *= !I3;
    Nef_polyhedron Cube2 = N1 * N2 * N3 * N4 * N5 * N6;

    CGAL_assertion(Cube1 == Cube2); // both are closed cube
    CGAL_assertion(Cube1 == Cube1.closure());
    CGAL_assertion(Cube1 == Cube1.regularization());
    CGAL_assertion((N1 - N1.boundary()) == N1.interior());
    CGAL_assertion(I1.closure() == I1.complement().interior().complement());
    CGAL_assertion(I1.regularization() == I1.interior().closure());
    return 0;
}

```

15.5.3 Transformation

Using the *std::transform* function, a Nef polyhedron can be translated, rotated and scaled. The usage is shown in the following example:

```

// examples/Nef_3/transformation.C

#include <CGAL/Gmpz.h>
#include <CGAL/Extended_homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>

typedef CGAL::Extended_homogeneous<CGAL::Gmpz> Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron;
typedef Nef_polyhedron::Plane_3 Plane_3;
typedef Nef_polyhedron::Vector_3 Vector_3;
typedef Nef_polyhedron::Aff_transformation_3 Aff_transformation_3;

int main() {

    Nef_polyhedron N(Plane_3(0,1,0,0));
    Aff_transformation_3 transl(CGAL::TRANSLATION, Vector_3(5, 7, 9));
    Aff_transformation_3 rotx90(1,0,0,
        0,0,-1,
        0,1,0,
        1);
}

```

```

Aff_transformation_3 scale(3,0,0,
    0,3,0,
    0,0,3,
    2);

N.transform(transl);
CGAL_assertion(N == Nef_polyhedron(Plane_3(0,1,0,-7)));
N.transform(rotx90);
CGAL_assertion(N == Nef_polyhedron(Plane_3(0,0,1,-7)));
N.transform(scale);
CGAL_assertion(N == Nef_polyhedron(Plane_3(0,0,2,-21)));

return 0;
}

```

15.5.4 The Interface between *Polyhedron_3* and *Nef_polyhedron_3*

Nef_polyhedron_3 provides an interface for the conversion between polyhedral surfaces represented with the *CGAL::Polyhedron_3* class and *Nef_polyhedron_3*. *Polyhedron_3* represents orientable 2-manifold objects with boundaries. However, we exclude surfaces with boundaries from the conversion to *Nef_polyhedron_3* since they have no properly defined volume.

Both conversion directions can only be performed if the boundary of the point set is an oriented closed 2-manifold. *Nef_polyhedron_3* provides the function *is_simple()* and *Polyhedron_3* provides the function *is_closed()* to test for this property. The usage is illustrated by the example program below.

The conversion gives us the possibility to use several file formats. *Polyhedron_3* can read the (.off) file format and can write the (.off), OpenInventor (.iv), VRML 1.0 and 2.0 (.wrl) and Wavefront Advanced Visualizer object format (.obj), see Section 10.4.

```

// file: examples/Nef_3/interface_polyhedron.C

#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>
#include <iostream>

typedef CGAL::Homogeneous<CGAL::Gmpz> Kernel;
typedef CGAL::Polyhedron_3<Kernel> Polyhedron;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron;
typedef Kernel::Vector_3 Vector_3;
typedef Kernel::Aff_transformation_3 Aff_transformation_3;

int main() {
    Polyhedron P;
    std::cin >> P;
    if(P.is_closed()) {
        Nef_polyhedron N1(P);
        Nef_polyhedron N2(N1);
    }
}

```

```

Aff_transformation_3 aff(CGAL::TRANSLATION, Vector_3(2,2,0,1));
N2.transform(aff);
N1 += N2;

if(N1.is_simple()) {
    N1.convert_to_Polyhedron(P);
    std::cout << P;
}
else
    std::cerr << "N1 is not a 2-manifold." << std::endl;
}
}

```

15.5.5 Using an Extended Kernel

The provided extended kernels are used the same way as any other CGAL kernel. The essential difference is, that coordinates are not represented by the number type that was used to parameterize the kernel type, but by a *Nef_polynomial* parametrized by that number type.

The example iterates all vertices of a given Nef polyhedron and decides whether it is an standard vertex or a vertex on the infimal box. Furthermore, it tests whether any of the vertices is at (R, R, R) . Recall that R was the symbolical value, large but finite, for the size of the infimal box.

```

// file: examples/Nef_3/extended_kernel.C

#include <CGAL/Gmpz.h>
#include <CGAL/Extended_homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>

typedef CGAL::Gmpz NT;
typedef CGAL::Extended_homogeneous<NT> Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron;
typedef Nef_polyhedron::RT RT;
typedef Nef_polyhedron::Point_3 Point_3;
typedef Nef_polyhedron::Plane_3 Plane_3;
typedef Nef_polyhedron::Vertex_const_iterator Vertex_const_iterator;

int main() {

    Nef_polyhedron N;
    std::cin >> N;

    Vertex_const_iterator v;
    for(v = N.vertices_begin(); v != N.vertices_end(); ++v) {
        Point_3 p(v->point());
        if(p.hx().degree() > 0 || p.hy().degree() > 0 || p.hz().degree() > 0)
            std::cout << "extended vertex at " << p << std::endl;
        else
            std::cout << "standard vertex at " << p << std::endl;

        if(p == Point_3(RT(0,1), RT(0,1), RT(0,1)))

```

```

        std::cout << "    found vertex (right,back,top) of the infimaximal box"
                    << std::endl;
    }

    return 0;
}

```

15.6 File I/O

Nef_polyhedron_3 provides an input and an output operator for a proprietary file format. It includes the complete incidence structure, the geometric data, and the marks of each item. The output depends on the output operators of the geometric primitives provided by the traits class, and on the output operators of the used number type. Therefore, it is necessary to use the same kernel and the same number type for input and output operations.

We recommend to use the CGAL kernels *Homogeneous*, *Simple_homogeneous*, or *Extended_homogeneous*. We provide compatibility between the input and output of these kernels. Especially, it is possible to write a bounded Nef polyhedron using the *Extended_homogeneous* kernel and to read it afterwards using one of the two others.

Using CGAL stream modifiers the following output formats can be chosen: *ASCII*(*set_ascii_mode*), *binary*(*set_binary_mode*) or *pretty*(*set_pretty_mode*). The mandatory format is the ASCII format.

```

// file: examples/Nef_3/nefIO.C

#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Extended_homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>
#include <fstream>

typedef CGAL::Gmpz NT;
typedef CGAL::Homogeneous<NT> SK;
typedef CGAL::Extended_homogeneous<NT> EK;
typedef CGAL::Nef_polyhedron_3<SK> Nef_polyhedron_S;
typedef CGAL::Nef_polyhedron_3<EK> Nef_polyhedron_E;

int main() {
    Nef_polyhedron_E E;
    Nef_polyhedron_S S;

    std::cin >> E;

    if(E.is_bounded()) {
        std::ofstream out("temp.nef3");
        out << E;
        std::ifstream in("temp.nef3");
        in >> S;
    }
}

```

15.7 Further Example Programs

15.7.1 Exploring a Sphere Map

A sphere map is explored by using the function *get_sphere_map*, which returns the sphere map of the specified vertex as a *Nef_polyhedron_S2*. *Nef_polyhedron_S2* provides the functionality necessary for the exploration. Note, that one has to use the type *Nef_polyhedron_S2* as specified in *Nef_polyhedron_3* as is shown in the following example.

```
// file: examples/Nef_3/exploration_SM.C

#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>

typedef CGAL::Homogeneous<CGAL::Gmpz> Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron_3;

int main() {

    // We've put the typedefs here as VC7 gives us an ICE if they are global typedefs
    typedef Nef_polyhedron_3::Vertex_const_iterator Vertex_const_iterator;
    typedef Nef_polyhedron_3::Nef_polyhedron_S2 Nef_polyhedron_S2;
    typedef Nef_polyhedron_S2::SVertex_const_handle SVertex_const_handle;
    typedef Nef_polyhedron_S2::SHalfedge_const_handle SHalfedge_const_handle;
    typedef Nef_polyhedron_S2::SHalfloop_const_handle SHalfloop_const_handle;
    typedef Nef_polyhedron_S2::SFace_const_iterator SFace_const_iterator;
    typedef Nef_polyhedron_S2::SFace_cycle_const_iterator
        SFace_cycle_const_iterator;

    Nef_polyhedron_3 N;
    std::cin >> N;

    Vertex_const_iterator v = N.vertices_begin();
    Nef_polyhedron_S2 S(N.get_sphere_map(v));

    int i=0;
    SFace_const_iterator sf;
    for(sf = S.sfaces_begin(); sf != S.sfaces_end(); sf++) {
        SFace_cycle_const_iterator it;
        std::cout << "the sface cycles of sface " << i++ << " start with an\n";
        for(it = sf->sface_cycles_begin(); it != sf->sface_cycles_end(); it++) {
            if (it.is_svertex())
                std::cout << " svertex at position "
                    << SVertex_const_handle(it)->point() << std::endl;
            else if (it.is_shalfedge())
                std::cout << " shalfedge from "
                    << SHalfedge_const_handle(it)->source()->point() << " to "
                    << SHalfedge_const_handle(it)->target()->point() << std::endl;
            else if (it.is_shalfloop())
                std::cout << " shalfloop lying in the plane "
```

```

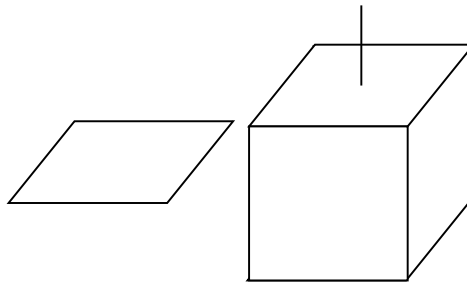
        << SHalfloop_const_handle(it)->circle() << std::endl;
    // other cases can not occur.
    }
}

return 0;
}

```

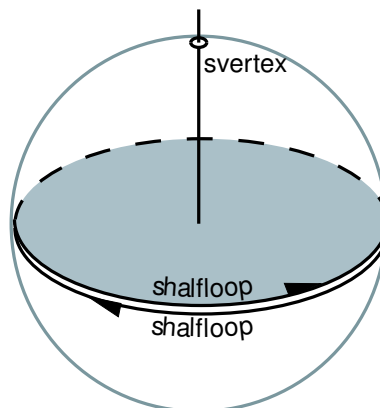
15.7.2 Exploring Shells

A *shell* of a Nef polyhedron is the connected part of the surface incident to a certain volume. Each halffacet, sface and shalfedge belongs to a single shell. The figure below illustrates the notion of a shell. It shows a Nef polyhedron with two volumes and three shells.



The first volume is the outer volume and the second volume is the interior of the cube. The first shell is the whole surface of the left object. The second shell is the outer surface of the right object, and the third shell is the inner surface of the right object.

In detail, the first shell consists of two halffacets, eight halfedges and four vertices. The second shell consists of the eight vertices of the cube plus the two endpoints of the antenna, all halffacets oriented outwards, and all halfedges. The third shell consists of the same eight vertices of the cube, plus the endpoint of the antenna that is in contact with the cube, all halffacets oriented inwards, and all halfedges (the same as for the second shell).



We discuss how sfaces, shalfedges, and sloops belong to the shells with a closeup view of the situation at the antenna foot. As you can see, there are three items on the sphere map - a shalfloop for each halffacet which intersects the sphere, and an svertex where the antenna intersects the sphere. The upper shalfloop lies on the halffacet which is oriented outwards and is therefore also oriented outwards. This shalfloop and the svertex

belong to the second shell. The other shalfloop lies on the inwards oriented halfacet and is oriented inwards, too. This shalfloop belongs to the third shell.

Nef_polyhedron_3 offers a visitor interface to explore a shell following the well-known visitor pattern [GHJV95]. The interface is illustrated by the following example.

```
// file: examples/Nef_3/shell_exploration.C

#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>

typedef CGAL::Homogeneous<CGAL::Gmpz> Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron;
typedef Nef_polyhedron::Vertex_const_handle Vertex_const_handle;
typedef Nef_polyhedron::Halfedge_const_handle Halfedge_const_handle;
typedef Nef_polyhedron::Halfacet_const_handle Halfacet_const_handle;
typedef Nef_polyhedron::SHalfedge_const_handle SHalfedge_const_handle;
typedef Nef_polyhedron::SHalfloop_const_handle SHalfloop_const_handle;
typedef Nef_polyhedron::SFace_const_handle SFace_const_handle;
typedef Nef_polyhedron::Volume_const_iterator Volume_const_iterator;
typedef Nef_polyhedron::Shell_entry_const_iterator Shell_entry_const_iterator;
typedef Kernel::Point_3 Point_3;

class Shell_explorer {
    bool first;
    const Nef_polyhedron& N;
    Vertex_const_handle v_min;

public:
    Shell_explorer(const Nef_polyhedron& N_)
        : first(true), N(N_) {}

    void visit(Vertex_const_handle v) {
        if(first || CGAL::lexicographically_xyz_smaller(v->point(), v_min->point())) {
            v_min = v;
            first=false;
        }
    }

    void visit(Halfedge_const_handle e) {}
    void visit(Halfacet_const_handle f) {}
    void visit(SHalfedge_const_handle se) {}
    void visit(SHalfloop_const_handle sl) {}
    void visit(SFace_const_handle sf) {}

    Vertex_const_handle& minimal_vertex() { return v_min; }
    void reset_minimal_vertex() { first = true; }
};

int main() {
    Nef_polyhedron N;
    std::cin >> N;
```



```

int ic = 0;
Volume_const_iterator c;
Shell_explorer SE(N);
CGAL_forall_volumes(c,N) {
    std::cout << "Volume " << ic++ << std::endl;
    int is = 0;
    Shell_entry_const_iterator it;
    CGAL_forall_shells_of(it,c) {
        SE.reset_minimal_vertex();
        N.visit_shell_objects(SFace_const_handle(it),SE);
        Point_3 p(SE.minimal_vertex()->point());
        std::cout << "  minimal vertex of shell " << is++
                    << " is at " << p << std::endl;
    }
}
}

```

The function *visit_shell_objects*(*SFace_const_handle sf*, *Visitor& V*) explores a shell starting at the *sf*. The second argument expects any class providing the (possibly empty) functions *visit(Vertex_const_handle)*, *visit(Halfedge_const_handle)* (remember that *Halfedge* is the same type as *SVertex*), *visit(Halfacet_const_handle)*, *visit(SHalfedge_const_handle)*, *visit(SHalfloop_const_handle)* and *visit(SFace_const_handle)*. The *visit_shell_objects* function will call *visit* for each item belonging to the shell once. There are no further requirements on that class.

In the example, the class *Shell_explorer* is passed as second argument to *visit_shell_objects*. Its task is to find the lexicographically smallest vertex of a shell. Its internal state consists of three variables. The first one is a reference to the explored Nef polyhedron. This reference is often necessary to retrieve information from the Nef polyhedron. The second variable *v_min* stores the smallest vertex found so far, and the third variable *first* is initialized to *false* to signal that no vertex has been visited so far. After the first vertex has been visited *first* is changed to *true*.

Shell_explorer provides further member functions. After the exploration of a shell the *minimal_vertex* function retrieves the smallest vertex. The *reset_minimal_vertex* function allows one to use the same instance of *Shell_explorer* on multiple shells. In this case, the *reset_minimal_vertex* function has to be called between the exploration of two shells.

The example program uses the *Shell_explorer* for each shell of the given Nef polyhedron once and reports the smallest vertex of each shell to the standard output.

15.7.3 Point Location

The *locate(Point_3 p)* function locates the point *p* in the Nef polyhedron and returns the item the point belongs to. The *locate* function returns an instance of *Object_handle*, which is a polymorphic handle type representing any handle type, no matter if it is mutable or const. For further usage of the result, the *Object_handle* has to be casted to the concrete handle type. The *CGAL::assign* function performs such a cast. It returns a boolean that reports the success or the failure of the cast. Looking at the possible return values of the *locate* function, the *Object_handle* can represent a *Vertex_const_handle*, a *Halfedge_const_handle*, a *Halfacet_handle*, or a *Volume_const_handle*. One of the four casts will succeed.

```
// file: examples/Nef_3/point_location.C
```

```

#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>

typedef CGAL::Homogeneous<CGAL::Gmpz> Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron_3;
    typedef Kernel::Point_3 Point_3;

int main() {
    //We've put the typedefs here as VC7 gives us an ICE if they are global typedefs
    typedef Nef_polyhedron_3::Vertex_const_handle Vertex_const_handle;
    typedef Nef_polyhedron_3::Halfedge_const_handle Halfedge_const_handle;
    typedef Nef_polyhedron_3::Halfacet_const_handle Halfacet_const_handle;
    typedef Nef_polyhedron_3::Volume_const_handle Volume_const_handle;
    typedef Nef_polyhedron_3::Object_handle Object_handle;

    Nef_polyhedron_3 N;
    std::cin >> N;

    Vertex_const_handle v;
    Halfedge_const_handle e;
    Halfacet_const_handle f;
    Volume_const_handle c;
    Object_handle o = N.locate(Point_3(0,0,0));
    if(CGAL::assign(v,o))
        std::cout << "Locating vertex" << std::endl;
    else if(CGAL::assign(e,o))
        std::cout << "Locating edge" << std::endl;
    else if(CGAL::assign(f,o))
        std::cout << "Locating facet" << std::endl;
    else if(CGAL::assign(c,o))
        std::cout << "Locating volume" << std::endl;
    //other cases can not occur

    return 0;
}

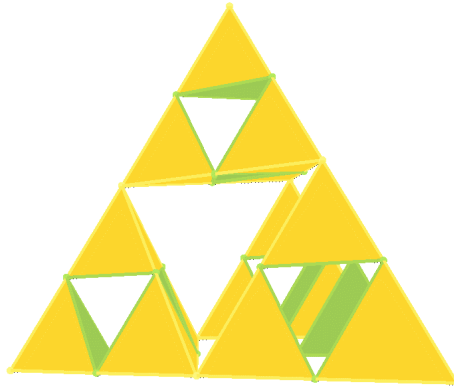
```

15.8 Visualization

With the *Qt_widget_OpenGL* class an interface to OpenGL visualization via Qt is offered. The class knows how to handle mouse movements and clicks and how to move and scale the 3D object displayed in the widget. *Qt_widget_OpenGL* is a basis for writing Qt widgets displaying 3D objects. A user can derive a new class from *Qt_widget_OpenGL* which implements the drawing method and configures the context menus.

15.8.1 Visualizing a 3D Nef polyhedron

Qt_widget_Nef_3 implements the drawing methods for displaying instances of *Nef_polyhedron_3*. The following example shows how to set up a QApplication with a main widget of type *Qt_widget_Nef_3* and how to start the viewer.



```
// Copyright (c) 2002 Max-Planck-Institute Saarbruecken (Germany)
// All rights reserved.
//
// This file is part of CGAL (www.cgal.org); you may redistribute it under
// the terms of the Q Public License version 1.0.
// See the file LICENSE.QPL distributed with CGAL.
//
// Licensees holding a valid commercial license may use this file in
// accordance with the commercial license agreement provided with the software.
//
// This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
// WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
//
// $URL: svn+ssh://scm.gforge.inria.fr/svn/cgal/branches/CGAL-3.2-branch/Nef_3/demo/Nef_3/visualization_3
// $Id: visualization_SNC.C 29577 2006-03-17 12:11:37Z hachenb $
//
//
// Author(s)      : Peter Hachenberger

#include <CGAL/basic.h>
#ifdef CGAL_USE_QT
#include <iostream>
int main(int, char*){
    std::cout << "Sorry, this demo needs QT..." << std::endl; return 0;}
#else
#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>
#include <CGAL/IO/Qt_widget_Nef_3.h>
#include <qapplication.h>

typedef CGAL::Homogeneous<CGAL::Gmpz> Kernel;
```

```

typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron_3;

int main(int argc, char* argv[]) {
    Nef_polyhedron_3 N;
    std::cin >> N;

    QApplication a(argc, argv);
    CGAL::Qt_widget_Nef_3<Nef_polyhedron_3>* w =
        new CGAL::Qt_widget_Nef_3<Nef_polyhedron_3>(N);
    a.setMainWidget(w);
    w->show();
    return a.exec();
}
#endif

```

15.8.2 Visualizing a Sphere Map

Qt_widget_Nef_S2 is a widget implemented on the basis of *Qt_widget_OpenGL*. It can be used to visualize the sphere map of a vertex in a *Nef_polyhedron_3* using the interface between *Nef_polyhedron_S2* and *Nef_polyhedron_3*.

```

// Copyright (c) 2002 Max-Planck-Institute Saarbruecken (Germany)
// All rights reserved.
//
// This file is part of CGAL (www.cgal.org); you may redistribute it under
// the terms of the Q Public License version 1.0.
// See the file LICENSE.QPL distributed with CGAL.
//
// Licensees holding a valid commercial license may use this file in
// accordance with the commercial license agreement provided with the software.
//
// This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
// WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
//
// $URL: svn+ssh://scm.gforge.inria.fr/svn/cgal/branches/CGAL-3.2-branch/Nef_3/demo/Nef_3/visualization_S
// $Id: visualization_SM.C 29697 2006-03-22 17:09:45Z afabri $
//
//
// Author(s) : Peter Hachenberger

#include <CGAL/basic.h>
#ifdef CGAL_USE_QT
#include <iostream>
int main(int, char*){
    std::cout << "Sorry, this demo needs QT..." << std::endl; return 0;}
#else
#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>
#include <CGAL/IO/Qt_widget_Nef_S2.h>
#include <qapplication.h>

```

```

typedef CGAL::Homogeneous<CGAL::Gmpz> Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron_3;

int main(int argc, char* argv[]) {

    // We've put the typedefs here as VC7 gives us an ICE if they are global typedefs
    typedef Nef_polyhedron_3::Vertex_const_iterator Vertex_const_iterator;
    typedef Nef_polyhedron_3::Nef_polyhedron_S2 Nef_polyhedron_S2;

    Nef_polyhedron_3 N;
    std::cin >> N;
    Vertex_const_iterator v = N.vertices_begin();
    Nef_polyhedron_S2 S(N.get_sphere_map(v));

    QApplication a(argc, argv);
    CGAL::Qt_widget_Nef_S2<Nef_polyhedron_S2>* w =
        new CGAL::Qt_widget_Nef_S2<Nef_polyhedron_S2>(S);
    a.setMainWidget(w);
    w->show();
    return a.exec();
}
#endif

```


3D Boolean Operations on Nef Polyhedra

Reference Manual

Peter Hachenberger, Lutz Kettner, and Michael Seel

A Nef polyhedron is any point set generated from a finite number of open halfspaces by set complement and set intersection operations. In our implementation of Nef polyhedra in 3-dimensional space, we offer a B-rep data structures that is closed under boolean operations and with all their generality. Starting from halfspaces (and also directly from oriented 2-manifolds), we can work with set union, set intersection, set difference, set complement, interior, exterior, boundary, closure, and regularization operations. In essence, we can evaluate a CSG-tree with halfspaces as primitives and convert it into a B-rep representation.

In fact, we work with two data structures; one that represents the local neighborhoods of vertices, which is in itself already a complete description, and a data structure that connects these neighborhoods up to a global data structure with edges, facets, and volumes. We offer a rich interface to investigate these data structures, their different elements and their connectivity. We provide affine (rigid) transformations and a point location query operation. We have a custom file format for storing and reading Nef polyhedra from files. We offer a simple OpenGL visualization for debugging and illustrations.

15.9 Classified Reference Pages

Classes

<code>CGAL::Nef_polyhedron_3<Traits></code>	page 1033
<code>CGAL::Nef_polyhedron_3<Traits>::Vertex</code>	page 1039
<code>CGAL::Nef_polyhedron_3<Traits>::Halfedge</code>	page 1040
<code>CGAL::Nef_polyhedron_3<Traits>::Halfacet</code>	page 1042
<code>CGAL::Nef_polyhedron_3<Traits>::Volume</code>	page 1044
<code>CGAL::Nef_polyhedron_3<Traits>::SHalfedge</code>	page 1045
<code>CGAL::Nef_polyhedron_3<Traits>::SHalfloop</code>	page 1047
<code>CGAL::Nef_polyhedron_3<Traits>::SFace</code>	page 1049
<code>CGAL::Nef_polyhedron_3<Traits>::SFace_cycle_iterator</code>	page 1051

Functions

`template<class Nef_polyhedron_3>`

std::size_t

OFF_to_nef_3(std::istream& in, Nef_polyhedron_3& N)

page [1053](#)

template <class Traits>

ostream& ostream& out << CGAL::Nef_polyhedron_3<Traits> N

page [1053](#)

template <class Traits>

istream& istream& in >> CGAL::Nef_polyhedron_3<Traits>& N

page [1054](#)

15.10 Alphabetical List of Reference Pages

<i>Halfedge</i>	page 1040
<i>Halffacet_cycle_iterator</i>	page 1050
<i>Halffacet</i>	page 1042
<i>Nef_polyhedron_3<Traits></i>	page 1033
<i>OFF_to_nef_3</i>	page 1052
<i>operator<<</i>	page 2800
<i>operator>></i>	page 2792
<i>Qt_widget_Nef_3<Nef_polyhedron_3</i>	page 1055
<i>SFace_cycle_iterator</i>	page 1051
<i>SFace</i>	page 1049
<i>SHalfedge</i>	page 1045
<i>SHalfloop</i>	page 1047
<i>Vertex</i>	page 1039
<i>Volume</i>	page 1044

CGAL::Nef_polyhedron_3<Traits>

Definition

A 3D Nef polyhedron is a subset of the 3-dimensional space that is the result of forming complements and intersections starting from a finite set H of 3-dimensional halfspaces. Nef polyhedra are closed under all binary set operations, i.e., intersection, union, difference, complement, and under the topological operations boundary, closure, and interior.

A 3D Nef polyhedron can be represented by the local pyramids of the minimal elements of its incidence structure. Without going into too much detail, a local pyramid essentially reflects the topologic and geometric situation at a certain location in a point set. For finite polyhedra the minimal elements of the incidence structure are vertices only. This means, that it suffices to model the topological and geometric situation of the vertices. For 3D Nef polyhedra, the local pyramid of a vertex is represented by a planar Nef polyhedra embedded on a sphere.

A *Nef_polyhedron_3* consists of vertices V , a sphere map for each vertex in V , edges E , facets F , volumes C , a mark for every item, and an incidence relation on them. Each edge and each facet is represented by two halfedges or two halffacets, respectively.

```
#include <CGAL/Nef_polyhedron_3.h>
```

Parameters

```
template < class Nef_polyhedronTraits_3,
            class Nef_polyhedronItems_3 = CGAL::SNC_items,
            class Nef_polyhedronMarks = bool >
class Nef_polyhedron_3;
```

The first parameter requires one of the following exact kernels: *Homogeneous*, *Simple_homogeneous*, *Extended_homogeneous_3* parametrized with *Gmpz*, *leda_integer* or any other number type modeling \mathbb{Z} , or *Cartesian*, *Simple_cartesian*, *Extended_cartesian_3* parametrized with *Gmpq*, *leda_rational*, *Quotient<Gmpz>* or any other number type modeling \mathbb{Q} .

The second parameter and the third parameter are for future considerations. Neither *Nef_polyhedronItems_3* nor *Nef_polyhedronMarks* is specified, yet. Do not use any other than the default types for these two template parameters.

Types

<i>Nef_polyhedron_3<Traits>:: Traits</i>	traits class selected for <i>Nef_polyhedronTraits_3</i> .
<i>Nef_polyhedron_3<Traits>:: Mark</i>	All object (vertices, edges, etc.) are attributed by a Mark. Mark equals bool.
<i>Nef_polyhedron_3<Traits>:: size_type</i>	size type of <i>Nef_polyhedron_3</i> .
<i>Nef_polyhedron_3<Traits>:: Vertex_const_handle</i>	non-mutable handle to a vertex.
<i>Nef_polyhedron_3<Traits>:: Halfedge_const_handle</i>	non-mutable handle to a halfedge.
<i>Nef_polyhedron_3<Traits>:: Halffacet_const_handle</i>	non-mutable handle to a halffacet.

<i>Nef_polyhedron_3<Traits>:: Volume_const_handle</i>	non-mutable handle to a volume.
<i>Nef_polyhedron_3<Traits>:: SVertex_const_handle</i>	non-mutable handle to a svertex.
<i>Nef_polyhedron_3<Traits>:: SHalfedge_const_handle</i>	non-mutable handle to a shalfedge.
<i>Nef_polyhedron_3<Traits>:: SHalfloop_const_handle</i>	non-mutable handle to a shalfloop.
<i>Nef_polyhedron_3<Traits>:: SFace_const_handle</i>	non-mutable handle to a sface.
<i>Nef_polyhedron_3<Traits>:: Vertex_const_iterator</i>	non-mutable iterator over all vertices.
<i>Nef_polyhedron_3<Traits>:: Halfedge_const_iterator</i>	non-mutable iterator over all halfedges.
<i>Nef_polyhedron_3<Traits>:: Halffacet_const_iterator</i>	non-mutable iterator over all halffacets.
<i>Nef_polyhedron_3<Traits>:: Volume_const_iterator</i>	non-mutable iterator over all volumes.
<i>Nef_polyhedron_3<Traits>:: SVertex_const_iterator</i>	non-mutable iterator over all svertices.
<i>Nef_polyhedron_3<Traits>:: SHalfedge_const_iterator</i>	non-mutable iterator over all shalfedges.
<i>Nef_polyhedron_3<Traits>:: SHalfloop_const_iterator</i>	non-mutable iterator over all shalfloops.
<i>Nef_polyhedron_3<Traits>:: SFace_const_iterator</i>	non-mutable iterator over all sfaces.
<i>Nef_polyhedron_3<Traits>:: SHalfedge_around_svertex_const_circulator</i>	
	non-mutable circulator of shalfedges around a svertex (cw).
<i>Nef_polyhedron_3<Traits>:: SHalfedge_around_sface_const_circulator</i>	
	non-mutable circulator of shalfedges around a sface (ccw).
<i>Nef_polyhedron_3<Traits>:: SHalfedge_around_facet_const_circulator</i>	
	non-mutable circulator of shalfedges around a halffacet (ccw).
<i>Nef_polyhedron_3<Traits>:: SFace_cycle_const_iterator</i>	
	non-mutable iterator over the cycles of a sface.
<i>Nef_polyhedron_3<Traits>:: Halffacet_cycle_const_iterator</i>	
	non-mutable iterator over the cycles of a halffacet.
<i>Nef_polyhedron_3<Traits>:: Shell_entry_const_iterator</i>	
	non-mutable iterator providing an entry to each shell.
<i>Nef_polyhedron_3<Traits>:: Object_handle</i>	a generic handle to an object. The kind of object (<i>vertex</i> , <i>halfedge</i> , <i>halffacet</i> , <i>volume</i> , <i>svertex</i> , <i>shalfedge</i> , <i>shalfloop</i> , <i>sface</i>) can be determined and the object can be assigned to a corresponding constant handle by one of the following functions: <i>bool assign(Vertex_const_handle& h, Object_handle)</i> <i>bool assign(Halfedge_const_handle& h, Object_handle)</i> <i>bool assign(Halffacet_const_handle& h, Object_handle)</i> <i>bool assign(Volume_const_handle& h, Object_handle)</i> <i>bool assign(SVertex_const_handle& h, Object_handle)</i> <i>bool assign(SHalfedge_const_handle& h, Object_handle)</i> <i>bool assign(SHalfloop_const_handle& h, Object_handle)</i> <i>bool assign(SFace_const_handle& h, Object_handle)</i> where each function returns <i>true</i> iff the assignment to <i>h</i> could be accomplished.
<i>Nef_polyhedron_3<Traits>:: Point_3</i>	location of vertices.

<i>Nef_polyhedron_3<Traits>:: Segment_3</i>	segment represented by a halfedge.
<i>Nef_polyhedron_3<Traits>:: Vector_3</i>	direction of a halfedge.
<i>Nef_polyhedron_3<Traits>:: Plane_3</i>	plane of a halffacet lies in.
<i>Nef_polyhedron_3<Traits>:: Aff_transformation_3</i>	affine transformation.

enum Boundary { EXCLUDED, INCLUDED};

construction selection.

enum Content { EMPTY, COMPLETE};

construction selection.

Nef_polyhedron_3<Traits>:: Nef_polyhedron_S2

a sphere map.

Nef_polyhedron_3<Traits>:: Polyhedron

a polyhedral surface.

Creation

Nef_polyhedron_3<Traits> N(Content space = EMPTY);

creates a Nef polyhedron and initializes it to the empty set if *plane == EMPTY* and to the whole space if *space == COMPLETE*.

Nef_polyhedron_3<Traits> N(Plane_3 p, Boundary b = INCLUDED);

creates a Nef polyhedron containing the halfspace left of *p* including *p* if *b==INCLUDED*, excluding *p* if *b==EXCLUDED*.

Nef_polyhedron_3<Traits> N(Polyhedron& P);

creates a Nef polyhedron, which represents the same point set as the polyhedral surface *P* does.

Nef_polyhedron_3<Traits> N(Input_iterator begin, Input_iterator end);

creates a Nef polyhedron consisting of a single polygon spanned by the list of points in the iterator range *[begin,end)*. If the points do not on a common supporting plane, the constructor tries to triangulate the polygon into multiple facets.If the construction does not succeed, the empty set is created.

Access Member Functions

<i>bool</i>	<i>N.is_simple()</i>	returns true, if <i>N</i> is a 2-manifold.
<i>bool</i>	<i>N.is_valid()</i>	checks the integrity of <i>N</i> .
<i>Size_type</i>	<i>N.number_of_vertices()</i>	returns the number of vertices.

<i>Size_type</i>	<i>N.number_of_halfedges()</i>	return the number of halfedges.
<i>Size_type</i>	<i>N.number_of_edges()</i>	returns the number of halfedge pairs.
<i>Size_type</i>	<i>N.number_of_halffacets()</i>	returns the number of halffacets.
<i>Size_type</i>	<i>N.number_of_facets()</i>	returns the number of halffacet pairs.
<i>Size_type</i>	<i>N.number_of_volumes()</i>	returns the number of volumes.

<i>Vertex_const_iterator</i>	<i>N.vertices_begin()</i>	iterator over all vertices.
<i>Vertex_const_iterator</i>	<i>N.vertices_end()</i>	past-the-end iterator.

<i>Halfedge_const_iterator</i>	<i>N.halfedges_begin()</i>	iterator over all halfedges.
<i>Halfedge_const_iterator</i>	<i>N.halfedges_end()</i>	past-the-end iterator.

<i>Halffacet_const_iterator</i>	<i>N.halffacets_begin()</i>	iterator over all halffacets.
<i>Halffacet_const_iterator</i>	<i>N.halffacets_end()</i>	past-the-end iterator.

<i>Volume_const_iterator</i>	<i>N.volumes_begin()</i>	iterator over all volumes.
<i>Volume_const_iterator</i>	<i>N.volumes_end()</i>	past-the-end iterator.

The following macros are provided: *CGAL_forall_vertices(v,N)*, *CGAL_forall_halfedges(e,N)*, *CGAL_forall_edges(e,N)*, *CGAL_forall_halffacets(f,N)*, *CGAL_forall_facets(f,N)*, *CGAL_forall_volumes(c,N)* where *N* is a *Nef_polyhedron_3*.

<i>Object_handle</i>	<i>N.locate(Point_3 p)</i>	returns a generic handle to the object (vertex, edge, facet or volume) which contains the point p in its relative interior.
----------------------	-----------------------------	-----------------------------------------------------------------------------------------------------------------------------

<i>Nef_polyhedron_S2</i>	<i>N.get_sphere_map(Vertex_const_iterator v)</i>	returns the neighborhood of a vertex modeled by a <i>Nef_polyhedron_S2</i> .
--------------------------	---------------------------------------------------	------------------------------------------------------------------------------

Point Set Predicates

<i>bool</i>	<i>N.is_empty()</i>	returns true, if <i>N</i> is the empty point set.
<i>bool</i>	<i>N.is_space()</i>	returns true, if <i>N</i> is the complete 3D space.
<i>bool</i>	<i>N == N1</i>	returns true, if <i>N</i> and <i>N1</i> comprise the same point sets.
<i>bool</i>	<i>N != N1</i>	returns true, if <i>N</i> and <i>N1</i> comprise different point sets.
<i>bool</i>	<i>N < N1</i>	returns true, if <i>N</i> is a proper subset of <i>N1</i> .
<i>bool</i>	<i>N > N1</i>	returns true, if <i>N</i> is a proper superset of <i>N1</i> .
<i>bool</i>	<i>N <= N1</i>	returns true, if <i>N</i> is a subset of <i>N1</i> .
<i>bool</i>	<i>N >= N1</i>	returns true, if <i>N</i> is a superset of <i>N1</i> .

Unary Set Operations

<i>Nef_polyhedron_3<Traits></i>	<i>N.complement()</i>	returns the complement of <i>N</i> .
<i>Nef_polyhedron_3<Traits></i>	<i>N.interior()</i>	returns the interior of <i>N</i> .
<i>Nef_polyhedron_3<Traits></i>	<i>N.boundary()</i>	returns the boundary of <i>N</i> .
<i>Nef_polyhedron_3<Traits></i>	<i>N.closure()</i>	returns the closure of <i>N</i> .

<i>Nef_polyhedron_3<Traits></i>	<i>N.regularization()</i>	returns the regularization, i.e. the closure of the interior, of <i>N</i> .
<i>Nef_polyhedron_3<Traits></i>	<i>!N</i>	returns the complement of <i>N</i> .

Binary Set Operations

<i>Nef_polyhedron_3<Traits></i>	<i>N.intersection(N1)</i>	return the intersection of <i>N</i> and <i>N1</i> .
<i>Nef_polyhedron_3<Traits></i>	<i>N.join(N1)</i>	return the union of <i>N</i> and <i>N1</i> . (Note that "union" is a C++ keyword and cannot be used for this operation.)
<i>Nef_polyhedron_3<Traits></i>	<i>N.difference(N1)</i>	return the difference between <i>N</i> and <i>N1</i> .
<i>Nef_polyhedron_3<Traits></i>	<i>N.symmetric_difference(N1)</i>	return the symmetric difference of <i>N</i> and <i>N1</i> .
<i>Nef_polyhedron_3<Traits></i>	<i>N * N1</i>	return the intersection of <i>N</i> and <i>N1</i> .
<i>Nef_polyhedron_3<Traits></i>	<i>N + N1</i>	return the union of <i>N</i> and <i>N1</i> .
<i>Nef_polyhedron_3<Traits></i>	<i>N - N1</i>	return the difference between <i>N</i> and <i>N1</i> .
<i>Nef_polyhedron_3<Traits></i>	<i>N ^ N1</i>	return the symmetric difference of <i>N</i> and <i>N1</i> .
<i>void</i>	<i>N *= N1</i>	intersects <i>N</i> and <i>N1</i> .
<i>void</i>	<i>N += N1</i>	unites <i>N</i> with <i>N1</i> .
<i>void</i>	<i>N -= N1</i>	subtracts <i>N1</i> from <i>N</i> .
<i>void</i>	<i>N ^= N1</i>	performs a symmetric intersection of <i>N</i> and <i>N1</i> .

Operations

<i>void</i>	<i>N.clear(Content space = EMPTY)</i>	make <i>N</i> the empty set if <i>space == EMPTY</i> and the complete 3D space if <i>space == COMPLETE</i> .
<i>void</i>	<i>N.transform(Aff_transformation_3 aff)</i>	applies an affine transformation to <i>N</i> .
<i>void</i>	<i>N.convert_to_Polyhedron(Polyhedron& P)</i>	converts <i>N</i> into a Polyhedron. Precondition: <i>N</i> is simple.
<i>void</i>	<i>N.visit_shell_objects(SFace_const_handle f, Visitor& V)</i>	calls the visit function of <i>V</i> for every item which belongs to the same shell as <i>sf</i> .

See Also

<i>CGAL::Nef_polyhedron_3<Traits>::Vertex</i>	page 1039
<i>CGAL::Nef_polyhedron_3<Traits>::Halfedge</i>	page 1040
<i>CGAL::Nef_polyhedron_3<Traits>::Halfacet</i>	page 1042
<i>CGAL::Nef_polyhedron_3<Traits>::Volume</i>	page 1044
<i>CGAL::Nef_polyhedron_3<Traits>::SHalfedge</i>	page 1045
<i>CGAL::Nef_polyhedron_3<Traits>::SHalfloop</i>	page 1047
<i>CGAL::Nef_polyhedron_3<Traits>::SFace</i>	page 1049
<i>CGAL::Nef_polyhedron_S2<Traits></i>	page 991
<i>CGAL::Polyhedron<Traits></i>	page ??

Example

This example program creates two Nef polyhedra - one representing the empty point set, one representing the whole 3D space. The complement of the latter one is computed and compared with the first one.

```
// file: examples/Nef_3/simple.C

#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>

typedef CGAL::Homogeneous<CGAL::Gmpz> Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron;

int main() {
    Nef_polyhedron N0(Nef_polyhedron::EMPTY);
    Nef_polyhedron N1(Nef_polyhedron::COMPLETE);

    CGAL_assertion (N0 == N1.complement());
    return 0;
}
```

CGAL::Nef_polyhedron_3<Traits>::Vertex

Definition

A vertex is a point in the 3-dimensional space. Its incidence structure can be accessed creating a sphere map of the vertex. This is done by the member function *get_sphere_map* of the class *Nef_polyhedron_3*.

```
#include <CGAL/Nef_polyhedron_3.h>
```

Types

The following types are the same as in *Nef_polyhedron_3<Traits>*.

Vertex::Mark type of mark.

Vertex::Point_3 point type stored in Vertex.

Creation

There is no need for a user to create a *Vertex* explicitly. The class *Nef_polyhedron_3<Traits>* manages the needed vertices internally.

Operations

Mark *v.mark()* the mark of *v* .

Point_3 *v.point()* the point of *v* .

CGAL::Nef_polyhedron_3<Traits> page [1033](#)

CGAL::Nef_polyhedron_S2<Traits> page [991](#)

CGAL::Nef_polyhedron_3<Traits>::Halfedge

Definition

A Halfedge has a double meaning. In the global incidence structure of a *Nef_polyhedron_3* it is an oriented edge going from one vertex to another. A halfedge also coincides with an svertex of the sphere map of its source vertex. Because of this, we offer the types *Halfedge* and *SVertex* which are the same. Furthermore, the redundant functions *center_vertex()* and *source()* are provided. The reason is, that we get the same vertex either if we want to have the source vertex of a halfedge, or if we want to have the vertex in the center of the sphere map a svertex lies on. Figure 15.9 on page 1045 and figure 15.9 on page 1042 illustrate the incidence of a svertex on a sphere map and of a halfedge in the global structure.

As part of the global incidence structure, the member functions *source* and *target* return the source and target vertex of an edge. The member function *twin()* returns the opposite halfedge.

Looking at the incidence structure on a sphere map, the member function *out_sedge* returns the first outgoing halfedge, and *incident_sface* returns the incident sface.

```
#include <CGAL/Nef_polyhedron_3.h>
```

Types

The following types are the same as in *Nef_polyhedron_3<Traits>*.

<i>Halfedge::Mark</i>	type of mark.
<i>Halfedge::Sphere_point</i>	sphere point type stored in Halfedge.
<i>Halfedge::Vertex_const_handle</i>	const handle to vertex.
<i>Halfedge::Halfedge_const_handle</i>	const handle to halfedge.
<i>Halfedge::SHalfedge_const_handle</i>	const handle to SHalfedge.
<i>Halfedge::SFace_const_handle</i>	const handle to SFace.

Creation

There is no need for a user to create a *Halfedge* explicitly. The class *Nef_polyhedron_3<Traits>* manages the needed halfedges internally.

Operations

<i>Mark</i>	<i>e.mark()</i>	the mark of <i>e</i> .
<i>Sphere_point</i>	<i>e.point()</i>	the sphere point of <i>e</i> .
<i>bool</i>	<i>e.is_isolated()</i>	returns —true— if <i>e</i> has no adjacent sedges.
<i>Vertex_const_handle</i>	<i>e.center_vertex()</i>	the center vertex of the sphere map <i>e</i> belongs to.

<i>Vertex_const_handle</i>	<i>e.source()</i>	the source vertex of <i>e</i> .
<i>Vertex_const_handle</i>	<i>e.target()</i>	the target vertex <i>e</i> .
<i>Halfedge_const_handle</i>	<i>e.twin()</i>	the twin of <i>e</i> .
<i>SHalfedge_const_handle</i>		
	<i>e.out_sedge()</i>	the first out sedege of <i>e</i> .
<i>SFace_const_handle</i>	<i>e.incident_sface()</i>	the incident sface of <i>e</i> .

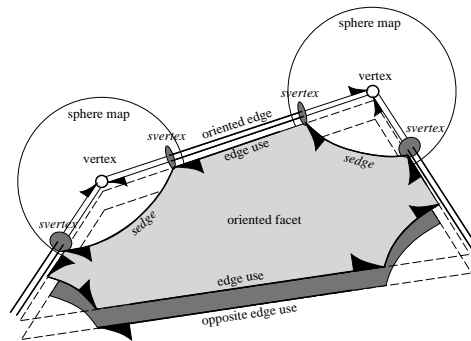
See Also

CGAL::Nef_polyhedron_3<Traits>::Vertex page [1039](#)
CGAL::Nef_polyhedron_3<Traits>::SHalfedge page [1045](#)
CGAL::Nef_polyhedron_3<Traits>::SFace page [1049](#)
CGAL::Nef_polyhedron_S2<Traits>::Sphere_point page [997](#)

CGAL::Nef_polyhedron_3<Traits>::Halffacet

Definition

A halffacet is an oriented, rectilinear bounded part of a plane. The following figure depicts the incidences to halfedges, vertices and the notion of facet cycles.



The member function *twin* returns the opposite halffacet, *incident_volume* returns the incident volume. A Halffacet cycle either consists of consecutive halfedges along the border (or a hole) of the halffacet, or of a single shalloop on the sphere map of a vertex isolated on the halffacet. The iterator range (*halffacet_cycles_begin()/halffacet_cycles_end()*) provides an entry element for each halffacet cycle of a halffacet.

```
#include <CGAL/Nef_polyhedron_3.h>
```

Types

The following types are the same as in *Nef_polyhedron_3<Traits>*.

<i>Halffacet:: Mark</i>	type of mark.
<i>Halffacet:: Plane_3</i>	plane type stored in Halffacet.
<i>Halffacet:: Object_list</i>	list of Object handles.
<i>Halffacet:: Halffacet_const_handle</i>	const handle to Halffacet.
<i>Halffacet:: Volume_const_handle</i>	const handle to volume.
<i>Halffacet:: Halffacet_cycle_const_iterator</i>	const iterator over the entries to all halffacet cycles of a halffacet.

Creation

There is no need for a user to create a *Halffacet* explicitly. The class *Nef_polyhedron_3<Traits>* manages the needed halffacets internally.

Operations

<i>Mark</i>	<i>f.mark()</i>	the mark of <i>f</i> .
<i>Plane_3</i>	<i>f.plane()</i>	the supporting plane of <i>f</i> .
<i>Halffacet_const_handle</i>	<i>f.twin()</i>	the twin of <i>f</i> .
<i>Volume_const_handle</i>	<i>f.incident_volume()</i>	the incident volume of <i>f</i> .
<i>Halffacet_cycle_const_iterator</i>	<i>f.facet_cycle_begin()</i>	iterator over the entries to all halffacet cycles of <i>f</i> .
<i>Halffacet_cycle_const_iterator</i>	<i>f.facet_cycle_end()</i>	past-the-end iterator.

See Also

CGAL::Nef_polyhedron_3<Traits>::Volume page [1044](#)
CGAL::Nef_polyhedron_3<Traits>::Halfedge page [1040](#)
CGAL::Nef_polyhedron_3<Traits>::SHalfedge page [1045](#)

CGAL::Nef_polyhedron_3<Traits>::Volume

Definition

A volume is a full-dimensional connected point set in \mathbb{R}^3 . It is bounded by several shells, i.e. a connected part of the boundary incident to a single volume. An entry element to each shell is provided by the iterator range (*shells_begin()/shells_end()*). A *Shell_entry_iterator* is assignable to *SFace_handle*.

```
#include <CGAL/Nef_polyhedron_3.h>
```

Types

The following types are the same as in *Nef_polyhedron_3<Traits>*.

<i>Volume:: Mark</i>	type of mark.
<i>Volume:: Object_list</i>	list of Object handles.
<i>Volume:: Volume_const_handle</i>	const handle to Volume.
<i>Volume:: Shell_entry_const_iterator</i>	const iterator over the entries to all shells adjacent to a volume.

Creation

There is no need for a user to create a *Volume* explicitly. The class *Nef_polyhedron_3<Traits>* manages the needed volumes internally.

Operations

<i>Mark</i>	<i>c.mark()</i>	the mark of <i>c</i> .
<i>Shell_entry_const_iterator</i>	<i>c.shells_begin()</i>	const iterator over the entries to all shells adjacent to <i>c</i> .
<i>Shell_entry_const_iterator</i>	<i>c.shells_end()</i>	past-the-end iterator.

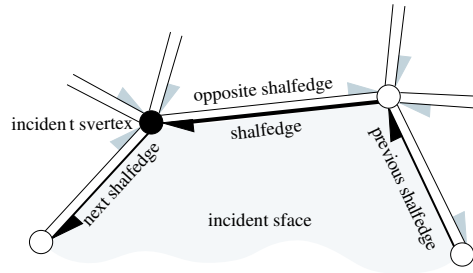
See Also

CGAL::Nef_polyhedron_3<Traits>::SFace page [1049](#)

CGAL::Nef_polyhedron_3<Traits>::SHalfedge

Definition

A shalfedge is a great arc on a sphere map. Figure 15.9 on page 1045 depicts the relationship between a shalfedge and its incident shalfedges, svertices, and sfaces on a sphere map. A shalfedge is an oriented sedge between two svertices. It is always paired with a shalfedge pointing in the opposite direction. The *twin()* member function returns this shalfedge of opposite orientation.



The *snext()* member function points to the successor shalfedge around this sface while the *sprev()* member function points to the preceding shalfedge. An successive assignments of the form *se = se->snext()* cycles counterclockwise around the sface (or hole).

Similarly, the successive assignments of the form *se = se->snext()->twin()* cycle clockwise around the svertex and traverse all halfedges incident to this svertex. The assignment *se = se->cyclic_adj_succ()* can be used as a shortcut.

The role of shalfedges in a facet is illustrated in Figure 15.9 on page 1042. The *facet()* member function returns the facet in which the shalfedge is part of one of the facet cycles. The successive assignment of the form *se = se->next()* cycles counterclockwise around the facet (or a hole of the facet).

A const circulators is provided for each of the three circular orders. The circulators are bidirectional and assignable to *SHalfedge_const_handle*.

```
#include <CGAL/Nef_polyhedron_3.h>
```

Types

The following types are the same as in *Nef_polyhedron_3<Traits>*.

SHalfedge::Mark type of mark.

SHalfedge::Sphere_circle sphere circle type stored in SHalfedge.

SHalfedge::Halfacet_const_handle const handle to Halfacet.

SHalfedge::SVertex_const_handle const handle to SVertex.

SHalfedge::SHalfedge_const_handle const handle to SHalfedge.

SHalfedge::SFace_const_handle const handle to SFace.

Creation

There is no need for a user to create a *SHalfedge* explicitly. The class *Nef_polyhedron_3<Traits>* manages the needed shalfedges internally.

Operations

<i>Mark</i>	<i>se.mark()</i>	the mark of <i>se</i> .
<i>Sphere_circle</i>	<i>se.circle()</i>	the sphere circle of <i>se</i> .
<i>SHalfedge_const_handle</i>	<i>se.twin()</i>	the twin of <i>se</i> .
<i>SVertex_const_handle</i>	<i>se.source()</i>	the source svertex of <i>se</i> .
<i>SVertex_const_handle</i>	<i>se.target()</i>	equals <i>twin()</i> -> <i>source()</i> .
<i>SHalfedge_const_handle</i>	<i>se.prev()</i>	the SHalfedge previous to <i>se</i> in a facet cycle.
<i>SHalfedge_const_handle</i>	<i>se.next()</i>	the next SHalfedge of <i>se</i> in a facet cycle.
<i>SHalfedge_const_handle</i>	<i>se.sprev()</i>	the SHalfedge previous to <i>se</i> in a sface cycle.
<i>SHalfedge_const_handle</i>	<i>se.snext()</i>	the next SHalfedge of <i>se</i> in a sface cycle.
<i>SHalfedge_const_handle</i>	<i>se.cyclic_adj_pred()</i>	the edge before <i>se</i> in the cyclic ordered adjacency list of <i>source()</i> .
<i>SHalfedge_const_handle</i>	<i>se.cyclic_adj_succ()</i>	the edge after <i>se</i> in the cyclic ordered adjacency list of <i>source()</i> .
<i>Halffacet_const_handle</i>	<i>se.facet()</i>	the facet that corresponds to <i>se</i> in the 3D incidence structure.
<i>SFace_const_handle</i>	<i>se.incident_sface()</i>	the incident sface of <i>se</i> .

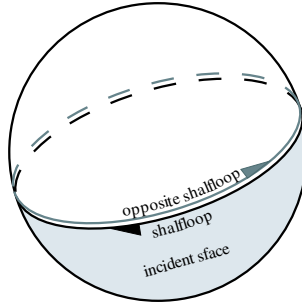
See Also

<i>CGAL::Nef_polyhedron_3<Traits>::Halfedge</i>	page 1040
<i>CGAL::Nef_polyhedron_3<Traits>::Halffacet</i>	page 1042
<i>CGAL::Nef_polyhedron_3<Traits>::SFace</i>	page 1049
<i>CGAL::Nef_polyhedron_S2<Traits>::Sphere_circle</i>	page 1000

CGAL::Nef_polyhedron_3<Traits>::SHalfloop

Definition

A shalfloop is a great circle on a sphere map. Figure 15.9 on page 1047 depicts the relationship between a shalfloop and its incident shalfloops, and sfaces on a sphere map. A shalfloop is an oriented sloop. It is always paired with a shalfloop whose supporting *Sphere_circle* is pointing in the opposite direction. The *twin()* member function returns this shalfloop of opposite orientation.



A sphere map having a shalfloop models the neighborhood of a vertex which is isolated on a facet. That facet is returned by the member function *facet*.

```
#include <CGAL/Nef_polyhedron_3.h>
```

Types

The following types are the same as in *Nef_polyhedron_3<Traits>*.

<i>SHalfloop:: Mark</i>	type of mark.
<i>SHalfloop:: Sphere_circle</i>	sphere circle type stored in SHalfloop.
<i>SHalfloop:: Halffacet_const_handle</i>	const handle to Halffacet.
<i>SHalfloop:: SHalfloop_const_handle</i>	const handle to SHalfloop.
<i>SHalfloop:: SFace_const_handle</i>	const handle to SFace.

Creation

There is no need for a user to create a *SHalfloop* explicitly. The class *Nef_polyhedron_3<Traits>* manages the needed shalfloops internally.

Operations

<i>Mark</i>	<i>se.mark()</i>	the mark of <i>se</i> .
<i>Sphere_circle</i>	<i>se.circle()</i>	the sphere circle of <i>se</i> .

<i>SHalfloop_const_handle</i>	<i>se.twin()</i>	the twin of <i>se</i> .
<i>Halffacet_const_handle</i>	<i>se.facet()</i>	the facet that corresponds to <i>se</i> in the 3D incidence structure.
<i>SFace_const_handle</i>	<i>se.incident_sface()</i>	the incident sface of <i>se</i> .

See Also

CGAL::Nef_polyhedron_3<Traits>::Halffacet page [1042](#)
CGAL::Nef_polyhedron_3<Traits>::SFace page [1049](#)
CGAL::Nef_polyhedron_S2<Traits>::Sphere_point page [997](#)

CGAL::Nef_polyhedron_3<Traits>::SFace

Definition

Figure 15.9 on page 1045 and figure 15.9 on page 1047 illustrate the incidences of an sface. An sface is described by its boundaries. An entry item to each boundary cycle can be accessed using the iterator range (*sface_cycles_begin()/sface_cycles_end()*). Additionally, *Nef_polyhedron_S2* provides the macro *CGAL_forall_sface_cycles_of*. The iterators are of type *SFace_cycle_const_iterator* and represent either a shalfedge, a shalfloop, or a svertex.

```
#include <CGAL/Nef_polyhedron_3.h>
```

Types

The following types are the same as in *Nef_polyhedron_3<Traits>*.

<i>SFace:: Mark</i>	type of mark.
<i>SFace:: Object_list</i>	list of Object handles.
<i>SFace:: Vertex_const_handle</i>	const handle to Vertex.
<i>SFace:: Volume_const_handle</i>	const handle to Volume.
<i>SFace:: SFace_const_handle</i>	const handle to SFace.
<i>SFace:: SFace_cycle_const_iterator</i>	const iterator over the entries to all sface cycles of a sface.

Creation

There is no need for a user to create a *SFace* explicitly. The class *Nef_polyhedron_3<Traits>* manages the needed sfaces internally.

Operations

<i>Mark</i>	<i>sf.mark()</i>	the mark of <i>sf</i> .
<i>Vertex_const_handle</i>	<i>sf.center_vertex()</i>	the center vertex of the sphere map <i>sf</i> belongs to.
<i>Volume_const_handle</i>	<i>sf.volume()</i>	the volume that corresponds to <i>sf</i> in the 3D incidence structure.
<i>SFace_cycle_const_iterator</i>	<i>sf.sface_cycle_begin()</i>	iterator over the entries to all sface cycles of <i>sf</i> .
<i>SFace_cycle_const_iterator</i>	<i>sf.sface_cycle_end()</i>	past-the-end iterator.

See Also

CGAL::Nef_polyhedron_3<Traits>::Vertex page 1039
CGAL::Nef_polyhedron_3<Traits>::Volume page 1044

CGAL::Nef_polyhedron_3<Traits>::Halffacet_cycle_iterator

Definition

The type *Halffacet_cycle_iterator* iterates over a list of *Object_handles*. Each item of that list can either be assigned to *SHalfedge_handle* or *SHalfloop_handle*. To find out which of these assignment works out, the member functions *is_shalfedge()* and *is_shalfloop()* are provided.

```
#include <CGAL/Nef_polyhedron_3.h>
```

Types

<i>Halffacet_cycle_iterator::SHalfedge_handle</i>	const handle to <i>SHalfedge</i> .
<i>Halffacet_cycle_iterator::SHalfloop_handle</i>	const handle to <i>SHalfloop</i> .

Creation

<i>Halffacet_cycle_iterator hfc;</i>	default constructor.
--------------------------------------	----------------------

Operations

<i>bool</i>	<i>hfc.is_shalfedge()</i>	returns true if <i>hfc</i> represents a <i>SHalfedge_handle</i> .
<i>bool</i>	<i>hfc.is_shalfloop()</i>	returns true if <i>hfc</i> represents a <i>SHalfloop_handle</i> .
<i>SHalfedge_handle</i>	<i>SHalfedge_handle(hfc)</i>	casts <i>hfc</i> to <i>SHalfedge_handle</i> .
<i>SHalfloop_handle</i>	<i>SHalfloop_handle(hfc)</i>	casts <i>hfc</i> to <i>SHalfloop_handle</i> .

See Also

<i>CGAL::Nef_polyhedron_3<Traits>::SHalfedge</i>	page 1045
<i>CGAL::Nef_polyhedron_3<Traits>::SHalfloop</i>	page 1047

CGAL::Nef_polyhedron_3<Traits>::SFace_cycle_iterator

Definition

The type *SFace_cycle_iterator* iterates over a list of *Object_handles*. Each item of that list can either be assigned to *SVertex_handle*, *SHalfedge_handle* or *SHalfloop_handle*. To find out which of these assignment works out, the member functions *is_svertex()*, *is_shalfedge()* and *is_shalfloop()* are provided.

```
#include <CGAL/Nef_polyhedron_3.h>
```

Types

<i>SFace_cycle_iterator::SVertex_handle</i>	const handle to <i>SVertex</i> .
<i>SFace_cycle_iterator::SHalfedge_handle</i>	const handle to <i>SHalfedge</i> .
<i>SFace_cycle_iterator::SHalfloop_handle</i>	const handle to <i>SHalfloop</i> .

Creation

<i>SFace_cycle_iterator sfc;</i>	default constructor.
----------------------------------	----------------------

Operations

<i>bool</i>	<i>sfc.is_svertex()</i>	returns true if <i>sfc</i> represents a <i>SVertex_handle</i> .
<i>bool</i>	<i>sfc.is_shalfedge()</i>	returns true if <i>sfc</i> represents a <i>SHalfedge_handle</i> .
<i>bool</i>	<i>sfc.is_shalfloop()</i>	returns true if <i>sfc</i> represents a <i>SHalfloop_handle</i> .
<i>SVertex_handle</i>	<i>SVertex_handle(sfc)</i>	casts <i>sfc</i> to <i>SVertex_handle</i> .
<i>SHalfedge_handle</i>	<i>SHalfedge_handle(sfc)</i>	casts <i>sfc</i> to <i>SHalfedge_handle</i> .
<i>SHalfloop_handle</i>	<i>SHalfloop_handle(sfc)</i>	casts <i>sfc</i> to <i>SHalfloop_handle</i> .

See Also

<i>CGAL::Nef_polyhedron_3<Traits>::Halfedge</i>	page 1040
<i>CGAL::Nef_polyhedron_3<Traits>::SHalfedge</i>	page 1045
<i>CGAL::Nef_polyhedron_3<Traits>::SHalfloop</i>	page 1047

CGAL::OFF_to_nef_3

Definition

This constructor creates a 3D Nef polyhedron from OFF file. OFF file is read from input stream *in*. The purpose of *OFF_to_nef_3* is to create a Nef polyhedron from an OFF file that cannot be handled by the *Nef_polyhedron_3* constructors. It handles double coordinates while using a homogenous kernel, non-coplanar facets, surfaces with boundaries, self-intersecting surfaces, and single facets. Every closed volume gets marked. The function returns the number of facets it could not handle.

```
template<class Nef_polyhedron_3>
std::size_t      OFF_to_nef_3( std::istream& in, Nef_polyhedron_3& N)
```

See Also

CGAL::Nef_polyhedron_3<Traits> page [1033](#)

CGAL::operator<<

Definition

This operator writes The Nef polyhedron N to the output stream *out* using a propriatary file format. It includes the complete incidence structure, the geometric data, and the marks of each item.

Using CGAL stream modifiers the following output formats can be chosen: ASCII(*set_ascii_mode*), binary(*set_binary_mode*) or pretty(*set_pretty_mode*). The mandatory format is the ASCII format. It is recommended to use this format for file input and output.

As the output depends on the output operators of the geometric primitives provided by the traits class, it might not be possible that the input operator and output operators of different traits classes are not compatible. We recommend to use the CGAL kernels *Homogeneous*, *Simple_homogeneous*, or *Extended_homogeneous* parametrized with any exact number type that models *mathbb{Z}* (e.g. *Gmpz* or *leda_integer*).

A bounded *Nef_polyhedron_3*<*Extended_homogeneous*> is automatically written as though *Nef_polyhedron_3*<*CGAL::Homogeneous*> or *Nef_polyhedron_3*<*CGAL::Simple_homogeneous*> is used. As a result, the input operator of each of these types can read the output.

```
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>
```

```
template <class Nef_polyhedronTraits_3>
ostream& ostream& out << CGAL::Nef_polyhedron_3<Nef_polyhedronTraits_3> N
```

See Also

CGAL::Nef_polyhedron_3<*Traits*> page [1033](#)
operator>> page [1054](#)

CGAL::operator>>

Definition

This operator reads a Nef polyhedron, which is given in the proprietary file format written by the input operator *in* and assigns it to *N*. It includes the complete incidence structure, the geometric data, and the marks of each item.

It is recommended to use the CGAL kernels *Homogeneous*, *Simple_homogeneous*, or *Extended_homogeneous* parametrized with any exact number type that models \mathbb{Z} (e.g. *Gmpz* or *leda_integer*). The input and output iterators of Nef polyhedra parametrized with either of these kernels are compatible as long as the Nef polyhedron is bounded. An unbounded Nef polyhedron can only be read by a Nef polyhedron parametrized with an extended kernel. It is also recommended to use the CGAL stream modifier *set_ascii_mode*.

The input operator and output operators of N

```
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>
```

```
template <class Nef_polyhedronTraits_3>
istream&                istream& in >> CGAL::Nef_polyhedron_3<Nef_polyhedronTraits_3>& N
```

See Also

CGAL::Nef_polyhedron_3<Traits> page [1033](#)
operator<< page [1053](#)

CGAL::Qt_widget_Nef_3<Nef_polyhedron_3

Definition

The class *Qt_widget_Nef_3* uses the OpenGL interface of Qt to display a *Nef_polyhedron_3*. Its purpose is to provide an easy to use viewer for *Nef_polyhedron_3*. There are no means provided to enhance the functionality of the viewer.

In addition to the functions inherited from the Qt class *OpenGLWidget* via, *Qt_widget_Nef_3* only has a single public constructor. For the usage of *Qt_widget_Nef_3* see the example below.

```
#include <CGAL/IO/Qt_widget_Nef_3.h>
```

Parameters

The template parameter expects an instantiation of *Nef_polyhedron_3<Traits>*.

Creation

```
Qt_widget_Nef_3<Nef_polyhedron_3> W( Nef_polyhedron_3 N);
```

Creates a widget *W* for displaying the 3D Nef polyhedron *N*.

See Also

CGAL::Nef_polyhedron_3<Traits> page [1033](#)

Example

This example reads a 3D Nef polyhedron from standard input and displays it in a Qt widget.

```
// Copyright (c) 2002 Max-Planck-Institute Saarbruecken (Germany)
// All rights reserved.
//
// This file is part of CGAL (www.cgal.org); you may redistribute it under
// the terms of the Q Public License version 1.0.
// See the file LICENSE.QPL distributed with CGAL.
//
// Licensees holding a valid commercial license may use this file in
// accordance with the commercial license agreement provided with the software.
//
// This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
// WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
//
// $URL: svn+ssh://scm.gforge.inria.fr/svn/cgal/branches/CGAL-3.2-branch/Nef_3/demo/Nef_3/visualization_S
// $Id: visualization_SNC.C 29577 2006-03-17 12:11:37Z hachenb $
//
```

```

//
// Author(s)      : Peter Hachenberger

#include <CGAL/basic.h>
#ifdef CGAL_USE_QT
#include <iostream>
int main(int, char*){
    std::cout << "Sorry, this demo needs QT..." << std::endl; return 0;}
#else
#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Nef_polyhedron_3.h>
#include <CGAL/IO/Nef_polyhedron_iostream_3.h>
#include <CGAL/IO/Qt_widget_Nef_3.h>
#include <qapplication.h>

typedef CGAL::Homogeneous<CGAL::Gmpz> Kernel;
typedef CGAL::Nef_polyhedron_3<Kernel> Nef_polyhedron_3;

int main(int argc, char* argv[]) {
    Nef_polyhedron_3 N;
    std::cin >> N;

    QApplication a(argc, argv);
    CGAL::Qt_widget_Nef_3<Nef_polyhedron_3>* w =
        new CGAL::Qt_widget_Nef_3<Nef_polyhedron_3>(N);
    a.setMainWidget(w);
    w->show();
    return a.exec();
}
#endif

```


Chapter 16

2D Straight Skeleton and Polygon Offsetting

Fernando Cacciola

Contents

16.1 Definitions	1057
16.1.1 2D Contour	1057
16.1.2 2D Polygon with Holes	1058
16.1.3 Inward Offset of a Non-degenerate Strictly-Simple Polygon with Holes	1058
16.1.4 Straight Skeleton of a 2D Non-degenerate Strictly-Simple Polygon with Holes	1059
16.2 Representation	1063
16.3 API	1066
16.3.1 Exterior Skeletons and Exterior Offset contours	1067
16.3.2 Example	1067
16.4 Straight Skeletons, Medial Axis and Voronoi Diagrams	1071
16.5 Usages of the Straight Skeletons	1072
16.6 Straight Skeleton of a General Figure in the Plane	1072

16.1 Definitions

16.1.1 2D Contour

A *2D contour* is a closed sequence (a cycle) of 3 or more *connected 2D oriented straight line segments* called *contour edges*. The endpoints of the contour edges are called *vertices*. Each contour edge shares its endpoints with at least two other contour edges.

If the edges intersect only at the vertices and at most are coincident along a line but do not *cross* one another, the contour is classified as *simple*.

A contour is topologically equivalent to a *disk* and if it is simple, is said to be a *Jordan Curve*.

Contours partition the plane in two open regions: one bounded and one unbounded. If the bounded region of a contour is a *singly-connected set*, the contour is said to be *strictly-simple*.

The *Orientation* of a contour is given by the order of the vertices around the region they bound. It can be *Clockwise* (CCW) or *Counter-clockwise* (CW).

The *bounded side* of a contour edge is the side facing the bounded region of the contour. If the contour is oriented CCW, the bounded side of an edge is its left side.

A contour with a null edge (a segment of length zero given by two consecutive coincident vertices), or with edges not connected to the bounded region (an antenna: 2 consecutive edges going forth and back along the same line), is said to be *degenerate* (collinear edges are *not* considered a degeneracy).

16.1.2 2D Polygon with Holes

A 2D *polygon* is a contour.

A 2D *polygon with holes* is a contour, called the *outer contour*, having zero or more contours, called *inner contours*, or *holes*, in its bounded region. The intersection of the bounded region of the outer contour and the unbounded regions of each inner contour is the *interior* of the polygon with holes. The orientation of the holes must be opposite to the orientation of the outer contour and there cannot be any intersection among any contour. A hole cannot be in the bounded region of any other hole.

A polygon with holes is strictly-simple if its interior is a singly-connected set.

The orientation of a polygon with holes is the orientation of its outer contour. The bounded side of *any* edge, whether of the outer contour or a hole, is the *same* for all edges. That is, if the outer contour is oriented CCW and the holes CW, both contour and hole edges face the polygon interior to their left.

Throughout the rest of this chapter the term *polygon* will be used as a shortcut for *polygon with holes*.

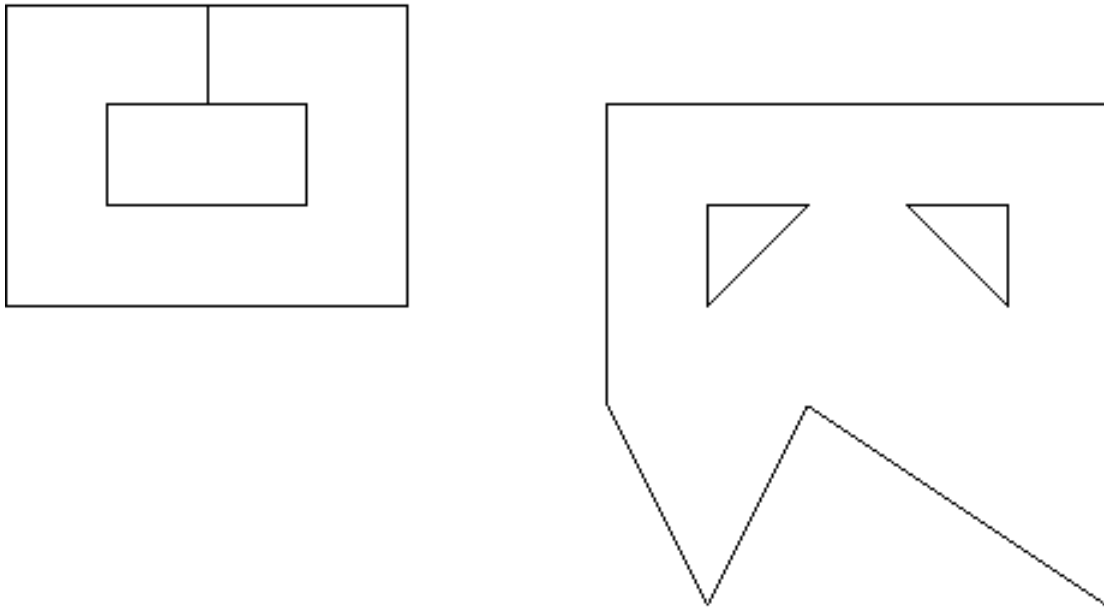


Figure 16.1: Examples of strictly simple polygons: One with no holes and two edges coincident (left) and one with 2 holes (right).

16.1.3 Inward Offset of a Non-degenerate Strictly-Simple Polygon with Holes

For any 2D non-degenerate strictly-simple polygon with holes called the *source*, there can exist a *set* of 0, 1 or more *inward offset polygons with holes*, or just offset polygons for short, at some euclidean distance $t > 0$

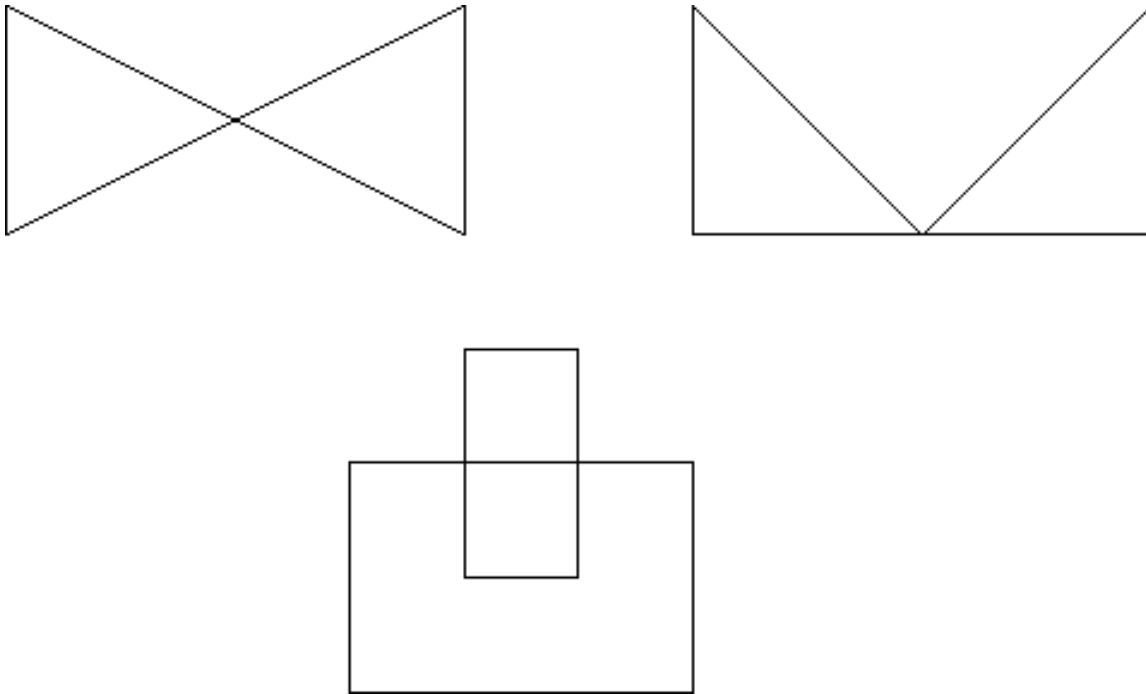


Figure 16.2: Examples of non-simple polygons: One folding into itself, that is, non-planar (left), one with a vertex touching an edge (right), and one with a hole crossing into the outside (bottom)

(each being strictly simple and non-degenerate). Any contour edge of such offset polygon, called an *offset edge* corresponds to *some* contour edge of the source polygon, called its *source edge*. An offset edge is parallel to its source edge and has the same orientation. The Euclidean distance between the *lines* supporting an offset edge and its source edge is exactly t .

An offset edge is always located to the bounded side of its source edge (which is an oriented straight line segment).

An offset polygon can have less, equal or more sides as its source polygon.

If the source polygon has no holes, no offset polygon has holes. If the source polygon has holes, any of the offset polygons can have holes itself, but it might as well have no holes at all (if the distance is sufficiently large).

Each offset polygon has the same orientation as the source polygon.

16.1.4 Straight Skeleton of a 2D Non-degenerate Strictly-Simple Polygon with Holes

The *2D straight skeleton* of a non-degenerate strictly-simple polygon with holes [AAAG95] is a special partitioning of the polygon interior into *straight skeleton regions* corresponding to the monotone areas traced by a continuous *inward offsetting* of the contour edges. Each region corresponds to exactly 1 contour edge.

These regions are bounded by angular bisectors of the supporting lines of the contour edges and each such region is itself a non-convex non-degenerate strictly-simple polygon.

Angular Bisecting Lines and Offset Bisectors

Given two points and a line passing through them, the perpendicular line passing through the midpoint is the bisecting line (or bisector) of those points.

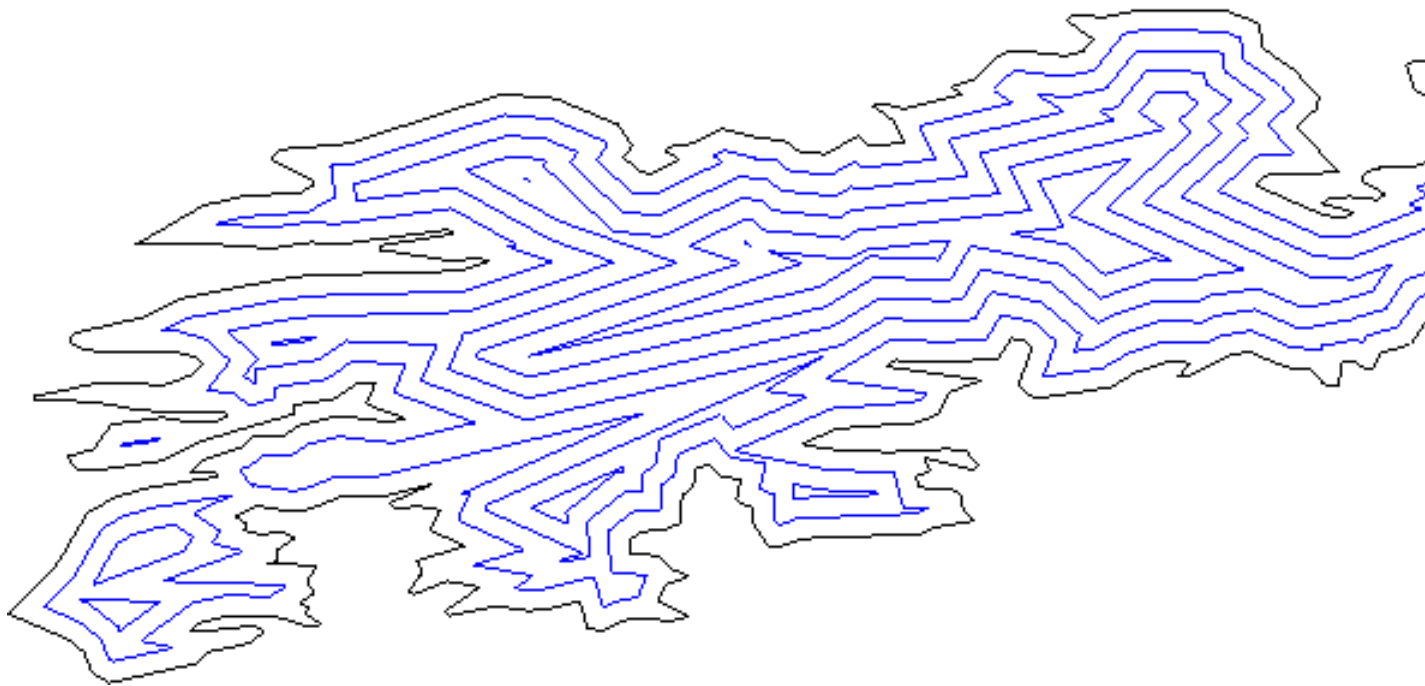


Figure 16.3: Offset contours of a sample polygon

Two non-parallel lines, intersecting at a point, are bisected by two other lines passing through that intersection point.

Two parallel lines are bisected by another parallel line placed halfway in between.

Given just one line, any perpendicular line can be considered the bisecting line (any bisector of any two points along the single line).

The bisecting lines of two edges are the lines bisecting the supporting lines of the edges (if the edges are parallel or collinear, there is just one bisecting line).

The halfplane to the bounded side of the line supporting a contour edge is called the *offset zone* of the contour edge.

Given any number of contour edges (not necessarily consecutive), the intersection of their offset zones is called their *combined offset zone*.

Any two contour edges define an *offset bisector*, as follows: If the edges are non-parallel, their bisecting lines can be decomposed as 4 rays originating at the intersection of the supporting lines. Only one of these rays is contained in the combined offset zone of the edges (which one depends on the possible combinations of orientations). This ray is the offset bisector of the non-parallel contour edges.

If the edges are parallel (but not collinear) and have opposite orientation, the entire and unique bisecting line is their offset bisector. If the edges are parallel but have the same orientation, there is no offset bisector between them.

If the edges are collinear and have the same orientation, their offset bisector is given by a perpendicular ray to the left of the edges which originates at the midpoint of the combined complement of the edges. (The *complement* of an edge/segment are the two rays along its supporting line which are not the segment and the *combined*



Figure 16.4: Straight skeleton of a complex shaggy contour

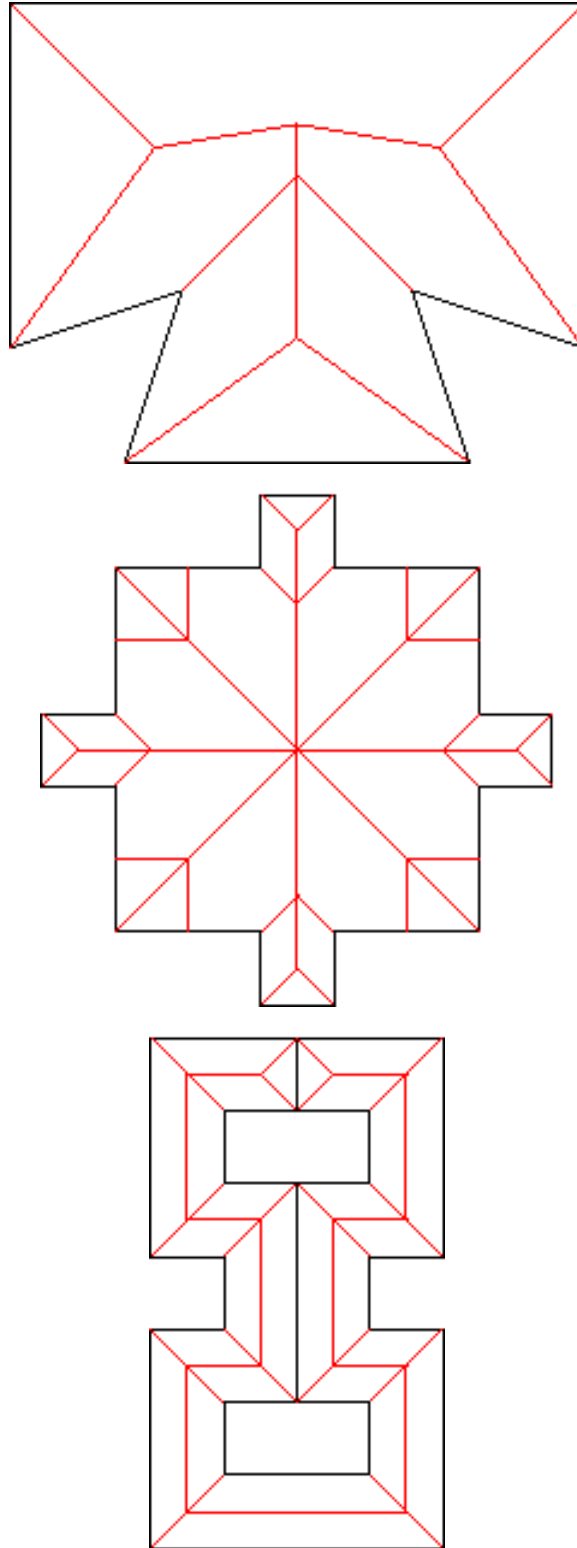


Figure 16.5: Other examples: A vertex-event (left), the case of several collinear edges (middle), and the case of a validly simple polygon with tangent edges (right).

complement of N collinear segments is the intersection of the complements of each segment). If the edges are collinear but have opposite orientation, there is no offset bisector between them.

Faces, Edges and Vertices

Each region of the partitioning defined by a straight skeleton is called a *face*. Each face is bounded by straight line segments, called *edges*. Exactly one edge per face is a *contour edge* (corresponds to a side of the polygon) and the rest of the edges, located in the interior of the polygon, are called *skeleton edges*, or *bisectors*.

The bisectors of the straight skeleton are segments of the offset bisectors as defined previously. Since an offset bisector is a ray of a bisecting line of 2 contour edges, each skeleton edge (or bisector) is uniquely given by two contour edges. These edges are called the *defining contour edges* of the bisector.

The intersection of the edges are called *vertices*. Although in a simple polygon, only 2 edges intersect at a vertex, in a straight skeleton, 3 or more edges intersect at any given vertex. That is, vertices in a straight skeleton have degree ≥ 3 .

A *contour vertex* is a vertex for which 2 of its incident edges are contour edges.

A *skeleton vertex* is a vertex whose incident edges are all skeleton edges.

A *contour bisector* is a bisector whose defining contour edges are consecutive. Such a bisector is incident upon 1 contour vertex and 1 skeleton vertex and touches the input polygon at exactly 1 endpoint.

An *inner bisector* is a bisector whose defining contour edges are not consecutive. Such a bisector is incident upon 2 skeleton vertices and is strictly contained in the interior of the polygon.

16.2 Representation

This CGAL package represents a straight skeleton as a specialized *Halfedge Data Structure* (HDS) whose vertices embeds 2D Points (see the *StraightSkeleton_2* concept in the reference manual for details).

Its halfedges, by considering the source and target points, implicitly embeds 2D oriented straight line segments (each halfedge per se does not embed a segment explicitly).

A face of the straight skeleton is represented as a face in the HDS. Both contour and skeleton edges are represented by pairs of opposite HDS halfedges, and both contour and skeleton vertices are represented by HDS vertices.

In a HDS, a border halfedge is a halfedge which is incident upon an unbounded face. In the case of the straight skeleton HDS, such border halfedges are oriented such that their left side faces outwards the polygon. Therefore, the opposite halfedge of any border halfedge is oriented such that its left side faces inward the polygon.

This CGAL package requires the input polygon (with holes) to be non-degenerate, strictly-simple, and oriented counter-clockwise.

The skeleton halfedges are oriented such that their *left* side faces inward the region they bound. That is, the vertices (both contour and skeleton) of a face are circulated in counter-clockwise order. There is one and only one contour halfedge incident upon any face.

The contours of the input polygon are traced by the border halfedges of the HDS (those facing outward), but in the opposite direction. That is, the vertices of the contours can only be traced from the straight skeleton data

structure by circulating the border halfedges, and the resulting vertex sequence will be reversed w.r.t the input vertex sequence.

A skeleton edge, according to the definition given in the previous section, is defined by 2 contour edges. In the representation, each one of the opposite halfedges that represent a skeleton edge is associated with one of the opposite halfedges that correspond to one of its defining contour edges. Thus, the 2 opposite halfedges of a skeleton edge link the edge to its 2 defining contour edges.

Starting from any border contour halfedge, circulating the structure walks through border counter halfedges and traces the vertices of the polygon's contours (in opposite order).

Starting from any non-border but contour halfedge, circulating the structure walks counter-clockwise around the face corresponding to that contour halfedge. The vertices around a face always describe a non-convex non-degenerate strictly-simple polygon.

A vertex is the intersection of contour and/or skeleton edges. Since a skeleton edge is defined by 2 contour edges, any vertex is itself defined by a unique set of contour edges. These are called the *defining contour edges* of the vertex.

A vertex is identified by its set of defining contour edges. Two vertices are distinct if they have differing sets of defining contour edges. Note that vertices can be distinct even if they are geometrically embedded at the same point.

The *degree* of a vertex is the number of halfedges around the vertex incident upon (pointing to) the vertex. As with any halfedge data structure, there is one outgoing halfedge for each incoming (incident) halfedge around a vertex. The degree of the vertex counts only incoming (incident) halfedges.

In a straight skeleton, the degree of a vertex is not only the number of incident halfedges around the vertex but also the number of defining contour halfedges. The vertex itself is the point where all the defining contour edges simultaneously collide.

Contour vertices have exactly two defining contour halfedges, which are the contour edges incident upon the vertex; and 3 incident halfedges. One and only one of the incident halfedges is a skeleton halfedge. The degree of a contour vertex is exactly 3.

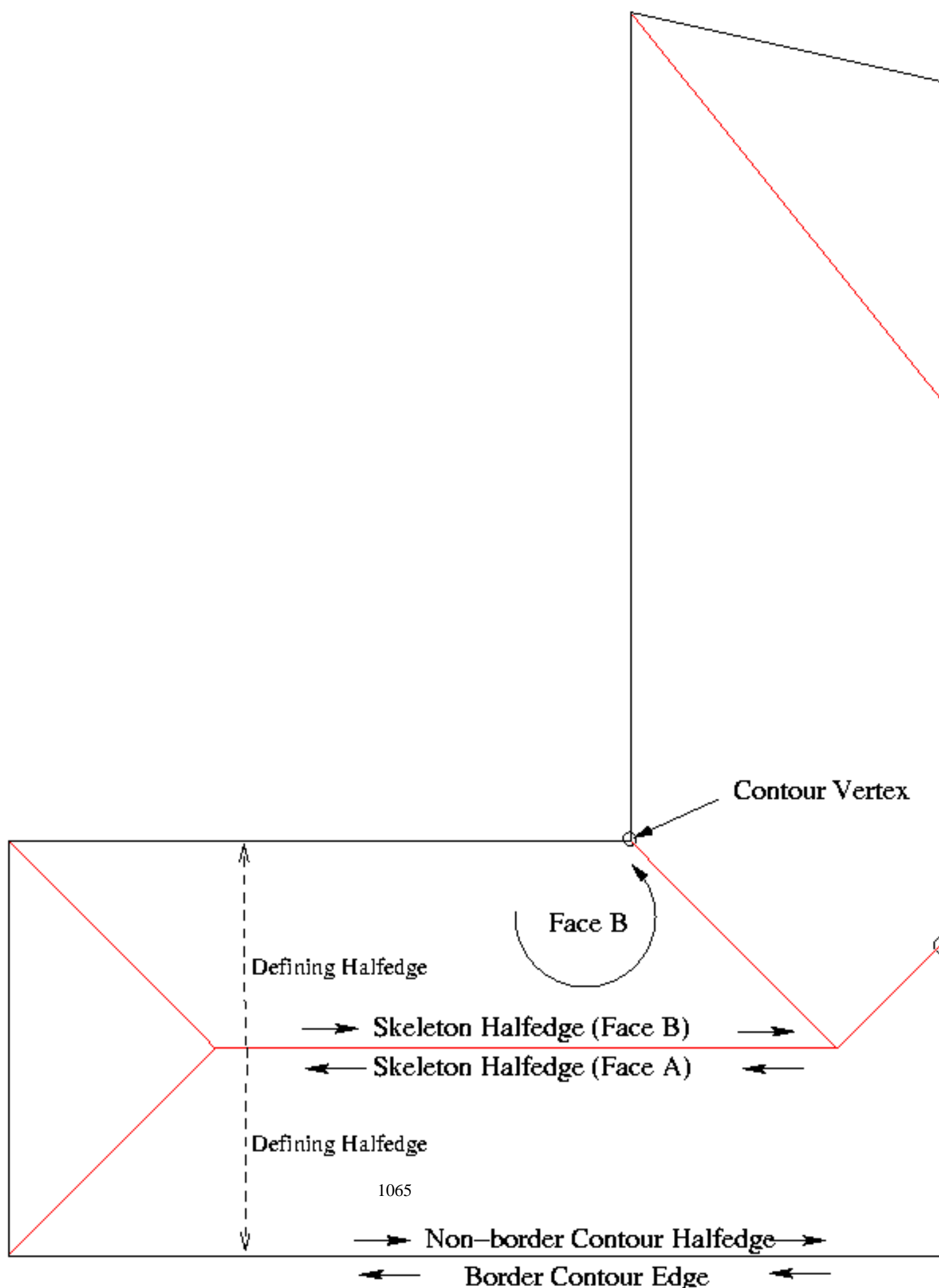
Skeleton vertices have at least 3 defining contour halfedges and 3 incident skeleton halfedges. If more than 3 edges collide simultaneously at the same point and time (like in any regular polygon with more than 3 sides), the corresponding skeleton vertex will have more than 3 defining contour halfedges and incident skeleton halfedges. That is, the degree of a skeleton vertex is ≥ 3 (the algorithm initially produces nodes of degree 3 but in the end all coincident nodes are merged to form higher degree nodes). All halfedges incident upon a skeleton vertex are skeleton halfedges.

The defining contour halfedges and incident halfedges around a vertex can be traced using the circulators provided by the vertex class. The degree of a vertex is not cached and cannot be directly obtained from the vertex, but you can calculate this number by manually counting the number of incident halfedges around the vertex.

Each vertex stores a 2D point and a *time*, which is the euclidean distance from the vertex's point to the *lines* supporting each of the defining contour edges of the vertex (the distance is the same to each line). Unless the polygon is convex, this distance is not equidistant to the edges, as in the case of a Medial Axis, therefore, the time of a skeleton vertex does not correspond to the distance from the polygon to the vertex (so it cannot be used to obtain the deep of a region in a shape, for instance).

If the polygon is convex, the straight skeleton is exactly equivalent to the polygon's voronoi diagram and each vertex time is the equidistance to the defining edges.

Contour vertices have time zero.



16.3 API

The straight skeleton data structure is defined by the *StraightSkeleton_2* concept and modeled in the *Straight_skeleton_2*<*Traits,Items,Alloc*> class.

The straight skeleton construction algorithm is encapsulated in the class *Straight_skeleton_builder_2*<*Gt,Ss*> which is parameterized on a geometric traits (class *Straight_skeleton_builder_traits*<*Kernel*>) and the Straight Skeleton class (*Ss*).

The offset contours construction algorithm is encapsulated in the class *Polygon_offset_builder_2*<*Ss,Gt,Container*> which is parameterized on the Straight Skeleton class (*Ss*), a geometric traits (class *Polygon_offset_builder_traits*<*Kernel*>) and a container type where the resulting offset polygons are generated.

To construct the straight skeleton of a polygon with holes the user must:

- (1) Instantiate the straight skeleton builder.
- (2) Enter one contour at a time, starting from the outer contour, via the method *enter_contour*. The input polygon with holes must be non-degenerate, strictly-simple and counter-clockwise oriented (see the definitions at the beginning of this chapter). Collinear edges are allowed. The insertion order of each hole is unimportant but the outer contour must be entered first.
- (3) Call *construct_skeleton* once *all* the contours have been entered. You cannot enter another contour once the skeleton has been constructed.

To construct a set of inward offset contours the user must:

- (1) Construct the straight skeleton of the source polygon with holes.
- (2) Instantiate the polygon offset builder passing in the straight skeleton as a parameter.
- (3) Call *construct_offset_contours* passing the desired offset distance and an output iterator that can store a *boost::shared_ptr* of *Container* instances into a resulting sequence (typically, a back insertion iterator)

Each element in the resulting sequence is an *offset contour*, given by a *boost::shared_ptr* holding a dynamically allocated instance of the *Container* type. Such a container can be any model of the *VertexContainer_2* concept, for example, a *CGAL::Polygon_2*, or just a *std::vector* of 2D points.

The resulting sequence of offset contours can contain both outer and inner contours. Each offset hole (inner offset contour) would logically belong in the interior of some of the outer offset contours. However, this algorithm returns a sequence of contours in arbitrary order and there is no indication whatsoever of the parental relationship between inner and outer contours.

On the other hand, each outer contour is counter-clockwise oriented while each hole is clockwise-oriented. And since offset contours do form simple polygons with holes, it is guaranteed that no hole will be inside another hole, no outer contour will be inside any other contour, and each hole will be inside exactly 1 outer contour.

Parental relationships are *not* automatically reconstructed by this algorithm because this relation is not directly given by the input polygon with holes and doing it *robustly* is a time-consuming operation.

A user can reconstruct the parental relationships as a post processing operation by testing each inner contour (which is identified by being clockwise) against each outer contour (identified as being counter-clockwise) for insideness.

This algorithm requires exact predicates but not exact constructions Therefore, the *Exact_predicates_inexact_constructions_kernel* should be used.

16.3.1 Exterior Skeletons and Exterior Offset contours

This CGAL package can only construct the straight skeleton and offset contours in the *interior* of a polygon with holes. However, constructing exterior skeletons and exterior offsets is possible:

Say you have some polygon made of 1 outer contour C0 and 1 hole C1, and you need to obtain some exterior offset contours.

The interior region of a polygon with holes is connected while the exterior region is not: there is an unbounded region outside the outer contour, and one bounded region inside each hole. To construct an offset contour you need to construct a straight skeleton. Thus, to construct exterior offset contours for a polygon with holes, you need to construct, *separately*, the exterior skeleton of the outer contour and the interior skeleton of each hole.

Constructing the interior skeleton of a hole is directly supported by this CGAL package; you just need to input the hole's vertices in reversed order as if it were an outer contour.

Constructing the exterior skeleton of the outer contour is possible by means of the following trick: place the contour as a hole of a big rectangle (call it *frame*). If the frame is sufficiently separated from the contour, the resulting skeleton will be practically equivalent to a *real* exterior skeleton.

To construct exterior offset contours in the inside of each hole you just use the skeleton constructed in the interior, and, if required, revert the orientation of each resulting offset contour.

Constructing exterior offset contours in the outside of the outer contour is just a little bit more involved: Since the contour is placed as a hole of a frame, you will always obtain 2 offset contours for any given distance; one is the offsetted frame and the other is the offsetted contour. Thus, from the resulting offset contour sequence, you always need to discard the offsetted frame, easily identified as the offset contour with the largest area.

It is necessary to place the frame sufficiently away from the contour. If it is not, it could occur that the outward offset contour collides and merges with the inward offset frame, resulting in 1 instead of 2 offset contours.

However, the proper separation between the contour and the frame is not directly given by the offset distance at which you want the offset contour. That distance must be at least the desired offset plus the largest euclidean distance between an offset vertex and its original.

This CGAL packages provides a helper function to compute the required separation: `compute_outer_frame_margin`

If you use this function to place the outer frame you are guaranteed to obtain an offset contour corresponding exclusively to the frame, which you can always identify as the one with the largest area and which you can simple remove from the result (to keep just the relevant outer contours).

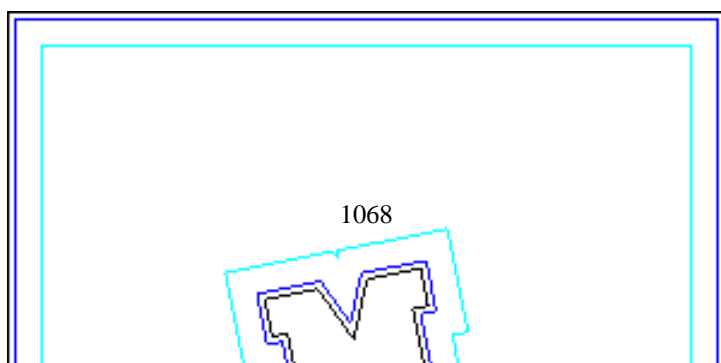
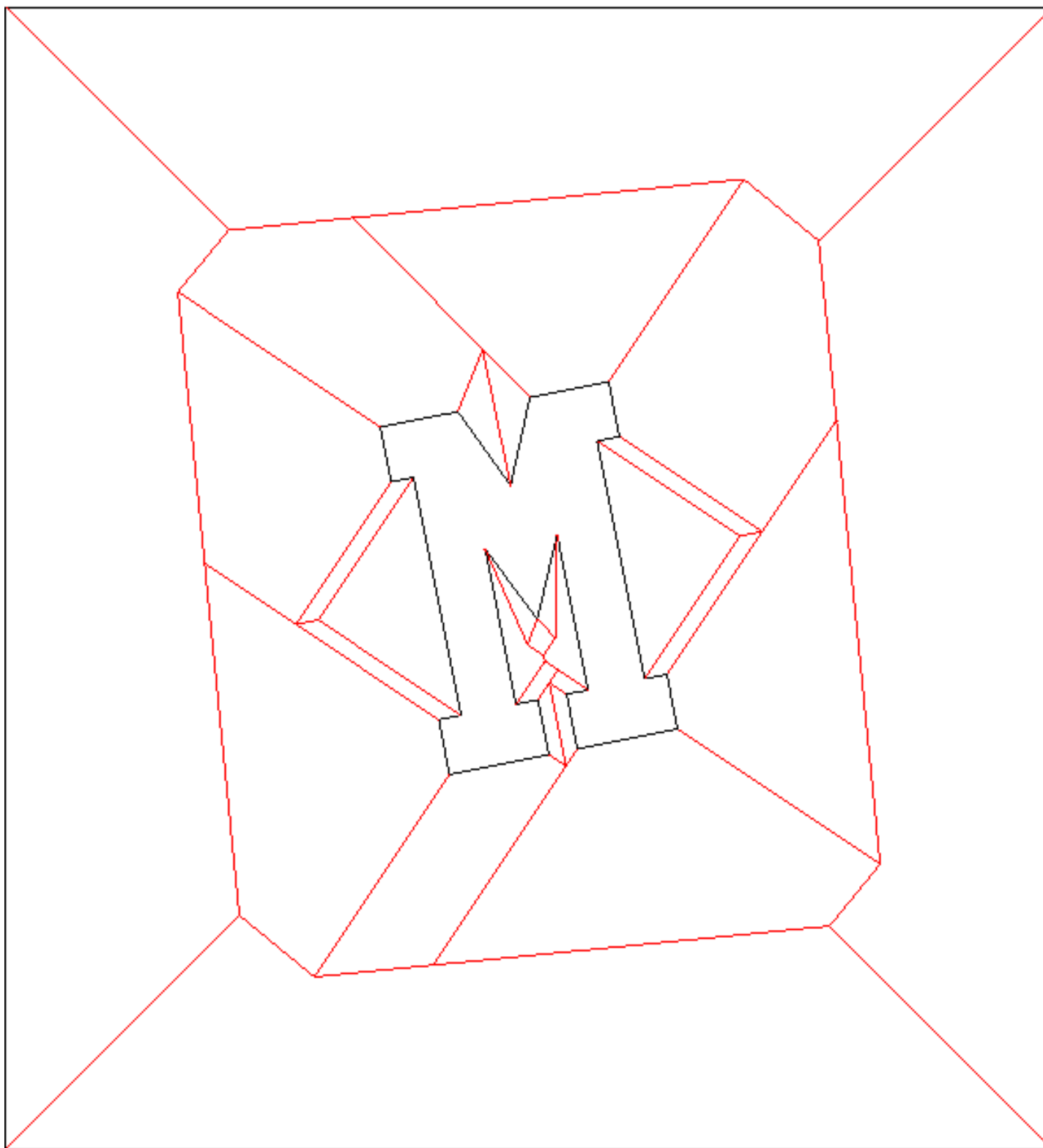
16.3.2 Example

```
#include<vector>
#include<iterator>
#include<iostream>
#include<iomanip>
#include<string>

#include<boost/shared_ptr.hpp>

#include<CGAL/basic.h>
#include<CGAL/Cartesian.h>
#include<CGAL/Polygon_2.h>
#include<CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include<CGAL/Straight_skeleton_builder_2.h>
#include<CGAL/Polygon_offset_builder_2.h>
#include<CGAL/compute_outer_frame_margin.h>

//
```



```

// This example illustrates how to use the CGAL Straight Skeleton package
// to construct an offset contour on the outside of a polygon
//

// This is the recommended kernel
typedef CGAL::Exact_predicates_inexact_constructions_kernel Kernel;

typedef Kernel::Point_2 Point_2;
typedef CGAL::Polygon_2<Kernel> Contour;
typedef boost::shared_ptr<Contour> ContourPtr;
typedef std::vector<ContourPtr> ContourSequence ;

typedef CGAL::Straight_skeleton_2<Kernel> Ss;

typedef Ss::Halfedge_iterator Halfedge_iterator;
typedef Ss::Halfedge_handle Halfedge_handle;
typedef Ss::Vertex_handle Vertex_handle;

typedef CGAL::Straight_skeleton_builder_traits_2<Kernel> SsBuilderTraits;
typedef CGAL::Straight_skeleton_builder_2<SsBuilderTraits,Ss> SsBuilder;

typedef CGAL::Polygon_offset_builder_traits_2<Kernel> OffsetBuilderTraits;
typedef CGAL::Polygon_offset_builder_2<Ss,OffsetBuilderTraits,Contour> OffsetBuilder;

int main()
{
    // A start-shaped polygon, oriented counter-clockwise as required for outer contours.
    Point_2 pts[] = { Point_2(-1,-1)
                     , Point_2(0,-12)
                     , Point_2(1,-1)
                     , Point_2(12,0)
                     , Point_2(1,1)
                     , Point_2(0,12)
                     , Point_2(-1,1)
                     , Point_2(-12,0)
                     } ;

    std::vector<Point_2> star(pts,pts+8);

    // We want an offset contour in the outside.
    // Since the package doesn't support that operation directly, we use the following trick:
    // (1) Place the polygon as a hole of a big outer frame.
    // (2) Construct the skeleton on the interior of that frame (with the polygon as a hole)
    // (3) Construc the offset contours
    // (4) Identify the offset contour that corresponds to the frame and remove it from the result

    double offset = 3 ; // The offset distance

    // First we need to determine the proper separation between the polygon and the frame.
    // We use this helper function provided in the package.
    boost::optional<double> margin = CGAL::compute_outer_frame_margin(star.begin(),star.end(),offset);

    // Proceed only if the margin was computed (an extremely sharp corner might cause overflow)

```

```

if ( margin )
{
    // Get the bbox of the polygon
    CGAL::Bbox_2 bbox = CGAL::bbox_2(star.begin(),star.end());

    // Compute the boundaries of the frame
    double fxmin = bbox.xmin() - *margin ;
    double fxmax = bbox.xmax() + *margin ;
    double fymin = bbox.ymin() - *margin ;
    double fymax = bbox.ymax() + *margin ;

    // Create the rectangular frame
    Point_2 frame[4]= { Point_2(fxmin,fymin)
                        , Point_2(fxmax,fymin)
                        , Point_2(fxmax,fymax)
                        , Point_2(fxmin,fymax)
                      } ;

    // Instantiate the skeleton builder
    SsBuilder ssb ;

    // Enter the frame
    ssb.enter_contour(frame,frame+4);

    // Enter the polygon as a hole of the frame (NOTE: as it is a hole we insert it in the opposite order)
    ssb.enter_contour(star.rbegin(),star.rend());

    // Construct the skeleton
    boost::shared_ptr<Ss> ss = ssb.construct_skeleton();

    // Proceed only if the skeleton was correctly constructed.
    if ( ss )
    {
        // Instantiate the container of offset contours
        ContourSequence offset_contours ;

        // Instantiate the offset builder with the skeleton
        OffsetBuilder ob(*ss);

        // Obtain the offset contours
        ob.construct_offset_contours(offset, std::back_inserter(offset_contours));

        // Locate the offset contour that corresponds to the frame
        // That must be the outmost offset contour, which in turn must be the one
        // with the largest unsigned area.
        ContourSequence::iterator f = offset_contours.end();
        double lLargestArea = 0.0 ;
        for (ContourSequence::iterator i = offset_contours.begin(); i != offset_contours.end(); ++ i )
        {
            double lArea = CGAL::abs( (*i)->area() ) ; //Take abs() as Polygon_2::area() is signed.
            if ( lArea > lLargestArea )
            {
                f = i ;
                lLargestArea = lArea ;
            }
        }
    }
}

```

```

    }
}

// Remove the offset contour that corresponds to the frame.
offset_contours.erase(f);

// Print out the skeleton
Halfedge_handle null_halfedge ;
Vertex_handle    null_vertex ;

// Dump the edges of the skeleton
for ( Halfedge_iterator i = ss->halfedges_begin(); i != ss->halfedges_end(); ++i )
{
    std::string edge_type = (i->is_bisector())? "bisector" : "contour";
    Vertex_handle s = i->opposite()->vertex();
    Vertex_handle t = i->vertex();
    std::cout << "(" << s->point() << ")->(" << t->point() << ")" << edge_type << std::endl;
}

// Dump the generated offset polygons

std::cout << offset_contours.size() << " offset contours obtained\n" ;

for (ContourSequence::const_iterator i = offset_contours.begin(); i != offset_contours.end(); ++ i )
{
    // Each element in the offset_contours sequence is a shared pointer to a Polygon_2 instance.

    std::cout << (*i)->size() << " vertices in offset contour\n" ;

    for (Contour::Vertex_const_iterator j = (*i)->vertices_begin(); j != (*i)->vertices_end(); ++ j )
        std::cout << "(" << j->x() << "," << j->y() << ")" << std::endl ;
}
}
}

return 0;
}

```

16.4 Straight Skeletons, Medial Axis and Voronoi Diagrams

The straight skeleton of a polygon is similar to the medial axis and the voronoi diagram of a polygon in the way it partitions it; however, unlike the medial axis and voronoi diagram, the bisectors are not equidistant to its defining edges but to the supporting lines of such edges. As a result, Straight Skeleton bisectors might not be located in the center of the polygon and so cannot be regarded as a proper Medial Axis in its geometrical meaning.

On the other hand, only reflex vertices (whose internal angle $> \pi$) are the source of deviations of the bisectors from its center location. Therefore, for convex polygons, the straight skeleton, the medial axis and the Voronoi diagram are exactly equivalent, and, if a non-convex polygon contains only vertices of low reflexivity, the straight skeleton bisectors will be placed nearly equidistant to their defining edges, producing a straight skeleton pretty much alike a proper medial axis.

16.5 Usages of the Straight Skeletons

The most natural usage of straight skeletons is offsetting: growing and shrinking polygons (provided by this CGAL package).

Another usage, perhaps its very first, is roof design: The straight skeleton of a polygonal roof directly gives the layout of each tent. If each skeleton edge is lifted from the plane a height equal to its offset distance, the resulting roof is "correct" in that water will always fall down to the contour edges (roof border) regardless of where in the roof it falls. [LD03] gives an algorithm for roof design based on the straight skeleton.

Just like medial axes, 2D straight skeletons can also be used for 2D shape description and matching. Essentially, all the applications of image-based skeletonization (for which there is a vast literature) are also direct applications of the straight skeleton, specially since skeleton edges are simply straight line segments.

Consider the subgraph formed only by *inner bisectors* (that is, only the skeleton halfedges which are not incident upon a contour vertex). Call this subgraph a *skeleton axis*. Each node in the skeleton axis whose degree is ≥ 3 roots more than one skeleton tree. Each skeleton tree roughly corresponds to a region in the input topologically equivalent to a rectangle; that is, without branches. For example, a simple letter "H" would contain 2 higher degree nodes separating the skeleton axis in 5 trees; while the letter "@" would contain just 1 higher degree node separating the skeleton axis in 2 curly trees.

Since a skeleton edge is a 2D straight line, each branch in a skeleton tree is a polyline. Thus, the path-length of the tree can be directly computed. Furthermore, the polyline for a particular tree can be interpolated to obtain curve-related information.

Pruning each skeleton tree cutting off branches whose length is below some threshold; or smoothing a given branch, can be used to reconstruct the polygon without undesired details, or fit into a particular canonical shape.

Each skeleton edge in a skeleton branch is associated with 2 contour edges which are facing each other. If the polygon has a bottleneck (it almost touches itself), a search in the skeleton graph measuring the distance between each pair of contour edges will reveal the location of the bottleneck, allowing you to cut the shape in two. Likewise, if two shapes are too close to each other along some part of their boundaries (a near contact zone), a similar search in an exterior skeleton of the two shapes at once would reveal the parts of near contact, allowing you to stitch the shapes. These *cut and stitch* operations can be directly executed in the straight skeleton itself instead of the input polygon (because the straight skeleton contains a graph of the connected contour edges).

16.6 Straight Skeleton of a General Figure in the Plane

A straight skeleton can also be defined for a general multiply-connected planar directed straight-line graph [AA95] by considering all the edges as embedded in an unbounded region. The only difference is that in this case some faces will be only partially bounded.

The current version of this CGAL package can only construct the straight skeleton in the interior of a simple polygon with holes, that is it doesn't handle general polygonal figures in the plane.

2D Straight Skeleton and Polygon Offsetting

Reference Manual

This chapter introduces the concepts and classes that correspond to the CGAL *2D Straight Skeleton* package. The packages provides an algorithm for the construction of the straight skeleton in the interior of a simple polygon with holes and an algorithm for the construction of inward offset contours based on the straight skeleton.

16.7 Classified Reference Pages

Concepts

StraightSkeletonVertex_2.....	page 1076
StraightSkeletonHalfedge_2	page 1079
StraightSkeleton_2.....	page 1075
StraightSkeletonBuilderTraits_2.....	page 1081
PolygonOffsetBuilderTraits_2.....	page 1084
VertexContainer_2.....	page 1086

Classes

<i>CGAL::Straight_skeleton_vertex_base_2</i> <Refs,Point,FT>	page 1088
<i>CGAL::Straight_skeleton_halfedge_base_2</i> <Refs>	page 1089
<i>CGAL::Straight_skeleton_2</i> <Traits,Items,Alloc>	page 1087
<i>CGAL::Straight_skeleton_builder_traits_2</i> <Kernel>	page 1090
<i>CGAL::Straight_skeleton_builder_2</i> <Gt,Ss>	page 1091
<i>CGAL::Polygon_offset_builder_traits_2</i> <Kernel>	page 1094
<i>CGAL::Polygon_offset_builder_2</i> <Ss,Gt,Container>	page 1095

16.8 Alphabetical List of Reference Pages

<i>compute_outer_frame_margin</i>	page 1098
<i>PolygonOffsetBuilderTraits_2</i>	page 1084

<i>Polygon_offset_builder_2</i> <Ss,Gt,Container>	page 1095
<i>Polygon_offset_builder_traits_2</i> <Kernel>	page 1094
<i>StraightSkeletonBuilderTraits_2</i>	page 1081
<i>StraightSkeletonHalfedge_2</i>	page 1079
<i>StraightSkeletonVertex_2</i>	page 1076
<i>StraightSkeleton_2</i>	page 1075
<i>Straight_skeleton_2</i> <Traits,Items,Alloc>	page 1087
<i>Straight_skeleton_builder_2</i> <Gt,Ss>	page 1091
<i>Straight_skeleton_builder_traits_2</i> <Kernel>	page 1090
<i>Straight_skeleton_halfedge_base_2</i> <Refs>	page 1089
<i>Straight_skeleton_vertex_base_2</i> <Refs,Point,FT>	page 1088
<i>VertexContainer_2</i>	page 1086

StraightSkeleton_2

Definition

The concept `StraightSkeleton_2` describes the requirements for the data structure used to represent a straight skeleton.

Refines

HalfedgeDS

Types

StraightSkeleton_2::Vertex

A model of the *StraightSkeletonVertex_2* concept used to represent the vertices of the straight skeleton

StraightSkeleton_2::Halfedge

A model of the *StraightSkeletonHalfedge_2* concept used to represent the halfedges of the straight skeleton

StraightSkeleton_2::Face

Any model of the *HalfedgeDSFace* concept

Has Models

CGAL::Straight_skeleton_2<Traits,Items,Alloc>.

This concept explicitly protects all the modifying operations of the base *HalfedgeDS* concept. Only the algorithm classes, or clients explicitly bypassing the protection mechanism, can modify a straight skeleton.

StraightSkeletonVertex_2

Definition

The concept `StraightSkeletonVertex_2` describes the requirements for the vertex type of the *StraightSkeleton_2* concept. It is a refinement of the *HalfedgeDSVertex* concept with support for storage of the incident halfedge. The `StraightSkeletonVertex_2` concept requires the geometric embedding to be a 2D point.

Refines

HalfedgeDSVertex

Types

<i>StraightSkeletonVertex_2:: Point_2</i>	The type of the 2D point being the geometric embedding of the vertex
<i>StraightSkeletonVertex_2:: FT</i>	A model of the <i>SqrtFieldNumberType</i> concept representing the time of a vertex (an Euclidean distance)
<i>StraightSkeletonVertex_2:: Halfedge_around_vertex_const_circulator</i> <i>StraightSkeletonVertex_2:: Halfedge_around_vertex_circulator</i>	The circulator type used to visit all the incident halfedges around a vertex
<i>StraightSkeletonVertex_2:: Defining_contour_halfedge_const_circulator</i> <i>StraightSkeletonVertex_2:: Defining_contour_halfedge_circulator</i>	The circulator type used to visit all the defining contour halfedges of a vertex

Creation

<i>StraightSkeletonVertex_2</i> <i>v</i> ;	Default constructor
<i>StraightSkeletonVertex_2</i> <i>v</i> (<i>int id</i> , <i>Point_2 p</i>);	Constructs a contour vertex with ID number <i>id</i> , at the point <i>p</i>
<i>StraightSkeletonVertex_2</i> <i>v</i> (<i>int id</i> , <i>Point_2 p</i> , <i>FT time</i>);	Constructs a skeleton vertex with ID number <i>id</i> , at point <i>p</i> and time <i>time</i> .

Access Functions

<i>int</i>	<i>v.id()</i>	The ID of the vertex.
<i>Point_2</i>	<i>v.point()</i>	The vertex point.
<i>FT</i>	<i>v.time()</i>	The time of the vertex: the distance from the vertex point to the lines supporting the defining contour edges
<i>Halfedge_handle</i>	<i>v.primary_bisector()</i>	

Halfedge_const_handle

v.primary_bisector() Returns the skeleton halfedge incident upon the vertex (called the *primary* bisector).

Halfedge_around_vertex_circulator

v.halfedge_around_vertex_begin()

Halfedge_around_vertex_const_circulator

v.halfedge_around_vertex_begin()

Returns a bi-directional circulator pointing to one of the incident halfedges (which one is unspecified).

There will always be as many incident halfedges as the degree of the vertex.

If this is a *contour* vertex, its degree is exactly 3, and from the halfedges pointed to by the circulator, 2 are contour and 1 is a bisector.

If this is an *skeleton* vertex, its degree is at least 3 and all of the halfedges pointed to by the circulator are bisectors.

Each halfedge pointed to by this circulator is the one which is oriented towards the vertex (according to the geometric embedding).

Defining_contour_halfedge_circulator

v.defining_contour_halfedges_begin()

Defining_contour_halfedge_const_circulator

v.defining_contour_halfedges_begin()

Returns a bi-directional circulator pointing to one of the defining contour halfedges of the vertex (which one is unspecified).

There will always be as many incident defining contour halfedges as the degree of the vertex.

Each halfedge pointed to by this circulator is the one having its left side facing inwards (which happens to be the contour halfedge for which *is_border()* is *false*).

Queries

<i>bool</i>	<i>v.is_contour()</i>	Returns <i>true</i> iff this is a contour vertex.
<i>bool</i>	<i>v.is_skeleton()</i>	Returns <i>true</i> iff this is a skeleton vertex.

Has Models

CGAL::Straight_skeleton_vertex_base_2<Refs,Point,FT>.

See Also

StraightSkeleton_2

StraightSkeletonHalfedge_2

CGAL::Straight_skeleton_vertex_base_2<Refs,Point,FT>
CGAL::Straight_skeleton_halfedge_base_2<Refs>

StraightSkeletonHalfedge_2

Definition

The concept `StraightSkeletonHalfedge_2` describes the requirements for the halfedge type of the `StraightSkeleton_2` concept. It is a refinement of the `HalfedgeDSHalfedge` concept. The `StraightSkeletonHalfedge_2` concept requires no geometric embedding at all. The only geometric embedding used by the Straight Skeleton Data Structure are the 2D points in the contour and skeleton vertices. However, for any halfedge, there is a 2D segment implicitly given by its *source* and *target* vertices.

Refines

`HalfedgeDSHalfedge`

Creation

<code>StraightSkeletonHalfedge_2 h;</code>	Default Constructor.
<code>StraightSkeletonHalfedge_2 h(int id);</code>	Constructs a halfedge with ID <i>id</i> . It is the links to other halfedges what determines if this is a contour edge, a contour-skeleton edge or an inner-skeleton edge.

Access Functions

<code>Halfedge_handle</code>	<code>h.defining_contour_edge()</code>	
<code>Halfedge_const_handle</code>	<code>h.defining_contour_edge()</code>	If this is a bisector halfedge, returns a handle to the inward-facing (non-border) contour halfedge corresponding to the defining contour edge which is to its left; if this is a contour halfedge, returns a handle to itself if <i>is_border()</i> is <i>false</i> , or to its opposite if it is <i>true</i> .

Queries

<code>bool</code>	<code>h.is_bisector()</code>	Returns <i>true</i> iff this is a bisector (or skeleton) halfedge (i.e. is not a contour halfedge).
<code>bool</code>	<code>h.is_inner_bisector()</code>	Returns <i>true</i> iff this is a bisector and is inner (i.e. is not incident upon a contour vertex).

Has Models

`CGAL::Straight_skeleton_halfedge_2<Refs>.`

See Also

StraightSkeleton_2

StraightSkeletonHalfedge_2

CGAL::Straight_skeleton_vertex_base_2<Refs,Point,FT>

CGAL::Straight_skeleton_halfedge_base_2<Refs>

StraightSkeletonBuilderTraits_2

Definition

The concept `StraightSkeletonBuilderTraits_2` describes the requirements for the geometric traits class required by the algorithm class `Straight_skeleton_builder_2<Gt, Ss>`.

Types

<code>StraightSkeletonBuilderTraits_2:: Kernel</code>	A model of the <i>Kernel</i> concept.
<code>StraightSkeletonBuilderTraits_2:: FT</code>	A <i>SqrtFieldNumberType</i> provided by the kernel. This type is used to represent the coordinates of the input points, of the skeleton nodes and as the event time stored in the skeleton nodes.
<code>boost::tuple<FT, FT> Vertex;</code>	A pair of (x,y) coordinates representing a 2D cartesian point.
<code>boost::tuple<Vertex, Vertex></code>	
<code>Edge;</code>	A pair of vertices representing an edge
<code>boost::tuple<Edge, Edge, Edge></code>	
<code>EdgeTriple;</code>	A triple of edges representing an event
<code>StraightSkeletonBuilderTraits_2:: Equal_2</code>	
	A predicate object type being a model of the <i>Kernel::Equal_2</i> function object concept.
<code>StraightSkeletonBuilderTraits_2:: Left_turn_2</code>	
	A predicate object type being a model of the <i>Kernel::LeftTurn_2</i> function object concept.
<code>StraightSkeletonBuilderTraits_2:: Collinear_2</code>	
	A predicate object type being a model of the <i>Kernel::Collinear_2</i> function object concept.
<code>StraightSkeletonBuilderTraits_2:: Do_ss_event_exist_2</code>	
	A predicate object type. Must provide <i>bool operator()(EdgeTriple const& et) const</i> , which determines if, given the 3 <i>oriented</i> lines defined by the 3 input edges (3 pair of points), there exist an Euclidean distance $t \geq 0$ for which the corresponding 3 <i>offset lines at t</i> (parallel lines at an Euclidean distance of t) intersect in a single point. <i>Precondition:</i> each edge in the triple must properly define an oriented line, that is, such points cannot be coincident.

StraightSkeletonBuilderTraits_2:: Compare_ss_event_times_2

A predicate object type.

Must provide *Comparison_result operator()(EdgeTriple const& x, EdgeTriple const& y) const*, which compares the *times* for the events determined by the edge-triples *x* and *y*.

The time of an event given by an edge triple (which defines 3 oriented lines) is the Euclidean distance *t* at which the corresponding offset lines at *t* intersect in a single point.

Precondition: *x* and *y* must be edge-triples corresponding to events that actually exist (as determined by the predicate *Exist_sls_event_2*).

StraightSkeletonBuilderTraits_2:: Compare_ss_event_distance_to_seed_2

A predicate object type.

Must provide *Comparison_result operator()(Point_2 const& p, EdgeTriple const& x, EdgeTriple const& y)*, which compares the Euclidean distance of the points of event for *x* and *y* to the point *p*.

The point of an event given by an edge triple (which defines 3 oriented lines) is the intersection point of the 3 corresponding offset lines at the time *t* of the event.

It must also provide *Comparison_result operator(EdgeTriple const& seed, EdgeTriple const& x, EdgeTriple const& y)()* which makes the same comparison as the first overload but where the seed point is given implicitly as the point of event for *seed*.

Precondition: *seed*, *x* and *y* must be edge-triples corresponding to events that actually exist (as determined by the predicate *Exist_sls_event_2*).

StraightSkeletonBuilderTraits_2:: Is_ss_event_inside_offset_zone_2

A predicate object type.

Must provide *bool operator()(EdgeTriple const& e, EdgeTriple const& zone)*, which determines if the point of event for *e* is inside the *offset zone* defined by the 3 *oriented* lines given by *zone*.

An offset zone given by 3 *oriented* lines is the intersection of the halfplanes to the left of each oriented line.

Precondition: *e* must be an edge-triple corresponding to an event that actually exist (as determined by the predicate *Exist_sls_event_2*), and the 3 oriented lines given by *zone* must be well defined (no point-pair can have coincident points).

StraightSkeletonBuilderTraits_2:: Are_ss_events_simultaneous_2

A predicate object type.

Must provide *bool operator()(EdgeTriple const& x, EdgeTriple const& y)*, which determines if the events given by *x* and *y* are coincident in time and space; that is, both triples of offset lines intersect at the same point and at the same Euclidean distance from their sources.

Precondition: *x* and *y* must be edge-triples corresponding to events that actually exist (as determined by the predicate *Exist_sls_event_2*).

StraightSkeletonBuilderTraits_2:: Construct_ss_event_time_and_point_2

A construction object type.

Must provide *boost::tuple< boost::optional<FT>, boost::optional<Point_2> > operator()(EdgeTriple const& e)*, which given the 3 *oriented* lines defined by the 3 input edges (3 pair of points), returns the Euclidean distance $t \geq 0$ and intersection point at which the corresponding 3 *offset lines* at t intersect.

If the values cannot be computed, not even approximately (because of overflow for instance), an empty optional must be returned.

Precondition: e must be an edge-triple corresponding to an event that actually exist (as determined by the predicate *Exist_sls_event_2*).

StraightSkeletonBuilderTraits_2:: Construct_ss_vertex_2

A construction object type.

Must provide *Vertex operator()(Point_2 const& p)*, which given a *Point_2* p returns a Vertex encapsulating the corresponding (x,y) pair of *cartesian* coordinates.

StraightSkeletonBuilderTraits_2:: Construct_ss_edge_2

A construction object type.

Must provide *Edge operator()(Point_2 const& s, Point_2 const& t)*, which given source and target points s and t returns an Edge encapsulating the corresponding input segment (in *cartesian* coordinates.)

StraightSkeletonBuilderTraits_2:: Construct_ss_triedge_2

A construction object type.

Must provide *Triedge operator()(Edge const& e0, Edge const& e1, Edge const& e2)*, which given the 3 edges that define an event, $e0$, $e1$ and $e2$, returns a Triedge encapsulating them.

Has Models

CGAL::Straight_skeleton_builder_traits_2<K>.

See Also

CGAL::Straight_skeleton_builder_2<Gt,Ss>

CGAL::Straight_skeleton_builder_traits_2<K>

PolygonOffsetBuilderTraits_2

Definition

The concept `PolygonOffsetBuilderTraits_2` describes the requirements for the geometric traits class required by the algorithm class `Polygon_offset_builder_2<Ss,Gt,Polygon_2>`.

Types

<code>PolygonOffsetBuilderTraits_2:: Kernel</code>	A model of the <i>Kernel</i> concept.
<code>PolygonOffsetBuilderTraits_2:: FT</code>	A <i>SqrtFieldNumberType</i> provided by the kernel. This type is used to represent the coordinates of the input points and to specify the desired offset distance.
<code>PolygonOffsetBuilderTraits_2:: Point_2</code>	A 2D point type
<code>boost::tuple<FT,FT> Vertex;</code>	A pair of (x,y) coordinates representing a 2D Cartesian point.
<code>boost::tuple<Vertex,Vertex></code>	
<code>Edge;</code>	A pair of vertices representing an edge
<code>boost::tuple<Edge,Edge,Edge></code>	
<code>EdgeTriple;</code>	A triple of edges representing an event

`PolygonOffsetBuilderTraits_2:: Compare_offset_against_event_time_2`

A predicate object type.
 Must provide `Comparison_result operator()(FT d, EdgeTriple const& et) const`, which compares the Euclidean distance d with the event time for et . Such event time is the Euclidean distance at which the *offset lines* intersect in a single point. The source of such offset lines is given by the 3 *oriented* lines defined by the edge-triple et
Precondition: et must be an edge-triple corresponding to an event that actually exist (that is, there must exist an offset distance $t > 0$ at which the offset lines do intersect at a single point.

`PolygonOffsetBuilderTraits_2:: Construct_offset_point_2`

A construction object type.
 Must provide `boost::optional<Point_2> operator()(FT t, Edge const& x, Edge const& y) const`, which constructs the point of intersection of the lines obtained by offsetting the oriented lines given by x and y an Euclidean distance t . If the point cannot be computed, not even approximately (because of overflow for instance), an empty optional must be returned.

Precondition: x and y must intersect in a single point

`PolygonOffsetBuilderTraits_2:: Construct_ss_vertex_2`

A construction object type.
 Must provide `Vertex operator()(Point_2 const& p)`, which given a *Point_2* p returns a *Vertex* encapsulating the corresponding (x,y) pair of Cartesian coordinates.

PolygonOffsetBuilderTraits_2:: Construct_ss_edge_2

A construction object type.

Must provide *Edge operator()*(*Point_2 const& s*, *Point_2 const& t*), which given source and target points *s* and *t* returns an *Edge* encapsulating the corresponding input segment (in Cartesian coordinates.)

PolygonOffsetBuilderTraits_2:: Construct_ss_triedge_2

A construction object type.

Must provide *Triedge operator()*(*Edge const& e0*, *Edge const& e1*, *Edge const& e2*), which given the 3 edges that define an event, *e0*, *e1* and *e2*, returns a *Triedge* encapsulating them.

Has Models

CGAL::Polygon_offset_builder_traits_2<K>.

See Also

CGAL::Polygon_offset_builder_2<Ss,Gt,Polygon_2>

CGAL::Polygon_offset_builder_traits_2<K>

VertexContainer_2

Introduction

A model for the `VertexContainer_2` concept defines the requirements for a resizable container of 2D points. It is used to output the offset polygons generated by the `Polygon_offset_builder_2<Ssds,Gt,Container>` class.

Types

<code>VertexContainer_2:: Point_2</code>	A 2D point type used to represent a vertex. Must be a model of the <code>Kernel::Point_2</code> concept
<code>VertexContainer_2:: size_type</code>	A unsigned integral type that can represent the number of vertices in the container.

Creation

<code>VertexContainer_2 c;</code>	Default constructor
<code>size_type</code> <code>c.size()</code>	Returns the number of vertices in the container.
<code>void</code> <code>c.push_back(Point_2 v)</code>	Adds the vertex <code>v</code> to the container, resizing its capacity if required.

Has Models

`CGAL::Polygon_2`

Any standard `BackInsertionSequence`, such as `vector`, `list` or `deque`, with a `value_type` being a model of the `Kernel::Point_2` concept.

CGAL::Straight_skeleton_2<Traits,Items,Alloc>

Definition

The class *Straight_skeleton_2*<Traits,Items,Alloc> provides a model for the *StraightSkeleton_2* concept which is the class type used to represent a straight skeleton.

It inherits from *HalfedgeDS_vector*<Traits,Items,Alloc>

```
#include <CGAL/Straight_skeleton_2.h>
```

Is Model for the Concepts

StraightSkeleton_2
DefaultConstructible
CopyConstructible
Assignable

See Also

StraightSkeletonVertex_2
StraightSkeletonHalfedge_2
StraightSkeleton_2

The only purpose of this class is to protect all the modifying operations in a *HalfedgeDS*. Normal users should not modify a straight skeleton. If an advanced user needs to get access to the modifying operations, it must call the required methods through the *::Base* class.

CGAL::Straight_skeleton_vertex_base_2<Refs,Point,FT>

Definition

The class *Straight_skeleton_vertex_base_2<Refs,Point,FT>* provides a model for the *StraightSkeletonVertex_2* concept which is the vertex type required by the *StraightSkeleton_2* concept. The class *Straight_skeleton_vertex_base_2<Refs,Point,FT>* has three template arguments: the first is the model of the *StraightSkeleton_2* concept (the vertex container), the second is a Point type, and the third is a model of the *SqrtFieldNumberType*, which is the numeric type used to represent the time of a vertex (a Euclidean distance).

This class can be used as a base class allowing users of the straight skeleton data structure to decorate a vertex with additional data. The concrete vertex class must be given in the *HalfedgeDSItems* template parameter of the instantiation of the *HalfedgeDS_default* class used as the model for the *Straight_skeleton_2* concept.

```
#include <CGAL/Straight_skeleton_vertex_base_2.h>
```

Is Model for the Concepts

StraightSkeletonVertex_2
DefaultConstructible
CopyConstructible
Assignable

See Also

StraightSkeletonVertex_2
StraightSkeletonHalfedge_2
StraightSkeleton_2
CGAL::Straight_skeleton_halfedge_base_2<Refs>

CGAL::Straight_skeleton_halfedge_base_2<Refs>

Definition

The class *Straight_skeleton_halfedge_base_2<Refs>* provides a model for the *StraightSkeletonHalfedge_2* concept which is the halfedge type required by the *StraightSkeleton_2* concept. The class *Straight_skeleton_halfedge_base_2<Refs>* has only one template arguments: a model of the *StraightSkeleton_2* concept (the halfedge container).

This class can be used as a base class allowing users of the straight skeleton data structure to decorate a halfedge with additional data. The concrete halfedge class must be given in the *HalfedgeDSItems* template parameter of the instantiation of the *HalfedgeDS_default* class used as the model for the *StraightSkeleton_2* concept.

```
#include <CGAL/Straight_skeleton_halfedge_base_2.h>
```

Is Model for the Concepts

StraightSkeletonHalfedge_2
DefaultConstructible
CopyConstructible
Assignable

See Also

StraightSkeletonHalfedge_2
StraightSkeletonVertex_2
StraightSkeleton_2
CGAL::Straight_skeleton_vertex_base_2<Refs,Point,FT>

CGAL::Straight_skeleton_builder_traits_2<Kernel>

Definition

The class *Straight_skeleton_builder_traits_2<Kernel>* provides a model for the *StraightSkeletonBuilderTraits_2* concept which is the traits class required by the *Straight_skeleton_builder_2* algorithm class. The class *Straight_skeleton_builder_traits_2<Kernel>* has one template argument: a 2D CGAL Kernel. This parameter must be a model for the *Kernel* concept, such as the *Exact_predicates_inexact_constructions_kernel*, which is the recommended one.

It is unspecified which subset of the kernel is used into the output sequence and the returned iterator will be equal to *out*.

For any given input polygon, it in this traits class (and by extension in the builder class). This is to avoid restricting the choices in the implementation.

```
#include <CGAL/Straight_skeleton_builder_traits_2.h>
```

Is Model for the Concepts

StraightSkeletonBuilderTraits_2

DefaultConstructible

CopyConstructible

See Also

CGAL::Straight_skeleton_builder_2<Gt,Sds>

CGAL::Straight_skeleton_builder_2<Gt,Ss>

Definition

The class *Straight_skeleton_builder_2*<Gt,Ss> encapsulates the construction of the 2D straight skeleton in the interior of a polygon with holes. Its first template parameter, *Gt*, must be a model of the *StraightSkeletonBuilderTraits_2* concept, and its second template parameter, *Ss*, must be a model of the *StraightSkeleton_2* concept.

```
#include <CGAL/Straight_skeleton_builder_2.h>
```

Types

<i>Straight_skeleton_builder_2</i> <Gt,Ss>:: <i>Gt</i>	The geometric traits (first template parameter)
<i>Straight_skeleton_builder_2</i> <Gt,Ss>:: <i>Ss</i>	The straight skeleton (second template parameter)
<i>Straight_skeleton_builder_2</i> <Gt,Ss>:: <i>Point_2</i>	The 2D point type as defined by the geometric traits

Creation

<i>Straight_skeleton_builder_2</i> <Gt,Ss> <i>b</i> ;	Default constructs the builder class.
-------------------------------------------------------	---------------------------------------

Methods

```
template<class InputPointIterator>
Straight_skeleton_builder_2&
```

```
    b.enter_contour( InputPointIterator aBegin, InputPointIterator aEnd)
```

Defines the *contours* that form the *non-degenerate strictly-simple polygon with holes* whose *straight skeleton* is to be built.

Each contour must be input in turn starting with the *outer contour* and following with the holes (if any). The order of the holes is unimportant but the outer contour must be entered first. The outer contour must be oriented counter-clockwise while holes must be oriented clockwise.

It is an error to enter more than one outer contour or to enter a hole which is not inside the outer contour or inside another hole. It is also an error to enter a contour which crosses or touches any one another. It is possible however to enter a contour that touches itself in such a way that its interior region is still well defined and singly-connected (see the User Manual for examples).

The sequence [aBegin,aEnd) must iterate over each 2D point that corresponds to a vertex of the contour being entered. Vertices cannot be coincident (except consecutively since the method simply skip consecutive coincident vertices). Consecutive collinear edges are allowed.

InputPointIterator must be an *InputIterator* whose *value_type* is *Point_2*.

`boost::shared_ptr<Ss>`

`b.construct_skeleton()`

Constructs and returns the 2D straight skeleton in the interior of the polygon with holes as defined by the contours entered first by calling `enter_contour`. All the contours of the polygon with holes must be entered before calling `construct_skeleton`.

After `construct_skeleton` completes, you cannot enter more contours and/or call `construct_skeleton()` again. If you need another straight skeleton for another polygon you must instantiate and use another builder.

The result is a dynamically allocated instance of the `Ss` class, wrapped in a `boost::shared_ptr`.

If the construction process fails for whatever reason (such as a nearly-degenerate vertex whose internal or external angle is almost zero), the return value will be `null`, represented by a default constructed `shared_ptr`.

The algorithm automatically checks the consistency of the result, thus, if it is not `null`, it is guaranteed to be valid.

Algorithm

The implemented algorithm is closely based on [FO98] with the addition of *vertex events* as described in [EE98].

It simulates a grassfire propagation of moving polygon edges as they move inward at constant and equal speed. That is, the continuous inward offsetting of the polygon.

Since edges move at equal speed their movement can be characterized in a simpler setup as the movement of vertices. Vertices move along the angular bisector of adjacent edges.

The trace of a moving vertex is described by the algorithm as a *bisector*. Every position along a bisector corresponds to the vertex between two offset (moved) edges. Since edges move at constant speed, every position along a bisector also corresponds to the distance those two edges moved so far.

From the perspective of a dynamic system of moving edges, such a distance can be regarded as an *instant* (in time). Therefore, every distinct position along a bisector corresponds to a distinct instant in the offsetting process.

As they move inward, edges can expand or contract w.r.t to the endpoints sharing a vertex. If a vertex has an internal angle $< \pi$, its incident edges will contract but if its internal angle $> \pi$, they will expand. The movement of the edges, along with their extent change, result in collisions between non-adjacent edges. These collisions are called *events*, and they occur when the colliding edges have moved a certain distance, that is, at certain *instants*.

If non-consecutive edges $E(j), E(k)$ move while edge $E(i)$ contracts, they can collide at the point when $E(i)$ shrinks to nothing (that is, the three edges might meet at a certain offset). This introduces a *topological change* in the polygon: Edges $E(j), E(k)$ are now adjacent, edge $E(i)$ disappears, and a new vertex appears. This topological change is called an *edge event*.

Similarly, consecutive expanding edges $E(i), E(i+1)$ sharing a reflex vertex (internal angle $\geq \pi$) might collide with any edge $E(j)$ on the rest of the same connected component of the polygon boundary (even far away from the initial edge's position). This also introduces a topological change: $E(j)$ gets split in two edges and the connected component having $E(i), E(i+1)$ and $E(j)$ is split in two unconnected parts: one having $E(i)$ and the corresponding subsegment of $E(j)$ and the other with $E(i+1)$ and the rest of $E(j)$. This is called a *split event*.

If a reflex vertex hits not an edge $E(j)$ but another reflex vertex $E(j), E(j+1)$, and viceversa (the reflex vertex $V(j)$ hits $V(i)$), there is no actual split and the two unconnected parts have $E(i), E(j)$ and $E(i+1), E(j+1)$ (or $E(i), E(j+1)$ and $E(i+1), E(j)$). This topological change is called a *vertex event*. Although similar to a split event in the sense

that two new unconnected contours emerge introducing two new contour vertices, in the case of a vertex event one of the new contour vertices might be reflex; that is, a vertex event *may* result in one of the offset polygons having a *reflex* contour vertex which was not in the original polygon.

Edges movement is described by vertices movement, and these by bisectors. Therefore, the collision between edges $E(j), E(i), E(k)$ (all in the same connected component) occurs when the moving vertices $E(j) \rightarrow E(i)$ and $E(i) \rightarrow E(k)$ meet ; that is, when the two bisectors describing the moving vertices *intersect* (Note: as the edges move inward and events occur, a vertex between edges A and B might exist even if A and B are not consecutive; that is, j and k are not necessarily i-1 and i+1 respectively, although initially they are).

Similarly, the collision between $E(i), E(i+1)$ with $E(j)$ (all in the same connected component) occurs when the bisector corresponding to the moving vertex $E(i) \rightarrow E(i+1)$ hits the moving edge $E(j)$.

Since each event changes the topology of the moving polygon, it is not possible or practical to foresee all events at once. Rather, the algorithm starts by estimating an initial set of potential events and from there it computes one next event at a time based on the previous one. The chaining of events is governed by their relative instants: events that occur first are processed first.

An initial set of *potential* split events is first computed independently (the computation of a potential split event is based solely on a reflex vertex and all other edges in the same connected component); and an initial set of *potential* edge events between initially consecutive bisectors is first computed independently (based solely on each bisector pair under consideration).

Events occur at certain instants and the algorithm must be able to order them accordingly. The correctness of the algorithm is uniquely and directly governed by the correct computation and ordering of the events. Any potential event might no longer be applicable after the topological change introduced by a prior event.

A grassfire propagation picks the next unprocessed event (starting from the first) and if it is still applicable processes it. Processing an event involves connecting edges, adding a new skeleton vertex (which corresponds to a contour vertex of the offset polygon) and calculating one new potential future event (which can be either an edge event or a split event -because of a prior vertex event-), based on the topological change just introduced. The propagation finishes when there are no new future events.

See Also

StraightSkeletonBuilderTraits_2

StraightSkeletonVertex_2

StraightSkeletonHalfedge_2

StraightSkeleton_2

CGAL::Straight_skeleton_builder_traits_2<K>

CGAL::Straight_skeleton_vertex_base_2<Refs,P,FT>

CGAL::Straight_skeleton_halfedge_base_2<Refs>

CGAL::Straight_skeleton_2<Traits,Items,Alloc>

CGAL::Polygon_offset_builder_traits_2<Kernel>

Definition

The class *Polygon_offset_builder_traits_2<Kernel>* provides a model for the *PolygonOffsetBuilderTraits_2* concept which is the traits class required by the *Polygon_offset_builder_2* algorithm class. The class *Polygon_offset_builder_traits_2<Kernel>* has one template argument: a 2D CGAL::Kernel. This parameter must be a model for the *Kernel* concept, such as the *Exact_predicates_inexact_constructions_kernel*, which is the recommended one. It is unspecified which subset of the kernel is used in this traits class (and by extension in the builder class). This is to avoid restricting the choices in the implementation.

```
#include <CGAL/Polygon_offset_builder_traits_2.h>
```

Is Model for the Concepts

PolygonOffsetBuilderTraits_2
DefaultConstructible
CopyConstructible

See Also

CGAL::Polygon_offset_builder_2<Ss,Gt,Polygon_2>

CGAL::Polygon_offset_builder_2<Ss,Gt,Container>

Definition

The class *Polygon_offset_builder_2*<*Ss,Gt,Container*> encapsulates the construction of inward offset contours of a 2D simple polygon with holes. The construction is based on the straight skeleton of the interior of the polygon. Its first template parameter, *Ss*, must be a model of the *StraightSkeleton_2* concept, its second template parameter, *Gt*, must be a model of the *StraightSkeletonBuilderTraits_2* concept, and its third template parameter must be a model of the *VertexContainer_2* concept.

```
#include <CGAL/Polygon_offset_builder_2.h>
```

Types

Polygon_offset_builder_2<*Ss,Gt,Container*>:: *Ss*

The straight skeleton (first template parameter)

Polygon_offset_builder_2<*Ss,Gt,Container*>:: *Gt*

The geometric traits (second template parameter)

Polygon_offset_builder_2<*Ss,Gt,Container*>:: *Container*

The container of 2D vertices that represents each offset contour generated by the algorithm (third template parameter)

Polygon_offset_builder_2<*Ss,Gt,Container*>:: *FT*

The *SqrtFieldNumberType* used to specify the desired offset distance, provided by the geometric traits *Gt*.

Creation

Polygon_offset_builder_2<*Ss,Gt,Container*> *b*(*Ss ss*);

Constructs the builder class using the given Straight Skeleton instance.

Methods

```
template<class OutputIterator>
```

OutputIterator *b.construct_offset_contours(FT t, OutputIterator out)*

Given the straight skeleton passed in the constructor which corresponds to the interior of a non-degenerate strictly-simple polygon with holes *P*, returns *all* the offset contours of *P* at the Euclidean distance *t*.

Such offset contours are simple polygons in the interior of *P*.

For any offset distance *t* there are 0, 1 or more offset contours.

For each resulting offset contour, a default constructed instance of *Container* type (which must be a model of the *VertexContainer_2* concept), is dynamically allocated and each offset vertex is added to it.

A *boost::shared_ptr* holding onto the dynamically allocated container is inserted into the output sequence via the *OutputIterator out*.

OutputIterator must be a model of the *OutputIterator* category whose *value_type* is a *boost::shared_ptr* holding the dynamically allocated instances of type *Container*.

The method returns an *OutputIterator* past-the-end of the resulting sequence, which contains each offset contour generated.

You can call *construct_offset_contours* with different offset distances (there is no need to construct the builder again). If you call it with an offset distance so large that there are no offset contours at that distance, no contour is inserted into the output sequence and the returned iterator will be equal to *out*.

For any given input polygon, its offset polygons at a certain distance are composed of several contours. This method returns all such contours in an unspecified order and with no parental relationship between them (that is why it is called *construct_offset_contours* and not *construct_offset_polygons*).

Those offset contours in the resulting sequence which are oriented counter-clockwise are outer contours and those oriented clockwise are holes. It is up to the user to match each hole to its parent in order to reconstruct the parent-hole relationship of the conceptual output. It is sufficient to test each hole against each parent as there won't be a hole inside a hole, a parent inside any other contour, or a hole inside more than one parent. The recommended insideness test is to perform a regularized Boolean set operation. *do_intersect* from the *Boolean Set Operations* package will work fine in our case since it is guaranteed that no hole would cross halfway outside any parent (in the presence of such cases, subtracting the parent from the hole works better as it correctly rejects halfway holes).

If there are degenerate, or nearly degenerate vertices; that is, vertices whose internal or external angle approaches 0, numerical overflow may prevent some of the polygon contours to be constructed. If that happens, the failed contour just won't be added into the resulting sequence.

See Also

VertexContainer_2

PolygonOffsetBuilderTraits_2

CGAL::Polygon_offset_builder_traits_2<Kernel>

CGAL::compute_outer_frame_margin

Definition

The function `compute_outer_frame_margin` computes the separation required between a polygon and the outer frame used to obtain an exterior skeleton suitable for the computation of outer offset polygons at a given distance.

```
#include <CGAL/compute_outer_frame_margin.h>
```

```
template <class InputIterator, class Traits>
boost::optional< typename Traits::FT >
```

```
compute_outer_frame_margin( InputIterator first,
                             InputIterator beyond,
                             typename Traits::FT offset,
                             Traits traits = Default_traits)
```

Given a non-degenerate strictly-simple 2D polygon whose vertices are passed in the range `[first,beyond)`, calculates the largest euclidean distance d between each input vertex and its corresponding offset vertex at a distance `offset`.

If such a distance can be approximately computed, returns an `optional<FT>` with the value $d + (\text{offset} * 1.05)$. If the distance cannot be computed, not even approximately, due to overflow for instance, returns an empty `optional<FT>` (an *absent result*).

This result is the required separation between the input polygon and the rectangular frame used to construct an exterior offset contour at distance `offset` (which is done by placing the polygon as a hole of that frame).

Such a separation must be computed in this way because if the frame is too close to the polygon, the inward offset contour from the frame could collide with the outward offset contour of the polygon, resulting in a merged contour offset instead of two contour offsets, one of them corresponding to the frame.

Simply using $2 * \text{offset}$ as the separation is incorrect since `offset` is the distance between an offset line and its original, not between an offset vertex and its original. The later, which is calculated by this function and needed to place the frame sufficiently away from the polygon, can be thousands of times larger than `offset`.

If the result is *absent*, any attempt to construct an exterior offset polygon at distance `offset` will fail. This will occur whenever the polygon has a vertex with an internal angle approaching 0 (because the offset vertex of a vertex whose internal angle equals 0 is at *infinity*).

Precondition: `offset > 0`.

Precondition: The range `[first,beyond)` contains the vertices of a non-degenerate strictly-simple 2D polygon.

The default traits class *Default_traits* is an instance of the class *Polygon_offset_builder_traits_2<Kernel>* parameterized on the kernel in which the type *InputIterator::value_type* is defined.

Requirements

1. *InputIterator::value_type* is equivalent to *Traits::Point_2*.
2. *Traits* must be a model for *PolygonOffsetBuilderTraits_2*

See Also

PolygonOffsetBuilderTraits_2
CGAL::Polygon_offset_builder_traits_2<Kernel>

Part VI

Arrangements

Chapter 17

2D Arrangements

Ron Wein, Efi Fogel, Baruch Zukerman, and Dan Halperin

Contents

17.1 Introduction	1104
17.2 The Main Arrangement Class	1105
17.2.1 A Simple Program	1106
17.2.2 Traversing the Arrangement	1107
17.2.3 Modifying the Arrangement	1110
17.3 Issuing Queries on an Arrangement	1121
17.3.1 Point-Location Queries	1121
17.3.2 Vertical Ray Shooting	1125
17.3.3 Batched Point-Location	1127
17.4 Free Functions in the Arrangement Package	1129
17.4.1 Incremental Insertion Functions	1129
17.4.2 Aggregated Insertion Functions	1132
17.4.3 Removing Vertices and Edges	1136
17.5 Traits Classes	1137
17.5.1 Traits Classes for Line Segments	1139
17.5.2 The Polyline-Traits Class	1143
17.5.3 A Traits Class for Circular Arcs and Line Segments	1145
17.5.4 A Traits Class for Conic Arcs	1150
17.5.5 A Traits Class for Arcs of Rational Functions	1155
17.5.6 Traits-Class Decorators	1158
17.6 The Notification Mechanism	1163
17.7 Extending the DCEL	1167
17.7.1 Extending the DCEL Faces	1167
17.7.2 Extending All DCEL Records	1170
17.8 Overlaying Arrangements	1172
17.8.1 Example for a Simple Overlay	1173
17.8.2 Example for a Face Overlay	1175
17.9 Storing the Curve History	1177
17.9.1 Traversing an Arrangement with History	1177
17.9.2 Modifying an Arrangement with History	1178
17.9.3 Examples	1178
17.10 Input/Output Functions	1183

17.10.1 Arrangements with Auxiliary Data	1184
17.10.2 Arrangements with Curve History	1187
17.11 Adapting to BOOST Graphs	1189
17.11.1 The Primal Arrangement Representation	1189
17.11.2 The Dual Arrangement Representation	1192
17.12 How To Speed Up Your Computation	1194

17.1 Introduction

Given a set C of planar curves, the *arrangement* $\mathcal{A}(C)$ is the subdivision of the plane into zero-dimensional, one-dimensional and two-dimensional cells, called *vertices*, *edges* and *faces*, respectively induced by the curves in C . Arrangements are ubiquitous in the computational-geometry literature and have many applications; see, e.g., [AS00, Hal04].

The curves in C can intersect each other (a single curve may also be self-intersecting or may be comprised of several disconnected branches) and are not necessarily x -monotone.¹ We construct a collection C'' of x -monotone subcurves that are pairwise disjoint in their interiors in two steps as follows. First, we decompose each curve in C into maximal x -monotone subcurves (and possibly isolated points), obtaining the collection C' . Note that an x -monotone curve cannot be self-intersecting. Then, we decompose each curve in C' into maximal connected subcurves not intersecting any other curve (or point) in C' . The collection C'' may also contain isolated points, if the curves of C contain such points. The arrangement induced by the collection C'' can be conveniently embedded as a planar graph, whose vertices are associated with curve endpoints or with isolated points, and whose edges are associated with subcurves. It is easy to see that $\mathcal{A}(C) = \mathcal{A}(C'')$. This graph can be represented using a *doubly-connected edge list* data-structure (DCEL for short), which consists of containers of vertices, edges and faces and maintains the incidence relations among these objects.

The main idea behind the DCEL data-structure is to represent each edge using a pair of directed *halfedges*, one going from the xy -lexicographically smaller (left) endpoint of the curve toward its the xy -lexicographically larger (right) endpoint, and the other, known as its *twin* halfedge, going in the opposite direction. As each halfedge is directed, we say it has a *source* vertex and a *target* vertex. Halfedges are used to separate faces, and to connect vertices (with the exception of *isolated vertices*, which are unconnected).

If a vertex v is the target of a halfedge e , we say that v and e are *incident* to each other. The halfedges incident to a vertex v form a circular list oriented in a clockwise order around this vertex. (An isolated vertex has no incident halfedges.)

Each halfedge e stores a pointer to its *incident face*, which is the face lying to its left. Moreover, every halfedge is followed by another halfedge sharing the same incident face, such that the target vertex of the halfedge is the same as the source vertex of the next halfedge. The halfedges are therefore connected in circular lists, and form chains, such that all edges of a chain are incident to the same face and wind along its boundary. We call such a chain a *connected component of the boundary* (or *CCB* for short).

The unique CCB of halfedges winding in a counterclockwise orientation along a face boundary is referred to as the *outer CCB* of the face. Exactly one unbounded face exists in every arrangement, as the arrangement package supports only bounded curves at this point. The unbounded face does not have an outer boundary. Any other connected component of the boundary of the face is called a *hole* (or *inner CCB*), and can be represented as a circular chain of halfedges winding in a clockwise orientation around it. Note that a hole does not necessarily

¹A continuous planar curve C is *x-monotone* if every vertical line intersects it at most once. For example, a non-vertical line segment is always x -monotone and so is the graph of any continuous function $y = f(x)$. For convenience, we treat vertical line segments as *weakly x-monotone*, as there exists a single vertical line that overlaps them. A circle of radius r centered at (x_0, y_0) is not x -monotone, as the vertical line $x = x_0$ intersects it at $(x_0, y_0 - r)$ and at $(x_0, y_0 + r)$.

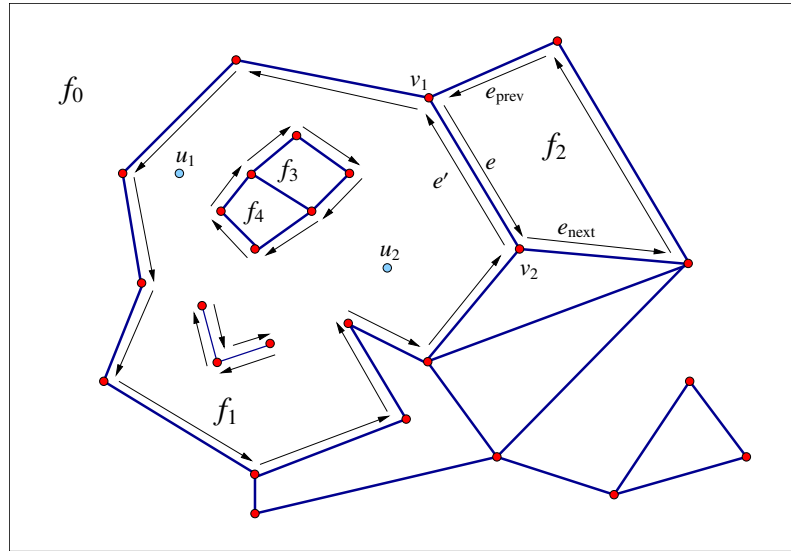


Figure 17.1: An arrangement of interior-disjoint line segments with some of the DCEL records that represent it. The unbounded face f_0 has a single connected component that forms a hole inside it, and this hole is comprised of several faces. The half-edge e is directed from its source vertex v_1 to its target vertex v_2 . This edge, together with its twin e' , correspond to a line segment that connects the points associated with v_1 and v_2 and separates the face f_1 from f_2 . The predecessor e_{prev} and successor e_{next} of e are part of the chain that form the outer boundary of the face f_2 . The face f_1 has a more complicated structure as it contains two holes in its interior: One hole consists of two adjacent faces f_3 and f_4 , while the other hole is comprised of two edges. f_1 also contains two isolated vertices u_1 and u_2 in its interior.

correspond to a single face, as it may have no area, or alternatively it may consist of several connected faces. Every face can have several holes contained in its interior (or no holes at all). In addition, every face may contain isolated vertices in its interior. See Figure 17.1 for an illustration of the various DCEL features. For more details on the DCEL data structure see [dBvKOS00, Chapter 2].

The rest of this chapter is organized as follows: In Section 17.2 we review in detail the interface of the *Arrangement_2* class-template, which is the central component in the arrangement package. In Section 17.3 we show how queries on an arrangement can be issued. In Section 17.4 we review some important free (global) functions that operate on arrangements, the most important ones being the free insertion-functions. Section 17.5 contains detailed descriptions of the various geometric traits classes included in the arrangement package. Using these traits classes it is possible to construct arrangements of different families of curves. In Section 17.6 we review the notification mechanism that allows external classes to keep track of the changes that an arrangement instance goes through. Section 17.7 explains how to extend the DCEL records, to store extra data with them, and to efficiently update this data. In Section 17.8 we introduce the fundamental operation of overlaying two arrangements. Section 17.9 describes the *Arrangement_with_history_2* class-template that extends the arrangement by storing additional history records with its curves. Finally, in Section 17.10 we review the arrangement input/output functions.

17.2 The Main Arrangement Class

The class *Arrangement_2*<Traits,Dcel> is the main class in the arrangement package. It is used to represent planar arrangements and provides the interface needed to construct them, traverse them, and maintain them. An arrangement is defined by a geometric *traits* class that determines the family of planar curves that form the

arrangement, and a DCEL class, which represents the *topological structure* of the planar subdivision. It supplies a minimal set of geometric operations (predicates and constructions) required to construct and maintain the arrangement and to operate on it.

The design of the arrangement package is guided by the need to separate between the representation of the arrangements and the various geometric algorithms that operate on them, and by the need to separate between the topological and geometric aspects of the planar subdivision. The separation is exhibited by the two template parameters of the *Arrangement_2* template:

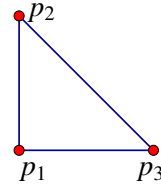
- The *Traits* template-parameter should be instantiated with a model of the *ArrangementBasicTraits_2* concept. The traits class defines the types of *x-monotone curves* and two-dimensional points, *X_monotone_curve_2* and *Point_2* respectively, and supports basic geometric predicates on them.

In the first sections of this chapter we always use *Arr_segment_traits_2* as our traits class, to construct arrangements of line segments. However, the arrangement package contains several other traits classes that can handle also polylines (continuous piecewise-linear curves), conic arcs, and arcs of rational functions. We exemplify the usage of these traits classes in Section 17.5.

- The *Dcel* template-parameter should be instantiated with a class that is a model of the *ArrangementDcel* concept. The value of this parameter is *Arr_default_dcel<Traits>* by default. However, in many applications it is necessary to extend the DCEL features; see Section 17.7 for further explanations and examples.

17.2.1 A Simple Program

The simple program listed below constructs a planar map of three line segments forming a triangle. The constructed arrangement is instantiated with the *Arr_segment_traits_2* traits class to handle segments only. The resulting arrangement consists of two faces, a bounded triangular face and the unbounded face. The program is not very useful as it is, since it ends immediately after the arrangement is constructed. We give more enhanced examples in the rest of this chapter.



```
#include <CGAL/Cartesian.h>
#include <CGAL/MP_Float.h>
#include <CGAL/Quotient.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>

typedef CGAL::Quotient<CGAL::MP_Float>      Number_type;
typedef CGAL::Cartesian<Number_type>        Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel> Traits_2;
typedef Traits_2::Point_2                   Point_2;
typedef Traits_2::X_monotone_curve_2        Segment_2;
typedef CGAL::Arrangement_2<Traits_2>       Arrangement_2;

int main()
{
    Arrangement_2    arr;
    Segment_2        cv[3];
    Point_2           p1 (0, 0), p2 (0, 4), p3 (4, 0);

    cv[0] = Segment_2 (p1, p2);
    cv[1] = Segment_2 (p2, p3);
    cv[2] = Segment_2 (p3, p1);
}
```

```

    insert (arr, &cv[0], &cv[3]);

    return (0);
}

```

17.2.2 Traversing the Arrangement

The simplest and most fundamental arrangement operations are the various traversal methods, which allow users to systematically go over the relevant features of the arrangement at hand.

As mentioned above, the arrangement is represented as a DCEL, which stores three containers of vertices, halfedges and faces. Thus, the *Arrangement_2* class supplies iterators for these containers. For example, the methods *vertices_begin()* and *vertices_end()* return *Arrangement_2::Vertex_iterator* objects that define the valid range of arrangement vertices. The value type of this iterator is *Arrangement_2::Vertex*. Moreover, the vertex-iterator type is equivalent to *Arrangement_2::Vertex_handle*, which serves as a pointer to a vertex. As we show next, all functions related to arrangement features accept handle types as input parameters and return handle types as their output.

In addition to the iterators for arrangement vertices, halfedges and faces, the arrangement class also provides *edges_begin()* and *edges_end()* that return *Arrangement_2::Edge_iterator* objects for traversing the arrangement edges. Note that the value type of this iterator is *Arrangement_2::Halfedge*, representing one of the twin halfedges that represent the edge.

All iterator, circulator² and handle types also have non-mutable (*const*) counterparts. These non-mutable iterators are useful to traverse an arrangement without changing it. For example, the arrangement has a non-constant member function called *vertices_begin()* that returns a *Vertex_iterator* object and another *const* member function that returns a *Vertex_const_iterator* object. In fact, all methods listed in this section that return an iterator, a circulator or a handle have non-mutable counterparts. It should be noted that, for example, *Vertex_handle* can be readily converted into a *Vertex_const_handle*, but not vice-verse.

Conversion of a non-mutable handle to a corresponding mutable handle are nevertheless possible, and can be performed using the static function *Arrangement_2::non_const_handle()* (see, e.g., Section 17.3.1). There are three variant of this function, one for each type of handle.

Traversal Methods for an Arrangement Vertex

A vertex is always associated with a geometric entity, namely with a *Point_2* object, which can be obtained by the *point()* method of the *Vertex* class nested within *Arrangement_2*.

The *is_isolated()* method determines whether a vertex is isolated or not. Recall that the halfedges incident to a non-isolated vertex, namely the halfedges that share a common target vertex, form a circular list around this vertex. The *incident_halfedges()* method returns a circulator of type *Arrangement_2::Halfedge_around_vertex_circulator* that enables the traversal of this circular list in a clockwise direction. The value type of this circulator is *Halfedge*.

The following function prints all the neighbors of a given arrangement vertex (assuming that the *Point_2* type can be inserted into the standard output using the << operator). The arrangement type is the same as in the simple example above.

```

void print_neighboring_vertices (Arrangement_2::Vertex_const_handle v)

```

²A *circulator* is used to traverse a circular list, such as the list of halfedges incident to a vertex — see below.

```

{
    if (v->is_isolated()) {
        std::cout << "The vertex (" << v->point() << ") is isolated" << std::endl;
        return;
    }

    Arrangement_2::Halfedge_around_vertex_const_circulator first, curr;
    first = curr = v->incident_halfedges();

    std::cout << "The neighbors of the vertex (" << v->point() << ") are:";
    do {
        // Note that the current halfedge is directed from u to v:
        Arrangement_2::Vertex_const_handle u = curr->source();
        std::cout << " (" << u->point() << ")";
    } while (++curr != first);
    std::cout << std::endl;
}

```

In case of an isolated vertex, it is possible to obtain the face that contains this vertex using the *face()* method.

Traversal Methods for an Arrangement Halfedge

Each arrangement edge, realized as a pair of twin halfedges, is associated with an *X_monotone_curve_2* object, which can be obtained by the *curve()* method of the *Halfedge* class nested in the *Arrangement_2* class.

The *source()* and *target()* methods return handles to the halfedge source and target vertices respectively. We can obtain a handle to the twin halfedge using the *twin()* method. From the definition of halfedges, it follows that if *he* is a halfedge handle, then:

- *he->curve()* is equivalent to *he->twin()->curve()*,
- *he->source()* is equivalent to *he->twin()->target()*, and
- *he->target()* is equivalent to *he->twin()->source()*.

Every halfedge has an incident face that lies to its left, which can be obtained by the *face()* method. Recall that a halfedge is always one link in a connected chain of halfedges that share the same incident face, known as a *CCB*. The *prev()* and *next()* methods return handles to the previous and next halfedges in the CCB respectively.

As the CCB is a circular list of halfedges, it is only natural to traverse it using a circulator. The *ccb()* method returns a *Arrangement_2::Ccb_halfedge_circulator* object for the halfedges along the CCB.

The function *print_ccb()* listed below prints all *x-monotone* curves along a given CCB (assuming that the *Point_2* and the *X_monotone_curve_2* types can be inserted into the standard output using the *<<* operator).

```

void print_ccb (Arrangement_2::Ccb_halfedge_const_circulator circ)
{
    Ccb_halfedge_const_circulator curr = circ;
    std::cout << "(" << curr->source()->point() << ")";
    do {
        Arrangement_2::Halfedge_const_handle he = curr->handle();
        std::cout << " [" << he->curve() << "]" "

```

```

        << "(" << he->target()->point() << ")";
    } while (++curr != circ);
    std::cout << std::endl;
}

```

Traversal Methods for an Arrangement Face

An arrangement of bounded curves always has a single unbounded face. The function *unbounded_face()* returns a handle to this face. (Note that an empty arrangement contains nothing *but* the unbounded face.)

Given a *Face* object, we can use the *is_unbounded()* method to determine whether it is unbounded. Bounded faces have an outer CCB, and the *outer_ccb()* method returns a circulator for the halfedges along this CCB. Note that the halfedges along this CCB wind in a counterclockwise orientation around the outer boundary of the face.

A face can also contain disconnected components in its interior, namely holes and isolated vertices:

- The *holes_begin()* and *holes_end()* methods return *Arrangement_2::Hole_iterator* iterators that define the range of holes inside the face. The value type of this iterator is *Ccb_halfedge_circulator*, defining the CCB that winds in a clockwise orientation around a hole.
- The *isolated_vertices_begin()* and *isolated_vertices_end()* methods return *Arrangement_2::Isolated_vertex_iterator* iterators that define the range of isolated vertices inside the face. The value type of this iterator is *Vertex*.

The function *print_face()* listed below prints the outer and inner boundaries of a given face, using the function *print_ccb()*, which was introduced in the previous subsection.

```

void print_face (Arrangement_2::Face_const_handle f)
{
    // Print the outer boundary.
    if (f->is_unbounded())
        std::cout << "Unbounded face. " << std::endl;
    else {
        std::cout << "Outer boundary: ";
        print_ccb (f->outer_ccb());
    }

    // Print the boundary of each of the holes.
    Arrangement_2::Hole_const_iterator hi;
    int index = 1;
    for (hi = f->holes_begin(); hi != f->holes_end(); ++hi, ++index) {
        std::cout << "    Hole #" << index << ": ";
        print_ccb (*hi);
    }

    // Print the isolated vertices.
    Arrangement_2::Isolated_vertex_const_iterator iv;
    for (iv = f->isolated_vertices_begin(), index = 1;
         iv != f->isolated_vertices_end(); ++iv, ++index)
    {
        std::cout << "    Isolated vertex #" << index << ": "
                  << "(" << iv->point() << ")" << std::endl;
    }
}

```

```
}
}
```

Additional Example

The function listed below prints the current setting of a given arrangement. This concludes the preview of the various traversal methods.³

```
void print_arrangement (const Arrangement_2& arr)
{
    // Print the arrangement vertices.
    Vertex_const_iterator vit;
    std::cout << arr.number_of_vertices() << " vertices:" << std::endl;
    for (vit = arr.vertices_begin(); vit != arr.vertices_end(); ++vit) {
        std::cout << "(" << vit->point() << ")";
        if (vit->is_isolated())
            std::cout << " - Isolated." << std::endl;
        else
            std::cout << " - degree " << vit->degree() << std::endl;
    }

    // Print the arrangement edges.
    Edge_const_iterator eit;
    std::cout << arr.number_of_edges() << " edges:" << std::endl;
    for (eit = arr.edges_begin(); eit != arr.edges_end(); ++eit)
        std::cout << "[" << eit->curve() << "]" << std::endl;

    // Print the arrangement faces.
    Face_const_iterator fit;
    std::cout << arr.number_of_faces() << " faces:" << std::endl;
    for (fit = arr.faces_begin(); fit != arr.faces_end(); ++fit)
        print_face (fit);
}
```

17.2.3 Modifying the Arrangement

In this section we review the various member functions of the *Arrangement_2* class that allow users to modify the topological structure of the arrangement by introducing new edges or vertices, modifying them, or removing them.

The arrangement member-functions that insert new curves into the arrangement, thus enabling the construction of a planar subdivision, are rather specialized, as they require a-priori knowledge on the location of the inserted curve. Indeed, for most purposes it is more convenient to construct an arrangement using the free (global) insertion-functions.

³The file *arr_print.h*, which can be found under the examples folder, includes this function and the rest of the functions listed in this section. Over there they are written in a more generic fashion, where the arrangement type serves as a template parameter for these functions, so different instantiations of the *Arrangement_2<Traits,Dcel>* template can be provided to the same function templates.

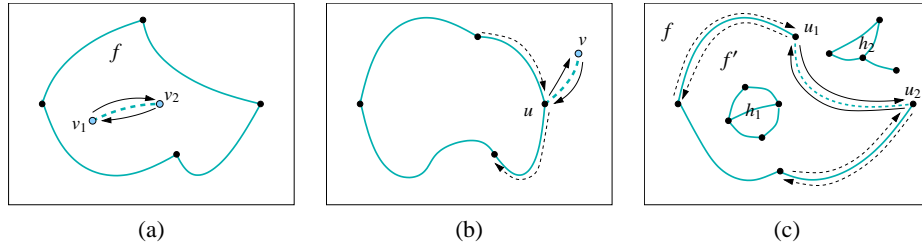


Figure 17.2: The various specialized insertion procedures. The inserted x -monotone curve is drawn with a light dashed line, surrounded by two solid arrows that represent the pair of twin half-edges added to the DCEL. Existing vertices are shown as black dots while new vertices are shown as light dots. Existing half-edges that are affected by the insertion operations are drawn as dashed arrows. (a) Inserting a curve as a new hole inside the face f . (b) Inserting a curve from an existing vertex u that corresponds to one of its endpoints. (c) Inserting an x -monotone curve whose endpoints are the already existing vertices u_1 and u_2 . In our case, the new pair of half-edges close a new face f' , where the hole h_1 , which used to belong to f , now becomes an enclave in this new face.

Inserting Non-Intersecting x -Monotone Curves

The most important functions that allow users to modify the arrangement, and perhaps the most frequently used ones, are the specialized insertion functions of x -monotone curves whose interior is disjoint from any other curve in the existing arrangement and do not contain any vertex of the arrangement. In addition, these function require that the location of the curve in the arrangement is known.

The motivation behind these rather harsh restrictions on the nature of the inserted curves is the decoupling of the topological arrangement representation from the various algorithms that operate on it. While the insertion of an x -monotone curve whose interior is disjoint from all existing arrangement features is quite straightforward (as we show next), inserting curves that intersect with the curves already inserted into the arrangement is much more complicated and requires the application of non-trivial geometric algorithms. These insertion operations are therefore implemented as free functions that operate on the arrangement and the inserted curve(s); see Section 17.4 for more details and examples.⁴

When an x -monotone curve is inserted into an existing arrangement, such that the interior of this curve is disjoint from any arrangement feature, only the following three scenarios are possible, depending on the status of the endpoints of the inserted subcurve:

1. In case both curve endpoints do not correspond to any existing arrangement vertex we have to create two new vertices corresponding to the curve endpoints and connect them using a pair of twin halfedges. This halfedge pair initiates a new hole inside the face that contains the curve in its interior.
2. If exactly one endpoint corresponds to an existing arrangement vertex (we distinguish between a vertex that corresponds to the left endpoint of the inserted curve and a vertex corresponding to its right endpoint), we have to create a new vertex that corresponds to the other endpoint of the curve and to connect the two vertices by a pair of twin halfedges that form an “antenna” emanating from the boundary of an existing connected component (note that if the existing vertex used to be isolated, this operation is actually equivalent to forming a new hole inside the face that contains this vertex).

⁴You may skip to Section 17.4, and return to this subsection at a later point in time.

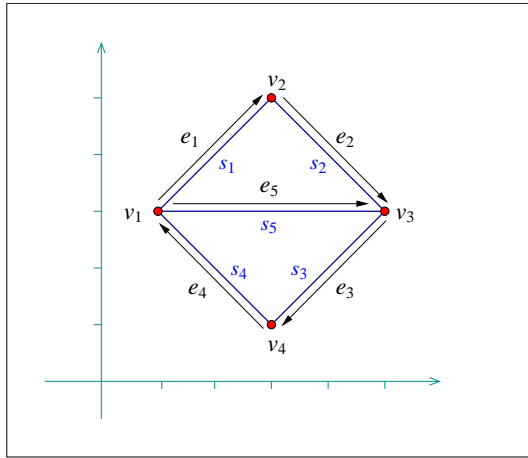
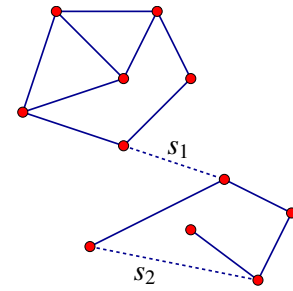


Figure 17.3: The arrangement of the line segments s_1, \dots, s_5 constructed in *ex_edge_insertion.C*. The arrows mark the direction of the halfedges returned from the various insertion functions.

3. If both endpoints correspond to existing arrangement vertices, we connect these vertices using a pair of twin halfedges. (If one or both vertices are isolated this case reduces to one of the two previous cases respectively.) The two following subcases may occur:

- Two disconnected components are merged into a single connected component (as is the case with the segment s_1 in the figure to the left).
- A new face is created, a face that splits from an existing arrangement face. In this case we also have to examine the holes and isolated vertices in the existing face and move the relevant ones inside the new face (as is the case with the segment s_2 in the figure to the left).



The *Arrangement_2* class offers insertion functions named *insert_in_face_interior()*, *insert_from_left_vertex()*, *insert_from_right_vertex()* and *insert_at_vertices()* that perform the special insertion procedures listed above. The first function accepts an x -monotone curve c and an arrangement face f that contains this curve in its interior. The other functions accept an x -monotone curve c and handles to the existing vertices that correspond to the curve endpoint(s). Each of the four functions returns a handle to one of the twin halfedges that have been created, where:

- *insert_in_face_interior(c, f)* returns a halfedge directed from the vertex corresponding to the left endpoint of c toward the vertex corresponding to its right endpoint.
- *insert_from_left_vertex(c, v)* and *insert_from_right_vertex(c, v)* returns a halfedge whose source is the vertex v that and whose target is the new vertex that has just been created.
- *insert_at_vertices(c, v1, v2)* returns a halfedge directed from v_1 to v_2 .

The following program demonstrates the usage of the four insertion functions. It creates an arrangement of five line segments, as depicted in Figure 17.3.⁵ As the arrangement is very simple, we use the simple Cartesian kernel of CGAL with integer coordinates for the segment endpoints. We also use the *Arr_segment_traits_2* class that enables the efficient maintenance of arrangements of line segments; see more details on this traits class

⁵Notice that in all figures in the rest of this chapter the coordinate axes are drawn only for illustrative purposes and are *not* part of the arrangement.

in Section 17.5. This example, as many others in this chapter, uses some print-utility functions from the file *print_arr.h*; these functions are also listed in Section 17.2.2.

```

//! \file examples/Arrangement_2/ex_edge_insertion.C
// Constructing an arrangement using the simple edge-insertion functions.

#include <CGAL/Simple_cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/IO/Arr_istream.h>

#include "arr_print.h"

typedef int                                     Number_type;
typedef CGAL::Simple_cartesian<Number_type>    Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>     Traits_2;
typedef Traits_2::Point_2                     Point_2;
typedef Traits_2::X_monotone_curve_2          Segment_2;
typedef CGAL::Arrangement_2<Traits_2>          Arrangement_2;
typedef Arrangement_2::Vertex_handle           Vertex_handle;
typedef Arrangement_2::Halfedge_handle         Halfedge_handle;

int main ()
{
    Arrangement_2    arr;

    Segment_2        s1 (Point_2 (1, 3), Point_2 (3, 5));
    Segment_2        s2 (Point_2 (3, 5), Point_2 (5, 3));
    Segment_2        s3 (Point_2 (5, 3), Point_2 (3, 1));
    Segment_2        s4 (Point_2 (3, 1), Point_2 (1, 3));
    Segment_2        s5 (Point_2 (1, 3), Point_2 (5, 3));

    Halfedge_handle e1 = arr.insert_in_face_interior (s1, arr.unbounded_face());
    Vertex_handle   v1 = e1->source();
    Vertex_handle   v2 = e1->target();
    Halfedge_handle e2 = arr.insert_from_left_vertex (s2, v2);
    Vertex_handle   v3 = e2->target();
    Halfedge_handle e3 = arr.insert_from_right_vertex (s3, v3);
    Vertex_handle   v4 = e3->target();
    Halfedge_handle e4 = arr.insert_at_vertices (s4, v4, v1);
    Halfedge_handle e5 = arr.insert_at_vertices (s5, v1, v3);

    print_arrangement (arr);
    return (0);
}

```

Observe that the first line segment is inserted in the interior of the unbounded face. The other line segments are inserted using the vertices created by the insertion of previous segments. The resulting arrangement consists of three faces, where the two bounded faces form together a hole in the unbounded face.

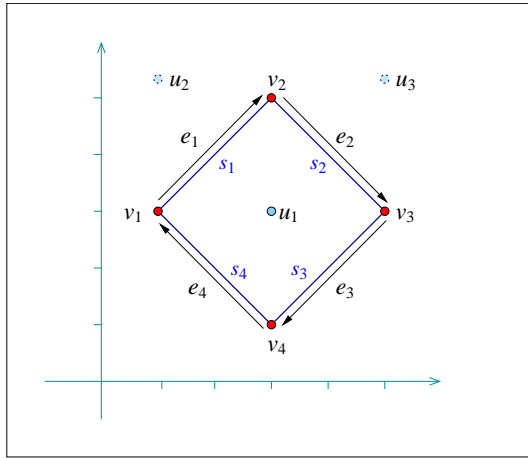


Figure 17.4: An arrangement of line segments containing three isolated vertices, as constructed in *ex_isolated_vertices.C*. The vertices u_2 and u_3 are eventually removed from the arrangement.

Manipulating Isolated Vertices

Isolated points are in general simpler geometric entities than curves and indeed the member functions that manipulate them are easier to understand.

The function *insert_in_face_interior*(p, f) inserts an isolated point p , located in the interior of a given face f , into the arrangement and returns a handle to the arrangement vertex it has created and associated with p . Naturally, this function has a precondition that p is really an isolated point, namely it does not coincide with any existing arrangement vertex and does not lie on any edge. As mentioned in Section 17.2.2, it is possible to obtain the face containing an isolated vertex handle v by calling $v \rightarrow \text{face}()$.

The function *remove_isolated_vertex*(v) receives a handle to an isolated vertex and removes it from the arrangement.

The following program demonstrates the usage of the arrangement member-functions for manipulating isolated vertices. It first inserts three isolated vertices located inside the unbounded face, then it inserts four line segments that form a rectangular hole inside the unbounded face (see Figure 17.4 for an illustration). Finally, it traverses the vertices and removes those isolated vertices that are still contained in the unbounded face (u_2 and u_3 in this case):

```

//! \file examples/Arrangement_2/ex_isolated_vertices.C
// Constructing an arrangement with isolated vertices.

#include <CGAL/Simple_cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>

#include "arr_print.h"

typedef int                                     Number_type;
typedef CGAL::Simple_cartesian<Number_type>    Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>      Traits_2;
typedef Traits_2::Point_2                      Point_2;
typedef Traits_2::X_monotone_curve_2           Segment_2;

```

```

typedef CGAL::Arrangement_2<Traits_2>           Arrangement_2;
typedef Arrangement_2::Vertex_handle           Vertex_handle;
typedef Arrangement_2::Halfedge_handle         Halfedge_handle;
typedef Arrangement_2::Face_handle             Face_handle;

int main ()
{
    Arrangement_2  arr;

    // Insert some isolated points:
    Face_handle     uf = arr.unbounded_face();
    Vertex_handle   u1 = arr.insert_in_face_interior (Point_2 (3, 3), uf);
    Vertex_handle   u2 = arr.insert_in_face_interior (Point_2 (1, 5), uf);
    Vertex_handle   u3 = arr.insert_in_face_interior (Point_2 (5, 5), uf);

    // Insert four segments that form a rectangular face:
    Segment_2       s1 (Point_2 (1, 3), Point_2 (3, 5));
    Segment_2       s2 (Point_2 (3, 5), Point_2 (5, 3));
    Segment_2       s3 (Point_2 (5, 3), Point_2 (3, 1));
    Segment_2       s4 (Point_2 (3, 1), Point_2 (1, 3));

    Halfedge_handle e1 = arr.insert_in_face_interior (s1, uf);
    Vertex_handle   v1 = e1->source();
    Vertex_handle   v2 = e1->target();
    Halfedge_handle e2 = arr.insert_from_left_vertex (s2, v2);
    Vertex_handle   v3 = e2->target();
    Halfedge_handle e3 = arr.insert_from_right_vertex (s3, v3);
    Vertex_handle   v4 = e3->target();
    Halfedge_handle e4 = arr.insert_at_vertices (s4, v4, v1);

    // Remove the isolated vertices located in the unbounded face.
    Arrangement_2::Vertex_iterator  curr_v, next_v;

    for (curr_v = arr.vertices_begin();
         curr_v != arr.vertices_end(); curr_v = next_v)
    {
        // Store an iterator to the next vertex (as we may delete curr_v and
        // invalidate the iterator).
        next_v = curr_v;
        ++next_v;

        if (curr_v->is_isolated() && curr_v->face() == uf)
            arr.remove_isolated_vertex (curr_v);
    }

    print_arrangement (arr);
    return (0);
}

```

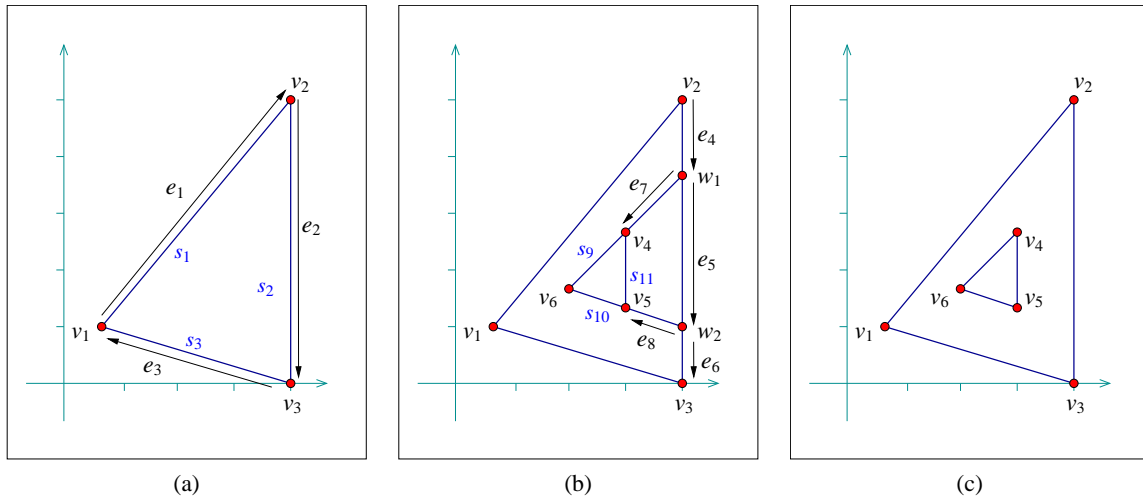


Figure 17.5: An arrangement of line segments as constructed in *ex_edge_manipulation.C*. Note that the edges e_7 and e_8 and the vertices w_1 and w_2 , introduced in step (b) are eventually removed in step (c).

Manipulating Halfedges

In the previous subsection we showed how to introduce new isolated vertices in the arrangement. But how does one create a vertex that lies on an existing arrangement edge (more precisely, on an x -monotone curve that is associated with an arrangement edge)?

It should be noted that such an operation involves the splitting of a curve at a given point in its interior, while the traits class used by *Arrangement_2* does not necessarily have the ability to perform such a split operation. However, if users have the ability to split an x -monotone curve into two at a given point p (this is usually the case when employing a more sophisticated traits class; see Section 17.5 for more details) they can use the *split_edge*(e , c_1 , c_2) function, where c_1 and c_2 are the two subcurves resulting from splitting the x -monotone curve associated with the halfedge e at some point (call it p) in its interior. The function splits the halfedge pair into two pairs, both incident to a new vertex v associated with p , and returns a handle to a halfedge whose source equals e 's source vertex and whose target is the new vertex v .

The reverse operation is also possible. Suppose that we have a vertex v of degree 2, whose two incident halfedges, e_1 and e_2 , are associated with the curves c_1 and c_2 . Suppose further that it is possible to merge these two curves into a single continuous x -monotone curve c . Calling *merge_edge*(e_1 , e_2 , c) will merge the two edges into a single edge associated with the curve c , essentially removing the vertex v from the arrangement.

Finally, the function *remove_edge*(e) removes the edge e from the arrangement. Note that this operation is the reverse of an insertion operation, so it may cause a connected component to split into two, or two faces to merge into one, or a hole to disappear. By default, if the removal of e causes one of its end-vertices to become isolated, we remove this vertex as well. However, users can control this behavior and choose to keep the isolated vertices by supplying additional Boolean flags to *remove_edge*() indicating whether the source and the target vertices are to be removed should they become isolated.

In the following example program we show how the edge-manipulation functions can be used. The program works in three steps, as demonstrated in Figure 17.5. Note that here we still stick to integer coordinates, but as we work on a larger scale we use an unbounded integer number-type (in this case, the *Gmpz* type taken from the GMP library) instead of the built-in *int* type.⁶ In case the GMP library is not installed (as indicated

⁶As a rule of thumb, one can use a bounded integer type for representing line segments whose coordinates are bounded by $\lfloor \sqrt[3]{M} \rfloor$, where M is the maximal representable integer value. This guarantees that no overflows occur in the computations carried out by the traits class, hence all traits-class predicates always return correct results.

by the *CGAL_USE_GMP* flag defined in *CGAL/basic.h*), we use *MP_Float*, a number-type included in CGAL's support library that is capable of storing floating-point numbers with unbounded mantissa. We also use the standard Cartesian kernel of CGAL as our kernel. This is recommended when the kernel is instantiated with a more complex number type, as we demonstrate in other examples in this chapter.

```

//! \file examples/Arrangement_2/ex_edge_manipulation.C
// Using the edge-manipulation functions.

#include <CGAL/basic.h>

#ifdef CGAL_USE_GMP
    #include <CGAL/Gmpz.h>
    typedef CGAL::Gmpz                                Number_type;
#else
    #include <CGAL/MP_Float.h>
    typedef CGAL::MP_Float                            Number_type;
#endif

#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>

#include "arr_print.h"

typedef CGAL::Cartesian<Number_type>                  Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>            Traits_2;
typedef Traits_2::Point_2                             Point_2;
typedef Traits_2::X_monotone_curve_2                  Segment_2;
typedef CGAL::Arrangement_2<Traits_2>                 Arrangement_2;
typedef Arrangement_2::Vertex_handle                  Vertex_handle;
typedef Arrangement_2::Halfedge_handle                 Halfedge_handle;

int main ()
{
    // Step (a) - construct a triangular face.
    Arrangement_2  arr;

    Segment_2      s1 (Point_2 (667, 1000), Point_2 (4000, 5000));
    Segment_2      s2 (Point_2 (4000, 0), Point_2 (4000, 5000));
    Segment_2      s3 (Point_2 (667, 1000), Point_2 (4000, 0));

    Halfedge_handle e1 = arr.insert_in_face_interior (s1, arr.unbounded_face());
    Vertex_handle   v1 = e1->source();
    Vertex_handle   v2 = e1->target();
    Halfedge_handle e2 = arr.insert_from_right_vertex (s2, v2);
    Vertex_handle   v3 = e2->target();
    Halfedge_handle e3 = arr.insert_at_vertices (s3, v3, v1);

    // Step (b) - create additional two faces inside the triangle.
    Point_2        p1 (4000, 3666), p2 (4000, 1000);
    Segment_2      s4 (Point_2 (4000, 5000), p1);
    Segment_2      s5 (p1, p2);
    Segment_2      s6 (Point_2 (4000, 0), p2);

```

```

Halfedge_handle e4 = arr.split_edge (e2, s4,
                                     Segment_2 (Point_2 (4000, 0), p1));
Vertex_handle   w1 = e4->target();
Halfedge_handle e5 = arr.split_edge (e4->next(), s5, s6);
Vertex_handle   w2 = e5->target();
Halfedge_handle e6 = e5->next();

Segment_2       s7 (p1, Point_2 (3000, 2666));
Segment_2       s8 (p2, Point_2 (3000, 1333));
Segment_2       s9 (Point_2 (3000, 2666), Point_2 (2000, 1666));
Segment_2       s10 (Point_2 (3000, 1333), Point_2 (2000, 1666));
Segment_2       s11 (Point_2 (3000, 1333), Point_2 (3000, 2666));

Halfedge_handle e7 = arr.insert_from_right_vertex (s7, w1);
Vertex_handle   v4 = e7->target();
Halfedge_handle e8 = arr.insert_from_right_vertex (s8, w2);
Vertex_handle   v5 = e8->target();
Vertex_handle   v6 = arr.insert_in_face_interior (Point_2 (2000, 1666),
                                                  e8->face());

arr.insert_at_vertices (s9, v4, v6);
arr.insert_at_vertices (s10, v5, v6);
arr.insert_at_vertices (s11, v4, v5);

// Step (c) - remove and merge faces to form a single hole in the triangle.
arr.remove_edge (e7);
arr.remove_edge (e8);

e5 = arr.merge_edge (e5, e6, Segment_2 (e5->source()->point(),
                                         e6->target()->point()));
e2 = arr.merge_edge (e4, e5, s2);

print_arrangement (arr);
return (0);
}

```

Note how we use the halfedge handles returned from *split_edge()* and *merge_edge()*. Also note the insertion of the isolated vertex v_6 located inside the triangular face (the incident face of e_7). This vertex seizes from being isolated, as it is gets connected to other vertices.

In this context, we should mention the two member functions *modify_vertex(v, p)*, which sets p to be the point associated with the vertex v , and *modify_edge(e, c)*, which sets c to be the x -monotone curve associated with the halfedge e . These functions have preconditions that p is geometrically equivalent to $v \rightarrow \text{point}()$ and c is equivalent to $e \rightarrow \text{curve}()$ (i.e., the two curves have the same graph), respectively, to avoid the invalidation of the geometric structure of the arrangement. At a first glance it may seem as these two functions are of little use. However, we should keep in mind that there may be extraneous data (probably non-geometric) associated with the point objects or with the curve objects, as defined by the traits class. With these two functions we can modify this data; see more details in Section 17.5.

In addition, we can use these functions to replace a geometric object (a point or a curve) with an equivalent object that has a more compact representation. For example, we can replace the point $(\frac{20}{40}, \frac{99}{33})$ associated with some vertex v , by $(\frac{1}{2}, 3)$.

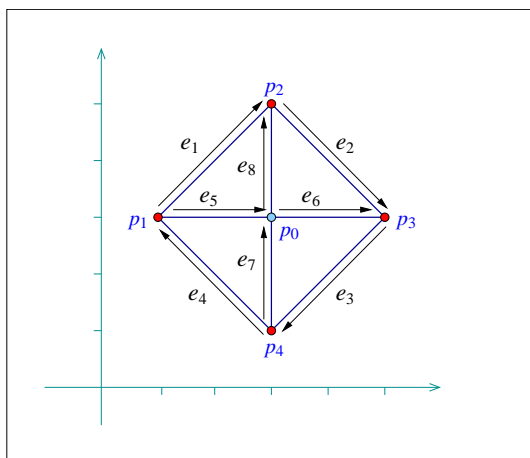
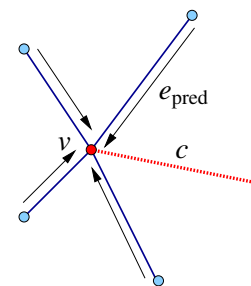


Figure 17.6: An arrangement of line segments, as constructed in *ex_special_edge_insertion.C*. Note that p_0 is initially inserted as an isolated point and later on connected to the other four vertices.

advanced

Advanced Insertion Functions

Assume that the specialized insertion function *insert_from_left_vertex*(c, v) is invoked for a curve c , whose left endpoint is already associated with a non-isolated vertex v . Namely, v has already several incident halfedges. It is necessary in this case to locate the exact place for the new halfedge mapped to the inserted new curve c in the circular list of halfedges incident to v . More precisely, it is sufficient to locate one of the halfedges *pred* directed toward v such that c is located between *pred* and *pred*->next() in clockwise order around v , in order to complete the insertion (see Figure 17.2 for an illustration). This may take $O(d)$ time where d is the degree of the vertex. However, if the halfedge *pred* is known in advance, the insertion can be carried out in constant time.



The *Arrangement_2* class provides the advanced versions of the specialized insertion functions for a curve c — namely we have *insert_from_left_vertex*($c, pred$) and *insert_from_right_vertex*($c, pred$) that accept a halfedge *pred* as specified above, instead of a vertex v . These functions are more efficient, as they take constant time and do not perform any geometric operations. Thus, they should be used when the halfedge *pred* is known. In case that the vertex v is isolated or that the predecessor halfedge for the new inserted curve is not known, the simpler versions of these insertion functions should be used.

Similarly, there exist two overrides of the *insert_at_vertices*() function: One that accept the two predecessor halfedges around the two vertices v_1 and v_2 that correspond to the curve endpoints, and one that accepts a handle for one vertex and a predecessor halfedge around the other vertex.

The following program shows how to construct the arrangement depicted in Figure 17.6 using the specialized insertion functions that accept predecessor halfedges:

```

//! \file examples/Arrangement_2/ex_special_edge_insertion.C
// Constructing an arrangement using the specialized edge-insertion functions.

#include <CGAL/Simple_cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>

```

```

#include <CGAL/Arrangement_2.h>

#include "arr_print.h"

typedef int                                     Number_type;
typedef CGAL::Simple_cartesian<Number_type>    Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>      Traits_2;
typedef Traits_2::Point_2                      Point_2;
typedef Traits_2::X_monotone_curve_2           Segment_2;
typedef CGAL::Arrangement_2<Traits_2>          Arrangement_2;
typedef Arrangement_2::Vertex_handle           Vertex_handle;
typedef Arrangement_2::Halfedge_handle         Halfedge_handle;

int main ()
{
    Arrangement_2    arr;

    Point_2          p0 (3, 3);
    Point_2          p1 (1, 3), p2 (3, 5), p3 (5, 3), p4 (3, 1);
    Segment_2        s1 (p1, p2);
    Segment_2        s2 (p2, p3);
    Segment_2        s3 (p3, p4);
    Segment_2        s4 (p4, p1);
    Segment_2        s5 (p1, p0);
    Segment_2        s6 (p0, p3);
    Segment_2        s7 (p4, p0);
    Segment_2        s8 (p0, p2);

    Vertex_handle    v0 = arr.insert_in_face_interior (p0, arr.unbounded_face());
    Halfedge_handle  e1 = arr.insert_in_face_interior (s1, arr.unbounded_face());
    Halfedge_handle  e2 = arr.insert_from_left_vertex (s2, e1);
    Halfedge_handle  e3 = arr.insert_from_right_vertex (s3, e2);
    Halfedge_handle  e4 = arr.insert_at_vertices (s4, e3, e1->twin());
    Halfedge_handle  e5 = arr.insert_at_vertices (s5, e1->twin(), v0);
    Halfedge_handle  e6 = arr.insert_at_vertices (s6, e5, e3->twin());
    Halfedge_handle  e7 = arr.insert_at_vertices (s7, e4->twin(), e6->twin());
    Halfedge_handle  e8 = arr.insert_at_vertices (s8, e5, e2->twin());

    print_arrangement (arr);
    return (0);
}

```

It is possible to perform even more refined operations on an *Arrangement_2* instance given specific topological information. As most of these operations are very fragile and perform no precondition testing on their input in order to gain efficiency, they are not included in the public interface of the arrangement class. Instead, the *Arr_accessor<Arrangement>* class allows access to these internal arrangement operations — see more details in the Reference Manual.

advanced

17.3 Issuing Queries on an Arrangement

One of the most important query types defined on arrangements is the *point-location* query: Given a point, find the arrangement cell that contains it. Typically, the result of a point-location query is one of the arrangement faces, but in degenerate situations the query point can be located on an edge or coincide with a vertex.

Point-location queries are not only common in many applications, they also play an important role in the free insertion-functions (see Section 17.4). Therefore, it is crucial to have the ability to answer such queries effectively for any arrangement instance.

17.3.1 Point-Location Queries

The arrangement package includes several classes (more precisely, class templates parameterized by an arrangement class) that model the *ArrangementPointLocation_2* concept. Namely, they all have a member function called *locate(q)* that accepts a query point *q* and result with a CGAL *Object* that wraps a handle to the arrangement cell containing the query point. This object can be assigned to either a *Face_const_handle*, *Halfedge_const_handle* or a *Vertex_const_handle*, depending on whether the query point is located inside a face, on an edge or on a vertex.

Note that the handles returned by the *locate()* functions are *const* (non-mutable) handles. If necessary, such handles may be casted to mutable handles using the static functions *Arrangement_2::non_const_handle()* provided by the arrangement class.

An instance of any point-location class must be attached to an *Arrangement_2* instance so we can use it to issue point-location queries. This attachment can be performed when the point-location instance is constructed, or at a later time, using the *init(arr)* method, where *arr* is the attached *Arrangement_2* instance. In this chapter we always use the first option.

The following function template, which can be found in the example file *point_location_utils.h*, accepts a point-location object (whose type can be any of the models to the *ArrangementPointLocation_2* concept) and a query point, and prints out the result of the point-location query for the given point. Observe how we use the function *CGAL::assign()* in order to cast the resulting *CGAL::Object* into a handle to an arrangement feature. The point-location object *pl* is assumed to be already associated with an arrangement:

```
template <class PointLocation>
void point_location_query
    (const PointLocation& pl,
     const typename PointLocation::Arrangement_2::Point_2& q)
{
    // Perform the point-location query.
    CGAL::Object obj = pl.locate (q);

    // Print the result.
    typedef typename PointLocation::Arrangement_2 Arrangement_2;

    typename Arrangement_2::Vertex_const_handle v;
    typename Arrangement_2::Halfedge_const_handle e;
    typename Arrangement_2::Face_const_handle f;

    std::cout << "The point " << q << " is located ";
    if (CGAL::assign (f, obj)) {
        // q is located inside a face:
```

```

    if (f->is_unbounded())
        std::cout << "inside the unbounded face." << std::endl;
    else
        std::cout << "inside a bounded face." << std::endl;
}
else if (CGAL::assign (e, obj)) {
    // q is located on an edge:
    std::cout << "on an edge: " << e->curve() << std::endl;
}
else if (CGAL::assign (v, obj)) {
    // q is located on a vertex:
    if (v->is_isolated())
        std::cout << "on an isolated vertex: " << v->point() << std::endl;
    else
        std::cout << "on a vertex: " << v->point() << std::endl;
}
else {
    CGAL_assertion_msg (false, "Invalid object.");
}
}

```

Choosing a Point-Location Strategy

Each of the various point-location classes employs a different algorithm or *strategy*⁷ for answering queries:

- *Arr_naive_point_location*<Arrangement> locates the query point naïvely, by exhaustively scanning all arrangement cells.
- *Arr_walk_along_a_line_point_location*<Arrangement> simulates a traversal, in reverse order, along an imaginary vertical ray emanating from the query point: It starts from the unbounded face of the arrangement and moves downward toward the query point until locating the arrangement cell containing it.
- *Arr_landmarks_point_location*<Arrangement, Generator> uses a set of “landmark” points whose location in the arrangement is known. Given a query point, it uses a KD-tree to find the nearest landmark and then traverses the straight line segment connecting this landmark to the query point.

There are various ways to select the landmark set in the arrangement, determined by the *Generator* template parameter. By default, the generator class *Arr_landmarks_vertices_generator* is used and the arrangement vertices are the selected landmarks, but other landmark generators, such as sampling random points or choosing points on a grid, are also available; see the Reference Manual for more details.

- *Arr_trapezoid_ric_point_location*<Arrangement> implements Mulmuley’s point-location algorithm [Mul90] (see also [dBvKOS00, Chapter 6]). The arrangement faces are decomposed into simpler cells of constant complexity known as *pseudo-trapezoids* and a search-structure (a directed acyclic graph) is constructed on top of these cells, allowing to locate the pseudo-trapezoid (hence the arrangement cell) containing a query point in expected logarithmic time.

The main advantage of the first two strategies is that they do not require any extra data, so the respective classes just store a pointer to an arrangement object and operate directly on it. Attaching such point-location objects to an existing arrangement has virtually no running-time cost at all, but the query time is linear in the size of the arrangement (the performance of the “walk” strategy is much better in practice, but its worst-case performance

⁷We use the term *strategy* following the design pattern taxonomy [GHJV95].

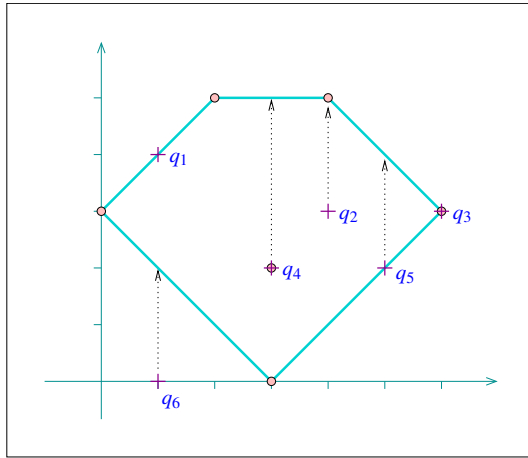


Figure 17.7: The arrangement of line segments, as constructed in *ex_point_location.C*, *ex_vertical_ray_shooting.C*, and *ex_batched_point_location.C*. The arrangement vertices are drawn as small discs, while the query points q_1, \dots, q_6 are marked with crosses.

is linear). Using these strategies is therefore recommended only when a relatively small number of point-location queries are issued by the application, or when the arrangement is constantly changing (i.e., changes in the arrangement structure are more frequent than point-location queries).

On the other hand, the landmarks and the trapezoid RIC strategies require auxiliary data structures on top of the arrangement, which they need to construct once they are attached to an arrangement object and need to keep up-to-date as this arrangement changes. The data structures needed by both strategies can be constructed in $O(N \log N)$ time (where N is the overall number of edges in the arrangement), but the construction needed by the landmark algorithm is in practice significantly faster. In addition, although both resulting data structures are asymptotically linear in size, the KD-tree that the landmark algorithm stores needs significantly less memory. We note that Mulmuley's algorithm guarantees a logarithmic query time, while the query time for the landmark strategy is only logarithmic on average — and we may have scenarios where the query time can be linear. In practice however, the query times of both strategies are competitive. For a detailed experimental comparison, see [HH05]

The main drawback in the current implementation of the landmark strategy, compared to the trapezoidal RIC strategy, is that while the updating the auxiliary data structures related to the trapezoidal decomposition is done very efficiently, the KD-tree maintained by the landmark algorithm needs to be frequently rebuilt as the arrangement changes. In addition, using the landmark point-location class adds some extra requirement from the traits class (that is, the traits class should be a model of a refined concept *ArrangementLandmarkTraits_2*; see Section 17.5 for the details). However, most built-in traits classes that come with the CGAL public release support these extra operations.

It is therefore recommended to use the *Arr_landmarks_point_location* class when the application frequently issues point-location queries on an arrangement that only seldom changes. If the arrangement is more dynamic and is frequently going through changes, the *Arr_trapezoid_ric_point_location* class should be the selected point-location strategy.

An Example

The following program constructs a simple arrangement of five line segments that form a pentagonal face, with a single isolated vertex in its interior, as depicted in Figure 17.7 (the arrangement construction is performed by

the function *construct_segment_arr()* whose listing is omitted here and can be found in *point_location_utils.h*). It then employs the naïve and the landmark strategies to issue several point-location queries on this arrangement:

```

//! \file examples/Arrangement_2/ex_point_location.C
// Answering point-location queries.

#include <CGAL/Simple_cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_naive_point_location.h>
#include <CGAL/Arr_landmarks_point_location.h>

#include "point_location_utils.h"

typedef int                                     Number_type;
typedef CGAL::Simple_cartesian<Number_type>    Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>      Traits_2;
typedef Traits_2::Point_2                      Point_2;
typedef CGAL::Arrangement_2<Traits_2>          Arrangement_2;
typedef CGAL::Arr_naive_point_location<Arrangement_2> Naive_pl;
typedef CGAL::Arr_landmarks_point_location<Arrangement_2> Landmarks_pl;

int main ()
{
    // Construct the arrangement.
    Arrangement_2    arr;
    Naive_pl         naive_pl (arr);
    Landmarks_pl     landmarks_pl;

    construct_segments_arr (arr);

    // Perform some point-location queries using the naive strategy.
    Point_2          q1 (1, 4);
    Point_2          q2 (4, 3);
    Point_2          q3 (6, 3);

    point_location_query (naive_pl, q1);
    point_location_query (naive_pl, q2);
    point_location_query (naive_pl, q3);

    // Attach the landmarks object to the arrangement and perform queries.
    Point_2          q4 (3, 2);
    Point_2          q5 (5, 2);
    Point_2          q6 (1, 0);

    landmarks_pl.attach (arr);

    point_location_query (landmarks_pl, q4);
    point_location_query (landmarks_pl, q5);
    point_location_query (landmarks_pl, q6);

    return (0);
}

```

Note that the program uses the auxiliary *point_location_query()* function template to nicely print the result of each query. This function can be found in the header file *point_location_utils.h*.

17.3.2 Vertical Ray Shooting

Another important query issued on arrangements is the vertical ray-shooting query: Given a query point, which arrangement feature do we encounter if we shoot a vertical ray emanating upward (or downward) from this point? In the general case the ray hits an edge, but it is possible that it hits a vertex, or that the arrangement does not have any feature lying directly above (or below) the query point.

All point-location classes listed in the previous section are also models of the *ArrangementVerticalRayShoot_2* concept. That is, they all have member functions called *ray_shoot_up(q)* and *ray_shoot_down(q)* that accept a query point *q* and output a CGAL *Object*. This can be assigned to either a *Halfedge_const_handle* or to a *Vertex_const_handle*. Alternatively, the returned value is a *Face_const_handle* for the unbounded face of the arrangement, in case there is no edge or vertex lying directly above (or below) *q*.

The following function template, *vertical_ray_shooting_query()*, which also located in the header file *point_location_utils.h*, accepts a vertical ray-shooting object, whose type can be any of the models to the *ArrangementVerticalRayShoot_2* concept and prints out the result of the upward vertical ray-shooting operations from a given query point. The ray-shooting object *vrs* is assumed to be already associated with an arrangement:

```
template <class VerticalRayShoot>
void vertical_ray_shooting_query
    (const VerticalRayShoot& vrs,
     const typename VerticalRayShoot::Arrangement_2::Point_2& q)
{
    // Perform the point-location query.
    CGAL::Object    obj = vrs.ray_shoot_up (q);

    // Print the result.
    typedef typename VerticalRayShoot::Arrangement_2 Arrangement_2;

    typename Arrangement_2::Vertex_const_handle    v;
    typename Arrangement_2::Halfedge_const_handle  e;
    typename Arrangement_2::Face_const_handle      f;

    std::cout << "Shooting up from " << q << " : ";
    if (CGAL::assign (e, obj)) {
        // We hit an edge:
        std::cout << "hit an edge: " << e->curve() << std::endl;
    }
    else if (CGAL::assign (v, obj)) {
        // We hit a vertex:
        if (v->is_isolated())
            std::cout << "hit an isolated vertex: " << v->point() << std::endl;
        else
            std::cout << "hit a vertex: " << v->point() << std::endl;
    }
    else if (CGAL::assign (f, obj)) {
        // We did not hit anything:
        CGAL_assertion (f->is_unbounded());
    }
}
```

```

        std::cout << "hit nothing." << std::endl;
    }
    else {
        CGAL_assertion_msg (false, "Invalid object.");
    }
}

```

The following program uses the auxiliary function listed above to perform vertical ray-shooting queries on an arrangement. The arrangement and the query points are exactly the same as in *ex_point_location.C* (see Figure 17.7):

```

//! \file examples/Arrangement_2/ex_vertical_ray_shooting.C
// Answering vertical ray-shooting queries.

#include <CGAL/MP_Float.h>
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_walk_along_line_point_location.h>
#include <CGAL/Arr_trapezoid_ric_point_location.h>

#include "point_location_utils.h"

typedef CGAL::MP_Float          Number_type;
typedef CGAL::Cartesian<Number_type> Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel> Traits_2;
typedef Traits_2::Point_2      Point_2;
typedef CGAL::Arrangement_2<Traits_2> Arrangement_2;
typedef CGAL::Arr_walk_along_line_point_location<Arrangement_2> Walk_pl;
typedef CGAL::Arr_trapezoid_ric_point_location<Arrangement_2> Trap_pl;

int main ()
{
    // Construct the arrangement.
    Arrangement_2 arr;
    Walk_pl walk_pl (arr);
    Trap_pl trap_pl;

    construct_segments_arr (arr);

    // Perform some vertical ray-shooting queries using the walk strategy.
    Point_2 q1 (1, 4);
    Point_2 q2 (4, 3);
    Point_2 q3 (6, 3);

    vertical_ray_shooting_query (walk_pl, q1);
    vertical_ray_shooting_query (walk_pl, q2);
    vertical_ray_shooting_query (walk_pl, q3);

    // Attach the trapezoid-RIC object to the arrangement and perform queries.
    Point_2 q4 (3, 2);
    Point_2 q5 (5, 2);

```

```

Point_2          q6 (1, 0);

trap_pl.attach (arr);
vertical_ray_shooting_query (trap_pl, q4);
vertical_ray_shooting_query (trap_pl, q5);
vertical_ray_shooting_query (trap_pl, q6);

return (0);
}

```

The number type we use in this example is CGAL's built-in *MP_Float* type which is a floating-point number with an unbounded mantissa and a 32-bit exponent. It supports construction from an integer or from a machine *float* or *double* and performs additions, subtractions and multiplications in an exact number.

17.3.3 Batched Point-Location

Suppose that at a given moment our application has to issue a relatively large number m of point-location queries on a specific arrangement instance. It is possible of course to define a point-location object and to issue separate queries on the arrangement. However, as explained in Section 17.3.1, choosing a simple point-location strategy (either the naïve or the walk strategy) means inefficient queries, while the more sophisticated strategies need to construct auxiliary structures that incur considerable overhead in running time.

On the other hand, the arrangement package includes a free *locate()* function that accepts an arrangement a range of query points as its input and sweeps through the arrangement to locate all query points in one pass. The function outputs the query results as pairs, where each pair is comprised of a query point and a CGAL *Object* that represents the cell containing the point (see Section 17.3.1). The output pairs are sorted in increasing *xy*-lexicographical order of the query point.

The batched point-location operation can be performed in $O((m + N) \log(m + N))$ time, where N is the number of edges in the arrangement. This means that when the number m of point-location queries is of the same order of magnitude as N , this operation is more efficient than issuing separate queries. This suggestion is also backed up by experimental results. Moreover, the batched point-location operation is also advantageous as it does not have to construct and maintain additional data structures.

The following program issues a batched point-location query, which is essentially equivalent to the six separate queries performed in *ex_point_location.C* (see Section 17.3.1):

```

//! \file examples/Arrangement_2/ex_batched_point_location.C
// Answering a batched point-location query.

#include <CGAL/MP_Float.h>
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_batched_point_location.h>
#include <list>

#include "point_location_utils.h"

typedef CGAL::MP_Float          Number_type;
typedef CGAL::Cartesian<Number_type> Kernel;

```

```

typedef CGAL::Arr_segment_traits_2<Kernel>           Traits_2;
typedef Traits_2::Point_2                           Point_2;
typedef CGAL::Arrangement_2<Traits_2>               Arrangement_2;
typedef std::pair<Point_2, CGAL::Object>             Query_result;

int main ()
{
    // Construct the arrangement.
    Arrangement_2    arr;

    construct_segments_arr (arr);

    // Perform a batched point-location query.
    std::list<Point_2>    query_points;
    std::list<Query_result> results;

    query_points.push_back (Point_2 (1, 4));
    query_points.push_back (Point_2 (4, 3));
    query_points.push_back (Point_2 (6, 3));
    query_points.push_back (Point_2 (3, 2));
    query_points.push_back (Point_2 (5, 2));
    query_points.push_back (Point_2 (1, 0));

    locate (arr, query_points.begin(), query_points.end(),
    std::back_inserter (results));

    // Print the results.
    std::list<Query_result>::const_iterator res_iter;
    Arrangement_2::Vertex_const_handle    v;
    Arrangement_2::Halfedge_const_handle  e;
    Arrangement_2::Face_const_handle      f;

    for (res_iter = results.begin(); res_iter != results.end(); ++res_iter)
    {
        std::cout << "The point (" << res_iter->first << ") is located ";
        if (CGAL::assign (f, res_iter->second))
        {
            // The current query point is located inside a face:
            if (f->is_unbounded())
                std::cout << "inside the unbounded face." << std::endl;
            else
                std::cout << "inside a bounded face." << std::endl;
        }
        else if (CGAL::assign (e, res_iter->second))
        {
            // The current query point is located on an edge:
            std::cout << "on an edge: " << e->curve() << std::endl;
        }
        else if (CGAL::assign (v, res_iter->second))
        {
            // The current query point is located on a vertex:
            if (v->is_isolated())
                std::cout << "on an isolated vertex: " << v->point() << std::endl;
            else

```



```

        std::cout << "on a vertex: " << v->point() << std::endl;
    }
}

return (0);
}

```

17.4 Free Functions in the Arrangement Package

In Section 17.2 we reviewed in details the *Arrangement_2* class, which represents two-dimensional subdivisions induced by bounded planar curves, and mentioned that its interface is minimal in the sense that the member functions hardly perform any geometric algorithms and are mainly used for maintaining the topological structure of the subdivision. In this section we explain how to utilize the free (global) functions that operate on arrangements. The implementation of these operations typically require non-trivial geometric algorithms or load some extra requirements on the traits class.

17.4.1 Incremental Insertion Functions

Inserting Non-Intersecting Curves

In Section 17.2 we explained how to construct arrangements of x -monotone curves that are pairwise disjoint in their interior, when the location of the segment endpoints in the arrangement is known. Here we relax this constraint, and allow the location of the inserted x -monotone curve endpoints to be arbitrary, as it may be unknown at the time of insertion. We retain, for the moment, the requirement that the interior of the inserted curve is disjoint from all existing arrangement edges and vertices.

The free function *insert_non_intersecting_curve*(*arr*, *c*, *pl*) inserts the x -monotone curve *c* into the arrangement *arr*, with the precondition that the interior of *c* is disjoint from all *arr*'s existing edges and vertices. The third argument *pl* is a point-location object attached to the arrangement, which is used for performing the insertion. It locates both curve endpoints in the arrangement, where each endpoint is expected to either coincide with an existing vertex or lie inside a face. It is possible to invoke one of the specialized insertion functions (see Section 17.2), based on the query results, and insert *c* at its proper position.⁸ The insertion operation thus hardly requires any geometric operations on top on the ones needed to answer the point-location queries. Moreover, it is sufficient that the arrangement class is instantiated with a traits class that models the *ArrangementBasicTraits_2* concept (or the *ArrangementLandmarkTraits_2* concept, if the landmark point-location strategy is used), which does not have to support the computation of intersection points between curves.

The variant *insert_non_intersecting_curve*(*arr*, *c*) is also available. Instead of accepting a user-defined point-location object, it defines a local instance of the walk point-location class and uses it to insert the curve.

Inserting x -Monotone Curves

The *insert_non_intersecting_curve*() function is very efficient, but its preconditions on the input curves are still rather restricting. Let us assume that the arrangement is instantiated with a traits class that models the refined *ArrangementXMonotoneTraits_2* concept and supports intersection computations (see Section 17.5 for the exact

⁸The *insert_non_intersecting_curve*() function, as all other functions reviewed in this section, is a function template, parameterized by an arrangement class and a point-location class (a model of the *ArrangementPointLocation_2* concept).

details). Given an x -monotone curve, it is sufficient to locate its left endpoint in the arrangement and to trace its *zone*, namely the set of arrangement features crossing the curve, until the right endpoint is reached. Each time the new curve c crosses an existing vertex or an edge, the curve is split into subcurves (in the latter case, we have to split the curve associated with the existing halfedge as well) and associate new edges with the resulting subcurves. Recall that an edge is represented by a pair of twin halfedges, so we split it into two halfedge pairs.

The free function `insert_x_monotone_curve(arr, c, pl)` performs this insertion operation. It accepts an x -monotone curve c , which may intersect some of the curves already in the arrangement arr , and inserts it into the arrangement by computing its zone. Users may supply a point-location object pl , or use the default walk point-location strategy (namely, the variant `insert_x_monotone_curve(arr, c)` is also available). The running-time of this insertion function is proportional to the complexity of the zone of the curve c .

— advanced —

In some cases users may have a prior knowledge of the location of the left endpoint of the x -monotone curve c they wish to insert, so they can perform the insertion without issuing any point-location queries. This can be done by calling `insert_x_monotone_curve(arr, c, obj)`, where obj is an object represents the location of c 's left endpoint in the arrangement — namely it wraps a `Vertex_const_handle`, a `Halfedge_const_handle` or a `Face_const_handle` (see also Section 17.3.1).

— advanced —

Inserting General Curves

So far all our examples were of arrangements of line segments, where the `Arrangement_2` template was instantiated with the `Arr_segment_traits_2` class. In this case, the fact that `insert_x_monotone_curve()` accepts an x -monotone curve does not seem to be a restriction, as all line segments are x -monotone (note that we consider vertical line segments to be *weakly* x -monotone).

Suppose that we construct an arrangement of circles. A circle is obviously not x -monotone, so we cannot use `insert_x_monotone_curve()` in this case.⁹ However, it is possible to subdivide each circle into two x -monotone circular arcs (its upper half and its lower half) and to insert each x -monotone arc separately.

The free function `insert_curve()` requires that the traits class used by the arrangement arr to be a model of the concept `ArrangementTraits_2`, which refines the `ArrangementXMonotoneTraits_2` concept. It has to define an additional `Curve_2` type (which may differ from the `X_monotone_curve_2` type), and support the subdivision of curves of this new type into x -monotone curves (see the exact details in Section 17.5). The `insert_curve(arr, c, pl)` function performs the insertion of the curve c , which does not need to be x -monotone, into the arrangement by subdividing it into x -monotone subcurves and inserting each one separately. Users may supply a point-location object pl , or use the default walk point-location strategy by calling `insert_curve(arr, c)`.

Inserting Points

The arrangement class enables us to insert a point as an isolated vertex in a given face. The free function `insert_point(arr, p, pl)` inserts a vertex into arr that corresponds to the point p at an arbitrary location. It uses the point-location object pl to locate the point in the arrangement (by default, the walk point-location strategy is used), and acts according to the result as follows:

- If p is located inside a face, it is inserted as an isolated vertex inside this face.

⁹Note that a key operation performed by `insert_x_monotone_curve()` is to locate the left endpoint of the curve in the arrangement. A circle, however, does not have any endpoints!

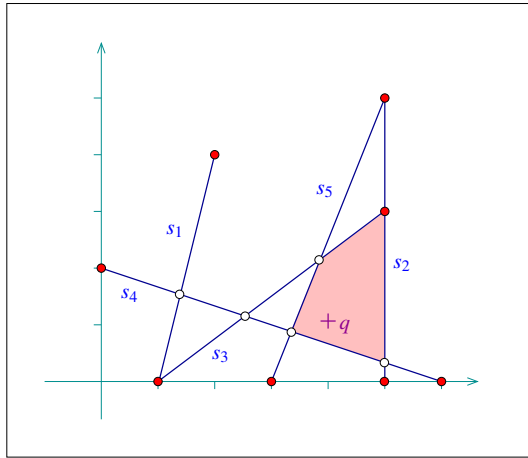


Figure 17.8: An arrangement of five intersecting line segments, as constructed in *ex_incremental_insertion.C* and *ex_aggregated_insertion.C*. The segment endpoints are marked by black disks and the arrangement vertices that correspond to intersection points are marked by circles. The query point q is marked with a cross and the face that contains it is shaded.

- If p lies on an edge, the edge is split to create a vertex associated with p .
- Otherwise, p coincides with an existing vertex and we are done.

In all cases, the function returns a handle to the vertex associated with p .

The arrangement *arr* should be instantiated with a traits class that models the *ArrangementXMonotoneTraits_2* concept, as the insertion operation may involve splitting curves.

An Example

The program below constructs an arrangement of intersecting line-segments. We know that s_1 and s_2 do not intersect, so we use *insert_non_intersecting_curve()* to insert them into the empty arrangement. The rest of the segments are inserted using *insert_x_monotone_curve()* or *insert_curve()*, which are equivalent in case of line segments. The resulting arrangement consists of 13 vertices, 16 edges, and 5 faces, as can be seen in Figure 17.8.

In the earlier examples, all arrangement vertices corresponded to segment endpoints. In this example we have additional vertices that correspond to intersection points between two segments. The coordinates of these intersection points are rational numbers, if the input coordinates are rational (or integer). Therefore, the *Quotient<int>* number type is used to represent the coordinates:

```

//! \file examples/Arrangement_2/ex_incremental_insertion.C
// Using the global incremental insertion functions.

#include <CGAL/Cartesian.h>
#include <CGAL/Quotient.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_walk_along_line_point_location.h>

#include "arr_print.h"

```

```

typedef CGAL::Quotient<int>                               Number_type;
typedef CGAL::Cartesian<Number_type>                      Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>                Traits_2;
typedef Traits_2::Point_2                                 Point_2;
typedef Traits_2::X_monotone_curve_2                     Segment_2;
typedef CGAL::Arrangement_2<Traits_2>                     Arrangement_2;
typedef CGAL::Arr_walk_along_line_point_location<Arrangement_2> Walk_pl;

int main ()
{
    // Construct the arrangement of five intersecting segments.
    Arrangement_2 arr;
    Walk_pl      pl(arr);

    Segment_2     s1 (Point_2(1, 0), Point_2(2, 4));
    Segment_2     s2 (Point_2(5, 0), Point_2(5, 5));
    Segment_2     s3 (Point_2(1, 0), Point_2(5, 3));
    Segment_2     s4 (Point_2(0, 2), Point_2(6, 0));
    Segment_2     s5 (Point_2(3, 0), Point_2(5, 5));

    insert_non_intersecting_curve (arr, s1, pl);
    insert_non_intersecting_curve (arr, s2, pl);
    insert_x_monotone_curve (arr, s3, pl);
    insert_x_monotone_curve (arr, s4, pl);
    insert_curve (arr, s5, pl);

    // Print the size of the arrangement.
    std::cout << "The arrangement size:" << std::endl
              << "    V = " << arr.number_of_vertices()
              << ",    E = " << arr.number_of_edges()
              << ",    F = " << arr.number_of_faces() << std::endl;

    // Perform a point-location query on the resulting arrangement and print
    // the boundary of the face that contains it.
    Point_2      q (4, 1);
    CGAL::Object  obj = pl.locate (q);

    Arrangement_2::Face_const_handle f;
    bool          success = CGAL::assign (f, obj);

    CGAL_assertion (success);
    std::cout << "The query point (" << q << ") is located in: ";
    print_face<Arrangement_2> (f);

    return (0);
}

```

17.4.2 Aggregated Insertion Functions

Let us assume that we have to insert a set of m input curves into an arrangement. It is possible to do this incrementally, inserting the curves one by one, as shown in the previous section. However, the arrangement

package provides three free functions that aggregately insert a range of curves into an arrangement:

- *insert_non_intersecting_curves(arr, begin, end)* inserts a range of x -monotone curves given by the input iterators $[begin, end)$ into an arrangement *arr*. The x -monotone curves should be pairwise disjoint in their interior and also interior-disjoint from all existing edges and vertices of *arr*.
- *insert_x_monotone_curves(arr, begin, end)* operates on a range of x -monotone curves that may intersect one another.
- *insert_curves(arr, begin, end)* inserts a range of of general (not necessarily x -monotone) curves of type *Curve_2*, given by the input iterators $[begin, end)$, into the arrangement *arr*.

We distinguish between two cases: (i) The given arrangement *arr* is empty (has only an unbounded face), so we have to construct it from scratch. (ii) We have to insert m input curves to a non-empty arrangement *arr*.

In the first case, we sweep over the input curves, compute their intersection points and construct the DCEL that represents their planar arrangement. This process is performed in $O((m+k)\log m)$ time, where k is the total number of intersection points. The running time is asymptotically better than the time needed for incremental insertion, if the arrangement is relatively sparse (when k is bounded by $\frac{m^2}{\log m}$), but in practice it is recommended to use this aggregated construction process even for dense arrangements, since the sweep-line algorithm needs less geometric operations compared to the incremental insertion algorithms and hence typically runs much faster in practice.

Another important advantage the aggregated insertion functions have is that they do not issue point-location queries. Thus, no point-location object needs to be attached to the arrangement. As explained in Section 17.3.1, there is a trade-off between construction time and query time in each of the point-location strategies, which affects the running times of the incremental insertion process. Naturally, this trade-off is irrelevant in case of aggregated insertion as above.

The example below shows how to construct the arrangement of line segments depicted in Figure 17.8 and built incrementally in *ex_incremental_insertion.C*, as shown in the previous section. We use the aggregated insertion function *insert_x_monotone_curves()* as we deal with line segments. Note that no point-location object needs to be defined and attached to the arrangement:

```

//! \file examples/Arrangement_2/ex_aggregated_insertion.C
// Using the global aggregated insertion functions.

#include <CGAL/Cartesian.h>
#include <CGAL/Quotient.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <list>

typedef CGAL::Quotient<int>           Number_type;
typedef CGAL::Cartesian<Number_type> Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel> Traits_2;
typedef Traits_2::Point_2            Point_2;
typedef Traits_2::X_monotone_curve_2 Segment_2;
typedef CGAL::Arrangement_2<Traits_2> Arrangement_2;

int main ()
{
    // Construct the arrangement of five intersecting segments.
    Arrangement_2 arr;

```

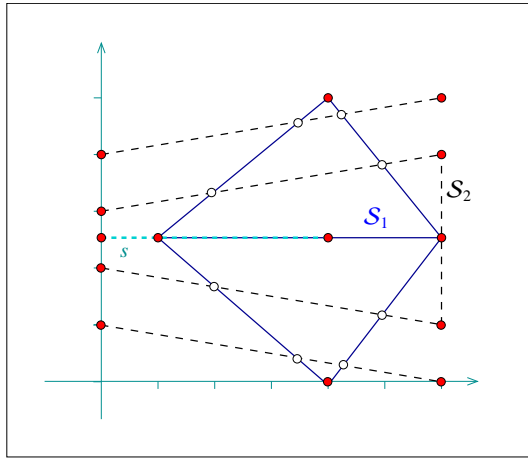


Figure 17.9: An arrangement of intersecting line segments, as constructed in *ex_global_insertion.C*. The segments of \mathcal{S}_1 are drawn in solid lines and the segments of \mathcal{S}_2 are drawn in dark dashed lines. Note that the segment s (light dashed line) overlaps one of the segments in \mathcal{S}_1 .

```
std::list<Segment_2>    segments;;

segments.push_back (Segment_2 (Point_2(1, 0), Point_2(2, 4)));
segments.push_back (Segment_2 (Point_2(5, 0), Point_2(5, 5)));
segments.push_back (Segment_2 (Point_2(1, 0), Point_2(5, 3)));
segments.push_back (Segment_2 (Point_2(0, 2), Point_2(6, 0)));
segments.push_back (Segment_2 (Point_2(3, 0), Point_2(5, 5)));

insert_x_monotone_curves (arr, segments.begin(), segments.end());

// Print the size of the arrangement.
std::cout << "The arrangement size:" << std::endl
    << "    V = " << arr.number_of_vertices()
    << ",    E = " << arr.number_of_edges()
    << ",    F = " << arr.number_of_faces() << std::endl;

return (0);
}
```

In case we have to insert a set of m curves into an existing arrangement, where we denote the number of edges in the arrangement by N . As a rule of thumb, if $m = o(\sqrt{N})$, we insert the curves one by one. For larger input sets, we use the aggregated insertion procedures.

In the example below we aggregately construct an arrangement of a set \mathcal{S}_1 containing five line segments. Then we insert a single segment using the incremental insertion function. Finally, we add a set \mathcal{S}_2 with five more line segments in an aggregated fashion. Notice that the line segments of \mathcal{S}_1 are pairwise interior-disjoint, so we use *insert_non_intersecting_curves()*. \mathcal{S}_2 also contain pairwise interior-disjoint segments, but as they intersect the existing arrangement, we have to use *insert_x_monotone_curves()* to insert them. Also note that the single segment s we insert incrementally overlaps an existing arrangement edge:

```
/// \file examples/Arrangement_2/ex_global_insertion.C
// Using the global insertion functions (incremental and aggregated).
```

```

#include <CGAL/Cartesian.h>
#include <CGAL/Quotient.h>
#include <CGAL/MP_Float.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_naive_point_location.h>

#include "arr_print.h"

typedef CGAL::Quotient<CGAL::MP_Float>          Number_type;
typedef CGAL::Cartesian<Number_type>           Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>      Traits_2;
typedef Traits_2::Point_2                      Point_2;
typedef Traits_2::X_monotone_curve_2           Segment_2;
typedef CGAL::Arrangement_2<Traits_2>          Arrangement_2;
typedef CGAL::Arr_naive_point_location<Arrangement_2> Naive_pl;

int main ()
{
    // Construct the arrangement of five intersecting segments.
    Arrangement_2    arr;
    Segment_2        S1 [5];

    S1[0] = Segment_2 (Point_2 (1, 2.5), Point_2 (4, 5));
    S1[1] = Segment_2 (Point_2 (1, 2.5), Point_2 (6, 2.5));
    S1[2] = Segment_2 (Point_2 (1, 2.5), Point_2 (4, 0));
    S1[3] = Segment_2 (Point_2 (4, 5), Point_2 (6, 2.5));
    S1[4] = Segment_2 (Point_2 (4, 0), Point_2 (6, 2.5));

    insert_non_intersecting_curves (arr, S1, S1 + 5);

    // Perform an incremental insertion of a single overlapping segment.
    Naive_pl          pl (arr);

    insert_x_monotone_curve (arr,
                             Segment_2 (Point_2 (0, 2.5), Point_2 (4, 2.5)),
                             pl);

    // Aggregately insert an additional set of five segments.
    Segment_2        S2 [5];

    S2[0] = Segment_2 (Point_2 (0, 4), Point_2 (6, 5));
    S2[1] = Segment_2 (Point_2 (0, 3), Point_2 (6, 4));
    S2[2] = Segment_2 (Point_2 (0, 2), Point_2 (6, 1));
    S2[3] = Segment_2 (Point_2 (0, 1), Point_2 (6, 0));
    S2[4] = Segment_2 (Point_2 (6, 1), Point_2 (6, 4));

    insert_x_monotone_curves (arr, S2, S2 + 5);

    // Print the size of the arrangement.
    std::cout << "The arrangement size:" << std::endl
               << "    V = " << arr.number_of_vertices()
               << ",    E = " << arr.number_of_edges()

```

```

    << ", F = " << arr.number_of_faces() << std::endl;

    return (0);
}

```

The number type used in the example above, *Quotient<MP_Float>*, is comprised of a numerator and a denominator of type *MP_Float*, namely floating-point numbers with unbounded mantissa. This number type is therefore capable of exactly computing the intersection points as long as the segment endpoints are given as floating-point numbers.

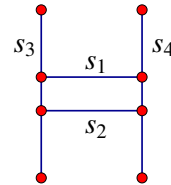
17.4.3 Removing Vertices and Edges

The free functions *remove_vertex()* and *remove_edge()* handle the removal of vertices and edges from an arrangement. The difference between these functions and the member functions of the *Arrangement_2* template having the same name is that they allow the merger of two curves associated with adjacent edges to form a single edge. Thus, they require that the traits class that instantiates the arrangement instance is a model of the refined *ArrangementXMonotoneTraits_2* concept (see Section 17.5).

The function *remove_vertex(arr, v)* removes the vertex *v* from the given arrangement *arr*, where *v* is either an isolated vertex or is a *redundant* vertex — namely, it has exactly two incident edges that are associated with two curves that can be merged to form a single *x*-monotone curve. If neither of the two cases apply, the function returns an indication that it has failed to remove the vertex.

The function *remove_edge(arr, e)* removes the edge *e* from the arrangement by simply calling *arr.remove_edge(e)* (see Section 17.2.3). In addition, if either of the end vertices of *e* becomes isolated or redundant after the removal of the edge, it is removed as well.

The following example demonstrates the usage of the free removal functions. It creates an arrangement of four line segment forming an H-shape with a double horizontal line. Then it removes the two horizontal edges and clears all redundant vertices, such that the final arrangement consists of just two edges associated with the vertical line segments:



```

//! \file examples/Arrangement_2/ex_global_removal.C
// Using the global removal functions.

#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_naive_point_location.h>

#include "arr_print.h"

typedef int
typedef CGAL::Cartesian<Number_type>
typedef CGAL::Arr_segment_traits_2<Kernel>
typedef Traits_2::Point_2
typedef Traits_2::X_monotone_curve_2
typedef CGAL::Arrangement_2<Traits_2>
typedef Arrangement_2::Vertex_handle

Number_type;
Kernel;
Traits_2;
Point_2;
Segment_2;
Arrangement_2;
Vertex_handle;

```



```

typedef Arrangement_2::Halfedge_handle      Halfedge_handle;
typedef CGAL::Arr_naive_point_location<Arrangement_2> Naive_pl;

int main ()
{
    // Create an arrangement of four line segments forming an H-shape.
    Arrangement_2  arr;
    Naive_pl       pl (arr);

    Segment_2      s1 (Point_2 (1, 3), Point_2 (4, 3));
    Halfedge_handle e1 = arr.insert_in_face_interior (s1, arr.unbounded_face());
    Segment_2      s2 (Point_2 (1, 4), Point_2 (4, 4));
    Halfedge_handle e2 = arr.insert_in_face_interior (s2, arr.unbounded_face());
    Segment_2      s3 (Point_2 (1, 1), Point_2 (1, 6));
    Segment_2      s4 (Point_2 (4, 1), Point_2 (4, 6));

    insert_curve (arr, s3, pl);
    insert_curve (arr, s4, pl);

    std::cout << "The initial arrangement:" << std::endl;
    print_arrangement (arr);

    // Remove the horizontal edge e1 from the arrangement using the member
    // function remove_edge(), then remove its end vertices.
    Vertex_handle  v1 = e1->source(), v2 = e1->target();

    arr.remove_edge (e1);
    remove_vertex (arr, v1);
    remove_vertex (arr, v2);

    // Remove the second horizontal edge e2 from the arrangement using the
    // free remove_edge() function.
    remove_edge (arr, e2);

    std::cout << "The final arrangement:" << std::endl;
    print_arrangement (arr);
    return (0);
}

```

17.5 Traits Classes

As mentioned in the introduction of this chapter, the traits class encapsulates the definitions of the geometric entities and implements the geometric predicates and constructions needed by the *Arrangement_2* class and by its peripheral algorithms. We also mention throughout the chapter that there are different levels of requirements from the traits class, namely the traits class can model different concept refinement-levels.

A model of the basic concept, *ArrangementBasicTraits_2*, needs to define the types *Point_2* and *X_monotone_curve_2*, where objects of the first type are the geometric mapping of arrangement vertices, and objects of the latter type are the geometric mapping of edges. In addition, it has to support the following set of predicates:

- Compare the x -coordinates of two points p and q .

- Compare two points p and q lexicographically, by their x -coordinates then by their y -coordinates.
- Return the left endpoint (similarly, the right endpoint) of an x -monotone curve c .
- Given an x -monotone curve c and a point p that lies in its x -range, determine whether p lies below, above or on c .
- Given two x -monotone curves c_1 and c_2 that share a common left endpoint (similarly, right endpoint) p , determine whether c_1 lies above or under c_2 immediately to the right (to the left) of p , or whether the two curves coincide there.
- Check two curves for equality (two curves are equal if their graph is the same).

This basic set of predicates is sufficient for constructing arrangements of x -monotone curves and points that are pairwise disjoint in their interiors and for performing point-location queries and vertical ray-shooting queries.

The landmark point-location strategy (see Section 17.3.1) needs its associated arrangement to be instantiated with a model of the refined *ArrangementLandmarkTraits_2* traits concept. A model of this concept must define a fixed precision number type (typically *double*) and support the additional operations:

- Given a point p , approximate the x and y -coordinates of p using the fixed precision number type. We use this operation for approximate computations — there are certain operations in the search for the location of the point that need not be exact and we can perform them faster than other operations.
- Given two points p_1 and p_2 , construct an x -monotone curve connecting p_1 and p_2 .

A traits class that models the *ArrangementXMonotoneTraits_2* concept, which refines the *ArrangementBasicTraits_2* concept, has to support the following functions:

- Compute all intersection points and overlapping sections of two given x -monotone curves. If possible, compute also the multiplicity of each intersection point.¹⁰ Knowing the multiplicity of an intersection point is not required, but it can speed up the arrangement construction.
- Split an x -monotone curve c into two subcurves at a point p lying in the interior of c .
- Given two x -monotone curve c_1 and c_2 that share a common endpoint, determine whether c_1 and c_2 are *mergeable*, that is, whether they can be merged to form a single continuous x -monotone curve of the type supported by the traits class.
- Merge two mergeable x -monotone curve c_1 and c_2 .

Using a model of the *ArrangementXMonotoneTraits_2*, it is possible to construct arrangements of sets of x -monotone curves (and points) that may intersect one another.

The concept *ArrangementTraits_2* refines the *ArrangementXMonotoneTraits_2* concept by adding the notion of a general, not necessarily x -monotone (and not necessarily connected) curve. A model of this concept must define the *Curve_2* type and support the division of a curve into a set of continuous x -monotone curves and isolated points. For example, the curve $C : (x^2 + y^2)(x^2 + y^2 - 1) = 0$ is the unit circle (the loci of all points for which $x^2 + y^2 = 1$) with the origin $(0,0)$ as a singular point in its interior. C should therefore be divided into two circular arcs (the upper part and the lower part of the unit circle) and a single isolated point.

Note that the refined model *ArrangementTraits_2* is required only when using the free *insert_curve()* and *insert_curves()* functions (see Section 17.4), which accept a *Curve_2* object in the incremental version, or a

¹⁰If the two curves intersect at a point p but have different tangents, p is of multiplicity 1. If the tangents are also equal but the their curvatures are not the same, p is of multiplicity 2, etc.

range of *Curve_2* objects in the aggregated version. In all other cases it is sufficient to use a model of the *ArrangementXMonotoneTraits_2* concept.

In the rest of this section we review the traits classes included in the public distribution of CGAL, that handle line segments, polylines and conic arcs. The last subsection overviews decorators for geometric traits classes distributed with CGAL, which extend other geometric traits-class by attaching auxiliary data with the geometric objects.

17.5.1 Traits Classes for Line Segments

The *Arr_segment_traits_2*<*Kernel*> class used so far in all example programs in this chapter is parameterized by a geometric kernel and uses the *Kernel::Point_2* type as its point type. However, neither the *Curve_2* nor the *X_monotone_curve_2* types are identical to the *Kernel::Segment_2* type. A kernel segment is typically represented by its two endpoints, and these may have a large bit-size representation, if the segment is intersected and split several times (in comparison with the representation of its original endpoints). The large representation may significantly slow down the various traits-class operations involving such a segment. In contrast, the *Arr_segment_traits_2* represents a segment using its supporting line and the two endpoints, such that most computations are performed on the supporting line, which never changes as the segment is split. It also caches some additional information with the segment to speed up various predicates. An *X_monotone_curve_2* object can still be constructed from two endpoints or from a kernel segment. Moreover, an *X_monotone_curve_2* instance can also be casted or assigned to a *Kernel::Segment_2* object. The two types are thus fully convertible to one another.

The *Arr_segment_traits_2*<*Kernel*> class is very efficient for maintaining arrangements of a large number of intersecting line segments, especially if it is instantiated with the appropriate geometric kernel. Using *Cartesian*<*Gmpq*> as the kernel type is generally a good choice; the coordinates of the segment endpoints are represented as exact rational numbers, and this ensures the robustness and correctness of any computation. However, if the GMP library is not installed, it is possible to use the *Quotient*<*MP_Float*> number-type, provided by the support library of CGAL, which is somewhat less efficient.¹¹

Exact computations are of course less efficient, compared to machine-precision floating-point arithmetic, so constructing an arrangement using the *Cartesian*<*Gmpq*> kernel (or, similarly, *Cartesian*<*Quotient*<*MP_Float*>>) is several times slower in comparison to a *Simple_cartesian*<*double*> kernel. However, in the latter case the correctness of the computation results is not guaranteed. In many cases it is possible to use *filtered* computations and benefit from both approaches, namely achieve fast running times with guaranteed results. In case we handle a set of line segments that have no degeneracies, namely no two segments share a common endpoint, and no three segments intersect at a common point — or alternatively, degeneracies exist but their number is relatively small — then filtered computation incur only a minor overhead compared to the floating-point arithmetic, while ensuring the correctness of the computation results.

In the following example we use the predefined *Exact_predicates_exact_constructions_kernel* for instantiating our segment-traits class. This kernel uses interval arithmetic to filter the exact computations. The program reads a set of line segments with integer coordinates from a file and computes their arrangement. By default it opens the *fan_grids.dat* input-file, located in the examples folder, which contains 104 line segments that form four “fan-like” grids and induce a dense arrangement, as illustrated in Figure 17.10(a):

```

//! \file examples/Arrangement_2/ex_predefined_kernel.C
// Constructing an arrangements of intersecting line segments using the
// predefined kernel with exact constructions and exact predicates.

```

¹¹Many of the example programs in the rest of the chapter include a header file named *arr_rational_nt.h*, which defines a type named *Number_type* as either *Gmpq* or *Quotient*<*MP_Float*>, depending on whether GMP is installed or not.

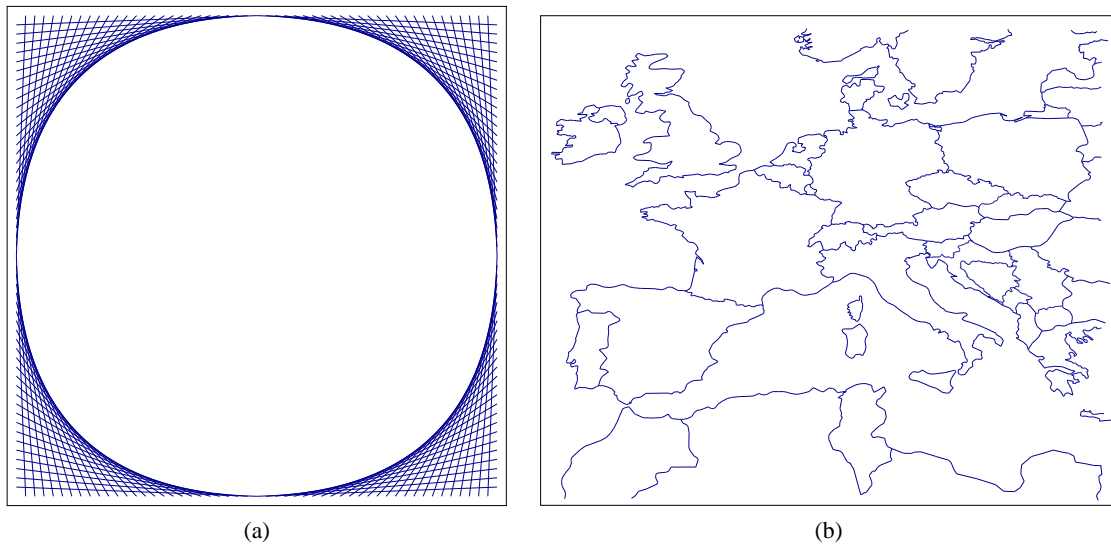


Figure 17.10: (a) An arrangement of 104 line segments from the input file *fan_grids.dat*. (b) An arrangement of more than 3000 interior disjoint line segments, defined in the input file *Europe.dat*.

```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Timer.h>
#include <list>
#include <fstream>

typedef CGAL::Exact_predicates_exact_constructions_kernel Kernel;
typedef Kernel::FT Number_type;
typedef CGAL::Arr_segment_traits_2<Kernel> Traits_2;
typedef Traits_2::Point_2 Point_2;
typedef Traits_2::X_monotone_curve_2 Segment_2;
typedef CGAL::Arrangement_2<Traits_2> Arrangement_2;

int main (int argc, char **argv)
{
    // Get the name of the input file from the command line, or use the default
    // fan_grids.dat file if no command-line parameters are given.
    char *filename = "fan_grids.dat";

    if (argc > 1)
        filename = argv[1];

    // Open the input file.
    std::ifstream in_file (filename);

    if (! in_file.is_open())
    {
        std::cerr << "Failed to open " << filename << " ..." << std::endl;
        return (1);
    }
}
```

```

// Read the segments from the file.
// The input file format should be (all coordinate values are integers):
// <n>                                     // number of segments.
// <sx_1> <sy_1> <tx_1> <ty_1>           // source and target of segment #1.
// <sx_2> <sy_2> <tx_2> <ty_2>           // source and target of segment #2.
//   :       :       :       :
// <sx_n> <sy_n> <tx_n> <ty_n>           // source and target of segment #n.
int                                     n;
std::list<Segment_2> segments;
int                                     sx, sy, tx, ty;
int                                     i;

in_file >> n;
for (i = 0; i < n; i++)
{
    in_file >> sx >> sy >> tx >> ty;

    segments.push_back (Segment_2 (Point_2 (Number_type (sx),
                                             Number_type (sy)),
                                   Point_2 (Number_type (tx),
                                             Number_type (ty))));
}
in_file.close();

// Construct the arrangement by aggregately inserting all segments.
Arrangement_2 arr;
CGAL::Timer timer;

std::cout << "Performing aggregated insertion of "
           << n << " segments." << std::endl;

timer.start();
insert_curves (arr, segments.begin(), segments.end());
timer.stop();

// Print the arrangement dimensions.
std::cout << "V = " << arr.number_of_vertices()
           << ", E = " << arr.number_of_edges()
           << ", F = " << arr.number_of_faces() << std::endl;

std::cout << "Construction took " << timer.time()
           << " seconds." << std::endl;

return 0;
}

```

The arrangement package also offers a simpler alternative segment-traits class. The traits class *Arr_non_caching_segment_basic_traits_2<Kernel>* models the *ArrangementBasicTraits_2* concept. It uses *Kernel::Point_2* as its point type and *Kernel::Segment_2* as its *x*-monotone curve type. As this traits class does not support intersecting and splitting segments, the kernel representation is sufficient. It is still less efficient than *Arr_segment_traits_2* for constructing arrangements of pairwise disjoint line segments in many cases, as it performs no caching at all, but using this traits class may be preferable as it reduces the memory consumption a bit, since no extra data is stored with the line segments.

The class *Arr_non_caching_segment_traits_2<Kernel>* inherits from *Arr_non_caching_segment_basic_traits_2<Kernel>* and extends it to be a model of the *ArrangementTraits_2* concept. It may thus be used to construct arrangement of intersecting line segments, but as explained above, for efficiency reasons it is recommended to use it only when the arrangement is very sparse and contains hardly any intersection points.

In the following example we read an input file containing a set of line segments that are pairwise disjoint in their interior. As the segments do not intersect, no new points are constructed and we can instantiate the *Arr_non_caching_segment_traits_basic_2<Kernel>* class-template with the predefined *Exact_predicates_inexact_constructions_kernel*. Note that we use the *insert_non_intersecting_curves()* function to construct the arrangement. By default, the example opens the *Europe.dat* input-file, located in the examples folder, which contains more than 3000 line segments with floating-point coordinates that form the map of Europe, as depicted in Figure 17.10(b):

```

//! \file examples/Arrangement_2/ex_predefined_kernel_non_intersecting.C
// Constructing an arrangement of non-intersecting line segments using the
// predefined kernel with exact predicates.

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Arr_non_caching_segment_basic_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Timer.h>
#include <list>
#include <fstream>

typedef CGAL::Exact_predicates_inexact_constructions_kernel   Kernel;
typedef Kernel::FT                                             Number_type;
typedef CGAL::Arr_non_caching_segment_basic_traits_2<Kernel> Traits_2;
typedef Traits_2::Point_2                                     Point_2;
typedef Traits_2::X_monotone_curve_2                          Segment_2;
typedef CGAL::Arrangement_2<Traits_2>                         Arrangement_2;

int main (int argc, char **argv)
{
    // Get the name of the input file from the command line, or use the default
    // Europe.dat file if no command-line parameters are given.
    char *filename = "Europe.dat";

    if (argc > 1)
        filename = argv[1];

    // Open the input file.
    std::ifstream in_file (filename);

    if (! in_file.is_open())
    {
        std::cerr << "Failed to open " << filename << " ..." << std::endl;
        return (1);
    }

    // Read the segments from the file.
    // The input file format should be (all coordinate values are double
    // precision floating-point numbers):
    // <n>                                     // number of segments.
    // <sx_1> <sy_1> <tx_1> <ty_1>           // source and target of segment #1.

```

```

// <sx_2> <sy_2> <tx_2> <ty_2>          // source and target of segment #2.
//   :       :       :       :
// <sx_n> <sy_n> <tx_n> <ty_n>          // source and target of segment #n.
int                                     n;
std::list<Segment_2> segments;
double                                sx, sy, tx, ty;
int                                    i;

in_file >> n;
for (i = 0; i < n; i++)
{
    in_file >> sx >> sy >> tx >> ty;

    segments.push_back (Segment_2 (Point_2 (Number_type (sx),
                                              Number_type (sy)),
                                    Point_2 (Number_type (tx),
                                              Number_type (ty))));
}
in_file.close();

// Construct the arrangement by aggregately inserting all segments.
Arrangement_2 arr;
CGAL::Timer timer;

std::cout << "Performing aggregated insertion of "
           << n << " segments." << std::endl;

timer.start();
insert_non_intersecting_curves (arr, segments.begin(), segments.end());
timer.stop();

// Print the arrangement dimensions.
std::cout << "V = " << arr.number_of_vertices()
           << ", E = " << arr.number_of_edges()
           << ", F = " << arr.number_of_faces() << std::endl;

std::cout << "Construction took " << timer.time()
           << " seconds." << std::endl;

return 0;
}

```

17.5.2 The Polyline-Traits Class

The *Arr_polyline_traits_2*<*SegmentTraits*> class can be used to maintain arrangements of polylines (a.k.a. poly-segments), which are continuous piecewise linear curves. A polyline can be created from a range of points, where the i -th and $(i+1)$ -st points in the range represent the endpoints of the i -th segment of the polyline. The polyline traits class is parameterized with a segment-traits class that supports the basic operations on segments.

Polylines are the simplest form of a curves that are not necessarily x -monotone. They can be used to approximate more complicated curves in a convenient manner, as the algebra needed to handle them is elementary — rational arithmetic is sufficient to construct an arrangement of polylines in an exact and robust manner. Note, however,

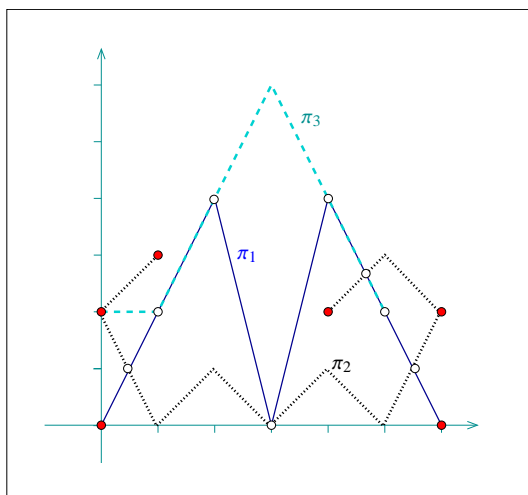


Figure 17.11: An arrangement of three polylines, as constructed in *ex_polylines.C*. Disks mark vertices associated with polyline endpoints, while circles mark vertices that correspond to intersection points. Note that π_2 is split into three x -monotone polylines, and that π_1 and π_3 have two overlapping sections.

that a single polyline can be split into many x -monotone polylines, and that the number of intersection points (or overlapping sections) between two polylines can also be large.

The `polyline-traits` class is a model of the *ArrangementTraits_2* concept and of the *ArrangementLandmarkTraits_2* concept. It inherits its point type from the `segment-traits` class, and defines the `polyline` type, which serves as its *Curve_2*. Polyline curve objects can be constructed from a range of points. They also enable the traversal over the range of defining points, whose first and past-the-end iterators can be obtained through the methods *begin()* and *end()*. The nested *X_monotone_curve_2* type inherits from *Curve_2*. The points in an x -monotone curve are always stored in lexicographically increasing order of their coordinates.

The following example program constructs an arrangement of three polylines, as depicted in Figure 17.11. Note that most points defining the polylines are not associated with arrangement vertices. The arrangement vertices are either the extreme points of each x -monotone polyline or the intersection points between two polylines:

```

//! \file examples/Arrangement_2/ex_polylines.C
// Constructing an arrangement of polylines.

#include <CGAL/Cartesian.h>
#include <CGAL/Quotient.h>
#include <CGAL/MP_Float.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arr_polyline_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <vector>
#include <list>

#include "arr_print.h"

typedef CGAL::Quotient<CGAL::MP_Float>           Number_type;
typedef CGAL::Cartesian<Number_type>             Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>       Segment_traits_2;

```



```

typedef CGAL::Arr_polyline_traits_2<Segment_traits_2> Traits_2;
typedef Traits_2::Point_2 Point_2;
typedef Traits_2::Curve_2 Polyline_2;
typedef CGAL::Arrangement_2<Traits_2> Arrangement_2;

int main ()
{
    Arrangement_2 arr;

    Point_2 points1[5];
    points1[0] = Point_2 (0, 0);
    points1[1] = Point_2 (2, 4);
    points1[2] = Point_2 (3, 0);
    points1[3] = Point_2 (4, 4);
    points1[4] = Point_2 (6, 0);
    Polyline_2 pi1 (&points1[0], &points1[5]);

    std::list<Point_2> points2;
    points2.push_back (Point_2 (1, 3));
    points2.push_back (Point_2 (0, 2));
    points2.push_back (Point_2 (1, 0));
    points2.push_back (Point_2 (2, 1));
    points2.push_back (Point_2 (3, 0));
    points2.push_back (Point_2 (4, 1));
    points2.push_back (Point_2 (5, 0));
    points2.push_back (Point_2 (6, 2));
    points2.push_back (Point_2 (5, 3));
    points2.push_back (Point_2 (4, 2));
    Polyline_2 pi2 (points2.begin(), points2.end());

    std::vector<Point_2> points3 (4);
    points3[0] = Point_2 (0, 2);
    points3[1] = Point_2 (1, 2);
    points3[2] = Point_2 (3, 6);
    points3[3] = Point_2 (5, 2);
    Polyline_2 pi3 (points3.begin(), points3.end());

    insert_curve (arr, pi1);
    insert_curve (arr, pi2);
    insert_curve (arr, pi3);

    print_arrangement (arr);
    return 0;
}

```

17.5.3 A Traits Class for Circular Arcs and Line Segments

Circles and circular arcs are the simplest form of non-linear curves. We handle circles whose centers have rational coordinates and whose squared radii is also rational. If we denote the circle center by (x_0, y_0) and its radius by r , then the equation of the circle — that is, $(x - x_0)^2 + (y - y_0)^2 = r^2$ — has rational coefficients. The intersection points of two such circles are therefore solutions of a quadratic equation with rational coefficients, or algebraic numbers of degree 2. The same applies for intersection points between such a rational circle and a line, or a line segment, with rational coefficients (a line whose equation is $ax + by + c = 0$, where a , b and c are

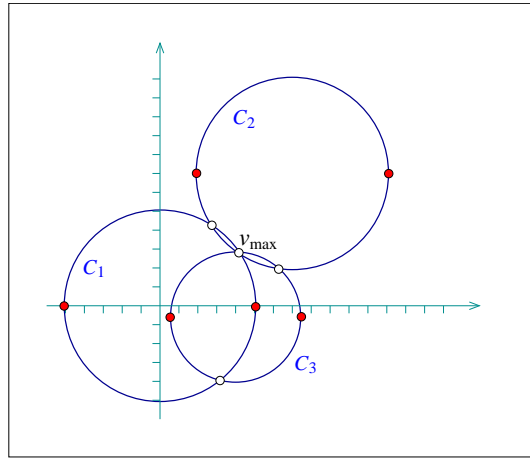


Figure 17.12: An arrangement of three circles constructed in *ex_circles.C*. Each circle is split into two x -monotone circular arcs, whose endpoints are drawn as disks. Circles mark vertices that correspond to intersection points. The vertex v_{\max} is a common intersection point of all three circles.

rational). Such numbers can be expressed as $\alpha + \beta\sqrt{\gamma}$, where α , β and γ are all rational numbers.

Arrangement of circular arcs and of line segment are very useful, as they occur in many applications. For example, when dilating a polygon by some radius we obtain a shape whose boundary is comprised of line segments, which correspond to dilated polygon edges, and circular arcs, which result from dilated polygon vertices. Using the arrangement of the boundary curves it is possible, for example, to compute the union of a set of dilated polygons.

The *Arr_circle_segment_traits_2*<*Kernel*> class-template is designed for efficient handling of arrangements of circular arcs and line segments. It is parameterized by a geometric kernel, and can handle arrangements of segments of *Kernel::Circle_2* objects (full circles are also supported) or of *Kernel::Line_2* objects — namely circular arcs and line segments. It is important to observe that the nested *Point_2* type defined by the traits class, whose coordinates are typically algebraic numbers of degree 2, is *not* the same as the *Kernel::Point_2* type, which is capable of representing a point with rational coordinates. The coordinates of a point are represented using the nested *CoordNT* number-type.

In the following example an arrangement of three full circles is constructed, as shown in Figure 17.12. Then, the vertex of maximal degree is searched for. The geometric mapping of this vertex is the point (4,3), as all three circles intersect at this point and the associated vertex has six incident edges:

```

//! \file examples/Arrangement_2/ex_circles.C
// Constructing an arrangement of circles using the conic-arc traits.

#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_circle_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>

typedef CGAL::Cartesian<Number_type>          Kernel;
typedef Kernel::Circle_2                      Circle_2;
typedef CGAL::Arr_circle_segment_traits_2<Kernel> Traits_2;
typedef Traits_2::CoordNT                    CoordNT;
typedef Traits_2::Point_2                    Point_2;
typedef Traits_2::Curve_2                    Curve_2;

```

```

typedef CGAL::Arrangement_2<Traits_2> Arrangement_2;

int main ()
{
    // Create a circle centered at the origin with radius 5.
    Kernel::Point_2 c1 = Kernel::Point_2 (0, 0);
    Number_type      sqr_r1 = Number_type (25);      // = 5^2
    Circle_2         circ1 = Circle_2 (c1, sqr_r1, CGAL::CLOCKWISE);
    Curve_2          cv1 = Curve_2 (circ1);

    // Create a circle centered at (7,7) with radius 5.
    Kernel::Point_2 c2 = Kernel::Point_2 (7, 7);
    Number_type      sqr_r2 = Number_type (25);      // = 5^2
    Circle_2         circ2 = Circle_2 (c2, sqr_r2, CGAL::CLOCKWISE);
    Curve_2          cv2 = Curve_2 (circ2);

    // Create a circle centered at (4,-0.5) with radius 3.5 (= 7/2).
    Kernel::Point_2 c3 = Kernel::Point_2 (4, Number_type (-1,2));
    Number_type      sqr_r3 = Number_type (49, 4);   // = 3.5^2
    Circle_2         circ3 = Circle_2 (c3, sqr_r3, CGAL::CLOCKWISE);
    Curve_2          cv3 = Curve_2 (circ3);

    // Construct the arrangement of the three circles.
    Arrangement_2    arr;

    insert_curve (arr, cv1);
    insert_curve (arr, cv2);
    insert_curve (arr, cv3);

    // Locate the vertex with maximal degree.
    Arrangement_2::Vertex_const_iterator vit;
    Arrangement_2::Vertex_const_handle   v_max;
    unsigned int                         max_degree = 0;

    for (vit = arr.vertices_begin(); vit != arr.vertices_end(); ++vit)
    {
        if (vit->degree() > max_degree)
        {
            v_max = vit;
            max_degree = vit->degree();
        }
    }

    std::cout << "The vertex with maximal degree in the arrangement is: "
               << "v_max = (" << v_max->point() << ") "
               << "with degree " << max_degree << "." << std::endl;

    return (0);
}

```

The *Curve_2* type nested in *Arr_circle_segment_traits_2* can be used to represent circles, circular arcs, or line segments. Curve objects can therefore be constructed from a *Kernel::Circle_2* object or from a *Kernel::Segment_2* object. A circular arc is typically defined by a supporting circle and two endpoints, where

the endpoints are instances of the *Point_2* type, with rational or irrational coordinates. The orientation of the arc is determined by the orientation of the supporting circle. Similarly, we also support the construction of lines segments given their supporting line (of type *Kernel::Line_2*) and two endpoints, which may have irrational coordinates (unlike the *Kernel::Segment_2* type).

Note that the *Kernel::Circle_2* type represents a circle whose *squared radius* is rational, where the radius itself may be irrational. However, if the radius is known to be rational, it is advisable to use it, for efficiency reasons. It is therefore also possible to construct a circle, or a circular arc specifying the circle center (a *Kernel::Point_2*), its rational radius, and its orientation. Finally, we also support the construction of a circular arcs that is defined by two endpoints and an arbitrary midpoint that lies on the arc in between its endpoint. In this case, all three points are required to have rational coordinates (to be kernel points).

The following example demonstrates the usage of the various construction methods for circular arcs and line segments. Note the usage of the constructor of *CoordNT* (*alpha*, *beta*, *gamma*), which creates a degree-2 algebraic number whose value is $\alpha + \beta\sqrt{\gamma}$.

```

//! \file examples/Arrangement_2/ex_circular_arc.C
// Constructing an arrangement of various circular arcs and line segments.

#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_circle_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>

typedef CGAL::Cartesian<Number_type>           Kernel;
typedef Kernel::Circle_2                       Circle_2;
typedef Kernel::Segment_2                     Segment_2;
typedef CGAL::Arr_circle_segment_traits_2<Kernel> Traits_2;
typedef Traits_2::CoordNT                     CoordNT;
typedef Traits_2::Point_2                     Point_2;
typedef Traits_2::Curve_2                     Curve_2;
typedef CGAL::Arrangement_2<Traits_2>         Arrangement_2;

int main ()
{
    std::list<Curve_2>  curves;

    // Create a circle centered at the origin with squared radius 2.
    Kernel::Point_2  c1 = Kernel::Point_2 (0, 0);
    Circle_2         circ1 = Circle_2 (c1, Number_type (2));

    curves.push_back (Curve_2 (circ1));

    // Create a circle centered at (2,3) with radius 3/2 - note that
    // as the radius is rational we use a different curve constructor.
    Kernel::Point_2  c2 = Kernel::Point_2 (2, 3);

    curves.push_back (Curve_2 (c2, Number_type(3, 2)));

    // Create a segment of the line (y = x) with rational endpoints.
    Kernel::Point_2  s3 = Kernel::Point_2 (-2, -2);
    Kernel::Point_2  t3 = Kernel::Point_2 (2, 2);
    Segment_2        seg3 = Segment_2 (s3, t3);

```

```

curves.push_back (Curve_2 (seg3));

// Create a line segment with the same supporting line ( $y = x$ ), but
// having one endpoint with irrational coefficients.
CoordNT      sqrt_15 = CoordNT (0, 1, 15); // = sqrt(15)
Point_2      s4 = Point_2 (3, 3);
Point_2      t4 = Point_2 (sqrt_15, sqrt_15);

curves.push_back (Curve_2 (seg3.supporting_line(), s4, t4));

// Create a circular arc that correspond to the upper half of the
// circle centered at (1,1) with squared radius 3. We create the
// circle with clockwise orientation, so the arc is directed from
//  $(1 - \sqrt{3}, 1)$  to  $(1 + \sqrt{3}, 1)$ .
Kernel::Point_2 c5 = Kernel::Point_2 (1, 1);
Circle_2      circ5 = Circle_2 (c5, 3, CGAL::CLOCKWISE);
CoordNT      one_minus_sqrt_3 = CoordNT (1, -1, 3);
CoordNT      one_plus_sqrt_3 = CoordNT (1, 1, 3);
Point_2      s5 = Point_2 (one_minus_sqrt_3, CoordNT (1));
Point_2      t5 = Point_2 (one_plus_sqrt_3, CoordNT (1));

curves.push_back (Curve_2 (circ5, s5, t5));

// Create a circular arc of the unit circle, directed clockwise from
//  $(-1/2, \sqrt{3}/2)$  to  $(1/2, \sqrt{3}/2)$ . Note that we orient the
// supporting circle accordingly.
Kernel::Point_2 c6 = Kernel::Point_2 (0, 0);
CoordNT      sqrt_3_div_2 = CoordNT (0, Number_type(1,2), 3);
Point_2      s6 = Point_2 (Number_type (-1, 2), sqrt_3_div_2);
Point_2      t6 = Point_2 (Number_type (1, 2), sqrt_3_div_2);

curves.push_back (Curve_2 (c6, 1, CGAL::CLOCKWISE, s6, t6));

// Create a circular arc defined by two endpoints and a midpoint,
// all having rational coordinates. This arc is the upper-right
// quarter of a circle centered at the origin with radius 5.
Kernel::Point_2 s7 = Kernel::Point_2 (0, 5);
Kernel::Point_2 mid7 = Kernel::Point_2 (3, 4);
Kernel::Point_2 t7 = Kernel::Point_2 (5, 0);

curves.push_back (Curve_2 (s7, mid7, t7));

// Construct the arrangement of the curves.
Arrangement_2 arr;

insert_curves (arr, curves.begin(), curves.end());

// Print the size of the arrangement.
std::cout << "The arrangement size:" << std::endl
    << "    V = " << arr.number_of_vertices()
    << ",    E = " << arr.number_of_edges()
    << ",    F = " << arr.number_of_faces() << std::endl;

return (0);

```

}

It is also possible to construct x -monotone curve objects, which represent x -monotone circular arcs or line segments, using similar constructors. Construction from a full circle is obviously not supported. See the Reference Manual for more details.

17.5.4 A Traits Class for Conic Arcs

A *conic curve* is an algebraic curve of degree 2. Namely, it is the locus of all points (x, y) satisfying the equation $C: rx^2 + sy^2 + txy + ux + vy + w = 0$, where the six coefficients $\langle r, s, t, u, v, w \rangle$ completely characterize the curve. The sign of the expression $\Delta_C = 4rs - t^2$ determines the type of curve:

- If $\Delta_C > 0$ the curve is an ellipse. A circle is a special case of an ellipse, where $r = s$ and $t = 0$.
- If $\Delta_C = 0$ the curve is a parabola — an unbounded conic curve with a single connected branch. When $r = s = t = 0$ we have a line, which can be considered as a degenerate parabola.
- If $\Delta_C < 0$ the curve is a hyperbola. That is, it is comprised of two disconnected unbounded branches.

As the arrangement package is suitable for bounded curves, we consider bounded segments of conic curves, referred to as *conic arcs*. A conic arc a may be either (i) a full ellipse, or (ii) defined by the tuple $\langle C, p_s, p_t, o \rangle$, where C is a conic curve and p_s and p_t are two points on C (namely $C(p_s) = C(p_t) = 0$) that define the *source* and *target* of the arc, respectively. The arc is formed by traversing C from the source to the target going in the orientation specified by o , which is typically clockwise or counterclockwise orientation (but may also be collinear in case of degenerate conic curves).

We always assume that the conic coefficients $\langle r, s, t, u, v, w \rangle$ are rational. When dealing with linear curves (line segments and polylines), similar assumptions guarantee that all intersection points also have rational coordinates, such that the arrangement of such curves can be constructed and maintained using only rational arithmetic. Unfortunately, this does not hold for conic curves, as the coordinates of intersection points of two conic curves with rational coefficients are in general algebraic numbers of degree 4.¹² In addition, conic arcs may not necessarily be x -monotone, and must be split at points where the tangent to the arc is vertical. In the general case, such points typically have coordinates that are algebraic numbers of degree 2. It is therefore clear that we have to use different number types to represent the conic coefficients and the point coordinates. Note that as arrangement vertices induced by intersection points and points with vertical tangents are likely to have algebraic coordinates, we also allow the original endpoints of the input arcs p_s and p_t to have algebraic coordinates.

The `Arr_conic_traits_2<RatKernel, AlgKernel, NtTraits>` class template is designed for efficient handling of arrangements of bounded conic arcs. The template has three parameters, defined as follows:

- The `RatKernel` class is a geometric kernel, whose field type is an exact rational type. It is used to define basic geometric entities (e.g., a line segment or a circle) with rational coefficients. Typically we use one of the standard CGAL kernels, instantiated with the number type `NtTraits::Rational` (see below).
- The `AlgKernel` class is a geometric kernel whose field type is an exact algebraic type. It is used to define points with algebraic coordinates. Typically we use one of the standard CGAL kernels, instantiated with the number type `NtTraits::Algebraic` (see below).

¹²Namely, they are roots of polynomials with integer coefficients of degree 4. However, in some special cases, for example when handling only circles and circular arcs, the coordinates of the intersection points are only of degree 2, namely they are solutions of quadratic equations.

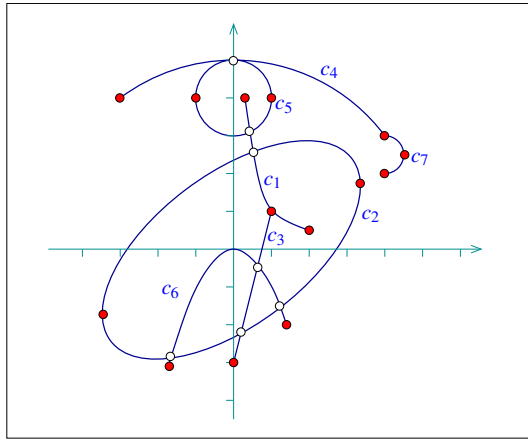


Figure 17.13: An arrangement of mixed conic arcs, as constructed in *ex_conics.C*.

- The *NtTraits* class (the number-type traits class) encapsulates all the numeric operations needed for performing the geometric computation carried out by the geometric traits class. It defines the *Integer*, *Rational* and *Algebraic* number-types, and supports several operations on these types, such as conversion between number types, solving quadratic equations and extracting the real roots of a polynomial with integer coefficients. It is highly recommended to use the *CORE_algebraic_number_traits* class, which is included in the arrangement package. It relies on the exact number types implemented in the *CORE* library and performs exact computations on the number types it defines.

The *Arr_conic_traits_2* models the *ArrangementTraits_2* and the *ArrangementLandmarkTraits_2* concepts. (It supports the landmark point-location strategy). Its *Point_2* type is derived from *AlgKernel::Point_2*, while the *Curve_2* type represents a bounded, not necessarily *x*-monotone, conic arc. The *X_monotone_curve_2* type is derived from *Curve_2*, but its constructors are to be used only by the traits class. Users should therefore construct only *Curve_2* objects and insert them into the arrangement using the *insert_curve()* or *insert_curves()* functions.

Conic arcs can be constructed from full ellipses or by specifying a supporting curve, two endpoints and an orientation. However, several constructors of *Curve_2* are available to allow for some special cases, such as circular arcs or line segments. The *Curve_2* (and the derived *X_monotone_curve_2*) classes also support basic access functions such as *source()*, *target()* and *orientation()*.

Examples for Arrangements of Conics

The following example demonstrates the usage of the various constructors for conic arcs. The resulting arrangement is depicted in Figure 17.13. Especially noteworthy are the constructor of a circular arc that accepts three points and the constructor that allows specifying approximate endpoints, where the exact endpoints are given explicitly as intersections of the supporting conic with two other conic curves. Also note that as the preconditions required by some of these constructors are rather complicated (see the Reference Manual for the details), a precondition violation does not cause the program to terminate — instead, an *invalid* arc is created. We can verify the validity of an arc by using the *is_valid()* method. Needless to say, inserting invalid arcs into an arrangement is not allowed.

```

/// \file examples/Arrangement_2/ex_conics.C
// Constructing an arrangement of various conic arcs.
#include <CGAL/basic.h>
```

```

#ifndef CGAL_USE_CORE
#include <iostream>
int main ()
{
    std::cout << "Sorry, this example needs CORE ..." << std::endl;
    return (0);
}
#else

#include <CGAL/Cartesian.h>
#include <CGAL/CORE_algebraic_number_traits.h>
#include <CGAL/Arr_conic_traits_2.h>
#include <CGAL/Arrangement_2.h>

typedef CGAL::CORE_algebraic_number_traits          Nt_traits;
typedef Nt_traits::Rational                        Rational;
typedef Nt_traits::Algebraic                      Algebraic;
typedef CGAL::Cartesian<Rational>                  Rat_kernel;
typedef Rat_kernel::Point_2                        Rat_point_2;
typedef Rat_kernel::Segment_2                     Rat_segment_2;
typedef Rat_kernel::Circle_2                      Rat_circle_2;
typedef CGAL::Cartesian<Algebraic>                 Alg_kernel;
typedef CGAL::Arr_conic_traits_2<Rat_kernel,
                                Alg_kernel,
                                Nt_traits>          Traits_2;
typedef Traits_2::Point_2                         Point_2;
typedef Traits_2::Curve_2                         Conic_arc_2;
typedef CGAL::Arrangement_2<Traits_2>              Arrangement_2;

int main ()
{
    Arrangement_2    arr;

    // Insert a hyperbolic arc, supported by the hyperbola  $y = 1/x$ 
    // (or:  $xy - 1 = 0$ ) with the endpoints (1/5, 4) and (2, 1/2).
    // Note that the arc is counterclockwise oriented.
    Point_2      ps1 (Rational(1,4), 4);
    Point_2      pt1 (2, Rational(1,2));
    Conic_arc_2   c1 (0, 0, 1, 0, 0, -1, CGAL::COUNTERCLOCKWISE, ps1, pt1);

    insert_curve (arr, c1);

    // Insert a full ellipse, which is  $(x/4)^2 + (y/2)^2 = 0$  rotated by
    //  $\phi=36.87$  degree (such that  $\sin(\phi) = 0.6$ ,  $\cos(\phi) = 0.8$ ),
    // yielding:  $58x^2 + 72y^2 - 48xy - 360 = 0$ .
    Conic_arc_2   c2 (58, 72, -48, 0, 0, -360);

    insert_curve (arr, c2);

    // Insert the segment (1, 1) -- (0, -3).
    Rat_point_2   ps3 (1, 1);
    Rat_point_2   pt3 (0, -3);
    Conic_arc_2   c3 (Rat_segment_2 (ps3, pt3));

```



```

insert_curve (arr, c3);

// Insert a circular arc supported by the circle  $x^2 + y^2 = 5^2$ ,
// with (-3, 4) and (4, 3) as its endpoints. We want the arc to be
// clockwise oriented, so it passes through (0, 5) as well.
Rat_point_2   ps4 (-3, 4);
Rat_point_2   pm4 (0, 5);
Rat_point_2   pt4 (4, 3);
Conic_arc_2    c4 (ps4, pm4, pt4);

CGAL_assertion (c4.is_valid());
insert_curve (arr, c4);

// Insert a full unit circle that is centered at (0, 4).
Rat_circle_2   circ5 (Rat_point_2(0,4), 1);
Conic_arc_2    c5 (circ5);

insert_curve (arr, c5);

// Insert a parabolic arc that is supported by a parabola  $y = -x^2$ 
// (or:  $x^2 + y = 0$ ) and whose endpoints are  $(-\sqrt{3}, -3) \sim (-1.73, -3)$ 
// and  $(\sqrt{2}, -2) \sim (1.41, -2)$ . Notice that since the x-coordinates
// of the endpoints cannot be accurately represented, we specify them
// as the intersections of the parabola with the lines  $y = -3$  and  $y = -2$ .
// Note that the arc is clockwise oriented.
Conic_arc_2    c6 =
    Conic_arc_2 (1, 0, 0, 0, 1, 0,          // The parabola.
                 CGAL::CLOCKWISE,
                 Point_2 (-1.73, -3),      // Approximation of the source.
                 0, 0, 0, 0, 1, 3,        // The line:  $y = -3$ .
                 Point_2 (1.41, -2),      // Approximation of the target.
                 0, 0, 0, 0, 1, 2);       // The line:  $y = -2$ .

CGAL_assertion (c6.is_valid());
insert_curve (arr, c6);

// Insert the right half of the circle centered at (4, 2.5) whose radius
// is 1/2 (therefore its squared radius is 1/4).
Rat_circle_2   circ7 (Rat_point_2(4, Rational(5,2)), Rational(1,4));
Point_2        ps7 (4, 3);
Point_2        pt7 (4, 2);
Conic_arc_2    c7 (circ7, CGAL::CLOCKWISE, ps7, pt7);

insert_curve (arr, c7);

// Print out the size of the resulting arrangement.
std::cout << "The arrangement size:" << std::endl
    << "    V = " << arr.number_of_vertices()
    << ",    E = " << arr.number_of_edges()
    << ",    F = " << arr.number_of_faces() << std::endl;

return (0);
}

```

```
#endif
```

The last example in this section demonstrates how the conic-traits class can handle intersection points with multiplicity. The supporting curves of the two arcs, a circle centered at $(0, \frac{1}{2})$ with radius $\frac{1}{2}$, and the hyperbola $y = \frac{x^2}{1-x}$,¹³ intersect at the origin such that the intersection point has multiplicity 3 (note that they both have the same horizontal tangent at $(0,0)$ and the same curvature 1). In addition, they have another intersection point at $(\frac{1}{2}, \frac{1}{2})$ of multiplicity 1:

```
#!/ \file examples/Arrangement_2/ex_conic_multiplicities.C
// Handling intersection points with multiplicity between conic arcs.
#include <CGAL/basic.h>

#ifdef CGAL_USE_CORE
#include <iostream>
int main ()
{
    std::cout << "Sorry, this example needs CORE ..." << std::endl;
    return (0);
}
#else

#include <CGAL/Cartesian.h>
#include <CGAL/CORE_algebraic_number_traits.h>
#include <CGAL/Arr_conic_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_naive_point_location.h>

#include "arr_print.h"

typedef CGAL::CORE_algebraic_number_traits          Nt_traits;
typedef Nt_traits::Rational                        Rational;
typedef Nt_traits::Algebraic                      Algebraic;
typedef CGAL::Cartesian<Rational>                 Rat_kernel;
typedef Rat_kernel::Point_2                       Rat_point_2;
typedef Rat_kernel::Segment_2                    Rat_segment_2;
typedef Rat_kernel::Circle_2                     Rat_circle_2;
typedef CGAL::Cartesian<Algebraic>               Alg_kernel;
typedef CGAL::Arr_conic_traits_2<Rat_kernel,
                                Alg_kernel,
                                Nt_traits>         Traits_2;
typedef Traits_2::Point_2                        Point_2;
typedef Traits_2::Curve_2                       Conic_arc_2;
typedef CGAL::Arrangement_2<Traits_2>            Arrangement_2;
typedef CGAL::Arr_naive_point_location<Arrangement_2> Naive_pl;

int main ()
{
    Arrangement_2  arr;
    Naive_pl       pl (arr);
```

¹³This curve can also be written as $C: x^2 + xy - y = 0$. It is a hyperbola since $\Delta_C = -1$.

```

// Insert a hyperbolic arc, supported by the hyperbola  $y = x^2/(1-x)$ 
// (or:  $x^2 + xy - y = 0$ ) with the endpoints  $(-1, 1/2)$  and  $(1/2, 1/2)$ .
// Note that the arc is counterclockwise oriented.
Point_2      ps1 (-1, Rational(1,2));
Point_2      pt1 (Rational(1,2), Rational(1,2));
Conic_arc_2   cv1 (1, 0, 1, 0, -1, 0, CGAL::COUNTERCLOCKWISE, ps1, pt1);

insert_curve (arr, cv1, pl);

// Insert the bottom half of the circle centered at  $(0, 1/2)$  whose radius
// is  $1/2$  (therefore its squared radius is  $1/4$ ).
Rat_circle_2  circ2 (Rat_point_2(0, Rational(1,2)), Rational(1,4));
Point_2      ps2 (-Rational(1,2), Rational(1,2));
Point_2      pt2 (Rational(1,2), Rational(1,2));
Conic_arc_2   cv2 (circ2, CGAL::COUNTERCLOCKWISE, ps2, pt2);

insert_curve (arr, cv2, pl);

// Print the resulting arrangement.
print_arrangement (arr);

return (0);
}

#endif

```

17.5.5 A Traits Class for Arcs of Rational Functions

A *rational function* is given by the equation $y = \frac{P(x)}{Q(x)}$, where P and Q are polynomials of arbitrary degrees. In particular, if $Q(x) = 1$, then the function is a simple polynomial function. A bounded *rational arc* is defined by the graph of a rational function over some interval $[x_{\min}, x_{\max}]$, where Q does not have any real roots in this interval (Thus, the arc does not contain any poles). Rational functions, and polynomial functions in particular, are not only interesting in their own right, they are also very useful for approximating or interpolating more complicated curves; see, e.g., [PTVF02, Chapter 3].

The computations with rational arcs are guaranteed to be robust and exact, assuming that the coefficient of the polynomials P and Q are rational numbers. The x -values that determine the interval over which the arc is defined can however be arbitrary algebraic numbers.

Using the `Arr_rational_traits_2<AlgKernel, NtTraits>` class template it is possible to construct and maintain arrangement of rational arcs. The template parameters are very similar to the ones used by the `Arr_conic_traits_2` class template; see the previous section. However, no rational kernel is needed. Also in this case it is recommended to use the `CORE_algebraic_number_traits` class, with a kernel instantiated with the *Algebraic* type defined by this class.

The `Arr_rational_traits_2` is a model of the `ArrangementTraits_2` concept (but not of the `ArrangementLandmarkTraits_2` concept, so it is not possible to use the landmark point-location strategy for arrangements of rational arcs). Its `Point_2` type is derived from `AlgKernel::Point_2`, while the `Curve_2` and `X_monotone_curve_2` types refer to the same class (note that a rational arc is always x -monotone). The traits class also defines the `Rat_vector` type, representing a vector of rational coefficients, (whose type is `NtTraits::Rational`). A rational arc can be constructed from a single vector of coefficients, specifying the polynomial P alone (and $Q(x) = 1$), or from two vectors of coefficients, specifying both P and Q .

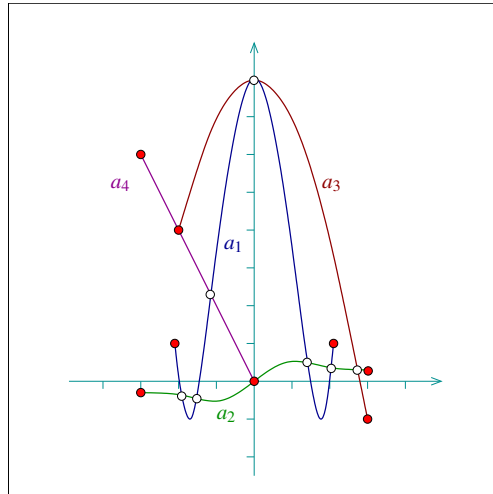


Figure 17.14: An arrangement of four arcs of rational functions, as constructed in *ex_rational_functions.C*.

The following example demonstrates the construction of an arrangement of rational arcs depicted in Figure 17.14. Note the usage of the two constructors, for polynomial arcs and for rational arcs:

```

//! \file examples/Arrangement_2/ex_rational_functions.C
// Constructing an arrangement of arcs of rational functions.
#include <CGAL/basic.h>

#ifdef CGAL_USE_CORE
#include <iostream>
int main ()
{
    std::cout << "Sorry, this example needs CORE ..." << std::endl;
    return (0);
}
#else

#include <CGAL/Cartesian.h>
#include <CGAL/CORE_algebraic_number_traits.h>
#include <CGAL/Arr_rational_arc_traits_2.h>
#include <CGAL/Arrangement_2.h>

typedef CGAL::CORE_algebraic_number_traits Nt_traits;
typedef Nt_traits::Rational Rational;
typedef Nt_traits::Algebraic Algebraic;
typedef CGAL::Cartesian<Algebraic> Alg_kernel;
typedef CGAL::Arr_rational_arc_traits_2<Alg_kernel,
                                       Nt_traits> Traits_2;
typedef Traits_2::Point_2 Point_2;
typedef Traits_2::Curve_2 Rational_arc_2;
typedef Traits_2::Rat_vector Rat_vector;
typedef std::list<Rational_arc_2> Rat_arcs_list;
typedef CGAL::Arrangement_2<Traits_2> Arrangement_2;

int main ()

```

```

{
    // Create an arc supported by the polynomial  $y = x^4 - 6x^2 + 8$ ,
    // defined over the interval  $[-2.1, 2.1]$ :
    Rat_vector      P1(5);
    P1[4] = 1; P1[3] = 0; P1[2] = -6; P1[1] = 0; P1[0] = 8;

    Rational_arc_2   a1 (P1, Algebraic(-2.1), Algebraic(2.1));

    // Create an arc supported by the function  $y = x / (1 + x^2)$ ,
    // defined over the interval  $[-3, 3]$ :
    Rat_vector      P2(2);
    P2[1] = 1; P2[0] = 0;

    Rat_vector      Q2(3);
    Q2[2] = 1; Q2[1] = 0; Q2[0] = 1;

    Rational_arc_2   a2 (P2, Q2, Algebraic(-3), Algebraic(3));

    // Create an arc supported by the parabola  $y = 8 - x^2$ ,
    // defined over the interval  $[-2, 3]$ :
    Rat_vector      P3(5);
    P3[2] = -1; P3[1] = 0; P3[0] = 8;

    Rational_arc_2   a3 (P3, Algebraic(-2), Algebraic(3));

    // Create an arc supported by the line  $y = -2x$ ,
    // defined over the interval  $[-3, 0]$ :
    Rat_vector      P4(2);
    P4[1] = -2; P4[0] = 0;

    Rational_arc_2   a4 (P4, Algebraic(-3), Algebraic(0));

    // Construct the arrangement of the four arcs.
    Arrangement_2     arr;
    std::list<Rational_arc_2> arcs;

    arcs.push_back (a1);
    arcs.push_back (a2);
    arcs.push_back (a3);
    arcs.push_back (a4);
    insert_curves (arr, arcs.begin(), arcs.end());

    // Print the arrangement size.
    std::cout << "The arrangement size:" << std::endl
              << "    V = " << arr.number_of_vertices()
              << ",    E = " << arr.number_of_edges()
              << ",    F = " << arr.number_of_faces() << std::endl;

    return 0;
}

#endif

```

17.5.6 Traits-Class Decorators

Geometric traits-class decorators allow users to attach auxiliary data to curves and to points. The data is automatically manipulated by the decorators and distributed to the constructed geometric entities. Note that additional information can alternatively be maintained by extending the vertex, halfedge, or face types provided by the DCEL class used by the arrangement; see the details in Section 17.7.

The arrangement package includes a generic traits-class decorator template named *Arr_curve_data_traits_2*<*BaseTraits*, *XMonotoneCurveData*, *Merge*, *CurveData*, *Convert*>. This decorator is used to attach a data field to curves and to *x*-monotone curves. It is parameterized by a base-traits class, which is one of the geometric traits classes described in the previous subsections, or a user-defined traits class. The curve-data decorator derives itself from the base-traits class, and in particular inherits its *Point_2* type. In addition:

- *Curve_2* is derived from the basic *BaseTraits::Curve_2* class, extending it by an extra field of type *CurveData*.
- *X_monotone_curve_2* is derived from the basic *BaseTraits::X_monotone_curve_2* class, extending it by an extra field of type *XMonotoneCurveData*.

Note that the *Curve_2* and *X_monotone_curve_2* are not the same, even if the *BaseTraits::Curve_2* and *BaseTraits::X_monotone_curve_2* are (as in the case of the segment-traits class for example). The extended curve types support the additional methods *data()* and *set_data()* for accessing and modifying the data field.

Users can create an extended curve (or an extended *x*-monotone curve) from a basic curve and a curve-data object. When curves are inserted into an arrangement, they may be split, and the decorator handles their data fields automatically:

- When a curve is subdivided into *x*-monotone subcurves, its data field of type *CurveData* is converted to an *XMonotoneCurveData* object *d* using the *Convert* functor. The object *d* is automatically associated with each of the resulting *x*-monotone subcurves.

Note that by default, the *CurveData* type is identical to the *XMonotoneCurveData* type (and the conversion functor *Convert* is trivially defined). Thus, the data field associated with the original curve is just duplicated and stored with the *x*-monotone subcurves.

- When an *x*-monotone curve is split into two, the decorator class automatically copies its data field to both resulting subcurves.
- When intersecting two *x*-monotone curves *c*₁ and *c*₂, the result may include overlapping sections, represented as *x*-monotone curves. In this case the data fields of *c*₁ and *c*₂ are merged into a single *XMonotoneCurveData* object, using the *Merge* functor, which is supplied as a parameter to the traits class-template. The resulting object is assigned to the data field of the overlapping subcurves.
- Merging two *x*-monotone curves is allowed only when (i) the two curves are geometrically mergeable — that is, the base-traits class allows to merge them — and (ii) the two curves store the same data field.

The *Arr_consolidated_curve_data_traits_2*<*BaseTraits*, *Data*> decorator specializes the generic curve-data decorator. It extends the basic *BaseTraits::Curve_2* by a single *Data* field, and the basic *BaseTraits::X_monotone_curve_2* with a *set* of (distinct) data objects. The *Data* type is required to support the equality operator, used to ensure that each set contains only distinct data objects with no duplicates. When a curve with a data field *d* is subdivided into *x*-monotone subcurves, each subcurve is associated with a set $S = \{d\}$. In case of an overlap between two *x*-monotone curves *c*₁ and *c*₂ with associated data sets *S*₁ and *S*₂, respectively, the overlapping subcurve is associated with the consolidated set $S_1 \cup S_2$.

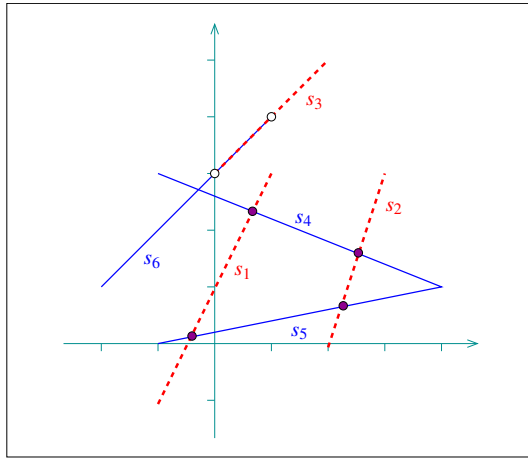


Figure 17.15: An arrangement of six red and blue segments, as constructed in *ex_consolidated_curve_data.C*. Disks correspond to red–blue intersection points, while circles mark the endpoints of red–blue overlaps.

Examples

In the following example, we use *Arr_segment_traits_2* as our base-traits class, attaching an additional *color* field to the segments using the consolidated curve-data traits class. A color may be either *blue* or *red*. Having constructed the arrangement of colored segments, as depicted in Figure 17.15, we detect the vertices that have incident edges mapped to both blue and red segments. Thus, they correspond to red–blue intersection points. We also locate the edge that corresponds to overlaps between red and blue line segments:

```

//! \file examples/Arrangement_2/ex_consolidated_curve_data.C
// Associating a color attribute with segments using the consolidated
// curve-data traits.

#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arr_consolidated_curve_data_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_landmarks_point_location.h>

enum Segment_color
{
    RED,
    BLUE
};

typedef CGAL::Cartesian<Number_type>          Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>     Segment_traits_2;
typedef Segment_traits_2::Curve_2              Segment_2;
typedef CGAL::Arr_consolidated_curve_data_traits_2
    <Segment_traits_2, Segment_color>          Traits_2;
typedef Traits_2::Point_2                      Point_2;
typedef Traits_2::Curve_2                      Colored_segment_2;
typedef CGAL::Arrangement_2<Traits_2>          Arrangement_2;
typedef CGAL::Arr_landmarks_point_location<Arrangement_2> Landmarks_pl;

```

```

int main ()
{
    // Construct an arrangement containing three RED line segments.
    Arrangement_2      arr;
    Landmarks_pl       pl (arr);

    Segment_2          s1 (Point_2(-1, -1), Point_2(1, 3));
    Segment_2          s2 (Point_2(2, 0), Point_2(3, 3));
    Segment_2          s3 (Point_2(0, 3), Point_2(2, 5));

    insert_curve (arr, Colored_segment_2 (s1, RED), pl);
    insert_curve (arr, Colored_segment_2 (s2, RED), pl);
    insert_curve (arr, Colored_segment_2 (s3, RED), pl);

    // Insert three BLUE line segments.
    Segment_2          s4 (Point_2(-1, 3), Point_2(4, 1));
    Segment_2          s5 (Point_2(-1, 0), Point_2(4, 1));
    Segment_2          s6 (Point_2(-2, 1), Point_2(1, 4));

    insert_curve (arr, Colored_segment_2 (s4, BLUE), pl);
    insert_curve (arr, Colored_segment_2 (s5, BLUE), pl);
    insert_curve (arr, Colored_segment_2 (s6, BLUE), pl);

    // Go over all vertices and print just the ones corresponding to intersection
    // points between RED segments and BLUE segments. Note that we skip endpoints
    // of overlapping sections.
    Arrangement_2::Vertex_const_iterator vit;
    Segment_color      color;

    for (vit = arr.vertices_begin(); vit != arr.vertices_end(); ++vit)
    {
        // Go over the incident halfedges of the current vertex and examine their
        // colors.
        bool          has_red = false;
        bool          has_blue = false;

        Arrangement_2::Halfedge_around_vertex_const_circulator eit, first;

        eit = first = vit->incident_halfedges();
        do
        {
            // Get the color of the current half-edge.
            if (eit->curve().data().size() == 1)
            {
                color = eit->curve().data().front();

                if (color == RED)
                    has_red = true;
                else if (color == BLUE)
                    has_blue = true;
            }
            ++eit;

```



```

    } while (eit != first);

    // Print the vertex only if incident RED and BLUE edges were found.
    if (has_red && has_blue)
    {
        std::cout << "Red-blue intersection at (" << vit->point() << ")"
                    << std::endl;
    }
}

// Locate the edges that correspond to a red-blue overlap.
Arrangement_2::Edge_iterator eit;

for (eit = arr.edges_begin(); eit != arr.edges_end(); ++eit)
{
    // Go over the incident edges of the current vertex and examine their
    // colors.
    bool has_red = false;
    bool has_blue = false;

    Traits_2::Data_container::const_iterator dit;

    for (dit = eit->curve().data().begin();
         dit != eit->curve().data().end(); ++dit)
    {
        if (*dit == RED)
            has_red = true;
        else if (*dit == BLUE)
            has_blue = true;
    }

    // Print the edge only if it corresponds to a red-blue overlap.
    if (has_red && has_blue)
    {
        std::cout << "Red-blue overlap at [" << eit->curve() << "]"
                    << std::endl;
    }
}

return (0);
}

```

In the following example, we use *Arr_polyline_traits_2* as our base-traits class, attaching an additional *name* field to each polyline using the generic curve-data traits class. In case of overlaps, we simply concatenate the names of the overlapping polylines. Also notice how we replace the curve associated with the edges that correspond to overlapping polylines with geometrically equivalent curves, but with a different data fields:

```

///! \file examples/Arrangement_2/ex_generic_curve_data.C
// Associating a name attribute with segments using the generic curve-data
// traits.

#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>

```

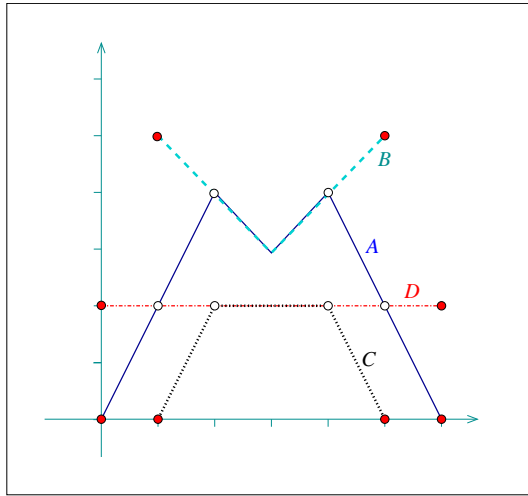


Figure 17.16: An arrangement of four polylines, named A–D, as constructed in *ex_generic_curve_data.C*.

```
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arr_polyline_traits_2.h>
#include <CGAL/Arr_curve_data_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <string>

// Define a functor for concatenating name fields.
typedef std::string Name;

struct Merge_names
{
    Name operator() (const Name& s1, const Name& s2) const
    {
        return (s1 + " " + s2);
    }
};

typedef CGAL::Cartesian<Number_type> Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel> Segment_traits_2;
typedef CGAL::Arr_polyline_traits_2<Segment_traits_2> Polyline_traits_2;
typedef Polyline_traits_2::Curve_2 Polyline_2;
typedef CGAL::Arr_curve_data_traits_2
    <Polyline_traits_2, Name, Merge_names> Traits_2;
typedef Traits_2::Point_2 Point_2;
typedef Traits_2::Curve_2 Curve_2;
typedef Traits_2::X_monotone_curve_2 X_monotone_curve_2;
typedef CGAL::Arrangement_2<Traits_2> Arrangement_2;

int main ()
{
    // Construct an arrangement of four polylines named A--D.
    Arrangement_2 arr;

    Point_2 points1[5] = {Point_2(0,0), Point_2(2,4), Point_2(3,3),
```

```

        Point_2(4,4), Point_2(6,0));
insert_curve (arr, Curve_2 (Polyline_2 (points1, points1 + 5), "A"));

Point_2      points2[3] = {Point_2(1,5), Point_2(3,3), Point_2(5,5)};
insert_curve (arr, Curve_2 (Polyline_2 (points2, points2 + 3), "B"));

Point_2      points3[4] = {Point_2(1,0), Point_2(2,2),
                          Point_2(4,2), Point_2(5,0)};
insert_curve (arr, Curve_2 (Polyline_2 (points3, points3 + 4), "C"));

Point_2      points4[2] = {Point_2(0,2), Point_2(6,2)};
insert_curve (arr, Curve_2 (Polyline_2 (points4, points4 + 2), "D"));

// Print all edges that correspond to an overlapping polyline.
Arrangement_2::Edge_iterator   eit;

for (eit = arr.edges_begin(); eit != arr.edges_end(); ++eit)
{
    if (eit->curve().data().length() > 1)
    {
        std::cout << "[" << eit->curve() << "]"   "
                    << "named: " << eit->curve().data() << std::endl;

        // Rename the curve associated with the edge.
        arr.modify_edge (eit,
            X_monotone_curve_2 (eit->curve(), "overlap"));
    }
}

return (0);
}

```

17.6 The Notification Mechanism

For some applications it is essential to know exactly what happens inside a specific arrangement-instance. For example, when a new curve is inserted into an arrangement, it might be desired to keep track of the faces that are split due to this insertion operation. Other important examples are the point-location strategies that require auxiliary data-structures (see Section 17.3.1), which must be notified on various local changes in the arrangement, in order to keep their data structures up-to-date. The arrangement package offers a mechanism that uses *observers* (see [GHJV95]) that can be attached to an arrangement instance and receive notifications about the changes this arrangement goes through.

The `Arr_observer<Arrangement>` class-template is parameterized with an arrangement class. It stores a pointer to an arrangement object, and is capable of receiving notifications *just before* a structural change occurs in the arrangement and *immediately after* such a change takes place. `Arr_observer` serves as a base class for other observer classes and defines a set of virtual notification functions, with default empty implementations.

The set of functions can be divided into three categories, as follows:

1. Notifiers of changes that affect the entire topological structure of the arrangement. This category consists of two pairs that notify the observer of the following changes:

- The arrangement is cleared.
 - The arrangement is assigned with the contents of another arrangement.
2. Pairs of notifiers of a local change that occurs in the topological structure. Most notifier functions belong to this category. The relevant local changes include:
- A new vertex is constructed and associated with a point.
 - An edge¹⁴ is constructed and associated with an x -monotone curve.
 - An edge is split into two edges.
 - An existing face is split into two faces, as a consequence of the insertion of a new edge.
 - A hole is created in the interior of a face.
 - Two holes are merged to form a single hole, as a consequence of the insertion of a new edge.
 - A hole is moved from one face to another, as a consequence of a face split.
 - Two edges are merged into one edge.
 - Two faces are merged into one face, as a consequence of the removal of an edge that used to separate them.
 - One hole is split into two, as a consequence of the deletion of an edge that used to connect the two components.
 - A vertex is removed.
 - An edge is removed.
 - A hole is deleted from the interior of a face.
3. Notifiers about a change applied by a free (global) function. This category consists of a single pair of notifiers, namely *before_global_change()* and *after_global_change()*. Neither of these functions is invoked by methods of the *Arrangement_2* class. Instead, they are called by the free functions themselves. It is implied that no point-location queries (or any other queries for that matter) are issued between the calls to the notification functions above.

See the Reference Manual for a detailed specification of the *Arr_observer* class along with the exact prototypes of all notification functions.

Each arrangement object stores a (possibly empty) list of pointers to *Arr_observer* objects, and whenever one of the structural changes listed in the first two categories above is about to take place, the arrangement object performs a *forward* traversal on this list and invokes the appropriate function of each observer. After the change takes place the observer list is traversed in a *backward* manner (from tail to head), and the appropriate notification function is invoked for each observer. This allows the nesting of observer objects.

Concrete arrangement-observer classes should inherit from *Arr_observer*. When an observer is constructed, it is attached to a valid arrangement supplied to the observed constructor, or alternatively the observer can be attached to the arrangement at a later time. When this happens, the observer instance inserts itself into the observer list of the associated arrangement and starts receiving notifications whenever this arrangement changes thereafter. Naturally, the observer object unregisters itself by removing itself from this list just before it is destroyed.

The trapezoidal RIC and the landmark point-location strategies both use observers to keep their auxiliary data structures up-to-date. Besides them, users can define their own observer classes, by inheriting from the base observer class and overriding the relevant notification functions, as required by their applications.

The following example shows how to define and use an observer class. The observer in the example keeps track of the arrangement faces, and prints a message whenever a face is split into two due to the insertion of an edge, and whenever two faces merge into one due to the removal of an edge. The layout of the arrangement is depicted in Figure 17.17:

¹⁴The term “edge” refers here to a pair of twin half-edges.

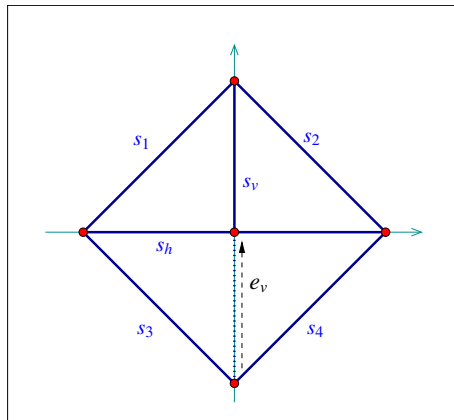


Figure 17.17: An arrangement of five line segments, as constructed in *ex_observer.C*. The halfedge e_v (dashed) is eventually removed, so that the final arrangement consists of four faces (one unbounded and three bounded ones).

```

//! \file examples/Arrangement_2/ex_observer.C
// Using a simple arrangement observer.

#include <CGAL/Cartesian.h>
#include <CGAL/Quotient.h>
#include <CGAL/MP_Float.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_observer.h>

typedef CGAL::Quotient<CGAL::MP_Float>           Number_type;
typedef CGAL::Cartesian<Number_type>             Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>       Traits_2;
typedef Traits_2::Point_2                        Point_2;
typedef Traits_2::X_monotone_curve_2            Segment_2;
typedef CGAL::Arrangement_2<Traits_2>           Arrangement_2;

// An arrangement observer, used to receive notifications of face splits and
// face mergers.
class My_observer : public CGAL::Arr_observer<Arrangement_2>
{
public:

    My_observer (Arrangement_2& arr) :
        CGAL::Arr_observer<Arrangement_2> (arr)
    {}

    virtual void before_split_face (Face_handle,
                                    Halfedge_handle e)
    {
        std::cout << "-> The insertion of : [ " << e->curve()
                    << " ] causes a face to split." << std::endl;
    }

    virtual void before_merge_face (Face_handle,

```

```

                                Face_handle,
                                Halfedge_handle e)
{
    std::cout << "-> The removal of : [ " << e->curve()
                << " ] causes two faces to merge." << std::endl;
}

};

int main ()
{
    // Construct the arrangement containing one diamond-shaped face.
    Arrangement_2 arr;
    My_observer    obs (arr);

    Segment_2      s1 (Point_2(-1, 0), Point_2(0, 1));
    Segment_2      s2 (Point_2(0, 1), Point_2(1, 0));
    Segment_2      s3 (Point_2(1, 0), Point_2(0, -1));
    Segment_2      s4 (Point_2(0, -1), Point_2(-1, 0));

    insert_non_intersecting_curve (arr, s1);
    insert_non_intersecting_curve (arr, s2);
    insert_non_intersecting_curve (arr, s3);
    insert_non_intersecting_curve (arr, s4);

    // Insert a vertical segment dividing the diamond into two, and a
    // a horizontal segment further dividing the diamond into four:
    Segment_2      s_vert (Point_2(0, -1), Point_2(0, 1));
    Arrangement_2::Halfedge_handle
                    e_vert = insert_non_intersecting_curve (arr, s_vert);

    Segment_2      s_horiz (Point_2(-1, 0), Point_2(1, 0));

    insert_curve (arr, s_horiz);

    std::cout << "The initial arrangement size:" << std::endl
                << "   V = " << arr.number_of_vertices()
                << ",   E = " << arr.number_of_edges()
                << ",   F = " << arr.number_of_faces() << std::endl;

    // Now remove a portion of the vertical segment.
    remove_edge (arr, e_vert);

    std::cout << "The final arrangement size:" << std::endl
                << "   V = " << arr.number_of_vertices()
                << ",   E = " << arr.number_of_edges()
                << ",   F = " << arr.number_of_faces() << std::endl;

    return (0);
}

```

Observers are especially useful when the DCEL records are extended and store additional data, as they help updating this data on-line. See Section [17.7](#) for more details and examples.

17.7 Extending the DCEL

For many applications of the arrangement package it is necessary to store additional information (perhaps of non-geometric nature) with the arrangement cells. As vertices are associated with *Point_2* objects and edges (halfedge pairs) are associated with *X_monotone_curve_2* objects, both defined by the traits class, it is possible to extend the traits-class type by using a traits-class decorator, as explained in Section 17.5.6, which may be a sufficient solution for some applications. However, the DCEL faces are not associated with any geometric object, so it is impossible to extend them using a traits-class decorator. Extending the DCEL face records comes handy in such cases. As a matter of fact, it is possible to conveniently extend all DCEL records (namely vertices, halfedges and faces), which can also be advantageous for some applications.

All examples presented so far use the default *Arr_default_dcel<Traits>*. This is done implicitly, as this class serves as a default parameter for the *Arrangement_2* template. The default DCEL class just associates points with vertices and *x*-monotone curves with halfedge, but nothing more. In this section we show how to use alternative DCEL types to extend the desired DCEL records.

17.7.1 Extending the DCEL Faces

The *Arr_face_extended_dcel<Traits, FaceData>* class-template is used to associate auxiliary data field of type *FaceData* to each face record in the DCEL.

When an *Arrangement_2* object is parameterized by this DCEL class, its nested *Face* type is extended with the access function *data()* and with the modifier *set_data()*. Using these extra functions it is straightforward to access and maintain the auxiliary face-data field.

Note that the extra data fields must be maintained by the application programmers. They may choose to construct their arrangement, and only then go over the faces and attach the appropriate data fields to the arrangement faces. However, in some cases the face data can only be computed when the face is created (split from another face, or merged with another face). In such cases one can use an arrangement observer tailored for this task, which receives updates whenever a face is modified and sets its data field accordingly.

The next example constructs an arrangement that contains seven bounded faces induced by six line segments (see Figure 17.18). An observer gets notified each time a new face *f* is created and it associates *f* with a running index, (where the index of the unbounded face is 0). As a result, the faces are numbered according to their creation order, as one can easily verify by examining the insertion order of the segments:¹⁵

```
#!/ \file examples/Arrangement_2/ex_face_extension.C
// Extending the arrangement-face records.

#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_extended_dcel.h>
#include <CGAL/Arr_observer.h>

typedef CGAL::Cartesian<Number_type>           Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>      Traits_2;
typedef Traits_2::Point_2                      Point_2;
```

¹⁵For simplicity, the particular observer used must be attached to an empty arrangement. It is not difficult however to modify the program to handle the general case of attaching a similar observer to a non-empty arrangement.

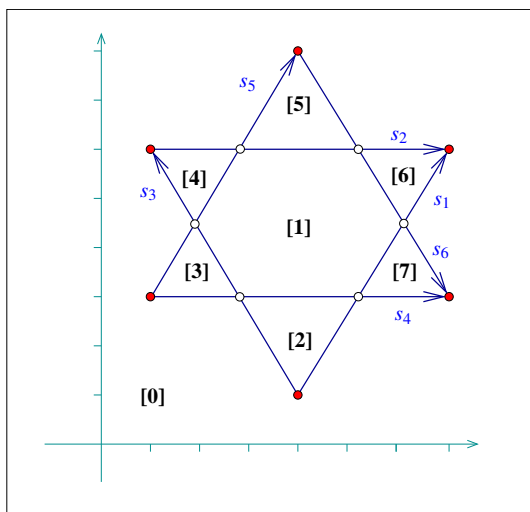


Figure 17.18: An arrangement of six line segments, as constructed in *ex_face_extension.C* and *ex_dcel_extension.C* (in *ex_dcel_extension.C* we treat the segments as directed, so they are drawn as arrows directed from the source to the target). The indices associated with the halfedges in *ex_face_extension.C* are shown in brackets.

```
typedef Traits_2::X_monotone_curve_2      Segment_2;
typedef CGAL::Arr_face_extended_dcel<Traits_2, int> Dcel;
typedef CGAL::Arrangement_2<Traits_2, Dcel> Arrangement_2;

// An arrangement observer, used to receive notifications of face splits and
// to update the indices of the newly created faces.
class Face_index_observer : public CGAL::Arr_observer<Arrangement_2>
{
private:
    int      n_faces;          // The current number of faces.

public:

    Face_index_observer (Arrangement_2& arr) :
        CGAL::Arr_observer<Arrangement_2> (arr),
        n_faces (0)
    {
        CGAL_precondition (arr.is_empty());

        arr.unbounded_face()->set_data (0);
        n_faces++;
    }

    virtual void after_split_face (Face_handle old_face,
                                   Face_handle new_face, bool )
    {
        // Assign index to the new face.
        new_face->set_data (n_faces);
        n_faces++;
    }
};
```



```

int main ()
{
    // Construct the arrangement containing two intersecting triangles.
    Arrangement_2      arr;
    Face_index_observer obs (arr);

    Segment_2      s1 (Point_2(4, 1), Point_2(7, 6));
    Segment_2      s2 (Point_2(1, 6), Point_2(7, 6));
    Segment_2      s3 (Point_2(4, 1), Point_2(1, 6));
    Segment_2      s4 (Point_2(1, 3), Point_2(7, 3));
    Segment_2      s5 (Point_2(1, 3), Point_2(4, 8));
    Segment_2      s6 (Point_2(4, 8), Point_2(7, 3));

    insert_non_intersecting_curve (arr, s1);
    insert_non_intersecting_curve (arr, s2);
    insert_non_intersecting_curve (arr, s3);
    insert_x_monotone_curve (arr, s4);
    insert_x_monotone_curve (arr, s5);
    insert_x_monotone_curve (arr, s6);

    // Go over all arrangement faces and print the index of each face and it
    // outer boundary. The face index is stored in its data field in our case.
    Arrangement_2::Face_const_iterator fit;
    Arrangement_2::Ccb_halfedge_const_circulator curr;

    std::cout << arr.number_of_faces() << " faces:" << std::endl;
    for (fit = arr.faces_begin(); fit != arr.faces_end(); ++fit)
    {
        std::cout << "Face no. " << fit->data() << ": ";
        if (fit->is_unbounded())
        {
            std::cout << "Unbounded." << std::endl;
        }
        else
        {
            curr = fit->outer_ccb();
            std::cout << curr->source()->point();
            do
            {
                std::cout << " --> " << curr->target()->point();
                ++curr;
            } while (curr != fit->outer_ccb());
            std::cout << std::endl;
        }
    }

    return (0);
}

```

17.7.2 Extending All DCEL Records

The `Arr_extended_dcel<Traits, VertexData, HalfedgeData, FaceData>` class-template is used to associate auxiliary data fields of types `VertexData`, `HalfedgeData`, and `FaceData` to each DCEL vertex, halfedge, and face record types, respectively.

When an `Arrangement_2` object is injected with this DCEL class, each one of its nested `Vertex`, `Halfedge` and `Face` classes is extended by the access function `data()` and by the modifier `set_data()`.

The next example shows how to use a DCEL with extended vertex, halfedge, and face records. In this example each vertex is associated with a color, which may be blue, red, or white, depending on whether the vertex is isolated, represents a segment endpoint, or whether it represents an intersection point. Each halfedge is associated with Boolean flag indicating whether its direction is the same as the direction of its associated segment (in this example segments are treated as directed objects). Each face is also extended to store the size of its outer boundary.

The constructed arrangement, depicted in Figure 17.18, is similar to the arrangement constructed in the previous example. Note that all auxiliary data fields are set during the construction phase. Also note that the data fields are properly maintained when the arrangement is copied to another arrangement instance:

```
//! \file examples/Arrangement_2/ex_dcel_extension.C
// Extending all DCEL records (vertices, edges and faces).

#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_extended_dcel.h>

enum Color {BLUE, RED, WHITE};

typedef CGAL::Cartesian<Number_type>           Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>     Traits_2;
typedef Traits_2::Point_2                     Point_2;
typedef Traits_2::X_monotone_curve_2          Segment_2;
typedef CGAL::Arr_extended_dcel<Traits_2,
                                Color, bool, int> Dcel;
typedef CGAL::Arrangement_2<Traits_2, Dcel>     Arrangement_2;

int main ()
{
    // Construct the arrangement containing two intersecting triangles.
    Arrangement_2 arr;

    Segment_2 s1 (Point_2(4, 1), Point_2(7, 6));
    Segment_2 s2 (Point_2(1, 6), Point_2(7, 6));
    Segment_2 s3 (Point_2(4, 1), Point_2(1, 6));
    Segment_2 s4 (Point_2(1, 3), Point_2(7, 3));
    Segment_2 s5 (Point_2(1, 3), Point_2(4, 8));
    Segment_2 s6 (Point_2(4, 8), Point_2(7, 3));

    insert_non_intersecting_curve (arr, s1);
    insert_non_intersecting_curve (arr, s2);
    insert_non_intersecting_curve (arr, s3);
}
```

```

insert_x_monotone_curve (arr, s4);
insert_x_monotone_curve (arr, s5);
insert_x_monotone_curve (arr, s6);

// Go over all arrangement vertices and set their colors according to our
// coloring convention.
Arrangement_2::Vertex_iterator      vit;
unsigned int                        degree;

for (vit = arr.vertices_begin(); vit != arr.vertices_end(); ++vit)
{
    degree = vit->degree();
    if (degree == 0)
        vit->set_data (BLUE);          // Isolated vertex.
    else if (degree <= 2)
        vit->set_data (RED);           // Vertex represents an endpoint.
    else
        vit->set_data (WHITE);         // Vertex represents an intersection point.
}

// Go over all arrangement edges and set their flags.
Arrangement_2::Edge_iterator      eit;
bool                              flag;

for (eit = arr.edges_begin(); eit != arr.edges_end(); ++eit)
{
    // Check if the halfedge has the same direction as its associated
    // segment. Note that its twin always has an opposite direction.
    flag = (eit->source()->point() == eit->curve().source());
    eit->set_data (flag);
    eit->twin()->set_data (!flag);
}

// For each arrangement face, print the outer boundary and its size.
Arrangement_2::Face_iterator      fit;
Arrangement_2::Ccb_halfedge_circulator  curr;
int                                boundary_size;

for (fit = arr.faces_begin(); fit != arr.faces_end(); ++fit)
{
    boundary_size = 0;
    if (! fit->is_unbounded())
    {
        curr = fit->outer_ccb();
        do
        {
            ++boundary_size;
            ++curr;
        } while (curr != fit->outer_ccb());
    }
    fit->set_data (boundary_size);
}

// Copy the arrangement and print the vertices.

```

```

Arrangement_2    arr2 = arr;

std::cout << "The arrangement vertices:" << std::endl;
for (vit = arr2.vertices_begin(); vit != arr2.vertices_end(); ++vit)
{
    std::cout << '(' << vit->point() << ") - ";
    switch (vit->data())
    {
        case BLUE   : std::cout << "BLUE." << std::endl; break;
        case RED    : std::cout << "RED."   << std::endl; break;
        case WHITE  : std::cout << "WHITE." << std::endl; break;
    }
}

return (0);
}

```

— advanced —

The various DCEL classes presented in this section are perfectly sufficient for most applications based on the arrangement package. However, users may also use their own implementation of a DCEL class to instantiate the *Arrangement_2* class-template, in case they need special functionality from their DCEL. Such a class must be a model of the concept *ArrangementDcel*, whose exact specification is listed in the Reference Manual.

— advanced —

17.8 Overlaying Arrangements

Assume that we are given two geographic maps represented as arrangements with some data objects attached to their faces, representing some geographic information — for example, a map of the annual precipitation in some country and a map of the vegetation in the same country. It is interesting to overlay the two maps to locate, for example, the regions where there is a pine forest and the annual precipitation is between 1000 mm and 1500 mm.

Computing the overlay of two planar arrangement is also useful for supporting Boolean set operations on polygons (or generalized polygons, see, e.g., [BEH⁺02]).

The function *overlay* (*arr_a*, *arr_b*, *ovl_arr*, *ovl_traits*) accepts two input arrangement instances *arr_a* and *arr_b*, and constructs their overlay instance *ovl_arr*. All three arrangements must use the same geometric primitives. In other words, their types must be defined using the same geometric traits-class. Let us assume that *arr_a* is of type *Arrangement_2*<*Traits*,*Dcel_A*>, *arr_b* is of type *Arrangement_2*<*Traits*,*Dcel_B*> and the resulting *ovl_arr* is of type *Arrangement_2*<*Traits*,*Dcel_R*>. The *ovl_traits* parameter is an instance of an *overlay traits-class*, which enables the creation of *Dcel_R* records in the overlaid arrangement from the DCEL features of *arr_a* and *arr_b* that they correspond to.

In principle, we distinguish between three levels of overlay:

Simple overlay: An overlay of two arrangements that store no additional data with their DCEL records. That is, they are defined using the default DCEL class *Arr_default_dcel*. Typically, the overlaid arrangement in this case stores no extra data with its DCEL records as well (or if it does, the additional data fields cannot be computed by the overlay operation), so by overlaying the two arrangement we just compute the arrangement of all curves that induce *arr_a* and *arr_b*. Note that the same result can be obtained using the

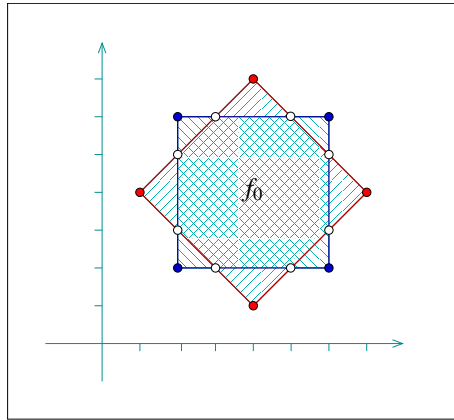


Figure 17.19: Overlaying two simple arrangements of line segments, as done in *ex_overlay.C* and *ex_face_extension_overlay.C*. In *ex_face_extension_overlay.C* the two bounded faces are considered as *marked*, and the octagonal face which is the intersection of the two marked faces is denoted by f_0 .

standard insertion operations, but users may choose to use overlay computation in order to achieve better running times.

The *Arr_default_overlay_traits* class should be used as an overlay traits-class for such simple overlay operations.

Face overlay: An overlay of two arrangements that store additional data fields with their faces (e.g., the geographic-map example given in the beginning of this section). The resulting overlaid arrangement typically also stores extraneous data fields with its faces, where the data field that is attached to an overlaid face can be computed from the data fields of the two faces (in *arr_a* and *arr_b*) that induce the overlaid face.

The *Arr_face_overlay_traits* class should be used as an overlay traits-class for face-overlay operations. It operates on arrangement, whose DCEL representation is based on the *Arr_face_extended_dcel* class-template (see Section 17.7.1). The face-overlay traits-class is parameterized by a functor that is capable of combining two face-data fields of types *Dcel_A::Face_data* and *Dcel_B::Face_data*, and computing the output *Dcel_R::Face_data* object. The overlay traits-class uses this functor to properly construct the overlaid faces.

Full overlay: An overlay of two arrangements that store additional data fields with all their DCEL records. That is, their DCEL classes are instantiations of the *Arr_extended_dcel* class-template (see Section 17.7.2), where the resulting arrangement also extends it DCEL records with data fields computed on the basis of the overlapping DCEL features of the two input arrangements.

In the following subsections we give some examples for the simple and the face-overlay operations and demonstrate how to use the auxiliary overlay traits-classes. For the full overlay operations users need to implement their specialized overlay traits-class, which models the *OverlayTraits* concept. The details of this concept are given in the Reference Manual.

17.8.1 Example for a Simple Overlay

The next program constructs two simple arrangements, as depicted in Figure 17.19 and computes their overlay:

```
///! \file examples/Arrangement_2/ex_overlay.C
```

```

// A simple overlay of two arrangements.

#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_overlay.h>
#include <CGAL/Arr_default_overlay_traits.h>

typedef CGAL::Cartesian<Number_type>           Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>      Traits_2;
typedef Traits_2::Point_2                      Point_2;
typedef Traits_2::X_monotone_curve_2           Segment_2;
typedef CGAL::Arrangement_2<Traits_2>          Arrangement_2;
typedef CGAL::Arr_default_overlay_traits<Arrangement_2> Overlay_traits;

int main ()
{
    // Construct the first arrangement, containing a square-shaped face.
    Arrangement_2      arr1;

    Segment_2          s1 (Point_2(2, 2), Point_2(6, 2));
    Segment_2          s2 (Point_2(6, 2), Point_2(6, 6));
    Segment_2          s3 (Point_2(6, 6), Point_2(2, 6));
    Segment_2          s4 (Point_2(2, 6), Point_2(2, 2));

    insert_non_intersecting_curve (arr1, s1);
    insert_non_intersecting_curve (arr1, s2);
    insert_non_intersecting_curve (arr1, s3);
    insert_non_intersecting_curve (arr1, s4);

    // Construct the second arrangement, containing a rhombus-shaped face.
    Arrangement_2      arr2;

    Segment_2          t1 (Point_2(4, 1), Point_2(7, 4));
    Segment_2          t2 (Point_2(7, 4), Point_2(4, 7));
    Segment_2          t3 (Point_2(4, 7), Point_2(1, 4));
    Segment_2          t4 (Point_2(1, 4), Point_2(4, 1));

    insert_non_intersecting_curve (arr2, t1);
    insert_non_intersecting_curve (arr2, t2);
    insert_non_intersecting_curve (arr2, t3);
    insert_non_intersecting_curve (arr2, t4);

    // Compute the overlay of the two arrangements.
    Arrangement_2      overlay_arr;
    Overlay_traits      overlay_traits;

    overlay (arr1, arr2, overlay_arr, overlay_traits);

    // Print the size of the overlaid arrangement.
    std::cout << "The overlaid arrangement size:" << std::endl
                << "    V = " << overlay_arr.number_of_vertices()
                << ",    E = " << overlay_arr.number_of_edges()

```

```

        << ", F = " << overlay_arr.number_of_faces() << std::endl;

    return (0);
}

```

17.8.2 Example for a Face Overlay

The following example shows how to compute the intersection of two polygons using the *overlay()* function. It uses a face-extended DCEL class to define our arrangement class. The DCEL extends each face with a Boolean flag. A polygon is represented as a *marked* arrangement face, (whose flag is set). The example uses a face-overlay traits class, instantiated with a functor that simply performs a logical *and* operations on Boolean flags. As a result, a face in the overlaid arrangement is marked only when it corresponds to an overlapping region of two marked cells in the input arrangements. Namely, it is part of the intersection of the two polygons.

The example computes the intersection between a square and a rhombus, (which is actually also a square). The resulting polygon is an octagon, denoted by f_0 in Figure 17.19:

```

//! \file examples/Arrangement_2/ex_face_extension_overlay.C
// A face overlay of two arrangement with extended face records.

#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Arr_extended_dcel.h>
#include <CGAL/Arr_overlay.h>
#include <CGAL/Arr_default_overlay_traits.h>

typedef CGAL::Cartesian<Number_type>           Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>      Traits_2;
typedef Traits_2::Point_2                      Point_2;
typedef Traits_2::X_monotone_curve_2           Segment_2;
typedef CGAL::Arr_face_extended_dcel<Traits_2, bool> Dcel;
typedef CGAL::Arrangement_2<Traits_2, Dcel>      Arrangement_2;
typedef CGAL::Arr_face_overlay_traits<Arrangement_2,
                                     Arrangement_2,
                                     Arrangement_2,
                                     std::logical_and<bool> > Overlay_traits;

int main ()
{
    // Construct the first arrangement, containing a square-shaped face.
    Arrangement_2 arr1;

    Segment_2 s1 (Point_2(2, 2), Point_2(6, 2));
    Segment_2 s2 (Point_2(6, 2), Point_2(6, 6));
    Segment_2 s3 (Point_2(6, 6), Point_2(2, 6));
    Segment_2 s4 (Point_2(2, 6), Point_2(2, 2));

    insert_non_intersecting_curve (arr1, s1);
    insert_non_intersecting_curve (arr1, s2);
    insert_non_intersecting_curve (arr1, s3);

```

```

insert_non_intersecting_curve (arr1, s4);

// Mark just the bounded face.
Arrangement_2::Face_iterator fit;

CGAL_assertion (arr1.number_of_faces() == 2);
for (fit = arr1.faces_begin(); fit != arr1.faces_end(); ++fit)
    fit->set_data (fit != arr1.unbounded_face());

// Construct the second arrangement, containing a rhombus-shaped face.
Arrangement_2 arr2;

Segment_2 t1 (Point_2(4, 1), Point_2(7, 4));
Segment_2 t2 (Point_2(7, 4), Point_2(4, 7));
Segment_2 t3 (Point_2(4, 7), Point_2(1, 4));
Segment_2 t4 (Point_2(1, 4), Point_2(4, 1));

insert_non_intersecting_curve (arr2, t1);
insert_non_intersecting_curve (arr2, t2);
insert_non_intersecting_curve (arr2, t3);
insert_non_intersecting_curve (arr2, t4);

// Mark just the bounded face.
CGAL_assertion (arr2.number_of_faces() == 2);
for (fit = arr2.faces_begin(); fit != arr2.faces_end(); ++fit)
    fit->set_data (fit != arr2.unbounded_face());

// Compute the overlay of the two arrangements, marking only the faces that
// are intersections of two marked faces in arr1 and arr2, respectively.
Arrangement_2 overlay_arr;
Overlay_traits overlay_traits;

overlay (arr1, arr2, overlay_arr, overlay_traits);

// Go over the faces of the overlaid arrangement and print just the marked
// ones.
Arrangement_2::Ccb_halfedge_circulator curr;

std::cout << "The union is: ";
for (fit = overlay_arr.faces_begin(); fit != overlay_arr.faces_end(); ++fit)
{
    if (! fit->data())
        continue;

    curr = fit->outer_ccb();
    std::cout << curr->source()->point();
    do
    {
        std::cout << " --> " << curr->target()->point();
        ++curr;
    } while (curr != fit->outer_ccb());
    std::cout << std::endl;
}

```



```

    return (0);
}

```

17.9 Storing the Curve History

As stated at the beginning of this chapter (Section 17.1), when one constructs an arrangement induced by a set C of arbitrary planar curves, she or he constructs a collection C'' of x -monotone subcurves of C that are pairwise disjoint in their interior, and these subcurves are associated with the arrangement edges (more precisely, with the DCEL halfedges). Doing so, the connection between the originating input curves and the arrangement edges is lost. This loss might be acceptable for some applications. However, in many practical cases it is important to determine the input curves that give rise to the final subcurves.

The *Arrangement_with_history_2*<*Traits*,*Dcel*> class-template extends the *Arrangement_2* class by keeping an additional container of input curves representing C , and by maintaining a cross-mapping between these curves and the arrangement edges they induce. The traits class that is used for instantiating the template should be a model of the *ArrangementTraits_2* concept (see Section 17.4.1). That is, it should define the *Curve_2* type (and not just the *X_monotone_curve_2* type). The *Dcel* parameter should model the *ArrangementDcel* concept. Users can use the default DCEL class or an extended DCEL class according to their needs.

17.9.1 Traversing an Arrangement with History

The *Arrangement_with_history_2* class extends the *Arrangement_2* class, thus all the iterator and circulator types that are defined by the arrangement class are also available in *Arrangement_with_history_2*. The reader is referred to Section 17.2.2 for a comprehensive review of these functions.

As mentioned above, the *Arrangement_with_history_2* class maintains a container of input curves, which can be accessed using curve handles. The member function *number_of_curves()* returns the number of input curves stored in the container, while *curves_begin()* and *curves_end()* return *Arrangement_with_history_2::Curve_iterator* objects that define the valid range of curves that induce the arrangement. The value type of this iterator is *Curve_2*. Moreover, the curve-iterator type is equivalent to *Arrangement_with_history_2::Curve_handle*, which is used for accessing the stored curves. Conveniently, the corresponding constant-iterator and constant-handle types are also defined.

As mentioned in the previous paragraph, a *Curve_handle* object *ch* serves as a pointer to a curve stored in an arrangement-with-history instance *arr*. Using this handle, it is possible to obtain the number of arrangement edges this curve induces by calling *arr.number_of_induced_edges(ch)*. The functions *arr.induced_edges_begin(ch)* and *arr.induced_edges_end(ch)* return iterators of type *Arrangement_with_history_2::Induced_edges_iterator* that define the valid range of edges induced by *ch*. The value type of these iterators is *Halfedge_handle*. It is thus possible to traverse all arrangement edges induced by an input curve.

It is also important to be able to perform the inverse mapping. Given an arrangement edge, we would like to be able to determine which input curve induces it. In case the edge represents an overlap of several curves, we should be able to trace all input curves that overlap over this edge. The *Arrangement_with_history_2* class is extended by several member functions that enable such an inverse mapping. Given a halfedge handle *e* in an arrangement with history *arr*, then *arr.number_of_originating_curves(e)* returns the number of curves that induce the edge (which should be 1 in non-degenerate cases, and 2 or more in case of overlaps), while *arr.originating_curves_begin(e)* and *arr.originating_curves_end(e)* return *Arrangement_with_history_2::Originating_curve_iterator* objects that define the range of curves that induce *e*. The value type of these iterator is *Curve_2*.

It is possible to overlay two *Arrangement_with_history_2* instances instantiated by the same traits class. In this case, the resulting arrangement will store a consolidated container of input curves, and automatically preserve the cross-mapping between the arrangement edges and the consolidated curve set. Users can employ an overlay-traits class to maintain any type of auxiliary data stored with the DCEL features (see Section 17.8).

17.9.2 Modifying an Arrangement with History

As the *Arrangement_with_history_2* class extends the *Arrangement_2* class, it inherits the fundamental modification operations, such as *assign()* and *clear()*, from it. The vertex-manipulation functions are also inherited and supported (see Sections 17.2.3 and 17.4.1 for the details). However, there are some fundamental differences between the interfaces of the two classes, which we highlight in this subsection.

The most significant difference between the arrangement-with-history class and the basic arrangement class is the way they handle their input curves. *Arrangement_with_history_2* always stores the *Curve_2* objects that induce it, thus it is impossible to insert *x*-monotone curves into an arrangement with history. The free *insert_non_intersecting_curve()* and *insert_x_monotone_curve()* (as well as their aggregated versions) are therefore not available for arrangement-with-history instances and only the free *insert_curve()* and *insert_curves()* functions (the incremental insertion function and the aggregated insertion function) are supported — see also Section 17.4.1. Notice however that while the incremental insertion function *insert_curve(arr,c)* for an *Arrangement_2* object *arr* does not have a return value, the corresponding arrangement-with-history function returns a *Curve_handle* to the inserted curve.

As we are able to keep track of all edges induced by an input curve, we also provide a free function that removes a curve from an arrangement. By calling *remove(arr,ch)*, where *ch* is a valid curve handle, the given curve is deleted from the curve container, and all edges induced solely by this curve (i.e., excluding overlapping edges) are removed from the arrangement. The function returns the number of edges that have been removed.

In some cases, users may need to operate directly on the arrangement edges. We first mention that the specialized insertion functions (see Section 17.2.3) are not supported, as they accept *x*-monotone curves. Insertion can only be performed via the free insertion-functions. The other edge-manipulation functions (see Section 17.2.3) are however available, but have a different interface that does not use *x*-monotone curves:

- Invoking *split_edge(e,p)* splits the edge *e* at a given point *p* that lies in its interior.
- Invoking *merge_edge(e1,e2)* merges the two given edges. There is a precondition that *e1* and *e2* shared a common end-vertex of degree 2, and that the *x*-monotone subcurves associated with these edges are mergeable.
- It is possible to remove an edge by simply invoking *remove_edge(e)*.

In all cases, the maintenance of cross-pointers for the appropriate input curves will be done automatically.

It should be noted that it is possible to attach observers to an arrangement-with-history instance in order to get detailed notifications of the changes the arrangements undergoes (see Section 17.6 for the details).

17.9.3 Examples

In the following example we construct a simple arrangement of six line segments, as depicted in Figure 17.20, while maintaining the curve history. The example demonstrates the usage of the special traversal functions. It also shows how to issue point-location queries on the resulting arrangement, using the auxiliary function *point_location_query()* defined in the header file *point_location_utils.h* (see also Section 17.3.1).

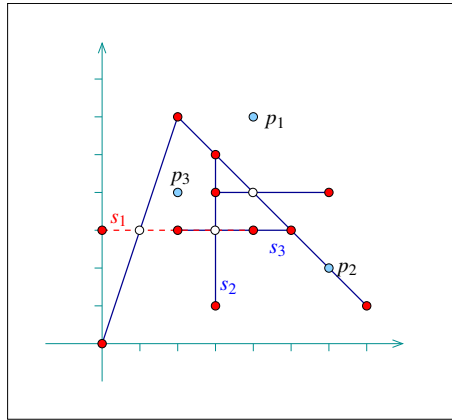


Figure 17.20: An arrangement with history as constructed in *ex_curve_history.C*. Note that s_1 and s_3 overlap over two edges. The point-location query points are drawn as lightly shaded dots.

```
// file: examples/Arrangement_2/ex_curve_history.C
// Constructing an arrangement with curve history.

#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_with_history_2.h>
#include <CGAL/Arr_trapezoid_ric_point_location.h>

#include "point_location_utils.h"

typedef CGAL::Cartesian<Number_type>           Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>      Traits_2;
typedef Traits_2::Point_2                      Point_2;
typedef Traits_2::Curve_2                     Segment_2;
typedef CGAL::Arrangement_with_history_2<Traits_2> Arr_with_hist_2;
typedef Arr_with_hist_2::Curve_handle          Curve_handle;
typedef CGAL::Arr_trapezoid_ric_point_location<Arr_with_hist_2> Point_location;

int main ()
{
    Arr_with_hist_2  arr;

    // Insert s1, s2 and s3 incrementally:
    Segment_2        s1 (Point_2 (0, 3), Point_2 (4, 3));
    Curve_handle      c1 = insert_curve (arr, s1);
    Segment_2        s2 (Point_2 (3, 2), Point_2 (3, 5));
    Curve_handle      c2 = insert_curve (arr, s2);
    Segment_2        s3 (Point_2 (2, 3), Point_2 (5, 3));
    Curve_handle      c3 = insert_curve (arr, s3);

    // Insert three additional segments aggregately:
    Segment_2        segs[3];
    segs[0] = Segment_2 (Point_2 (2, 6), Point_2 (7, 1));
    segs[1] = Segment_2 (Point_2 (0, 0), Point_2 (2, 6));
```

```

segs[2] = Segment_2 (Point_2 (3, 4), Point_2 (6, 4));
insert_curves (arr, segs, segs + 3);

// Print out the curves and the number of edges each one induces.
Arr_with_hist_2::Curve_iterator      cit;

std::cout << "The arrangement contains "
            << arr.number_of_curves() << " curves:" << std::endl;
for (cit = arr.curves_begin(); cit != arr.curves_end(); ++cit)
{
    std::cout << "Curve [" << *cit << "]" induces "
              << arr.number_of_induced_edges(cit) << " edges." << std::endl;
}

// Print the arrangement edges, along with the list of curves that
// induce each edge.
Arr_with_hist_2::Edge_iterator      eit;
Arr_with_hist_2::Originating_curve_iterator  ocit;

std::cout << "The arrangement is comprised of "
            << arr.number_of_edges() << " edges:" << std::endl;
for (eit = arr.edges_begin(); eit != arr.edges_end(); ++eit)
{
    std::cout << "[" << eit->curve() << "]. Originating curves: ";
    for (ocit = arr.originating_curves_begin (eit);
         ocit != arr.originating_curves_end (eit); ++ocit)
    {
        std::cout << " [" << *ocit << "]" << std::flush;
    }
    std::cout << std::endl;
}

// Perform some point-location queries:
Point_location  pl (arr);

Point_2        p1 (4, 6);
point_location_query (pl, p1);
Point_2        p2 (6, 2);
point_location_query (pl, p2);
Point_2        p3 (2, 4);
point_location_query (pl, p3);

return (0);
}

```

The following example demonstrates the usage of the free *remove()* function. We construct an arrangement of nine circles, while keeping a handle to each inserted circle. We then remove the large circle C_0 , which induces 18 edges, as depicted in Figure 17.21. The example also shows how to use the *split_edge()* and *merge_edge()* functions when operating on an arrangement-with-history instance:

```

///! \file examples/Arrangement_2/ex_edge_manipulation_curve_history.C
// Removing curves and manipulating edges in an arrangement with history.

```

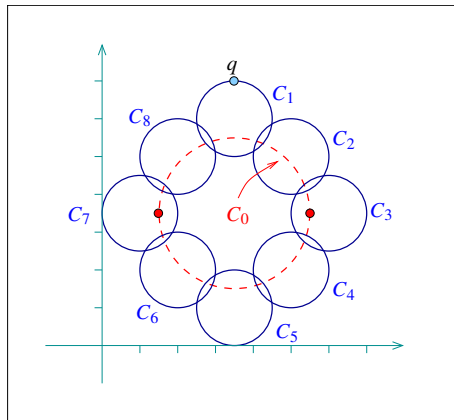


Figure 17.21: An arrangement with history of nine circle as constructed in *ex_edge_manipulation_curve_history.C*. Note the vertical tangency points of C_0 , marked as dark dots, which subdivide this circle into an upper half and a lower half, each consists of 9 edges. The large circle C_0 is eventually removed from the arrangement, with all 18 edges it induces.

```
#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_circle_segment_traits_2.h>
#include <CGAL/Arrangement_with_history_2.h>

typedef CGAL::Cartesian<Number_type>          Kernel;
typedef Kernel::Point_2                       Rat_point_2;
typedef Kernel::Circle_2                     Circle_2;
typedef CGAL::Arr_circle_segment_traits_2<Kernel> Traits_2;
typedef Traits_2::Point_2                    Point_2;
typedef Traits_2::Curve_2                    Curve_2;
typedef CGAL::Arrangement_with_history_2<Traits_2> Arr_with_hist_2;
typedef Arr_with_hist_2::Curve_handle        Curve_handle;
typedef CGAL::Arr_walk_along_line_point_location<Arr_with_hist_2> Point_location;

int main ()
{
    // Construct an arrangement containing nine circles: C[0] of radius 2 and
    // C[1], ..., C[8] of radius 1.
    const Number_type _7_halves = Number_type (7, 2);
    Arr_with_hist_2   arr;
    Curve_2           C[9];
    Curve_handle      handles[9];
    int               k;

    C[0] = Circle_2 (Rat_point_2 (_7_halves, _7_halves), 4, CGAL::CLOCKWISE);
    C[1] = Circle_2 (Rat_point_2 (_7_halves, 6), 1, CGAL::CLOCKWISE);
    C[2] = Circle_2 (Rat_point_2 (5, 6), 1, CGAL::CLOCKWISE);
    C[3] = Circle_2 (Rat_point_2 (6, _7_halves), 1, CGAL::CLOCKWISE);
    C[4] = Circle_2 (Rat_point_2 (5, 2), 1, CGAL::CLOCKWISE);
    C[5] = Circle_2 (Rat_point_2 (_7_halves, 1), 1, CGAL::CLOCKWISE);
    C[6] = Circle_2 (Rat_point_2 (2, 2), 1, CGAL::CLOCKWISE);
```

```

C[7] = Circle_2 (Rat_point_2 (1, _7_halves), 1, CGAL::CLOCKWISE);
C[8] = Circle_2 (Rat_point_2 (2, 5), 1, CGAL::CLOCKWISE);

for (k = 0; k < 9; k++)
    handles[k] = insert_curve (arr, C[k]);

std::cout << "The initial arrangement size:" << std::endl
    << "    V = " << arr.number_of_vertices()
    << ",    E = " << arr.number_of_edges()
    << ",    F = " << arr.number_of_faces() << std::endl;

// Remove the large circle C[0].
std::cout << "Removing C[0] : ";
std::cout << remove_curve (arr, handles[0])
    << " edges have been removed." << std::endl;

std::cout << "The arrangement size:" << std::endl
    << "    V = " << arr.number_of_vertices()
    << ",    E = " << arr.number_of_edges()
    << ",    F = " << arr.number_of_faces() << std::endl;

// Locate the point q, which should be on an edge e.
Point_location                pl (arr);
const Point_2                 q = Point_2 (_7_halves, 7);
CGAL::Object                  obj = pl.locate (q);
Arr_with_hist_2::Halfedge_const_handle e;
bool                           success = CGAL::assign (e, obj);

CGAL_assertion (success);

// Split the edge e to two edges e1 and e2;
Arr_with_hist_2::Halfedge_handle e1, e2;

e1 = arr.split_edge (arr.non_const_handle (e), q);
e2 = e1->next();

std::cout << "After edge split: "
    << "V = " << arr.number_of_vertices()
    << ",    E = " << arr.number_of_edges()
    << ",    F = " << arr.number_of_faces() << std::endl;

// Merge back the two split edges.
arr.merge_edge (e1, e2);

std::cout << "After edge merge: "
    << "V = " << arr.number_of_vertices()
    << ",    E = " << arr.number_of_edges()
    << ",    F = " << arr.number_of_faces() << std::endl;

return (0);
}

```

17.10 Input/Output Functions

In some cases, we would like to reuse an arrangement instance constructed by our application in the future — for example, our arrangement may represent a very complicated geographical map and we have various applications that need to answer point-location queries on this map. Naturally, we can store the set of curves that induces the arrangement, but this implies that we need to construct the arrangement from scratch each time we need to reuse it. A more efficient solution would be to save the arrangement to a file, so that other application can reread it from there.

We provide an *inserter* (the `<<` operator) and an *extractor* (the `>>` operator) for the `Arrangement_2<Traits,Dcel>` class, such that an arrangement instance can be inserted into an output stream or read from an input stream. The arrangement is written using a simple predefined textual format that encodes the arrangement topology, as well as all geometric entities associated with vertices and edges.

To use the input/output operators, we require that the `Point_2` type and the `X_monotone_curve_2` type defined by the traits class both support the `<<` and `>>` operators. The `Arr_conic_traits_2` class (see Section 17.5.4) and the `Arr_rational_arc_traits_2` class (see Section 17.5.5) currently do not provide these operator for the geometric types they define, so only arrangements of line segments or of polylines can be written or read.

The following example constructs the arrangement depicted in Figure 17.7 and writes it to an output file. It also demonstrates how to re-read the arrangement from a file:

```
#!/ \file examples/Arrangement_2/ex_io.C
// Using the arrangement I/O operators.

#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/IO/Arr_iostream.h>
#include <fstream>

#include "point_location_utils.h"

typedef CGAL::Cartesian<Number_type>           Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>     Traits_2;
typedef CGAL::Arrangement_2<Traits_2>         Arrangement_2;

int main ()
{
    // Construct the arrangement.
    Arrangement_2 arr;

    construct_segments_arr (arr);

    std::cout << "Writing an arrangement of size:" << std::endl
              << "   V = " << arr.number_of_vertices()
              << ",   E = " << arr.number_of_edges()
              << ",   F = " << arr.number_of_faces() << std::endl;

    // Write the arrangement to a file.
    std::ofstream out_file ("arr_ex_io.dat");
```

```

out_file << arr;
out_file.close();

// Read the arrangement from the file.
Arrangement_2 arr2;
std::ifstream in_file ("arr_ex_io.dat");

in_file >> arr2;
in_file.close();

std::cout << "Read an arrangement of size:" << std::endl
          << "   V = " << arr2.number_of_vertices()
          << ",   E = " << arr2.number_of_edges()
          << ",   F = " << arr2.number_of_faces() << std::endl;

return (0);
}

```

advanced

17.10.1 Arrangements with Auxiliary Data

The inserter and extractor both ignore any auxiliary data stored with the arrangement features, thus they are ideal for arrangements instantiated using the *Arr_default_dcel* class. However, as explained in Section 17.7, one can easily extend the arrangement faces by using the *Arr_face_extended_dcel* template, or extend all DCEL records by using the *Arr_extended_dcel* template. In such cases, it may be crucial that the auxiliary data fields are written to the file or read from there.

The arrangement package includes the free functions *write(arr, os, formatter)*, which writes the arrangement *arr* to an output stream *os*, and *read(arr, os, formatter)*, which reads the arrangement *arr* from an input stream *is*. Both operations are performed using a *formatter* object, which defines the I/O format. The package contains three formatter classes:

- *Arr_text_formatter<Arrangement>* defines a simple textual I/O format for the arrangement topology and geometry, disregarding any auxiliary data that may be associated with the arrangement features. This is the default formatter used by the arrangement inserter and the arrangement extractor, as defined above.
- *Arr_face_extended_text_formatter<Arrangement>* operates on arrangements whose DCEL representation is based on the *Arr_face_extended_dcel<Traits, FaceData>* class (see Section 17.7.1). It supports reading and writing the auxiliary data objects stored with the arrangement faces provided that the *FaceData* class supports an inserter and an extractor.
- *Arr_extended_dcel_text_formatter<Arrangement>* operates on arrangements whose DCEL representation is based on the *Arr_extended_dcel<Traits, VertexData, HalfedgeData, FaceData>* class (see Section 17.7.2). It supports reading and writing the auxiliary data objects stored with the arrangement vertices, edges and faces, provided that the *VertexData*, *HalfedgeData* and *FaceData* classed all have inserters and extractors.

The following example constructs the same arrangement as the example *ex_dcel_extension* does (see Section 17.7.2) which is depicted in Figure 17.18, and writes it to an output file. It also demonstrates how to re-read the arrangement from a file:


```

///! \file examples/Arrangement_2/ex_dcel_extension_io.C
// Using the I/O operators for arrangements with extended DCEL records.

#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arr_extended_dcel.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/IO/Arr_text_formatter.h>
#include <CGAL/IO/Arr_iostream.h>
#include <fstream>

enum Color {BLUE, RED, WHITE};

std::ostream& operator<< (std::ostream& os, const Color& color)
{
    switch (color)
    {
        case BLUE: os << "BLUE"; break;
        case RED: os << "RED"; break;
        case WHITE: os << "WHITE"; break;
        default: os << "ERROR!";
    }
    return (os);
}

std::istream& operator>> (std::istream& is, Color& color)
{
    std::string str;
    is >> str;

    if (str == "BLUE")
        color = BLUE;
    else if (str == "RED")
        color = RED;
    else if (str == "WHITE")
        color = WHITE;

    return (is);
}

typedef CGAL::Cartesian<Number_type> Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel> Traits_2;
typedef Traits_2::Point_2 Point_2;
typedef Traits_2::X_monotone_curve_2 Segment_2;
typedef CGAL::Arr_extended_dcel<Traits_2,
                                Color, bool, int> Dcel;
typedef CGAL::Arrangement_2<Traits_2, Dcel> Arrangement_2;
typedef CGAL::Arr_extended_dcel_text_formatter<Arrangement_2> Formatter;

int main ()
{
    // Construct the arrangement containing two intersecting triangles.
    Arrangement_2 arr;

```

```

Segment_2      s1 (Point_2(4, 1), Point_2(7, 6));
Segment_2      s2 (Point_2(1, 6), Point_2(7, 6));
Segment_2      s3 (Point_2(4, 1), Point_2(1, 6));
Segment_2      s4 (Point_2(1, 3), Point_2(7, 3));
Segment_2      s5 (Point_2(1, 3), Point_2(4, 8));
Segment_2      s6 (Point_2(4, 8), Point_2(7, 3));

insert_non_intersecting_curve (arr, s1);
insert_non_intersecting_curve (arr, s2);
insert_non_intersecting_curve (arr, s3);
insert_x_monotone_curve (arr, s4);
insert_x_monotone_curve (arr, s5);
insert_x_monotone_curve (arr, s6);

// Go over all arrangement vertices and set their colors.
Arrangement_2::Vertex_iterator      vit;
unsigned int                        degree;

for (vit = arr.vertices_begin(); vit != arr.vertices_end(); ++vit)
{
    degree = vit->degree();
    if (degree == 0)
        vit->set_data (BLUE);          // Isolated vertex.
    else if (degree <= 2)
        vit->set_data (RED);           // Vertex represents an endpoint.
    else
        vit->set_data (WHITE);         // Vertex represents an intersection point.
}

// Go over all arrangement edges and set their flags.
Arrangement_2::Edge_iterator      eit;
bool                              flag;

for (eit = arr.edges_begin(); eit != arr.edges_end(); ++eit)
{
    // Check if the halfedge has the same direction as its associated
    // segment. Note that its twin always has an opposite direction.
    flag = (eit->source()->point() == eit->curve().source());
    eit->set_data (flag);
    eit->twin()->set_data (!flag);
}

// Go over all arrangement faces and print their outer boundary and indices.
Arrangement_2::Face_iterator      fit;
Arrangement_2::Ccb_halfedge_circulator      curr;
int                              boundary_size;

for (fit = arr.faces_begin(); fit != arr.faces_end(); ++fit)
{
    boundary_size = 0;
    if (! fit->is_unbounded())
    {
        curr = fit->outer_ccb();
    }
}

```

```

    do
    {
        ++boundary_size;
        ++curr;
    } while (curr != fit->outer_ccb());
    }
    fit->set_data (boundary_size);
}

// Write the arrangement to a file.
std::ofstream    out_file ("arr_ex_dcel_io.dat");
Formatter        formatter;

write (arr, out_file, formatter);
out_file.close();

// Read the arrangement from the file.
Arrangement_2    arr2;
std::ifstream    in_file ("arr_ex_dcel_io.dat");

read (arr2, in_file, formatter);
in_file.close();

std::cout << "The arrangement vertices: " << std::endl;
for (vit = arr2.vertices_begin(); vit != arr2.vertices_end(); ++vit)
    std::cout << '(' << vit->point() << ") - " << vit->data() << std::endl;

return (0);
}

```

External users may write their own formatter classes by implementing models to the *ArrangementInputFormatter* and the *ArrangementOutputFormatter*, as defined in the Reference Manual. Doing so, they can define other I/O formats, such as an XML-based format or a binary format.

advanced

17.10.2 Arrangements with Curve History

In Section 17.9 we introduced the *Arrangement_with_history_2<Traits,Dcel>* class that stores the set of curves inducing the arrangement and maintains the relations between these curves and the edges they induce. Naturally, when reading or writing an arrangement-with-history instance we would like this information to be written to the output stream or retrieved from the input stream alongside with the basic arrangement structure.

The arrangement package supplies an inserter and an extractor for the *Arrangement_with_history_2<Traits,Dcel>* class. The arrangement is represented using a simple predefined textual format, where we require that the *Curve_2* type defined by the traits class — as well as the *Point_2* type and the *X_monotone_curve_2* types — support the << and >> operators.

The following example constructs the same arrangement as example *ex_curve_history* does (see Section 17.9.3) which is depicted in Figure 17.20, and writes it to an output file. It also demonstrates how to re-read the arrangement-with-history from a file:

```

/// \file examples/Arrangement_2/ex_io_curve_history.C
// Using the arrangement-with-history I/O operators.

#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_with_history_2.h>
#include <CGAL/IO/Arr_with_history_iostream.h>
#include <fstream>

typedef CGAL::Cartesian<Number_type>           Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>     Traits_2;
typedef Traits_2::Point_2                     Point_2;
typedef Traits_2::Curve_2                     Segment_2;
typedef CGAL::Arrangement_with_history_2<Traits_2> Arr_with_hist_2;

int main ()
{
    Arr_with_hist_2  arr;

    // Insert six additional segments aggregately:
    Segment_2        segs[6];
    segs[0] = Segment_2 (Point_2 (2, 6), Point_2 (7, 1));
    segs[1] = Segment_2 (Point_2 (3, 2), Point_2 (3, 5));
    segs[2] = Segment_2 (Point_2 (2, 3), Point_2 (5, 3));
    segs[3] = Segment_2 (Point_2 (2, 6), Point_2 (7, 1));
    segs[4] = Segment_2 (Point_2 (0, 0), Point_2 (2, 6));
    segs[5] = Segment_2 (Point_2 (3, 4), Point_2 (6, 4));
    insert_curves (arr, segs, segs + 6);

    std::cout << "Writing an arrangement of "
                << arr.number_of_curves() << " input segments:" << std::endl
                << "    V = " << arr.number_of_vertices()
                << ",    E = " << arr.number_of_edges()
                << ",    F = " << arr.number_of_faces() << std::endl;

    // Write the arrangement to a file.
    std::ofstream      out_file ("arr_ex_io_hist.dat");

    out_file << arr;
    out_file.close();

    // Read the arrangement from the file.
    Arr_with_hist_2    arr2;
    std::ifstream      in_file ("arr_ex_io_hist.dat");

    in_file >> arr2;
    in_file.close();

    std::cout << "Read an arrangement of "
                << arr2.number_of_curves() << " input segments:" << std::endl
                << "    V = " << arr2.number_of_vertices()
                << ",    E = " << arr2.number_of_edges()
                << ",    F = " << arr2.number_of_faces() << std::endl;

```

```

    return (0);
}

```

— *advanced* —

The arrangement package also includes the free functions *write(arr, os, formatter)* and *read(arr, os, formatter)* that operate on a given arrangement-with-history instance *arr*. Both functions are parameterized by a *formatter* object, which define the I/O format. The package contains a template called, *Arr_with_hist_text_formatter<ArrangementFormatter>*, which extends an arrangement formatter class (see Section 17.10.1) and defines a simple textual input/output format.

— *advanced* —

17.11 Adapting to BOOST Graphs

BOOST¹⁶ is a collection of portable C++ libraries that extend the Standard Template Library (STL). The BOOST Graph Library (BGL), which one of the libraries in the collection, offers an extensive set of generic graph algorithms parameterized through templates. As our arrangements are embedded as planar graphs, it is only natural to extend the underlying data structure with the interface that the BGL expects, and gain the ability to perform the operations that the BGL supports, such as shortest-path computation. This section describes how apply the graph algorithms implemented in the BGL to *Arrangement_2* instances.

An instance of *Arrangement_2* is adapted to a BOOST graph through the provision of a set of free functions that operate on the arrangement features and conform with the relevant BGL concepts. Besides the straightforward adaptation, which associates a vertex with each DCEL vertex and an edge with each DCEL halfedge, the package also offer a *dual* adaptor, which associates a graph vertex with each DCEL face, such that two vertices are connected, iff there is a DCEL halfedge that connects the two corresponding faces.

17.11.1 The Primal Arrangement Representation

Arrangement instances are adapted to BOOST graphs by specializing the *boost::graph_traits* template for *Arrangement_2* instances. The graph-traits states the graph concepts that the arrangement class models (see below) and defines the types required by these concepts.

In this specialization the *Arrangement_2* vertices correspond to the graph vertices, where two vertices are adjacent if there is at least one halfedge connecting them. More precisely, *Arrangement_2::Vertex_handle* is the graph-vertex type, while *Arrangement_2::Halfedge_handle* is the graph-edge type. As halfedges are directed, we consider the graph to be directed as well. Moreover, as several interior-disjoint *x*-monotone curves (say circular arcs) may share two common endpoints, inducing an arrangement with two vertices that are connected with several edges, we allow parallel edges in our BOOST graph.

Given an *Arrangement_2* instance, we can efficiently traverse its vertices and halfedges. Thus, the arrangement graph is a model of the concepts *VertexListGraph* and *EdgeListGraph* introduced by the BGL. At the same time, we use an iterator adapter of the circulator over the halfedges incident to a vertex (*Halfedge_around_vertex_circulator* — see Section 17.2.2), so it is possible to go over the ingoing and outgoing edges of a vertex in linear time. Thus, our arrangement graph is a model of the concept *BidirectionalGraph* (this concept refines *IncidenceGraph*, which requires only the traversal of outgoing edges).

¹⁶See also BOOST's homepage at: www.boost.org.

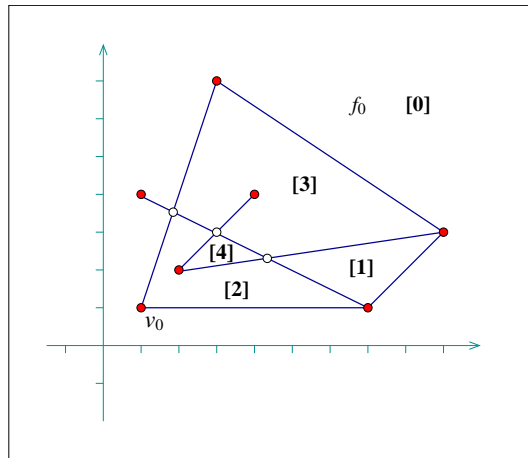


Figure 17.22: An arrangement of 7 line segments, as constructed by *ex_bgl_primal_adapter.C* and *ex_bgl_dual_adapter.C*. The breadth-first visit times for the arrangement faces, starting from the unbounded face f_0 , are shown in brackets.

It is important to notice that the vertex descriptors we use are *Vertex_handle* objects and *not* vertex indices. However, in order to gain more efficiency in most BGL algorithm, it is better to have them indexed $0, 1, \dots, (n - 1)$, where n is the number of vertices. We therefore introduce the *Arr_vertex_index_map*<Arrangement> class-template, which maintains a mapping of vertex handles to indices, as required by the BGL. An instance of this class must be attached to a valid arrangement vertex when it is created. It uses the notification mechanism (see Section 17.6) to automatically maintain the mapping of vertices to indices, even when new vertices are inserted into the arrangement or existing vertices are removed.

In most algorithm provided by the BGL, the output is given by *property maps*, such that each map entry corresponds to a vertex. For example, when we compute the shortest paths from a given source vertex s to all other vertices we can obtain a map of distances and a map of predecessors — namely for each v vertex we have its distance from s and a descriptor of the vertex that precedes v in the shortest path from s . If the vertex descriptors are simply indices, one can use vectors to efficiently represent the property maps. As this is not the case with the arrangement graph, we offer the *Arr_vertex_property_map*<Arrangement,Type> template allows for an efficient mapping of *Vertex_handle* objects to properties of type *Type*. Note however that unlike the *Arr_vertex_index_map* class, the vertex property-map class is not kept synchronized with the number of vertices in the arrangement, so it should not be reused in calls to BGL functions in case the arrangement is modified in between these calls.

In the following example we construct an arrangement of 7 line segments, as shown in Figure 17.22, then use Dijkstra's shortest-paths algorithm from the BGL to compute the graph distance of all vertices from the leftmost vertex in the arrangement v_0 . Note the usage of the *Arr_vertex_index_map* and the *Arr_vertex_property_map* classes. The latter one, instantiated by the type *double* is used to map vertices to their distances from v_0 .

```

//! \file examples/Arrangement_2/ex_bgl_primal_adapter.C
// Adapting an arrangement to a BGL graph.

#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>

#include <boost/graph/dijkstra_shortest_paths.hpp>

```

```

#include <CGAL/graph_traits_Arrangement_2.h>
#include <CGAL/Arr_vertex_map.h>

typedef CGAL::Cartesian<Number_type>           Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>      Traits_2;
typedef Traits_2::Point_2                      Point_2;
typedef Traits_2::X_monotone_curve_2           Segment_2;
typedef CGAL::Arrangement_2<Traits_2>          Arrangement_2;

// A functor used to compute the length of an edge.
class Edge_length_func
{
public:

    // Boost property type definitions:
    typedef boost::readable_property_map_tag    category;
    typedef double                               value_type;
    typedef value_type                          reference;
    typedef Arrangement_2::Halfedge_handle       key_type;

    double operator() (Arrangement_2::Halfedge_handle e) const
    {
        const double    x1 = CGAL::to_double (e->source()->point().x());
        const double    y1 = CGAL::to_double (e->source()->point().y());
        const double    x2 = CGAL::to_double (e->target()->point().x());
        const double    y2 = CGAL::to_double (e->target()->point().y());
        const double    diff_x = x2 - x1;
        const double    diff_y = y2 - y1;

        return (std::sqrt (diff_x*diff_x + diff_y*diff_y));
    }
};

double get (Edge_length_func edge_length, Arrangement_2::Halfedge_handle e)
{
    return (edge_length (e));
}

int main ()
{
    Arrangement_2    arr;

    // Construct an arrangement of seven intersecting line segments.
    // We keep a handle for the vertex v_0 that corresponds to the point (1,1).
    Arrangement_2::Halfedge_handle    e =
        insert_non_intersecting_curve (arr, Segment_2 (Point_2 (1, 1),
                                                         Point_2 (7, 1)));
    Arrangement_2::Vertex_handle    v0 = e->source();
    insert_curve (arr, Segment_2 (Point_2 (1, 1), Point_2 (3, 7)));
    insert_curve (arr, Segment_2 (Point_2 (1, 4), Point_2 (7, 1)));
    insert_curve (arr, Segment_2 (Point_2 (2, 2), Point_2 (9, 3)));
    insert_curve (arr, Segment_2 (Point_2 (2, 2), Point_2 (4, 4)));
    insert_curve (arr, Segment_2 (Point_2 (7, 1), Point_2 (9, 3)));
    insert_curve (arr, Segment_2 (Point_2 (3, 7), Point_2 (9, 3)));

```

```

// Create a mapping of the arrangement vertices to indices.
CGAL::Arr_vertex_index_map<Arrangement_2>    index_map (arr);

// Perform Dijkstra's algorithm from the vertex v0.
Edge_length_func                             edge_length;
CGAL::Arr_vertex_property_map<Arrangement_2,
                                double>       dist_map (index_map);

boost::dijkstra_shortest_paths (arr, v0,
                                boost::vertex_index_map (index_map).
                                weight_map (edge_length).
                                distance_map (dist_map));

// Print the results:
Arrangement_2::Vertex_iterator               vit;

std::cout << "The distances of the arrangement vertices from ("
            << v0->point() << ") : " << std::endl;
for (vit = arr.vertices_begin(); vit != arr.vertices_end(); ++vit)
{
    std::cout << "(" << vit->point() << ") at distance "
              << dist_map[vit] << std::endl;
}

return (0);
}

```

17.11.2 The Dual Arrangement Representation

It is possible to give a dual graph representation for an arrangement instance, such that each arrangement face corresponds to a graph vertex and two vertices are adjacent iff the corresponding faces share a common edge on their boundaries. This is done by specializing the *boost::graph_traits* template for *Dual<Arrangement_2>* instances, where *Dual<Arrangement_2>* is a template specialization that gives a dual interpretation to an arrangement instance.

In dual representation, *Arrangement_2::Face_handle* is the graph-vertex type, while *Arrangement_2::Halfedge_handle* is the graph-edge type. We treat the graph edges as directed, such that a halfedge *e* is directed from *f*₁, which is its incident face, to *f*₂, which is the incident face of its twin halfedge. As two arrangement faces may share more than a single edge on their boundary, we allow parallel edges in our BOOST graph. As is the case in the primal graph, the dual arrangement graph is also a model of the concepts *VertexListGraph*, *EdgeListGraph* and *BidirectionalGraph* (thus also of *IncidenceGraph*).

Since we use *Face_handle* objects as the vertex descriptors, we define the *Arr_face_index_map<Arrangement>* class-template, which maintains an efficient mapping of face handles to indices. We also provide the template *Arr_face_property_map<Arrangement,Type>* for associating arbitrary data with the arrangement faces.

In the following example we construct the same arrangement as in example *ex_bgl_primal_adapter.C* (see Figure 17.22), and perform breadth-first search on the graph faces, starting from the unbounded face. We extend the DCEL faces with an unsigned integer, marking the discover time of the face and use a breadth-first-search visitor to obtain these times and update the faces accordingly:

```

///! \file examples/Arrangement_2/ex_bgl_dual_adapter.C

```



```

// Adapting the dual of an arrangement to a BGL graph.

#include "arr_rational_nt.h"
#include <CGAL/Cartesian.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arr_extended_dcel.h>
#include <CGAL/Arrangement_2.h>

#include <boost/graph/dijkstra_shortest_paths.hpp>

#include <CGAL/graph_traits_Dual_Arrangement_2.h>
#include <CGAL/Arr_face_map.h>

#include "arr_print.h"

typedef CGAL::Cartesian<Number_type>           Kernel;
typedef CGAL::Arr_segment_traits_2<Kernel>      Traits_2;
typedef Traits_2::Point_2                      Point_2;
typedef Traits_2::X_monotone_curve_2           Segment_2;
typedef CGAL::Arr_face_extended_dcel<Traits_2,
                                     unsigned int> Dcel;
typedef CGAL::Arrangement_2<Traits_2, Dcel>      Arrangement_2;
typedef CGAL::Dual<Arrangement_2>               Dual_arrangement_2;

// A BFS visitor class that associates each vertex with its discover time.
// In our case graph vertices represent arrangement faces.
template <class IndexMap>
class Discover_time_bfs_visitor : public boost::default_bfs_visitor
{
private:
    const IndexMap *index_map;      // Mapping vertices to indices.
    unsigned int    time;           // The current time stamp.

public:
    // Constructor.
    Discover_time_bfs_visitor (const IndexMap& imap) :
        index_map (&imap),
        time (0)
    {}

    // Write the discover time for a given vertex.
    template <typename Vertex, typename Graph>
    void discover_vertex (Vertex u, const Graph& g)
    {
        u->set_data (time);
        time++;
    }
};

int main ()
{
    Arrangement_2  arr;

```

```

// Construct an arrangement of seven intersecting line segments.
insert_curve (arr, Segment_2 (Point_2 (1, 1), Point_2 (7, 1)));
insert_curve (arr, Segment_2 (Point_2 (1, 1), Point_2 (3, 7)));
insert_curve (arr, Segment_2 (Point_2 (1, 4), Point_2 (7, 1)));
insert_curve (arr, Segment_2 (Point_2 (2, 2), Point_2 (9, 3)));
insert_curve (arr, Segment_2 (Point_2 (2, 2), Point_2 (4, 4)));
insert_curve (arr, Segment_2 (Point_2 (7, 1), Point_2 (9, 3)));
insert_curve (arr, Segment_2 (Point_2 (3, 7), Point_2 (9, 3)));

// Create a mapping of the arrangement faces to indices.
CGAL::Arr_face_index_map<Arrangement_2>      index_map (arr);

// Perform breadth-first search from the unbounded face, and use the BFS
// visitor to associate each arrangement face with its discover time.
Discover_time_bfs_visitor<CGAL::Arr_face_index_map<Arrangement_2> >
                                bfs_visitor (index_map);
Arrangement_2::Face_handle      uf = arr.unbounded_face();

boost::breadth_first_search (Dual_arrangement_2 (arr), uf,
                             boost::vertex_index_map (index_map).
                             visitor (bfs_visitor));

// Print the results:
Arrangement_2::Face_iterator    fit;

for (fit = arr.faces_begin(); fit != arr.faces_end(); ++fit)
{
    std::cout << "Discover time " << fit->data() << " for ";
    if (fit != uf)
    {
        std::cout << "face ";
        print_ccb<Arrangement_2> (fit->outer_ccb());
    }
    else
        std::cout << "the unbounded face." << std::endl;
}

return (0);
}

```

17.12 How To Speed Up Your Computation

Before the specific tips, we remind you that compiling programs with debug flags disabled and with optimization flags enabled significantly reduces the running time.

1. When the curves to be inserted into an arrangement are x -monotone and pairwise disjoint in their interior to start with, then it is more efficient (in running time) and less demanding (in traits-class functionality) to use the non-intersection insertion-functions instead of the general ones; e.g., *insert_x_monotone_curve()*.
2. When the curves to be inserted into an arrangement are segments that are pairwise disjoint in their interior, it is more efficient to use the traits class *Arr_non_caching_segment_traits_2* rather than the default one

(*Arr_segment_traits_2*).

If the segments may intersect each other, the default traits class *Arr_segment_traits_2* can be safely used with the somehow limited number type *Quotient<MP_float>*.

On rare occasions the traits class *Arr_non_caching_segment_traits_2* exhibits slightly better performance than the default one (*Arr_segment_traits_2* even when the segments intersect each other, due to the small overhead of the latter (optimized) traits class. (For example, when the the so called LEDA rational kernel is used).

3. Prior knowledge of the combinatorial structure of the arrangement can be used to accelerate operations that insert x -monotone curves, whose interior is disjoint from existing edges and vertices of the arrangement. The specialized insertion functions, i.e., *insert_in_face_interior()*, *insert_from_left_vertex()*, *insert_from_right_vertex()*, and *insert_at_vertices()* can be used according to the available information. These functions hardly involve any geometric operations, if at all. They accept topologically related parameters, and use them to operate directly on the DCEL records, thus saving algebraic operations, which are especially expensive when high-degree curves are involved.

A polygon, represented by a list of segments along its boundary, can be inserted into an empty arrangement as follows. First, one segment is inserted using *insert_in_face_interior()* into the unbounded face. Then, a segment with a common end point is inserted using either *insert_from_left_vertex()* or *insert_from_right_vertex()*, and so on with the rest of the segments except for the last, which is inserted using *insert_at_vertices()*, as both endpoints of which are the mapping of known vertices.

4. The main trade-off among point-location strategies, is between time and storage. Using the naive or walk strategies, for example, takes more query time but does not require preprocessing or maintenance of auxiliary structures and saves storage space.
5. If point-location queries are not performed frequently, but other modifying functions, such as removing, splitting, or merging edges are, then using a point-location strategy that does not require the maintenance of auxiliary structures, such as the the naive or walk strategies, is preferable.
6. There is a trade-off between two modes of the trapezoidal RIC strategy that enables the user to choose whether preprocessing should be performed or not. If preprocessing is not used, the creation of the structure is faster. However, for some input sequences the structure might be unbalanced and therefore queries and updates might take longer, especially, if many removal and split operations are performed.
7. When the curves to be inserted into an arrangement are available in advance (as opposed to supplied on-line), it is advised to use the more efficient aggregate (sweep-based) insertion over the incremental insertion; e.g., *insert_curves()*.
8. The various traits classes should be instantiated with an exact number type to ensure robustness, when the input of the operations to be carried out might be degenerate, although inexact number types could be used at the user's own risk.
9. Maintaining short bit-lengths of coordinate representations may drastically decrease the time consumption of arithmetic operations on the coordinates. This can be achieved by caching certain information or normalization (of rational numbers). However, both solutions should be used cautiously, as the former may lead to an undue space consumption, and indiscriminate normalization may considerably slow down the overall process.
10. Geometric functions (e.g., traits methods) dominate the time consumption of most operations. Thus, calls to such function should be avoided or at least their number should be decreased, perhaps at the expense of increased combinatorial-function calls or increased space consumption. For example, repetition of geometric-function calls could be avoided by storing the results obtained by the first call, and reusing them when needed.

Design and Implementation History

The code of this package is the result of a long development process. Initially (and until version 3.1), the code was spread among several packages, namely *Topological_map*, *Planar_map_2*, *Planar_map_with_intersections_2* and *Arrangement_2*, that were developed by :

Ester Ezra, Eyal Flato, Efi Fogel, Dan Halperin, Iddo Hanniel, Idit Haran, Shai Hirsch, Eugene Lipovetsky, Oren Nechushtan, Sigal Raab, Ron Wein, Baruch Zukerman and Tali Zvi.

In version 3.2, as part of the ACS project, the packages have gone through a major re-design, resulting in an improved *Arrangement_2* package. The code of the new package was restructured and developed by :
Efi Fogel, Idit Haran, Ron Wein and Baruch Zukerman.

2D Arrangements

Reference Manual

Ron Wein, Efi Fogel, Baruch Zukerman, and Dan Halperin

Given a set C of planar curves, the *arrangement* $\mathcal{A}(C)$ is the subdivision of the plane induced by the curves in C into maximally connected cells. The cells can be 0-dimensional (*vertices*), 1-dimensional (*edges*) or 2-dimensional (*faces*).

The class `Arrangement_2<Traits,Dcel>` encapsulates a data structure that maintains arrangements of arbitrary bounded planar curves. It comes with a variety of algorithms that operate on planar arrangement, such as point-location queries and overlay computations, which are implemented as peripheral classes or as free (global) functions.

17.13 Classified Reference Pages

Concepts

<code>ArrangementDcel</code>	page 1236
<code>ArrangementDcelVertex</code>	page 1239
<code>ArrangementDcelHalfedge</code>	page 1241
<code>ArrangementDcelFace</code>	page 1244
<code>ArrangementDcelHole</code>	page 1246
<code>ArrangementDcelIsolatedVertex</code>	page 1247
<code>ArrangementBasicTraits_2</code>	page 1256
<code>ArrangementLandmarkTraits_2</code>	page 1259
<code>ArrangementXMonotoneTraits_2</code>	page 1261
<code>ArrangementTraits_2</code>	page 1263
<code>ArrangementInputFormatter</code>	page 1292
<code>ArrangementOutputFormatter</code>	page 1296
<code>ArrWithHistoryInputFormatter</code>	page 1323
<code>ArrWithHistoryOutputFormatter</code>	page 1325
<code>ArrangementPointLocation_2</code>	page 1303
<code>ArrangementVerticalRayShoot_2</code>	page 1305

Classes

<i>CGAL::Arrangement_2<Traits,Dcel></i>	page 1201
<i>CGAL::Arr_accessor<Arrangement></i>	page 1210
<i>CGAL::Arr_observer<Arrangement></i>	page 1312
<i>CGAL::Arrangement_2<Traits,Dcel>::Vertex</i>	page 1216
<i>CGAL::Arrangement_2<Traits,Dcel>::Halfedge</i>	page 1217
<i>CGAL::Arrangement_2<Traits,Dcel>::Face</i>	page 1218
<i>CGAL::Arr_dcel_base<V,H,F></i>	page 1248
<i>CGAL::Arr_default_dcel<Traits></i>	page 1250
<i>CGAL::Arr_face_extended_dcel<Traits,FData,V,H,F></i>	page 1251
<i>CGAL::Arr_extended_dcel<Traits,VData,HData,FData,V,H,F></i>	page 1252
<i>CGAL::Arr_segment_traits_2<Kernel></i>	page 1265
<i>CGAL::Arr_non_caching_segment_traits_2<Kernel></i>	page 1267
<i>CGAL::Arr_polyline_traits_2<SegmentTraits></i>	page 1268
<i>CGAL::Arr_circle_segment_traits_2<Kernel></i>	page 1271
<i>CGAL::Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits></i>	page 1277
<i>CGAL::Arr_rational_arc_traits_2<AlgKernel,NtTraits></i>	page 1283
<i>CGAL::Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv></i>	page 1286
<i>CGAL::Arr_consolidated_curve_data_traits_2<Traits,Data></i>	page 1289
<i>CGAL::Arr_text_formatter<Arrangement></i>	page 1300
<i>CGAL::Arr_face_extended_text_formatter<Arrangement></i>	page 1301
<i>CGAL::Arr_extended_dcel_text_formatter<Arrangement></i>	page 1302
<i>CGAL::Arr_with_history_text_formatter<ArrFormatter></i>	page 1327
<i>CGAL::Arr_naive_point_location<Arrangement></i>	page 1307
<i>CGAL::Arr_walk_along_line_point_location<Arrangement></i>	page 1308
<i>CGAL::Arr_trapezoid_ric_point_location<Arrangement></i>	page 1309
<i>CGAL::Arr_landmarks_point_location<Arrangement,Generator></i>	page 1310

Functions

<i>CGAL::is_valid</i>	page 1219
<i>CGAL::insert_curve</i>	page 1220
<i>CGAL::insert_curves</i>	page 1221
<i>CGAL::insert_x_monotone_curve</i>	page 1222
<i>CGAL::insert_x_monotone_curves</i>	page 1223
<i>CGAL::insert_non_intersecting_curve</i>	page 1224
<i>CGAL::insert_non_intersecting_curves</i>	page 1225
<i>CGAL::insert_point</i>	page 1226
<i>CGAL::remove_edge</i>	page 1227
<i>CGAL::remove_vertex</i>	page 1228
<i>CGAL::locate</i>	page 1311
<i>CGAL::overlay</i>	page 1229
<i>CGAL::read</i>	page 1234
<i>CGAL::write</i>	page 1235
<i>CGAL::remove_curve</i>	page 1322

17.14 Alphabetical List of Reference Pages

<i>ArrangementBasicTraits_2</i>	page 1256
<i>ArrangementDcelFace</i>	page 1244
<i>ArrangementDcelHalfedge</i>	page 1241
<i>ArrangementDcelHole</i>	page 1246
<i>ArrangementDcelIsolatedVertex</i>	page 1247
<i>ArrangementDcelVertex</i>	page 1239
<i>ArrangementDcel</i>	page 1236
<i>ArrangementInputFormatter</i>	page 1292
<i>ArrangementLandmarkTraits_2</i>	page 1259
<i>ArrangementOutputFormatter</i>	page 1296
<i>ArrangementPointLocation_2</i>	page 1303
<i>ArrangementTraits_2</i>	page 1263
<i>ArrangementVerticalRayShoot_2</i>	page 1305
<i>ArrangementXMonotoneTraits_2</i>	page 1261
<i>Arrangement_2<Traits,Dcel></i>	page 1201
<i>Arrangement_with_history_2<Traits,Dcel></i>	page 1318
<i>ArrWithHistoryInputFormatter</i>	page 1323
<i>ArrWithHistoryOutputFormatter</i>	page 1325
<i>Arr_accessor<Arrangement></i>	page 1210
<i>Arr_circle_segment_traits_2<Kernel></i>	page 1271
<i>Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits></i>	page 1277
<i>Arr_consolidated_curve_data_traits_2<Traits,Data></i>	page 1289
<i>Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv></i>	page 1286
<i>Arr_dcel_base<V,H,F></i>	page 1248
<i>Arr_default_dcel<Traits></i>	page 1250
<i>Arr_default_overlay_traits<Arrangement></i>	page 1232
<i>Arr_extended_dcel<Traits,VData,HData,FData,V,H,F></i>	page 1252
<i>Arr_extended_dcel.text_formatter<Arrangement></i>	page 1302
<i>Arr_extended_face<FaceBase,FData></i>	page 1255
<i>Arr_extended_halfedge<HalfedgeBase,HData></i>	page 1254
<i>Arr_extended_vertex<VertexBase,VData></i>	page 1253
<i>Arr_face_extended_dcel<Traits,FData,V,H,F></i>	page 1251
<i>Arr_face_extended.text_formatter<Arrangement></i>	page 1301
<i>Arr_face_overlay_traits<Arr_A,Arr_B,Arr_R,OvlFaceData></i>	page 1233
<i>Arr_landmarks_point_location<Arrangement,Generator></i>	page 1310
<i>Arr_naive_point_location<Arrangement></i>	page 1307
<i>Arr_non_caching_segment_basic_traits_2<Kernel></i>	page 1266
<i>Arr_non_caching_segment_traits_2<Kernel></i>	page 1267
<i>Arr_observer<Arrangement></i>	page 1312
<i>Arr_polyline_traits_2<SegmentTraits></i>	page 1268
<i>Arr_rational_arc_traits_2<AlgKernel,NtTraits></i>	page 1283
<i>Arr_segment_traits_2<Kernel></i>	page 1265
<i>Arr_text_formatter<Arrangement></i>	page 1300
<i>Arr_trapezoid_ric_point_location<Arrangement></i>	page 1309
<i>Arr_walk_along_line_point_location<Arrangement></i>	page 1308
<i>Arr_with_history_text_formatter<ArrFormatter></i>	page 1327
<i>Face</i>	page 1218
<i>Halfedge</i>	page 1217
<i>insert_curves</i>	page 1221
<i>insert_curve</i>	page 1220
<i>insert_non_intersecting_curves</i>	page 1225
<i>insert_non_intersecting_curve</i>	page 1224

<i>insert_point</i>	page 1226
<i>insert_x_monotone_curves</i>	page 1223
<i>insert_x_monotone_curve</i>	page 1222
<i>is_valid</i>	page 1219
<i>locate</i>	page 1311
<i>OverlayTraits</i>	page 1230
<i>overlay</i>	page 1229
<i>read</i>	page 1234
<i>remove_curve</i>	page 1322
<i>remove_edge</i>	page 1227
<i>remove_vertex</i>	page 1228
<i>Vertex</i>	page 1216
<i>write</i>	page 1235

CGAL::Arrangement_2<Traits,Dcel>

Definition

An object *arr* of the class *Arrangement_2<Traits,Dcel>* represents the planar subdivision induced by a set of *x*-monotone curves and isolated points into maximally connected cells. The arrangement is represented as a doubly-connected edge-list (DCEL) such that each DCEL vertex is associated with a point of the plane and each edge is associated with an *x*-monotone curve whose interior is disjoint from all other edges and vertices. Recall that an arrangement edge is always comprised of a pair of twin DCEL halfedges.

The *Arrangement_2<Traits,Dcel>* template has two parameters:

- The *Traits* template-parameter should be instantiated with a model of the *ArrangementBasicTraits_2* concept. The traits class defines the types of *x*-monotone curves and two-dimensional points, namely *X_monotone_curve_2* and *Point_2*, respectively, and supports basic geometric predicates on them.
- The *Dcel* template-parameter should be instantiated with a class that is a model of the *ArrangementDcel* concept. The value of this parameter is by default *Arr_default_dcel<Traits>*.

The available traits classes and DCEL classes are described below.

Self is an abbreviation of the *Arrangement_2<Traits,Dcel>* type hereafter.

```
#include <CGAL/Arrangement_2.h>
```

Types

<i>Arrangement_2<Traits,Dcel>:: Traits_2</i>	the traits class in use.
<i>Arrangement_2<Traits,Dcel>:: Dcel</i>	the DCEL representation of the arrangement.
<i>typedef Arrangement_2<Traits_2,Dcel></i>	<i>Self</i> ;
<i>typedef typename Traits_2::Point_2</i>	<i>Point_2</i> ; the point type, as defined by the traits class.
<i>typedef typename Traits_2::X_monotone_curve_2</i>	<i>X_monotone_curve_2</i> ; the <i>x</i> -monotone curve type, as defined by the traits class.
<i>typedef typename Dcel::Size</i>	<i>Size</i> ; the size type (equivalent to <i>size_t</i>).
<i>Arrangement_2<Traits,Dcel>:: Vertex</i>	represents a 0-dimensional cell in the subdivision. A vertex is always associated with a point.
<i>Arrangement_2<Traits,Dcel>:: Halfedge</i>	represents (together with its twin — see below) a 1-dimensional cell in the subdivision. A halfedge is always associated with an <i>x</i> -monotone curve.
<i>Arrangement_2<Traits,Dcel>:: Face</i>	represents a 2-dimensional cell in the subdivision.

The following handles, iterators, and circulators all have respective constant counterparts (for example, in addition to *Vertex_iterator* the type *Vertex_const_iterator* is also defined). See [MS96] for a discussion of constant versus mutable iterator types. The mutable types are assignable to their constant counterparts.

Vertex_iterator, *Halfedge_iterator*, and *Face_iterator* are equivalent to the respective handle types (namely, *Vertex_handle*, *Halfedge_handle*, and *Face_handle*). Thus, wherever the handles appear in function parameter lists, the respective iterators can be passed as well.

<i>Arrangement_2<Traits,Dcel>:: Vertex_handle</i>	a handle for an arrangement vertex.
<i>Arrangement_2<Traits,Dcel>:: Halfedge_handle</i>	a handle for a halfedge. The halfedge and its twin form together an arrangement edge.
<i>Arrangement_2<Traits,Dcel>:: Face_handle</i>	a handle for an arrangement face.
<i>Arrangement_2<Traits,Dcel>:: Vertex_iterator</i>	a bidirectional iterator over the vertices of the arrangement. Its value-type is <i>Vertex</i> .
<i>Arrangement_2<Traits,Dcel>:: Halfedge_iterator</i>	a bidirectional iterator over the halfedges of the arrangement. Its value-type is <i>Halfedge</i> .
<i>Arrangement_2<Traits,Dcel>:: Edge_iterator</i>	a bidirectional iterator over the edges of the arrangement. (That is, it skips every other halfedge.) Its value-type is <i>Halfedge</i> .
<i>Arrangement_2<Traits,Dcel>:: Face_iterator</i>	a bidirectional iterator over the faces of arrangement. Its value-type is <i>Face</i> .
<i>Arrangement_2<Traits,Dcel>:: Halfedge_around_vertex_circulator</i>	a bidirectional circulator over the halfedges that have a given vertex as their target. Its value-type is <i>Halfedge</i> .
<i>Arrangement_2<Traits,Dcel>:: Ccb_halfedge_circulator</i>	a bidirectional circulator over the halfedges of a CCB (connected component of the boundary). Its value-type is <i>Halfedge</i> . Each bounded face has a single CCB representing its outer boundary, and may have several inner CCBs representing its holes.
<i>Arrangement_2<Traits,Dcel>:: Hole_iterator</i>	a bidirectional iterator over the holes (i.e., inner CCBs) contained inside a given face. Its value type is <i>Ccb_halfedge_circulator</i> .
<i>Arrangement_2<Traits,Dcel>:: Isolated_vertex_iterator</i>	a bidirectional iterator over the isolated vertices contained inside a given face. Its value type is <i>Vertex</i> .

Creation

<i>Arrangement_2<Traits,Dcel> arr;</i>	constructs an empty arrangement containing one unbounded face, which corresponds to the entire plane.
<i>Arrangement_2<Traits,Dcel> arr(Self other);</i>	copy constructor.
<i>Arrangement_2<Traits,Dcel> arr(Traits_2 *traits);</i>	constructs an empty arrangement that uses the given <i>traits</i> instance for performing the geometric predicates.

Assignment Methods

<i>Self</i> &	<i>arr = other</i>	assignment operator.
<i>void</i>	<i>arr.assign(Self other)</i>	assigns the contents of another arrangement.
<i>void</i>	<i>arr.clear()</i>	clears the arrangement.

Access Functions

<i>Traits_2*</i>	<i>arr.get_traits()</i>	returns the traits object used by the arrangement instance. A <i>const</i> version is also available.
<i>bool</i>	<i>arr.is_empty()</i>	determines whether the arrangement is empty (contains only the unbounded face, with no vertices or edges).

All *_begin()* and *_end()* methods listed below also have *const* counterparts, returning constant iterators instead of mutable ones:

• Accessing the Arrangement Vertices:

<i>Size</i>	<i>arr.number_of_vertices()</i>	returns the number of vertices in the arrangement.
<i>Size</i>	<i>arr.number_of_isolated_vertices()</i>	returns the total number of isolated vertices in the arrangement.
<i>Vertex_iterator</i>	<i>arr.vertices_begin()</i>	returns the begin-iterator of the vertices in the arrangement.
<i>Vertex_iterator</i>	<i>arr.vertices_end()</i>	returns the past-the-end iterator of the vertices in the arrangement.

• Accessing the Arrangement Edges:

<i>Size</i>	<i>arr.number_of_halfedges()</i>	returns the number of halfedges in the arrangement.
<i>Halfedge_iterator</i>	<i>arr.halfedges_begin()</i>	returns the begin-iterator of the halfedges in the arrangement.
<i>Halfedge_iterator</i>	<i>arr.halfedges_end()</i>	returns the past-the-end iterator of the halfedges in the arrangement.
<i>Size</i>	<i>arr.number_of_edges()</i>	returns the number of edges in the arrangement (equivalent to <i>arr.number_of_halfedges() / 2</i>).
<i>Edge_iterator</i>	<i>arr.edges_begin()</i>	returns the begin-iterator of the edges in the arrangement.
<i>Edge_iterator</i>	<i>arr.edges_end()</i>	returns the past-the-end iterator of the edges in the arrangement.

- *Accessing the Arrangement Faces:*

<i>Face_handle</i>	<i>arr.unbounded_face()</i>	returns a handle for the unbounded face of the arrangement.
<i>Size</i>	<i>arr.number_of_faces()</i>	returns the number of faces in the arrangement.
<i>Face_iterator</i>	<i>arr.faces_begin()</i>	returns the begin-iterator of the faces in the arrangement.
<i>Face_iterator</i>	<i>arr.faces_end()</i>	returns the past-the-end iterator of the faces in the arrangement.

— *advanced* —

Casting away Constness

It is sometime necessary to convert a constant (non-mutable) handle to a mutable handle. For example, the result of a point-location query is a non-mutable handle for the arrangement cell containing the query point. Assume that the query point lies on a edge, so we obtain a *Halfedge_const_handle*; if we wish to use this handle and remove the edge, we first need to cast away its “constness”.

<i>Vertex_handle</i>	<i>arr.non_const_handle(Vertex_const_handle v)</i>	casts the given constant vertex handle to an equivalent mutable handle.
----------------------	-----------------------------------------------------	-------------------------------------------------------------------------

<i>Halfedge_handle</i>	<i>arr.non_const_handle(Halfedge_const_handle e)</i>	casts the given constant halfedge handle to an equivalent mutable handle.
------------------------	-------------------------------------------------------	---------------------------------------------------------------------------

<i>Face_handle</i>	<i>arr.non_const_handle(Halfedge_const_handle f)</i>	casts the given constant face handle to an equivalent mutable handle.
--------------------	-------------------------------------------------------	-----------------------------------------------------------------------

— *advanced* —

Modifiers

- *Specialized Insertion Methods:*

<i>Vertex_handle</i>	<i>arr.insert_in_face_interior(Point_2 p, Face_handle f)</i>	inserts the point <i>p</i> into the arrangement as an isolated vertex in the interior of the face <i>f</i> and returns a handle for the newly created vertex. <i>Precondition:</i> <i>p</i> lies in the interior of the face <i>f</i> .
----------------------	---------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Halfedge_handle *arr.insert_in_face_interior(X_monotone_curve_2 c, Face_handle f)*

inserts the curve *c* as a new hole (inner component) of the face *f*. As a result, two new vertices that correspond to *c*'s endpoints are created and connected with a newly created halfedge pair. The function returns a handle for one of the new halfedges corresponding to the inserted curve, directed in lexicographic increasing order (from left to right).
Precondition: *c* lies entirely in the interior of the face *f* and is disjoint from all existing arrangement vertices and edges (in particular, both its endpoints are not already associated with existing arrangement vertices).

Halfedge_handle *arr.insert_from_left_vertex(X_monotone_curve_2 c, Vertex_handle v)*

inserts the curve *c* into the arrangement, such that its left endpoint corresponds to a given arrangement vertex. As a result, a new vertex that correspond to *c*'s right endpoint is created and connected to *v* with a newly created halfedge pair. The function returns a handle for one of the new halfedges corresponding to the inserted curve, directed towards the newly created vertex — that is, directed in lexicographic increasing order (from left to right).
Precondition: The interior of *c* is disjoint from all existing arrangement vertices and edges.
Precondition: *v* is associated with the left endpoint of *c*.
Precondition: The right endpoint of *c* is not already associated with an existing arrangement vertex.

Halfedge_handle *arr.insert_from_right_vertex(X_monotone_curve_2 c, Vertex_handle v)*

inserts the curve *c* into the arrangement, such that its right endpoint corresponds to a given arrangement vertex. As a result, a new vertex that correspond to *c*'s left endpoint is created and connected to *v* with a newly created halfedge pair. The function returns a handle for one of the new halfedges corresponding to the inserted curve, directed to the newly created vertex — that is, directed in lexicographic decreasing order (from right to left).
Precondition: The interior of *c* is disjoint from all existing arrangement vertices and edges.
Precondition: *v* is associated with the right endpoint of *c*.
Precondition: The left endpoint of *c* is not already associated with an existing arrangement vertex.

```
Halfedge_handle arr.insert_at_vertices( X_monotone_curve_2 c, Vertex_handle v1, Vertex_handle v2)
```

inserts the curve c into the arrangement, such that both c 's endpoints correspond to existing arrangement vertices, given by $v1$ and $v2$. The function creates a new halfedge pair that connects the two vertices, and returns a handle for the halfedge directed from $v1$ to $v2$.

Precondition: The interior of c is disjoint from all existing arrangement vertices and edges.

Precondition: $v1$ and $v2$ are associated with c 's endpoints.

Precondition: If $v1$ and $v2$ are already connected by an edge, this edge represents an x -monotone curve that is interior-disjoint from c).

- *advanced*

```
Halfedge_handle arr.insert_from_left_vertex( X_monotone_curve_2 c, Halfedge_handle pred)
```

inserts the curve c into the arrangement, such that its left endpoint corresponds to a given arrangement vertex. This vertex is the target vertex of the halfedge $pred$, such that c is inserted to the circular list of halfedges around $pred \rightarrow target()$ right between $pred$ and its successor. The function returns a handle for one of the new halfedges directed (lexicographically) from left to right.

Precondition: The interior of c is disjoint from all existing arrangement vertices and edges.

Precondition: $pred \rightarrow target()$ is associated with the left endpoint of c , and c should be inserted after $pred$ in a clockwise order around this vertex.

Precondition: The right endpoint of c is not already associated with an existing arrangement vertex.

```
Halfedge_handle arr.insert_from_right_vertex( X_monotone_curve_2 c, Halfedge_handle pred)
```

inserts the curve c into the arrangement, such that its right endpoint corresponds to a given arrangement vertex. This vertex is the target vertex of the halfedge $pred$, such that c is inserted to the circular list of halfedges around $pred \rightarrow target()$ right between $pred$ and its successor. The function returns a handle for one of the new halfedges directed (lexicographically) from right to left.

Precondition: The interior of c is disjoint from all existing arrangement vertices and edges.

Precondition: $pred \rightarrow target()$ is associated with the right endpoint of c , and c should be inserted after $pred$ in a clockwise order around this vertex.

Precondition: The left endpoint of c is not already associated with an existing arrangement vertex.

[illegible]

Vertex_handle v2)

inserts the curve c into the arrangement, such that both c 's endpoints correspond to existing arrangement vertices, given by $predI \rightarrow target()$ and $v2$. The function creates a new halfedge pair that connects the two vertices (where the corresponding halfedge is inserted right between $predI$ and its successor around $predI$'s target vertex) and returns a handle for the halfedge directed from $predI \rightarrow target()$ to $v2$.

Precondition: The interior of c is disjoint from all existing arrangement vertices and edges.

Precondition: $pred1 \rightarrow target()$ and $v2$ are associated with c 's endpoints.

Precondition: If $pred1 \rightarrow target$ and $v2$ are already connected by an edge, this edge represents an x -monotone curve that is interior-disjoint from c).

[illegible]

inserts the curve c into the arrangement, such that both c 's endpoints correspond to existing arrangement vertices, given by $pred1 \rightarrow target()$ and $pred2 \rightarrow target()$. The function creates a new halfedge pair that connects the two vertices (with $pred1$ and $pred2$ indicate the exact place for these halfedges around the two target vertices) and returns a handle for the halfedge directed from $pred1 \rightarrow target()$ to $pred2 \rightarrow target()$.

Precondition: The interior of c is disjoint from all existing arrangement vertices and edges.

Precondition: $pred1 \rightarrow target()$ and $pred2 \rightarrow target()$ are associated with c 's endpoints.

Precondition: If $pred1 \rightarrow target$ and $pred2 \rightarrow target()$ are already connected by an edge, this edge represents an x -monotone curve that is interior-disjoint from c).

_____ *advanced* _____

- *Modifying Vertices and Edges:*

```
Vertex_handle      arr.modify_vertex( Vertex_handle v, Point_2 p)
```

sets p to be the point associated with the vertex v . The function returns a handle for the modified vertex (same as v).

Precondition: p is geometrically equivalent to the point currently associated with γ .

bool remove_target = true)

removes the edge e from the arrangement. Since the e may be the only edge incident to its source vertex (or its target vertex), this vertex can be removed as well. The flags *remove_source* and *remove_target* indicate whether the endpoints of e should be removed, or whether they should be left as isolated vertices in the arrangement. If the operation causes two faces to merge, the merged face is returned. Otherwise, the face to which the edge was incident is returned.

Miscellaneous

bool *arr.is_valid()*

returns *true* if *arr* represents a valid instance of *Arrangement_2<Traits,Dcel>*. In particular, the function checks the topological structure of the arrangement and assures that it is valid. In addition, the function performs several simple geometric tests to ensure the validity of some of the geometric properties of the arrangement. Namely, it checks that all arrangement vertices are associated with distinct points, and that the halfedges around every vertex are ordered clockwise.

See Also

ArrangementDcel(page [1236](#))

Arr_default_dcel<Traits> (page [1250](#))

ArrangementBasicTraits_2(page [1256](#))

CGAL::is_valid(page [1219](#))

Insertion functions (page [1220](#), page [1221](#), page [1222](#), page [1223](#), page [1224](#), page [1225](#), page [1226](#))

Removal functions (page [1227](#), page [1228](#))

CGAL::overlay(page [1229](#))

Input/output functions (page [1234](#),page [1235](#))

CGAL::Arr_accessor<Arrangement>

Definition

Arr_accessor<*Arrangement*> provides an access to some of the private *Arrangement* functions. Users may use these functions to achieve more efficient programs when they have the exact topological information required by the specialized functions.

It is however highly recommended to be very careful when using the accessor functions that modify the arrangement. As we have just mentioned, these functions have very specific requirement on their input on one hand, and perform no preconditions on the other hand, so providing incorrect topological input may invalidate the arrangement.

```
#include <CGAL/Arr_accessor.h>
```

Types

Arr_accessor<*Arrangement*>::*Arrangement_2* the type of the associated arrangement.

```
typedef typename Arrangement_2::Point_2
```

```
Point_2;                    the point type.
```

```
typedef typename Arrangement_2::X_monotone_curve_2
```

```
X_monotone_curve_2;
```

the *x*-monotone curve type.

```
typedef typename Arrangement_2::Vertex_handle
```

```
Vertex_handle;
```

```
typedef typename Arrangement_2::Halfedge_handle
```

```
Halfedge_handle;
```

```
typedef typename Arrangement_2::Face_handle
```

```
Face_handle;
```

```
typedef typename Arrangement_2::Ccb_halfedge_circulator
```

```
Ccb_halfedge_circulator;
```

represents the boundary of a connected component (CCB).

Creation

```
Arr_accessor<Arrangement> acc( Arrangement_2& arr);
```

constructs an accessor attached to the given arrangement *arr*.

Accessing the notification functions

void *acc.notify_before_global_change()*

notifies the arrangement observer that a global change is going to take place (for the usage of the global functions that operate on arrangements).

void *acc.notify_after_global_change()*

notifies the arrangement observer that a global change has taken place (for the usage of the global functions that operate on arrangements).

— *advanced* —

Arrangement Predicates

Halfedge_handle *acc.locate_around_vertex(Vertex_handle v, X_monotone_curve_2 c)*

locates a place for the curve *c* around the vertex *v* and returns a halfedge whose target is *v*, where *c* should be inserted between this halfedge and the next halfedge around *v* in a clockwise order.

int *acc.halfedge_distance(Halfedge_const_handle e1, Halfedge_const_handle e2)*

counts the number of edges along the path from *e1* to *e2*. In case the two halfedges do not belong to the same connected component, the function returns (-1).

bool *acc.is_inside_new_face(Halfedge_handle pred1, Halfedge_handle pred2, X_monotone_curve_2 c)*

determines whether a new halfedge we are about to create, which is to be associated with the curve *c* and directed from *pred1*->*target()* to *pred2*->*target()*, lies on the inner CCB of the new face that will be created, introducing this new edge.
Precondition: *pred1*->*target()* and *pred2*->*target()* are associated with *c*'s endpoints.
Precondition: *pred1* and *pred2* belong to the same connected component, such that a new face is created by connecting *pred1*->*target()* and *pred2*->*target()*.

bool *acc.point_is_in(Point_2 p, Halfedge_const_handle he)*

determines whether a given point lies within the region bounded by a boundary of the connected component that *he* belongs to. Note that if the function returns *true*, then *p* is contained within *he*->*face()* (but not on its boundary), or inside one of the holes inside this face, or it may coincide with an isolated vertex in this face.

<i>Halfedge_handle</i>	<code>acc.insert_at_vertices_ex(X_monotone_curve_2 c, Halfedge_handle pred1, Halfedge_handle pred2, Comparison_result res, bool& new_face)</code> <p>inserts the curve <i>c</i> into the arrangement, such that both <i>c</i>'s endpoints correspond to existing arrangement vertices, given by <i>pred1</i>-><i>target()</i> and <i>pred2</i>-><i>target()</i>. <i>res</i> is the comparison result between these two end-vertices. The function creates a new halfedge pair that connects the two vertices (with <i>pred1</i> and <i>pred2</i> indicate the exact place for these halfedges around the two target vertices) and returns a handle for the halfedge directed from <i>pred1</i>-><i>target()</i> to <i>pred2</i>-><i>target()</i>. The output flag <i>new_face</i> indicates whether a new face has been created following the insertion of the new curve. <i>Precondition:</i> It is the user's responsibility that <i>pred1</i>-><i>target()</i> and <i>pred2</i>-><i>target()</i> are associated with <i>c</i>'s endpoints and that if a new face is created, then <i>is_inside_new_face</i> (<i>pred1</i>, <i>pred2</i>, <i>c</i>) is <i>true</i>.</p>
<i>void</i>	<code>acc.insert_isolated_vertex(Face_handle f, Vertex_handle v)</code> <p>inserts <i>v</i> as an isolated vertex inside <i>f</i>. <i>Precondition:</i> It is the user's responsibility that <i>v</i>-><i>point()</i> is contained in the interior of the given face.</p>
<i>void</i>	<code>acc.move_hole(Face_handle f1, Face_handle f2, Ccb_halfedge_circulator hole)</code> <p>moves the given hole from the interior of the face <i>f1</i> inside the face <i>f2</i>. <i>Precondition:</i> It is the user's responsibility that <i>hole</i> is currently contained in <i>f1</i> and should be moved to <i>f2</i>.</p>
<i>bool</i>	<code>acc.move_isolated_vertex(Face_handle f1, Face_handle f2, Vertex_handle v)</code> <p>moves the given isolated vertex from the interior of the face <i>f1</i> inside the face <i>f2</i>. <i>Precondition:</i> It is the user's responsibility that <i>v</i> is indeed an isolated vertex currently contained in <i>f1</i> and should be moved to <i>f2</i>.</p>
<i>void</i>	<code>acc.relocate_in_new_face(Halfedge_handle he)</code> <p>relocates all holes and isolated vertices to their proper position immediately after a face has split due to the insertion of a new halfedge, namely after <i>insert_at_vertices_ex()</i> was invoked and indicated that a new face has been created. <i>he</i> is the halfedge returned by <i>insert_at_vertices_ex()</i>, such that <i>he</i>-><i>twin()</i>-><i>face</i> is the face that has just been split and <i>he</i>-><i>face()</i> is the newly created face.</p>

<i>void</i>	<i>acc.relocate_holes_in_new_face(Halfedge_handle he)</i>	relocates all holes in a new face, as detailed above.
<i>void</i>	<i>acc.relocate_isolated_vertices_in_new_face(Halfedge_handle he)</i>	relocates all isolated vertices in a new face, as detailed above.
<i>Vertex_handle</i>	<i>acc.modify_vertex_ex(Vertex_handle v, Point_2 p)</i>	<p>modifies the point associated with the vertex v (the point may be geometrically different than the one currently associated with v). The function returns a handle to the modified vertex (same as v).</p> <p><i>Precondition:</i> It is the user's responsibility that no other arrangement vertex is already associated with p.</p>
<i>Halfedge_handle</i>	<i>acc.modify_edge_ex(Halfedge_handle e, X_monotone_curve_2 c)</i>	<p>modifies the x-monotone curve associated with the edge e (the curve c may be geometrically different than the one currently associated with e). The function returns a handle to the modified edge (same as e).</p> <p><i>Precondition:</i> It is the user's responsibility that the interior of c is disjoint from all existing arrangement vertices and edges.</p>
<i>Halfedge_handle</i>	<i>acc.split_edge_ex(Halfedge_handle he, Point_2 p, X_monotone_curve_2 c1, X_monotone_curve_2 c2)</i>	<p>splits a given edge into two at the split point p, and associate the x-monotone curves $c1$ and $c2$ with the resulting edges, such that $c1$ connects $he \rightarrow source()$ with p and $c2$ connects p with $he \rightarrow target()$. The function return a handle to the split halfedge directed from $he \rightarrow source()$ to the split point p.</p> <p><i>Precondition:</i> It is the user's responsibility that the endpoints of $c1$ and $c2$ correspond to p and to he's end-vertices, as indicated above.</p>
<i>Halfedge_handle</i>	<i>acc.split_edge_ex(Halfedge_handle he, Vertex_handle v, X_monotone_curve_2 c1, X_monotone_curve_2 c2)</i>	<p>splits a given edge into two at by the vertex v, and associate the x-monotone curves $c1$ and $c2$ with the resulting edges, such that $c1$ connects $he \rightarrow source()$ with v and $c2$ connects v with $he \rightarrow target()$. The function return a handle to the split halfedge directed from $he \rightarrow source()$ to v.</p> <p><i>Precondition:</i> It is the user's responsibility that the endpoints of $c1$ and $c2$ correspond to v and to he's end-vertices, as indicated above. It is also assumed that v has no incident edges.</p>

Face_handle

```
acc.remove_edge_ex( Halfedge_handle he,  
bool remove_source = true,  
bool remove_target = true)
```

removes the edge *he* from the arrangement, such that if the edge removal causes the creation of a new hole, *he->target()* lies on the boundary of this hole. The flags *remove_source* and *remove_target* indicate whether the end-vertices of *he* should be removed as well, in case they have no other incident edges. If the operation causes two faces to merge, the merged face is returned. Otherwise, the face to which the edge was incident is returned.

└────────── *advanced* ─────────┘

CGAL::Arrangement_2<Traits,Dcel>::Vertex

Definition

An object v of the class *Vertex* represents an arrangement vertex, that is — a 0-dimensional cell, associated with a point on the plane.

Inherits From

typename Dcel::Vertex

Creation

<i>Vertex</i> <i>v</i> ;	default constructor.
--------------------------	----------------------

Access Functions

All non-const methods listed below also have *const* counterparts that return constant handles, iterators or circulators:

<i>bool</i>	<i>v.is_isolated()</i>	checks whether the vertex is isolated. (It has no incident edges.)
-------------	------------------------	--------------------------------------------------------------------

<i>typename Dcel::Size</i>	<i>v.degree()</i>	returns the number of edges incident to <i>v</i> .
----------------------------	-------------------	----------------------------------------------------

<i>Halfedge_around_vertex_circulator</i>	<i>v.incident_halfedges()</i>
------------------------------------------	-------------------------------

returns a circulator circulator that allows going over the halfedges incident to v (that have v as their target). The edges are traversed in a clockwise direction around v .

Precondition: v is *not* an isolated vertex.

<i>Face_handle</i>	<i>v.face()</i>	returns a handle to the face that contains <i>v</i> in its interior.
--------------------	-----------------	----------------------------------------------------------------------

Precondition: v is an isolated vertex.

<i>typename Traits::Point_2</i>	<i>v.point()</i>	returns the point associated with the vertex.
---------------------------------	------------------	-----------------------------------------------

CGAL::Arrangement_2<Traits,Dcel>::Face

Definition

An object e of the class *Face* represents an arrangement edge, namely a 2-dimensional arrangement cell. Every arrangement contains exactly one *unbounded* face, an a number of bounded face. Each bounded face has a boundary comprised of a halfedge chain winding in a counterclockwise orientation around it. A face may also contain holes, which are defined by clockwise-oriented halfedge chains, and isolated vertices.

Inherits From

typename Dcel::Face

Creation

<i>Face</i> <i>f</i> ;	default constructor.
------------------------	----------------------

Access Functions

All non-const methods listed below also have *const* counterparts that return constant handles, iterators or circulators:

<i>bool</i>	<i>f.is_unbounded()</i>	returns whether the face is unbounded.
-------------	-------------------------	----------------------------------------

Ccb_halfedge_circulator

f.outer_ccb() returns a circulator that allows going over the outer boundary of *f*. The edges along the CCB are traversed in a counter-clockwise direction.
Precondition: *f* is not an unbounded face.

<i>Hole_iterator</i>	<i>f.holes_begin()</i>	returns an iterator for traversing all the holes (inner CCBs) of <i>f</i> .
----------------------	------------------------	-----------------------------------------------------------------------------

<i>Hole_iterator</i>	<i>f.holes_end()</i>	returns a past-the-end iterator for the holes of <i>f</i> .
----------------------	----------------------	-------------------------------------------------------------

Isolated_vertex_iterator

`f.isolated_vertices_begin()`

returns an iterator for traversing all the isolated vertices contained in the interior of f .

Isolated_vertex_iterator

`f.isolated_vertices_end()`

returns a past-the-end iterator for the isolated vertices contained inside `f`.

CGAL::is_valid

```
#include <CGAL/Arrangement_2.h>
```

```
template<class Traits, class Dcel>
```

```
bool is_valid( Arrangement_2<Traits, Dcel> arr)
```

Checks the validity of the given arrangement. It first invokes the member function *arr.is_valid()* to ensure the topological correctness of the arrangement. Then it performs additional validity tests. First, it checks that all *x*-monotone curves associated with arrangement edges are pairwise disjoint in their interior. Then it makes sure that all holes and all isolated vertices are located within the proper arrangement faces. The traits class in use must be a model of the concept *ArrangementXMonotoneTraits_2*. Note that the test carried out by this functions may take a considerable amount of time; it should therefore be used only for debugging purposes.

CGAL::insert_curve

```
#include <CGAL/Arrangement_2.h>
```

```
template<class Traits, class Dcel, class PointLocation>
void insert_curve( Arrangement_2<Traits,Dcel>& arr,
                  typename Traits::Curve_2 c,
                  PointLocation pl = walk_pl)
```

Inserts the given curve c into the arrangement arr , where no restrictions are made on the nature of the inserted curve. The *Traits* parameter must be a model of the *ArrangementTraits_2* concept. That is, it should define the *Curve_2* type and support its subdivision into x -monotone subcurves (and perhaps isolated points). Each subcurve is in turn inserted into the arrangement by locating its left endpoint and computing its zone until reaching the right endpoint. The point-location object pl , which must be a model of the *ArrangementPointLocation_2* concept, is used for answering point-location queries during the insertion process. By default, the function uses the “walk along line” point-location strategy — namely an instance of the class *Arr_walk_along_line_point_location*<*Arrangement_2*<*Traits*,*Dcel*>>.

Precondition: If provided, pl must be attached to the given arrangement arr .

```
#include <CGAL/Arrangement_with_history_2.h>
```

```
template<class Traits, class Dcel, class PointLocation>
typename Arrangement_with_history_2<Traits,Dcel>::Curve_handle
insert_curve( Arrangement_with_history_2<Traits,Dcel>& arr,
              typename Traits::Curve_2 c,
              PointLocation pl = walk_pl)
```

Inserts the given curve c into the arrangement arr , and returns a handle to the inserted curve. The point-location object pl , which should be a model of the *ArrangementPointLocation_2* concept, is used for answering point-location queries during the insertion process. By default, the function uses the “walk along line” point-location strategy — namely an instance of the class *Arr_walk_along_line_point_location*<*Arrangement_with_history_2*<*Traits*,*Dcel*>>.

Precondition: If provided, pl is attached to the given arrangement arr .

CGAL::insert_curves

```
#include <CGAL/Arrangement_2.h>
```

```
template<class Traits, class Dcel, InputIterator>
void          insert_curves( Arrangement_2<Traits,Dcel>& arr,
                           InputIterator first,
                           InputIterator last)
```

Aggregately inserts the given range of curves $[first, last)$ into the arrangement *arr*, where no restrictions are made on the nature of the inserted curves. The *Traits* parameter must be a model of the *ArrangementTraits_2* concept. The input curves are subdivided into *x*-monotone subcurves (and perhaps isolated points), which are inserted into the arrangement using the sweep-line algorithm.

Precondition: The value-type of *InputIterator* is *Traits::Curve_2*.

```
#include <CGAL/Arrangement_with_history_2.h>
```

```
template<class Traits, class Dcel, InputIterator>
void          insert_curves( Arrangement_with_history_2<Traits,Dcel>& arr,
                           InputIterator first,
                           InputIterator last)
```

Aggregately inserts the given range of curves $[first, last)$ into the arrangement *arr*.

Precondition: The value-type of *InputIterator* is *Traits::Curve_2*.

CGAL::insert_x_monotone_curve

```
#include <CGAL/Arrangement_2.h>
```

```
template<class Traits, class Dcel, class PointLocation>
void insert_x_monotone_curve( Arrangement_2<Traits,Dcel>& arr,
                             typename Traits::X_monotone_curve_2 xc,
                             PointLocation pl = walk_pl)
```

Inserts the given x -monotone curve xc into the arrangement arr . The *Traits* parameter must be a model of the *ArrangementXMonotoneTraits_2* concept. xc is inserted into the arrangement by locating its left endpoint and computing its zone until reaching the right endpoint. The point-location object pl , which must be a model of the *ArrangementPointLocation_2* concept, is used for locating the left endpoint of xc in the existing arrangement. By default, the function uses the “walk along line” point-location strategy — namely an instance of the class *Arr_walk_along_line_point_location<Arrangement_2<Traits,Dcel>>*.

Precondition: If provided, pl must be attached to the given arrangement arr .

```
template<class Traits, class Dcel>
void insert_x_monotone_curve( Arrangement_2<Traits,Dcel>& arr,
                             typename Traits::X_monotone_curve_2 xc,
                             Object obj)
```

Inserts the given x -monotone curve xc into the arrangement arr . The *Traits* parameter must be a model of the *ArrangementXMonotoneTraits_2* concept. The object obj , which either wraps a *Vertex_const_handle*, a *Halfedge_const_handle* or a *Face_const_handle*, represents the location of xc ’s left endpoint in the arrangement. xc is thus inserted into the arrangement from the feature represented by obj by computing its zone until reaching the right endpoint (no point-location query is issued).

CGAL::insert_x_monotone_curves

```
#include <CGAL/Arrangement_2.h>
```

```
template<class Traits, class Dcel, InputIterator>  
void          insert_x_monotone_curves( Arrangement_2<Traits,Dcel>& arr,  
                                       InputIterator first,  
                                       InputIterator last)
```

Inserts the given range of x -monotone curves $[first, last)$ into the arrangement arr . The insertion is performed in an aggregated manner, using the sweep-line algorithm. The *Traits* parameter must be a model of the *ArrangementXMonotoneTraits_2* concept.

Precondition: The value-type of *InputIterator* is *Traits::X_monotone_curve_2*.

CGAL::insert_non_intersecting_curve

```
#include <CGAL/Arrangement_2.h>
```

```
template<class Traits, class Dcel, class PointLocation>
typename Arrangement_2<Traits,Dcel>::Halfedge_handle
```

```
insert_non_intersecting_curve( Arrangement_2<Traits,Dcel>& arr,
                               typename Traits::X_monotone_curve_2 xc,
                               PointLocation pl = walk_pl)
```

inserts the given x -monotone curve xc into the arrangement arr , where the interior of xc is disjoint from all existing arrangement vertices and edges. Under this assumption, it is possible to locate the endpoints of xc in the arrangement and use one of the specialized insertion member-functions of arr according to the results. As no intersection are computed, the *Traits* parameter must model the restricted *ArrangementBasicTraits_2* concept. The point-location object pl , which must model the *ArrangementPointLocation_2* concept, is used for answering the two point-location queries on xc 's endpoints. The insertion operation creates a single new edge, that is, two twin halfedges, and the function returns a handle for the one directed lexicographically in increasing order (from left to right). By default, the function uses the “walk along line” point-location strategy — namely, an instance of the class *Arr_walk_along_line_point_location<Arrangement_2<Traits,Dcel>>*.

Precondition: If provided, pl must be attached to the given arrangement arr .

CGAL::insert_non_intersecting_curves

```
#include <CGAL/Arrangement_2.h>
```

```
template<class Traits, class Dcel, InputIterator>  
void          insert_non_intersecting_curves( Arrangement_2<Traits,Dcel>& arr,  
                                              InputIterator first,  
                                              InputIterator last)
```

Aggregately inserts the given range of x -monotone curves $[first, last)$ into the arrangement arr . The input curves should be pairwise disjoint in their interior and pairwise interior-disjoint from all existing arrangement vertices and edges. The *Traits* parameter must be a model of the *ArrangementBasicTraits_2* concept.

Precondition: The value-type of *InputIterator* is *Traits::X_monotone_curve_2*.

CGAL::insert_point

```
#include <CGAL/Arrangement_2.h>
```

```
template<class Traits, class Dcel, class PointLocation>
```

```
typename Arrangement_2<Traits,Dcel>::Vertex_handle
```

```
insert_point( Arrangement_2<Traits,Dcel>& arr,
              typename Traits::Point_2 p,
              PointLocation pl = walk_pl)
```

inserts the point p as an arrangement vertex into arr . The function uses the point-location object pl (a model of the *ArrangementPointLocation_2* concept) to locate p in arr . If it coincides with an existing vertex, there is nothing left to do; if it lies on an edge, the edge is split at p ; otherwise, p is contained inside a face, and is inserted as an isolated vertex inside this face. At any case, the function returns a handle for the vertex associated with p . The *Triats* parameter must model the refined concept *ArrangementXMonotoneTraits_2*, which requires split operations. By default, the function uses the “walk along line” point-location strategy — namely, an instance of the class *Arr_walk_along_line_point_location<Arrangement_2<Traits,Dcel> >*.

Precondition: If provided, pl must be attached to the given arrangement arr .

CGAL::remove_edge

```
#include <CGAL/Arrangement_2.h>
```

```
template<class Traits, class Dcel>
```

```
typename Arrangement_2<Traits,Dcel>::Face_handle
```

```
remove_edge( Arrangement_2<Traits,Dcel>& arr,  
             typename Arrangement_2<Traits,Dcel>::Halfedge_handle e)
```

removes an edge given by one of the twin halfedges e that forms it, from the arrangement arr . Using this function is equivalent to invoking `arr.remove_edge(e , $true$, $true$)` — that is, to remove the edge and its end-vertices, in case they become isolated. However, this free function requires that $Traits$ be a model of the refined concept *ArrangementXMonotoneTraits_2*, which requires merge operations on x -monotone curves. If one of the end-vertices of e becomes redundant after e is removed (see `remove_vertex()` for the definition of a redundant vertex), it is removed and its incident edges are merged. If the edge-removal operation causes two faces to merge, the merged face is returned. Otherwise, the face to which the edge was incident is returned.

CGAL::remove_vertex

```
#include <CGAL/Arrangement_2.h>
```

```
template <class Traits, class Dcel>
```

```
bool remove_vertex( Arrangement_2<Traits,Dcel>& arr,  
                   typename Arrangement_2<Traits,Dcel>::Vertex_handle v)
```

Tries to removed the vertex v from the given arrangement arr . The vertex can be removed if it is either an isolated vertex and has no incident edge, or if it is a *redundant* vertex — that is, it has exactly two incident edges whose associated curves can be merged to form a single x -monotone curve. The assumption is that *Traits* is a model of the refines *ArrangementXMonotoneTraits_2*, such that it supports merge operations on x -monotone curves. The function returns whether it succeeded in deleting v from the arrangement.

CGAL::overlay

```
#include <CGAL/Arr_overlay.h>
```

```
template<class Traits, class Dcel1, class Dcel2, class RedDcel, class OverlayTraits>
void overlay( Arrangement_2<Traits,Dcel1> arr1,
              Arrangement_2<Traits,Dcel2> arr2,
              Arrangement_2<Traits,ResDcel>& res,
              OverlayTraits ovl_tr)
```

Computes the overlay of two input arrangement instances *arr1* and *arr2* and sets the output arrangement *res* to represent the overlaid arrangement. All three arrangements are instantiated using the same geometric traits class, but may be represented using different DCEL classes. In order to properly construct the overlaid DCEL that represents *res*, the function uses the overlay-traits object *ovl_tr*, which should be a model of the *OverlayTraits* concept, which is able to construct records of the *ResDcel* class on the basis of the *Dcel1* and *Dcel2* records that induce them.

Precondition: *res* does not refer to either *arr1* or *arr2* (that is, “self overlay” is not supported).

```
#include <CGAL/Arrangement_with_history_2.h>
```

```
template<class Traits, class Dcel1, class Dcel2, class RedDcel, class OverlayTraits>
void overlay( Arrangement_with_history_2<Traits,Dcel1> arr1,
              Arrangement_with_history_2<Traits,Dcel2> arr2,
              Arrangement_2<Traits,ResDcel>& res,
              OverlayTraits ovl_tr)
```

Computes the overlay of two input arrangement-with-history instances *arr1* and *arr2* and sets the output arrangement *res* to represent the overlaid arrangement. All three arrangements are instantiated using the same geometric traits class, but may be represented using different DCEL classes. In order to properly construct the overlaid DCEL that represents *res*, the function uses the overlay-traits object *ovl_tr*, which should be a model of the *OverlayTraits* concept, which is able to construct records of the *ResDcel* class on the basis of the *Dcel1* and *Dcel2* records that induce them. The function also constructs a consolidated set of curves that induce *res*.

See Also

OverlayTraits(page [1230](#))

OverlayTraits

A model for the OverlayTraits should be able to operate on records (namely, vertices, halfedges and faces) of two input DCEL classes, named *Dcel_A* and *Dcel_B*, and construct the records of an output DCEL class, referred to as *Dcel_R*.

Models for the concept are used by the global *overlay()* function to maintain the auxiliary data stored with the DCEL records of the resulting overlaid arrangement, based on the contents of the input records.

Types

<i>OverlayTraits:: Vertex_handle_A</i>	a constant handle a vertex in <i>Dcel_A</i> .
<i>OverlayTraits:: Halfedge_handle_A</i>	a constant handle to a halfedge in <i>Dcel_A</i> .
<i>OverlayTraits:: Face_handle_A</i>	a constant handle to a face <i>Dcel_A</i> .
<i>OverlayTraits:: Vertex_handle_B</i>	a constant handle to a vertex in <i>Dcel_B</i> .
<i>OverlayTraits:: Halfedge_handle_B</i>	a constant handle to a halfedge in <i>Dcel_B</i> .
<i>OverlayTraits:: Face_handle_B</i>	a constant handle to a face in <i>Dcel_B</i> .
<i>OverlayTraits:: Vertex_handle_R</i>	a handle to a vertex in <i>Dcel_R</i> .
<i>OverlayTraits:: Halfedge_handle_R</i>	a handle to a halfedge in <i>Dcel_R</i> .
<i>OverlayTraits:: Face_handle_R</i>	a handle to a faces in <i>Dcel_R</i> .

Functions

Whenever a vertex in the overlaid arrangement is created, one of the following functions is called in order to attach the appropriate auxiliary data to this vertex:

<i>void</i>	<i>ovl_tr.create_vertex(Vertex_handle_A v1, Vertex_handle_B v2, Vertex_handle_R v)</i>	constructs the vertex <i>v</i> induced by the coinciding vertices <i>v1</i> and <i>v2</i> .
<i>void</i>	<i>ovl_tr.create_vertex(Vertex_handle_A v1, Halfedge_handle_B e2, Vertex_handle_R v)</i>	constructs the vertex <i>v</i> induced by the vertex <i>v1</i> that lies on the halfedge <i>e2</i> .
<i>void</i>	<i>ovl_tr.create_vertex(Vertex_handle_A v1, Face_handle_B f2, Vertex_handle_R v)</i>	constructs the vertex <i>v</i> induced by the vertex <i>v1</i> that lies inside the face <i>f2</i> .
<i>void</i>	<i>ovl_tr.create_vertex(Halfedge_handle_A e1, Vertex_handle_B v2, Vertex_handle_R v)</i>	constructs the vertex <i>v</i> induced by the vertex <i>v2</i> that lies on the halfedge <i>e1</i> .

void *ovl_tr.create_vertex(Face_handle_A f1, Vertex_handle_B v2, Vertex_handle_R v)*

constructs the vertex v induced by the vertex $v2$ that lies inside the face $f1$.

void *ovl_tr.create_vertex(Halfedge_handle_A e1, Halfedge_handle_B e2, Vertex_handle_R v)*

constructs the vertex v induced by the intersection of the halfedges $e1$ and $e2$.

Whenever an edge in the overlaid arrangement is created, one of the following functions is called in order to attach the appropriate auxiliary data to this vertex. Note that an edge is created after both its end-vertices are created, (and the corresponding *create_vertex()* methods were invoked). In all cases, the edge is represented by a halfedge e directed in lexicographic decreasing order (from right to left). The *create_edge()* method should attach auxiliary data to the twin halfedge (namely to $e \rightarrow \text{twin}()$) as well:

void *ovl_tr.create_edge(Halfedge_handle_A e1, Halfedge_handle_B e2, Halfedge_handle_R e)*

constructs the halfedge e induced by an overlap between the halfedges $e1$ and $e2$.

void *ovl_tr.create_edge(Halfedge_handle_A e1, Face_handle_B f2, Halfedge_handle_R e)*

constructs the halfedge e induced by the halfedge $e1$ that lies inside the face $f2$.

void *ovl_tr.create_edge(Face_handle_A f1, Halfedge_handle_B e2, Halfedge_handle_R e)*

constructs the halfedge e induced by the halfedge $e2$ that lies inside the face $f1$.

The following function is invoked whenever a new face is created. It is guaranteed that all halfedges along the face boundary have already been created and have their auxiliary data fields attached to them:

void *ovl_tr.create_face(Face_handle_A f1, Face_handle_B f2, Face_handle_R f)*

constructs the face f induced by the an overlap between the faces $f1$ and $f2$.

Has Models

Arr_default_overlay_traits<Arrangement> (page [1232](#))

Arr_face_overlay_traits<Arr1, Arr2, ResArr, OvlFaceData> (page [1233](#))

See Also

overlay (page [1229](#))

CGAL::Arr_default_overlay_traits<Arrangement>

Definition

An instance of *Arr_default_overlay_traits*<Arrangement> should be used for overlaying two arrangements of type *Arrangement* that store no auxiliary data with their DCEL records, where the resulting overlaid arrangement stores no auxiliary DCEL data as well. This class simply gives empty implementation for all traits-class functions.

```
#include <CGAL/Arr_default_overlay_traits.h>
```

Is Model for the Concepts

OverlayTraits

See Also

overlay (page [1229](#))

CGAL::Arr_face_overlay_traits<Arr_A,Arr_B,Arr_R,OvlFaceData>

Definition

An instance of *Arr_face_overlay_traits*<Arr_A,Arr_B,Arr_R,OvlFaceData> should be used for overlaying two arrangements of types Arr_A and Arr_B, which are instantiated using the same geometric traits-class and with the DCEL classes *Dcel_A* and *Dcel_B* respectively, in order to store their overlay in an arrangement of type Arr_R, which is instantiated using a third DCEL class *Dcel_R*. All three DCEL classes are assumed to be instantiations of the *Arr_face_extended_dcel* template with types *FaceData_A*, *FaceData_B* and *FaceData_R*, respectively.

This class gives empty implementation for all overlay traits-class functions, except the function that computes the overlay of two faces. In this case, it uses the functor *OvlFaceData*, which accepts a *FaceData_A* object and a *FaceData_B* object and computes a corresponding *FaceData_R* object, in order to set the auxiliary data of the overlay face.

```
#include <CGAL/Arr_default_overlay_traits.h>
```

Is Model for the Concepts

OverlayTraits

See Also

overlay (page [1229](#))

CGAL::Arr_face_extended_dcel<Traits,FData,V,H,F> (page [1251](#))

CGAL::read

```
#include <CGAL/IO/Arr_iostream.h>
```

```
template<class Traits, class Dcel, class Formatter>
std::istream& read( Arrangement_2<Traits,Dcel>& arr, std::istream& is, Formatter& formatter)
```

Reads the arrangement *arr* from the given input stream using a specific input format. *formatter*, which must be a model of the *ArrangementInputFormatter*, defines the input format.

```
template<class Traits, class Dcel>
std::istream& std::ostream& is >> Arrangement_2<Traits,Dcel>& arr
```

Reads the arrangement *arr* from the given input stream using the input format defined by the *Arr_text_formatter* class — that is, only the basic geometric and topological features of the arrangement are read and no auxiliary data is attached to the DCEL features.

```
#include <CGAL/IO/Arr_with_history_iostream.h>
```

```
template<class Traits, class Dcel, class Formatter>
std::istream& read( Arrangement_with_history_2<Traits,Dcel>& arr,
std::istream& is,
Formatter& formatter)
```

Reads the arrangement-with-history instance *arr* from the given input stream using a specific input format. *formatter*, which must be a model of the *ArrWithHistoryInputFormatter*, defines the input format.

```
template<class Traits, class Dcel>
std::istream& std::ostream& is >> Arrangement_with_history_2<Traits,Dcel>& arr
```

Reads the arrangement-with-history instance *arr* from the given input stream using the default input format.

See Also

[write](#)(page 1235)

CGAL::write

```
#include <CGAL/IO/Arr_iostream.h>
```

```
template<class Traits, class Dcel, class Formatter>
std::ostream& write( Arrangement_2<Traits,Dcel> arr, std::ostream& os, Formatter& formatter)
```

Writes the arrangement *arr* into the given output stream using a specific output format. *formatter*, which must be a model of the *ArrangementOutputFormatter*, defines the output format.

```
template<class Traits, class Dcel>
std::ostream& std::ostream& os << Arrangement_2<Traits,Dcel> arr
```

Writes the arrangement *arr* into the given output stream using the output format defined by the *Arr_text_formatter* class — that is, only the basic geometric and topological features of the arrangement are written, without any auxiliary data that may be attached to the DCEL features.

```
#include <CGAL/IO/Arr_with_history_iostream.h>
```

```
template<class Traits, class Dcel, class Formatter>
std::ostream& write( Arrangement_with_history_2<Traits,Dcel> arr,
                    std::ostream& os,
                    Formatter& formatter)
```

Writes the arrangement-with-history instance *arr* into the given output stream using a specific output format. *formatter*, which must be a model of the *ArrWithHistoryOutputFormatter*, defines the output format.

```
template<class Traits, class Dcel>
std::ostream& std::ostream& os << Arrangement_with_history_2<Traits,Dcel> arr
```

Writes the arrangement-with-history instance *arr* into the given output stream using the default output format.

See Also

read(page [1234](#))

ArrangementDcel

A doubly-connected edge-list (DCEL for short) data-structure. It consists of three containers of records: vertices V , halfedges E , and faces F . It maintains the incidence relation among them. The halfedges are ordered in pairs sometimes referred to as twins, such that each halfedge pair represent an edge.

A model of the ArrangementDcel concept must provide the following types and operations. (In addition to the requirements here, the local types *Vertex*, *Halfedge*, *Face* *Hole* and *Isolated_vertex* must be models of the concepts *ArrangementDcelVertex* (page 1239), *ArrangementDcelHalfedge* (page 1241), *ArrangementDcelFace* (page 1244), *ArrangementDcelHole* (page 1246) and *ArrangementDcelIsolatedVertex* (page 1247) respectively.)

Types

<i>ArrangementDcel:: Vertex</i>	the vertex type.
<i>ArrangementDcel:: Halfedge</i>	the halfedge type.
<i>ArrangementDcel:: Face</i>	the face type.
<i>ArrangementDcel:: Hole</i>	the hole type.
<i>ArrangementDcel:: Isolated_vertex</i>	the isolated vertex type.
<i>ArrangementDcel:: Size</i>	used to represent size values (e.g., <i>size_t</i>).
<i>ArrangementDcel:: Vertex_iterator</i>	a bidirectional iterator over the vertices. Its value-type is <i>Vertex</i> .
<i>ArrangementDcel:: Halfedge_iterator</i>	a bidirectional iterator over the halfedges. Its value-type is <i>Halfedge</i> .
<i>ArrangementDcel:: Face_iterator</i>	a bidirectional iterator over the faces. Its value-type is <i>Face</i> .

The non-mutable iterators *Vertex_const_iterator*, *Halfedge_const_iterator* and *Face_const_iterator* are also defined.

Creation

<i>ArrangementDcel dcel;</i>	constructs an empty DCEL with one unbounded face.
<i>Face*</i> <i>dcel.assign(Self other, const Face *uf)</i>	assigns the contents of the <i>other</i> DCEL whose unbounded face is given by <i>uf</i> , to <i>dcel</i> . The function returns a pointer to the unbounded face of <i>dcel</i> after the assignment.

Access Functions

<i>Size</i>	<i>dcel.size_of_vertices()</i>	returns the number of vertices.
<i>Size</i>	<i>dcel.size_of_halfedges()</i>	returns the number of halfedges (always even).
<i>Size</i>	<i>dcel.size_of_faces()</i>	returns the number of faces.

<i>Size</i>	<i>dcel.size_of_holes()</i>	returns the number of holes (the number of connected components).
<i>Size</i>	<i>dcel.size_of_isolated_vertices()</i>	returns the number of isolated vertices.

The following operations have an equivalent *const* operations that return the corresponding non-mutable iterators:

<i>Vertex_iterator</i>	<i>dcel.vertices_begin()</i>	returns a begin-iterator of the vertices in <i>dcel</i> .
<i>Vertex_iterator</i>	<i>dcel.vertices_end()</i>	returns a past-the-end iterator of the vertices in <i>dcel</i> .
<i>Halfedge_iterator</i>	<i>dcel.halfedges_begin()</i>	returns a begin-iterator of the halfedges in <i>dcel</i> .
<i>Halfedge_iterator</i>	<i>dcel.halfedges_end()</i>	returns a past-the-end iterator of the halfedges in <i>dcel</i> .
<i>Vertex_iterator</i>	<i>dcel.faces_begin()</i>	returns a begin-iterator of the faces in <i>dcel</i> .
<i>Vertex_iterator</i>	<i>dcel.faces_end()</i>	returns a past-the-end iterator of the faces in <i>dcel</i> .

Modifiers

The following operations allocate a new element of the respective type. Halfedges are always allocated in pairs of opposite halfedges. The and their opposite pointers are automatically set.

<i>Vertex*</i>	<i>dcel.new_vertex()</i>	creates a new vertex.
<i>Halfedge*</i>	<i>dcel.new_edge()</i>	creates a new pair of twin halfedges.
<i>Face*</i>	<i>dcel.new_face()</i>	creates a new face.
<i>Hole*</i>	<i>dcel.new_hole()</i>	creates a new hole record.
<i>Isolated_vertex*</i>	<i>dcel.new_isolated_vertex()</i>	creates a new isolated vertex record.
<i>void</i>	<i>dcel.delete_vertex(Vertex* v)</i>	deletes the vertex <i>v</i> .
<i>void</i>	<i>dcel.delete_edge(Halfedge* e)</i>	deletes the halfedge <i>e</i> as well as its twin.
<i>void</i>	<i>dcel.delete_face(Face* f)</i>	deletes the face <i>f</i> .
<i>void</i>	<i>dcel.delete_hole(Hole* ho)</i>	deletes the hole <i>ho</i> .
<i>void</i>	<i>dcel.delete_isolated_vertex(Isolated_vertex* iv)</i>	deletes the isolated vertex <i>iv</i> .

Has Models

Arr_dcel_base<*V,H,F*> (page [1248](#))

Arr_default_dcel<*Traits*> (page [1250](#))

Arr_face_extended_dcel<*Traits,FData,V,H,F*> (page [1251](#))

Arr_extended_dcel<*Traits,VData,HData,FData,V,H,F*> (page [1252](#))

See Also

ArrangementDcelVertex (page [1239](#))

ArrangementDcelHalfedge (page [1241](#))

ArrangementDcelFace (page [1244](#))

ArrangementDcelHole (page [1246](#))

ArrangementDcelIsolatedVertex (page [1247](#))

ArrangementDcelVertex

Definition

A vertex record in a DCEL data structure. A vertex is always associated with a point. However, the vertex record only stores a pointer to the associated point, and the actual *Point* object is stored elsewhere.

A vertex usually has several halfedges incident to it, such that it is possible to access one of these halfedges directly and to traverse all incident halfedges around the vertex. However, the DCEL may also contain isolated vertices that have no incident halfedges. In this case, the vertex stores an isolated vertex-information record, indicating the face that contains this vertex in its interior.

Types

<i>ArrangementDcelVertex::Halfedge</i>	the corresponding DCEL halfedge type.
<i>ArrangementDcelVertex::Isolated_vertex</i>	the corresponding DCEL isolated vertex-information type.
<i>ArrangementDcelVertex::Point</i>	the point type associated with the vertex.

Creation

<i>ArrangementDcelVertex</i> <i>v</i> ;	default constructor.
<i>void</i> <i>v.assign(Self other)</i>	assigns <i>v</i> with the contents of the <i>other</i> vertex.

Access Functions

All functions below also have *const* counterparts, returning non-mutable pointers or references:

<i>bool</i> <i>v.is_isolated()</i>	returns whether the vertex is isolated (has no incident halfedges).
<i>Halfedge*</i> <i>v.halfedge()</i>	returns an incident halfedge that has <i>v</i> as its target. <i>Precondition:</i> <i>v</i> is <i>not</i> an isolated vertex.
<i>Isolated_vertex*</i> <i>v.isolated_vertex()</i>	returns the isolated vertex-information record. <i>Precondition:</i> <i>v</i> is an isolated vertex.
<i>Point&</i> <i>v.point()</i>	returns the associated point.

Modifiers

<i>void</i> <i>v.set_halfedge(Halfedge* e)</i>	sets the incident halfedge, marking <i>v</i> as a regular vertex.
-------------------------------------------------	-------------------------------------------------------------------

void *v.set_isolated_vertex(Isolated_vertex* iv)*

sets the isolated vertex-information record, marking *v* as an isolated vertex.

void *v.set_point(Point* p)* sets the associated point.

See Also

ArrangementDcel (page [1236](#))

ArrangementDcelHalfedge (page [1241](#))

ArrangementDcellIsolatedVertex (page [1247](#))

ArrangementDcelHalfedge

Definition

A halfedge record in a DCEL data structure. Two halfedges with opposite directions always form an edge (a halfedge pair). The halfedges form together chains, defining the boundaries of connected components, such that all halfedges along a chain have the same incident face. Note that the chain the halfedge belongs to may form the outer boundary of a bounded face (an outer CCB) or the boundary of a hole inside a face (an inner CCB).

An edge is always associated with a curve, but the halfedge records only store a pointer to the associated curve, and the actual curve objects are stored elsewhere. Two opposite halfedges are always associated with the same curve.

Types

<i>ArrangementDcelHalfedge::Vertex</i>	the corresponding DCEL halfedge type.
<i>ArrangementDcelHalfedge::Face</i>	the corresponding DCEL face type.
<i>ArrangementDcelHalfedge::Hole</i>	the corresponding DCEL hole type.

<i>ArrangementDcelHalfedge::X_monotone_curve</i>	the curve type associated with the edge.
--------------------------------------------------	------------------------------------------

Creation

<i>ArrangementDcelHalfedge e;</i>	default constructor.
<i>void e.assign(Self other)</i>	assigns <i>e</i> with the contents of the <i>other</i> halfedge.

Access Functions

<i>Comparison_result e.direction()</i>	returns <i>SMALLER</i> if <i>e</i> 's source vertex is lexicographically smaller than it target, and <i>LARGER</i> if it is lexicographically larger than the target.
<i>bool e.is_on_hole()</i>	determines whether the <i>e</i> lies on an outer CCB of a bounded face, or on an inner CCB (a hole inside a face). The function returns <i>true</i> if <i>e</i> lies on a hole.

All functions below also have *const* counterparts, returning non-mutable pointers or references:

<i>Halfedge* e.opposite()</i>	returns the twin halfedge.
<i>Halfedge* e.prev()</i>	returns the previous halfedge along the chain.
<i>Halfedge* e.next()</i>	returns the next halfedge along the chain.

<i>Vertex*</i>	<i>e.vertex()</i>	returns the target vertex.
<i>Face*</i>	<i>e.face()</i>	returns the incident face. <i>Precondition:</i> <i>e</i> lies on the outer boundary of this face.
<i>Hole*</i>	<i>e.hole()</i>	returns the hole (inner CCB) <i>e</i> belongs to. <i>Precondition:</i> <i>e</i> lies on a hole inside its incident face.
<i>X_monotone_curve&</i>	<i>e.curve()</i>	returns the associated curve.

Modifiers

<i>void</i>	<i>e.set_opposite(Halfedge* opp)</i>	sets the opposite halfedge.
<i>void</i>	<i>e.set_direction(Comparison_result dir)</i>	sets the lexicographical order between <i>e</i> 's source and target vertices to be <i>dir</i> . The direction of the opposite halfedge is also set to the opposite direction. <i>Precondition:</i> <i>dir</i> is either <i>SMALLER</i> or <i>LARGER</i> (and not <i>EQUAL</i>).
<i>void</i>	<i>e.set_prev(Halfedge* prev)</i>	sets the previous halfedge of <i>e</i> along the chain, and updates the cross-pointer <i>prev->next()</i> .
<i>void</i>	<i>e.set_next(Halfedge* next)</i>	sets the next halfedge of <i>e</i> along the chain, and updates the cross-pointer <i>next->prev()</i> .
<i>void</i>	<i>e.set_vertex(Vertex* v)</i>	sets the target vertex.
<i>void</i>	<i>e.set_face(Face* f)</i>	sets the incident face, marking that <i>e</i> lies on the outer CCB of the face <i>f</i> .
<i>void</i>	<i>e.set_hole(Hole* ho)</i>	sets the incident hole, marking that <i>e</i> lies on an inner CCB.
<i>void</i>	<i>e.set_curve(X_monotone_curve* c)</i>	sets the associated curve of <i>e</i> and its opposite halfedge.

See Also

ArrangementDcel (page [1236](#))

ArrangementDcelVertex (page [1239](#))

ArrangementDcelFace (page [1244](#))

ArrangementDcelHole (page [1246](#))

ArrangementDcelFace

Definition

A face record in a DCEL data structure. A face may either be unbounded, otherwise it has an incident halfedge along the chain defining its outer boundary. A face may also contain holes and isolated vertices in its interior.

Types

<i>ArrangementDcelFace:: Vertex</i>	the corresponding DCEL vertex type.
<i>ArrangementDcelFace:: Halfedge</i>	the corresponding DCEL halfedge type.
<i>ArrangementDcelFace:: Hole_iterator</i>	a bidirectional iterator over the holes in inside the face. Its value-type is <i>Halfedge*</i> .
<i>ArrangementDcelFace:: Isolated_vertex_iterator</i>	a bidirectional iterator over the isolated vertices in inside the face. Its value-type is <i>Vertex*</i> .

The non-mutable iterators *Hole_const_iterator*, and *Isolated_vertex_const_iterator* are also defined.

Creation

<i>ArrangementDcelFace f;</i>	default constructor.
<i>void f.assign(Self other)</i>	assigns <i>f</i> with the contents of the <i>other</i> face.

Access Functions

All functions below also have *const* counterparts, returning non-mutable pointers or iterators:

<i>Halfedge*</i>	<i>f.halfedge()</i>	returns an incident halfedge along the outer boundary of the face. If <i>f</i> is the unbounded face, the function returns <i>NULL</i> .
<i>size_t</i>	<i>f.number_of_holes()</i>	returns the number of holes inside <i>f</i> .
<i>Hole_iterator</i>	<i>f.holes_begin()</i>	returns a begin-iterator for the holes inside <i>f</i> .
<i>Hole_iterator</i>	<i>f.holes_end()</i>	returns a past-the-end iterator for the holes inside <i>f</i> .
<i>size_t</i>	<i>f.number_of_isolated_vertices()</i>	returns the number of isolated vertices inside <i>f</i> .
<i>Isolated_vertex_iterator</i>	<i>f.isolated_vertices_begin()</i>	returns a begin-iterator for the isolated vertices inside <i>f</i> .

Isolated_vertex_iterator

f.isolated_vertices_end()

returns a past-the-end iterator for the isolated vertices inside *f*.

Modifiers

void *f.set_halfedge(Halfedge* e)*

sets the incident halfedge.

void *f.add_hole(Halfedge* e)*

adds *e* as a hole inside *f*.

void *f.erase_hole(Hole_iterator it)*

removes the hole that *it* points to from inside *f*.

void *f.add_isolated_vertex(Vertex* v)*

adds *v* as an isolated vertex inside *f*.

void *f.erase_isolated_vertex(Isolated_vertex_iterator it)*

removes the isolated vertex that *it* points to from inside *f*.

See Also

ArrangementDcel (page [1236](#))

ArrangementDcelVertex (page [1239](#))

ArrangementDcelHalfedge (page [1241](#))

ArrangementDcelHole

Definition

A hole record in a DCEL data structure, which stores the face that contains the hole in its interior, along with an iterator for the hole in the holes' container of this face.

Types

ArrangementDcelHole::Face the corresponding DCEL face type.

typedef Face::Hole_iterator

Hole_iterator;

Creation

ArrangementDcelHole ho; default constructor.

Access Functions

All functions below also have *const* counterparts, returning non-mutable pointers or iterators:

*Face** *ho.face()* returns the incident face, which contains *ho* in its interior.

Hole_iterator *ho.iterator()* returns an iterator for the hole.

Modifiers

void *ho.set_face(Face* f)* sets the incident face.

void *ho.set_iterator(Hole_iterator it)*
sets the hole iterator.

See Also

ArrangementDcel (page [1236](#))

ArrangementDcelFace (page [1244](#))

ArrangementDcellIsolatedVertex

Definition

An isolated vertex-information record in a DCEL data structure, which stores the face that contains the isolated vertex in its interior, along with an iterator for the isolated vertex in the isolated vertices' container of this face.

Types

ArrangementDcellIsolatedVertex::Face the corresponding DCEL face type.

```
typedef Face::Isolated_vertex_iterator
        Isolated_vertex_iterator;
```

Creation

ArrangementDcellIsolatedVertex iv; default constructor.

Access Functions

All functions below also have *const* counterparts, returning non-mutable pointers or iterators:

*Face** *iv.face()* returns the incident face, which contains *iv* in its interior.

Isolated_vertex_iterator
 iv.iterator() returns an iterator for the isolated vertex.

Modifiers

void *iv.set_face(Face* f)* sets the incident face.

void *iv.set_iterator(Isolated_vertex_iterator it)*
 sets the isolated vertex iterator.

See Also

ArrangementDcel (page [1236](#))
ArrangementDcelFace (page [1244](#))

CGAL::Arr_dcel_base<V,H,F>

Definition

The *Arr_dcel_base*<*V,H,F*> class is an important ingredient in the definition of DCEL data structures. It serves as a basis class for any instance of the *Dcel* template parameter of the *Arrangement_2* template. In particular it is the basis class of the default *Dcel* template parameter, and the basis class of any extended DCEL. The template parameters *V*, *H*, and *F* must be instantiated with models of the concepts *ArrangementDcelVertex*, *ArrangementDcelHalfedge*, and *ArrangementDcelFace* respectively.

```
#include <CGAL/Arr_dcel_base.h>
```

Is Model for the Concepts

ArrangementDcel

Class Arr_vertex_base<Point>

Definition

The basic DCEL vertex type. Serves as a basis class for an extended vertex record with auxiliary data fields. The *Point* parameter is the type of points associated with the vertices.

Is Model for the Concepts

ArrangementDcelVertex

Class Arr_halfedge_base<Curve>

Definition

The basic DCEL halfedge type. Serves as a basis class for an extended halfedge record with auxiliary data fields. The *Curve* parameter is the type of *x*-monotone curves associated with the vertices.

Is Model for the Concepts

ArrangementDcelHalfedge

Class Arr_face_base

Definition

The basic DCEL face type. Serves as a basis class for an extended face record with auxiliary data fields.

Is Model for the Concepts

ArrangementDcelFace

CGAL::Arr_default_dcel<Traits>

Definition

The default DCEL class used by the *Arrangement_2* class-template is parameterized by a traits class, which is a model of the *ArrangementBasicTraits_2* concept. It simply uses the nested *Traits::Point_2* and *Traits::X_monotone_curve_2* to instantiate the base vertex and halfedge types, respectively. Thus, the default DCEL records store no other information, except for the topological incidence relations and the geometric data attached to vertices and edges.

```
#include <CGAL/Arr_default_dcel.h>
```

Is Model for the Concepts

ArrangementDcel

Inherits From

```
Arr_dcel_base<Arr_vertex_base<typename Traits::Point_2>,
              Arr_halfedge_base<typename Traits::X_monotone_curve_2>,
              Arr_face_base>
```

Types

```
Arr_default_dcel<Traits>:: template <class T> rebind
```

allows the rebinding of the DCEL with a different traits class *T*.

See Also

Arr_dcel_base<*V,H,F*> (page [1248](#))

CGAL::Arr_face_extended_dcel<Traits,FData,V,H,F>

Definition

The *Arr_face_extended_dcel<Traits,FData,V,H,F>* class-template extends the DCEL face-records, making it possible to store extra (non-geometric) data with the arrangement faces. The class should be instantiated by an *FData* type which represents the extra data stored with each face.

Note that all types of DCEL features (namely vertex, halfedge and face) are provided as template parameters. However, by default they are defined as follows:

```
V = Arr_vertex_base<typename Traits::Point_2>
H = Arr_halfedge_base<typename Traits::X_monotone_curve_2>
F = Arr_face_base
```

```
#include <CGAL/Arr_extended_dcel.h>
```

Is Model for the Concepts

ArrangementDcel

Inherits From

Arr_dcel_base<V, H, Arr_extended_face<F, FData> >

Types

Arr_face_extended_dcel<Traits,FData,V,H,F>:: template <class T> rebind

allows the rebinding of the DCEL with a different traits class *T*.

See Also

Arr_dcel_base<V,H,F> (page [1248](#))

CGAL::Arr_extended_dcel<Traits,VData,HData,FData,V,H,F>

Definition

The *Arr_extended_dcel<Traits,VData,HData,FData,V,H,F>* class-template extends the topological-features of the DCEL namely the vertex, halfedge, and face types. While it is possible to maintain extra (non-geometric) data with the curves or points of the arrangement by extending their types respectively, it is also possible to extend the vertex, halfedge, or face types of the DCEL through inheritance. As the technique to extend these types is somewhat cumbersome and difficult for inexperienced users, the *Arr_extended_dcel<Traits,VData,HData,FData,V,H,F>* class-template provides a convenient way to do that. Each one of the three features is extended with a corresponding data type provided as parameters. This class template is also parameterized with a traits class used to extract default values for the vertex, halfedge, and face base classes, which are the remaining three template parameters respectively. The default values follow:

```
V = Arr_vertex_base<typename Traits::Point_2>
H = Arr_halfedge_base<typename Traits::X_monotone_curve_2>
F = Arr_face_base
```

```
#include <CGAL/Arr_extended_dcel.h>
```

Is Model for the Concepts

ArrangementDcel

Inherits From

```
Arr_dcel_base<Arr_extended_vertex<V, VData>,
              Arr_extended_halfedge<H, HData>,
              Arr_extended_face<F, FData> >
```

Types

```
Arr_extended_dcel<Traits,VData,HData,FData,V,H,F>:: template <class T> rebind
```

allows the rebinding of the DCEL with a different traits class *T*.

See Also

Arr_dcel_base<V,H,F> (page [1248](#))

CGAL::Arr_extended_vertex<VertexBase,VData>

Definition

The *Arr_extended_vertex<VertexBase,VData>* class-template extends the vertex topological-features of the DCEL. It is parameterized by a vertex base-type *VertexBase* and a data type *VData* used to extend the vertex base-type.

```
#include <CGAL/Arr_extended_dcel.h>
```

Is Model for the Concepts

ArrangementDcelVertex

Inherits From

VertexBase

Creation

void *v.assign(Self other)* assigns *v* with the contents of the *other* vertex.

Access Functions

VData *v.data()* obtains the auxiliary data (a non-const version, returning a reference to a mutable data object is also available).

Modifiers

void *v.set_data(VData data)* sets the auxiliary data.

See Also

Arr_dcel_base<V,H,F> (page [1248](#))

CGAL::Arr_extended_halfedge<HalfedgeBase,HData>

Definition

The *Arr_extended_halfedge*<*HalfedgeBase*,*HData*> class-template extends the halfedge topological-features of the DCEL. It is parameterized by a halfedge base-type *HalfedgeBase* and a data type *HData* used to extend the halfedge base-type.

```
#include <CGAL/Arr_extended_dcel.h>
```

Is Model for the Concepts

ArrangementDcelHalfedge

Inherits From

HalfedgeBase

Creation

void *he.assign(Self other)* assigns *he* with the contents of the *other* vertex.

Access Functions

HData *he.data()* obtains the auxiliary data (a non-const version, returning a reference to a mutable data object is also available).

Modifiers

void *he.set_data(HData data)* sets the auxiliary data.

See Also

Arr_dcel_base<*V,H,F*> (page [1248](#))

CGAL::Arr_extended_face<FaceBase,FData>

Definition

The *Arr_extended_face<FaceBase,FData>* class-template extends the face topological-features of the DCEL. It is parameterized by a face base-type *FaceBase* and a data type *FData* used to extend the face base-type.

```
#include <CGAL/Arr_extended_dcel.h>
```

Is Model for the Concepts

ArrangementDcelFace

Inherits From

FaceBase

Creation

void *f.assign(Self other)* assigns *f* with the contents of the *other* vertex.

Access Functions

FData *f.data()* obtains the auxiliary data (a non-const version, returning a reference to a mutable data object is also available).

Modifiers

void *f.set_data(FData data)* sets the auxiliary data.

See Also

Arr_dcel_base<V,H,F> (page [1248](#))

ArrangementBasicTraits_2

Definition

The basic arrangement-traits concept defines the minimal set of geometric predicates needed for the construction and maintenance of instances of the *Arrangement_2* class, as well as performing simple queries (such as point-location queries) on such arrangements.

A model of this concept must define nested *Point_2* and *X_monotone_curve_2* types, which represent planar points and *x*-monotone curves (a vertical segment is also considered to be *weakly x-monotone*), respectively. The *x*-monotone curves are assumed to be pairwise disjoint in their interiors, so they do not intersect, and their endpoints are representable as *Point_2* objects.

Types

ArrangementBasicTraits_2::Point_2 represents a point on the plane.

ArrangementBasicTraits_2::X_monotone_curve_2
represents a planar (weakly) *x*-monotone curve.

Tags

ArrangementBasicTraits_2::Has_left_category
indicates whether the nested functor *Compare_at_x_left_2* is provided.

Functor Types

ArrangementBasicTraits_2::Compare_x_2
provides the operator :
Comparison_result operator() (Point_2 p1, Point_2 p2)
which returns *SMALLER*, *EQUAL* or *LARGER* according to the *x*-ordering of points *p1* and *p2*.

ArrangementBasicTraits_2::Compare_xy_2
provides the operator :
Comparison_result operator() (Point_2 p1, Point_2 p2)
which returns *SMALLER*, *EQUAL*, or *LARGER* according to the lexicographic *xy*-order of the points *p1* and *p2*.

ArrangementBasicTraits_2:: Construct_min_vertex_2

provides the operator :

Point_2 operator() (X_monotone_curve_2 c)

which returns the lexicographically smaller (left) endpoint of *c*.

ArrangementBasicTraits_2:: Construct_max_vertex_2

provides the operator :

Point_2 operator() (X_monotone_curve_2 c)

which returns the lexicographically larger (right) endpoint of *c*.

ArrangementBasicTraits_2:: Is_vertical_2

provides the operator :

bool operator() (X_monotone_curve_2 c)

which determines whether *c* is a vertical segment.

ArrangementBasicTraits_2:: Compare_y_at_x_2

provides the operator :

Comparison_result operator() (Point_2 p, X_monotone_curve_2 c)

which compares the y-coordinates of *p* and the vertical projection of *p* on *c*, and returns *SMALLER*, *EQUAL* or *LARGER* according to the result.

ArrangementBasicTraits_2:: Compare_y_at_x_left_2

provides the operator :

Comparison_result operator() (X_monotone_curve_2 c1, X_monotone_curve_2 c2, Point_2 p)

which accepts two *x*-monotone curves *c1* and *c2* that have a common right endpoint *p*, and returns *SMALLER*, *EQUAL* or *LARGER* according to the relative position of the two curves immediately to the left of *p*. Note that in case one of the *x*-monotone curves is a vertical segment (emanating downward from *p*), it is always considered to be below the other curve.

ArrangementBasicTraits_2:: Compare_y_at_x_right_2

provides the operator :

Comparison_result operator() (X_monotone_curve_2 c1, X_monotone_curve_2 c2, Point_2 p)

which accepts two *x*-monotone curves *c1* and *c2* that have a common left endpoint *p*, and returns *SMALLER*, *EQUAL* or *LARGER* according to the relative position of the two curves immediately to the right of *p*. Note that in case one of the *x*-monotone curves is a vertical segment emanating upward from *p*, it is always considered to be above the other curve.

ArrangementBasicTraits_2:: Equal_2

provides the operators :

bool operator() (Point_2 p1, Point_2 p2)

which determines whether *p1* and *p2* are geometrically equivalent; and :

bool operator() (X_monotone_curve_2 c1, X_monotone_curve_2 c2)

which determines whether *c1* and *c2* are geometrically equivalent (have the same graph).

Creation

ArrangementBasicTraits_2 traits;

default constructor.

ArrangementBasicTraits_2 traits(other);

copy constructor

ArrangementBasicTraits_2 traits = other

assignment operator.

Accessing Functor Objects

Compare_x_2 traits.compare_x_2_object()

Compare_xy_2 traits.compare_xy_2_object()

Construct_min_vertex_2 traits.construct_min_vertex_2_object()

Construct_max_vertex_2 traits.construct_max_vertex_2_object()

Is_vertical_2 traits.is_vertical_2_object()

Compare_y_at_x_2 traits.compare_y_at_x_2_object()

Compare_y_at_x_left_2 traits.compare_y_at_x_left_2_object()

Compare_y_at_x_right_2 traits.compare_y_at_x_right_2_object()

Equal_2 traits.equal_2_object()

Has Models

CGAL::Arr_segment_traits_2<Kernel>

CGAL::Arr_non_caching_segment_basic_traits_2<Kernel>

CGAL::Arr_non_caching_segment_traits_2<Kernel>

CGAL::Arr_polyline_traits_2<SegmentTraits>

CGAL::Arr_circle_segment_traits_2<Kernel>

CGAL::Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>

CGAL::Arr_rational_arc_traits_2<AlgKernel,NtTraits>

CGAL::Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv>

CGAL::Arr_consolidated_curve_data_traits_2<Traits,Data>

ArrangementLandmarkTraits_2

Definition

The landmark-traits concept refines the basic traits concept by adding operations needed for the landmarks point-location strategy, namely — approximating points and connecting points with a simple x -monotone curve.

A model of this concept must define the *Approximate_number_type*, which is used to approximate the coordinates of *Point_2* instances. It is recommended to define the approximated number type as the built-in *double* type.

Refines

ArrangementBasicTraits_2

Types

ArrangementLandmarkTraits_2::Approximate_number_type

the number type used to approximate point coordinates.

Functor Types

ArrangementLandmarkTraits_2::Approximate_2

provides the operator :

Approximate_number_type operator() (Point_2 p, int i)

which returns an approximation of p 's x -coordinate (if $i == 0$), or of p 's y -coordinate (if $i == 1$).

ArrangementLandmarkTraits_2::Construct_x_monotone_curve_2

provides the operator :

X_monotone_curve_2 operator() (Point_2 p1, Point_2 p2)

returns an x -monotone curve connecting $p1$ and $p2$ (i.e., the two input points are its endpoints).

Creation

ArrangementLandmarkTraits_2 traits;

default constructor.

ArrangementLandmarkTraits_2 traits(other);

copy constructor.

ArrangementLandmarkTraits_2 traits = other

assignment operator.

Accessing Functor Objects

<i>Approximate_2</i>	<i>traits.approximate_2_object()</i>
<i>Construct_x_monotone_curve_2</i>	<i>traits.construct_x_monotone_curve_2_object()</i>

Has Models

CGAL::Arr_non_caching_segment_basic_traits_2<Kernel>
CGAL::Arr_segment_traits_2<Kernel>
CGAL::Arr_polyline_traits_2<SegmentTraits>
CGAL::Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>

See Also

ArrangementBasicTraits_2 (page [1256](#))

ArrangementXMonotoneTraits_2

Definition

This concept refines the basic arrangement-traits concept. A model of this concept is able to handle x -monotone curves that intersect in their interior (and points that coincide with curve interiors). This is necessary for constructing arrangements of sets of intersecting x -monotone curves.

As the resulting structure, represented by the *Arrangement_2* class, stores pairwise interior-disjoint curves, the input curves are split at the intersection points before being inserted into the arrangement. A model of this refined concept therefore needs to compute the intersections (and possibly overlaps) between two x -monotone curves and to support curve splitting. The reverse merge operation is optionally supported.

Refines

ArrangementBasicTraits_2

Tags

ArrangementXMonotoneTraits_2::Has_merge_category indicates whether the nested functors *Are_mergeable_2* and *Merge_2* are provided.

Types

ArrangementXMonotoneTraits_2::Multiplicity the multiplicity type.

Functor Types

ArrangementXMonotoneTraits_2::Intersect_2

provides the operator (templated by the *OutputIterator* type) :

OutputIterator operator() (*X_monotone_curve_2* *c1*, *X_monotone_curve_2* *c2*, *OutputIterator* *oi*)

which computes the intersections of *c1* and *c2* and inserts them in an ascending lexicographic xy -order into the output iterator. The value-type of *OutputIterator* is *CGAL::Object*, where each *Object* either wraps a *pair<Point_2, Multiplicity>* instance, which represents an intersection point with its multiplicity (in case the multiplicity is undefined or not known, it should be set to 0) or an *X_monotone_curve_2* instance, representing an overlapping subcurve of *c1* and *c2*. The operator returns a past-the-end iterator for the output sequence.

ArrangementXMonotoneTraits_2::Split_2

provides the operator :

void operator() (*X_monotone_curve_2* *c*, *Point_2* *p*, *X_monotone_curve_2*& *c1*, *X_monotone_curve_2*& *c2*)

which accepts an input curve *c* and a split point *p* in its interior. It splits *c* at the split point into two subcurves *c1* and *c2*, such that *p* is *c1*'s right endpoint and *c2*'s left endpoint.

The two following functor types are optional. If they are supported, the *Has_merge_category* tag should be defined as *Tag_true* (and *Tag_false* otherwise):

ArrangementXMonotoneTraits_2::Are_mergeable_2

provides the operator :

bool operator() (X_monotone_curve_2 c1, X_monotone_curve_2 c2)

which accepts two *x*-monotone curves *c1* and *c2* that share a common endpoint, and determines whether they can be merged to form a single continuous *x*-monotone curve.

ArrangementXMonotoneTraits_2::Merge_2

provides the operator :

void operator() (X_monotone_curve_2 c1, X_monotone_curve_2 c2, X_monotone_curve_2& c)

which accepts two *mergeable x*-monotone curves *c1* and *c2* (see above), and sets *c* to be the merged curve.

Creation

ArrangementXMonotoneTraits_2 traits;

default constructor.

ArrangementXMonotoneTraits_2 traits(other);

copy constructor

ArrangementXMonotoneTraits_2 traits = other

assignment operator.

Accessing Functor Objects

Intersect_2

traits.intersect_2_object()

Split_2

traits.split_2_object()

Are_mergeable_2

traits.are_mergeable_2_object()

Merge_2

traits.merge_2_object()

Has Models

CGAL::Arr_segment_traits_2<Kernel>

CGAL::Arr_non_caching_segment_traits_2<Kernel>

CGAL::Arr_polyline_traits_2<SegmentTraits>

CGAL::Arr_circle_segment_traits_2<Kernel>

CGAL::Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>

CGAL::Arr_rational_arc_traits_2<AlgKernel,NtTraits>

CGAL::Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv>

CGAL::Arr_consolidated_curve_data_traits_2<Traits,Data>

See Also

ArrangementBasicTraits_2 (page [1256](#))

ArrangementTraits_2

Definition

This refined arrangement-traits concept allows the construction of arrangement of *general* planar curves. Models of this concept are used by the free *insert()* functions of the arrangement package and by the *Arrangement_with_history_2* class.

A model of this concept must define the nested *Curve_2* type, which represents a general planar curve that is not necessarily *x*-monotone and is not necessarily connected. Such curves are eventually subdivided into *x*-monotone subcurves and isolated points (represented by the *Point_2* and *X_monotone_curve_2* types, defined in the basic traits concept).

Refines

ArrangementXMonotoneTraits_2

Types

ArrangementTraits_2::Curve_2 represents a general planar curve.

Functor Types

ArrangementTraits_2::Make_x_monotone_2

provides the operator (parameterized by the *OutputIterator* type) :
OutputIterator operator() (Curve_2 c, OutputIterator oi)
 which subdivides the input curve *c* into *x*-monotone subcurves and isolated points, and inserts the results into a container through the given output iterator. The value type of *OutputIterator* is *CGAL::Object*, where each *Object* wraps either an *X_monotone_curve_2* instance or a *Point_2* instance. The operator returns a past-the-end iterator for the output sequence.

Creation

<i>ArrangementTraits_2 traits;</i>	default constructor.
<i>ArrangementTraits_2 traits(other);</i>	copy constructor
<i>ArrangementTraits_2 traits = other</i>	assignment operator.

Accessing Functor Objects

Make_x_monotone_2 traits.make_x_monotone_2_object()

Has Models

CGAL::Arr_segment_traits_2<*Kernel*>
CGAL::Arr_non_caching_segment_traits_2<*Kernel*>
CGAL::Arr_polyline_traits_2<*SegmentTraits*>
CGAL::Arr_circle_segment_traits_2<*Kernel*>
CGAL::Arr_conic_traits_2<*RatKernel*,*AlgKernel*,*NtTraits*>
CGAL::Arr_rational_arc_traits_2<*AlgKernel*,*NtTraits*>
CGAL::Arr_curve_data_traits_2<*Tr*,*XData*,*Mrg*,*CData*,*Cnv*>
CGAL::Arr_consolidated_curve_data_traits_2<*Traits*,*Data*>

See Also

ArrangementBasicTraits_2 (page [1256](#))
ArrangementXMonotoneTraits_2 (page [1261](#))
ArrangementLandmarkTraits_2 (page [1259](#))

CGAL::Arr_segment_traits_2<Kernel>

Definition

The traits class *Arr_segment_traits_2<Kernel>* is a model of the *ArrangementTraits_2* concept that allow the construction and maintenance of arrangements of line segments. It should be parameterized with a CGAL-kernel model that is templated in turn with a number type. To avoid numerical errors and robustness problems, the number type should support exact rational arithmetic — that is, the number type should support the arithmetic operations $+$, $-$, \times and \div that should be carried out without loss of precision.

For example, instantiating the traits template with kernels such as *Cartesian<Quotient<MP_Float> >*, or *Homogeneous<Gmpz>* ensures the exact and robust operation of the application. In particular, the *Cartesian<Gmpq>* achieves the fastest running times in most cases. Using other (inexact) number types (for example, instantiating the template with *Simple_cartesian<double>*) is possible at the user's own risk: selecting an inexact number type usually leads to faster running time at the expense of possible robustness problems.

For optimal performance, we recommend instantiating the traits class with the default *Exact_predicates_exact_constructions_kernel* provided by CGAL. Using this kernel guarantees exactness and robustness, while it incurs only a minor overhead (in comparison to working with a fast, inexact number type) for most inputs.

Arr_segment_traits_2<Kernel> defines *Kernel::Point_2* as its point type. However, it does *not* define *Kernel::Segment_2* as its curve type, as one may expect. The reason is that the kernel segment is represented by its two endpoints only, while the traits class needs to store extra data with its segments, in order to efficiently operate on them. Nevertheless, the nested *X_monotone_curve_2* and *Curve_2* types (in this case both types refer to the same class, as every line segment is (weakly) *x*-monotone) are however convertible to the *Kernel::Segment_2*.

Arr_segment_traits_2<Kernel> achieves faster running times than the *Arr_non_caching_segment_traits_2<Kernel>* traits-class, when arrangements with relatively many intersection points are constructed. It also allows for working with less accurate, yet computationally efficient number types, such as *Quotient<MP_Float>*, which represents floating-point numbers with an unbounded mantissa, but with a bounded exponent. Using this traits class is therefore highly recommended for almost all applications that rely on arrangements of line segments. On the other hand, *Arr_segment_traits_2<Kernel>* uses more space and stores extra data with each segment, so constructing arrangements of huge sets of non-intersecting segments (or segments that intersect very sparsely) could be more efficient with the *Arr_non_caching_segment_traits_2* traits-class.

```
#include <CGAL/Arr_segment_traits_2.h>
```

Is Model for the Concepts

ArrangementTraits_2

ArrangementLandmarkTraits_2

CGAL::Arr_non_caching_segment_basic_traits_2<Kernel>

Definition

The traits class *Arr_non_caching_segment_basic_traits_2<Kernel>* is a model of the *ArrangementTraits_2* concept that allow the construction and maintenance of arrangements of sets of pairwise interior-disjoint line segments. It is templated with a CGAL-*Kernel* model, and it is derived from it. This traits class is a thin layer above the parameterized kernel. It inherits the *Point_2* from the kernel and its *X_monotone_curve_2* type is defined as *Kernel::Segment_2*. Most traits-class functor are inherited from the kernel functor, and the traits class only supplies the necessary functors that are not provided by the kernel. The kernel is parameterized with a number type, which should support the arithmetic operations $+$, $-$ and \times in an exact manner in order to avoid robustness problems. Using *Cartesian<MP_Float>* or *Cartesian<Gmpz>* are possible instantiations for the kernel. Using other (inexact) number types (for example, instatiating the template with *Simple_cartesian<double>*) is also possible, at the user's own risk.

```
#include <CGAL/Arr_non_caching_segment_basic_traits_2.h>
```

Is Model for the Concepts

ArrangementLandmarkTraits_2

CGAL::Arr_non_caching_segment_traits_2<Kernel>

Definition

The traits class *Arr_non_caching_segment_traits_2<Kernel>* is a model of the *ArrangementTraits_2* concept that allows the construction and maintenance of arrangements of line segments. It is parameterized with a CGAL-*Kernel* type, and it is derived from it. This traits class is a thin layer above the parameterized kernel. It inherits the *Point_2* from the kernel and its *X_monotone_curve_2* and *Curve_2* types are both defined as *Kernel::Segment_2*. Most traits-class functor are inherited from the kernel functor, and the traits class only supplies the necessary functors that are not provided by the kernel. The kernel is parameterized with a number type, which should support exact rational arithmetic in order to avoid robustness problems, although other number types could be used at the user's own risk.

The traits-class implementation is very simple, yet may lead to a cascaded representation of intersection points with exponentially long bit-lengths, especially if the kernel is parameterized with a number type that does not perform normalization (e.g. *Quotient<MP_Float>*). The *Arr_segment_traits_2* traits class avoids this cascading problem, and should be the default choice for implementing arrangements of line segments. It is recommended to use *Arr_non_caching_segment_traits_2<Kernel>* only for very sparse arrangements of huge sets of input segments.

```
#include <CGAL/Arr_non_caching_segment_traits_2.h>
```

Is Model for the Concepts

ArrangementTraits_2
ArrangementLandmarkTraits_2

Inherits From

Arr_non_caching_segment_basic_traits_2<Kernel>

See Also

Arr_segment_traits_2<Kernel>

CGAL::Arr_polyline_traits_2<SegmentTraits>

Definition

The traits class *Arr_polyline_traits_2<SegmentTraits>* is a model of the *ArrangementTraits_2* concept. It handles piecewise linear curves, commonly referred to as polylines. Each polyline is a chain of segments, where each two neighboring segments in the chain share a common endpoint. The traits class exploits the functionality of the *SegmentTraits* template-parameter to handle the segments that comprise the polyline curves.

The class instantiated for the template parameter *SegmentTraits* must be a model of the *ArrangementTraits_2* concept that handles line segments (e.g., *Arr_segment_traits_2<Kernel>* or *Arr_non_caching_segment_cached_traits_2<Kernel>*, where the first alternative is recommended).

The number type used by the injected segment traits should support exact rational arithmetic (that is, the number type should support the arithmetic operations $+$, $-$, \times and \div that should be carried out without loss of precision), in order to avoid robustness problems, although other inexact number types could be used at the user's own risk.

```
#include <CGAL/Arr_polyline_traits_2.h>
```

Is Model for the Concepts

ArrangementTraits_2

ArrangementLandmarkTraits_2

Class Arr_polyline_traits_2<SegmentTraits>::Curve_2

The *Curve_2* class nested within the polyline traits is used to represent general continuous piecewise-linear curves (a polyline can be self-intersecting) and support their construction from any range of points.

The copy and default constructor as well as the assignment operator are provided for polyline curves. In addition, an *operator<<* for the curves is defined for standard output streams, and an *operator>>* for the curves is defined for standard input streams.

Types

Arr_polyline_traits_2<SegmentTraits>::Curve_2::const_iterator

A bidirectional iterator that allows traversing the points that comprise a polyline curve.

Arr_polyline_traits_2<SegmentTraits>::Curve_2::const_reverse_iterator

A bidirectional iterator that allows traversing the points that comprise a polyline curve.

Creation

Arr_polyline_traits_2<SegmentTraits>::Curve_2 pi;

default constructor that constructs an empty polyline.

template <class InputIterator>

Arr_polyline_traits_2<SegmentTraits>::Curve_2 pi(Iterator first, Iterator last);

constructs a polyline defined by the given range of points *[first, last)* (the value-type of *InputIterator* must be *SegmentTraits::Point_2*. If the range contains $(n + 1)$ points labeled (p_0, p_1, \dots, p_n) , the generated polyline consists of n segments, where the k th segment is defined by the endpoints $[p_{k-1}, p_k]$. The first point in the range is considered as the source point of the polyline while the last point is considered as its target.

Precondition: There are at least two points in the range.

Access Functions

<i>size_t</i>	<i>pi.points()</i>	returns the number of points that comprise the polyline. Note that if there are n points in the polyline, it is comprised of $(n - 1)$ segments.
---------------	--------------------	----------------------------------------------------------------------------------------------------------------------------------------------------

<i>const_iterator</i>	<i>pi.begin()</i>	returns an iterator pointing at the source point of the polyline.
-----------------------	-------------------	-------------------------------------------------------------------

<i>const_iterator</i>	<i>pi.end()</i>	returns an iterator pointing after the end of the polyline.
-----------------------	-----------------	-------------------------------------------------------------

<i>const_iterator</i>	<i>pi.rbegin()</i>	returns an iterator pointing at the target point of the polyline.
-----------------------	--------------------	-------------------------------------------------------------------

<i>const_iterator</i>	<i>pi.rend()</i>	returns an iterator pointing before the beginning of the polyline.
-----------------------	------------------	--------------------------------------------------------------------

<i>size_t</i>	<i>pi.size()</i>	returns the number of line segments comprising the polyline (equivalent to <i>pi.points() - 1</i>).
---------------	------------------	------------------------------------------------------------------------------------------------------

typename SegmentTraits::X_monotone_curve_2

<i>pi[size_t k]</i>	returns the k th segment of the polyline. <i>Precondition:</i> k is not greater or equal to <i>pi.size() - 1</i> .
----------------------	---------------------------------------------------------------------------------------------------------------------------

<i>Bbox_2</i>	<i>pi.bbox()</i>	return a bounding box of the polyline <i>pi</i> .
---------------	------------------	---------------------------------------------------

Operations

<i>void</i>	<i>pi.push_back(Point_2 p)</i>	adds a new point to the polyline, which becomes the new target point of <i>pi</i> .
-------------	---------------------------------	-------------------------------------------------------------------------------------

void *pi.clear()* resets the polyline.

Class `Arr_polyline_traits_2<SegmentTraits>::X_monotone_curve_2`

The *X_monotone_curve_2* class nested within the polyline traits is used to represent *x*-monotone piecewise linear curves. It inherits from the *Curve_2* type. It has a default constructor and a constructor from a range of points, just like the *Curve_2* class. However, there is precondition that the point range define an *x*-monotone polyline.

The points that define the *x*-monotone polyline are always stored in an ascending lexicographical *xy*-order, so their order may be reversed with respect to the input sequence. Also note that the *x*-monotonicity ensures that an *x*-monotone polyline is never self-intersecting (thus, a self-intersecting polyline will be subdivided to several interior-disjoint *x*-monotone subcurves).

See Also

Arr_segment_traits_2<Kernel>

Arr_non_caching_segment_traits_2<Kernel>

CGAL::Arr_circle_segment_traits_2<Kernel>

Definition

The class *Arr_circle_segment_traits_2<Kernel>* is a model of the *ArrangementTraits_2* concept and can be used to construct and maintain arrangements of circular arcs and line segments.

The traits class must be instantiated with a geometric kernel, such that the supporting circles of the circular arcs are of type *Kernel::Circle_2* and the supporting lines of the line segments are of type *Kernel::Line_2*. Thus, the coordinates of the center of supporting circles, and its squared radius are of type *Kernel::FT*, which should be an exact rational number-type; similarly, the coefficients of each supporting line $ax + by + c = 0$ are also of type *Kernel::FT*. Note however that the intersection point between two such arcs do not have rational coordinates in general. For this reason, we do not require the endpoints of the input arcs and segments to have rational coordinates.

The nested *Point_2* type defined by the traits class is therefore *different* than the *Kernel::Point_2* type. Its coordinates are of type *CoordNT*, and represent real numbers obtained from solving quadratic equations with rational coordinates. A number of type *CoordNT* can therefore be expressed as $\alpha + \beta\sqrt{\gamma}$, where α , β and γ are all rational numbers. The definition of the curve and *x*-monotone curve types nested in the traits class are detailed below.

```
#include <CGAL/Arr_circle_segment_traits_2.h>
```

Is Model for the Concepts

ArrangementTraits_2

Class Arr_circle_segment_traits_2<Kernel>::CoordNT

The *CoordNT* number-type nested within the traits class represents an algebraic number of degree 2; it can be represented as $\alpha + \beta\sqrt{\gamma}$, where α , β and γ are all rational numbers of type *Kernel::FT*.

Types

Arr_circle_segment_traits_2<Kernel>::CoordNT::Rational
the *Kernel::FT* type.

Creation

```
Arr_circle_segment_traits_2<Kernel>::CoordNT x;  
creates a variable whose value is 0.
```

```
Arr_circle_segment_traits_2<Kernel>::CoordNT x(Rational alpha);  
creates a variable whose value is the rational number  $\alpha$ .
```

Arr_circle_segment_traits_2<Kernel>::CoordNT *x*(*Rational* *alpha*, *Rational* *beta*, *Rational* *gamma*);

creates a variable whose value is the algebraic number $\alpha + \beta\sqrt{\gamma}$.

Access Functions

<i>bool</i>	<i>x.is_rational()</i>	determines whether <i>x</i> is rational.
<i>Rational</i>	<i>x.alpha()</i>	returns α .
<i>Rational</i>	<i>x.beta()</i>	returns β (0 if <i>x</i> is rational).
<i>Rational</i>	<i>x.gamma()</i>	returns γ (0 if <i>x</i> is rational).

Operators

The *CoordNT* number-type supports all arithmetic operations with rational numbers of type *Kernel::FT*. For example, it is possible to compute $x + q$, $q - x$, $x += q$, etc. where *q* is rational.

The global functions *CGAL::sign(x)*, *CGAL::square(x)*, *CGAL::to_double(x)* and *CGAL::compare(x,y)*, where *x* and *y* are of type *CoordNT*, are also supported.

Class *Arr_circle_segment_traits_2<Kernel>::Point_2*

The *Point_2* number-type nested within the traits class represents a Cartesian point whose coordinates are algebraic numbers of type *CoordNT*.

Types

Arr_circle_segment_traits_2<Kernel>::Point_2:: Rational

the *Kernel::FT* type.

Arr_circle_segment_traits_2<Kernel>::Point_2:: CoordNT

the algebraic number-type.

Creation

Arr_circle_segment_traits_2<Kernel>::Point_2 *p*;

default constructor.

Arr_circle_segment_traits_2<Kernel>::Point_2 *p*(*Rational* *x*, *Rational* *y*);

creates the point (*x*,*y*).

Arr_circle_segment_traits_2<Kernel>::Point_2 *p*(*CoordNT* *x*, *CoordNT* *y*);

creates the point (*x*,*y*).

Access Functions

CoordNT *p.x()* returns the *x*-coordinate.

CoordNT *p.y()* returns the *y*-coordinate.

Class *Arr_circle_segment_traits_2<Kernel>::Curve_2*

The *Curve_2* class nested within the traits class can represent arbitrary circular arcs, full circles and line segments and support their construction in various ways. The copy and default constructor as well as the assignment operator are provided. In addition, an *operator<<* for the curves is defined for standard output streams.

Creation

Arr_circle_segment_traits_2<Kernel>::Curve_2 *cv*(*typename Kernel::Segment_2* *seg*);

constructs an curve corresponding to the line segment *seg*.

Arr_circle_segment_traits_2<Kernel>::Curve_2 *cv*(*typename Kernel::Point_2* *source*,
typename Kernel::Point_2 *target*)

constructs an curve corresponding to the line segment directed from *source* to *target*, both having rational coordinates.

Arr_circle_segment_traits_2<Kernel>::Curve_2 *cv*(*typename Kernel::Line_2* *line*,
Point_2 *source*,
Point_2 *target*)

constructs an curve corresponding to the line segment supported by the given line, directed from *source* to *target*. Note that the two endpoints may have one-root coordinates.

Precondition: Both endpoints must lie on the given supporting line.

Arr_circle_segment_traits_2<Kernel>::Curve_2 *cv*(*typename Kernel::Circle_2* *circ*);

constructs an curve corresponding to the given circle. *circ* has a center point with rational coordinates and its *squared* radius is rational.

Arr_circle_segment_traits_2<Kernel>::Curve_2 *cv*(*typename Kernel::Point_2* *c*,
typename Kernel::FT *r*,
Orientation *orient* = *COUNTERCLOCKWISE*)

constructs an curve corresponding to a circle centered at the rational point *c* whose radius *r* is rational.

Arr_circle_segment_traits_2<Kernel>::Curve_2 *cv*(*typename Kernel::Circle_2* *circ*,
Point_2 *source*,
Point_2 *target*)

constructs a circular arc supported by *circ*, which has a center point with rational coordinates and whose *squared* radius is rational, with the given endpoints. The orientation of the arc is the same as the orientation of *circ*.

Precondition: Both endpoints must lie on the given supporting circle.

Arr_circle_segment_traits_2<Kernel>::Curve_2 *cv*(*typename Kernel::Point_2* *c*,
typename Kernel::FT *r*,
Orientation *orient*,
Point_2 *source*,
Point_2 *target*)

constructs a circular arc supported by a circle centered at the rational point *c* whose radius *r* is rational, directed from *source* to *target* with the given orientation.

Precondition: Both endpoints must lie on the supporting circle.

Arr_circle_segment_traits_2<Kernel>::Curve_2 *cv*(*typename Kernel::Point_2* *source*,
typename Kernel::Point_2 *mid*,
typename Kernel::Point_2 *target*)

constructs an circular arc whose endpoints are *source* and *target* that passes through *mid*. All three points have rational coordinates.

Precondition: The three points must not be collinear.

Access Functions

<i>bool</i>	<i>cv.is_full()</i>	indicates whether the curve represents a full circle.
<i>Point_2</i>	<i>cv.source()</i>	returns the source point.
<i>Point_2</i>	<i>cv.target()</i>	<i>Precondition:</i> <i>cv</i> is not a full circle. returns the target point.
		<i>Precondition:</i> <i>cv</i> is not a full circle.
<i>Orientation</i>	<i>cv.orientation()</i>	returns the orientation of the curve (<i>COLLINEAR</i> in case of line segments).
<i>bool</i>	<i>cv.is_linear()</i>	determines whether <i>cv</i> is a line segment.
<i>bool</i>	<i>cv.is_circular()</i>	determines whether <i>cv</i> is a circular arc.

typename Kernel::Line_2

cv.supporting_line() returns the supporting line of *cv*.
Precondition: *cv* is a line segment.

typename Kernel::Circle_2

cv.supporting_circle()

returns the supporting circle of *cv*.

Precondition: *cv* is a circular arc.

Class Arr_circle_segment_traits_2<Kernel>::X_monotone_curve_2

The *X_monotone_curve_2* class nested within the traits class can represent *x*-monotone and line segments (which are always weakly *x*-monotone). The copy and default constructor as well as the assignment operator are provided. In addition, an *operator<<* for the curves is defined for standard output streams.

Creation

Arr_circle_segment_traits_2<Kernel>::X_monotone_curve_2 xcv(typename Kernel::Point_2 source,
typename Kernel::Point_2 target)

constructs an curve corresponding to the line segment directed from *source* to *target*, both having rational coordinates.

Arr_circle_segment_traits_2<Kernel>::X_monotone_curve_2 xcv(typename Kernel::Line_2 line,
Point_2 source,
Point_2 target)

constructs an curve corresponding to the line segment supported by the given line, directed from *source* to *target*. Note that the two endpoints may have one-root coordinates.

Precondition: Both endpoints must lie on the given supporting line.

Arr_circle_segment_traits_2<Kernel>::X_monotone_curve_2 xcv(typename Kernel::Circle_2 circ,
Point_2 source,
Point_2 target,
Orientation orient)

constructs a circular arc supported by *circ*, which has a center point with rational coordinates and whose *squared* radius is rational, with the given endpoints. The orientation of the arc is determined by *orient*.

Precondition: Both endpoints must lie on the given supporting circle.

Precondition: The circular arc is *x*-monotone.

Access Functions

Point_2 *xcv.source()* returns the source point of *xcv*.

<i>Point_2</i>	<i>xcv.target()</i>	returns the target point of <i>xcv</i> .
<i>Point_2</i>	<i>xcv.left()</i>	returns the left (lexicographically smaller) endpoint of <i>xcv</i> .
<i>Point_2</i>	<i>xcv.right()</i>	returns the right (lexicographically larger) endpoint of <i>xcv</i> .
<i>Orientation</i>	<i>xcv.orientation()</i>	returns the orientation of the curve (<i>COLLINEAR</i> in case of line segments).
<i>bool</i>	<i>xcv.is_linear()</i>	determines whether <i>xcv</i> is a line segment.
<i>bool</i>	<i>xcv.is_circular()</i>	determines whether <i>xcv</i> is a circular arc.
<i>typename Kernel::Line_2</i>		
	<i>xcv.supporting_line()</i>	returns the supporting line of <i>xcv</i> . <i>Precondition: xcv is a line segment.</i>
<i>typename Kernel::Circle_2</i>		
	<i>xcv.supporting_circle()</i>	returns the supporting circle of <i>xcv</i> . <i>Precondition: xcv is a circular arc.</i>
<i>Bbox_2</i>	<i>xcv.bbox()</i>	returns a bounding box of the arc <i>xcv</i> .

CGAL::Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>

Definition

The class `Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>` is a model of the *ArrangementTraits_2* concept and can be used to construct and maintain arrangements of bounded segments of algebraic curves of degree 2 at most, also known as *conic curves*.

A general conic curve C is the locus of all points (x,y) satisfying the equation: $rx^2 + sy^2 + txy + ux + vy + w = 0$, where:

- If $4rs - t^2 > 0$, C is an ellipse. A special case occurs when $r = s$ and $t = 0$, when C becomes a circle.
- If $4rs - t^2 < 0$, C is a hyperbola.
- If $4rs - t^2 = 0$, C is a parabola. A degenerate case occurs when $r = s = t = 0$, when C is a line.

A *bounded conic arc* is defined as either of the following:

- A full ellipse (or a circle) C .
- The tuple $\langle C, p_s, p_t, o \rangle$, where C is the supporting conic curve, with the arc endpoints being p_s and p_t (the source and target points, respectively). The orientation o indicates whether we proceed from p_s to p_t in a clockwise or in a counterclockwise direction. Note that C may also correspond to a line or to pair of lines — in this case o may specify a *COLLINEAR* orientation.

A very useful subset of the set of conic arcs are line segments and circular arcs, as arrangements of circular arcs and line segments have some interesting applications (e.g. offsetting polygons, motion planning for a disc robot, etc.). Circular arcs and line segment are simpler objects and can be dealt with more efficiently than arbitrary arcs. For these reasons, it is possible to construct conic arcs from segments and from circles. Using these constructors is highly recommended: It is more straightforward and also speeds up the arrangement construction. However, in case the set of input curves contain only circular arcs and line segments, it is recommended to use the `Arr_circle_segment_2` class to achieve faster running times.

In our representation, all conic coefficients (namely r, s, t, u, v, w) must be rational numbers. This guarantees that the coordinates of all arrangement vertices (in particular, those representing intersection points) are algebraic numbers of degree 4 (a real number α is an algebraic number of degree d if there exist a polynomial p with integer coefficient of degree d such that $p(\alpha) = 0$). We therefore require separate representations of the curve coefficients and the point coordinates. The *NtTraits* should be instantiated with a class that defines nested *Integer*, *Rational* and *Algebraic* number types and supports various operations on them, yielding certified computation results (for example, it can convert rational numbers to algebraic numbers and can compute roots of polynomials with integer coefficients). The other template parameters, *RatKernel* and *AlgKernel* should be geometric kernels templated with the *NtTraits::Rational* and *NtTraits::Algebraic* number types, respectively. It is recommended to instantiate the *CORE_algebraic_number_traits* class as the *NtTraits* parameter, with *Cartesian<NtTraits::Rational>* and *Cartesian<NtTraits::Algebraic>* instantiating the two kernel types, respectively. The number types in this case are provided by the CORE library, with its ability to exactly represent simple algebraic numbers.

The traits class inherits its point type from *AlgKernel::Point_2*, and defines a curve and x -monotone curve types, as detailed below.

```
#include <CGAL/Arr_conic_traits_2.h>
```

Is Model for the Concepts

ArrangementTraits_2

ArrangementLandmarkTraits_2

Class **Arr_conic_traits_2**<RatKernel,AlgKernel,NtTraits>::Curve_2

The *Curve_2* class nested within the conic-arc traits can represent arbitrary conic arcs and support their construction in various ways. The copy and default constructor as well as the assignment and equality operators are provided for conic arcs. In addition, an *operator<<* for the curves is defined for standard output streams.

Types

Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>::Curve_2:: *Rational*

the *NtTraits::Rational* type (and also the *RatKernel::FT* type).

Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>::Curve_2:: *Algebraic*

the *NtTraits::Algebraic* type (and also the *AlgKernel::FT* type).

Creation

Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>::Curve_2 *a*(typename *RatKernel::Segment_2* *seg*);

constructs an arc corresponding to the line segment *seg*.

Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>::Curve_2 *a*(typename *RatKernel::Circle_2* *circ*);

constructs an arc corresponding to the full circle *circ* (note that this circle has a center point with rational coordinates and rational squared radius).

Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>::Curve_2 *a*(typename *RatKernel::Circle_2* *circ*,
Orientation *o*,
Point_2 *ps*,
Point_2 *pt*)

constructs a circular arc supported by the circle *circ*, going in the given orientation *o* from the source point *ps* to its target point *pt*.

Precondition: *ps* and *pt* both lie on the circle *circ*.

Precondition: *o* is not *COLLINEAR*.

Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>::Curve_2 *a*(*typename RatKernel::Point_2* *p1*,
typename RatKernel::Point_2 *p2*,
typename RatKernel::Point_2 *p3*)

constructs a circular arc going from *p1* (its source point) through *p2* to *p3* (its target point). Note that all three points have rational coordinates. The orientation of the arc is determined automatically.

Precondition: The three points are not collinear.

Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>::Curve_2 *a*(*Rational* *r*,
Rational *s*,
Rational *t*,
Rational *u*,
Rational *v*,
Rational *w*)

constructs a conic arc that corresponds to the full conic curve $rx^2 + sy^2 + txy + ux + vy + w = 0$.

Precondition: As a conic arc must be bounded, the given curve must be an ellipse, that is $4rs - t^2 > 0$.

Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>::Curve_2 *a*(*Rational* *r*,
Rational *s*,
Rational *t*,
Rational *u*,
Rational *v*,
Rational *w*,
Orientation *o*,
Point_2 *ps*,
Point_2 *pt*)

constructs a conic arc supported by the conic curve $rx^2 + sy^2 + txy + ux + vy + w = 0$, going in the given orientation *o* from the source point *ps* to its target point *pt*.

Precondition: *ps* and *pt* both satisfy the equation of the supporting conic curve and define a bounded segment of this curve (e.g. in case of a hyperbolic arc, both point should be located on the same branch of the hyperbola).

Precondition: *o* is not *COLLINEAR* if the supporting conic is curves, and must be *COLLINEAR* if it is not curved (a line or a line-pair).

Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>::Curve_2 *a*(*typename RatKernel::Point_2* *p1*,
typename RatKernel::Point_2 *p2*,
typename RatKernel::Point_2 *p3*,
typename RatKernel::Point_2 *p4*,

typename RatKernel::Point_2 p5)

constructs a conic arc going from $p1$ (its source point) through $p2$, $p3$ and $p4$ (in this order) to $p5$ (its target point). Note that all five points have rational coordinates. The orientation of the arc is determined automatically.

Precondition: No three points of the five are not collinear.

Precondition: The five points define a valid arc, in their given order.

Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>::Curve_2 a(Rational r,
Rational s,
Rational t,
Rational u,
Rational v,
Rational w,
Orientation o,
Point_2 app_ps,
Rational r1,
Rational s1,
Rational t1,
Rational u1,
Rational v1,
Rational w1,
Point_2 app_pt,
Rational r2,
Rational s2,
Rational t2,
Rational u2,
Rational v2,
Rational w2)

constructs a conic arc supported by the conic curve $rx^2 + sy^2 + txy + ux + vy + w = 0$, going in the given orientation o from its source point to its target point. In this case only some approximations of the endpoints (*app_ps* and *app_pt*, respectively) is available, and their exact locations are given implicitly, specified by the intersections of the supporting conic curve with $r_1x^2 + s_1y^2 + t_1xy + u_1x + v_1y + w_1 = 0$ and $r_2x^2 + s_2y^2 + t_2xy + u_2x + v_2y + w_2 = 0$, respectively.

Precondition: The two auxiliary curves specifying the endpoints really intersect with the supporting conic curve, such that the arc endpoints define a bounded segment of the supporting curve (e.g. in case of a hyperbolic arc, both point should be located on the same branch of the hyperbola).

Precondition: o is not *COLLINEAR* if the supporting conic is curves, and must be *COLLINEAR* if it is not curved (a line or a line-pair).

Access Functions

<i>bool</i>	<i>a.is_valid()</i>	indicates whether <i>a</i> is a valid conic arc. As the precondition to some of the constructor listed above are quite complicated, their violation does not cause the program to abort. Instead, the constructed arc is invalid (a defaultly constructed arc is also invalid). It is however recommended to check that a constructed arc is valid before inserting it to an arrangement, as this operation <i>will</i> cause the program to abort.
<i>bool</i>	<i>a.is_x_monotone()</i>	determines whether the arc is <i>x</i> -monotone, namely each vertical line intersects it at most once. A vertical line segment is also considered (weakly) <i>x</i> -monotone.
<i>bool</i>	<i>a.is_y_monotone()</i>	determines whether the arc is <i>y</i> -monotone, namely each horizontal line intersects it at most once. A horizontal line segment is also considered (weakly) <i>x</i> -monotone.
<i>bool</i>	<i>a.is_full_conic()</i>	indicates whether the arc represents a full conic curve (en ellipse or a circle).

The six following methods return the coefficients of the supported conic, after their conversion to integer number (represented by the *Integer* type of the *NtTraits* class):

<i>typename NtTraits::Integer</i>	<i>a.r()</i>	returns the coefficient of x^2 .
<i>typename NtTraits::Integer</i>	<i>a.s()</i>	returns the coefficient of t^2 .
<i>typename NtTraits::Integer</i>	<i>a.t()</i>	returns the coefficient of xy .
<i>typename NtTraits::Integer</i>	<i>a.u()</i>	returns the coefficient of x .
<i>typename NtTraits::Integer</i>	<i>a.v()</i>	returns the coefficient of y .
<i>typename NtTraits::Integer</i>	<i>a.w()</i>	returns the free coefficient.
<i>Point_2</i>	<i>a.source()</i>	returns the source point of the arc. <i>Precondition:</i> <i>a</i> is not a full conic curve.
<i>Point_2</i>	<i>a.target()</i>	returns the target point of the arc. <i>Precondition:</i> <i>a</i> is not a full conic curve.
<i>Orientation</i>	<i>a.orientation()</i>	returns the orientation of the arc.
<i>Bbox_2</i>	<i>a.bbox()</i>	return a bounding box of the arc <i>a</i> .

Operations

<i>void</i>	<i>a.set_source(Point_2 ps)</i>	sets a new source point for the conic arc. <i>Precondition:</i> <i>ps</i> lies on the supporting conic of <i>a</i> .
<i>void</i>	<i>a.set_target(Point_2 pt)</i>	sets a new target point for the conic arc. <i>Precondition:</i> <i>pt</i> lies on the supporting conic of <i>a</i> .

Class **Arr_conic_traits_2<RatKernel,AlgKernel,NtTraits>::X_monotone_curve_2**

The *X_monotone_curve_2* class nested within the conic-arc traits is used to represent *x*-monotone conic arcs. It inherits from the *Curve_2* type, therefore supports the access methods and the operations listed above.

For efficiency reasons, we recommend users not to construct x -monotone conic arc directly, but rather use the *Make_x_monotone_2* functor supplied by the conic-arc traits class to convert conic curves to x -monotone curves.

Creation

Arr_conic_traits_2<*RatKernel*,*AlgKernel*,*NtTraits*>::*X_monotone_curve_2* *xa*(*Curve_2* *arc*);

converts the given arc to an x -monotone arc.

Precondition: *arc* is x -monotone.

Access Functions

<i>Point_2</i>	<i>xa.left()</i>	returns the left (lexicographically smaller) endpoint of <i>xa</i> .
<i>Point_2</i>	<i>xa.right()</i>	returns the right (lexicographically larger) endpoint of <i>xa</i> .

CGAL::Arr_rational_arc_traits_2<AlgKernel,NtTraits>

Definition

The traits class *Arr_rational_arc_traits_2<AlgKernel,NtTraits>* is a model of the *ArrangementTraits_2* concept. It handles bounded segments of rational functions, referred to as *rational arcs*, and enables the construction and maintenance of arrangements of such arcs. A rational function $y = \frac{P(x)}{Q(x)}$ is defined by two polynomials P and Q of arbitrary degrees. In particular, if $Q(x) = 1$ then the function is a simple polynomial function. A bounded rational arc is defined by the graph of a rational function over some interval $[x_{\min}, x_{\max}]$, where Q does not have any real roots in this interval (thus the arc does not contain any poles). Rational functions, and polynomial functions in particular, are not only interesting in their own right, they are also very useful for approximating or interpolating more complicated curves.

In our representation, all polynomial coefficients (the coefficients of P and Q) must be rational numbers. This guarantees that the x -coordinates of all arrangement vertices (in particular, those representing intersection points) can be represented as roots of polynomials with integer coefficients — namely, algebraic numbers. The y -coordinates can be obtained by simple arithmetic operations on the x -coordinates, hence they are also algebraic numbers.

We therefore require separate representations of the curve coefficients and the point coordinates. The *NtTraits* should be instantiated with a class that defines nested *Integer*, *Rational* and *Algebraic* number types and supports various operations on them, yielding certified computation results (for example, it can convert rational numbers to algebraic numbers and can compute roots of polynomials with integer coefficients). The *AlgKernel* template-parameter should be a geometric kernel templated with the *NtTraits::Algebraic* number-type. It is recommended to instantiate the *CORE_algebraic_number_traits* class as the *NtTraits* parameter, with *Cartesian<NtTraits::Algebraic>* instantiating the kernel. The number types in this case are provided by the *CORE* library, with its ability to exactly represent simple algebraic numbers.

The traits class defined its point type to be *AlgKernel::Point_2*, and defines a curve type (and an identical x -monotone curve type, as a rational arc is always x -monotone by definition) as detailed below.

```
#include <CGAL/Arr_rational_arc_traits_2.h>
```

Is Model for the Concepts

ArrangementTraits_2

Class Arr_rational_arc_traits_2<AlgKernel,NtTraits>::Curve_2

The *Curve_2* class nested within the rational-arc traits is used to represent rational arcs and support their construction from a single polynomial and the range where the arc is defined or a pair of polynomials and a pair of corresponding ranges. The copy and default constructor as well as the assignment operator are provided for polyline curves. In addition, an *operator<<* for the curves is defined for standard output streams.

Types

Arr_rational_arc_traits_2<AlgKernel,NtTraits>::Curve_2::Rat_vector

A vector of rational numbers (equivalent to *std::vector<typename NtTraits::Rational>*).

Arr_rational_arc_traits_2<AlgKernel,NtTraits>::Curve_2:: Polynomial

the *NtTraits::Polynomial* type (a polynomial with integer coefficients).

Creation

Arr_rational_arc_traits_2<AlgKernel,NtTraits>::Curve_2 a;

default constructor.

*Arr_rational_arc_traits_2<AlgKernel,NtTraits>::Curve_2 a(Rat_vector p_coeffs,
typename NtTraits::Algebraic s_x,
typename NtTraits::Algebraic t_x)*

constructs an arc supported by the polynomial $y = P(x)$, defined over the interval $[x_s, x_t]$, given by the x -coordinates of the arc's source and target. The vector *p_coeffs* specifies the coefficients of $P(x)$, where the polynomial degree is *p_coeffs.size() - 1* and $p[k]$ is the coefficient of x^k in P .
Precondition: $s_x \neq t_x$.

*Arr_rational_arc_traits_2<AlgKernel,NtTraits>::Curve_2 a(Rat_vector p_coeffs,
Rat_vector q_coeffs,
typename NtTraits::Algebraic s_x,
typename NtTraits::Algebraic t_x)*

constructs an arc supported by the rational function $y = \frac{P(x)}{Q(x)}$, defined over the interval $[s_x, t_x]$, given by the x -coordinates of the arc's source and target. The vectors *p_coeffs* and *q_coeffs* specify the coefficients of $P(x)$ and $Q(x)$, respectively (see above).
Precondition: $x_{min} < x_{max}$.
Precondition: For each $x_{min} \leq x \leq x_{max}$, $Q(x) \neq 0$.

Access Functions

bool a.is_valid()

indicates whether *a* is a valid rational arc. As the precondition $Q(x) \neq 0$ to the constructor from two polynomials is quite complicated, its violation does not cause the program to abort. Instead, the constructed arc is invalid (a defaultly constructed arc is also invalid). It is however recommended to check that a constructed arc is valid before inserting it to an arrangement, as this operation *will* cause the program to abort.

Polynomial a.numerator()

returns a polynomial with integer coefficients equivalent to $P(x)$.

<i>Polynomial</i>	<i>a.denominator()</i>	returns a polynomial with integer coefficients equivalent to $Q(x)$.
<i>Point_2</i>	<i>a.source()</i>	returns the source point of the arc. <i>Precondition:</i> a is not a full conic curve.
<i>Point_2</i>	<i>a.target()</i>	returns the target point of the arc. <i>Precondition:</i> a is not a full conic curve.
<i>Point_2</i>	<i>a.left()</i>	returns the left (lexicographically smaller) endpoint of a .
<i>Point_2</i>	<i>a.right()</i>	returns the right (lexicographically larger) endpoint of a .

CGAL::Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv>

Definition

The class *Arr_curve_data_traits_2*<*Tr*,*XData*,*Mrg*,*CData*,*Cnv*> is a model of the *ArrangementTraits_2* concept and serves as a decorator class that allows the extension of the curves defined by the base traits-class (the *Tr* parameter), which serves as a geometric traits-class (a model of the *ArrangementTraits_2* concept), with extraneous (non-geometric) data fields.

The traits class inherits its point type from *Traits::Point_2*, and defines an extended *Curve_2* and *X_monotone_curve_2* types, as detailed below.

Each *Curve_2* object is associated with a single data field of type *CData*, and each *X_monotone_curve_2* object is associated with a single data field of type *XData*. When a curve is subdivided into *x*-monotone subcurves, its data field is converted using the conversion functor, which is specified by the *Cnv* template-parameter, and the resulting objects is copied to all *X_monotone_curve_2* objects induced by this curve. The conversion functor should provide an operator with the following prototype:

```
XData operator() (const CData& d) const;
```

By default, the two data types are the same, so the conversion operator is trivial:

```
CData = XData
```

```
Cnv = _Default_convert_functor<CData,XData>
```

In case two (or more) *x*-monotone curves overlap, their data fields are merged to a single field, using the merge functor, which is specified by the *Mrg* template-parameter. This functor should provide an operator with the following prototype:

```
XData operator() (const XData& d1, const XData& d2) const;
```

which returns a single data object that represents the merged data field of *d1* and *d2*. The *x*-monotone curve that represents the overlap is associated with the output of this functor.

```
#include <CGAL/Arr_curve_data_traits_2.h>
```

Is Model for the Concepts

```
ArrangementTraits_2
```

Types

```
typedef Tr Base_traits_2; the base traits-class.
```

```
typedef typename Base_traits_2::Curve_2
```

```
Base_curve_2; the base curve.
```

```
typedef typename Base_traits_2::X_monotone_curve_2
```

```
Base_x_monotone_curve_2;
```

```
the base x-monotone curve curve.
```

<i>typedef Mrg</i>	<i>Merge;</i>	the merge functor.
<i>typedef Cnv</i>	<i>Convert;</i>	the conversion functor.
<i>typedef CData</i>	<i>Curve_data;</i>	the type of data associated with curves.
<i>typedef XData</i>	<i>X_monotone_curve_data;</i>	the type of data associated with <i>x</i> -monotone curves.

Inherits From

Base_traits_2

Class **Arr_curve_data_traits_2**<Tr,XData,Mrg,CData,Cnv>::Curve_2

The *Curve_2* class nested within the curve-data traits extends the *Base_traits_2::Curve_2* type with an extra data field of type *Data*.

Inherits From

Base_curve_2

Creation

Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv>::Curve_2 *cv*;

default constructor.

Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv>::Curve_2 *cv*(*Base_curve_2 base*);

constructs curve from the given *base* curve with uninitialized data field.

Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv>::Curve_2 *cv*(*Base_curve_2 base*, *Data data*);

constructs curve from the given *base* curve with an attached *data* field.

Access Functions

Curve_data *cv.data()* returns the data field (a non-const version, which returns a reference to the data object, is also available).

void *cv.set_data(Curve_data data)*

sets the data field.

Class Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv>::X_monotone_curve_2

The *X_monotone_curve_2* class nested within the curve-data traits extends the *Base_traits_2::X_monotone_curve_2* type with an extra data field.

Inherits From

Base_x_monotone_curve_2

Creation

Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv>::X_monotone_curve_2 *xcv*;

default constructor.

Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv>::X_monotone_curve_2 *xcv*(*Base_x_monotone_curve_2* *base*)

constructs an *x*-monotone curve from the given *base* curve with uninitialized data field.

Arr_curve_data_traits_2<Tr,XData,Mrg,CData,Cnv>::X_monotone_curve_2 *xcv*(*Base_x_monotone_curve_2* *base*,
X_monotone_curve_data *data*)

constructs an *x*-monotone curve from the given *base x*-monotone curve with an attached *data* field.

Access Functions

X_monotone_curve_data

xcv.data() returns the field (a non-const version, which returns a reference to the data object, is also available).

void *xcv.set_data(X_monotone_curve_data data)*

sets the data field.

CGAL::Arr_consolidated_curve_data_traits_2<Traits,Data>

Definition

The class *Arr_consolidated_curve_data_traits_2*<*Traits*,*Data*> is a model of the *ArrangementTraits_2* concept and serves as a decorator class that allows the extension of the curves defined by the *Traits* parameter, which serves as a geometric traits-class (a model of the *ArrangementTraits_2* concept), with extraneous (non-geometric) data fields of type *Data*.

The traits class inherits its point type from *Traits::Point_2*, and defines an extended *Curve_2* and *X_monotone_curve_2* types, as detailed below.

Each *Curve_2* object is associated with a single data field of type *Data*, and each *X_monotone_curve_2* object is associated with a set of unique data objects. When a curve is subdivided into *x*-monotone subcurves, all these subcurves are associated with a list containing a single data object, copies from the inducing curve. When an *x*-monotone curve is split, its data set is duplicated to both resulting subcurves. In case two (or more) *x*-monotone curves overlap, their data sets are consolidated and are associated with the *x*-monotone curve that represents the overlap.

```
#include <CGAL/Arr_consolidated_curve_data_traits_2.h>
```

Is Model for the Concepts

ArrangementTraits_2

Inherits From

```
Arr_curve_data_traits_2<Traits,
    _Unique_list<Data>,
    _Consolidate_unique_lists<Data>,
    Data>
```

Types

```
typedef Traits          Base_traits_2;          the base traits-class.
typedef typename Base_traits_2::Curve_2
```

```
          Base_curve_2;          the base curve.
typedef typename Base_traits_2::X_monotone_curve_2
```

```
          Base_x_monotone_curve_2;
```

the base *x*-monotone curve.

```
Arr_consolidated_curve_data_traits_2<Traits,Data>:: typedef Data_container
```

a set of data objects that is associated with an *x*-monotone curve.

Arr_consolidated_curve_data_traits_2<Traits,Data>:: typedef Data_iterator

a non-mutable iterator for the data objects in the data container.

Class Arr_consolidated_curve_data_traits_2<Traits,Data>::Data_container

The *Data_container* class nested within the consolidated curve-data traits and associated with the *Traits::X_monotone_curve_2* type is maintained as a list with unique data objects. This representation is simple and efficient in terms of memory consumption. It also requires that the *Data* class supports only the equality operator. Note however that most set operations require linear time.

Creation

Arr_consolidated_curve_data_traits_2<Traits,Data>:: Data_container dset;

default constructor.

Arr_consolidated_curve_data_traits_2<Traits,Data>:: Data_container dset(Data data);

constructs set containing a single *data* object.

Access Functions

<i>std::size_t</i>	<i>dset.size()</i>	returns the number of data objects in the set.
<i>Data_iterator</i>	<i>dset.begin()</i>	returns an iterator pointing to the first data object.
<i>Data_iterator</i>	<i>dset.end()</i>	returns a past-the-end iterator for the data objects.
<i>Data</i>	<i>dset.front()</i>	returns the first data object inserted into the set. <i>Precondition:</i> The number of data objects is not 0.
<i>Data</i>	<i>dset.back()</i>	returns the last data object inserted into the set. <i>Precondition:</i> The number of data objects is not 0.

Predicates

<i>bool</i>	<i>dset == Data_container other</i>	check if the two sets contain the same data objects (regardless of order).
<i>Data_iterator</i>	<i>dset.find(Data data)</i>	find the given <i>data</i> object in the set and returns an iterator for this object, or <i>end()</i> if it is not found.

Modifiers

<i>bool</i>	<i>dset.insert(Data data)</i>	inserts the given <i>data</i> object into the set. Returns <i>true</i> on success, or <i>false</i> if the set already contains the object.
<i>bool</i>	<i>dset.erase(Data data)</i>	erases the given <i>data</i> object from the set. Returns <i>true</i> on success, or <i>false</i> if the set does not contain the object.
<i>void</i>	<i>dset.clear()</i>	clears the set.

ArrangementInputFormatter

A model for the ArrangementInputFormatter concept supports a set of functions that enable reading an arrangement from an input stream using a specific format.

Types

ArrangementInputFormatter::Arrangement_2 the type of arrangement to input.

typedef typename Arrangement_2::Point_2

Point_2; the point type.

typedef typename Arrangement_2::X_monotone_curve_2

X_monotone_curve_2;

the *x*-monotone curve type.

typedef typename Arrangement_2::Size

Size;

typedef typename Arrangement_2::Vertex_handle

Vertex_handle;

typedef typename Arrangement_2::Halfedge_handle

Halfedge_handle;

typedef typename Arrangement_2::Face_handle

Face_handle;

Creation

ArrangementInputFormatter inf; default constructor.

ArrangementInputFormatter inf(std::istream& is);

constructs a formatter that reads from *is*.

void inf.set_in(std::istream& is)

directs *inf* to read from *is*.

Access Functions

std::istream& inf.in()

returns the stream that *inf* reads from.

Precondition: *inf* is directed to a valid output stream.

Formatted Input Functions

<i>void</i>	<i>inf.read_arrangement_begin()</i>	reads a message indicating the beginning of the arrangement.
<i>void</i>	<i>inf.read_arrangement_end()</i>	reads a message indicating the end of the arrangement.
<i>Size</i>	<i>inf.read_size(const char *label = NULL)</i>	reads a size value, which is supposed to be preceeded by the given label.
<i>void</i>	<i>inf.read_vertices_begin()</i>	reads a message indicating the beginning of the vertex records.
<i>void</i>	<i>inf.read_vertices_end()</i>	reads a message indicating the end of the vertex records.
<i>void</i>	<i>inf.read_edges_begin()</i>	reads a message indicating the beginning of the edge records.
<i>void</i>	<i>inf.read_edges_end()</i>	reads a message indicating the end of the edge records.
<i>void</i>	<i>inf.read_faces_begin()</i>	reads a message indicating the beginning of the face records.
<i>void</i>	<i>inf.read_faces_end()</i>	reads a message indicating the end of the face records.
<i>void</i>	<i>inf.read_vertex_begin()</i>	reads a message indicating the beginning of a single vertex record.
<i>void</i>	<i>inf.read_vertex_end()</i>	reads a message indicating the end of a single vertex record.
<i>std::size_t</i>	<i>inf.read_vertex_index()</i>	reads and returns a vertex index.
<i>void</i>	<i>inf.read_point(Point_2& p)</i>	reads a point.

<i>void</i>	<i>inf.read_vertex_data(Vertex_handle v)</i>	reads an auxiliary vertex-data object and associates it with the vertex <i>v</i> .
<i>void</i>	<i>inf.read_edge_begin()</i>	reads a message indicating the beginning of a single edge record.
<i>void</i>	<i>inf.read_edge_end()</i>	reads a message indicating the end of a single edge record.
<i>std::size_t</i>	<i>inf.read_halfedge_index()</i>	reads and returns halfedge index.
<i>void</i>	<i>inf.read_x_monotone_curve(X_monotone_curve_2& c)</i>	reads an <i>x</i> -monotone curve.
<i>void</i>	<i>inf.read_halfedge_data(Halfedge_handle he)</i>	reads an auxiliary halfedge-data object and associates it with the halfedge <i>he</i> .
<i>void</i>	<i>inf.read_face_begin()</i>	reads a message indicating the beginning of a single face record.
<i>void</i>	<i>inf.read_face_end()</i>	reads a message indicating the end of a single face record.
<i>void</i>	<i>inf.read_outer_ccb_begin()</i>	reads a message indicating the beginning of the outer CCB of the current face.
<i>void</i>	<i>inf.read_outer_ccb_end()</i>	reads a message indicating the end of the outer CCB of the current face.
<i>void</i>	<i>inf.read_holes_begin()</i>	reads a message indicating the beginning of the container of holes inside the current face.
<i>void</i>	<i>inf.read_holes_end()</i>	reads a message indicating the end of the container of holes inside the current face.
<i>void</i>	<i>inf.read_inner_ccb_begin()</i>	reads a message indicating the beginning of an inner CCB of the current face.

<i>void</i>	<i>inf.read_inner_ccb_end()</i>	reads a message indicating the end of an inner CCB of the current face.
<i>void</i>	<i>inf.read_ccb_halfedges_begin()</i>	reads a message indicating the beginning a connected component boundary.
<i>void</i>	<i>inf.read_ccb_halfedges_end()</i>	reads a message indicating the end of a connected component boundary.
<i>void</i>	<i>inf.read_isolated_vertices_begin()</i>	reads a message indicating the beginning of the container of isolated vertices inside the current face.
<i>void</i>	<i>inf.read_isolated_vertices_end()</i>	reads a message indicating the end of the container of isolated vertices inside the current face.
<i>void</i>	<i>inf.read_face_data(Face_handle f)</i>	reads an auxiliary face-data object and associates it with the face <i>f</i> .

Has Models

Arr_text_formatter<Arrangement> (page [1300](#))

Arr_face_extended_text_formatter<Arrangement> (page [1301](#))

Arr_extended_dcel_text_formatter<Arrangement> (page [1302](#))

ArrangementOutputFormatter

A model for the ArrangementOutputFormatter concept supports a set of functions that enable writing an arrangement to an output stream using a specific format.

Types

ArrangementOutputFormatter::Arrangement_2

the type of arrangement to output.

typedef typename Arrangement_2::Point_2

Point_2; the point type.

typedef typename Arrangement_2::X_monotone_curve_2

X_monotone_curve_2;

the *x*-monotone curve type.

typedef typename Arrangement_2::Size

Size;

typedef typename Arrangement_2::Vertex_const_handle

Vertex_const_handle;

typedef typename Arrangement_2::Halfedge_const_handle

Halfedge_const_handle;

typedef typename Arrangement_2::Face_const_handle

Face_const_handle;

Creation

ArrangementOutputFormatter outf; default constructor.

ArrangementOutputFormatter outf(std::ostream& os);

constructs a formatter that writes to *os*.

void outf.set_out(std::ostream& os)

directs *outf* to write to *os*.

Access Functions

std::ostream& *outf.out()* returns the stream that *outf* writes to.
Precondition: outf is directed to a valid output stream.

Formatted Output Functions

void *outf.write_arrangement_begin()*
writes a message indicating the beginning of the arrangement.

void *outf.write_arrangement_end()*
writes a message indicating the end of the arrangement.

void *outf.write_size(const char *label, Size size)*
writes a size value, preceeded by a given label.

void *outf.write_vertices_begin()*
writes a message indicating the beginning of the vertex records.

void *outf.write_vertices_end()*
writes a message indicating the end of the vertex records.

void *outf.write_edges_begin()*
writes a message indicating the beginning of the edge records.

void *outf.write_edges_end()*
writes a message indicating the end of the edge records.

void *outf.write_faces_begin()*
writes a message indicating the beginning of the face records.

void *outf.write_faces_end()*
writes a message indicating the end of the face records.

void *outf.write_vertex_begin()*
writes a message indicating the beginning of a single vertex record.

void *outf.write_vertex_end()*
writes a message indicating the end of a single vertex record.

<i>void</i>	<i>outf.write_vertex_index(std::size_t idx)</i>	writes a vertex index.
<i>void</i>	<i>outf.write_point(Point_2 p)</i>	writes a point.
<i>void</i>	<i>outf.write_vertex_data(Vertex_const_handle v)</i>	writes the auxiliary data associated with the vertex.
<i>void</i>	<i>outf.write_edge_begin()</i>	writes a message indicating the beginning of a single edge record.
<i>void</i>	<i>outf.write_edge_end()</i>	writes a message indicating the end of a single edge record.
<i>void</i>	<i>outf.write_halfedge_index(std::size_t idx)</i>	writes a halfedge index.
<i>void</i>	<i>outf.write_x_monotone_curve(X_monotone_curve_2 c)</i>	writes an <i>x</i> -monotone curve.
<i>void</i>	<i>outf.write_halfedge_data(Halfedge_const_handle he)</i>	writes the auxiliary data associated with the halfedge.
<i>void</i>	<i>outf.write_face_begin()</i>	writes a message indicating the beginning of a single face record.
<i>void</i>	<i>outf.write_face_end()</i>	writes a message indicating the end of a single face record.
<i>void</i>	<i>outf.write_outer_ccb_begin()</i>	writes a message indicating the beginning of the outer CCB of the current face.
<i>void</i>	<i>outf.write_outer_ccb_end()</i>	writes a message indicating the end of the outer CCB of the current face.

<i>void</i>	<i>outf.write_holes_begin()</i>	writes a message indicating the beginning of the container of holes inside the current face.
<i>void</i>	<i>outf.write_holes_end()</i>	writes a message indicating the end of the container of holes inside the current face.
<i>void</i>	<i>outf.write_ccb_halfedges_begin()</i>	writes a message indicating the beginning a connected component's boundary.
<i>void</i>	<i>outf.write_ccb_halfedges_end()</i>	writes a message indicating the end of a connected component's boundary.
<i>void</i>	<i>outf.write_isolated_vertices_begin()</i>	writes a message indicating the beginning of the container of isolated vertices inside the current face.
<i>void</i>	<i>outf.write_isolated_vertices_end()</i>	writes a message indicating the end of the container of isolated vertices inside the current face.
<i>void</i>	<i>outf.write_face_data(Face_const_handle f)</i>	writes the auxiliary data associated with the face.

Has Models

Arr_text_formatter<Arrangement> (page [1300](#))

Arr_face_extended_text_formatter<Arrangement> (page [1301](#))

Arr_extended_dcel_text_formatter<Arrangement> (page [1302](#))

CGAL::Arr_text_formatter<Arrangement>

Definition

Arr_text_formatter<Arrangement> defines the format of an arrangement in an input or output stream (typically a file stream), thus enabling reading and writing an *Arrangement* instance using a simple text format. The arrangement is assumed to store no auxiliary data with its DCEL records (and if there are such records they will not be written or read by the formatter).

The *Arr_text_formatter<Arrangement>* class assumes that the nested *Point_2* and the *Curve_2* types defined by the *Arrangement* template-parameter can both be written to an input stream using the << operator and read from an input stream using the >> operator.

```
#include <CGAL/IO/Arr_text_formatter.h>
```

Is Model for the Concepts

ArrangementInputFormatter
ArrangementOutputFormatter

See Also

read (page [1234](#))
write (page [1235](#))

CGAL::Arr_face_extended_text_formatter<Arrangement>

Definition

Arr_face_extended_text_formatter<Arrangement> defines the format of an arrangement in an input or output stream (typically a file stream), thus enabling reading and writing an *Arrangement* instance using a simple text format. The *Arrangement* class should be instantiated with a DCEL class which in turn instantiates the *Arr_face_extended_dcel* template with a *FaceData* type. The formatter supports reading and writing the data objects attached to the arrangement faces as well.

The *Arr_face_extended_text_formatter<Arrangement>* class assumes that the nested *Point_2* and the *Curve_2* types defined by the *Arrangement* template-parameter and that the *FaceData* type can all be written to an input stream using the << operator and read from an input stream using the >> operator.

```
#include <CGAL/IO/Arr_text_formatter.h>
```

Is Model for the Concepts

ArrangementInputFormatter
ArrangementOutputFormatter

See Also

read (page [1234](#))
write (page [1235](#))
Arr_face_extended_dcel<Traits,FData,V,H,F> (page [1251](#))

CGAL::Arr_extended_dcel_text_formatter<Arrangement>

Definition

Arr_extended_dcel_text_formatter<Arrangement> defines the format of an arrangement in an input or output stream (typically a file stream), thus enabling reading and writing an *Arrangement* instance using a simple text format. The *Arrangement* class should be instantiated with a DCEL class which in turn instantiates the *Arr_extended_dcel* template with the *VertexData*, *HalfedgeData* and *FaceData* types. The formatter supports reading and writing the data objects attached to the arrangement vertices, halfedges and faces.

The *Arr_extended_dcel_text_formatter<Arrangement>* class assumes that the nested *Point_2* and the *Curve_2* types defined by the *Arrangement* template-parameter, as well as the *VertexData*, *HalfedgeData* and *FaceData* types, can all be written to an input stream using the << operator and read from an input stream using the >> operator.

```
#include <CGAL/IO/Arr_text_formatter.h>
```

Is Model for the Concepts

ArrangementInputFormatter
ArrangementOutputFormatter

See Also

read (page [1234](#))

write (page [1235](#))

Arr_extended_dcel<Traits,VData,HData,FData,V,H,F> (page [1252](#))

ArrangementPointLocation_2

Definition

A model of the ArrangementPointLocation_2 concept can be attached to an *Arrangement_2* instance and answer point-location queries on this arrangement. Namely, given a *Arrangement_2::Point_2* object, representing a point in the plane, it returns the arrangement cell containing it. In the general case, the query point is contained inside an arrangement face, but in degenerate situations it may lie on an edge or coincide with an arrangement vertex.

Types

ArrangementPointLocation_2::Arrangement_2

the associated arrangement type.

ArrangementPointLocation_2::Point_2

equivalent to *Arrangement_2::Point_2*.

Creation

ArrangementPointLocation_2 pl;

default constructor.

ArrangementPointLocation_2 pl(Arrangement_2 arr);

constructs a point-location object *pl* attached to the given arrangement *arr*.

Query Functions

Object

pl.locate(Point_2 q)

locates the arrangement cell that contains the query point *q* and returns a handle for this cell. The function returns an *Object* instance that wraps either of the following types:

- *Arrangement_2::Face_const_handle*, in case *q* is contained inside an arrangement face;
- *Arrangement_2::Halfedge_const_handle*, in case *q* lies on an arrangement edge;
- *Arrangement_2::Vertex_const_handle*, in case *q* coincides with an arrangement vertex.

Precondition: *pl* is attached to a valid arrangement instance.

Operations

void *pl.attach(Arrangement_2 arr)*

attaches *pl* to the given arrangement *arr*.

void *pl.detach()*

detaches *pl* from the arrangement it is currently attached to.

Has Models

Arr_naive_point_location<Arrangement>

Arr_walk_along_a_line_point_location<Arrangement>

Arr_trapezoid_ric_point_location<Arrangement>

Arr_landmarks_point_location<Arrangement,Generator>

ArrangementVerticalRayShoot_2

Definition

A model of the `ArrangementVerticalRayShoot_2` concept can be attached to an `Arrangement_2` instance and answer vertical ray-shooting queries on this arrangement. Namely, given a `Arrangement_2::Point_2` object, representing a point in the plane, it returns the arrangement feature (edge or vertex) that lies strictly above it (or below it). By “strictly” we mean that if the query point lies on an arrangement edge (or on an arrangement vertex) this edge will *not* be the query result, but the feature lying above or below it. (An exception to this rule is the degenerate situation where the query point lies in the interior of a vertical edge.) Note that it may happen that the query point lies above the upper envelope (or below the lower envelope) of the arrangement, so that the vertical ray emanating from it may go to infinity without hitting any arrangement feature on its way. In this case the unbounded face is returned.

Types

`ArrangementVerticalRayShoot_2::Arrangement_2`

the associated arrangement type.

`ArrangementVerticalRayShoot_2::Point_2`

equivalent to `Arrangement_2::Point_2`.

Creation

`ArrangementVerticalRayShoot_2 rs;`

default constructor.

`ArrangementVerticalRayShoot_2 rs(Arrangement_2 arr);`

constructs a ray-shooting object `rs` attached to the given arrangement `arr`.

Query Functions

Object

`rs.ray_shoot_up(Point_2 q)`

locates the arrangement feature that is first hit by an upward-directed vertical ray emanating from the query point q , and returns a handle for this feature. The function returns an *Object* instance that is a wrapper for one of the following types:

- `Arrangement_2::Halfedge_const_handle`, in case the vertical ray hits an arrangement edge;
- `Arrangement_2::Vertex_const_handle`, in case the vertical ray hits an arrangement vertex.
- `Arrangement_2::Face_const_handle` for the unbounded arrangement face, in case q lies above the upper envelope of the arrangement.

Precondition: `rs` is attached to a valid arrangement instance.

Object

rs.ray_shoot_down(Point_2 q)

locates the arrangement feature that is first hit by a downward-directed vertical ray emanating from the query point q , and returns a handle for this feature. The function returns an *Object* instance that is a wrapper for one of the following types:

- *Arrangement_2::Halfedge_const_handle*, in case the vertical ray hits an arrangement edge;
- *Arrangement_2::Vertex_const_handle*, in case the vertical ray hits an arrangement vertex.
- *Arrangement_2::Face_const_handle* for the unbounded arrangement face, in case q lies below the lower envelope of the arrangement.

Precondition: *rs* is attached to a valid arrangement instance.

Operations

void

rs.attach(Arrangement_2 arr)

attaches *rs* to the given arrangement *arr*.

void

rs.detach()

detaches *rs* from the arrangement it is currently attached to.

Has Models

Arr_naive_point_location<Arrangement>

Arr_walk_along_a_line_point_location<Arrangement>

Arr_trapezoid_ric_point_location<Arrangement>

Arr_landmarks_point_location<Arrangement,Generator>

CGAL::Arr_naive_point_location<Arrangement>

Definition

The *Arr_naive_point_location<Arrangement>* class implements a naïve algorithm that traverses all the vertices and halfedges in the arrangement in search for an answer to a point-location query. The query time is therefore linear in the complexity of the arrangement. Naturally, this point-location strategy could turn into a heavy time-consuming process when applied to dense arrangements.

```
#include <CGAL/Arr_naive_point_location.h>
```

Is Model for the Concepts

ArrangementPointLocation_2
ArrangementVerticalRayShoot_2

CGAL::Arr_walk_along_line_point_location<Arrangement>

Definition

The *Arr_walk_along_line_point_location<Arrangement>* class implements a very simple point-location (and vertical ray-shooting) strategy that improves the naïve one. The algorithm considers an imaginary vertical ray emanating from the query point, and simulates a walk along the zone of this ray, starting from the unbounded face until reaching the query point. In dense arrangements this walk can considerably reduce the number of traversed arrangement edges, with respect to the naïve algorithm.

The walk-along-a-line point-location object (just like the naïve one) does not use any auxiliary data structures. Thus, attaching it to an existing arrangement takes constant time, and any ongoing updates to this arrangement do not affect the point-location object. It is therefore recommended to use the “walk” point-location strategy for arrangements that are constantly changing, especially if the number of issued queries is not large.

```
#include <CGAL/Arr_walk_along_line_point_location.h>
```

Is Model for the Concepts

ArrangementPointLocation_2
ArrangementVerticalRayShoot_2

CGAL::Arr_trapezoid_ric_point_location<Arrangement>

Definition

The *Arr_trapezoid_ric_point_location<Arrangement>* class implements the incremental randomized algorithm introduced by Mulmuley [Mul90] as presented by Seidel [Sei91] (see also [dBvKOS00, Chapter 6]). It subdivides each arrangement face to pseudo-trapezoidal cells, each of constant complexity, and constructs and maintains a search structure on top of these cells, such that each query can be answered in $O(\log n)$ time, where n is the complexity of the arrangement.

Constructing the search structures takes $O(n \log n)$ time, such that attaching a trapezoidal point-location object to an existing arrangement may incur some overhead in running times. In addition, the point-location object needs to keep its auxiliary data structures up-to-date as the arrangement goes through structural changes. It is therefore recommended to use this point-location strategy for static arrangements (or arrangements that do not alter frequently), and when the number of issued queries is relatively large.

```
#include <CGAL/Arr_trapezoid_ric_point_location.h>
```

Is Model for the Concepts

ArrangementPointLocation_2
ArrangementVerticalRayShoot_2

CGAL::Arr_landmarks_point_location<Arrangement,Generator>

The *Arr_landmarks_point_location*<*Arrangement*,*Generator*> class implements a Jump & Walk algorithm, where special points, referred to as “landmarks”, are chosen in a preprocessing stage, their place in the arrangement is found, and they are inserted into a data-structure that enables efficient nearest-neighbor search (a KD-tree). Given a query point, the nearest landmark is located and a “walk” strategy is applied from the landmark to the query point.

There are various strategies to select the landmark set in the arrangement, where the strategy is determined by the *Generator* template parameter. The following landmark-generator classes are available:

Arr_landmarks_vertices_generator — The arrangement vertices are used as the landmarks set.

Arr_random_landmarks_generator — n random points in the bounding box of the arrangement are chosen as the landmarks set.

Arr_halton_landmarks_generator — n Halton points in the bounding box of the arrangement are chosen as the landmarks set.

Arr_middle_edges_landmarks_generator — The midpoint of each arrangement edge is computed, and the resulting set of points is used as the landmarks set. This generator can be applied only for arrangements of line segments.

Arr_grid_landmarks_generator — A set of n landmarks are chosen on a $\lceil\sqrt{n}\rceil \times \lceil\sqrt{n}\rceil$ grid that covers the bounding box of the arrangement.

The *Arr_landmarks_vertices_generator* class is the default generator and used if no *Generator* parameter is specified.

It is recommended to use the landmarks point-location strategy when the application frequently issues point-location queries on a rather static arrangement that the changes applied to it are mainly insertions of curves and not deletions of them.

```
#include <CGAL/Arr_landmarks_point_location.h>
```

Is Model for the Concepts

ArrangementPointLocation_2

ArrangementVerticalRayShoot_2

CGAL::locate

```
#include <CGAL/Arr_batched_point_location.h>
```

```
template<class Arrangement, class PointsIterator, class OutputIterator>
OutputIterator      locate( Arrangement arr,
                           PointsIterator points_begin,
                           PointsIterator points_end,
                           OutputIterator oi)
```

Performs a batched point-location operation on an arrangement. The function accepts a range of query points, defined by $[points_begin, points_end)$ and locates each point in the arrangement. The query-results are returned in through the output iterator, whose value type is $std::pair<Point_2, Object>$. Namely, each result is given as a point and an object representing the arrangement feature that contains it (an *Object* that may be either *Face_const_handle*, *Halfedge_const_handle* or *Vertex_const_handle*). The result pair in output sequence are sorted by an increasing *xy*-lexicographical order on the query points. The function returns a past-the-end iterator for the output sequence.

Precondition: The value-type of *PointsIterator* is *Traits::Point_2*.

Precondition: The value-type of *OutputIterator* is $std::pair< Traits::Point_2, Object>$.

CGAL::Arr_observer<Arrangement>

Definition

Arr_observer<Arrangement> serves as an abstract base class for all observer classes that are attached to an arrangement instance of type *Arrangement* and receive notifications from the arrangement. This base class handles the attachment of the observer to a given arrangement instance or to the detachment of the observer from this arrangement instance. It also gives a default empty implementation to all notification functions that are invoked by the arrangement to notify the observer on local or global changes it undergoes. The notification functions are all virtual functions, so they can be overridden by the concrete observer classes that inherit from *Arr_observer<Arrangement>*.

In order to implement a concrete arrangement observer-class, one simply needs to derive from *Arr_observer<Arrangement>* and override the relevant notification functions. For example, if only face-split events are of interest, it is sufficient to override just *before_split_face()* (or just *after_split_face()*).

```
#include <CGAL/Arr_observer.h>
```

Types

Arr_observer<Arrangement>::Arrangement_2 the type of the associated arrangement.

```
typedef typename Arrangement_2::Point_2
```

Point_2; the point type.

```
typedef typename Arrangement_2::X_monotone_curve_2
```

X_monotone_curve_2;

the x-monotone curve type.

```
typedef typename Arrangement_2::Vertex_handle
```

Vertex_handle;

```
typedef typename Arrangement_2::Halfedge_handle
```

Halfedge_handle;

```
typedef typename Arrangement_2::Face_handle
```

Face_handle;

```
typedef typename Arrangement_2::Ccb_halfedge_circulator
```

Ccb_halfedge_circulator;

represents the boundary of a connected component (CCB). In particular, holes are represented by a circulator for their outer CCB.

Creation

Arr_observer<Arrangement> *obs*; constructs an observer that is unattached to any arrangement instance.

Arr_observer<Arrangement> *obs*(*Arrangement_2* & *arr*);
constructs an observer and attaches it to the given arrangement *arr*.

Modifiers

void *obs.attach*(*Arrangement_2* & *arr*)
attaches the observer to the given arrangement *arr*.

void *obs.detach*() detaches the observer from its arrangement.

Notifications on Global Arrangement Operations

virtual void *obs.before_assign*(*Arrangement_2* *arr*)
issued just before the attached arrangement is assigned with the contents of another arrangement *arr*.

virtual void *obs.after_assign*() issued immediately after the attached arrangement has been assigned with the contents of another arrangement.

virtual void *obs.before_clear*() issued just before the attached arrangement is cleared.
virtual void *obs.after_clear*(*Face_handle* *uf*)
issued immediately after the attached arrangement has been cleared, so it now consists only of a the unbounded face *uf*.

virtual void *obs.before_global_change*()
issued just before a global function starts to modify the attached arrangement. It is guaranteed that no queries (especially no point-location queries) are issued until the termination of the global function is indicated by *after_global_change*().

virtual void *obs.after_global_change*()
issued immediately after a global function has stopped modifying the attached arrangement.

Notifications on Attachment or Detachment

<i>virtual void</i>	<i>obs.before_attach(Arrangement_2 arr)</i>	issued just before the observer is attached to the arrangement instance <i>arr</i> .
<i>virtual void</i>	<i>obs.after_attach()</i>	issued immediately after the observer has been attached to an arrangement instance.
<i>virtual void</i>	<i>obs.before_detach()</i>	issued just before the observer is detached from its arrangement instance.
<i>virtual void</i>	<i>obs.after_detach()</i>	issued immediately after the observer has been detached from its arrangement instance.

Notifications on Local Changes in the Arrangement

<i>virtual void</i>	<i>obs.before_create_vertex(Point_2 p)</i>	issued just before a new vertex that corresponds to the point <i>p</i> is created.
<i>virtual void</i>	<i>obs.after_create_vertex(Vertex_handle v)</i>	issued immediately after a new vertex <i>v</i> has been created. Note that the vertex still has no incident edges and is not connected to any other vertex.
<i>virtual void</i>	<i>obs.before_create_edge(X_monotone_curve_2 c, Vertex_handle v1, Vertex_handle v2)</i>	issued just before a new edge that corresponds to the <i>x</i> -monotone curve <i>c</i> and connects the vertices <i>v1</i> and <i>v2</i> is created.
<i>virtual void</i>	<i>obs.after_create_edge(Halfedge_handle e)</i>	issued immediately after a new edge <i>e</i> has been created. The halfedge that is sent to this function is always directed from <i>v1</i> to <i>v2</i> (see above).
<i>virtual void</i>	<i>obs.before_modify_vertex(Vertex_handle v, Point_2 p)</i>	issued just before a vertex <i>v</i> is modified to be associated with the point <i>p</i> .
<i>virtual void</i>	<i>obs.after_modify_vertex(Vertex_handle v)</i>	issued immediately after an existing vertex <i>v</i> has been modified.
<i>virtual void</i>	<i>obs.before_modify_edge(Halfedge_handle e, X_monotone_curve_2 c)</i>	issued just before an edge <i>e</i> is modified to be associated with the <i>x</i> -monotone curve <i>c</i> .

<i>virtual void</i>	<i>obs.after_modify_edge(Halfedge_handle e)</i>	issued immediately after an existing edge <i>e</i> has been modified.
<i>virtual void</i>	<i>obs.before_split_edge(Halfedge_handle e, Vertex_handle v, X_monotone_curve_2 c1, X_monotone_curve_2 c2)</i>	issued just before an edge <i>e</i> is split into two edges that should be associated with the <i>x</i> -monotone curves <i>c1</i> and <i>c2</i> . The vertex <i>v</i> corresponds to the split point, and will be used to separate the two resulting edges.
<i>virtual void</i>	<i>obs.after_split_edge(Halfedge_handle e1, Halfedge_handle e2)</i>	issued immediately after an existing edge has been split into the two given edges <i>e1</i> and <i>e2</i> .
<i>virtual void</i>	<i>obs.before_split_face(Face_handle f, Halfedge_handle e)</i>	issued just before a face <i>f</i> is split into two, as a result of the insertion of the edge <i>e</i> into the arrangement.
<i>virtual void</i>	<i>obs.after_split_face(Face_handle f1, Face_handle f2, bool is_hole)</i>	issued immediately after the existing face <i>f1</i> has been split, such that a portion of it now forms a new face <i>f2</i> . The flag <i>is_hole</i> designates whether <i>f2</i> forms a hole inside <i>f1</i> .
<i>virtual void</i>	<i>obs.before_split_hole(Face_handle f, Ccb_halfedge_circulator h, Halfedge_handle e)</i>	issued just before a hole <i>h</i> inside a face <i>f</i> is split into two, as a result of the removal of the edge <i>e</i> from the arrangement.
<i>virtual void</i>	<i>obs.after_split_hole(Face_handle f, Ccb_halfedge_circulator h1, Ccb_halfedge_circulator h2)</i>	issued immediately after a hole inside the face <i>f</i> has been split, resulting in the two holes <i>h1</i> and <i>h2</i> .
<i>virtual void</i>	<i>obs.before_add_hole(Face_handle f, Halfedge_handle e)</i>	issued just before the edge <i>e</i> is inserted as a new hole inside the face <i>f</i> .
<i>virtual void</i>	<i>obs.after_add_hole(Ccb_halfedge_circulator h)</i>	issued immediately after a new hole <i>h</i> has been created. The hole always consists of a single pair of twin halfedges.

<i>virtual void</i>	<i>obs.before_add_isolated_vertex(Face_handle f, Vertex_handle v)</i>	issued just before the vertex <i>v</i> is inserted as an isolated vertex inside the face <i>f</i> .
<i>virtual void</i>	<i>obs.after_add_isolated_vertex(Vertex_handle v)</i>	issued immediately after the vertex <i>v</i> has been set as an isolated vertex.
<i>virtual void</i>	<i>obs.before_merge_edge(Halfedge_handle e1, Halfedge_handle e2, X_monotone_curve_2 c)</i>	issued just before the two edges <i>e1</i> and <i>e2</i> are merged to form a single edge that will be associated with the <i>x</i> -monotone curve <i>c</i> .
<i>virtual void</i>	<i>obs.after_merge_edge(Halfedge_handle e)</i>	issued immediately after two edges have been merged to form the edge <i>e</i> .
<i>virtual void</i>	<i>obs.before_merge_face(Face_handle f1, Face_handle f2, Halfedge_handle e)</i>	issued just before the two edges <i>f1</i> and <i>f2</i> are merged to form a single face, following the removal of the edge <i>e</i> from the arrangement.
<i>virtual void</i>	<i>obs.after_merge_face(Face_handle f)</i>	issued immediately after two faces have been merged to form the face <i>f</i> .
<i>virtual void</i>	<i>obs.before_merge_hole(Face_handle f, Ccb_halfedge_circulator h1, Ccb_halfedge_circulator h2, Halfedge_handle e)</i>	issued just before two holes <i>h1</i> and <i>h2</i> inside the face <i>f</i> are merged to form a single connected component, following the insertion of the edge <i>e</i> into the arrangement.
<i>virtual void</i>	<i>obs.after_merge_hole(Face_handle f, Ccb_halfedge_circulator h)</i>	issued immediately after two holes have been merged to form a single hole <i>h</i> inside the face <i>f</i> .
<i>virtual void</i>	<i>obs.before_move_hole(Face_handle from_f, Face_handle to_f, Ccb_halfedge_circulator h)</i>	issued just before the hole <i>h</i> is moved from one face to another. This can happen if the face <i>to_f</i> containing the hole has just been split from <i>from_f</i> .

<i>virtual void</i>	<i>obs.after_move_hole(Ccb_halfedge_circulator h)</i>	issued immediately after the hole <i>h</i> has been moved to a new face.
<i>virtual void</i>	<i>obs.before_move_isolated_vertex(Face_handle from_f, Face_handle to_f, Vertex_handle v)</i>	issued just before the isolated vertex <i>v</i> is moved from one face to another. This can happen if the face <i>to_f</i> containing the isolated vertex has just been split from <i>from_f</i> .
<i>virtual void</i>	<i>obs.after_move_isolated_vertex(Vertex_handle v)</i>	issued immediately after the isolated vertex <i>v</i> has been moved to a new face.
<i>virtual void</i>	<i>obs.before_remove_vertex(Vertex_handle v)</i>	issued just before the vertex <i>v</i> is removed from arrangement.
<i>virtual void</i>	<i>obs.after_remove_vertex()</i>	issued immediately after a vertex has been removed (and deleted) from the arrangement.
<i>virtual void</i>	<i>obs.before_remove_edge(Halfedge_handle e)</i>	issued just before the edge <i>e</i> is removed from arrangement.
<i>virtual void</i>	<i>obs.after_remove_edge()</i>	issued immediately after an edge has been removed (and deleted) from the arrangement.
<i>virtual void</i>	<i>obs.before_remove_hole(Face_handle f, Ccb_halfedge_circulator h)</i>	issued just before the hole <i>f</i> is removed from inside the face <i>f</i> .
<i>virtual void</i>	<i>obs.after_remove_hole(Face_handle f)</i>	issued immediately after a hole has been removed (and deleted) from inside the face <i>f</i> .

CGAL::Arrangement_with_history_2<Traits,Dcel>

Definition

An object *arr* of the class *Arrangement_with_history_2<Traits,Dcel>* represents the planar subdivision induced by a set of input curves \mathcal{C} . The arrangement is represented as a doubly-connected edge-list (DCEL). As is the case for the *Arrangement_2<Traits,Dcel>*, each DCEL vertex is associated with a point and each edge is associated with an x -monotone curve whose interior is disjoint from all other edges and vertices. Each such x -monotone curve is a subcurve of some $C \in \mathcal{C}$ — or may represent an overlap among several curves in \mathcal{C} .

The *Arrangement_with_history_2<Traits,Dcel>* class-template extends the *Arrangement_2* class-template by keeping an additional container of input curves representing \mathcal{C} , and by maintaining a cross-mapping between these curves and the arrangement edges they induce. This way it is possible to determine the inducing curve(s) of each arrangement edge. This mapping also allows the traversal of input curves, and the traversal of edges induced by each curve.

The *Arrangement_with_history_2<Traits,Dcel>* template has two parameters:

- The *Traits* template-parameter should be instantiated with a model of the *ArrangementTraits_2* concept. The traits class defines the *Curve_2* type, which represents an input curve. It also defines the types of x -monotone curves and two-dimensional points, namely *X_monotone_curve_2* and *Point_2*, respectively, and supports basic geometric predicates on them.
- The *Dcel* template-parameter should be instantiated with a class that is a model of the *ArrangementDcel* concept. The value of this parameter is by default *Arr_default_dcel<Traits>*.

Self is an abbreviation of the *Arrangement_with_history_2<Traits,Dcel>* type hereafter.

```
#include <CGAL/Arrangement_with_history_2.h>
```

Types

```
Arrangement_with_history_2<Traits,Dcel>:: Traits_2
```

the traits class in use.

```
Arrangement_with_history_2<Traits,Dcel>:: Dcel
```

the DCEL representation of the arrangement.

```
typedef typename Traits_2::Point_2
```

```
Point_2;
```

the point type, as defined by the traits class.

```
typedef typename Traits_2::X_monotone_curve_2
```

```
X_monotone_curve_2;
```

the x -monotone curve type, as defined by the traits class.

typedef typename Traits_2::Curve_2

Curve_2; the curve type, as defined by the traits class.

In addition, the nested types *Vertex*, *Halfedge* and *Face* are defined, as well as all handle, iterator and circulator types, as defined by the *Arrangement_2* class-template (page 1201).

Arrangement_with_history_2<Traits,Dcel>:: Curve_handle

a handle for an input curve.

Arrangement_with_history_2<Traits,Dcel>:: Curve_iterator

a bidirectional iterator over the curves that induce the arrangement. Its value-type is *Curve_2*.

Arrangement_with_history_2<Traits,Dcel>:: Induced_edge_iterator

an iterator over the edges induced by an input curve. Its value type is *Halfedge_handle*.

Arrangement_with_history_2<Traits,Dcel>:: Originating_curve_iterator

an iterator for the curves that originate a given arrangement edge. Its value type is *Curve_handle*.

Creation

Arrangement_with_history_2<Traits,Dcel> arr;

constructs an empty arrangement containing one unbounded face, which corresponds to the whole plane.

Arrangement_with_history_2<Traits,Dcel> arr(Self other);

copy constructor.

*Arrangement_with_history_2<Traits,Dcel> arr(Traits_2 *traits);*

constructs an empty arrangement that uses the given *traits* instance for performing the geometric predicates.

Assignment Methods

Self& *arr = other* assignment operator.

void *arr.assign(Self other)*

assigns the contents of another arrangement.

<i>void</i>	<i>arr.clear()</i>	clears the arrangement.
-------------	--------------------	-------------------------

Access Functions

See the *Arrangement_2* reference pages (page [1201](#)) for the full list.

- Accessing the Input Curves:

<i>Size</i>	<i>arr.number_of_curves()</i>	returns the number of input curves that induce the arrangement.
-------------	-------------------------------	-----------------------------------------------------------------

<i>Curve_iterator</i>	<i>arr.curves_begin()</i>	returns the begin-iterator of the curves inducing the arrangement.
-----------------------	---------------------------	--------------------------------------------------------------------

<i>Curve_iterator</i>	<i>arr.curves_end()</i>	returns the past-the-end iterator of the curves inducing the arrangement.
-----------------------	-------------------------	---------------------------------------------------------------------------

<i>Size</i>	<i>arr.number_of_induced_edges(Curve_handle ch)</i>	returns the number of arrangement edges induced by the curve <i>ch</i> .
-------------	------------------------------------------------------	--------------------------------------------------------------------------

<i>Induced_edge_iterator</i>	<i>arr.induced_edges_begin(Curve_handle ch)</i>	returns the begin-iterator of the edges induced by the curve <i>ch</i> .
------------------------------	--------------------------------------------------	--------------------------------------------------------------------------

<i>Induced_edge_iterator</i>	<i>arr.induced_edges_end(Curve_handle ch)</i>	returns the past-the-end iterator of the edges induced by the curve <i>ch</i> .
------------------------------	------------------------------------------------	---------------------------------------------------------------------------------

<i>Size</i>	<i>arr.number_of_originating_curves(Halfedge_handle e)</i>	returns the number of input curves that originate the edge <i>e</i> .
-------------	-------------------------------------------------------------	-----------------------------------------------------------------------

<i>Originating_curve_iterator</i>	<i>arr.originating_curves_begin(Halfedge_handle e)</i>	returns the begin-iterator of the curves originating the edge <i>e</i> .
-----------------------------------	---------------------------------------------------------	--------------------------------------------------------------------------

<i>Originating_curve_iterator</i>	<i>arr.originating_curves_end(Halfedge_handle e)</i>	returns the past-the-end iterator of the curves originating the edge <i>e</i> .
-----------------------------------	-------------------------------------------------------	---------------------------------------------------------------------------------

Modifiers

See the *Arrangement_2* reference pages (page 1201) for the full list of functions for modifying arrangement vertices.

- *Modifying Arrangement Edges:*

The following functions override their counterparts in the *Arrangement_2* class, as they also maintain the cross-relationships between the input curves and the edges they induce.

Halfedge_handle *arr.split_edge(Halfedge_handle e, Point_2 p)*

splits the edge e into two edges (more precisely, into two halfedge pairs), at a given split point p . The function returns a handle for the halfedge whose source is the same as $e \rightarrow source()$ and whose target vertex is the split point.

Precondition: p lies in the interior of the curve associated with e .

<i>Halfedge_handle</i>	<i>arr.merge_edge(Halfedge_handle e1, Halfedge_handle e2)</i>
------------------------	----------------------------------------------------------------

merges the edges represented by *e1* and *e2* into a single edge. The function returns a handle for one of the merged halfedges.

Precondition: $e1$ and $e2$ share a common end-vertex, of degree 2, and the x -monotone curves associated with $e1$ and $e2$ are mergeable into a single x -monotone curves.

[illegible]

removes the edge e from the arrangement. Since the e may be the only edge incident to its source vertex (or its target vertex), this vertex can be removed as well. The flags *remove_source* and *remove_target* indicate whether the endpoints of e should be removed, or whether they should be left as isolated vertices in the arrangement. If the operation causes two faces to merge, the merged face is returned. Otherwise, the face to which the edge was incident is returned.

See Also

ArrangementDcel (page 1236)

Arr_default_dcel<Traits> (page 1250)

ArrangementTraits_2 (page 1263)

Arrangement_2<Traits,Dcel> (page 1201)

insertion functions (page 1220, page 1221)

removal functions (page 1322)

overlaying arrangements (page 1229)

CGAL::remove_curve

```
#include <CGAL/Arrangement_with_history_2.h>
```

```
template <class Traits, class Dcel>
```

```
Size      remove_curve( Arrangement_with_history_2<Traits,Dcel>& arr,  
                        typename Arrangement_with_history_2<Traits,Dcel>::Curve_handle ch)
```

Removes a curve, specified by its handle *ch*, from the arrangement, by deleting all the edges it induces. The function returns the number of deleted edges.

ArrWithHistoryInputFormatter

A model for the ArrWithHistoryInputFormatter concept supports a set of functions that enable reading an arrangement-with-history instance from an input stream using a specific format.

Refines

ArrangementInputFormatter

Types

ArrWithHistoryInputFormatter::Arr_with_history_2

the type of arrangement to input.

typedef typename Arrangement_2::Curve_2

Curve_2; the inducing curve type.

Formatted Input Functions

void inf.read_curves_begin()

reads a message indicating the beginning of the inducing curves.

void inf.read_curves_end()

reads a message indicating the end of the inducing curves.

void inf.read_curve_begin()

reads a message indicating the beginning of a single curve record.

void inf.read_curve_end()

reads a message indicating the end of a single curve record.

void inf.read_curve(Curve_2& c)

reads a curve.

void inf.read_induced_edges_begin()

reads a message indicating the beginning of the set of edges induced by the current curve.

void inf.read_induced_edges_end()

reads a message indicating the end of the induced edges set.

Has Models

Arr_with_history_text_formatter<*ArrFormatter*> (page [1327](#))

ArrWithHistoryOutputFormatter

A model for the ArrWithHistoryOutputFormatter concept supports a set of functions that enable writing an arrangement-with-history instance to an output stream using a specific format.

Refines

ArrangementOutputFormatter

Types

ArrWithHistoryOutputFormatter:: Arr_with_history_2

the type of arrangement to output.

typedef typename Arrangement_2::Curve_2

Curve_2; the inducing curve type.

Formatted Output Functions

void outf.write_curves_begin()

writes a message indicating the beginning of the inducing curves.

void outf.write_curves_end()

writes a message indicating the end of the inducing curves.

void outf.write_curve_begin()

writes a message indicating the beginning of a single curve record.

void outf.write_curve_end()

writes a message indicating the end of a single curve record.

void outf.write_curve(Curve_2 c)

writes a curve.

void outf.write_induced_edges_begin()

writes a message indicating the beginning of the set of edges induced by the current curve.

void *outf.write_induced_edges_end()*

writes a message indicating the end of the induced edges set.

Has Models

Arr_with_history_text_formatter<*ArrFormatter*> (page [1327](#))

CGAL::Arr_with_history_text_formatter<ArrFormatter>

Definition

Arr_with_history_text_formatter<ArrFormatter> defines the format of an arrangement in an input or output stream (typically a file stream), thus enabling reading and writing an arrangement-with-history instance using a simple text format.

The *ArrFormatter* parameter serves as a base class for *Arr_with_history_text_formatter<ArrFormatter>* and must be a model of the *ArrangementInputFormatter* and the *ArrangementOutputFormatter* concepts. It is used to read or write the base arrangement, while the derived class is responsible for reading and writing the set of curves inducing the arrangement and maintaining the relations between these curves and the edges they induce.

#include <CGAL/IO/Arr_with_history_text_formatter.h>

Is Model for the Concepts

ArrWithHistoryInputFormatter
ArrWithHistoryOutputFormatter

See Also

read (page [1234](#))
write (page [1235](#))

Chapter 18

2D Intersection of Curves

Baruch Zukerman, Ron Wein, and Efi Fogel

Contents

18.1 Introduction	1329
18.1.1 A Simple Program	1329

18.1 Introduction

Let $\mathcal{C} = \{C_1, C_2, \dots, C_n\}$ be a set of curves. We wish to compute all intersection points between two curves in the set in an output-sensitive manner, without having to go over all $O(n^2)$ curve pairs. To this end, we sweep an imaginary line l from $x = -\infty$ to $x = \infty$ over the plane. While sweeping the plane, we keep track of the order of curves intersecting it. This order changes at a finite number of *event points*, such that we only have to calculate the intersection points between two curves when they become contiguous. For more details on the *sweep-line algorithm* see, for example, [dBvKOS00, Chapter 2].

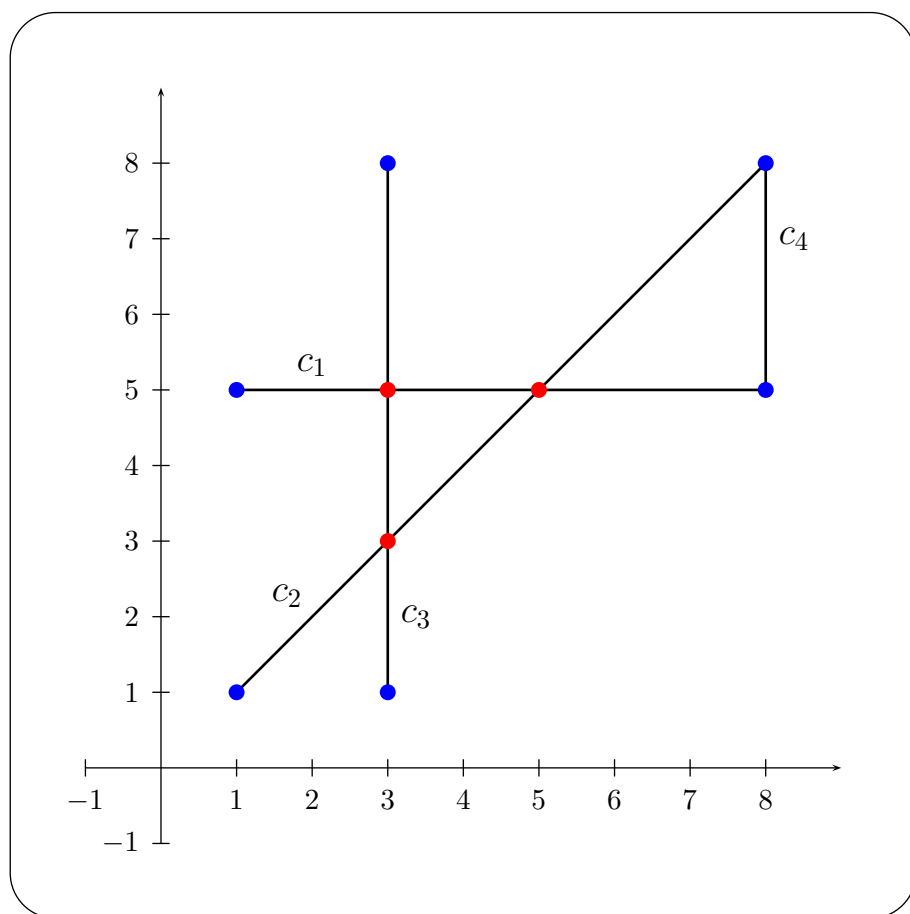
This chapter describes three functions implemented using the sweep-line algorithm: given a collection of input curves, compute all intersection points, compute the set of subcurves that are pairwise interior-disjoint induced by them, and checking whether there is at least one pair of curves among them that intersect in their interior.

The implementation is robust. It supports general curves and handles all degenerate cases, including overlapping curves, vertical segments, and tangency between curves. The robustness of the algorithm is guaranteed if the functions are instantiated with a traits class that employs certified computations. This traits class must be a model of the *ArrangementTraits_2* concept — see the Chapter 17 for more details.

The complexity of the sweep-line algorithm is $O((n+k)\log n)$ where n is the number of the input curves and k is the number of intersection points induced by these curves.

18.1.1 A Simple Program

The simple program listed below computes intersection points induced by a set of four input segments illustrated in figure 18.1.



```

//! \file examples/Arrangement_2/ex_sweep_line.C
// Computing intersection points among curves using the sweep line.

#include <CGAL/Cartesian.h>
#include <CGAL/MP_Float.h>
#include <CGAL/Quotient.h>
#include <CGAL/Sweep_line_2_algorithms.h>
#include <list>

typedef CGAL::Quotient<CGAL::MP_Float>      NT;
typedef CGAL::Cartesian<NT>                Kernel;
typedef Kernel::Point_2                     Point_2;
typedef Kernel::Segment_2                   Segment_2;

int main()
{
    // Construct the input segments.
    Segment_2 segments[] = {Segment_2 (Point_2 (1, 5), Point_2 (8, 5)),
                             Segment_2 (Point_2 (1, 1), Point_2 (8, 8)),
                             Segment_2 (Point_2 (3, 1), Point_2 (3, 8)),
                             Segment_2 (Point_2 (8, 5), Point_2 (8, 8))};

    // Compute all intersection points.
    std::list<Point_2>      pts;

    CGAL::get_intersection_points (segments, segments + 4,
                                   std::back_inserter (pts));

    // Print the result.
    std::copy (pts.begin(), pts.end(),
               std::ostream_iterator<Point_2>(std::cout, "\n"));

    return (0);
}

```

Design and Implementation History

The current version of the sweep-line algorithm was written by Baruch Zukerman, based on previous implementations by Ester Ezra and Tali Zvi.

2D Intersection of Curves

Reference Manual

Baruch Zukerman and Ron Wein

This chapter describes three functions implemented using the sweep-line algorithm: given a collection \mathcal{C} of planar curves, compute all intersection points among them, obtain the set of maximal pairwise interior-disjoint subcurves of the curves in \mathcal{C} , or check whether there is at least one pair of curves in \mathcal{C} that intersect in their interior.

The first two operations are performed in an output-sensitive manner.

Functions

CGAL::get_intersection_points page [1334](#)
CGAL::get_subcurves page [1335](#)
CGAL::do_curves_intersect page [1336](#)

18.2 Alphabetical List of Reference Pages

do_curves_intersect page [1336](#)
get_intersection_points page [1334](#)
get_subcurves page [1335](#)

CGAL::get_intersection_points

```
#include <CGAL/Sweep_line_2_algorithms.h>

template <class InputIterator, class OutputIterator, class Traits>
OutputIterator      get_intersection_points( InputIterator curves_begin,
                                           InputIterator curves_end,
                                           OutputIterator points,
                                           bool report_endpoints = false,
                                           Traits traits = Default_traits())
```

given a range of curves, compute all intersection points between two (or more) input curves. If the flag *report_endpoints* is *true*, then the curve endpoints are reported as well. The *Traits* type must be a model of the *ArrangementTraits_2* concept, such that the value-type of *InputIterator* is *Traits::Curve_2*, and the value-type of *OutputIterator* is *Traits::Point_2*. The output points are reported in an increasing *xy*-lexicographical order.

CGAL::get_subcurves

```
#include <CGAL/Sweep_line_2_algorithms.h>
```

```
template <class InputIterator, class OutputIterator, class Traits>
OutputIterator      get_subcurves( InputIterator curves_begin,
                                   InputIterator curves_end,
                                   OutputIterator subcurves,
                                   bool multiple_overlaps = false,
                                   Traits traits = Default_traits())
```

given a range of curves, compute all x -monotone subcurves that are pairwise disjoint in their interior, as induced by the input curves. If the flag *multiple_overlaps* is *true*, then a subcurve that represents an overlap of k input curves is reported k times; otherwise, each subcurve is reported only once. The *Traits* type must be a model of the *ArrangementTraits_2* concept, such that the value-type of *InputIterator* is *Traits::Curve_2*, and the value-type of *OutputIterator* is *Traits::X_monotone_curve_2*.

CGAL::do_curves_intersect

```
#include <CGAL/Sweep_line_2_algorithms.h>
```

```
template <class InputIterator, class Traits>  
bool do_curves_intersect( InputIterator curves_begin,  
                          InputIterator curves_end,  
                          Traits traits = Default_traits())
```

given a range of curves, check whether there is at least one pair of curves that intersect in their interior. The function returns *true* if such a pair is found, and *false* if all curves are pairwise disjoint in their interior. The *Traits* type must be a model of the *ArrangementTraits_2* concept, such that the value-type of *InputIterator* is *Traits::Curve_2*.

Chapter 19

2D Snap Rounding

Eli Packer

Contents

19.1 Introduction	1337
19.2 What is Snap Rounding/Iterated Snap Rounding	1338
19.3 Terms and Software Design	1338
19.4 Four Line Segment Example	1338

19.1 Introduction

Snap Rounding (SR, for short) is a well known method for converting arbitrary-precision arrangements of segments into a fixed-precision representation [GGHT97, GM98, Hob99]. In the study of robust geometric computing, it can be classified as a finite precision approximation technique. Iterated Snap Rounding (ISR, for short) is a modification of SR in which each vertex is at least half-the-width-of-a-pixel away from any non-incident edge [HP02]. This package supports both methods. Algorithmic details and experimental results are given in [HP02].

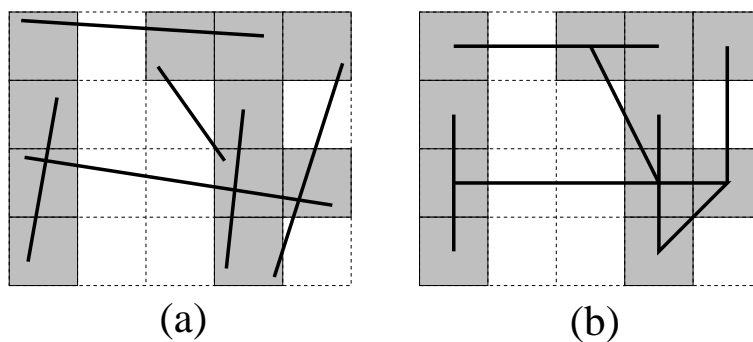


Figure 19.1: An arrangement of segments before (a) and after (b) SR (hot pixels are shaded)

19.2 What is Snap Rounding/Iterated Snap Rounding

Given a finite collection S of segments in the plane, the arrangement of S denoted $\mathcal{A}(S)$ is the subdivision of the plane into vertices, edges, and faces induced by S . A *vertex* of the arrangement is either a segment endpoint or the intersection of two segments. Given an arrangement of segments whose vertices are represented with arbitrary-precision coordinates, the SR procedure `snap_rounding_2<Traits,InputIterator,OutputContainer>` proceeds as follows. We tile the plane with a grid of unit squares, *pixels*, each centered at a point with integer coordinates. A pixel is *hot* if it contains a vertex of the arrangement. Each vertex of the arrangement is replaced by the center of the hot pixel containing it and each edge e is replaced by the polygonal chain through the centers of the hot pixels met by e , in the same order as they are met by e . Figure 19.1 demonstrates the results of SR.

In a snap-rounded arrangement, the distance between a vertex and a non-incident edge can be extremely small compared with the width of a pixel in the grid used for rounding. ISR is a modification of SR which makes a vertex and a non-incident edge well separated (the distance between each is at least half-the-width-of-a-pixel). However, the guaranteed quality of the approximation in ISR degrades. Figure 19.2 depicts the results of SR and ISR on the same input. Conceptually, the ISR procedure is equivalent to repeated application of SR, namely we apply SR to the original set of segments, then we use the output of SR as input to another round of SR and so on until all the vertices are well separated from non-incident edges. Algorithmically we operate differently, as this repeated application of SR would have resulted in an efficient overall process. The algorithmic details are given in [HP02].

19.3 Terms and Software Design

Our package supports both schemes, implementing the algorithm described in [HP02]. Although the paper only describes an algorithm for ISR, it is easy to derive an algorithm for SR, by performing only the first rounding level for each segment.

The input to the program is a set S of n segments, $S = \{s_1, \dots, s_n\}$ and the output is a set G of n polylines, with a polyline g_i for each input segments s_i . An input segment is given by the coordinates of its endpoints. An output polyline is given by the ordered set of vertices v_0, \dots, v_k along the polyline. The polyline consists of the segments $(v_0v_1), \dots, (v_{k-1}v_k)$.

There are three template parameters: *Traits* is the underlying geometry, i.e., the number type used and the coordinate representation. *InputIterator* is the type of the iterators that point to the first and after-the-last elements of the input. Finally, *OutputContainer* is the type of the output container.

Since the algorithm requires kernel functionalities such as the rounding to the center of a pixel, a special traits class must be provided. The precise description of the requirements is given by the concept `SnapRoundingTraits_2`. The class `Snap_rounding_traits_2` is a model of this concept.

19.4 Four Line Segment Example

The following example generates an ISR representation of an arrangement of four line segments. In particular it produces a list of points that are the vertices of the resulting polylines in a plane tiled with one-unit square pixels.

```
#include <CGAL/basic.h>
```

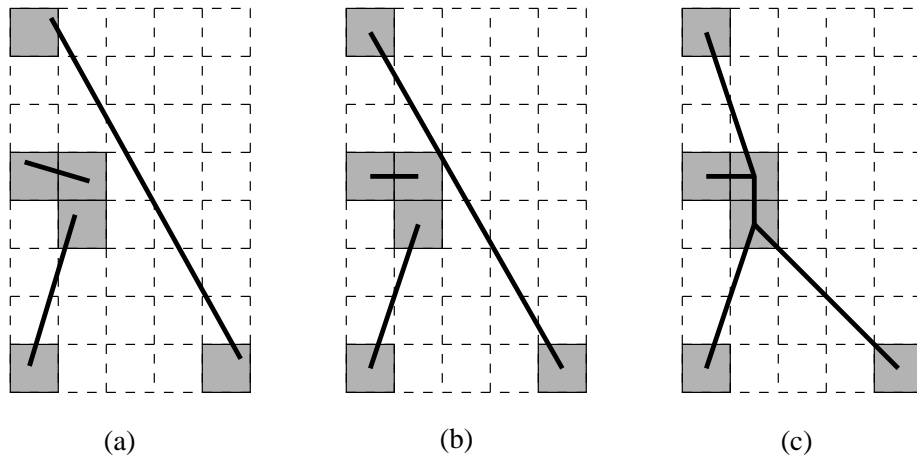


Figure 19.2: An arrangement of segments before (a), after SR (b) and ISR (c) (hot pixels are shaded).

```
#include <CGAL/Cartesian.h>
#include <CGAL/Quotient.h>
#include <CGAL/MP_Float.h>
#include <CGAL/Snap_rounding_traits_2.h>
#include <CGAL/Snap_rounding_2.h>

typedef CGAL::Quotient<CGAL::MP_Float>          Number_type;
typedef CGAL::Cartesian<Number_type>           Kernel;
typedef CGAL::Snap_rounding_traits_2<Kernel>    Traits;
typedef Kernel::Segment_2                      Segment_2;
typedef Kernel::Point_2                       Point_2;
typedef std::list<Segment_2>                   Segment_list_2;
typedef std::list<Point_2>                     Polyline_2;
typedef std::list<Polyline_2>                  Polyline_list_2;

int main()
{
    Segment_list_2 seg_list;
    Polyline_list_2 output_list;

    seg_list.push_back(Segment_2(Point_2(0, 0), Point_2(10, 10)));
    seg_list.push_back(Segment_2(Point_2(0, 10), Point_2(10, 0)));
    seg_list.push_back(Segment_2(Point_2(3, 0), Point_2(3, 10)));
    seg_list.push_back(Segment_2(Point_2(7, 0), Point_2(7, 10)));

    // Generate an iterated snap-rounding representation, where the centers of
    // the hot pixels bear their original coordinates, using 5 kd trees:
    CGAL::snap_rounding_2<Traits, Segment_list_2::const_iterator, Polyline_list_2>
        (seg_list.begin(), seg_list.end(), output_list, 1.0);

    int counter = 0;
    Polyline_list_2::const_iterator iter1;
    for (iter1 = output_list.begin(); iter1 != output_list.end(); ++iter1) {
        std::cout << "Polyline number " << ++counter << ":\n";
        Polyline_2::const_iterator iter2;
        for (iter2 = iter1->begin(); iter2 != iter1->end(); ++iter2)
```

```

        std::cout << "      (" << iter2->x() << ":" << iter2->y() << ")\n";
    }

    return(0);
}

```

This program generates four polylines, one for each input segment. The exact output follows:

Polyline number 1:

```

(0/4:0/4)
(12/4:12/4)
(20/4:20/4)
(28/4:28/4)
(40/4:40/4)

```

Polyline number 2:

```

(0/4:40/4)
(12/4:28/4)
(20/4:20/4)
(28/4:12/4)
(40/4:0/4)

```

Polyline number 3:

```

(12/4:0/4)
(12/4:12/4)
(12/4:28/4)
(12/4:40/4)

```

Polyline number 4:

```

(28/4:0/4)
(28/4:12/4)
(28/4:28/4)
(28/4:40/4)

```

The package is supplied with a graphical demo program that opens a window, allows the user to edit segments dynamically, applies a selected snap-rounding procedures, and displays the result onto the same window (see `<CGAL_ROOT>/demo/Snap_rounding_2/demo.C`).

2D Snap Rounding Reference Manual

Eli Packer

Snap Rounding is a method for converting arbitrary-precision arrangements of segments into a fixed-precision representation.

19.5 Alphabetical List of Reference Pages

<i>SnapRoundingTraits_2</i>	page 1344
<i>snap_rounding_2</i>	page 1342
<i>Snap_rounding_traits_2<Kernel></i>	page 1347

CGAL::snap_rounding_2

Definition

Snap Rounding (SR, for short) is a well known method for converting arbitrary-precision arrangements of segments into a fixed-precision representation [GGHT97, GM98, Hob99]. In the study of robust geometric computing, it can be classified as a finite precision approximation technique. Iterated Snap Rounding (ISR, for short) is a modification of SR in which each vertex is at least half-the-width-of-a-pixel away from any non-incident edge [HP02]. This package supports both methods. Algorithmic details and experimental results are given in [HP02].

Given a finite collection \mathcal{S} of segments in the plane, the arrangement of \mathcal{S} denoted $\mathcal{A}(\mathcal{S})$ is the subdivision of the plane into vertices, edges, and faces induced by \mathcal{S} . A *vertex* of the arrangement is either a segment endpoint or the intersection of two segments. Given an arrangement of segments whose vertices are represented with arbitrary-precision coordinates, SR proceeds as follows. We tile the plane with a grid of unit squares, *pixels*, each centered at a point with integer coordinates. A pixel is *hot* if it contains a vertex of the arrangement. Each vertex of the arrangement is replaced by the center of the hot pixel containing it and each edge e is replaced by the polygonal chain through the centers of the hot pixels met by e , in the same order as they are met by e .

In a snap-rounded arrangement, the distance between a vertex and a non-incident edge can be extremely small compared with the width of a pixel in the grid used for rounding. ISR is a modification of SR which makes a vertex and a non-incident edge well separated (the distance between each is at least half-the-width-of-a-pixel). However, the guaranteed quality of the approximation in ISR degrades. For more details on ISR see [HP02].

The traits used here must support (arbitrary-precision) rational number type as this is a basic requirement of SR.

```
#include <CGAL/Snap_rounding_2.h>
```

```
template < class Traits, class InputIterator, class OutputContainer >
void      snap_rounding_2( InputIterator begin,
                           InputIterator end,
                           OutputContainer& output_container,
                           typename Traits::FT pixel_size,
                           bool do_isr = true,
                           bool int_output = true,
                           unsigned int number_of_kd_trees = 1)
```

The first two parameters denote the first and after-the-last iterators of the input segments. The third parameter is a reference to a container of the output polylines. Since a polyline is composed of a sequence of points, a polyline is a container itself. The fifth parameter determines whether to apply ISR or SR.

The forth parameter denotes the pixel size w . The plane will be tiled with square pixels of width w such that the origin is the center of a pixel. The sixth parameter denotes the output representation. If the value of the sixth parameter is *true* then the centers of pixels constitute the integer grid, and hence the vertices of the output polylines will be integers. For example, the coordinates of the center of the pixel to the right of the pixel containing the origin will be $(1,0)$ regardless of the pixel width. If the value of the sixth parameter is *false*, then the centers of hot pixels (and hence the vertices of the output polylines) will bear their original coordinates, which may not necessarily be integers. In the latter case, the coordinates of the center of the pixel to the right of the pixel containing the origin, for example, will be $(w,0)$.

The seventh (and last) parameter is briefly described next; for a detailed description see [HP02].

A basic query used in the algorithm is to report the hot pixels of size w that a certain segment s intersects. An alternative way to do the same is to query the hot pixels' centers contained in a Minkowski sum of s with a pixel of width w centered at the origin; we denote this Minkowski sum by $M(s)$. Since efficiently implementing this kind of query is difficult, we use an orthogonal range-search structure instead. We query with the bounding box $B(M(s))$ of $M(s)$ in a two-dimensional kd-tree which stores the centers of hot pixels. Since $B(M(s))$ in general is larger than $M(s)$, we still need to filter out the hot pixels which do not intersect s .

While this approach is easy to implement with CGAL, it may incur considerable overhead since the area of $B(M(s))$ may be much larger than the area of $M(s)$, possibly resulting in many redundant hot pixels to filter out. Our heuristic solution, which we describe next, is to use a cluster of kd-trees rather than just one. The cluster includes several kd-trees, each has the plane, and hence the centers of hot pixels, rotated by a different angle in the first quadrant of the plane; for our purpose, a rotation by angles outside this quadrant is symmetric to a rotation by an angle in the first quadrant.

Given a parameter c , the angles of rotation are $(i-1)\frac{\pi}{2c}$, $i = 1, \dots, c$, and we construct a kd-tree corresponding to each of these angles. Then for a query segment s , we choose the kd-tree for which the area of $B(M(s))$ is the smallest, in order to (potentially) get less hot pixels to filter out. Since constructing many kd-trees may be costly, our algorithm avoids building a kd-tree which it expects to be queried a relatively small number of times (we estimate this number in advance). How many kd-trees should be used? It is difficult to provide a simple answer for that. There are inputs for which the time to build more than one kd-tree is far greater than the time saved by having to filter out less hot pixels (sparse arrangements demonstrate this behavior), and there are inputs which benefit from using several kd-trees. Thus, the user can control the number of kd-trees with the parameter *number_of_kd_trees*. Typically, but not always, one kd-tree (the default) is sufficient.

Precondition: *pixel_size* must have a positive value and *number_of_kd_trees* must be a positive integer.

SnapRoundingTraits_2

Definition

The concept `SnapRoundingTraits_2` lists the set of requirements that must be fulfilled by an instance of the *Traits* template-parameter of the function `snap_rounding_2<Traits, InputIterator, OutputContainer>()`. This concept provides the types of the geometric primitives used in this class and some function object types for the required predicates on those primitives.

Refines

This concept refines the standard concepts `DefaultConstructible`, `Assignable` and `CopyConstructible`. It also refines the concept `SweepLineTraits_2` (page ??). An instance of this concept is used as the traits class for the `Sweep_line_2.get_intersection_points()` operation. The requirements listed below are induced by components of the `snap_rounding_2()` function other than the call to `Sweep_line_2.get_intersection_points()`. Naturally, some of them may already be listed in `SweepLineTraits_2`.

Types

<code>SnapRoundingTraits_2:: FT</code>	The number type. This type must fulfill the requirements on <i>FieldNumberType</i>
<code>SnapRoundingTraits_2:: Point_2</code>	The point type.
<code>SnapRoundingTraits_2:: Segment_2</code>	The segment type.
<code>SnapRoundingTraits_2:: Iso_rectangle_2</code>	The iso-rectangle type.
<code>SnapRoundingTraits_2:: Construct_vertex_2</code>	Function object. Must provide the operator <i>Point_2 operator()(Segment_2 seg, int i)</i> , which returns the source or target of <i>seg</i> . If <i>i</i> modulo 2 is 0, the source is returned, otherwise the target is returned.
<code>SnapRoundingTraits_2:: Construct_segment_2</code>	Function object. Must provide the operator <i>Segment_2 operator()(Point_2 p, Point_2 q)</i> , which introduces a segment with source <i>p</i> and target <i>q</i> . The segment is directed from the source towards the target.
<code>SnapRoundingTraits_2:: Construct_iso_rectangle_2</code>	Function object. Must provide the operator <i>Iso_rectangle_2 operator()(Point_2 left, Point_2 right, Point_2 bottom, Point_2 top)</i> , which introduces an iso-oriented rectangle for whose minimal <i>x</i> coordinate is the one of <i>left</i> , the maximal <i>x</i> coordinate is the one of <i>right</i> , the minimal <i>y</i> coordinate is the one of <i>bottom</i> , the maximal <i>y</i> coordinate is the one of <i>top</i> .
<code>SnapRoundingTraits_2:: To_double</code>	Function object. Must provide the operator <i>double operator()(FT)</i> , which computes an approximation of a given number of type <i>FT</i> . The precision of this operation is of not high significance, as it is only used in the implementation of the heuristic technique to exploit a cluster of kd-trees rather than just one.

<i>SnapRoundingTraits_2:: Compare_x_2</i>	Function object. Must provide the operator <i>Comparison_result operator()(Point_2 p, Point_2 q)</i> which returns <i>SMALLER</i> , <i>EQUAL</i> or <i>LARGER</i> according to the <i>x</i> -ordering of points <i>p</i> and <i>q</i> .
<i>SnapRoundingTraits_2:: Compare_y_2</i>	Function object. Must provide the operator <i>Comparison_result operator()(Point_2 p, Point_2 q)</i> which returns <i>SMALLER</i> , <i>EQUAL</i> or <i>LARGER</i> according to the <i>y</i> -ordering of points <i>p</i> and <i>q</i> .
<i>SnapRoundingTraits_2:: Snap_2</i>	Rounds a point to a center of a pixel (unit square) in the grid used by the Snap Rounding algorithm. Note that no conversion to an integer grid is done yet. Must have the syntax <i>void operator()(Point_2 p, FT pixel_size, FT &x, FT &y)</i> where <i>p</i> is the input point, <i>pixel_size</i> is the size of the pixel of the grid, and <i>x</i> and <i>y</i> are the <i>x</i> and <i>y</i> -coordinates of the rounded point respectively.
<i>SnapRoundingTraits_2:: Integer_grid_point_2</i>	Convert coordinates into an integer representation where one unit is equal to pixel size. For instance, if a point has the coordinates (3.7, 5.3) and the pixel size is 0.5, then the new point will have the coordinates of (7, 10). Note, however, that the number type remains the same here, although integers are represented. Must have the syntax <i>Point_2 operator()(Point_2 p, NT pixel_size)</i> where <i>p</i> is the converted point and <i>pixel_size</i> is the size of the pixel of the grid.
<i>SnapRoundingTraits_2:: Minkowski_sum_with_pixel_2</i>	Returns the vertices of a polygon, which is the Minkowski sum of a segment and a square centered at the origin with edge size <i>pixel edge</i> . Must have the syntax <i>void operator()(std::list<Point_2>& vertices_list, Segment_2 s, NT unit_square)</i> where <i>vertices_list</i> is the list of the vertices of the Minkowski sum polygon, <i>s</i> is the input segment and <i>unit_square</i> is the edge size of the pixel.

Creation

This concept refines the standard concepts `DefaultConstructible`, `Assignable` and `CopyConstructible`.

Operations

The following functions construct the required function objects occasionally referred as functors listed above.

<i>Construct_vertex_2</i>	<i>traits.construct_vertex_2_object()</i>
<i>Construct_segment_2</i>	<i>traits.construct_segment_2_object()</i>
<i>Construct_iso_rectangle_2</i>	<i>traits.construct_iso_rectangle_2_object()</i>
<i>Compare_x_2</i>	<i>traits.compare_x_2_object()</i>
<i>Compare_y_2</i>	<i>traits.compare_y_2_object()</i>
<i>Snap_2</i>	<i>traits.snap_2_object()</i>
<i>Integer_grid_point_2</i>	<i>traits.integer_grid_point_2_object()</i>
<i>Minkowski_sum_with_pixel_2</i>	<i>traits.minkowski_sum_with_pixel_2_object()</i>

Has Models

CGAL::Snap_rounding_traits<*Kernel*>

See Also

CGAL::Snap_rounding_2<*Traits*>

CGAL::Snap_rounding_traits_2<Kernel>

The class *Snap_rounding_traits_2<Kernel>* is a model of the *SnapRoundingTraits_2* concept, and is the only traits class supplied with the package. This class should be instantiated with an exact geometric kernel that conforms to the CGAL kernel-concept, such as the *Cartesian<gmpq>* kernel.

This geometric kernel must provide an (arbitrary-precision) rational number type (*FT*), *Point_2*, *Segment_2* and *Iso_rectangle_2*. It should be possible to cast numbers of the number type *FT* to double-precision representation. That is, the function *CGAL::to_double(FT)* must be supported.

The *CGAL::to_double()* function is used to implement the operation that rounds the coordinates of a point to a center of a pixel. This operation is one of the traits-concept requirement. The coordinates are converted to double, rounded down to the nearest grid point, and finally adjusted to lie on a center of a pixel. Notice that if *CGAL::to_double()* returns the closet value, then when it rounds up a given coordinate, the resulting ISR, may be imprecise, and the distance between some vertex and a non-incident edge can be slightly less than the guaranteed half-the width-of-a-pixel.

```
#include <CGAL/Snap_rounding_traits_2.h>
```

Is Model for the Concepts

SnapRoundingTraits_2

Part VII

Triangulations and Delaunay Triangulations

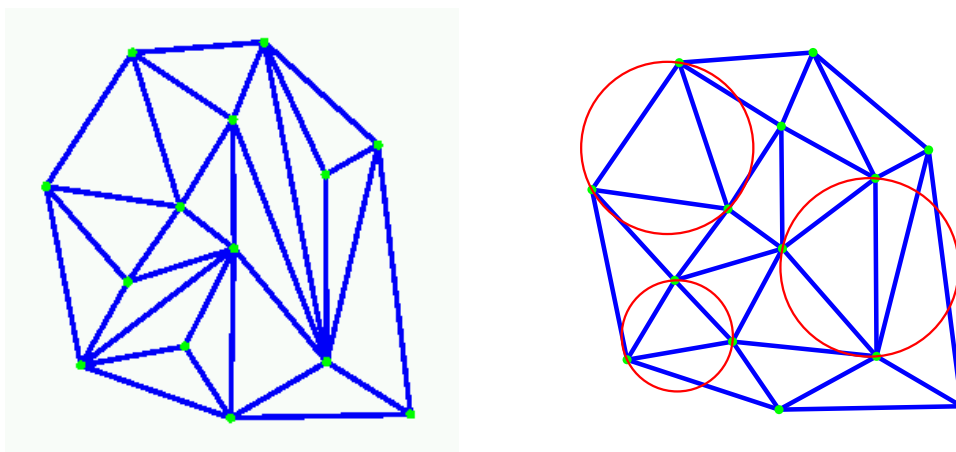
Chapter 20

2D Triangulations

Mariette Yvinec

Contents

20.1 Definitions	1352
20.2 Representation	1353
20.3 Software Design	1354
20.4 Basic Triangulations	1356
20.4.1 Description	1356
20.4.2 Example of a Basic Triangulation	1358
20.5 Delaunay Triangulations	1359
20.5.1 Description	1359
20.5.2 Example : a Delaunay Terrain	1360
20.5.3 Example : Voronoi Diagram	1360
20.6 Regular Triangulations	1361
20.6.1 Description	1361
20.6.2 Example : a Regular Triangulation	1363
20.7 Constrained Triangulations	1364
20.8 Constrained Delaunay Triangulations	1365
20.8.1 Example : a constrained Delaunay triangulation	1366
20.9 Constrained Triangulations Plus	1367
20.9.1 Example : Building a triangulated arrangement of segments	1367
20.10 The Triangulation Hierarchy	1368
20.10.1 Examples of the Use of a Triangulation Hierarchy	1369
20.11 Flexibility: Using Customized Vertices and Faces	1371
20.12 Design and Implementation History	1374



This chapter describes the two dimensional triangulations of CGAL. Section 20.1 recalls the main definitions about triangulations. Sections 20.2 discusses the way two-dimensional triangulations are represented in CGAL. Section 20.3 presents the overall software design of the 2D triangulations package. The next sections present the different two dimensional triangulations classes available in CGAL : basic triangulations (section 20.4), Delaunay triangulations (Section 20.5), regular triangulations (Section 20.6), constrained triangulations (Section 20.7), and constrained Delaunay triangulations (Section 20.8). Section 20.9 describes a class which implements a constrained or constrained Delaunay triangulation with an additional data structure to describe how the constraints are refined by the edges of the triangulations. Section 20.10 describes a hierarchical data structure for fast point location queries. At last, Section 20.11 explains how the user can benefit from the flexibility of CGAL triangulations using customized classes for faces and vertices.

20.1 Definitions

A two dimensional triangulation can be roughly described as a set T of triangular facets such that :

- two facets either are disjoint or share a lower dimensional face (edge or vertex).
- the set of facets in T is connected for the adjacency relation.
- the domain U_T which is the union of facets in T has no singularity.

More precisely, a triangulation can be described as a simplicial complex. Let us first record a few definitions.

A simplicial complex is a set T of simplices such that

- any face of a simplex in T is a simplex in T
- two simplices in T either are disjoint or share a common subspace.

The dimension d of a simplicial complex is the maximal dimension of its simplices.

A simplicial complex T is pure if any simplex of T is included in a simplex of T with maximal dimension.

Two simplices in T with maximal dimension d are said to be adjacent if they share a $(d - 1)$ dimensional subspace. A simplicial complex is connected if the adjacency relation defines a connected graph over the set of simplices of T with maximal dimension.

The union U_T of all simplices in T is called the domain of T . A point p in the domain of T is said to singular if its surrounding in U_T is neither a topological ball nor a topological disc.

Then, a two dimensional triangulation can be described as a two dimensional simplicial complex that is pure, connected and without singularity.

Each facet of a triangulation can be given an orientation which in turn induces an orientation on the edges incident to that facet. The orientation of two adjacent facets are said to be consistent if they induce opposite orientations on their common incident edge. A triangulation is said to be orientable if the orientation of each facet can be chosen in such a way that all pairs of incident facets have consistent orientations.

The data structure underlying CGAL triangulations allows to represent the combinatorics of any orientable two dimensional triangulations without boundaries. On top of this data structure, the 2D triangulations classes take care of the geometric embedding of the triangulation and are designed to handle planar triangulations. The plane of the triangulation may be embedded in a higher dimensional space.

The triangulations of CGAL are complete triangulations which means that their domain is the convex hull of their vertices. Because any planar triangulation can be completed, this is not a real restriction. For instance, a triangulation of a polygonal region can be constructed and represented as a subset of a constrained triangulation in which the region boundary edges have been input as constrained edges (see Section 20.7, 20.8 and 20.9).

Strictly speaking, the term *face* should be used to design a face of any dimension, and the two-dimensional faces of a triangulation should be properly called *facets*. However, following a common usage, we hereafter often call *faces*, the facets of a two dimensional triangulation.

20.2 Representation

The set of faces

A 2D triangulation of CGAL can be viewed as a planar partition whose bounded faces are triangular and cover the convex hull of the set of vertices. The single unbounded face of this partition is the complementary of the convex hull. In many applications, such as Kirkpatrick's hierarchy or incremental Delaunay construction, it is convenient to deal with only triangular faces. Therefore, a fictitious vertex, called the *infinite vertex* is added to the triangulation as well as *infinite edges* and *infinite faces* incident to it.. Each infinite edge is incident to the infinite vertex and to a vertex of the convex hull. Each infinite face is incident to the infinite vertex and to a convex hull edge.

Therefore, each edge of the triangulation is incident to exactly two faces and the set of faces of a triangulation is topologically equivalent to a two-dimensional sphere. This extends to lower dimensional triangulations arising in degenerate cases or when the triangulations as less than three vertices. Including the infinite faces, a one dimensional triangulation is a ring of edges and vertices topologically equivalent to a 1-sphere. A zero dimensional triangulation, whose domain is reduced to a single point, is represented by two vertices that is topologically equivalent to a 0-sphere.

Note that the *infinite vertex* has no significant coordinates and that no geometric predicate can be applied on it nor on an infinite face.

A representation based on faces and vertices

Because a triangulation is a set of triangular faces with constant-size complexity, triangulations are not implemented as a layer on top of a planar map. CGAL uses a proper internal representation of triangulations based on faces and vertices rather than on edges. Such a representation saves storage space and results in faster algorithms [BDTY00].

The basic elements of the representation are vertices and faces. Each triangular face gives access to its three incident vertices and to its three adjacent faces. Each vertex gives access to one of its incident faces and through that face to the circular list of its incident faces.

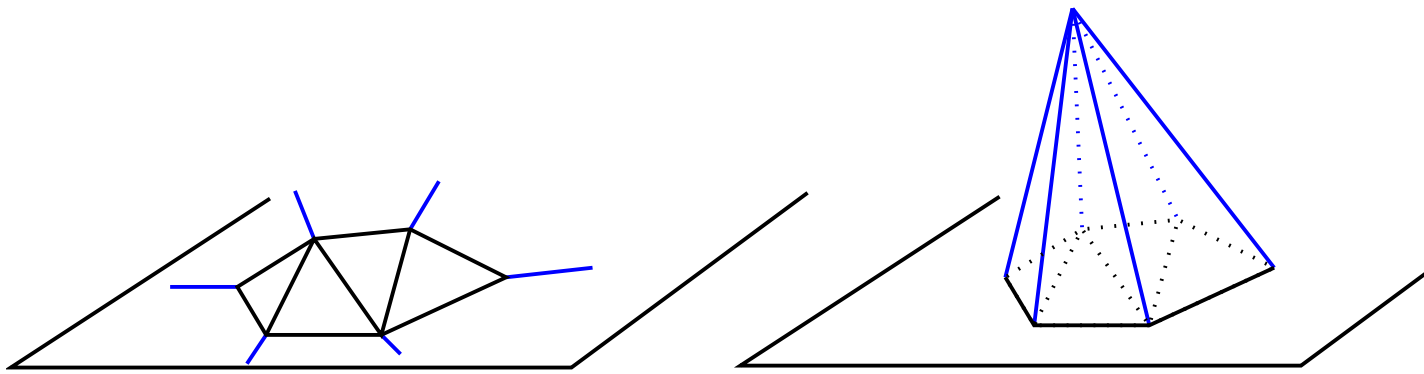


Figure 20.1: Infinite vertex and infinite faces

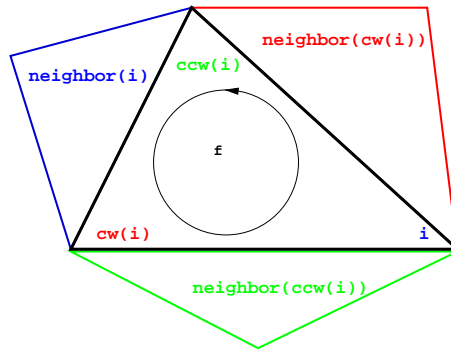


Figure 20.2: Vertices and neighbors.

The three vertices of a face are indexed with 0, 1 and 2 in counterclockwise order. The neighbors of a face are also indexed with 0,1,2 in such a way that the neighbor indexed by i is opposite to the vertex with the same index. See Figure 20.2, the functions $ccw(i)$ and $cw(i)$ shown on this figure compute respectively $i + 1$ and $i - 1$ modulo 3.

The edges are not explicitly represented, they are only implicitly represented through the adjacency relations of two faces. Each edge has two implicit representations : the edge of a face f which is opposed to the vertex indexed i , can be represented as well as an edge of the $neighbor(i)$ of f .

20.3 Software Design

The triangulations classes of CGAL provide high-level geometric functionalities such as location of a point in the triangulation, insertion or removal of a point. They are build as a layer on top of a data structure called the triangulation data structure. The triangulation data structure can be thought of as a container for the faces and vertices of the triangulation. This data structure also takes care of all the combinatorial aspects of the triangulation.

This separation between the geometric aspect and the combinatorial part is reflected in the software design by the fact that the triangulation classes have two template parameters :

- the first parameter stands for a **geometric traits** class providing the geometric primitives (points, segments

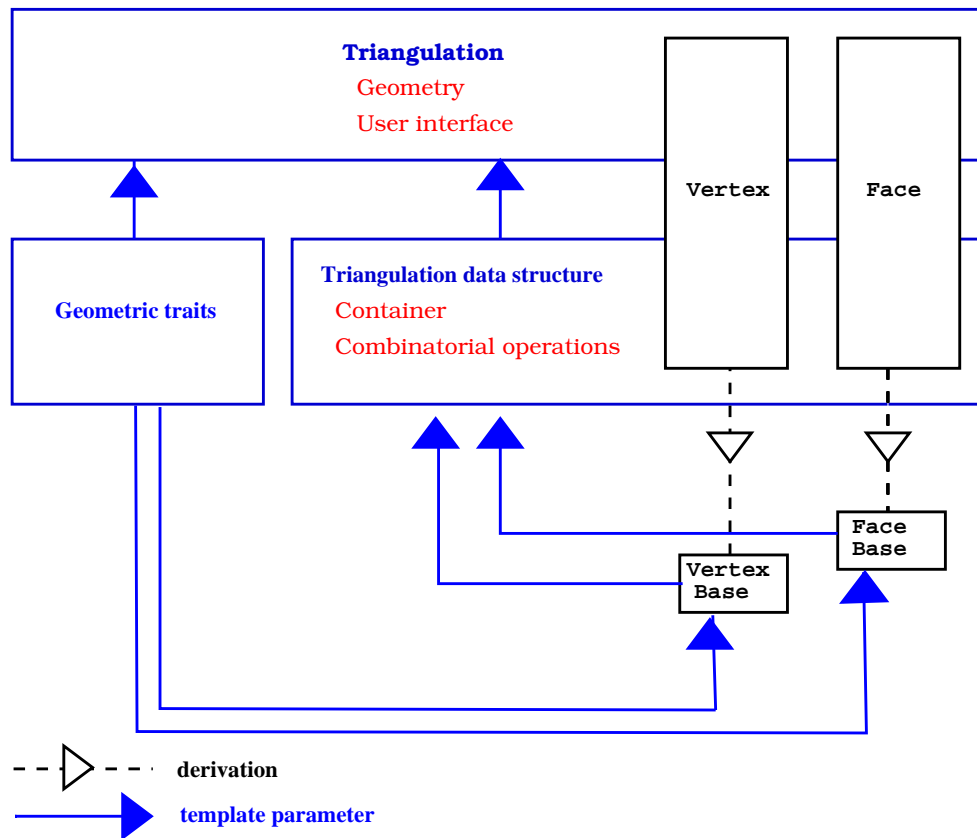


Figure 20.3: The triangulations software design.

and triangles) of the triangulation and the elementary operations (predicate or constructions) on those objects.

- the second parameter stands for a **triangulation data structure** class. The concept of triangulation data structure is described in Section 21.2 of Chapter 21. The triangulation data structure defines the types used to represent the faces and vertices of the triangulation, as well as additional types (handles, iterators and circulators) to access and visit the faces and vertices.

CGAL provides the class *Triangulation_data_structure_2*<*Vb*,*Fb*> as a default model of triangulation data structure. The class *Triangulation_data_structure_2*<*Vb*,*Fb*> has two template parameters standing for a vertex class and a face class. CGAL defines concepts for these template parameters and provide default models for these concepts. The vertex and base classes are templated by the geometric traits which allows them to have some knowledge of the geometric primitives of the triangulation. Those default vertex and face base classes can be replaced by user customized base classes in order, for example, to deal with additional properties attached to the vertices or faces of a triangulation. See section 20.11 for more details on the way to make use of this flexibility.

The Figure 20.3 summarizes the design of the triangulation package, showing the three layers (base classes, triangulation data structure and triangulation) forming this design.

The top triangulation level, responsible for the geometric embedding of the triangulation comes in different flavors according to the different kind of triangulations: basic, Delaunay, regular, constrained or constrained Delaunay. Each kind of triangulations correspond to a different class. Figure 20.4 summarizes the derivation dependencies of CGAL 2D triangulations classes. Any 2D triangulation class is parametrized by a geometric

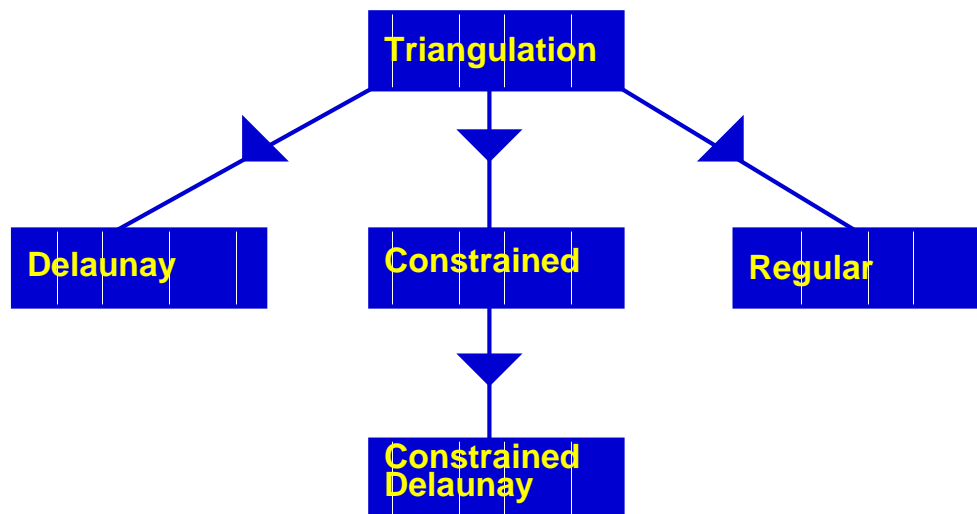


Figure 20.4: The derivation tree of 2D triangulations.

traits and a triangulation data structure. While a unique concept *TriangulationDataStructure_2* describes the triangulation data structure requirements for any triangulation class, the concept of geometric traits actually depends on the triangulation class. In general, the requirements for the vertex and face base classes are described by the basic concepts *TriangulationVertexBase_2* and *TriangulationFaceBase_2*. However, some triangulation classes requires base classes implementing refinements of the basic concepts.

20.4 Basic Triangulations

20.4.1 Description

The class *Triangulation_2*<*Traits*,*Tds*> serves as a base class for the other 2D triangulations classes and implements the user interface to a triangulation.

The vertices and faces of the triangulations are accessed through *handles*, *iterators* and *circulators*. A handle is a model of the concept *Handle* which basically offers the two dereference operators * and -> . A circulator is a type devoted to visit circular sequences. Handles are used whenever the accessed element is not part of a sequence. Iterators and circulators are used to visit all or parts of the triangulation.

The iterators and circulators are all bidirectional and non mutable. The circulators and iterators are convertible to the handles with the same value type, so that when calling a member function, any handle type argument can be replaced by an iterator or a circulator with the same value type.

The triangulation class allows to visit the vertices and neighbors of a face in clockwise or counterclockwise order.

There are circulators to visit the edges or faces incident to a given vertex or the vertices adjacent to it. Another circulator type allows to visit all the faces traversed by a given line. Circulators step through infinite features as well as through finite ones.

The triangulation class offers some iterators to visit all the faces, edges or vertices and also iterators to visit selectively the finite faces, edges or vertices.

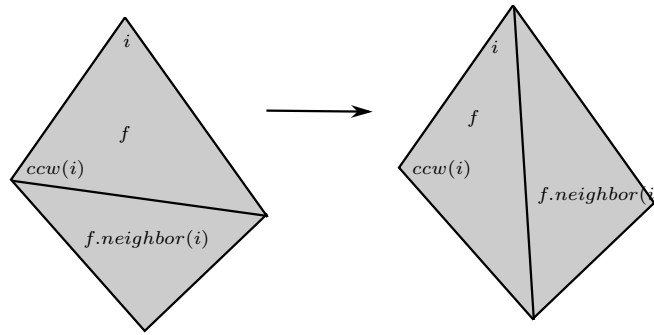


Figure 20.5: Flip.

Figure 20.6: Flip.

The triangulation class provides methods to test the infinite character of any feature, and also methods to test the presence in the triangulation of a particular feature (edge or face) given its vertices.

The triangulation class provides a method to locate a given point with respect to a triangulation. In particular, this method reports whether the point coincides with a vertex of the triangulation, lies on an edge, in a face or outside of the convex hull. In case of a degenerate lower dimensional triangulation, the query point may also lie outside the triangulation affine hull.

The triangulation class also provides methods to locate a point with respect to a given finite face of the triangulation or with respect to its circumcircle. The faces of the triangulation and their circumcircles have the counterclockwise orientation.

The triangulation can be modified by several functions : insertion of a point, removal of a vertex, flipping of an edge. The flipping of an edge is possible when the union of the two incident faces forms a convex quadrilateral (see Figure 20.6).

Implementation

Locate is implemented by a line walk. The walk begins at a vertex of the face which is given as an optional argument or at an arbitrary vertex of the triangulation if no optional argument is given. It takes time $O(n)$ in the worst case, but only $O(\sqrt{n})$ on average if the vertices are distributed uniformly at random. The class *Triangulation_hierarchy_2*<Traits,Tds>, described in section 20.10, implements a data structure designed to offer an alternate more efficient point location algorithm.

Insertion of a point is done by locating a face that contains the point, and splitting this face into three new faces. If the point falls outside the convex hull, the triangulation is restored by flips. Apart from the location, insertion takes a time $O(1)$. This bound is only an amortized bound for points located outside the convex hull.

Removal of a vertex is done by removing all adjacent triangles, and retriangulating the hole. Removal takes a time at most proportionnal to d^2 , where d is the degree of the removed vertex, which is $O(1)$ for a random vertex.

The face, edge, and vertex iterators on finite features are derived from their counterparts visiting all (finite and infinite) features which are themselves derived from the corresponding iterators of the triangulation data structure.

Geometric Traits

The geometric traits of a triangulation is required to provide the geometric objects (points, segments and triangles) building up the triangulation together with the geometric predicates on those objects. The required predicates are:

- comparison of the x or y coordinates of two points.
- the orientation test which computes the order type of three given point.

The concept *TriangulationTraits_2* describes the requirements for the geometric traits class of a triangulation. The CGAL kernel classes are models for this concept. The CGAL library also provides dedicated models of *TriangulationTraits_2* using the kernel geometric objects and predicates. These classes are themselves templated with a CGAL kernel and extract the required types and predicates from the kernel. The traits class *Triangulation_euclidean_traits_2<R>* is designed to deal with ordinary two dimensional points. The class *Triangulation_euclidean_traits_xy_3<R>* is a geometric traits class to build the triangulation of a terrain. Such a triangulation is a two-dimensional triangulation embedded in three dimensional space. The data points are three-dimensional points. The triangulation is build according to the projections of those points on the xy plane and then lifted up to the original three-dimensional data points. This is especially usefull to deal with GIS terrains. Instead of really projecting the three-dimensional points and maintaining a mapping between each point and its projection (which costs space and is error prone), the traits class supplies geometric predicates that ignore the z -coordinates of the points. See Section 20.5 for an example. CGAL provides also the geometric traits classes *Triangulation_euclidean_traits_yz_3<R>* and *Triangulation_euclidean_traits_zx_3<R>* to deal with projections on the xz plane and yz -plane, respectively.

20.4.2 Example of a Basic Triangulation

The following program creates a triangulation of 2D points using the default kernel *Exact_predicate_inexact_constructions_kernel* as geometric traits and the default triangulation data structure. The input points are read from a file and inserted in the triangulation. Finally points on the convex hull are written to `cout`.

```
// file: examples/Triangulation_2/triangulation_prog1.C

#include <fstream>

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Triangulation_2.h>

struct K : CGAL::Exact_predicates_inexact_constructions_kernel {};

typedef CGAL::Triangulation_2<K>          Triangulation;
typedef Triangulation::Vertex_circulator Vertex_circulator;
typedef Triangulation::Point             Point;

int main() {
    std::ifstream in("data/triangulation_prog1.cin");
    std::istream_iterator<Point> begin(in);
    std::istream_iterator<Point> end;

    Triangulation t;
    t.insert(begin, end);

    Vertex_circulator vc = t.incident_vertices(t.infinite_vertex()),
        done(vc);
```

```

if (vc != 0) {
    do { std::cout << vc->point() << std::endl;
        }while(++vc != done);
    }
    return 0;
}

```

20.5 Delaunay Triangulations

20.5.1 Description

The class *Delaunay_triangulation_2*<*Traits*,*Tds*> is designed to represent the Delaunay triangulation of a set of data points in the plane. A Delaunay triangulation fulfills the following *empty circle property* (also called *Delaunay property*): the circumscribing circle of any facet of the triangulation contains no data point in its interior. For a point set with no subset of four cocircular points the Delaunay triangulation is unique, it is dual to the Voronoi diagram of the set of points.

The class *Delaunay_triangulation_2*<*Traits*,*Tds*> derives from the class *Triangulation_2*<*Traits*,*Tds*>.

The class *Delaunay_triangulation_2*<*Traits*,*Tds*> inherits the types defined by the basic class *Triangulation_2*<*Traits*,*Tds*>. Additional types, provided by the traits class, are defined to represent the dual Voronoi diagram.

The class *Delaunay_triangulation_2*<*Traits*,*Tds*> overwrites the member functions that insert a new point in the triangulation or remove a vertex from it to maintain the Delaunay property. It also has a member function (*Vertex_handle nearest_vertex(const Point& p)*) to answer nearest neighbor queries and member functions to construct the elements (vertices and edges) of the dual Voronoi diagram.

Geometric traits

The geometric traits has to be a model of the concept *DelaunayTriangulationTraits_2* which refines the concept *TriangulationTraits_2*. In particular this concept provides the *side_of_oriented_circle* predicate which, given four points *p, q, r, s* decides the position of the point *s* with respect to the circle passing through *p, q* and *r*. The *side_of_oriented_circle* predicate actually defines the Delaunay triangulation. Changing this predicate allows to build variant of Delaunay triangulations for different metrics such that L_1 or L_∞ metric or any metric defined by a convex object. However, the user of an exotic metric must be careful that the constructed triangulation has to be a triangulation of the convex hull which means that convex hull edges have to be Delaunay edges. This is granted for any smooth convex metric (like L_2) and can be ensured for other metrics (like L_∞) by the addition to the point set of well chosen sentinel points.

The CGAL kernel classes and the class *Triangulation_euclidean_traits_2*<*R*> are models of the concept *DelaunayTriangulationTraits_2* for the euclidean metric. The traits class for terrains, *Triangulation_euclidean_traits_xy_3*<*R*>,

Triangulation_euclidean_traits_yz_3<*R*>, and

Triangulation_euclidean_traits_zx_3<*R*>

are also models of *DelaunayTriangulationTraits_2* except that they do not fulfill the requirements for the duality functions and nearest vertex queries.

Implementation

The insertion of a new point in the Delaunay triangulation is performed using first the insertion member function of the basic triangulation and second performing a sequence of flips to restore the Delaunay property. The

number of flips that have to be performed is $O(d)$ if the new vertex has degree d in the updated Delaunay triangulation. For points distributed uniformly at random, each insertion takes time $O(1)$ on average, once the point has been located in the triangulation.

Removal calls the removal in the triangulation and then retriangulates the hole created in such a way that the Delaunay criterion is satisfied. Removal of a vertex of degree d takes time $O(d^2)$. The degree d is $O(1)$ for a random vertex in the triangulation.

After having performed a point location, the nearest neighbor of a point is found in time $O(n)$ in the worst case, but in time $O(1)$ for vertices distributed uniformly at random and any query point.

20.5.2 Example : a Delaunay Terrain

The following code creates a Delaunay triangulation with the usual Euclidean metric for the vertical projection of a terrain model. The points have elevation, that is they are 3D points, but the predicates used to build the Delaunay triangulation are computed using only the x and y coordinates of these points.

```
// file: examples/Triangulation_2/terrain.C

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Triangulation_euclidean_traits_xy_3.h>
#include <CGAL/Delaunay_triangulation_2.h>

#include <fstream>

struct K : CGAL::Exact_predicates_inexact_constructions_kernel {};

typedef CGAL::Triangulation_euclidean_traits_xy_3<K>   Gt;
typedef CGAL::Delaunay_triangulation_2<Gt>   Delaunay;

typedef K::Point_3   Point;

int main()
{
    std::ifstream in("data/terrain.cin");
    std::istream_iterator<Point> begin(in);
    std::istream_iterator<Point> end;

    Delaunay dt;
    dt.insert(begin, end);
    std::cout << dt.number_of_vertices() << std::endl;
    return 0;
}
```

20.5.3 Example : Voronoi Diagram

The following code computes the edges of Voronoi diagram of a set of data points and counts the number of finite edges and the number of rays of this diagram

```
// file: examples/Triangulation_2/Voronoi.C
```



```

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>

#include <fstream>

struct K : CGAL::Exact_predicates_inexact_constructions_kernel {};

typedef CGAL::Delaunay_triangulation_2<K> Triangulation;
typedef Triangulation::Edge_iterator Edge_iterator;
typedef Triangulation::Point Point;

int main( )
{
    std::ifstream in("data/voronoi.cin");
    std::istream_iterator<Point> begin(in);
    std::istream_iterator<Point> end;
    Triangulation T;
    T.insert(begin, end);

    int ns = 0;
    int nr = 0;
    Edge_iterator eit =T.edges_begin();
    for ( ; eit !=T.edges_end(); ++eit) {
        CGAL::Object o = T.dual(eit);
        K::Segment_2 s;
        K::Ray_2 r;
        if (CGAL::assign(s,o)) {++ns;}
        if (CGAL::assign(r,o)) {++nr;}
    }
    std::cout << "The voronoi diagram as " << ns << "finite edges "
        << " and " << nr << " rays" << std::endl;
    return 0;
}

```

20.6 Regular Triangulations

20.6.1 Description

Let $PW = \{(p_i, w_i), i = 1, \dots, n\}$ be a set of weighted points where each p_i is a point and each w_i is a scalar called the weight of point p_i . Alternatively, each weighted point (p_i, w_i) can be regarded as a sphere (or a circle, depending on the dimensionality of p_i) with center p_i and radius $r_i = \sqrt{w_i}$.

The power diagram of the set PW is a space partition in which each cell corresponds to a sphere (p_i, w_i) of PW and is the locus of points p whose power with respect to (p_i, w_i) is less than its power with respect to any other sphere in PW . In the two-dimensional space, the dual of this diagram is a triangulation whose domain covers the convex hull of the set $P = \{p_i, i = 1, \dots, n\}$ of center points and whose vertices form a subset of P . Such a triangulation is called a regular triangulation. Three points p_i, p_j and p_k of P form a triangle in the regular triangulation of PW iff there is a point p of the plane with equal powers with respect to (p_i, w_i) , (p_j, w_j) and (p_k, w_k) and such that this power is less than the power of p with respect to any other sphere in PW .

Let us defined the power product of two weighted points (p_i, w_i) and (p_j, w_j) as:

$$\Pi(p_i, w_i, p_j, w_j) = p_i p_j^2 - w_i - w_j.$$

$\Pi(p_i, w_i, p_j, 0)$ is simply the power of point p_j with respect to the sphere (p_i, w_i) , and two weighted points are said to be orthogonal if their power product is null. The power circle of three weighted points (p_i, w_i) , (p_j, w_j) and (p_k, w_k) is defined as the unique circle (π, ω) orthogonal to (p_i, w_i) , (p_j, w_j) and (p_k, w_k) .

The regular triangulation of the sets PW satisfies the following *regular property* (which just reduces to the Delaunay property when all the weights are null): a triangle $p_i p_j p_k$ is a face of the regular triangulation of PW iff the power product of any weighted point (p_l, w_l) of PW with the power circle of (p_i, w_i) , (p_j, w_j) and (p_k, w_k) is positive or null. We call power test of (p_i, w_i) , (p_j, w_j) , (p_k, w_k) , and (p_l, w_l) , the predicates which amount to compute the sign of the power product of (p_l, w_l) with respect to the power circle of (p_i, w_i) , (p_j, w_j) and (p_k, w_k) . This predicate amounts to computing the sign of the following determinant

$$\begin{vmatrix} 1 & x_i & y_i & x_i^2 + y_i^2 - w_i \\ 1 & x_j & y_j & x_j^2 + y_j^2 - w_j \\ 1 & x_k & y_k & x_k^2 + y_k^2 - w_k \\ 1 & x_l & y_l & x_l^2 + y_l^2 - w_l \end{vmatrix}$$

A pair of neighboring faces $p_i p_j p_k$ and $p_i p_j p_l$ is said to be locally regular (with respect to the weights in PW) if the power test of (p_i, w_i) , (p_j, w_j) , (p_k, w_k) , and (p_l, w_l) is positive. A classical result of computational geometry establishes that a triangulation of the convex hull of P such that any pair of neighboring faces is regular with respect to PW , is a regular triangulation of PW .

Alternatively, the regular triangulation of the weighted points set PW can be obtained as the projection on the two dimensional plane of the convex hull of the set of three dimensional points $P' = \{(p_i, p_i^2 - w_i), i = 1, \dots, n\}$.

The class *Regular_triangulation_2*<Traits, Tds> is designed to maintain the regular triangulation of a set of 2d weighted points. It derives from the class *Triangulation_2*<Traits, Tds>. The functions *insert* and *remove* are overwritten to handle weighted points and maintain the regular property. The vertices of the regular triangulation of a set of weighted points PW correspond only to a subset of PW . Some of the input weighed points have no cell in the dual power diagrams and therefore do not correspond to a vertex of the regular triangulation. Such a point is called a hidden point. Because hidden points can reappear later on as vertices when some other point is removed, they have to be stored somewhere. The regular triangulation store those points in special vertices, called hidden vertices. A hidden point can reappear as vertex of the triangulation only when the two dimensional face that hides it is removed from the triangulation. To deal with this feature, each face of a regular triangulation stores a list of hidden vertices. The points in those vertices are reinserted in the triangulation when the face is removed.

Regular triangulation have member functions to construct the vertices and edges of the dual power diagrams.

The geometric traits

The geometric traits of a regular triangulation must provide a weighted point type and a power test on these weighted points. The concept *RegularTriangulationTraits_2*, is a refinement of the concept *TriangulationTraits_2*. CGAL provides the class *Regular_triangulation_euclidean_traits_2*<Rep, Weight> which is a model for the traits concept *RegularTriangulationTraits_2*. The class *Regular_triangulation_euclidean_traits_2*<Rep, Weight> derives from the class *Triangulation_euclidean_traits_2*<Rep> and uses a *Weighted_point* type derived from the type *Point_2* of *Triangulation_euclidean_traits_2*<Rep>.

Note that, since the type *Weighted_point* is not defined in CGAL kernels, plugging a filtered kernel such as *Exact_predicates_exact_constructions_kernel* in *Regular_triangulation_euclidean_traits_2*<K, Weight> will

in fact not provide exact predicates on weighted points. To solve this, there is also another model of the traits concept, *Regular_triangulation_filtered_traits_2<FK>*, which is providing filtered predicates (exact and efficient). The argument *FK* must be a model of the *Kernel* concept, and it is also restricted to be a instance of the *Filtered_kernel* template.

The Vertex type and Face Type of a Regular Triangulation

The base vertex type of a regular triangulation includes a boolean to mark the hidden state of the vertex. Therefore CGAL defines the concept *RegularTriangulationVertexBase_2* which refine the concept *TriangulationVertexBase_2* and provides a default model for this concept.

The base face type of a regular triangulation is required to provide a list of hidden vertices, designed to store the points hidden by the face. It has to be a model of the concept *RegularTriangulationFaceBase_2*. CGAL provides the templated class *Regular_triangulation_face_base_2<Traits>* as a default base class for faces of regular triangulations.

20.6.2 Example : a Regular Triangulation

The following code creates a regular triangulation of a set of weighted points and output the number of vertices and the number of hidden vertices.

```
// file: examples/Triangulation_2/regular.C

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Regular_triangulation_euclidean_traits_2.h>
#include <CGAL/Regular_triangulation_filtered_traits_2.h>
#include <CGAL/Regular_triangulation_2.h>

#include <fstream>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef CGAL::Regular_triangulation_filtered_traits_2<K> Traits;
typedef CGAL::Regular_triangulation_2<Traits> Regular_triangulation;

int main()
{
    Regular_triangulation rt;
    std::ifstream in("data/regular.cin");

    Regular_triangulation::Weighted_point wp;
    int count = 0;
    while(in >> wp){
        count++;
        rt.insert(wp);
    }
    rt.is_valid();
    std::cout << "number of inserted points : " << count << std::endl;
    std::cout << "number of vertices : " ;
    std::cout << rt.number_of_vertices() << std::endl;
    std::cout << "number of hidden vertices : " ;
    std::cout << rt.number_of_hidden_vertices() << std::endl;
```

```

    return 0;
}

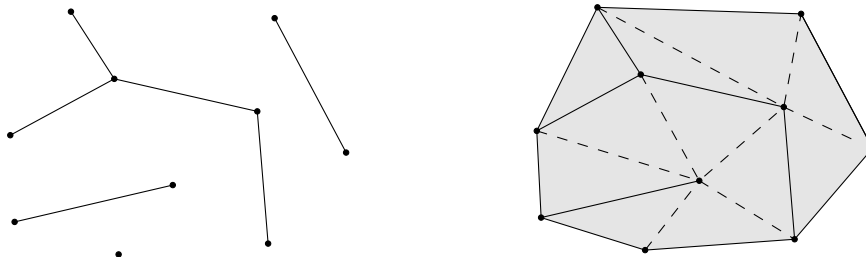
```

20.7 Constrained Triangulations

A constrained triangulation is a triangulation of a set of points that has to include among its edges a given set of segments joining the points. The corresponding edges are called *constrained edges*.

The endpoints of constrained edges are of course vertices of the triangulation. However the triangulation may include other vertices as well. There are three versions of constrained triangulations.

- In the basic version, the constrained triangulation does not handle intersecting constraints, and the set of input constraints is required to be a set of segments that do not intersect except possibly at their endpoints. Any number of constrained edges are allowed to share the same endpoint. Vertical constrained edges or constrained edges with null length are allowed.
- The two other versions support intersecting input constraints. In those versions, input constraints are allowed to be intersecting, overlapping or partially overlapping segments. The triangulation introduces additional vertices at each point that is the proper intersection points of two constraints. A single constraint intersecting other constraints will then appear as several edges in the triangulation. The two versions dealing with intersecting constraints differ in the way intersecting constraints are dealt with.
 - One of them is designed to be robust when predicates are evaluated exactly but constructions (i. e. intersection computations) are approximative.
 - The other one is designed to be used with exact arithmetic (meaning exact evaluation of predicates and exact computation of intersections.) This last version finds its full efficiency when used in conjunction with a constraint hierarchy data structure (which allows one to avoid the cascading of intersection computations) as provided in the class *Constrained_triangulation_plus_2*. See section 20.9.



A constrained triangulation is represented in the CGAL library as an object of the class *Constrained_triangulation_2*<*Traits*,*Tds*,*Itag*>. The third parameter *Itag* is the intersection tag which serves to choose how intersecting constraints are dealt with. This parameter has to be instantiated by one of the following classes : *CGAL::No_intersection_tag* when input constraints do not intersect
CGAL::Exact_predicates_tag if the geometric traits provides exact predicates but approximative constructions
CGAL::Exact_intersections_tag when an exact predicates and exact constructions are provided.

The class *Constrained_triangulation_2*<*Traits*,*Tds*, *Itag*> inherits from *Triangulation_2*<*Traits*,*Tds*>. It defines an additional type *Constraint* to represent the constraints. A constraint is represented as a pair of points.

A constrained triangulation can be created from a list of constrained edges. The class *Constrained_triangulation_2*<*Traits*,*Tds*,*Itag*> overrides the insertion and removal of a point to take care of the information

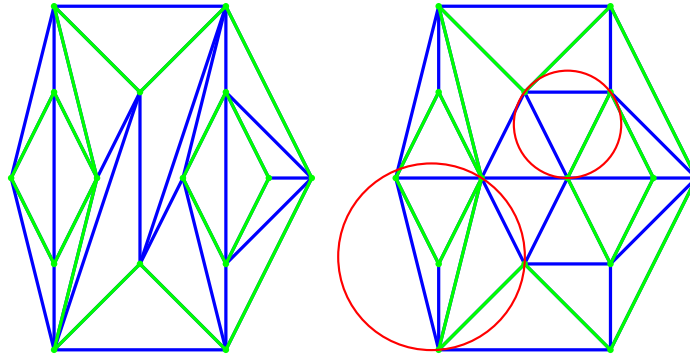


Figure 20.7: Constrained and Constrained Delaunay triangulation : the constraining edges are the green edges, a constrained triangulation is shown on the left, the constrained Delaunay triangulation with two examples of circumcircles is shown on the right.

about constrained edges. The class also allows inline insertion of a new constraint, given by its two endpoints or the removal of a constraint.

The Geometric Traits

The geometric traits of a constraint triangulation has to be a model of the concept *TriangulationTraits_2*. When intersections of input constraints are supported, the geometric traits class has to be a model of the concept *ConstrainedTriangulationTraits_2* which refines the concept *TriangulationTraits_2* providing additional function object types to compute the intersection of two segments.

The Base Face of a Constrained Triangulation

The information about constrained edges is stored in the faces of the triangulation. The base face of a Constrained Triangulation has to be a model for the concept *ConstrainedTriangulationFaceBase_2* which refines the concept *TriangulationFaceBase_2*. The concept *ConstrainedTriangulationFaceBase_2* requires member functions the get and set the constrained status of the edges.

CGAL provides a default base face class for constrained triangulations. This class, named *ConstrainedTriangulationFaceBase_2<Traits>*, derives from the class *TriangulationFaceBase_2<Traits>* and adds three booleans to store the status of its edges.

20.8 Constrained Delaunay Triangulations

A constrained Delaunay triangulation is a triangulation with constrained edges which tries to be as much Delaunay as possible. As constrained edges are not necessarily Delaunay edges, the triangles of a constrained Delaunay triangulation do not necessarily fulfill the empty circle property but they fulfill a weaker *constrained empty circle property*. To state this property, it is convenient to think of constrained edges as blocking the view. Then, a triangulation is constrained Delaunay iff the circumscribing circle of any facet encloses no vertex visible from the interior of the facet. As in the case of constrained triangulations, three different versions of Delaunay constrained triangulations are provided. The first version handle set of constraints which do not intersect except

possibly at the endpoints. The two other versions handle intersecting input constraints. One of them is designed to be designed to be robust when used in conjunction with a geometric traits providing exact predicates and approximative constructions (such as a *CGAL::Filtered_Kernel* or any kernel providing filtered exact predicates). The third version is designed to be used with an exact arithmetic number type.

The CGAL class *Constrained_Delaunay_triangulation_2*<*Traits,Tds,Itag*> is designed to represent constrained Delaunay triangulations.

As in the case of constraints triangulation, the third parameter *Itag* is the intersection tag and serves to choose how intersecting constraints are dealt with. It can be instantiated with one of the following class : *CGAL::No_intersection_tag*, *CGAL::Exact_predicates_tag*, *CGAL::Exact_intersections_tag* (see Section 20.7).

A constrained Delaunay triangulation is not a Delaunay triangulation but it is a constrained triangulation. Therefore the class *Constrained_Delaunay_triangulation_2*<*Traits,Tds,Itag*> derives from the class *Constrained_triangulation_2*<*Traits,Tds,Itag*>.

The constrained Delaunay triangulation has member functions to override the insertion and removal of a point or of a constraint. Each of those member function takes care to restore the constrained empty circle property.

The Geometric Traits

The geometric traits of a constrained Delaunay triangulation is required to provide the *side_of_oriented_circle* predicate as the geometric traits of a Delaunay triangulation and has to a model of the concept *DelaunayTriangulationTraits_2*. When intersecting input constraints are supported the geometric traits is further required to provide function objects to compute constraints intersections. Then the geometric traits has to be at the same time a model of the concept *ConstrainedTriangulationTraits_2*.

The face base class

Information about the status (constrained or not) of the edges of the triangulation has to be stored in the face class and the base face class of a constrained Delaunay triangulation has to be a model of *ConstrainedTriangulationFaceBase_2*.

20.8.1 Example : a constrained Delaunay triangulation

The following code inserts a set of intersecting constraint segments into a triangulation and counts the number of constrained edges of the resulting triangulation.

```
// file : examples/Triangulation_2/constrained.C

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Constrained_Delaunay_triangulation_2.h>

struct K : CGAL::Exact_predicates_inexact_constructions_kernel {};

typedef CGAL::Triangulation_vertex_base_2<K> Vb;
typedef CGAL::Constrained_triangulation_face_base_2<K> Fb;
typedef CGAL::Triangulation_data_structure_2<Vb,Fb> TDS;
typedef CGAL::Exact_predicates_tag Itag;
```

```

typedef CGAL::Constrained_Delaunay_triangulation_2<K, TDS, Itag> CDT;
typedef CDT::Point Point;

int
main( )
{
    CDT cdt;
    std::cout << "Inserting a grid of 5x5 constraints " << std::endl;
    for (int i = 1; i < 6; ++i)
        cdt.insert_constraint( Point(0,i), Point(6,i));
    for (int j = 1; j < 6; ++j)
        cdt.insert_constraint( Point(j,0), Point(j,6));

    assert(cdt.is_valid());
    int count = 0;
    for (CDT::Finite_edges_iterator eit = cdt.finite_edges_begin();
        eit != cdt.finite_edges_end();
        ++eit)
        if (cdt.is_constrained(*eit)) ++count;
    std::cout << "The number of resulting constrained edges is ";
    std::cout << count << std::endl;
    return 0;
}

```

20.9 Constrained Triangulations Plus

The class *Constrained_triangulation_plus_2*<Tr> provides a constrained triangulation with an additional data structure called the constraint hierarchy that keeps track of the input constraints and of their refinement in the triangulation. The class *Constrained_triangulation_plus_2*<Tr> inherits from its template parameter Tr, which has to be instantiated by a constrained or constrained Delaunay triangulation. According to its intersection tag, the base class will support intersecting input constraints or not. When intersections of input constraints are supported, the base class constructs a triangulation of the arrangement of the constraints, introducing new vertices at each proper intersection points and refining the input constraints into subconstraints which appear as edges (more precisely as constrained edges) of the triangulation. The data structure maintains for each input constraint the sequence of intersection vertices added on this constraint. The constraint hierarchy also allows the user to retrieve the set of constrained edges of the triangulation, and for each constrained edge, the set of input constraints that overlap it.

The class *Constrained_triangulation_plus_2*<Tr> is especially useful when the base constrained triangulation class handles intersections of constraints and uses an exact number type, i.e. when its intersection tag is *CGAL::Exact_intersections_tag*. Indeed in this case, the *Constrained_triangulation_plus_2*<Tr> is specially designed to avoid cascading in the computations of intersection points.

20.9.1 Example : Building a triangulated arrangement of segments

The following code inserts a set of intersecting constraint segments into a triangulation and counts the number of constrained edges of the resulting triangulation.

```
// file: examples/Triangulation_2/constrained_plus.C
```

```

#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/intersections.h>
#include <CGAL/Constrained_Delaunay_triangulation_2.h>
#include <CGAL/Constrained_triangulation_plus_2.h>

struct K : CGAL::Exact_predicates_exact_constructions_kernel {};

typedef CGAL::Triangulation_vertex_base_2<K>          Vb;
typedef CGAL::Constrained_triangulation_face_base_2<K> Fb;
typedef CGAL::Triangulation_data_structure_2<Vb,Fb>   TDS;
typedef CGAL::Exact_intersections_tag                 Itag;
typedef CGAL::Constrained_Delaunay_triangulation_2<K,TDS,Itag> CDT;
typedef CGAL::Constrained_triangulation_plus_2<CDT>    CDTplus;
typedef CDTplus::Point                                 Point;

int
main( )
{
    CDTplus cdt;
    std::cout << "Inserting a grid 5 x 5 of constraints " << std::endl;
    for (int i = 1; i < 6; ++i)
        cdt.insert_constraint( Point(0,i), Point(6,i));
    for (int j = 1; j < 6; ++j)
        cdt.insert_constraint( Point(j,0), Point(j,6));

    assert(cdt.is_valid());
    int count = 0;
    for (CDTplus::Subconstraint_iterator scit = cdt.subconstraints_begin();
         scit != cdt.subconstraints_end();
         ++scit) ++count;
    std::cout << "The number of resulting constrained edges is  "
              << count << std::endl;
    return 0;
}

```

20.10 The Triangulation Hierarchy

The class *Triangulation_hierarchy_2<Tr>* implements a triangulation augmented with a data structure to answer efficiently point location queries. The data structure is a hierarchy of triangulations. The triangulation at the lowest level is the original triangulation where operations and point location are to be performed. Then at each succeeding level, the data structure stores a triangulation of a small random sample of the vertices of the triangulation at the preceeding level. Point location is done through a top down nearest neighbor query. The nearest neighbor query is first performed naively in the top level triangulation. Then, at each following level, the nearest neighbor at that level is found through a linear walk performed from the nearest neighbor found at the preceeding level. Because the number of vertices in each triangulation is only a small fraction of the number of vertices of the preceeding triangulation, the data structure remains small and achieves fast point location queries on real data. As proved in [Dev98], this structure has an optimal behaviour when it is built for Delaunay triangulations. However it can be used as well for other triangulations and the class *Triangulation_hierarchy_2<Tr>* is templated by a parameter which is to be instantiated by one of the CGAL triangulation classes.

The class *Triangulation_hierarchy_2<Tr>* inherits from the triangulation type passed as template parameter *Tr*.

The insert and remove member functions are overwritten to update the data structure at each operation. The locate queries are also overwritten to take advantage of the data structure for a fast processing.

The Vertex of a Triangulation Hierarchy

The base vertex class of a triangulation hierarchy has to be a model of the concept *TriangulationHierarchyVertexBase_2* which extends the concept *TriangulationVertexBase_2*. This extension adds access and setting member functions for two pointers to the corresponding vertices in the triangulations of the next and preceeding levels.

CGAL provides the class *Triangulation_hierarchy_vertex_base_2<Vb>* which is a model for the concept *TriangulationHierarchyVertexBase_2*. This class is templated by a parameter *Vb* which is to be instantiated by a model of the concept *TriangulationVertexBase_2*. The class *Triangulation_hierarchy_vertex_base_2<Vb>* inherits from its template parameter *Vb*. This design allows to use for *Vb* either the default vertex class or a user customized vertex class with additional functionalities.

20.10.1 Examples of the Use of a Triangulation Hierarchy

The following program is example of the standard use of a triangulation hierarchy to enhance the efficiency of a Delaunay triangulation. The program output the number of vertices at the different levels of the hierarchy.

```
// file: examples/Triangulation_2/hierarchy.C

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/Triangulation_hierarchy_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/algorithm.h>
#include <cassert>

struct K : CGAL::Exact_predicates_inexact_constructions_kernel {};

typedef CGAL::Triangulation_vertex_base_2<K> Vbb;
typedef CGAL::Triangulation_hierarchy_vertex_base_2<Vbb> Vb;
typedef CGAL::Triangulation_face_base_2<K> Fb;
typedef CGAL::Triangulation_data_structure_2<Vb,Fb> Tds;
typedef CGAL::Delaunay_triangulation_2<K,Tds> Dt;
typedef CGAL::Triangulation_hierarchy_2<Dt> Triangulation;
typedef Triangulation::Point Point;
typedef CGAL::Creator_uniform_2<double,Point> Creator;

int main( )
{
    std::cout << "insertion of 1000 random points" << std::endl;
    Triangulation t;
    CGAL::Random_points_in_square_2<Point,Creator> g(1.);
    CGAL::copy_n( g, 1000, std::back_inserter(t));

    //verbose mode of is_valid ; shows the number of vertices at each level
    std::cout << "The number of vertices at successive levels" << std::endl;
```

```

    assert(t.is_valid(true));

    return 0;
}

```

The following program shows how to use a triangulation hierarchy in conjunction with a constrained triangulation plus.

```

// file: examples/Triangulation_2/constrained_hierarchy_plus.C

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Constrained_Delaunay_triangulation_2.h>
#include <CGAL/Triangulation_hierarchy_2.h>
#include <CGAL/Constrained_triangulation_plus_2.h>

struct K : CGAL::Exact_predicates_inexact_constructions_kernel {};

typedef CGAL::Triangulation_vertex_base_2<K>          Vbb;
typedef CGAL::Triangulation_hierarchy_vertex_base_2<Vbb> Vb;
typedef CGAL::Constrained_triangulation_face_base_2<K> Fb;
typedef CGAL::Triangulation_data_structure_2<Vb,Fb>    TDS;
typedef CGAL::Exact_predicates_tag                      Itag;
typedef CGAL::Constrained_Delaunay_triangulation_2<K,TDS,Itag> CDT;
typedef CGAL::Triangulation_hierarchy_2<CDT>           CDTH;
typedef CGAL::Constrained_triangulation_plus_2<CDTH>    Triangulation;

typedef Triangulation::Point                            Point;

int
main( )
{
    Triangulation cdt;
    std::cout << "Inserting a grid 5 x 5 of constraints " << std::endl;
    for (int i = 1; i < 6; ++i)
        cdt.insert_constraint( Point(0,i), Point(6,i));
    for (int j = 1; j < 6; ++j)
        cdt.insert_constraint( Point(j,0), Point(j,6));

    int count = 0;
    for (Triangulation::Subconstraint_iterator scit = cdt.subconstraints_begin();
         scit != cdt.subconstraints_end();
         ++scit) ++count;
    std::cout << "The number of resulting constrained edges is ";
    std::cout << count << std::endl;

    //verbose mode of is_valid ; shows the number of vertices at each level
    std::cout << "The number of vertices at successive levels" << std::endl;
    assert(cdt.is_valid(true));

    return 0;
}

```

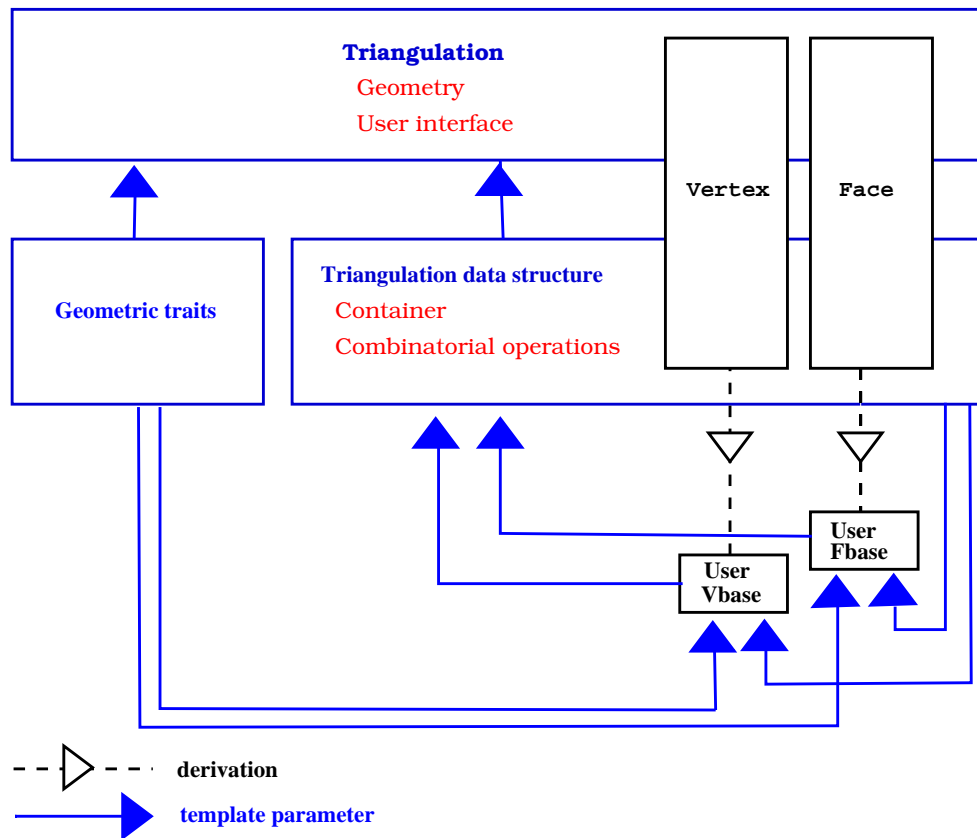


Figure 20.8: The cyclic dependency in triangulations software design.

20.11 Flexibility: Using Customized Vertices and Faces

To be able to adapt to various needs, a highly flexible design has been selected for 2D triangulations. We have already seen that the triangulation classes have two parameters : a geometric traits class and a triangulation data structure class which the user can instantiate with his own customized classes.

The most useful flexibility however comes from the fact that the triangulation data structure itself has two template parameters to be instantiated by classes for the vertices and faces of the triangulation. Using his own customized classes to instantiate these parameters, the user can easily build up a triangulation with additional information or functionality in the vertices and faces.

A cyclic dependency

To insure flexibility, the triangulation data structure is templated by the vertex and face base classes. Also since incidence and adjacency relations are stored in vertices and faces, the base classes have to know the types of handles on vertices and faces provided by the triangulation data structure. Thus the vertex and base classes have to be themselves parameterized by the triangulation data structure, and there is a cyclic dependency on template parameter.

Previously, this cyclic dependency was avoided by using only *void** pointers in the interface of base classes. These *void** were converted to appropriate types at the triangulation data structure levels. This solution had

some drawbacks : mainly the user could not add in the vertices or faces of the triangulation a functionality related to types defined by the triangulation data structure, for instance a handle to a vertex, and he was lead to use himself *void** pointers). The new solution to resolve the template dependency is based on a rebind mechanism similar to the mechanism used in the standard allocator class `std::allocator`. The rebind mechanism is described in Section 21.3 of Chapter 21. For now, we will just notice that the design requires the existence in the vertex and face base classes of a nested template class *Rebind_TDS* defining a type *Other* used by the rebinding mechanism.

The two following examples show how the user can put in use the flexibility offered by the base classes parameters.

Adding colors

The first example corresponds to a case where the user wishes to add in the vertices or faces of the triangulation an additional information that does not depend on types provided by the triangulation data structure. In that case, predefined classes *Triangulation_vertex_base_with_info_2*<*Info*,*Traits*,*Vb*> or *Triangulation_face_base_with_info_2*<*Info*,*Traits*,*Vb*> can be used. Those classes have a template parameter *Info* devoted to handle additional information. The following examples shows how to add a *CGAL::Color* in the triangulation faces.

```
// file : examples/Triangulation_2/colored_face.C

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/IO/Color.h>
#include <CGAL/Triangulation_2.h>
#include <CGAL/Triangulation_face_base_with_info_2.h>

struct K : CGAL::Exact_predicates_inexact_constructions_kernel {};

typedef CGAL::Triangulation_vertex_base_2<K> Vb;
typedef CGAL::Triangulation_face_base_with_info_2<CGAL::Color,K> Fb;
typedef CGAL::Triangulation_data_structure_2<Vb,Fb> Tds;
typedef CGAL::Triangulation_2<K,Tds> Triangulation;

typedef Triangulation::Face_handle Face_handle;
typedef Triangulation::Finite_faces_iterator Finite_faces_iterator;
typedef Triangulation::Point Point;

int main() {
    Triangulation t;
    t.insert(Point(0,1));
    t.insert(Point(0,0));
    t.insert(Point(2,0));
    t.insert(Point(2,2));

    Finite_faces_iterator fc = t.finite_faces_begin();
    for( ; fc != t.finite_faces_end(); ++fc) fc->info() = CGAL::BLUE;

    Point p(0.5,0.5);
    Face_handle fh = t.locate(p);
    fh->info() = CGAL::RED;

    return 0;
}
```

```
}
```

Adding handles

The second example shows how the user can still derive and plug in his own vertex or face class when he would like to have additional functionalities depending on types provided by the triangulation data structure.

```
// file : examples/Triangulation_2/adding_handles.C

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Triangulation_2.h>
#include <cassert>

/* A facet with an additionnal handle */
template < class Gt, class Vb = CGAL::Triangulation_vertex_base_2<Gt> >
class My_vertex_base
: public Vb
{
    typedef Vb                                Base;
public:
    typedef typename Vb::Vertex_handle        Vertex_handle;
    typedef typename Vb::Face_handle          Face_handle;
    typedef typename Vb::Point                Point;

    template < typename TDS2 >
    struct Rebind_TDS {
        typedef typename Vb::template Rebind_TDS<TDS2>::Other    Vb2;
        typedef My_vertex_base<Gt,Vb2>                            Other;
    };

private:
    Vertex_handle va_;

public:
    My_vertex_base() : Base() {}
    My_vertex_base(const Point & p) : Base(p) {}
    My_vertex_base(const Point & p, Face_handle f) : Base(f,p) {}
    My_vertex_base(Face_handle f) : Base(f) {}

    void set_associated_vertex(Vertex_handle va) { va_ = va;}
    Vertex_handle get_associated_vertex() {return va_ ; }
};

struct K : CGAL::Exact_predicates_inexact_constructions_kernel {};

typedef My_vertex_base<K> Vb;
typedef CGAL::Triangulation_data_structure_2<Vb> Tds;
typedef CGAL::Triangulation_2<K,Tds> Triangulation;

typedef Triangulation::Vertex_handle Vertex_handle;
typedef Triangulation::Finite_faces_iterator Finite_faces_iterator;
typedef Triangulation::Point Point;
```

```

int main() {
    Triangulation t;
    Vertex_handle v0 = t.insert(Point(0,1));
    Vertex_handle v1 = t.insert(Point(0,0));
    Vertex_handle v2 = t.insert(Point(2,0));
    Vertex_handle v3 = t.insert(Point(2,2));

    // associate vertices as you like
    v0->set_associated_vertex(v1);
    v1->set_associated_vertex(v2);
    v2->set_associated_vertex(v3);
    v3->set_associated_vertex(v0);
    assert( v0->get_associated_vertex() == v1);

    return 0;
}

```

20.12 Design and Implementation History

The code of this package is the result of a long development process. Here follows a tentative list of people who added their stone to this package : Jean-Daniel Boissonnat, Hervé Brönnimann, Olivier Devillers, Andreas Fabri, Frédéric Fichel, Julia Flötotto, Monique Teillaud and Mariette Yvinec.

2D Triangulations

Reference Manual

Mariette Yvinec

A triangulation is a 2-dimensional simplicial complex which is pure connected and without singularities. Thus a triangulation can be viewed as a collection of triangular faces, such that two faces either have an empty intersection or share an edge or a vertex.

The basic elements of the representation are vertices and faces. Each triangular face gives access to its three incident vertices and to its three adjacent faces. Each vertex gives access to one of its incident faces and through that face to the circular list of its incident faces. The edges are not explicitly represented, they are only represented through the adjacencies relations of two faces.

The triangulation classes of CGAL depend on two template parameters. The first template parameter stands for a geometric traits class which is assumed to provide the geometric objects (points, segments and triangles) forming the triangulation and the geometric predicates on those objects. The second template parameter stands for a model of triangulation data structure acting as a container for faces and vertices while taking care of the combinatorial aspects of the triangulation. The concepts and models relative to the triangulation data structure are described in Chapter 21.

20.13 Classified Reference Pages

Concepts

TriangulationTraits_2	page 1424
DelaunayTriangulationTraits_2	page 1399
RegularTriangulationTraits_2	page 1408
ConstrainedTriangulationTraits_2	page 1381
ConstrainedDelaunayTriangulationTraits_2	page 1378
TriangulationFaceBase_2	page 1422
TriangulationVertexBase_2	page 1426
ConstrainedTriangulationFaceBase_2	page 1379
RegularTriangulationFaceBase_2	page 1407
RegularTriangulationVertexBase_2	page 1410
TriangulationHierarchyVertexBase_2	page 1423

Classes

<i>CGAL::Triangulation_2<Traits,Tds></i>	page 1428
<i>CGAL::Delaunay_triangulation_2<Traits,Tds></i>	page 1401
<i>CGAL::Regular_triangulation_2<Traits,Tds></i>	page 1411
<i>CGAL::Constrained_triangulation_2<Traits,Tds,Itag></i>	page 1388
<i>CGAL::Constrained_Delaunay_triangulation_2<Traits,Tds,Itag></i>	page 1383
<i>CGAL::Constrained_triangulation_plus_2<Tr></i>	page 1394
<i>CGAL::Triangulation_hierarchy_2<Tr></i>	page 1450
<i>CGAL::Triangulation_euclidean_traits_2<K></i>	page 1444
<i>CGAL::Triangulation_euclidean_traits_xy_3<K></i>	page 1445
<i>CGAL::Regular_triangulation_euclidean_traits_2<K,Weight></i>	page 1418
<i>CGAL::Regular_triangulation_filtered_traits_2<FK></i>	page 1419
<i>CGAL::Triangulation_face_base_2<Traits,Fb></i>	page 1448
<i>CGAL::Triangulation_vertex_base_2<Traits,Vb></i>	page 1452
<i>CGAL::Regular_triangulation_face_base_2<Traits,Fb></i>	page 1420
<i>CGAL::Regular_triangulation_vertex_base_2<Traits,Vb></i>	page 1421
<i>CGAL::Constrained_triangulation_face_base_2<Traits,Fb></i>	page 1393
<i>CGAL::Triangulation_vertex_base_with_info_2<Info,Traits,Vb></i>	page 1453
<i>CGAL::Triangulation_face_base_with_info_2<Info,Traits,Fb></i>	page 1449
<i>CGAL::Triangulation_hierarchy_vertex_base_2<Vb></i>	page 1451
<i>CGAL::Weighted_point<Pt,Wt></i>	page 1454
<i>CGAL::Triangulation_cw_ccw_2</i>	page 1442

Enum

<i>CGAL::Triangulation_2<Traits,Tds>::Locate_type</i>	page 1406
-------------------------------------------------------------------	-----------

20.14 Alphabetical List of Reference Pages

<i>ConstrainedDelaunayTriangulationTraits_2</i>	page 1378
<i>ConstrainedTriangulationFaceBase_2</i>	page 1379
<i>ConstrainedTriangulationTraits_2</i>	page 1381
<i>Constrained_Delaunay_triangulation_2<Traits,Tds,Itag></i>	page 1383
<i>Constrained_triangulation_2<Traits,Tds,Itag></i>	page 1388
<i>Constrained_triangulation_face_base_2<Traits,Fb></i>	page 1393
<i>Constrained_triangulation_plus_2<Tr></i>	page 1394
<i>DelaunayTriangulationTraits_2</i>	page 1399
<i>Delaunay_triangulation_2<Traits,Tds></i>	page 1401
<i>Locate_type</i>	page 1406
<i>RegularTriangulationFaceBase_2</i>	page 1407
<i>RegularTriangulationTraits_2</i>	page 1408
<i>RegularTriangulationVertexBase_2</i>	page 1410
<i>Regular_triangulation_2<Traits,Tds></i>	page 1411

<i>Regular_triangulation_euclidean_traits_2<K,Weight></i>	page 1418
<i>Regular_triangulation_face_base_2<Traits,Fb></i>	page 1420
<i>Regular_triangulation_filtered_traits_2<FK></i>	page 1419
<i>Regular_triangulation_vertex_base_2<Traits,Vb></i>	page 1421
<i>TriangulationFaceBase_2</i>	page 1422
<i>TriangulationHierarchyVertexBase_2</i>	page 1423
<i>TriangulationTraits_2</i>	page 1424
<i>TriangulationVertexBase_2</i>	page 1426
<i>Triangulation_2<Traits,Tds></i>	page 1428
<i>Triangulation_cw_ccw_2</i>	page 1442
<i>Triangulation_euclidean_traits_2<K></i>	page 1444
<i>Triangulation_euclidean_traits_xy_3<K></i>	page 1445
<i>Triangulation_face_base_2<Traits,Fb></i>	page 1448
<i>Triangulation_face_base_with_info_2<Info,Traits,Fb></i>	page 1449
<i>Triangulation_hierarchy_2<Tr></i>	page 1450
<i>Triangulation_hierarchy_vertex_base_2<Vb></i>	page 1451
<i>Triangulation_vertex_base_2<Traits,Vb></i>	page 1452
<i>Triangulation_vertex_base_with_info_2<Info,Traits,Vb></i>	page 1453
<i>Weighted_point<Pt,Wt></i>	page 1454

ConstrainedDelaunayTriangulationTraits_2

Definition

The concept `ConstrainedDelaunayTriangulationTraits_2` defines the requirements for the geometric traits class of a constrained Delaunay triangulation that supports intersections of input constraints. This is the case when the template parameter *Itag* of (*Constrained_Delaunay_triangulation_2*<*Traits*,*Tds*,*Itag*>) is instantiated by one of the tag classes *Exact_intersections_tag* or *Exact_predicates_tag*. The concept `ConstrainedDelaunayTriangulationTraits_2` refines both the concept *DelaunayTriangulationTraits_2* and the concept *ConstrainedTriangulationTraits_2*.

Refines

DelaunayTriangulationTraits_2
ConstrainedTriangulationTraits_2

Has Models

The kernels of CGAL are models for this traits class.

See Also

TriangulationTraits_2
ConstrainedTriangulationTraits_2
Constrained_triangulation_2<*Gt*,*Tds*,*Itag*>
Constrained_Delaunay_triangulation_2<*Gt*,*Tds*,*Itag*>

ConstrainedTriangulationFaceBase_2

Definition

In a constrained triangulation, the information about constrained edges is stored in the faces of the triangulation. The base face of a constrained triangulation has to be a model of the concept `ConstrainedTriangulationFaceBase_2` which refines the concept `TriangulationFaceBase_2` providing functionalities to deal with constraints.

Refines

`TriangulationFaceBase_2`

Types

Defines the same types as the `TriangulationFaceBase_2` concept

Access Functions

`bool f.is_constrained(int i)`

returns true if the edge between f and its neighbor $f.\text{neighbor}(i)$ is constrained.
Precondition: $0 \leq i \leq 2$.

— advanced —

Modifiers

`void f.set_constraint(int i, bool b)`

sets the edge between f and its neighbor $f.\text{neighbor}(i)$ as a constrained or unconstrained edge according to b .

`void f.set_constraints(bool c0, bool c1, bool c2)`

sets the status (constrained or unconstrained) of the three edges of f .

`void f.reorient()`

Changes the orientation of f by exchanging $\text{vertex}(0)$ with $\text{vertex}(1)$ and $\text{neighbor}(0)$ with $\text{neighbor}(1)$ and the corresponding constrained status.

`void f.ccw_permute()`

performs a counterclockwise permutation of the vertices, neighbors and constrained status of f .

`void f.cw_permute()`

performs a clockwise permutation of the vertices and neighbors and constrained status of f .

Miscellaneous

bool *f.is_valid()* tests the validity of face *f* as a face of a plain triangulation and additionally checks if the edges of *f* are consistently marked as constrained or unconstrained edges in face *f* and its neighbors.

└────────── *advanced* ─────────┘

Has Models

CGAL::Constrained_triangulation_face_base_2<Traits>

See Also

TriangulationFaceBase_2

CGAL::Constrained_triangulation_2<Traits,Tds>

CGAL::Constrained_triangulation_face_base_2<Traits>

ConstrainedTriangulationTraits_2

Definition

The concept `ConstrainedTriangulationTraits_2` defines the requirements for the geometric traits class of a constrained triangulation (*Constrained_Triangulation_2*<*Traits*,*Tds*,*Itag*>) that supports intersections of input constraints (i. e. when the template parameter *Itag* is instantiated by one of the tag classes *Exact_intersections_tag* or *Exact_predicates_tag*). This concept refines the concept *TriangulationTraits_2*, adding requirements for function objects to compute the intersection points of two constraints. When *Exact_predicates_tag* is used, the traits class is also required to provide additional types to compute the squared distance between a point and a line

Refines

TriangulationTraits_2

Types

ConstrainedTriangulationTraits_2:: Intersect_2

A function object whose operator() computes the intersection of two segments :

Object_2 operator()(Segment_2 s1, Segment_2 s2); Returns the intersection of *s1* and *s2*.

When the constrained triangulation is instantiated with the intersection tag *Exact_predicates_tag*, the used algorithms needs to be able to compare some distances between points and lines and the following types are further required.

ConstrainedTriangulationTraits_2:: RT

A number type supporting the comparison operator <.

ConstrainedTriangulationTraits_2:: Line_2

The line type.

ConstrainedTriangulationTraits_2:: Construct_line_2

A function object whose operator() constructs a line from two points :

Line_2 operator()(Point_2 p1, Point_2 p2);

ConstrainedTriangulationTraits_2:: Compute_squared_distance_2

A function object with an operator() designed to compute the squared distance between a line and a point : *RT operator()(Point_2 p1, Line_2);* Return the squared distance between *p* and *l*.

Access to constructor object

Intersect_2

traits.intersect_2_object()

Construct_line_2 *traits.construct_line_2_object()*

required when the intersection tag is *Exact_predicates_tag*.

Compute_squared_distance_2

traits.compute_squared_distance_2_object()

required when the intersection tag is *Exact_predicates_tag*.

Has Models

The kernels of CGAL are models for this traits class.

See Also

TriangulationTraits_2

ConstrainedDelaunayTriangulationTraits_2

CGAL:Constrained_Triangulation_2<*Traits*,*Tds*,*Itag*>

CGAL::Constrained_Delaunay_triangulation_2<Traits,Tds,Itag>

Definition

A constrained Delaunay triangulation is a triangulation with constrained edges which tries to be as much Delaunay as possible. Constrained edges are not necessarily Delaunay edges, therefore a constrained Delaunay triangulation is not a Delaunay triangulation. A constrained Delaunay is a triangulation whose faces do not necessarily fulfill the empty circle property but fulfill a weaker property called the *constrained empty circle*. To state this property, it is convenient to think of constrained edges as blocking the view. Then, a triangulation is constrained Delaunay if the circumscribing circle of any of its triangular faces includes in its interior no vertex that is visible from the interior of the triangle. The class *Constrained_Delaunay_triangulation_2<Traits,Tds,Itag>* is designed to represent constrained Delaunay triangulations.

As in the case of constrained triangulations, three different versions of Delaunay constrained triangulations are offered depending on whether the user wishes to handle intersecting input constraints or not. The desired version can be selected through the instantiation of the third template parameter *Itag* which can be one of the following : *CGAL::No_intersection_tag* if intersections of input constraints are disallowed, *CGAL::Exact_predicates_tag* allows intersections between input constraints and is to be used when the traits class provides exact predicates but approximate constructions of the intersection points. *CGAL::Exact_intersections_tag* allows intersections between input constraints and is to be used in conjunction with an exact arithmetic type.

The template parameters *Tds* has to be instantiate with a model of *TriangulationDataStructure_2*. The geometric traits of a constrained Delaunay triangulation is required to provide the *side_of_oriented_circle* test as the geometric traits of a Delaunay triangulation and the *Traits* parameter has to be instantiated with a model *DelaunayTriangulationTraits_2*. When intersection of input constraints are supported, the geometric traits class is required to provide additional function object types to compute the intersection of two segments. and has then to be also a model of the concept *ConstrainedTriangulationTraits_2*.

A constrained Delaunay triangulation is not a Delaunay triangulation but it is a constrained triangulation. Therefore the class *Constrained_Delaunay_triangulation_2<Traits,Tds,Itag>* derives from the class *Constrained_triangulation_2<Traits,Tds>*. Also, information about the status (constrained or not) of the edges of the triangulation is stored in the faces. Thus the nested *Face* type of a constrained triangulation offers additional functionalities to deal with this information. These additional functionalities induce additional requirements on the base face class plugged into the triangulation data structure of a constrained Delaunay triangulation. The base face of a constrained Delaunay triangulation has to be a model of the concept *ConstrainedTriangulationFaceBase_2*.

CGAL provides a default for the template parameters. If *Gt* is the geometric traits parameter, the default for *ConstrainedTriangulationFaceBase_2* is the class *CGAL::Constrained_triangulation_face_base_2<Gt>* and the default for the triangulation data structure parameter is the class *CGAL::Triangulation_data_structure_2<CGAL::Triangulation_vertex_base_2<Gt>, CGAL::Constrained_triangulation_face_base_2<Gt>>*. The default intersection tag is *CGAL::No_intersection_tag*.

```
#include <CGAL/Constrained_Delaunay_triangulation_2.h>
```

Inherits From

Constrained_triangulation_2<Traits,Tds,Itag>

Types

All types used in this class are inherited from the base class *Constrained_triangulation_2**<Traits,Tds,Itag>*.

Creation

`Constrained_Delaunay_triangulation_2<Traits,Tds,Itag> cdt(Traits t = Traits());`

Introduces an empty constrained Delaunay triangulation *cdt*.

$$\text{Constrained_Delaunay_triangulation_2} \langle \text{Traits}, \text{Tds}, \text{Itag} \rangle \text{ cdt}(\text{Constrained_Delaunay_triangulation_2 cdt1});$$

Copy constructor, all faces and vertices are duplicated and the constrained status of edges is copied.

$$\text{Constrained_Delaunay_triangulation_2}\langle \text{Traits}, \text{Tds}, \text{Itag} \rangle \text{ cdt}(\text{list}\langle \text{Constrained} \rangle \& \text{lc}, \text{Traits } t = \text{Traits}());$$

Introduces a constrained triangulation, the constrained edges of which are the edges of the list lc .

[illegible]

A templated constructor which introduces and builds a constrained triangulation with constrained edges in the range $[first, last)$.

Precondition: The *value_type* of *first* and *last* is *Constraint*.

Insertion and Removal

The following member functions overwrite the corresponding members of the base class to include a step restoring the Delaunay constrained property after modification of the triangulation.

```
Vertex_handle      cdt.insert( Point p, Face_handle f = Face_handle())
```

Inserts point p in the triangulation. If present f is used as an hint for the location of p .

<i>Vertex_handle</i>	<i>cdt.insert(Point p, Locate_type& lt, Face_handle loc, int li)</i>
----------------------	---------------------------------------------------------------------------

Same as above except that the location of the point p to be inserted is assumed to be given by (lt, loc, i) .

Vertex_handle cdt.push_back(Point p)

Equivalent to $insert(p)$.

template < class InputIterator >

int *cdt.insert(InputIterator first, InputIterator last)*

Inserts the points in the range $[first, last)$. Returns the number of inserted points.

Precondition: The *value_type* of *first* and *last* is *Point*.

void *cdt.insert_constraint(Point a, Point b)*

Inserts segment *ab* as a constrained edge in the triangulation.

void *cdt.push_back(Constraint c)*

Inserts constraints *c* as above.

void *cdt.insert_constraint(Vertex_handle va, Vertex_handle vb)*

Inserts the line segment whose endpoints are the vertices *va* and *vb* as a constrained edge *e* in the triangulation.

void *cdt.remove(Vertex_handle & v)*

Removes vertex *v*.

Precondition: Vertex *v* is not incident to a constrained edge.

void *cdt.remove_incident_constraints(Vertex_handle v)*

Make the edges incident to vertex *v* unconstrained edges.

void *cdt.remove_constraint(Face_handle f, int i)*

Edge (f,i) is no longer constrained.

Queries

The following template member functions query the set of faces in conflict with a point *p*. The notion of conflict refers here to a constrained Delaunay setting which means the following. Constrained edges are considered as visibility obstacles and a point *p* is considered to be in conflict with a face *f* iff it is visible from the interior of *f* and included in the circumcircle of *f*.

template <class OutputItFaces, class OutputItBoundaryEdges>

std::pair<OutputItFaces,OutputItBoundaryEdges>

cdt.get_conflicts_and_boundary(Point p,
OutputItFaces fit,
OutputItBoundaryEdges eit,

Face_handle start)

OutItFaces is an output iterator with *Face_handle* as value type. *OutItBoundaryEdges* stands for an output iterator with *Edge* as value type. This members function outputs in the container pointed to by *fit* the faces which are in conflict with point *p*. It outputs in the container pointed to by *eit* the boundary of the zone in conflict with *p*. The boundary edges of the conflict zone are ouput in counterclockwise order and each edge is described through its incident face which is not in conflict with *p*. The function returns in a `std::pair` the resulting output iterators.

```
template <class OutputItFaces>
OutputItFaces      cdt.get_conflicts( Point p, OutputItFaces fit, Face_handle start)
```

Same as above except that only the faces in conflict with *p* are output. The function returns the resulting output iterator.

```
template <class OutputItBoundaryEdges>
OutputItBoundaryEdges

      cdt.get_boundary_of_conflicts( Point p,
                                   OutputItBoundaryEdges eit,
                                   Face_handle start)
```

OutputItBoundaryEdges stands for an output iterator with *Edge* as value type. This functions outputs in the container pointed to by *eit*, the boundary of the zone in conflict with *p*. The boundary edges of the conflict zone are ouput in counterclockwise order and each edge is described through the incident face which is not in conflict with *p*. The function returns the resulting output iterator.

Checking

<i>bool</i>	<i>cdt.is_valid()</i>	Checks if the triangulation is valid and if each constrained edge is consistently marked constrained in its two incident faces.
-------------	-----------------------	---------------------------------------------------------------------------------------------------------------------------------

└────────── advanced ─────────┘

Flips

<i>bool</i>	<i>cdt.is_flipable(Face_handle f, int i)</i>	Determines if edge (f,i) can be flipped. Returns true if edge (f,i) is not constrained and the circle circumscribing <i>f</i> contains the vertex of $f->neighbor(i)$ opposite to edge (f,i) .
-------------	-----------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

void *cdt.flip(Face_handle& f, int i)*

Flip *f* and *f->neighbor(i)*.

void *cdt.propagating_flip(List_edges & edges)*

Makes the triangulation constrained Delaunay by flipping edges. List edges contains an initial list of edges to be flipped. The returned triangulation is constrained Delaunay if the initial list contains at least all the edges of the input triangulation that failed to be constrained Delaunay. (An edge is said to be constrained Delaunay if it is either constrained or locally Delaunay.)

└────────── *advanced* ─────────┘

See Also

CGAL::Constrained_triangulation_2<Traits,Tds,Itag>
TriangulationDataStructure_2
DelaunayTriangulationTraits_2
ConstrainedTriangulationTraits_2
ConstrainedDelaunayTriangulationTraits_2
ConstrainedTriangulationFaceBase_2

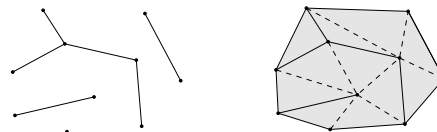
CGAL::Constrained_triangulation_2<Traits,Tds,Itag>

Definition

A constrained triangulation is a triangulation of a set of points which has to include among its edges a given set of segments joining the points. The given segments are called *constraints* and the corresponding edges in the triangulation are called *constrained edges*.

The endpoints of constrained edges are of course vertices of the triangulation. However the triangulation may include other vertices as well. There are three versions of constrained triangulations

- In the basic version, the constrained triangulation does not handle intersecting constraints, and the set of input constraints is required to be a set of segments that do not intersect except possibly at their endpoints. Any number of constrained edges are allowed to share the same endpoint. Vertical constrained edges are allowed as well as constrained edges with null length.
- The two other versions support intersecting input constraints. In those versions, input constraints are allowed to be intersecting, overlapping or partially overlapping segments. The triangulation introduces additional vertices at each point which is a proper intersection point of two constraints. A single constraint intersecting other constraints will then appear as the union of several constrained edges of the triangulation. The two versions dealing with intersecting constraints, slightly differ in the way intersecting constraints are dealt with.
 - One of them is designed to be robust when predicates are evaluated exactly but constructions (i. e. intersection computations) are approximative.
 - The other one is designed to be used with an exact arithmetic (meaning exact evaluation of predicates and exact computation of intersections.) This last version finds its full efficiency when used in conjunction with a constraint hierarchy data structure as provided in the class *Constrained_triangulation_plus_2*. See section 20.9.



The class *Constrained_triangulation_2<Traits,Tds,Itag>* of the CGAL library implements constrained triangulations. The template parameter *Traits* stands for a geometric traits class. It has to be a model of the concept *TriangulationTraits_2*. When intersection of input constraints are supported, the geometric traits class is required to provide additional function object types to compute the intersection of two segments. It has then to be a model of the concept *ConstrainedTriangulationTraits_2*. The template parameter *Tds* stands for a triangulation data structure class that has to be a model of the concept *TriangulationDataStructure_2*. The third parameter *Itag* is the intersection tag which serves to choose between the different strategies to deal with constraints intersections. CGAL provides three valid types for this parameter :

CGAL::No_intersection_tag disallows intersections of input constraints,

CGAL::Exact_predicates_tag is to be used when the traits class provides exact predicates but approximate constructions of the intersection points.

CGAL::Exact_intersections_tag is to be used in conjunction with an exact arithmetic type.

The information about constrained edges is stored in the faces of the triangulation. Thus the nested *Face* type of a constrained triangulation offers additional functionalities to deal with this information. These additional

functionalities induce additional requirements on the base face class plugged into the triangulation data structure of a constrained Delaunay triangulation. The base face of a constrained Delaunay triangulation has to be a model of the concept *ConstrainedTriangulationFaceBase_2*.

CGAL provides default instantiations for the template parameters *Tds* and *Itag*, and for the *ConstrainedTriangulationFaceBase_2*. If *Gt* is the geometric traits parameter, the default for *ConstrainedTriangulationFaceBase_2* is the class *CGAL::Constrained_triangulation_face_base_2<Gt>* and the default for the triangulation data structure parameter is the class *CGAL::Triangulation_data_structure_2 <CGAL::Triangulation_vertex_base_2<Gt>, CGAL::Constrained_triangulation_face_base_2<Gt> >*. The default intersection tag is *CGAL::No_intersection_tag*.

```
#include <CGAL/Constrained_triangulation_2.h>
```

Inherits From

Triangulation_2<Traits,Tds>

Types

```
typedef std::pair<Point,Point>
```

Constraint; The type of input constraints

```
typedef Itag                      Intersection_tag;                      The intersection tag which decides how intersections between input constraints are dealt with.
```

Creation

```
Constrained_triangulation_2<Traits,Tds,Itag> ct;
```

default constructor.

```
Constrained_triangulation_2<Traits,Tds,Itag> ct( Constrained_triangulation_2 ct1);
```

Copy constructor, all faces and vertices are duplicated and the constrained status of edges is copied.

```
Constrained_triangulation_2<Traits,Tds,Itag> ct( std::list<Constraint>& lc, Traits t = Traits());
```

Introduces a constrained triangulation, the constrained edges of which are the edges of the list *lc*.

```
template<class InputIterator>
```

```
Constrained_triangulation_2<Traits,Tds,Itag> ct( InputIterator first, InputIterator last, Traits t=Traits());
```

A templated constructor which introduces and builds a constrained triangulation with constrained edges in the range *[first, last)*.

Precondition: The *value_type* of *first* and *last* is *Constraint*.

Queries

bool *ct.is_constrained(Edge e)*

Returns true if edge *e* is a constrained edge.

bool *ct.are_there_incident_constraints(Vertex_handle v)*

Returns true if at least one of the edges incident to vertex *v* is constrained.

template<class OutputItEdges>

OutputItEdges *ct.incident_constraints(Vertex_handle v, OutputItEdges out)*

OutputItEdges is an output iterator with *Edge* as value type. Outputs the constrained edges incident to *v* in the sequence pointed to by *out* and returns the resulting output iterator.

Insertion and removal

Vertex_handle *ct.insert(Point p, Face_handle f = Face_handle())*

Inserts point *p* and restores the status (constrained or not) of all the touched edges. If present *f* is used as an hint for the location of *p*.

Vertex_handle *ct.insert(Point p, Locate_type& lt, Face_handle loc, int li)*

Same as above except that the location of the point *p* to be inserted is assumed to be given by *(lt,loc,i)*.

Vertex_handle *ct.push_back(Point p)*

Equivalent to *insert(p)*.

template < class InputIterator >

int *ct.insert(InputIterator first, InputIterator last)*

Inserts the points in the range $[first, last)$. Returns the number of inserted points.

Precondition: The *value_type* of *first* and *last* is *Point*.

void *ct.insert_constraint(Point a, Point b)*

Inserts points *a* and *b*, and inserts segment *ab* as a constraint. Removes the faces crossed by segment *ab* and creates new faces instead. If a vertex *c* lies on segment *ab*, constraint *ab* is replaced by the two constraints *ac* and *cb*. Apart from the insertion of *a* and *b*, the algorithm runs in time proportionnal to the number of removed triangles.

Precondition: The relative interior of segment *ab* does not intersect the relative interior of another constrained edge.

void *ct.push_back(Constraint c)*

Inserts constraints *c* as above.

void *ct.insert_constraint(Vertex_handle va, Vertex_handle vb)*

Inserts the line segment *s* whose endpoints are the vertices *va* and *vb* as a constrained edge *e*. The triangles intersected by *s* are removed and new ones are created.

void *ct.remove(Vertex_handle v)*

Removes a vertex *v*.

Precondition: Vertex *v* is not incident to a constrained edge.

void *ct.remove_incident_constraints(Vertex_handle v)*

Make the edges incident to vertex *v* unconstrained edges.

void *ct.remove_constrained_edge(Face_handle f, int i)*

Make edge *(f,i)* no longer constrained.

┌────────── *advanced* ───────────┐
|
bool *ct.is_valid(bool verbose = false, int level = 0)*

Checks the validity of the triangulation and the consistency of the constrained marks in edges.

└────────── *advanced* ───────────┘

I/O

ostream & *ostream& os << Constrained_triangulation_2<Traits,Tds> Ct*

Writes the triangulation and, for each face *f*, and integers *i=0,1,2*, write “C” or “N” depending whether edge *(f,i)* is constrained or not.

See Also

CGAL::Triangulation_2<Traits,Tds>,
TriangulationDataStructure_2,
TriangulationTraits_2
ConstrainedTriangulationTraits_2
ConstrainedTriangulationFaceBase_2

Implementation

The insertion of a constrained edge runs in time proportionnal to the number of triangles intersected by this edge.

CGAL::Constrained_triangulation_face_base_2<Traits,Fb>

Definition

The class *Constrained_triangulation_face_base_2<Traits,Fb>* is the default model for the concept *ConstrainedTriangulationFaceBase_2* to be used as base face class of constrained triangulations.

```
#include <CGAL/Constrained_triangulation_face_base_2.h>
```

Is Model for the Concepts

ConstrainedTriangulationFaceBase_2

Parameters

The first template parameter is a geometric traits.

The second template parameter has to be a model of the concept *TriangulationFaceBase_2*. Its default is *CGAL::Triangulation_face_base_2<Traits>*

Inherits From

The class *Constrained_triangulation_face_base_2<Traits,Fb>* derives from its parameter *Fb*. and add three boolean to deal with information about constrained edges.

The member functions *cw(int i)*, *ccw(int i)* and *reorient* are overloaded to update information about constrained edges.

See Also

TriangulationFaceBase_2
ConstrainedTriangulationFaceBase_2
CGAL::Constrained_triangulation_2<Traits,Tds>
CGAL::Triangulation_face_base_2<Traits>

CGAL::Constrained_triangulation_plus_2<Tr>

The class *Constrained_triangulation_plus_2<Tr>* implements a constrained triangulation with an additional data structure called the constraint hierarchy that keeps track of the input constraints and of their refinement in the triangulation.

The class *Constrained_triangulation_plus_2<Tr>* inherits from its template parameter *Tr*, which has to be instantiated by a constrained or constrained Delaunay triangulation.

According to its intersection tag, the base class will support intersecting input constraints or not. When intersections of input constraints are supported, the base class constructs a triangulation of the arrangement of the constraints, introducing new vertices at each proper intersection point and refining the input constraints into subconstraints which are edges (more precisely constrained edges) of the triangulation. In this context, the constraint hierarchy keeps track of the input constraints and of their refinement in the triangulation. This data structure maintains for each input constraints the sequence of intersection vertices added on this constraint. The constraint hierarchy also allows the user to retrieve the set of constrained edges of the triangulation, and for each constrained edge, the set of input constraints that overlap it.

```
#include <CGAL/Constrained_triangulation_plus_2.h>
```

Inherits From

Tr

Types

<i>typedef Tr</i>	<i>Triangulation;</i>	the triangulation base class.
<i>typedef Itag</i>	<i>Intersection_tag;</i>	the intersection tag.s

```
Constrained_triangulation_plus_2<Tr>:: Constraint_iterator;
```

An iterator to visit all the input constraints. The order of visit is arbitrary. The value type of this iterator is a pair *std::pair<Vertex_handle, Vertex_handle>* corresponding to the endpoints of the constraint.

```
Constrained_triangulation_plus_2<Tr>:: Subconstraint_iterator;
```

An iterator to visit all the subconstraints of the triangulation. The order of visit is arbitrary. The value type of this iterator is a pair *std::pair<Vertex_handle, Vertex_handle>* corresponding to the vertices of the subconstraint.

```
Constrained_triangulation_plus_2<Tr>:: Vertices_in_constraint_iterator;
```

An iterator on the vertices of the chain of triangulation edges representing a constraint. The value type of this iterator is *Vertex_handle*.

<i>typedef</i>	<i>Context</i> ;	This type is intended to describe a constraint enclosing a sub-constraint and the position of the subconstraint in this constraint. It provides three member functions <i>vertices_begin()</i> , <i>vertices_end()</i> and <i>current()</i> returning iterators of the type <i>Vertices_in_constraint_iterator</i> on the sequence of vertices of the enclosing constraint. These iterators point respectively on the first vertex of the enclosing constraint, past the last vertex and on the first vertex of the subconstraint.
<i>typedef</i>	<i>Context_iterator</i> ;	An iterator on constraints enclosing a given subconstraint. The value type of this iterator is <i>Context</i> .

Creation

Constrained_triangulation_plus_2<Tr> *ctp*(*Geom_traits* gt=*Geom_traits*());

Introduces an empty triangulation.

Constrained_triangulation_plus_2<Tr> *ctp*(*Constrained_triangulation_plus_2* ct);

Copy constructor.

Constrained_triangulation_plus_2<Tr> *ctp*(*std::list*<*Constrained*>& lc, *Geom_traits* t = *Geom_traits*());

Introduces and builds a constrained triangulation from the list of constraints *lc*.

template<*class InputIterator*>
Constrained_triangulation_plus_2<Tr> *ctp*(*InputIterator* first,
InputIterator last,
Geom_traits gt= *Geom_traits*())

Introduces and builds a constrained triangulation from the constraints in the range *[first, last)*.

Precondition: The *value_type* of *first* and *last* is *Constraint*.

void *~ctp*()

Destructor. All vertices and faces are deleted. The constraint hierarchy is deleted.

Assignment

Constrained_triangulation_plus_2

ctp = *Constrained_triangulation_plus_2* tr

Assignment. All the vertices and faces are duplicated. The constraint hierarchy is also duplicated.

void *ctp.swap(Constrained_triangulation_plus_2 tr)*

The triangulations *tr* and *ctp* are swapped. This operation should be preferred to *ctp = tr* or to *t(tr)* if *tr* is deleted after that.

Insertion and Removal

The class *Constrained_triangulation_plus_2<Tr>* overwrites the following insertion and removal member functions for points and constraints.

Vertex_handle *ctp.insert(Point p, Face_handle start = Face_handle())*

Inserts point *p* as a vertex of the triangulation.

Vertex_handle *ctp.insert(Point p, Locate_type lt, Face_handle loc, int li)*

inserts a point *p* whose location is assumed to be given by *(lt,loc,li)*.

Vertex_handle *ctp.push_back(Point p)*

Equivalent to *insert(p)*.

template < class InputIterator >

int *ctp.insert(InputIterator first, InputIterator last)*

Inserts the points in the range *[first, last)*. Returns the number of inserted points.

Precondition: The *value_type* of *first* and *last* is *Point*.

void *ctp.insert_constraint(Point a, Point b)*

Inserts the constraint segment *ab* in the triangulation.

void *ctp.push_back(Constraint c)*

Inserts the constraint *c*.

void *ctp.insert_constraint(Vertex_handle va, Vertex_handle vb)*

Inserts a constraint whose endpoints are the vertices pointed by *va* and *vb* in the triangulation.

<i>void</i>	<i>ctp.remove_constraint(Vertex_handle va, Vertex_handle vb)</i>	<p>Removes the constraint joining the vertices pointed by <i>va</i> and <i>vb</i>.</p> <p><i>Precondition:</i> <i>va</i> and <i>vb</i> have to refer to the endpoint vertices of an input constraint.</p>
Queries		
<i>Constraint_iterator</i>	<i>ctp.constraints_begin()</i>	Returns a <i>Constraint_iterator</i> pointing on the first constraint.
<i>Constraint_iterator</i>	<i>ctp.constraints_end()</i>	Returns a <i>Constraint_iterator</i> pointing past the last constraint.
<i>Subconstraint_iterator</i>	<i>ctp.subconstraints_begin()</i>	Returns a <i>Subconstraint_iterator</i> pointing on the first subconstraint.
<i>Subconstraint_iterator</i>	<i>ctp.subconstraints_end()</i>	Returns a <i>Subconstraint_iterator</i> pointing past the last subconstraint.
<i>int</i>	<i>ctp.number_of_enclosing_constraints(Vertex_handle va, Vertex_handle vb)</i>	<p>Returns the number of constraints enclosing the subconstraint (<i>va,vb</i>).</p> <p><i>Precondition:</i> <i>va</i> and <i>vb</i> refer to the vertices of a constrained edge of the triangulation.</p>
<i>Context</i>	<i>ctp.context(Vertex_handle va, Vertex_handle vb)</i>	<p>Returns the <i>Context</i> relative to one of the constraint enclosing the subconstraint (<i>va,vb</i>).</p> <p><i>Precondition:</i> <i>va</i> and <i>vb</i> refer to the vertices of a constrained edge of the triangulation.</p>
<i>Context_iterator</i>	<i>ctp.contexts_begin(Vertex_handle va, Vertex_handle vb)</i>	<p>Returns an iterator pointing on the first <i>Context</i> of the sequence of <i>Contexts</i> corresponding to the constraints enclosing the subconstraint(<i>va,vb</i>).</p> <p><i>Precondition:</i> <i>va</i> and <i>vb</i> refer to the vertices of a constrained edge of the triangulation.</p>
<i>Context_iterator</i>	<i>ctp.contexts_end(Vertex_handle va, Vertex_handle vb)</i>	<p>Returns an iterator past the last <i>Context</i> of the sequence of <i>Contexts</i> corresponding to the constraints enclosing the (<i>va,vb</i>).</p> <p><i>Precondition:</i> <i>va</i> and <i>vb</i> refer to the vertices of a constrained edge of the triangulation.</p>

Vertices_in_constraint_iterator

ctp.vertices_in_constraint_begin(Vertex_handle va, Vertex_handle vb)

Returns an iterator on the first vertex on the constraint (*va,vb*)

Precondition: *va* and *vb* refer to the vertices of an input constraint.

Vertices_in_constraint_iterator

ctp.vertices_in_constraint_end(Vertex_handle va, Vertex_handle vb)

Returns an iterator past the last vertex on the constraint (*va,vb*)

Precondition: *va* and *vb* refer to the vertices of an input constraints.

See Also

CGAL::Constrained_triangulation_2<Traits,Tds>

CGAL::Constrained_Delaunay_triangulation_2<Traits,Tds>

ConstrainedTriangulationTraits_2

ConstrainedDelaunayTriangulationTraits_2

DelaunayTriangulationTraits_2

Definition

In addition to the requirements of the concept *TriangulationTraits_2* described page 1424, the concept *DelaunayTriangulationTraits_2* provide a predicate to check the empty circle property. The corresponding predicate type is called type *Side_of_oriented_circle_2*.

The additional types *Line_2*, *Ray_2* and the constructor objects *Construct_ray_2* *Construct_circumcenter_2*, *Construct_bisector_2*, *Construct_midpoint* are used to build the dual Voronoi diagram and are required only if the dual functions are called. The additional predicate type *Compare_distance_2* is required if calls to *nearest_vertex(..)* are issued.

Refines

TriangulationTraits_2

Types

DelaunayTriangulationTraits_2:: Line_2 The line type. This type is required only if some dual functions are called.

DelaunayTriangulationTraits_2:: Ray_2 The type for ray. This type is required only if some dual functions are called.

DelaunayTriangulationTraits_2:: Side_of_oriented_circle_2

Predicate type. Provides the operator :
Oriented_side operator()(Point p, Point q, Point r, Point s) which takes four points *p,q,r,s* as arguments and returns *ON_POSITIVE_SIDE*, *ON_NEGATIVE_SIDE* or *ON_ORIENTED_BOUNDARY* according to the position of points *s* with respect to the oriented circle through *p,q* and *r*.

DelaunayTriangulationTraits_2:: Compare_distance_2

Predicate type. Provides the operator :
Comparison_result operator()(Point_2 p, Point_2 q, Point_2 r) which returns *SMALLER*, *EQUAL* or *LARGER* according to the distance between *p* and *q* being smaller, equal or larger than the distance between *p* and *r*. This type is only require if *nearest_vertex* queries are issued.

DelaunayTriangulationTraits_2:: Construct_circumcenter_2

Constructor object. Provides the operator :
Point_2 operator()(Point_2 p, Point_2 q, Point_2 r) which returns the circumcenter of the three points *p, q* and *r*. This type is required only if functions relative to the dual Voronoi diagram are called.

DelaunayTriangulationTraits_2:: Construct_bisector_2

Constructor object. Provides the operator :
Line_2 operator()(Point_2 p, Point_2 q) which constructs the bisector line of points p and q . This type is required only if functions relative to the dual Voronoi diagram are called.

DelaunayTriangulationTraits_2:: Construct_ray_2

A constructor object to build a ray from a point and a line.
Provides :
Ray_2 operator() (Point_2 p, Line_2 l);

Creation

DelaunayTriangulationTraits_2 traits; default constructor.
DelaunayTriangulationTraits_2 traits(dt); copy constructor
DelaunayTriangulationTraits_2 traits = traits2

Assignment operator.

Access to predicate and constructor objects

Side_of_oriented_circle_2 traits.side_of_oriented_circle_2_object()

The following functions are required only if member functions of the Delaunay triangulation relative to the dual Voronoi diagram are called. *Compare_distance_2 traits.compare_distance_2_object()*

Construct_circumcenter_2 traits.construct_circumcenter_2_object()

Construct_bisector_2 traits.construct_bisector_2_object()

Construct_direction_2 traits.construct_direction_2_object()

Construct_ray_2 traits.construct_ray_2_object()

Has Models

CGAL kernels

CGAL::Triangulation_euclidean_traits_2<Rep>.

The following traits class provide everything except types and member functions required for the dual Voronoi diagram ;

CGAL::Triangulation_euclidean_traits_xy_3<Rep>.

CGAL::Triangulation_euclidean_traits_yz_3<Rep>.

CGAL::Triangulation_euclidean_traits_zx_3<Rep>.

See Also

TriangulationTraits_2

CGAL::Delaunay_triangulation_2<Traits,Tds>

Definition

The class *Delaunay_triangulation_2<Traits,Tds>* is designed to represent the Delaunay triangulation of a set of points in a plane. A Delaunay triangulation of a set of points is a triangulation of the sets of points that fulfills the following *empty circle property* (also called *Delaunay property*): the circumscribing circle of any facet of the triangulation contains no point of the set in its interior. For a point set with no case of cocircularity of more than three points, the Delaunay triangulation is unique, it is the dual of the Voronoi diagram of the points.

Inherits From

Triangulation_2<Traits,Tds>.

Parameters

The template parameter *Tds* is to be instantiated with a model of *TriangulationDataStructure_2*. CGAL provides a default instantiation for this parameter, which is the class *CGAL::Triangulation_data_structure_2 <CGAL::Triangulation_vertex_base_2<Traits>, CGAL::Triangulation_face_base_2<Traits> >*.

The geometric traits *Traits* is to be instantiated with a model of *DelaunayTriangulationTraits_2*. The concept *DelaunayTriangulationTraits_2* refines the concept *TriangulationTraits_2*, providing a predicate type to check the empty circle property.

Changing this predicate type allows to build Delaunay triangulations for different metrics such that L_1 or L_∞ or any metric defined by a convex object. However, the user of an exotic metric must be careful that the constructed triangulation has to be a triangulation of the convex hull which means that convex hull edges have to be Delaunay edges. This is granted for any smooth convex metric (like L_2) and can be ensured for other metrics (like L_∞) by the addition to the point set of well chosen sentinel points. The concept of *DelaunayTriangulationTraits_2* is described page [1399](#).

When dealing with a large triangulations, the user is advised to encapsulate the Delaunay triangulation class into a triangulation hierarchy, which means to use the class *Triangulation_hierarchy_2<Tr>* with the template parameter instantiated with *Delaunay_triangulation_2<Traits,Tds>*. The triangulation hierarchy will then offer the same functionalities but be much more efficient for locations and insertions.

```
#include <CGAL/Delaunay_triangulation_2.h>
```

Inherits From

Triangulation_2<Traits,Tds>

Types

Inherits all the types defined in *Triangulation_2<Traits,Tds>*.

Creation

Delaunay_triangulation_2<Traits,Tds> dt(Traits gt = Traits());

default constructor.

Delaunay_triangulation_2<Traits,Tds> dt(tr);

copy constructor. All the vertices and faces are duplicated.

Insertion and Removal

The following insertion and removal functions overwrite the functions inherited from the class *Triangulation_2*<Traits,Tds> to maintain the Delaunay property.

Vertex_handle dt.insert(Point p, Face_handle f=Face_handle())

inserts point *p*. If point *p* coincides with an already existing vertex, this vertex is returned and the triangulation is not updated. Optional parameter *f* is used to initialize the location of *p*.

Vertex_handle dt.insert(Point p, Locate_type& lt, Face_handle loc, int li)

inserts a point *p*, the location of which is supposed to be given by (*lt,loc,li*), see the description of member function *locate* in class *Triangulation_2*<Traits,Tds>.

Vertex_handle dt.push_back(Point p)

equivalent to *insert(p)*.

template < class InputIterator >

int dt.insert(InputIterator first, InputIterator last)

inserts the points in the range [*first, last*). Returns the number of inserted points.

Precondition: The *value_type* of *first* and *last* is *Point*.

void dt.remove(Vertex_handle v)

removes the vertex from the triangulation.

Note that the other modifier functions of *Triangulation_2*<Traits,Tds> are not overwritten. Thus a call to *insert_in_face* *insert_in_edge*, *insert_outside_convex_hull*, *insert_outside_affine_hull* or *flip* on a valid Delaunay triangulation might lead to a triangulation which is no longer a Delaunay triangulation.

Queries

Vertex_handle *dt.nearest_vertex(Point p, Face_handle f=Face_handle())*

returns any nearest vertex of *p*. The implemented function begins with a location step and *f* may be used to initialize the location.

template <class OutputItFaces, class OutputItBoundaryEdges>
std::pair<OutputItFaces, OutputItBoundaryEdges>

dt.get_conflicts_and_boundary(Point p,
OutputItFaces fit,
OutputItBoundaryEdges eit,
Face_handle start)

OutputItFaces is an output iterator with *Face_handle* as value type. *OutputItBoundaryEdges* stands for an output iterator with *Edge* as value type. This members function outputs in the container pointed to by *fit* the faces which are in conflict with point *p* i. e. the faces whose circumcircle contains *p*. It outputs in the container pointed to by *eit* the the boundary of the zone in conflict with *p*. The boundary edges of the conflict zone are ouput in counter-clockwise order and each edge is described through its incident face which is not in conflict with *p*. The function returns in a *std::pair* the resulting output iterators.

template <class OutputItFaces>
OutputItFaces *dt.get_conflicts(Point p, OutputItFaces fit, Face_handle start)*

same as above except that only the faces in conflict with *p* are output. The function returns the resulting output iterator.

template <class OutputItBoundaryEdges>
OutputItBoundaryEdges

dt.get_boundary_of_conflicts(Point p,
OutputItBoundaryEdges eit,
Face_handle start)

OutputItBoundaryEdges stands for an output iterator with *Edge* as value type. This function outputs in the container pointed to by *eit*, the boundary of the zone in conflict with *p*. The boundary edges of the conflict zone are ouput in counterclockwise order and each edge is described through the incident face which is not in conflict with *p*. The function returns the resulting output iterator.

Voronoi diagram

The following member functions provide the elements of the dual Voronoi diagram.

<i>Point</i>	<i>dt.dual(Face_handle f)</i>	Returns the center of the circle circumscribed to face <i>f</i> . <i>Precondition: f</i> is not infinite
<i>Object</i>	<i>dt.dual(Edge e)</i>	returns a segment, a ray or a line supported by the bisector of the endpoints of <i>e</i> . If faces incident to <i>e</i> are both finite, a segment whose endpoints are the duals of each incident face is returned. If only one incident face is finite, a ray whose endpoint is the dual of the finite incident face is returned. Otherwise both incident faces are infinite and the bisector line is returned.
<i>Object</i>	<i>dt.dual(Edge_circulator ec)</i>	Idem
<i>Object</i>	<i>dt.dual(Edge_iterator ei)</i>	Idem
<i>template < class Stream></i> <i>Stream&</i>		
	<i>dt.draw_dual(Stream & ps)</i>	output the dual voronoi diagram to stream <i>ps</i> .

Predicates

<i>Oriented_side</i>	<i>dt.side_of_oriented_circle(Face_handle f, Point p)</i>	Returns the side of <i>p</i> with respect to the circle circumscribing the triangle associated with <i>f</i>
----------------------	------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------

└────────── *advanced* ─────────┘

Miscellaneous

The checking function *is_valid()* is also overwritten to additionally test the empty circle property.

<i>bool</i>	<i>dt.is_valid(bool verbose = false, int level = 0)</i>	tests the validity of the triangulation as a <i>Triangulation_2</i> and additionally tests the Delaunay property. This method is mainly useful for debugging Delaunay triangulation algorithms designed by the user.
-------------	----------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

└────────── *advanced* ─────────┘

See Also

CGAL::Triangulation_2<Traits,Tds>,
TriangulationDataStructure_2,
DelaunayTriangulationTraits_2,
Triangulation_hierarchy_2<Tr>.

Implementation

Insertion is implemented by inserting in the triangulation, then performing a sequence of Delaunay flips. The number of flips is $O(d)$ if the new vertex is of degree d in the new triangulation. For points distributed uniformly at random, insertion takes time $O(1)$ on average.

Removal calls the removal in the triangulation and then retriangulates the hole in such a way that the Delaunay criterion is satisfied. Removal of a vertex of degree d takes time $O(d^2)$. The degree d is $O(1)$ for a random vertex in the triangulation.

After a point location step, the nearest neighbor is found in time $O(n)$ in the worst case, but in time $O(1)$ for vertices distributed uniformly at random and any query point.

CGAL::Triangulation_2<Traits,Tds>::Locate_type

Definition

The enum *Locate_type* is defined by the *Triangulation_2<Traits,Tds>* class to specify which case occurs when locating a point in the triangulation.

```
enum Locate_type { VERTEX=0, EDGE, FACET, OUTSIDE_CONVEX_HULL, OUTSIDE_AFFINE_HULL};
```

The locate type is :

VERTEX when the located point coincides with a vertex of the triangulation

EDGE when the point is in the relative interior of an edge

FACET when the point is in the interior of a facet

OUTSIDE_CONVEX_HULL when the point is outside the convex hull but in the affine hull of the current triangulation

OUTSIDE_AFFINE_HULL when the point is outside the affine hull of the current triangulation.

See Also

CGAL::Triangulation_2<Traits,Tds>.

RegularTriangulationFaceBase_2

Definition

The regular triangulation of a set of weighted points does not necessarily have one vertex for each of the input points. Some of the input weighed points have no cell in the dual power diagrams and therefore do not correspond to a vertex of the regular triangulation. Those weighted points are said to be *hidden* points. A point which is hidden at a given time may appear later as a vertex of the regular triangulation upon removal on some other weighted point. Therefore, hidden points have to be stored somewhere. The regular triangulation store those hidden points in special vertices called *hidden* vertices.

A hidden point can appear as vertex of the triangulation only when the two dimensional face where its point component is located (the face which hides it) is removed. Therefore we decided to store in each face of a regular triangulation the list of hidden vertices whose points are located in the face. Thus points hidden by a face are easily reinserted in the triangulation when the face is removed.

The base face of a regular triangulation has to be a model of the concept `RegularTriangulationFaceBase_2` , which refines the concept `TriangulationFaceBase_2` by adding in the face a list to store hidden vertices.

Refines

`TriangulationFaceBase_2`

Types

`typedef std::list<Vertex_handle>`

`Vertex_list;` An std list of hidden vertices.

Access Functions

`Vertex_list&` `rfb.vertex_list()` Returns a reference to the list of vertices hidden by the face.

Has Models

`CGAL::Regular_triangulation_face_base_2`

See Also

`TriangulationFaceBase_2`

`RegularTriangulationVertexBase_2`

RegularTriangulationTraits_2

Definition

The concept `RegularTriangulationTraits_2` describe the requirements for the traits class of regular triangulations. It refines the concept `TriangulationTraits_2` providing the type `Weighted_point_2` and the *power-test* predicate on those weighted points. A weighted point is basically a point augmented with a scalar weight. It can be seen as a circle when the weight is interpreted as a square radius. The *power-test* on weighted points is the fundamental test to build regular triangulations as the *side_of_oriented_circle* test is the fundamental test of Delaunay triangulations.

Refines

`TriangulationTraits_2`

Types

`RegularTriangulationTraits_2::Bare_point` Another name for the point type.
`RegularTriangulationTraits_2::Weighted_point_2`

The weighted point type, it has to be a model of the concept `WeightedPoint`.

`RegularTriangulationTraits_2::Power_test_2` A predicate object type. Must provide the operators:
 — `Oriented_side operator() (Weighted_point_2 p, Weighted_point_2 q, Weighted_point_2 r, Weighted_point_2 s)` which is the power test for points p, q, r and s .
Precondition: the bare points corresponding to p, q, r are not collinear.
 — `Oriented_side operator() (Weighted_point_2 p, Weighted_point_2 q, Weighted_point_2 r)` which is the degenerated power test for collinear points p, q, r .
Precondition: the bare points corresponding to p, q, r are collinear and $p \neq q$.
 — `Oriented_side operator() (Weighted_point_2 p, Weighted_point_2 q)` which is the degenerated power test for weighted points p and q whose corresponding bare-points are identical.
Precondition: the bare points corresponding to p and q are identical.

The following type/predicate is required if a call to `nearest_power_vertex` is issued:

`RegularTriangulationTraits_2::Compare_power_distance_2`

A predicate object type. Must provide the operator:
`Comparison_result operator()(Bare_point p, Weighted_point_2 q, Weighted_point_2 r)`, which compares the power distance between p and q to the power distance between p and r .

RegularTriangulationTraits_2::Construct_weighted_circumcenter_2

A constructor object which constructs the weighted circumcenter of three weighted points. Provides the operator *Bare_point operator()* (*Weighted_point_2 p*, *Weighted_point_2 q*, *Weighted_point_2 r*);

RegularTriangulationTraits_2::Construct_radical_axis_2

A constructor type which constructs the radical axis of two weighted points. Provides the operator : *Line_2 operator()* (*Weighted_point_2 p*, *Weighted_point_2 q*);

Creation

RegularTriangulationTraits_2 traits; default constructor.

RegularTriangulationTraits_2 traits(*RegularTriangulationTraits_2*);

copy constructor.

RegularTriangulationTraits_2&

traits = RegularTriangulationTraits_2

assignement operator

Access to predicate and constructors objects

<i>Power_test_2</i>	<i>traits.power_test_2_object()</i>
<i>Compare_power_distance_2</i>	<i>traits.compare_power_distance_2_object()</i>
<i>Construct_weighted_circumcenter_2</i>	<i>traits.construct_weighted_circumcenter_2_object()</i>
<i>Construct_radical_axis_2</i>	<i>traits.construct_radical_axis_2_object()</i>

Has Models

CGAL::Regular_triangulation_traits_2<Rep>
CGAL::Regular_triangulation_filtered_traits_2<FK>

See Also

TriangulationTraits_2
CGAL::Regular_triangulation_2<Traits,Tds>

RegularTriangulationVertexBase_2

Definition

The regular triangulation of a set of weighted points does not necessarily have one vertex for each of the input points. Some of the input weighed points have no cell in the dual power diagrams and therefore do not correspond to a vertex of the regular triangulation. Those weighted point are said to be *hidden* points. A point which is hidden at a given time may appear later as a vertex of the regular triangulation upon removal on some other weighted point. Therefore, hidden points have to be stored somewhere. The regular triangulation store those hidden points in special vertices called *hidden* vertices.

A hidden point can appear as vertex of the triangulation only when the two dimensional face where its point component is located (the face which hides it) is removed. Therefore we decided to store in each face of a regular triangulation the list of hidden vertices whose points are located in the face. Thus points hidden by a face are easily reinserted in the triangulation when the face is removed.

The base vertex of a regular triangulation has to be a model of the concept `RegularTriangulationVertexBase_2`. The concept `RegularTriangulationVertexBase_2` refines the concept `TriangulationVertexBase_2`, just adding a boolean to mark if the vertex is a vertex of the triangulation or a hidden vertex.

Refines

`TriangulationVertexBase_2`

Access Functions

bool *rvb.is_hidden()* returns *true*, iff the vertex is hidden.

void *rvb.set_hidden(bool b)*

Mark the vertex as hidden or as not hidden.

Has Models

`CGAL::Regular_triangulation_vertex_base_2`

See Also

`TriangulationVertexBase_2` `CGAL::Regular_triangulation_vertex_base_2`

CGAL::Regular_triangulation_2<Traits,Tds>

Definition

The class *Regular_triangulation_2<Traits,Tds>* is designed to maintain the regular triangulation of a set of weighted points.

Let $PW = \{(p_i, w_i), i = 1, \dots, n\}$ be a set of weighted points where each p_i is a point and each w_i is a scalar called the weight of point p_i . Alternatively, each weighted point (p_i, w_i) can be regarded as a two dimensional sphere with center p_i and radius $r_i = \sqrt{w_i}$.

The power diagram of the set PW is a planar partition such that each cell corresponds to sphere (p_i, w_i) of PW and is the locus of points p whose power with respect to (p_i, w_i) is less than its power with respect to any other sphere (p_j, w_j) in PW . The dual of this diagram is a triangulation whose domain covers the convex hull of the set $P = \{p_i, i = 1, \dots, n\}$ of center points and whose vertices are a subset of P . Such a triangulation is called a regular triangulation. The three points p_i, p_j and p_k of P form a triangle in the regular triangulation of PW iff there is a point p of the plane whose powers with respect to (p_i, w_i) , (p_j, w_j) and (p_k, w_k) are equal and less than the power of p with respect to any other sphere in PW .

Let us defined the power product of two weighted points (p_i, w_i) and (p_j, w_j) as:

$$\Pi(p_i, w_i, p_j, w_j) = p_i p_j^2 - w_i - w_j.$$

$\Pi(p_i, w_i, p_j, 0)$ is simply the power of point p_j with respect to the sphere (p_i, w_i) , and two weighted points are said to be orthogonal if their power product is null. The power circle of three weighted points (p_i, w_i) , (p_j, w_j) and (p_k, w_k) is defined as the unique circle (π, ω) orthogonal to (p_i, w_i) , (p_j, w_j) and (p_k, w_k) .

The regular triangulation of the sets PW satisfies the following *regular property* (which just reduces to the Delaunay property when all the weights are null): a triangle $p_i p_j p_k$ of the regular triangulation of PW is such that the power product of any weighted point (p_l, w_l) of PW with the power circle of (p_i, w_i) , (p_j, w_j) is (p_k, w_k) is positive or null. We call power test of the weighted point (p_l, w_l) with respect to the face $p_i p_j p_k$, the predicates testing the sign of the power product of (p_l, w_l) with respect to the power circle of (p_i, w_i) , (p_j, w_j) is (p_k, w_k) . This power product is given by the following determinant

$$\begin{vmatrix} 1 & x_i & y_i & x_i^2 + y_i^2 - w_i \\ 1 & x_j & y_j & x_j^2 + y_j^2 - w_j \\ 1 & x_k & y_k & x_k^2 + y_k^2 - w_k \\ 1 & x_l & y_l & x_l^2 + y_l^2 - w_l \end{vmatrix}$$

A pair of neighboring faces $p_i p_j p_k$ and $p_i p_j p_l$ is said to be locally regular (with respect to the weights in PW) if the power test of (p_l, w_l) with respect to $p_i p_j p_k$ is positive. A classical result of computational geometry establishes that a triangulation of the convex hull of P such that any pair of neighboring faces is regular with respect to PW , is a regular triangulation of PW .

Alternatively, the regular triangulation of the weighted points set PW can be obtained as the projection on the two dimensional plane of the convex hull of the set of three dimensional points $P' = \{(p_i, p_i^2 - w_i), i = 1, \dots, n\}$.

The vertices of the regular triangulation of a set of weighted points PW form only a subset of the set of center points of PW . Therefore the insertion of a weighted point in a regular triangulation does not necessarily imply the creation of a new vertex. If the new inserted point does not appear as a vertex in the regular triangulation, it is said to be hidden.

Hidden points are stored in special vertices called hidden vertices. A hidden point is considered as hidden by the facet of the triangulation where its point component is located : in fact, the hidden point can appear as vertex of the triangulation only if this facet is removed. Each face of a regular triangulation stores the list of hidden vertices whose points are located in the facet. When a facet is removed, points hidden by this facet are reinserted in the triangulation.

```
#include <CGAL/Regular_triangulation_2.h>
```

Parameters

The geometric traits parameter *Traits* has to be instantiated with a model of the concept *RegularTriangulationTraits_2*. The concept *RegularTriangulationTraits_2* refines the concept *TriangulationTraits_2* by adding the type *Weighted_point_2* to describe weighted points and the type *Power_test_2* to perform power tests on weighted points.

The *Tds* parameter has to be instantiated by a model of *TriangulationDataStructure_2*. The face base of a regular triangulation has to be a model of the concept *RegularTriangulationFaceBase_2*. while the vertex base class has to be a model of *RegularTriangulationVertexBase_2*. CGAL provides a default instantiation for the *Tds* parameter by the class *CGAL::Triangulation_data_structure_2 < CGAL::Regular_triangulation_vertex_base_2<Traits>, CGAL::Regular_Triangulation_face_base_2<Traits> >*.

Inherits From

Triangulation_2<Traits,Tds>

Types

```
typedef Traits::Distance      Distance;
```

```
typedef Traits::Line          Line;
```

```
typedef Traits::Ray           Ray;
```

```
typedef Traits::Bare_point     Bare_point;
```

```
typedef Traits::Weighted_point Weighted_point;
```

```
Regular_triangulation_2<Traits,Tds>:: All_vertices_iterator
```

An iterator that allows to enumerate the vertices that are not hidden.

```
Regular_triangulation_2<Traits,Tds>:: Finite_vertices_iterator
```

An iterator that allows to enumerate the finite vertices that are not hidden.

```
Regular_triangulation_2<Traits,Tds>:: Hidden_vertices_iterator;
```

An iterator that allows to enumerate the hidden vertices.

Creation

```
Regular_triangulation_2<Traits,Tds> rt( Traits gt = Traits());
```

Introduces an empty regular triangulation *rt*.

```
Regular_triangulation_2<Traits,Tds> rt( Regular_triangulation_2 rt);
```

Copy constructor.

Insertion and Removal

```
Vertex_handle rt.insert( Weighted_point p, Face_handle f=Face_handle())
```

inserts weighted point *p* in the regular triangulation. If the point *p* does not appear as a vertex of the triangulation, the returned vertex is a hidden vertex. If given the parameter *f* is used as an hint for the place to start the location process of point *p*.

```
Vertex_handle rt.insert( Weighted_point p, Locate_type lt, Face_handle loc, int li)
```

insert a weighted point *p* whose bare-point is assumed to be located in *lt,loc,li*.

```
Vertex_handle rt.push_back( Point p)      Equivalent to insert(p).
```

```
template < class InputIterator >  
int rt.insert( InputIterator first, InputIterator last)
```

inserts the weighted points in the range $[first, last)$. Returns the number of created vertices.

Precondition: The *value_type* of *first* and *last* is *Weighted_point*.

```
void rt.remove( Vertex_handle v)      removes the vertex from the triangulation.
```

Queries

```
template <class OutputItFaces, class OutputItBoundaryEdges, class OutputItHiddenVertices>  
CGAL::Triple<OutputItFaces,OutputItBoundaryEdges,OutputItHiddenVertices>
```

```
rt.get_conflicts_and_boundary_and_hidden_vertices( Weighted_point p,  
                                                    OutputItFaces fit,  
                                                    OutputItBoundaryEdges eit,  
                                                    OutputItHiddenVertices vit,
```

Face_handle start)

OutputItFaces is an output iterator with *Face_handle* as value type. *OutputItBoundaryEdges* stands for an output iterator with *Edge* as value type. *OutputItHiddenVertices* is an output iterator with *Vertex_handle* as value type. This member function outputs in the container pointed to by *fit* the faces which are in conflict with point *p* i. e. the faces whose power circles have negative power wrt. *p*. It outputs in the container pointed to by *eit* the boundary of the zone in conflict with *p*. It inserts the vertices that would be hidden by *p* into the container pointed to by *vit*. The boundary edges of the conflict zone are output in counter-clockwise order and each edge is described through its incident face which is not in conflict with *p*. The function returns in a *CGAL::Triple* the resulting output iterators.

```
template <class OutputItFaces, class OutputItBoundaryEdges>
std::pair<OutputItFaces,OutputItBoundaryEdges>
```

```
    rt.get_conflicts_and_boundary( Weighted_point p,
                                   OutputItFaces fit,
                                   OutputItBoundaryEdges eit,
                                   Face_handle start)
```

same as above except that only the faces in conflict with *p* and the boundary edges of the conflict zone are output via the corresponding output iterators. The function returns in a *std::pair* the resulting output iterators.

```
template <class OutputItFaces, class OutputItHiddenVertices>
std::pair<OutputItFaces,OutputItHiddenVertices>
```

```
    rt.get_conflicts_and_hidden_vertices( Weighted_point p,
                                           OutputItFaces fit,
                                           OutputItHiddenVertices vit,
                                           Face_handle start)
```

same as above except that only the faces in conflict with *p* and the vertices that would be hidden by *p* are output via the corresponding output iterators. The function returns in a *std::pair* the resulting output iterators.

```
template <class OutputItBoundaryEdges, class OutputItHiddenVertices>
std::pair<OutputItBoundaryEdges,OutputItHiddenVertices>
```

```
    rt.get_boundary_of_conflicts_and_hidden_vertices( Weighted_point p,
                                                       OutputItBoundaryEdges eit,
                                                       OutputItHiddenVertices vit,
```

Face_handle start)

same as above except that only the vertices that would be hidden by p and the boundary of the zone in conflict with p are output via the corresponding output iterators. The boundary edges of the conflict zone are output in counterclockwise order and each edge is described through the incident face which is not in conflict with p . The function returns in a `std::pair` the resulting output iterators.

template <class OutputItFaces>
OutputItFaces

rt.get_conflicts(Point p, OutputItFaces fit, Face_handle start)

same as above except that only the faces in conflict with p are output. The function returns the resulting output iterator.

template <class OutputItBoundaryEdges>
OutputItBoundaryEdges

rt.get_boundary_of_conflicts(Point p, OutputItBoundaryEdges eit, Face_handle start)

same as above except that only the boundary edges of the conflict zone are output in counterclockwise order where each edge is described through the incident face which is not in conflict with p . The function returns the resulting output iterator.

template <class OutputItHiddenVertices>
OutputItHiddenVertices

rt.get_hidden_vertices(Point p, OutputItHiddenVertices vit, Face_handle start)

same as above except that only the vertices that would be hidden by p are output. The function returns the resulting output iterator.

Vertex_handle *rt.nearest_power_vertex(Bare_point p)*

Returns the vertex of the triangulation which is nearest to p with respect to the power distance. This means that the power of the query point p with respect to the weighted point in the nearest vertex is smaller than the power of p with respect to the weighted point in any other vertex. Ties are broken arbitrarily. The default constructed handle is returned if the triangulation is empty.

Access functions

<i>int</i>	<i>rt.number_of_vertices()</i>	returns the number of finite vertices that are not hidden.
<i>int</i>	<i>rt.number_of_hidden_vertices()</i>	
		returns the number of hidden vertices.
<i>Hidden_vertices_iterator</i>	<i>rt.hidden_vertices_begin()</i>	
		starts at an arbitrary hidden vertex.
<i>Hidden_vertices_iterator</i>	<i>rt.hidden_vertices_end()</i>	past the end iterator for the sequence of hidden vertices.
<i>Finite_vertices_iterator</i>	<i>rt.finite_vertices_begin()</i>	starts at an arbitrary unhidden finite vertex
<i>Finite_vertices_iterator</i>	<i>rt.finite_vertices_end()</i>	Past-the-end iterator
<i>All_vertices_iterator</i>	<i>rt.all_vertices_end()</i>	starts at an arbitrary unhidden vertex.
<i>All_vertices_iterator</i>	<i>rt.all_vertices_begin()</i>	past the end iterator.

Dual power diagram

The following member functions provide the elements of the dual power diagram.

<i>Point</i>	<i>rt.weighted_circumcenter(Face_handle f)</i>	
		returns the center of the circle orthogonal to the three weighted points corresponding to the vertices of face <i>f</i> . <i>Precondition:</i> <i>f</i> is not infinite
<i>Point</i>	<i>rt.dual(Face_handle f)</i>	same as <i>weighted_circumcenter</i>
<i>Object</i>	<i>rt.dual(Edge e)</i>	If both incident faces are finite, returns a segment whose endpoints are the duals of each incident face. If only one incident face is finite, returns a ray whose endpoint is the dual of the finite incident face and supported by the line which is the bisector of the edge's endpoints. If both incident faces are infinite, returns the line which is the bisector of the edge's endpoints otherwise.
<i>Object</i>	<i>rt.dual(Edge_circulator ec)</i>	
		Idem
<i>Object</i>	<i>rt.dual(Edge_iterator ei)</i>	
		Idem
<i>template < class Stream></i> <i>Stream&</i>	<i>rt.draw_dual(Stream & ps)</i>	
		output the dual power diagram to stream <i>ps</i> .

Predicates

Oriented_side *rt.power_test(Face_handle f, Weighted_point p)*

Returns the power test of *p* with respect to the power circle associated with *f*

————— *advanced* —————

Miscellaneous

bool *rt.is_valid(bool verbose = false, int level = 0)*

Tests the validity of the triangulation as a *Triangulation_2* and additionally test the regularity of the triangulation. This method is useful to debug regular triangulation algorithms implemented by the user.

————— *advanced* —————

See Also

CGAL::Triangulation_2<Traits,Tds>,
TriangulationDataStructure_2,
RegularTriangulationTraits_2
RegularTriangulationFaceBase_2
RegularTriangulationVertexBase_2
CGAL::Regular_triangulation_face_base_2<Traits>
CGAL::Regular_triangulation_vertex_base_2<Traits>

CGAL::Regular_triangulation_euclidean_traits_2<K,Weight>

Definition

Regular_triangulation_euclidean_traits_2<K,Weight> is a model for the concept *RegularTriangulationTraits_2*. This traits class is templated by a kernel class *K* and a weight type *Weight*. This class inherits from *K* and uses a *Weighted_point* type derived from the type *K::Point_2*.

Note that this template class is specialized for *CGAL::Exact_predicates_inexact_constructions_kernel*, so that it is as if *Regular_triangulation_filtered_traits_2* was used, i.e. you get filtered predicates automatically.

```
#include <CGAL/Regular_triangulation_euclidean_traits_2.h>
```

Is Model for the Concepts

RegularTriangulationTraits_2

Inherits From

K

See Also

RegularTriangulationTraits_2

CGAL::Regular_triangulation_filtered_traits_2

CGAL::Regular_triangulation_2

CGAL::Regular_triangulation_filtered_traits_2<FK>

Definition

The class *Regular_triangulation_filtered_traits_2<FK>* is designed as a traits class for the class *Regular_triangulation_2<RegularTriangulationTraits_2,TriangulationDataStructure_2>*. Its difference with *Regular_triangulation_euclidean_traits_2* is that it provides filtered predicates which are meant to be fast and exact.

The first argument *FK* must be a model of the *Kernel* concept, and it is also restricted to be an instance of the *Filtered_kernel* template.

```
#include <CGAL/Regular_triangulation_filtered_traits_2.h>
```

Is Model for the Concepts

RegularTriangulationTraits_2

Inherits From

Regular_triangulation_euclidean_traits_2<FK>

See Also

CGAL::Regular_triangulation_euclidean_traits_2.

CGAL::Regular_triangulation_face_base_2<Traits,Fb>

Definition

The class *Regular_triangulation_face_base_2<Traits,Fb>* is a model of the concept *RegularTriangulationFaceBase_2*. It is the default face base class of regular triangulations.

```
#include <CGAL/Regular_triangulation_face_base_2.h>
```

Parameters

The template parameters *Traits* has to be a model of *RegularTriangulationTraits_2*.

The template parameter *Fb* has to be a model of *TriangulationFaceBase_2*. By default, this parameter is instantiated by *CGAL::Triangulation_face_base_2<Traits>*.

Is Model for the Concepts

RegularTriangulationFaceBase_2

Inherits From

Fb

See Also

RegularTriangulationFaceBase_2

RegularTriangulationTraits_2

CGAL::Regular_triangulation_2<Traits,Tds>

CGAL::Regular_triangulation_vertex_base_2<Traits>

CGAL::Regular_triangulation_vertex_base_2<Traits,Vb>

Definition

The class *Regular_triangulation_vertex_base_2<Traits,Vb>* is a model of the concept *RegularTriangulationVertexBase_2*. It is the default vertex base class of regular triangulations.

```
#include <CGAL/Regular_triangulation_vertex_base_2.h>
```

Parameters

The template parameters *Traits* has to be a model of *RegularTriangulationTraits_2*.

The template parameters *Vb* has to be a model of the concept *TriangulationVertexBase_2* and is by default instantiated by *CGAL::Triangulation_vertex_base_2<Traits>*.

Is Model for the Concepts

RegularTriangulationVertexBase_2

Inherits From

Triangulation_vertex_base_2<Traits,Tds>

See Also

CGA::Triangulation_vertex_base_2<Traits,Vb>

CGAL::Regular_triangulation_2<Traits,Tds>

CGAL::Regular_triangulation_face_base_2<Traits>

TriangulationFaceBase_2

Definition

The concept `TriangulationFaceBase_2` describes the requirements for the base face class of a triangulation data structure that is itself plugged into a basic triangulation or a Delaunay triangulation.

This concept refines the concept *TriangulationDSFaceBase_2* and could add geometric information. In fact, currently the triangulations of CGAL do not store any geometric information in the faces and, thus this concept is just equal to *TriangulationDSFaceBase_2* and only provided for symmetry with the vertex case.

Refines

TriangulationDSFaceBase_2

Has Models

CGAL::Triangulation_face_base_2<Traits>

See Also

TriangulationVertexBase_2

CGAL::Triangulation_face_base_2<Traits>

CGAL::Triangulation_2<Traits,Tds>

CGAL::Delaunay_triangulation_2<Traits,Tds>

TriangulationHierarchyVertexBase_2

Definition

The vertex of a triangulation included in a triangulation hierarchy has to provide some pointers to the corresponding vertices in the triangulations of the next and preceeding levels. Therefore, the concept *TriangulationHierarchyVertexBase_2* refines the concept *TriangulationVertexBase_2*, adding handles to the corresponding vertices in the next and previous level triangulations.

Refines

TriangulationVertexBase_2

Operations

<i>Vertex_handle</i>	<i>v.up()</i>	returns the corresponding vertex (if any) of the next level triangulation;
<i>Vertex_handle</i>	<i>v.down()</i>	returns the corresponding vertex of the previous level triangulation;
<i>void</i>	<i>v.set_up(Vertex_handle u)</i>	sets the handle pointing to to the corresponding vertex of the next level triangulation;
<i>void</i>	<i>v.set_down(Vertex_handle d)</i>	sets the handle pointing to the corresponding vertex of the previous level triangulation;

Has Models

CGAL::Triangulation_hierarchy_vertex_base_2<Vb>

See Also

Triangulation_hierarchy_2<Tr>

TriangulationTraits_2

Definition

The concept `TriangulationTraits_2` describes the set of requirements to be fulfilled by any class used to instantiate the first template parameter of the class `Triangulation_2<Traits,Tds>`. This concept provides the types of the geometric primitives used in the triangulation and some function object types for the required predicates on those primitives.

Types

<code>TriangulationTraits_2:: Point_2</code>	The point type.
<code>TriangulationTraits_2:: Segment_2</code>	The segment type.
<code>TriangulationTraits_2:: Triangle_2</code>	The triangle type.
<code>TriangulationTraits_2:: Construct_segment_2</code>	A constructor object for <code>Segment_2</code> . Provides : <code>Segment_2 operator()(Point_2 p,Point_2 q)</code> , which constructs a segment from two points.
<code>TriangulationTraits_2:: Construct_triangle_2</code>	A constructor object for <code>Triangle_2</code> . Provides : <code>Triangle_2 operator()(Point_2 p,Point_2 q,Point_2 r)</code> , which constructs a triangle from three points.
<code>TriangulationTraits_2:: Compare_x_2</code>	Predicate object. Provides the operator : <code>Comparison_result operator()(Point p, Point q)</code> which returns <code>SMALLER</code> , <code>EQUAL</code> or <code>LARGER</code> according to the <i>x</i> -ordering of points <i>p</i> and <i>q</i> .
<code>TriangulationTraits_2:: Compare_y_2</code>	Predicate object. Provides the operator : <code>Comparison_result operator()(Point p, Point q)</code> which returns (<code>SMALLER</code> , <code>EQUAL</code> or <code>LARGER</code>) according to the <i>y</i> -ordering of points <i>p</i> and <i>q</i> .
<code>TriangulationTraits_2:: Orientation_2</code>	Predicate object. Provides the operator : <code>Orientation operator()(Point p, Point q, Point r)</code> which returns <code>LEFT_TURN</code> , <code>RIGHT_TURN</code> or <code>COLLINEAR</code> depending on <i>r</i> being, with respect to the oriented line <i>pq</i> , on the left side , on the right side or on the line.
<code>TriangulationTraits_2:: Side_of_oriented_circle_2</code>	Predicate object. Must provide the operator <code>Oriented_side operator()(Point p, Point q, Point r, Point s)</code> which takes four points <i>p,q,r,s</i> as arguments and returns <code>ON_POSITIVE_SIDE</code> , <code>ON_NEGATIVE_SIDE</code> or <code>ON_ORIENTED_BOUNDARY</code> according to the position of points <i>s</i> with respect to the oriented circle through through <i>p,q</i> and <i>r</i> . This type is required only if the function <code>side_of_oriented_circle(Face_handle f, Point p)</code> is called.

TriangulationTraits_2::Construct_circumcenter_2

Constructor object. Provides the operator :
Point operator()(Point p, Point q, Point r)
which returns the circumcenter of the three points *p*, *q*
and *r*. This type is required only if the function *Point*
circumcenter(Face_handle f) is called.

Creation

Only a default constructor, copy constructor and an assignment operator are required. Note that further constructors can be provided.
TriangulationTraits_2 traits; default constructor.
TriangulationTraits_2 traits(gtr); Copy constructor

TriangulationTraits_2 traits = gtr Assignment operator.

Predicate functions

The following functions give access to the predicate and constructor objects.

Construct_segment_2 traits.construct_segment_2_object()

Construct_triangle_2 traits.construct_triangle_2_object()

Comparison_x_2 traits.compare_x_2_object()

Comparison_y_2 traits.compare_y_2_object()

Orientation_2 traits.orientation_2_object()

Side_of_oriented_circle_2

traits.side_of_oriented_circle_2_object()

Required only if *side_of_oriented_circle* is called.

Construct_circumcenter_2

traits.construct_circumcenter_2_object()

Required only if *circumcenter* is called.

Has Models

All the CGAL Kernels

CGAL::Triangulation_euclidean_traits_2<K>

CGAL::Triangulation_euclidean_traits_xy_3<K>

CGAL::Triangulation_euclidean_traits_yz_3<K>

CGAL::Triangulation_euclidean_traits_zx_3<K>

See Also

CGAL::Triangulation_2<Traits,Tds>

TriangulationVertexBase_2

Definition

The concept `TriangulationVertexBase_2` describes the requirements for the vertex base class of a triangulation data structure to be plugged in a basic, Delaunay or constrained triangulations.

The concept `TriangulationVertexBase_2` refines the concept `TriangulationDSVertexBase_2` adding geometric information : the vertex base of a triangulation stores a point.

Refines

`TriangulationDSVertexBase_2`

Types

`TriangulationVertexBase_2::Point`

Must be the same as the point type `TriangulationTraits_2::Point_2` defined by the geometric traits class of the triangulation.

Creation

`TriangulationVertexBase_2 v(Point p);` constructs a vertex embedded in point p .

`TriangulationVertexBase_2 v(Point p, Face_handle f);`
constructs a vertex embedded in point p and pointing on face f .

Access Functions

`Point v.point()` returns the point.

Setting

`void v.set_point(Point p)`
sets the point.

I/O

`istream& istream& is >> & v`

Inputs the non-combinatorial information given by the vertex: the point and other possible information.

ostream&

ostream& os << v

Outputs the non combinatorial operation given by the vertex: the point and other possible information.

Has Models

CGAL::TriangulationVertexBase_2<Traits>.

See Also

TriangulationDataStructure_2

TriangulationDataStructure_2::Vertex

CGAL::Triangulation_vertex_base_2<Traits>

CGAL::Triangulation_2<Traits,Tds>

Definition

The class *Triangulation_2<Traits,Tds>* is the basic class designed to handle triangulations of set of points *A* in the plane.

Such a triangulation has vertices at the points of *A* and its domain covers the convex hull of *A*. It can be viewed as a planar partition of the plane whoses bounded faces are triangular and cover the convex hull of *A*. The single unbounded face of this partition is the complementary of the convex hull of *A*.

In many applications, it is convenient to deal only with triangular faces. Therefore, we add to the triangulation a fictitious vertex, called the *infinite vertex* and we make each convex hull edge incident to an *infinite* face having as third vertex the *infinite vertex*. In that way, each edge is incident to exactly two faces and special cases at the boundary of the convex hull are simpler to deal with.

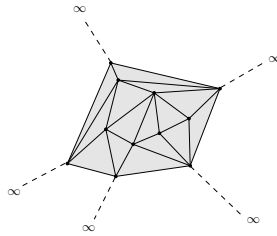


Figure 20.9: The infinite vertex.

The class *Triangulation_2<Traits,Tds>* implements this point of view and therefore considers the triangulation of the set of points as a set of triangular, finite and infinite faces. Although it is convenient to draw a triangulation as in figure 20.9, note that the *infinite vertex* has no significant coordinates and that no geometric predicate can be applied on it or on an infinite face.

A triangulation is a collection of vertices and faces that are linked together through incidence and adjacency relations. Each face give access to its three incident vertices and to its three adjacent faces. Each vertex give access to one of its incident faces.

The three vertices of a face are indexed with 0, 1 and 2 in counterclockwise order. The neighbor of a face are also indexed with 0,1,2 in such a way that the neighbor indexed by *i* is opposite to the vertex with the same index.

The triangulation class offer two functions *int cw(int i)* and *int ccw(int i)* which given the index of a vertex in a face compute the index of the next vertex of the same face in clockwise or counterclockwise order. Thus, for example the neighbor *neighbor(cw(i))* is the neighbor of *f* which is next to *neighbor(i)* turning clockwise around *f*. The face *neighbor(cw(i))* is also the first face encountered after *f* when turning clockwise around vertex *i* of *f* (see Figure 20.10).

```
#include <CGAL/Triangulation_2.h>
```

Parameters

The class *Triangulation_2<Traits,Tds>* has two template parameters. The first one *Traits* is the geometric traits, it is to be instantiated by a model of the concept *TriangulationTraits_2*.

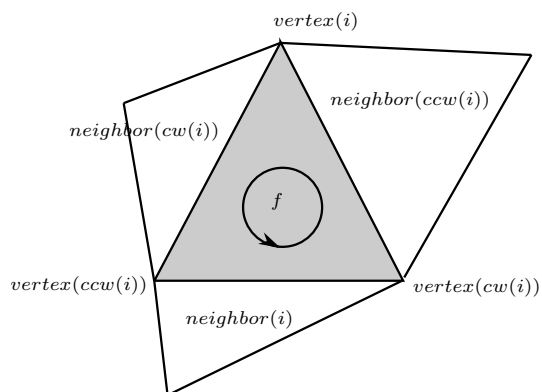


Figure 20.10: Vertices and neighbors.

The second parameter is the triangulation data structure, it has to be instantiated by a model of the concept *TriangulationDataStructure_2*. By default, the triangulation data structure is instantiated by `CGAL::Triangulation_data_structure_2 < CGAL::Triangulation_vertex_base_2<Gt>, CGAL::Triangulation_face_base_2<Gt> >>`.

Inherits From

Triangulation_cw_ccw_2 This class provides the functions *cw(i)* et *ccw(i)*.

Types

<code>typedef Traits</code>	<code>Geom_traits;</code>	the traits class.
<code>typedef Tds</code>	<code>Triangulation_data_structure;</code>	the triangulation data structure type.
<code>typedef Traits::Point_2t</code>	<code>Point;</code>	the point type
<code>typedef Traits::Segment_2</code>	<code>Segment;</code>	the segment type
<code>typedef Traits::Triangle_2</code>	<code>Triangle;</code>	the triangle type
<code>typedef Tds::Vertex</code>	<code>Vertex;</code>	the vertex type.
<code>typedef Tds::Face</code>	<code>Face;</code>	the face type.
<code>typedef Tds::Edge</code>	<code>Edge;</code>	the edge type.
<code>typedef Tds::size_type</code>	<code>size_type;</code>	Size type (an unsigned integral type)
<code>typedef Tds::difference_type</code>	<code>difference_type;</code>	Difference type (a signed integral type)

The vertices and faces of the triangulations are accessed through *handles*, *iterators* and *circulators*. The handles are models of the concept *Handle* which basically offers the two dereference operators `*` and `->`. The iterators and circulators are all bidirectional and non mutable. The circulators and iterators are convertible to handles with the same value type, so that whenever a handle appear in the parameter list of a function, an appropriate iterator or circulator can be passed as well.

The edges of the triangulation can also be visited through iterators and circulators, the edge circulators and iterators are also bidirectional and non mutable.

In the following, we called *infinite* any face or edge incident to the infinite vertex and the infinite vertex itself. Any other feature (face, edge or vertex) of the triangulation is said to be *finite*. Some iterators (the *All* iterators

) allows to visit finite or infinite feature while others (the *Finite* iterators) visit only finite features. Circulators visit infinite features as well as finite ones.

<code>typedef Tds::Vertex_handle</code>	<code>Vertex_handle;</code>	handle to a vertex
<code>typedef Tds::Face_handle</code>	<code>Face_handle;</code>	handle to a face
<code>typedef Tds::Face_iterator</code>	<code>All_faces_iterator;</code>	iterator over all faces.
<code>typedef Tds::Edge_iterator</code>	<code>All_edges_iterator;</code>	iterator over all edges
<code>typedef Tds::Vertex_iterator</code>	<code>All_vertices_iterator;</code>	iterator over all vertices
<code>Triangulation_2<Traits,Tds>:: Finite_faces_iterator</code>		iterator over finite faces.
<code>Triangulation_2<Traits,Tds>:: Finite_edges_iterator</code>		iterator over finite edges.
<code>Triangulation_2<Traits,Tds>:: Finite_vertices_iterator</code>		iterator over finite vertices.
<code>Triangulation_2<Traits,Tds>:: Point_iterator</code>		iterator over the points corresponding the finite vertices of the triangulation.
<code>Triangulation_2<Traits,Tds>:: Line_face_circulator</code>		circulator over all faces intersected by a line.
<code>Triangulation_2<Traits,Tds>:: Face_circulator</code>		circulator over all faces incident to a given vertex.
<code>Triangulation_2<Traits,Tds>:: Edge_circulator</code>		circulator over all edges incident to a given vertex.
<code>Triangulation_2<Traits,Tds>:: Vertex_circulator</code>		circulator over all vertices incident to a given vertex.

The triangulation class also defines the following enum type to specify which case occurs when locating a point in the triangulation.

```
enum Locate_type { VERTEX=0, EDGE, FACE, OUTSIDE_CONVEX_HULL, OUTSIDE_AFFINE_HULL};
```

The locate type is *OUTSIDE_CONVEX_HULL* when the point is outside the convex hull but in the affine hull of the current triangulation.

The locate type is *OUTSIDE_AFFINE_HULL* when the point is outside the affine hull of the current triangulation.

Creation

```
Triangulation_2<Traits,Tds> t;           default constructor.
Triangulation_2<Traits,Tds> t( Traits gt = Traits());
```

Introduces an empty triangulation *t*.

Triangulation_2<*Traits*,*Tds*> *t*(*Triangulation_2 tr*);

Copy constructor. All the vertices and faces are duplicated. After the copy, *t* and *tr* refer to different triangulations : if *tr* is modified, *t* is not.

Triangulation_2 *t = tr*

Assignment. All the vertices and faces are duplicated. After the assignment, *t* and *tr* refer to different triangulations : if *tr* is modified, *t* is not.

void *t.swap(Triangulation_2& tr)*

The triangulations *tr* and *t* are swapped. *t.swap(tr)* should be preferred to *t = tr* or to *t(tr)* if *tr* is deleted after that.

void *t.clear()*

Deletes all faces and finite vertices resulting in an empty triangulation.

Access Functions

Geom_traits *t.geom_traits()*
TriangulationDataStructure_2

Returns a const reference to the triangulation traits object.

t.tds()

Returns a const reference to the triangulation data structure.

————— *advanced* —————

Non const access

The responsibility of keeping a valid triangulation belongs to the user when using advanced operations allowing a direct manipulation of the *tds*.

TriangulationDataStructure_2&

t.tds()

Returns a reference to the triangulation data structure.

This method is mainly a help for users implementing their own triangulation algorithms.

————— *advanced* —————

int *t.dimension()*

Returns the dimension of the convex hull.

size_type *t.number_of_vertices()*

Returns the number of finite vertices.

size_type *t.number_of_faces()*

Returns the number of finite faces.

Face_handle *t.infinite_face()*

a face incident to the *infinite_vertex*.

Vertex_handle *t.infinite_vertex()*

the *infinite_vertex*.

Vertex_handle *t.finite_vertex()*

a vertex distinct from the *infinite_vertex*.

Predicates

The class *Triangulation_2*<*Traits*,*Tds*> provides methods to test the finite or infinite character of any feature, and also methods to test the presence in the triangulation of a particular feature (edge or face).

bool t.is_infinite(Vertex_handle v) *true* iff *v* is the *infinite_vertex*.
bool t.is_infinite(Face_handle f) *true* iff face *f* is infinite.
bool t.is_infinite(Face_handle f, int i) *true* iff edge (*f,i*) is infinite.
bool t.is_infinite(Edge e) *true* iff edge *e* is infinite.
bool t.is_infinite(Edge_circulator ec) *true* iff edge **ec* is infinite.
bool t.is_infinite(Edge_iterator ei) *true* iff edge **ei* is infinite.

bool t.is_edge(Vertex_handle va, Vertex_handle vb)
true if there is an edge having *va* and *vb* as vertices.

bool t.is_edge(Vertex_handle va, Vertex_handle vb, Face_handle& fr, int & i)
as above. In addition, if *true* is returned, the edge with vertices *va* and *vb* is the edge *e=(fr,i)* where *fr* is a handle to the face incident to *e* and on the right side of *e* oriented from *va* to *vb*.

bool t.includes_edge(Vertex_handle va, Vertex_handle & vb, Face_handle& fr, int & i)
true if the line segment from *va* to *vb* includes an edge *e* incident to *va*. If *true*, *vb* becomes the other vertex of *e*, *e* is the edge (*fr,i*) where *fr* is a handle to the face incident to *e* and on the right side *e* oriented from *va* to *vb*.

bool t.is_face(Vertex_handle v1, Vertex_handle v2, Vertex_handle v3)
true if there is a face having *v1*, *v2* and *v3* as vertices.

bool t.is_face(Vertex_handle v1, Vertex_handle v2, Vertex_handle v3, Face_handle &fr)
as above. In addition, if *true* is returned, *fr* is a handle to the face with *v1*, *v2* and *v3* as vertices.

Queries

The class *Triangulation_2<Traits,Tds>* provides methods to locate a given point with respect to a triangulation. It also provides methods to locate a point with respect to a given finite face of the triangulation.

Face_handle t.locate(Point query, Face_handle f = Face_handle())

If the point *query* lies inside the convex hull of the points, a face that contains the query in its interior or on its boundary is returned.

If the point *query* lies outside the convex hull of the triangulation but in the affine hull, the returned face is an infinite face which is a proof of the point's location :

- for a two dimensional triangulation, it is a face (∞, p, q) such that *query* lies to the left of the oriented line *pq* (the rest of the triangulation lying to the right of this line).

- for a degenerate one dimensional triangulation it is the (degenerate one dimensional) face $(\infty, p, NULL)$ such that *query* and the triangulation lie on either side of *p*.

If the point *query* lies outside the affine hull, the returned *Face_handle* is *NULL*.

The optional *Face_handle* argument, if provided, is used as a hint of where the locate process has to start its search.

Face_handle *t.locate(Point query, Locate_type& lt, int& li, Face_handle h =Face_handle())*

Same as above. Additionally, the parameters *lt* and *li* describe where the query point is located. The variable *lt* is set to the locate type of the query. If *lt*==*VERTEX* the variable *li* is set to the index of the vertex, and if *lt*==*EDGE* *li* is set to the index of the vertex opposite to the edge. Be careful that *li* has no meaning when the query type is *FACE*, *OUTSIDE_CONVEX_HULL*, or *OUTSIDE_AFFINE_HULL* or when the triangulation is 0-dimensional.

Oriented_side *t.oriented_side(Face_handle f, Point p)*

Returns on which side of the oriented boundary of *f* lies the point *p*.
Precondition: *f* is finite.

Oriented_side *t.side_of_oriented_circle(Face_handle f, Point p)*

Returns on which side of the circumcircle of face *f* lies the point *p*. The circle is assumed to be counterclockwisely oriented, so its positive side correspond to its bounded side. This predicate is available only if the corresponding predicates on points is provided in the geometric traits class.

Modifiers

The following operations are guaranteed to lead to a valid triangulation when they are applied on a valid triangulation.

void *t.flip(Face_handle f, int i)*

Exchanges the edge incident to *f* and *f->neighbor(i)* with the other diagonal of the quadrilateral formed by *f* and *f->neighbor(i)*.
Precondition: The faces *f* and *f->neighbor(i)* are finite faces and their union form a convex quadrilateral.

Vertex_handle *t.insert(Point p, Face_handle f = Face_handle())*

Inserts point *p* in the triangulation and returns the corresponding vertex.
 If point *p* coincides with an already existing vertex, this vertex is returned and the triangulation remains unchanged.
 If point *p* is on an edge, the two incident faces are split in two.
 If point *p* is strictly inside a face of the triangulation, the face is split in three.
 If point *p* is strictly outside the convex hull, *p* is linked to all visible points on the convex hull to form the new triangulation.
 At last, if *p* is outside the affine hull (in case of degenerate 1-dimensional or 0-dimensional triangulations), *p* is linked all the other vertices to form a triangulation whose dimension is increased by one. The last argument *f* is an indication to the underlying locate algorithm of where to start.

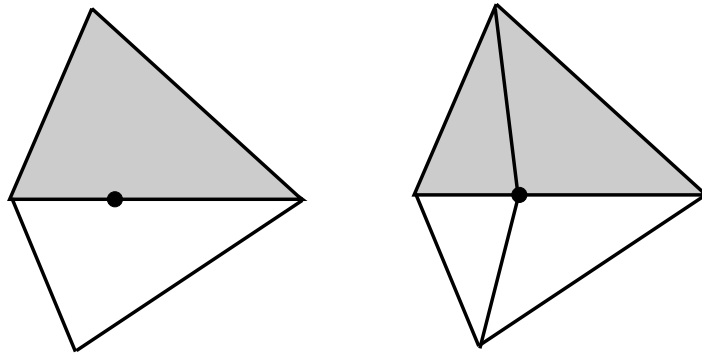


Figure 20.11: Insertion of a point on an edge.

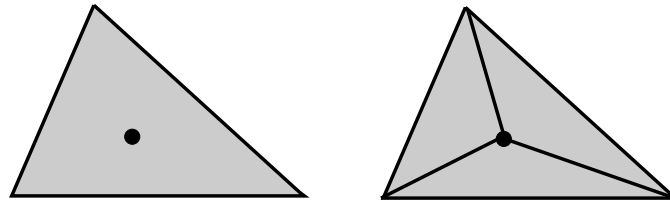


Figure 20.12: Insertion in a face.

Vertex_handle *t.insert(Point p, Locate_type lt, Face_handle loc, int li)*

Same as above except that the location of the point *p* to be inserted is assumed to be given by (lt, loc, i) (see the description of the *locate* method above.)

Vertex_handle *t.push_back(Point p)*

Equivalent to *insert(p)*.

template < class InputIterator >
int *t.insert(InputIterator first, InputIterator last)*

Inserts the points in the range $[first, last)$. Returns the number of inserted points.

Precondition: The *value_type* of *InputIterator* is *Point*.

void *t.remove(Vertex_handle v)*

Removes the vertex from the triangulation. The created hole is retriangulated.

Precondition: Vertex *v* must be finite.

— *advanced* —

The following member functions offer more specialized versions of the insertion or removal operations to be used when one knows to be in the corresponding case.

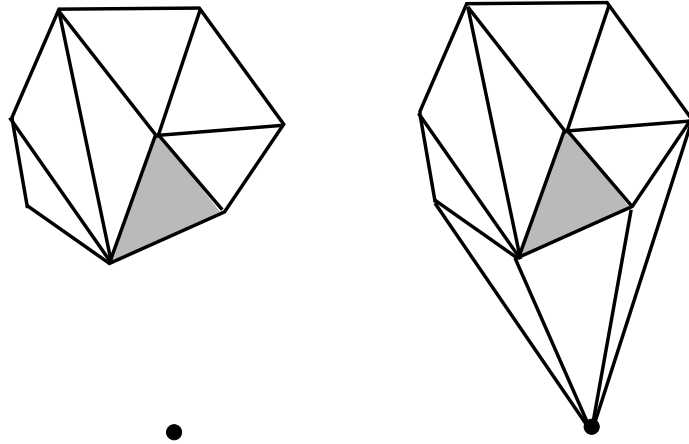


Figure 20.13: Insertion outside the convex hull.

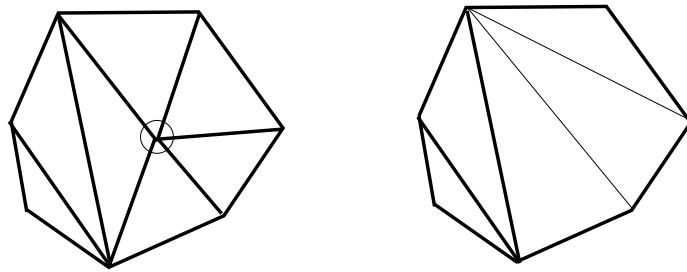


Figure 20.14: Removal

Vertex_handle *t.insert_first(Point p)*

Inserts the first finite vertex .

Vertex_handle *t.insert_second(Point p)*

Inserts the second finite vertex .

Vertex_handle *t.insert_in_face(Point p, Face_handle f)*

Inserts vertex v in face f . Face f is modified, two new faces are created.

Precondition: The point in vertex v lies inside face f .

Vertex_handle *t.insert_in_edge(Point p, Face_handle f, int i)*

Inserts vertex v in edge i of f .

Precondition: The point in vertex v lies on the edge opposite to the vertex i of face f .

Vertex_handle *t.insert_outside_convex_hull(Point p, Face_handle f)*

Inserts a point which is outside the convex hull but in the affine hull.

Precondition: The handle f points to a face which is a proof of the location of p , see the description of the *locate* method above.

Vertex_handle *t.insert_outside_affine_hull(Point p)*

Inserts a point which is outside the affine hull.

void *t.remove_degree_3(Vertex_handle v)*

Removes a vertex of degree three. Two of the incident faces are destroyed, the third one is modified.

Precondition: Vertex v is a finite vertex with degree three.

void *t.remove_second(Vertex_handle v)*

Removes the before last finite vertex.

void *t.remove_first(Vertex_handle v)*

Removes the last finite vertex.

The following fonctions are mainly intended to be used in conjunction with the *find_conflicts()* member fonctions of Delaunay and constrained Delaunay triangulations to perform insertions.

```
template<class EdgeIt>
Vertex_handle t.star_hole( Point p, EdgeIt edge_begin, EdgeIt edge_end)
```

creates a new vertex v and use it to star the hole whose boundary is described by the sequence of edges $[edge_begin, edge_end]$. Returns a handle to the new vertex.

```
template<class EdgeIt, class FaceIt>
Vertex_handle t.star_hole( Point p,
                          EdgeIt edge_begin,
                          EdgeIt edge_end,
                          FaceIt face_begin,
                          FaceIt face_end)
```

same as above, except that the algorithm first recycles faces in the sequence $[face_begin, face_end]$ and create new ones only when the sequence is exhausted.

_____ advanced _____

Traversal of the Triangulation

A triangulation can be seen as a container of faces and vertices. Therefore the triangulation provides several iterators and circulators that allow to traverse it (completely or partially).

Face, Edge and Vertex Iterators

The following iterators allow respectively to visit finite faces, finite edges and finite vertices of the triangulation. These iterators are non mutable, bidirectional and their value types are respectively *Face*, *Edge* and *Vertex*. They are all invalidated by any change in the triangulation.

<i>Finite_vertices_iterator</i>	<i>t.finite_vertices_begin()</i>	Starts at an arbitrary finite vertex
<i>Finite_vertices_iterator</i>	<i>t.finite_vertices_end()</i>	Past-the-end iterator
<i>Finite_edges_iterator</i>	<i>t.finite_edges_begin()</i>	Starts at an arbitrary finite edge
<i>Finite_edges_iterator</i>	<i>t.finite_edges_end()</i>	Past-the-end iterator
<i>Finite_faces_iterator</i>	<i>t.finite_faces_begin()</i>	Starts at an arbitrary finite face
<i>Finite_faces_iterator</i>	<i>t.finite_faces_end()</i>	Past-the-end iterator
<i>Point_iterator</i>	<i>t.points_begin()</i>	
<i>Point_iterator</i>	<i>t.points_end()</i>	Past-the-end iterator

The following iterators allow respectively to visit all (finite or infinite) faces, edges and vertices of the triangulation. These iterators are non mutable, bidirectional and their value types are respectively *Face*, *Edge* and *Vertex*. They are all invalidated by any change in the triangulation.

<i>All_vertices_iterator</i>	<i>t.all_vertices_begin()</i>	Starts at an arbitrary vertex
<i>All_vertices_iterator</i>	<i>t.all_vertices_end()</i>	Past-the-end iterator
<i>All_edges_iterator</i>	<i>t.all_edges_begin()</i>	Starts at an arbitrary edge

<i>All_edges_iterator</i>	<i>t.all_edges_end()</i>	Past-the-end iterator
<i>All_faces_iterator</i>	<i>t.all_faces_begin()</i>	Starts at an arbitrary face
<i>All_faces_iterator</i>	<i>t.all_faces_end()</i>	Past-the-end iterator

Line Face Circulator

The triangulation defines a circulator that allows to visit all faces that are intersected by a line. A face f is considered as being intersected by the oriented line l if either:

- f is a finite face whose interior intersects l , or
- f is a finite face with an edge collinear with l and lies to the left of l , or
- f is an infinite face incident to a convex hull edge whose interior is intersected by l , or
- f is an infinite face incident to a convex hull vertex lying on l and the finite edge of f lies to the left of l .

The circulator has a singular value if the line l intersect no finite face of the triangulation. This circulator is non-mutable and bidirectional. Its value type is *Face*.

Line_face_circulator *t.line_walk(Point p, Point q, Face_handle f = Face_handle())*

This function returns a circulator that allows to visit the faces intersected by the line pq . If there is no such face the circulator has a singular value.

The starting point of the circulator is the face f , or the first finite face traversed by l , if f is omitted.

The circulator wraps around the *infinite_vertex* : after the last traversed finite face, it steps through the infinite face adjacent to this face then through the infinite face adjacent to the first traversed finite face then through the first finite traversed face again.

Precondition: Points p and q must be different points.

Precondition: If $f \neq \text{NULL}$, it must point to a finite face and the point p must be inside or on the boundary of f .

Figure 20.15 illustrates which finite faces are enumerated. Lines l_1 and l_2 have no face to their left. Lines l_3 and l_4 have faces to their left. Note that the finite faces that are only vertex incident to lines l_3 and l_4 are not enumerated.

A line face circulator is invalidated if the face the circulator refers to is changed.

Face, Edge and Vertex Circulators

The triangulation also provides circulators that allows to visit respectively all faces or edges incident to a given vertex or all vertices adjacent to a given vertex. These circulators are non-mutable and bidirectional. The *operator++* moves the circulator counterclockwise around the vertex while the *operator--* moves clockwise. A face circulator is invalidated by any modification of the face pointed to. An edge or a vertex circulator are invalidated by any modification of one of the two faces incident to the edge pointed to.

Face_circulator *t.incident_faces(Vertex_handle v)*

Starts at an arbitrary face incident to v .

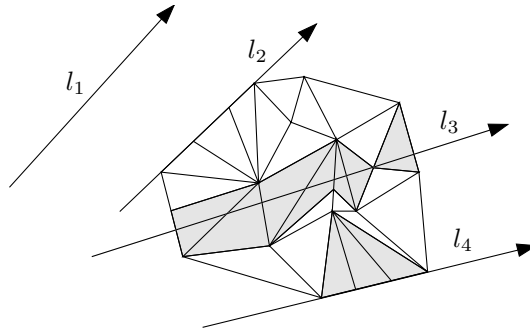


Figure 20.15: The line face circulator.

<i>Face_circulator</i>	<i>t.incident_faces(Vertex_handle v, Face_handle f)</i>	Starts at face <i>f</i> . <i>Precondition:</i> Face <i>f</i> is incident to vertex <i>v</i> .
<i>Edge_circulator</i>	<i>t.incident_edges(Vertex_handle v)</i>	Starts at an arbitrary edge incident to <i>v</i> .
<i>Edge_circulator</i>	<i>t.incident_edges(Vertex_handle v, Face_handle f)</i>	Starts at the first edge of <i>f</i> incident to <i>v</i> , in counterclockwise order around <i>v</i> . <i>Precondition:</i> Face <i>f</i> is incident to vertex <i>v</i> .
<i>Vertex_circulator</i>	<i>t.incident_vertices(Vertex_handle v)</i>	Starts at an arbitrary vertex incident to <i>v</i> .
<i>Vertex_circulator</i>	<i>t.incident_vertices(Vertex_handle v, Face_handle f)</i>	Starts at the first vertex of <i>f</i> adjacent to <i>v</i> in counterclockwise order around <i>v</i> . <i>Precondition:</i> Face <i>f</i> is incident to vertex <i>v</i> .

Traversal of the Convex Hull

Applied on the *infinite_vertex* the above functions allow to visit the vertices on the convex hull and the infinite edges and faces. Note that a counterclockwise traversal of the vertices adjacent to the *infinite_vertex* is a clockwise traversal of the convex hull.

<i>Face_circulator</i>	<i>t.incident_faces(t.infinite_vertex())</i>
<i>Face_circulator</i>	<i>t.incident_faces(t.infinite_vertex(), Face_handle f)</i>
<i>Edge_circulator</i>	<i>t.incident_edges(t.infinite_vertex())</i>
<i>Edge_circulator</i>	<i>t.incident_edges(t.infinite_vertex(), Face_handle f)</i>
<i>Vertex_circulator</i>	<i>t.incident_vertices(t.infinite_vertex() v)</i>
<i>Vertex_circulator</i>	<i>t.incident_vertices(t.infinite_vertex(), Face_handle f)</i>

Miscellaneous

<i>int</i>	<i>t.ccw(int i)</i>	Returns $i + 1$ modulo 3. <i>Precondition:</i> $0 \leq i \leq 2$.
<i>int</i>	<i>t.cw(int i)</i>	Returns $i + 2$ modulo 3. <i>Precondition:</i> $0 \leq i \leq 2$.
<i>Triangle</i>		
	<i>t.triangle(Face_handle f)</i>	Returns the triangle formed by the three vertices of <i>f</i> . <i>Precondition:</i> The face is finite.
<i>Segment</i>	<i>t.segment(Face_handle f, int i)</i>	Returns the line segment formed by the vertices <i>ccw(i)</i> and <i>cw(i)</i> of face <i>f</i> . <i>Precondition:</i> $0 \leq i \leq 2$. The vertices <i>ccw(i)</i> and <i>cw(i)</i> of <i>f</i> are finite.
<i>Segment</i>	<i>t.segment(Edge e)</i>	Returns the line segment corresponding to edge <i>e</i> . <i>Precondition:</i> <i>e</i> is a finite edge
<i>Segment</i>	<i>t.segment(Edge_circulator ec)</i>	Returns the line segment corresponding to edge <i>*ec</i> . <i>Precondition:</i> <i>*ec</i> is a finite edge.
<i>Segment</i>	<i>t.segment(Edge_iterator ei)</i>	Returns the line segment corresponding to edge <i>*ei</i> . <i>Precondition:</i> <i>*ei</i> is a finite edge.
<i>Point</i>	<i>t.circumcenter(Face_handle f)</i>	Compute the circumcenter of the face pointed to by <i>f</i> . This function is available only if the corresponding function is provided in the geometric traits.

— advanced —

Setting

void *t.set_infinite_vertex(Vertex_handle v)*

Checking

The responsibility of keeping a valid triangulation belongs to the users if advanced operations are used. Obviously the advanced user, who implements higher levels operations may have to make a triangulation invalid at some times. The following method is provided to help the debugging.

bool *t.is_valid(bool verbose = false, int level = 0)*

Checks the combinatorial validity of the triangulation and also the validity of its geometric embedding. This method is mainly a debugging help for the users of advanced features.

— advanced —

I/O

The I/O operators are defined for *iostream*. The format for the *iostream* is an internal format.

ostream& *ostream& os << T*

Inserts the triangulation *t* into the stream *os*.
Precondition: The insert operator must be defined for *Point*.

istream& *istream*& *is* >> *T* Reads a triangulation from stream *is* and assigns it to *t*.
Precondition: The extract operator must be defined for *Point*.

The information output in the *iostream* is:

- the dimension, the number of vertices (including the infinite one), and the number of faces (including infinite ones).
- for each vertex (except the infinite vertex), the non combinatorial information stored in that vertex (point, etc.).
- for each faces, the indices of its vertices and the non combinatorial information (if any) in this face. - for each face again the indices of the neighboring faces.

The index of an item (vertex of face) is the rank of this item in the output order. When dimension < 2, the same information is output for faces of maximal dimension instead of faces.

CGAL also provides stream operators << to draw triangulations on *CGAL::Window_stream*, the LEDA based graphic package, and on *CGAL::Qt_widget*, the Qt based graphic package. These operators requires respectively the include statements :

```
#include CGAL/IO/Window_stream.h
```

```
#include CGAL/IO/Qt_widget_Triangulation_2.h
```

See the chapters on *Window_stream* and on *Qt_widget* in the Support Library manual.

Implementation

Locate is implemented by a line walk from a vertex of the face given as optional parameter (or from a finite vertex of *infinite_face()* if no optional parameter is given). It takes time $O(n)$ in the worst case, but only $O(\sqrt{n})$ on average if the vertices are distributed uniformly at random.

Insertion of a point is done by locating a face that contains the point, and then splitting this face. If the point falls outside the convex hull, the triangulation is restored by flips. Apart from the location, insertion takes a time time $O(1)$. This bound is only an amortized bound for points located outside the convex hull.

Removal of a vertex is done by removing all adjacent triangles, and retriangulating the hole. Removal takes time $O(d^2)$ in the worst case, if d is the degree of the removed vertex, which is $O(1)$ for a random vertex.

The face, edge, and vertex iterators on finite features are derived from their counterparts visiting all (finite and infinite) features which are themselves derived from the corresponding iterators of the triangulation data structure.

See Also

TriangulationTraits_2

TriangulationDataStructure_2

TriangulationDataStructure_2::Face

TriangulationDataStructure_2::Vertex

CGAL::Triangulation_data_structure_2<Vb,Fb>

CGAL::Triangulation_vertex_base_2<Traits>

CGAL::Triangulation_face_base_2<Traits>

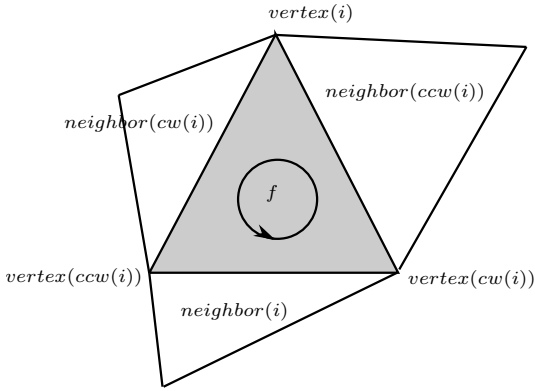


Figure 20.16: Vertices and neighbors.

CGAL::Triangulation_cw_ccw_2

Definition

The class *Triangulation_cw_ccw_2* offer two functions *int cw(int i)* and *int ccw(int i)* which given the index of a vertex in a face compute the index of the next vertex of the same face in clockwise or counterclockwise order. This works also for neighbor indexes. Thus, for example the neighbor *neighbor(cw(i))* of a face *f* is the neighbor which is next to *neighbor(i)* turning clockwise around *f*. The face *neighbor(cw(i))* is also the first face encountered after *f* when turning clockwise around vertex *i* of *f*.

Many of the classes in the triangulation package inherit from *Triangulation_cw_ccw_2*. This is for instance the case for *CGAL::Triangulation_2<Traits,Tds>::Face*. Thus, for example the neighbor *neighbor(cw(i))* of a face *f* is the neighbor which is next to *neighbor(i)* turning clockwise around *f*. The face *neighbor(cw(i))* is also the first face encountered after *f* when turning clockwise around vertex *i* of *f*.

```
#include <CGAL/Triangulation_2.h>
```

Creation

<i>Triangulation_cw_ccw_2</i>	<i>a</i> ;	default constructor.
-------------------------------	------------	----------------------

Operations

<i>int</i>	<i>a.ccw(const int i)</i>	returns the index of the neighbor or vertex that is next to the neighbor or vertex with index <i>i</i> in counterclockwise order around a face.
<i>int</i>	<i>a.cw(const int i)</i>	returns the index of the neighbor or vertex that is next to the neighbor or vertex with index <i>i</i> in counterclockwise order around a face.

See Also

CGAL::Triangulation_2<Traits,Tds>

CGAL::TriangulationDSFace_2

CGAL::Triangulation_euclidean_traits_2<K>

Definition

The class *Triangulation_euclidean_traits_2*<*K*> can be used to instantiate the geometric traits class of basic and Delaunay triangulations. The templated parameter *K* has to be instantiated by a model of the *Kernel* concept. The class *Triangulation_euclidean_traits_2*<*K*> uses types and predicates defined *K*.

```
#include <CGAL/Triangulation_euclidean_traits_2.h>
```

Is Model for the Concepts

TriangulationTraits_2

DelaunayTriangulationTraits_2

See Also

TriangulationTraits_2

DelaunayTriangulationTraits_2

CGAL::Triangulation_2<*Traits*,*Tds*>

CGAL::Delaunay_triangulation_2<*Traits*,*Tds*>

CGAL::Triangulation_euclidean_traits_xy_3<*K*>

CGAL::Triangulation_euclidean_traits_xy_3<K>

Definition

The class *Triangulation_euclidean_traits_xy_3<K>* is a geometric traits class which allows to triangulate a terrain. This traits class is designed to build a two dimensional triangulation embedded in 3D space, i.e. a triangulated surface, such that its on the *xy* plane is a Delaunay triangulation. This is a usual construction for GIS terrains. Instead of really projecting the 3D points and maintaining a mapping between each point and its projection (which costs space and is error prone) the class *Triangulation_euclidean_traits_xy_3<K>* supplies geometric predicates that ignore the *z*-coordinate of the points.

The class is a model of the concept *DelaunayTriangulationTraits_2* except that it does not provide the type and constructors required to build the dual Voronoi diagram.

Parameters

The template parameter *K* has to be instantiated by a model of the *Kernel* concept. *Triangulation_euclidean_traits_xy_3<K>* uses types and predicates defined in *K*.

```
#include <CGAL/Triangulation_euclidean_traits_xy_3.h>
```

Types

```
typedef Point_3<K>      Point_2;
typedef Segment_3<K>    Segment_2;

typedef Triangle_3<K>    Triangle_2;
```

The following predicates and constructor types are provided

Triangulation_euclidean_traits_xy_3<K>::Construct_segment_2

A constructor object for *Segment_2*. Provides :
Segment_2 operator()(Point_2 p, Point_2 q),
 which constructs a segment from two points.

Triangulation_euclidean_traits_xy_3<K>::Construct_triangle_2

A constructor object for *Triangle_2*. Provides :
Triangle_2 operator()(Point_2 p, Point_2 q, Point_2 r),
 which constructs a triangle from three points.

Triangulation_euclidean_traits_xy_3<K>::Compare_x_2

Predicate object. Provides the operator :
Comparison_result operator()(Point_2 p, Point_2 q)
 which returns *SMALLER*, *EQUAL* or *LARGER* according to the *x*-ordering of points *p* and *q*.

Triangulation_euclidean_traits_xy_3<K>:: Compare_y_2

Predicate object. Provides the operator :
Comparison_result operator()(Point_2 p, Point_2 q)
which returns (*SMALLER*, *EQUAL* or *LARGER*) according to the y-ordering of points *p* and *q*.

Triangulation_euclidean_traits_xy_3<K>:: Orientation_2

Predicate object. Provides the operator :
Orientation operator()(Point_2 p, Point_2 q, Point_2 r)
which returns *LEFT_TURN*, *RIGHT_TURN* or *COLLINEAR* according to the position of the projection of *r* with respect to the projection of the oriented line *pq*.

Triangulation_euclidean_traits_xy_3<K>:: Side_of_oriented_circle_2

Predicate object. Provides the operator : *Oriented_side operator()(Point_2 p, Point_2 q, Point_2 r, Point_2 s)* which takes four points *p, q, r, s* as arguments and returns *ON_POSITIVE_SIDE*, *ON_NEGATIVE_SIDE* or, *ON_ORIENTED_BOUNDARY* according to the position of the projection of points with respect to the oriented circle through the projections of *p, q* and *r*.

Creation

Only a default constructor, copy constructor and an assignment operator are required. Note that further constructors can be provided.

Triangulation_euclidean_traits_xy_3<K> traits;

default constructor.

Triangulation_euclidean_traits_xy_3<K> traits(Triangulation_euclidean_traits_xy_3 tr);

Copy constructor.

Triangulation_euclidean_traits_xy_3 traits = Triangulation_euclidean_traits_xy_3 tr

Assignment operator.

Access to predicate objects

The following access functions are provided

<i>Construct_segment_2</i>	<i>traits.construct_segment_2_object()</i>
<i>Construct_triangle_2</i>	<i>traits.construct_triangle_2_object()</i>
<i>Comparison_x_2</i>	<i>traits.compare_x_2_object()</i>
<i>Comparison_y_2</i>	<i>traits.compare_y_2_object()</i>
<i>Orientation_2</i>	<i>traits.orientation_2_object()</i>
<i>Side_of_oriented_circle_2</i>	<i>traits.side_of_oriented_circle_2_object()</i>

See Also

TriangulationTraits_2

DelaunayTriangulationTraits_2

CGAL::Triangulation_2<Traits,Tds>

CGAL::Delaunay_triangulation_2<Traits,Tds>

CGAL provides also predefined geometric traits class *Triangulation_euclidean_traits_yz_3<K>* and *Triangulation_euclidean_traits_zx_3<K>* to deal with projections on the *xz*- or the *yz*-plane, respectively.

```
#include <CGAL/Triangulation_euclidean_traits_xz_3.h>
```

```
#include <CGAL/Triangulation_euclidean_traits_yz_3.h>
```

CGAL::Triangulation_face_base_2<Traits,Fb>

Definition

The class *Triangulation_face_base_2*<Traits,Fb> is a model for the concept *TriangulationFaceBase_2*. It is the default face base class for basic and Delaunay triangulation.

These default base class can be used directly or can serve as a base to derive other base classes with some additional attribute (a color for example) tuned for specific applications.

Parameters

The first template parameter of *Triangulation_face_base_2*<Traits,Fb> is a geometric traits class. The geometric traits is actually not used by the class.

The second template parameter has to be a model of the concept *TriangulationDSFaceBase_2* and will serve as a base class for *Triangulation_face_base_2*<Traits,Fb> . CGAL provides a default instantiation for this parameter which is *Triangulation_ds_face_base_2*<>.

```
#include <CGAL/Triangulation_face_base_2.h>
```

Is Model for the Concepts

TriangulationFaceBase_2

See Also

CGAL::Triangulation_ds_face_base_2<Tds>

CGAL::Triangulation_vertex_base_2<Traits,Vb>

CGAL::Triangulation_2<Traits,Tds>

CGAL::Triangulation_face_base_with_info_2<Info,Traits,Fb>

Definition

The class *Triangulation_face_base_with_info_2<Info,Traits,Fb>* is a model of the concept *TriangulationFaceBase_2* to be plugged into the triangulation data structure of a triangulation class. It provides an easy way to add some user defined information in the faces of a triangulation.

```
#include <CGAL/Triangulation_face_base_with_info_2.h>
```

Parameters

The first template argument is the information the user would like to add to a face. It has to be *DefaultConstructible* and *Assignable*.

The second template argument is a geometric traits class and is actually not used in *Triangulation_face_base_with_info_2<Info,Traits,Fb>* .

The third parameter is a face base class from which *Triangulation_face_base_with_info_2<Info,Traits,Fb>* derives.

Inherits From

Fb

Is Model for the Concepts

Because *Triangulation_face_base_with_info_2<Info,Traits,Fb>* derives from the class instantiating its third parameter, it will be a model of the same face base concept as its parameter : *TriangulationFaceBase_2*, *ConstrainedTriangulationFaceBase_2* , or *RegularTriangulationFaceBase_2*

Types

```
typedef Info          Info;
```

Access Functions

<code>const Info& f.info() const</code>	Returns a const reference to the object of type <i>Info</i> stored in the face.
<code>Info& f.info()</code>	Returns a reference to the object of type <i>Info</i> stored in the face.

See Also

CGAL::Triangulation_face_base_2<Traits,Fb>
CGAL::Constrained_triangulation_face_base_2<Traits,Fb>
CGAL::Regular_triangulation_face_base_2<Traits,Fb>

CGAL::Triangulation_hierarchy_2<Tr>

Definition

The class *Triangulation_hierarchy_2<Tr>* implements a triangulation augmented with a data structure which allows fast point location queries.

The data structure is a hierarchy of triangulations. The triangulation at the lowest level is the original triangulation where operations and point location are to be performed. Then at each succeeding level, the data structure stores a triangulation of a small random sample of the vertices of the triangulation at the preceding level.

Point location is done through a top-down nearest neighbor query. The nearest neighbor query is first performed naively in the top level triangulation. Then, at each following level, the nearest neighbor at that level is found through a linear walk performed from the nearest neighbor found at the preceding level.

Because the number of vertices in each triangulation is only a small fraction of the number of vertices of the preceding triangulation the data structure remains small and achieves fast point location queries on real data. As proved in [Dev98], this structure has an optimal behaviour when it is built for Delaunay triangulations. However it can be used as well for other triangulations. The class *Triangulation_hierarchy_2<Tr>* is templated by a parameter which is to be instantiated by anyone of the CGAL triangulation classes.

```
#include <CGAL/Triangulation_hierarchy_2.h>
```

Inherits From

Tr

Types

The class *Triangulation_hierarchy_2<Tr>* inherits the types from its base triangulation class *Tr*.

The class *Triangulation_hierarchy_2<Tr>* offers exactly the same functionalities as the triangulation *Tr* does. Location queries are overloaded to benefit from the data structure. Modifiers (insertion, removal) are overloaded to take care of updating the data structure.

Be careful that I/O operations are not overloaded. Writing a *Triangulation_hierarchy_2<Tr>* into a file writes only the lowest level triangulation and drop the hierarchy and reading it from a file results in a triangulation whose efficiency will be that of an ordinary triangulation.

See Also

CGAL::Triangulation_2<Traits,Tds>

CGAL::Delaunay_triangulation_2<Traits,Tds> *TriangulationHierarchyVertexBase_2*,

CGAL::Triangulation_hierarchy_vertex_base_2<Vb>

CGAL::Triangulation_hierarchy_vertex_base_2<Vb>

Definition

The class *Triangulation_hierarchy_vertex_base_2<Vb>* is designed to be used as a vertex base class of a triangulation plugged into a *Triangulation_hierarchy_2<Tr>*.

It is a model of the concept *TriangulationHierarchyVertexBase_2* which refines the concept *TriangulationVertexBase_2*.

This class is templated by a parameter *Vb* which is to be instantiated by a model of the concept *TriangulationVertexBase_2*. The class *Triangulation_hierarchy_vertex_base_2<Vb>* inherits from the class *Vb*. This design allows to use either the default vertex base class or a user customized vertex base with additional functionalities.

```
#include <CGAL/Triangulation_hierarchy_2.h>
```

Is Model for the Concepts

TriangulationHierarchyVertexBase_2

Inherits From

Vb

See Also

TriangulationVertexBase_2

TriangulationHierarchyVertexBase_2

CGAL::Triangulation_vertex_base_2<Traits>

CGAL::Triangulation_vertex_base_2<Traits,Vb>

Definition

The class *Triangulation_vertex_base_2<Traits,Vb>* is the default model for the concept *TriangulationVertexBase_2*.

Triangulation_vertex_base_2<Traits,Vb> can be simply plugged in the triangulation data structure of a triangulation, or used as a base class to derive other base vertex classes tuned for specific applications.

```
#include <CGAL/Triangulation_vertex_base_2.h>
```

Parameters

Triangulation_vertex_base_2<Traits,Vb> is templated by a geometric traits class which provide the type *Point*. It is strongly recommended to instantiate this traits class with the model used for the triangulation traits class. This ensures that the point type defined by *Triangulation_vertex_base_2<Traits,Vb>* is the same as the point type defined by the triangulation.

The second template parameter of *Triangulation_vertex_base_2<Traits,Vb>* has to be a model of the concept *TriangulationDSVertexBase_2*. By default this parameter is instantiated by *CGAL::Triangulation_ds_vertex_base_2<>*.

Is Model for the Concepts

TriangulationVertexBase_2

Inherits From

Vb

See Also

CGAL::Triangulation_ds_vertex_base_2<Tds>
CGAL::Triangulation_face_base_2<Traits,Fb>
CGAL::Regular_triangulation_vertex_base_2<Traits,Vb>
CGAL::Triangulation_vertex_base_with_info_2<Info,Traits,Vb>

CGAL::Triangulation_vertex_base_with_info_2<Info,Traits,Vb>

Definition

The class *Triangulation_vertex_base_with_info_2<Info,Traits,Vb>* is designed to be used as a base vertex class of a triangulation. It provides an easy way to add some user defined information in vertices.

```
#include <CGAL/Triangulation_vertex_base_with_info_2.h>
```

Parameters

The first template parameter is the information the user would like to add to a vertex. It has to be *DefaultConstructible* and *Assignable*.

The second template parameter is the geometric traits class which provides the *Point_2*. It is strongly recommended to instantiate this parameter with the traits class used for the triangulation. This ensures that the point type defined by *Triangulation_vertex_base_with_info_2<Info,Traits,Vb>* matches the point type defined by the triangulation.

The third template parameter is a vertex base class from which *Triangulation_vertex_base_with_info_3* derives. By default this parameter is instantiated by *CGAL::Triangulation_vertex_base_2<Traits>*.

Is Model for the Concepts

The parameter *Vb* is a model of some vertex base concept. *Triangulation_vertex_base_with_info_2<Info,Traits,Vb>* derives from *Vb* and will be a model of the same vertex base concept : *TriangulationVertexBase_2*, or *RegularTriangulationVertexBase_2*.

Types

```
typedef Info          Info;
```

Access Functions

<i>const Info&</i>	<i>v.info() const</i>	Returns a const reference to the object of type <i>Info</i> stored in the vertex.
<i>Info&</i>	<i>v.info()</i>	Returns a reference to the object of type <i>Info</i> stored in the vertex.

See Also

CGAL::Triangulation_face_base_with_info_2<Info,Traits,Fb>
CGAL::Triangulation_vertex_base_2<Traits,Vb>
CGAL::Regular_triangulation_vertex_base_2<Traits,Vb>

CGAL::Weighted_point<Pt,Wt>

Definition

The class *Weighted_point*<Pt,Wt> provides a type associating a point type *Pt* with a weight type *Wt*. It is used in the traits classes *Regular_triangulation_euclidean_traits_2* and *Regular_triangulation_euclidean_traits_3*.

```
#include <CGAL/Weighted_point.h>
```

Inherits From

Pt

Types

<i>Pt</i>	<i>Point</i> ;	The point type
<i>Wt</i>	<i>Weight</i> ;	The weight type.

Creation

```
Weighted_point<Pt,Wt> wp( Point p=Point(), Weight w= Weight(0));  
Weighted_point<Pt,Wt> wp( Weighted_point wq);
```

copy constructor.

Access Functions

<i>Point</i>	<i>wp.point</i> ()
<i>Weight</i>	<i>wp.weight</i> ()

See Also

CGAL::Regular_triangulation_euclidean_traits_2<Rep,Weight>
CGAL::Regular_triangulation_euclidean_traits_3<R,Weight>.

Chapter 21

2D Triangulation Data Structure

Sylvain Pion and Mariette Yvinec

Contents

21.1 Definition	1455
21.1.1 A data structure based on faces and vertices	1455
21.1.2 The set of faces and vertices	1456
21.2 The Concept of Triangulation Data Structure	1456
21.3 The Default Triangulation Data Structure	1457
21.3.1 Flexibility	1457
21.3.2 The cyclic dependancy of template parameters	1458
21.3.3 The rebind mechanism	1458
21.3.4 Making use of the flexibility	1459

21.1 Definition

A triangulation data structure is a data structure designed to handle the representation of a two dimensional triangulation. The concept of triangulation data structure was primarily designed to serve as a data structure for CGAL 2D triangulation classes which are triangulations embedded in a plane. However it appears that the concept is more general and can be used for any orientable triangulated surface without boundary, whatever may be the dimensionality of the space the triangulation is embedded in.

21.1.1 A data structure based on faces and vertices

The representation of CGAL 2D triangulations is based on faces and vertices, Edges are only implicitly represented trough the adjacency relations between two faces.

The triangulation data structure can be seen as a container for faces and vertices maintaining incidence and adjacency relations among them.

Each triangular face gives access to its three incident vertices and to its three adjacent faces. Each vertex gives access to one of its incident faces and through that face to the circular list of its incident faces.

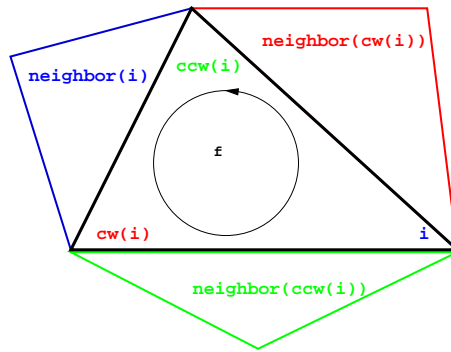


Figure 21.1: Vertices and neighbors.

The three vertices of a face are indexed with 0, 1 and 2. The neighbors of a face are also indexed with 0,1,2 in such a way that the neighbor indexed by i is opposite to the vertex with the same index. See Figure 21.1, the functions $ccw(i)$ and $cw(i)$ shown on this figure compute respectively $i + 1$ and $i - 1$ modulo 3

Each edge has two implicit representations : the edge of a face f which is opposed to the vertex indexed i , can be represented as well as an edge of the $neighbor(i)$ of f .

This kind of representation of simplicial complexes extends in any dimension. More precisely, in dimension d , the data structure will explicitly represents cells (i. e. faces of maximal dimension) and vertices (i. e. faces of dimension 0). All faces of dimension between 1 and $d - 1$ will have an implicit representation. The 2D triangulation data structure can represent simplicial complexes of dimension 2, 1 or 0.

21.1.2 The set of faces and vertices

The set of faces maintained by a 2D triangulation data structure is such that each edge is incident to two faces. In other words, the set of maintained faces is topologically equivalent to a two-dimensional triangulated sphere.

This rule extends to lower dimensional triangulation data structure arising in degenerate cases or when the triangulations have less than three vertices. A one dimensional triangulation structure maintains a set of vertices and edges which forms a ring topologically equivalent to a 1-sphere.

A zero dimensional triangulation data structure only includes two adjacent vertices that is topologically equivalent to a 0-sphere.

21.2 The Concept of Triangulation Data Structure

A model of *TriangulationDataStructure_2* can be seen has a container for the faces and vertices of the triangulation. This class is also responsible for the combinatorial integrity of the triangulation. This means that the triangulation data structure maintains proper incidence and adjacency relations among the vertices and faces of a triangulation while combinatorial modifications of the triangulation are performed. The term combinatorial modification refers to operations which do not involve any knowledge about the geometric embedding of the triangulation. For example, the insertion of a new vertex in a given face, or in a given edge, the suppression of a vertex of degree three, the flip of two edge are examples of combinatorial operation performed at the data structure level.

The triangulation data structure is required to provide :

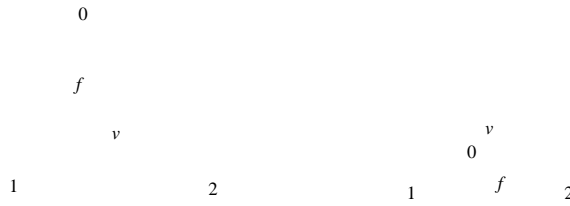


Figure 21.2: Insertion of a new vertex, splitting a face

- the types *Vertex* and *Face* for the vertices and faces of the triangulations
- the type *Vertex_handle* and *Face_handle* which are models of the concept *Handle* and through which the vertices and faces are accessed.
- iterators to visit all the vertices, edges and faces of the triangulation,
- circulators to visit all the vertices, edges and faces incident to a given vertex

The triangulation data structure is responsible for the creation and removal of faces and vertices (memory management). It provides function that gives the number of faces, edges and vertices of the triangulation.

The triangulation data structure provides member functions to perform the following combinatorial transformation of the triangulation:

- flip of two adjacent faces,
- addition of a new vertex splitting a given face see Figure 21.2,
- addition of a new vertex splitting a given edge,
- addition of a new vertex raising by one the dimension of a degenerate – lower dimensional triangulation,
- removal of a vertex incident to three faces,
- removal of a vertex lowering the dimension of the triangulation

21.3 The Default Triangulation Data Structure

CGAL provides the class `CGAL::Triangulation_data_structure_2<Vb,Fb>` as a default triangulation data structure.

21.3.1 Flexibility

In order to provide flexibility, the default triangulation data structure is templated by two parameters which stand respectively for a vertex base class and a face base class. The concept *TriangulationDSVertexBase_2* and *TriangulationDSFaceBase_2* describe the requirements for the vertex and face classes of a triangulation data structure.

This design allows the user to plug in the triangulation data structure his own vertex or face classes tuned for his application.

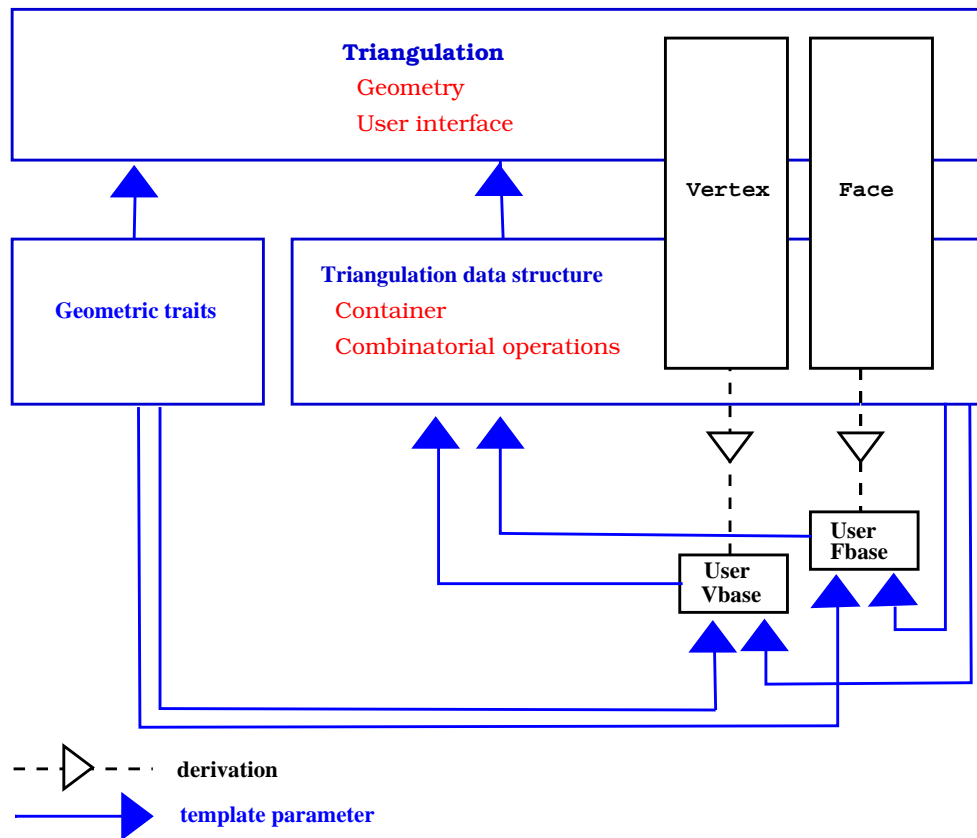


Figure 21.3: The cyclic dependency in triangulations software design.

21.3.2 The cyclic dependency of template parameters

Since adjacency and incidence relation are stored in vertices and faces, the vertex and face classes have to know the types of handles on faces and vertices provided by the triangulation data structure. Therefore, vertex and face classes need to be templated by the triangulation data structure. Because the triangulation data structure is itself templated by the vertex and face classes this induces a cyclic dependency. See figure 21.3.

21.3.3 The rebind mechanism

The solution proposed by CGAL to resolve this cyclic dependency is based on a rebind mechanism similar to the mechanism used in the standard allocator class `std::allocator`. The vertex and face classes plugged in the instantiation of a triangulation data structure are themselves instantiated with a fake data structure. The triangulation data structure will then rebind these classes, plugging itself at the place of the fake data structure, before using them to derive the vertex and face classes. The rebinding is performed through a nested template class *Rebind.TDS* in the vertex and face class, which provide the rebound class as a type called *Other*.

Here is how it works schematically. First, here is the rebinding taking place in the triangulation data structure.

```
template < class Vb, class Fb >
class Triangulation_data_structure
{
```

```

typedef Triangulation_data_structure<Vb,Fb>    Self;

// Rebind the vertex and face base to the actual TDS (Self).
typedef typename Vb::template Rebind_TDS<Self>::Other  VertexBase;
typedef typename Fb::template Rebind_TDS<Self>::Other  FaceBase;

// ... further internal machinery leads to the final public types:
public:
    typedef ...  Vertex;
    typedef ...  Face;
    typedef ...  Vertex_handle;
    typedef ...  Face_handle;
};

```

Then, here is the vertex class with its nested *Rebind_TDS* template class and its template parameter set by default to an internal type faking a triangulation data structure.

```

template < class TDS = an internal type faking a triangulation data
structure >
class Vertex_base
{
public:
    template < class TDS2 >
    struct Rebind_TDS {
        typedef Vertex_base<TDS2>    Other;
    };
    ...
};

```

Imagine an analog *Face_base* class. The triangulation data structure is then instantiated as follows :

```

typedef Triangulation_data_structure< Vertex_base<>, Face_base<> > TDS;

```

21.3.4 Making use of the flexibility

There are several possibilities to make use of the flexibility offered by the triangulation data structure.

- First, when the user needs to have, in vertices and faces, additional information which do not depend on types defined by the triangulated data structure, predefined classes *Triangulation_vertex_base_with_info* and *Triangulation_face_base_with_info* can be plugged in. Those classes have a template parameter *Info* to be instantiated by a user defined type. They store a data member of this type and gives access to it.
- Second, the user can derive his own base classes from the default base classes : *Triangulation_ds_vertex_base_2*, and *Triangulation_ds_face_base_2* are the default base classes to be plugged in a triangulation data structure used alone. Triangulation classes requires a data structure in which other base classes have been plugged in. The default base classes for most of the triangulation classes are *Triangulation_vertex_base_2*, and *Triangulation_face_base_2* are the default base classes to be used when the triangulation data structure is plugged in a triangulation class.

When derivation is used, the rebinding mechanism is slightly more involved, because it is necessary to rebind the base class itself. However the user will be able to use in his classes references to types provided by the triangulation data structure. For example,

```

template < class Gt, class Vb = CGAL::Triangulation_vertex_base_2<Gt> >
class My_vertex_base
    : public Vb
{
public :
    template < typename TDS2 >
    struct Rebind_TDS {
        typedef typename Vb::template Rebind_TDS<TDS2>::Other    Vb2;
        typedef My_vertex_base<Gt,Vb2>                            Other;
    };

    typedef typename Vb::Triangulation_data_structure    Tds;
    typedef typename Tds::Vertex_handle                  Vertex_handle;
    .....
};

```

- At last the user can write his own base classes. If the triangulation data structure is used alone, the requirements for the base classes are described by the concepts *TriangulationDSVertexBase_2* and *TriangulationDSFaceBase_2*, documented page [1476](#) and page [1471](#). If the triangulation data structure is plugged into a triangulation class, the concepts for the vertex and base classes depends on the triangulation class. The most basic concepts, valid for basic and Delaunay triangulations are *TriangulationVertexBase_2* and *TriangulationFaceBase_2*, documented page [1426](#) and page [1422](#).

See section [20.11](#) for examples of using the triangulation data structure flexibility.

2D Triangulation Data Structure Reference Manual

Sylvain Pion and Mariette Yvinec

The triangulation data structure can be seen as a container for the faces and vertices of a triangulation. This class also takes care of all the combinatorial operations performed on the triangulation.

The class `CGAL::Triangulation_data_structure_2<Vb,Fb>` is a model of the concept `TriangulationDataStructure_2`, which includes the subconcepts `TriangulationDataStructure_2::Face` and `TriangulationDataStructure_2::Vertex`.

To ensure all the **flexibility** of the triangulation classes, described and in Section 20.11 of Chapter 20, the model `CGAL::Triangulation_data_structure_2<Vb,Fb>` has two templates parameters. The class `CGAL::Triangulation_data_structure_2<Vb,Fb>` derives its `Vertex` and `Face` types from the two template parameters `Vb` and `Fb` respectively.

If the triangulation data structure is used alone, these parameters have to be instantiated by models of the concepts `TriangulationDSFaceBase_2` and `TriangulationDSVertexBase_2`. These concepts are described in this chapter together with their default models `CGAL::Triangulation_ds_face_base_2<Tds>` and `CGAL::Triangulation_ds_vertex_base_2<Tds>`.

If the triangulation data structure is plugged into a triangulation class, the parameters have to be instantiated by models of different refining concepts according to the actual type of the triangulation. These refining concepts and their models are described in Chapter 20

21.4 Classified Reference Pages

Concepts

<code>TriangulationDataStructure_2</code>	page 1463
<code>TriangulationDataStructure_2::Face</code>	page 1474
<code>TriangulationDataStructure_2::Vertex</code>	page 1478
<code>TriangulationDSFaceBase_2</code>	page 1471
<code>TriangulationDSVertexBase_2</code>	page 1476

Classes

<i>CGAL::Triangulation_data_structure_2<Vb,Fb></i>	page 1480
<i>CGAL::Triangulation_ds_face_base_2<Tds></i>	page 1482
<i>CGAL::Triangulation_ds_vertex_base_2<Tds></i>	page 1483
 <i>CGAL::Triangulation_cw_ccw_2</i>	page 1442

21.5 Alphabetical List of Reference Pages

<i>TriangulationDataStructure_2::Face</i>	page 1474
<i>TriangulationDataStructure_2::Vertex</i>	page 1478
<i>TriangulationDataStructure_2</i>	page 1463
<i>TriangulationDSFaceBase_2</i>	page 1471
<i>TriangulationDSVertexBase_2</i>	page 1476
<i>Triangulation_data_structure_2<Vb,Fb></i>	page 1480
<i>Triangulation_ds_face_base_2<Tds></i>	page 1482
<i>Triangulation_ds_vertex_base_2<Tds></i>	page 1483

TriangulationDataStructure_2

Definition

The concept `TriangulationDataStructure_2` describes the requirements for the second template parameter of the basic triangulation class `Triangulation_2<Traits,Tds>` and of all other 2D triangulation classes.

The concept can be seen as a container for the faces and vertices of the triangulation. The concept `TriangulationDataStructure_2` includes two subconcepts `TriangulationDataStructure_2::Vertex` and `TriangulationDataStructure_2::Face`, described respectively page 1478 and page 1474.

The `TriangulationDataStructure_2` maintains incidence and adjacency relations among vertices and faces.

Each triangular face gives access to its three incident vertices and to its three adjacent faces. Each vertex gives access to one of its incident faces and through that face to the circular list of its incident faces.

The three vertices of a face are indexed with 0, 1 and 2. The neighbors of a face are also indexed with 0,1,2 in such a way that the neighbor indexed by i is opposite to the vertex with the same index.

Each edge has two implicit representations : the edge of a face f which is opposed to the vertex indexed i , can be represented as well as an edge of the `neighbor(i)` of f . See Figure 20.2

The triangulation data structure is responsible for the combinatorial integrity of the triangulation. This means that the triangulation data structure allows to perform some combinatorial operations on the triangulation and guarantees the maintainance on proper incidence and adjacency relations among the vertices and faces. The term combinatorial operations means that those operations are purely topological and do not depend on the geometric embedding. Insertion of a new vertex in a given face, or in a given edge, suppression of a vertex of degree three, flip of two edges are examples of combinatorial operations.

Types

<code>TriangulationDataStructure_2::size_type</code>	Size type (unsigned integral type)
<code>TriangulationDataStructure_2::difference_type</code>	Difference type (signed integral type)
<code>TriangulationDataStructure_2::Vertex</code>	The vertex type. Requirements for this type are described in concept <code>TriangulationDataStructure_2::Vertex</code> page 1478.
<code>TriangulationDataStructure_2::Face</code>	The face type. Requirements for this type are described in concept <code>TriangulationDataStructure_2::Face</code> page 1474.

Vertices and faces are accessed via `Vertex_handle` and `Face_handle`. These types are models of the concept `Handles` which basically supports the two dereference operators `*` and `->`.

<code>TriangulationDataStructure_2::Vertex_handle</code>	Handle to a vertex
<code>TriangulationDataStructure_2::Face_handle</code>	Handle to a face.
<code>typedef std::pair<Face_handle,int> Edge;</code>	The edge type. The <code>Edge(f,i)</code> is edge common to faces f and $f.neighbor(i)$. It is also the edge joining the vertices <code>vertex(cw(i))</code> and <code>vertex(ccw(i))</code> of f .

The following iterators allow one to visit all the vertices, edges and faces of a triangulation data structure. They are all bidirectional, non-mutable iterators.

TriangulationDataStructure_2:: Face_iterator
TriangulationDataStructure_2:: Edge_iterator
TriangulationDataStructure_2:: Vertex_iterator

The following circulators allow to visit all the edges or faces incident to a given vertex and all the vertices adjacent to a given vertex. They are all bidirectional and non mutable.

TriangulationDataStructure_2:: Face_circulator
TriangulationDataStructure_2:: Edge_circulator
TriangulationDataStructure_2:: Vertex_circulator

Iterators and circulators are convertible to the corresponding handles, thus they can be passed directly as argument to the functions expecting a handle.

Creation

TriangulationDataStructure_2 tds; default constructor.

TriangulationDataStructure_2 tds(tds1); Copy constructor. All the vertices and faces are duplicated.

TriangulationDataStructure_2& tds = tds1

Assignment. All the vertices and faces of *tds1* are duplicated in *tds* . Former faces and vertices of *tds* , if any, are deleted

Vertex_handle tds.copy_tds(tds1, Vertex_handle v = Vertex_handle())

tds1 is copied into *tds*. If *v != NULL*, the vertex of *tds* corresponding to *v* is returned, otherwise *Vertex_handle()* is returned.

Precondition: The optional argument *v* is a vertex of *tds1*.

void tds.swap(& tds1) Swaps *tds* and *tds1*. Should be preferred to *tds=tds1* or *tds(tds1)* when *tds1* is deleted after that.

void tds.clear() Deletes all faces and all finite vertices.

Access Functions

int tds.dimension() returns the dimension of the triangulation.
size_type

size_type tds.number_of_vertices() returns the number of vertices in the data structure.
size_type

tds.number_of_faces() returns the number of two dimensional faces in the data structure.

size_type

tds.number_of_edges()

returns the number of edges in the triangulation data structure.

size_type

tds.number_of_full_dim_faces()

returns the number of full dimensional faces, i.e. faces of dimension equal to the dimension of the triangulation. This is the actual number of faces stored in the triangulation data structure.

— *advanced* —

Setting

void tds.set_dimension(int n)

sets the dimension.

— *advanced* —

Queries

bool tds.is_vertex(Vertex_handle v)

returns true if *v* is a vertex of *tds*.

bool tds.is_edge(Face_handle fh, int i)

tests whether (fh, i) is an edge of *tds*. Answers *false* when $dimension() < 1$.

bool tds.is_edge(Vertex_handle va, Vertex_handle vb)

returns true if *va vb* is an edge of *tds*.

bool tds.is_edge(Vertex_handle va, Vertex_handle vb, Face_handle &fr, int &i)

as previous. In addition, if true is returned *fr* and *i* are set such that the pair (fr, i) is the description of the ordered edge *va vb*.

bool tds.is_face(Face_handle fh)

tests whether *fh* is a face of *tds*. Answers *false* when $dimension() < 2$.

bool tds.is_face(Vertex_handle v1, Vertex_handle v2, Vertex_handle v3)

true if there is a face having *v1*, *v2* and *v3* as vertices.

bool tds.is_face(Vertex_handle v1, Vertex_handle v2, Vertex_handle v3, Face_handle &fr)

as above. In addition, if *true* is returned, *fr* is a pointer to the face with *v1*, *v2* and *v3* as vertices.

Traversing the triangulation

Face_iterator tds.faces_begin()

visits all faces

Face_iterator tds.faces_end()

Vertex_iterator tds.vertices_begin()

visits all vertices

Vertex_iterator tds.vertices_end()

<i>Edge_iterator</i>	<i>tds.edges_begin()</i>	visits all edges
<i>Edge_iterator</i>	<i>tds.edges_end()</i>	

Three circulator classes allow to traverse the edges or faces incident to a vertex or the vertices adjacent to this vertex.. A face circulator is invalidated by any modification of the face it points to. An edge circulator is invalidated by any modification of anyone of the two faces incident to the edge pointed to. A vertex circulator that turns around vertex v and that has as value a pointer to vertex w , is invalidated by any modification of anyone of the two faces incident to v and w .

<i>Vertex_circulator</i>	<i>tds.incident_vertices(Vertex_handle v, Face_handle f=NULL)</i>
--------------------------	--------------------------------------------------------------------

Precondition: If the face f is given, it has to be incident to be a face of tds incident to v and the circulator begins with the vertex $f->vertex(ccw(i))$ if i is the index of v in f .

<i>Edge_circulator</i>	<i>tds.incident_edges(Vertex_handle v, Face_handle f=NULL)</i>
------------------------	-----------------------------------------------------------------

Precondition: If the face f is given, it has to be a face of tds incident to v and the circulator begins with the edge $(f,cw(i))$ of f if i is the index of v in f .

<i>Face_circulator</i>	<i>tds.incident_faces(Vertex_handle v, Face_handle f=NULL)</i>
------------------------	-----------------------------------------------------------------

Precondition: If the face f is given, it has to be a face of tds incident to v and the circulator begins with the face f .

<i>Vertex_handle</i>	<i>tds.mirror_vertex(Face_handle f, int i)</i>
----------------------	-------------------------------------------------

returns vertex of $f->neighbor(i)$.

<i>int</i>	<i>tds.mirror_index(Face_handle f, int i)</i>
------------	------------------------------------------------

returns the index of f as a neighbor of $f->neighbor(i)$.

Modifiers

The following modifier member functions guarantee the combinatorial validity of the resulting triangulation.

<i>void</i>	<i>tds.flip(Face_handle f, int i)</i>	exchanges the edge incident to f and $f->neighbor(i)$ with the other diagonal of the quadrilateral formed by f and $f->neighbor(i)$.
-------------	----------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------

<i>Vertex_handle</i>	<i>tds.insert_first()</i>	creates the first vertex and returns a pointer to it.
<i>Vertex_handle</i>	<i>tds.insert_second()</i>	creates the second vertex and returns a pointer to it.

<i>Vertex_handle</i>	<i>tds.insert_in_edge(Face_handle f, int i)</i>	adds a vertex v splitting edge i of face f . Return a pointer to v .
----------------------	--------------------------------------------------	------------------------------------------------------------------------------

<i>Vertex_handle</i>	<i>tds.insert_in_face(Face_handle f)</i>	adds a vertex v splitting face f in three. Face f is modified, two new faces are created. Return a pointer to v
----------------------	-------------------------------------------	-------------------------------------------------------------------------------------------------------------------------

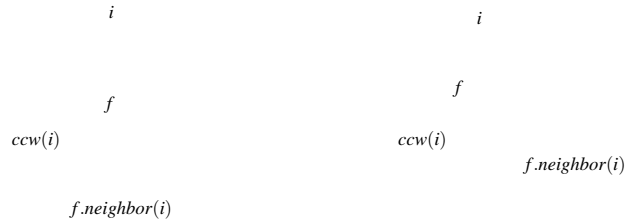


Figure 21.4: Flip.



Figure 21.5: Insertion

Vertex_handle *tds.insert_dim_up(Vertex_handle w, bool orient=true)*

adds a vertex v , increasing by one the dimension of the triangulation. Vertex v and the existing vertex w are linked to all the vertices of the triangulation. The boolean *orient* decides the final orientation of all faces. A pointer to vertex v is returned.

void *tds.remove_degree_3(Vertex_handle v, Face *f=NULL)*

removes a vertex of degree 3. Two of the incident faces are destroyed, the third one is modified. If parameter f is specified, it has to be a face incident to v and will be the modified face.

Precondition: Vertex v is a finite vertex with degree 3 and, if specified, face f is incident to v .

void *tds.remove_second(Vertex_handle v)*

removes the before last vertex.

void *tds.remove_first(Vertex_handle v)*

removes the last vertex.

void *tds.remove_dim_down(Vertex_handle v)*

removes vertex v incident to all other vertices and decreases by one the dimension of the triangulation.

Precondition: if the dimension is 2, the number of vertices is more than 3, if the dimension is 1, the number of vertices is 2.

————— *advanced* —————

The following modifiers are required for convenience of the advanced user. They do not guarantee the combinatorial validity of the resulting triangulation.

```
template< class EdgeIt>
Vertex_handle    tds.star_hole( EdgeIt edge_begin, EdgeIt edge_end)
```

creates a new vertex v and use it to star the hole whose boundary is described by the sequence of edges $[edge_begin, edge_end[$. Returns a pointer to the vertex.

```
template< class EdgeIt, class FaceIt>
Vertex_handle    tds.star_hole( EdgeIt edge_begin, EdgeIt edge_end, FaceIt face_begin, FaceIt face_end)
```

same as above, except that, to build the new faces, the algorithm first recycles faces in the sequence $[face_begin, face_end[$ and create new ones when the sequence is exhausted.

```
template< class EdgeIt>
void             tds.star_hole( Vertex_handle v, EdgeIt edge_begin, EdgeIt edge_end)
```

uses vertex v to star the hole whose boundary is described by the sequence of edges $[edge_begin, edge_end[$.

```
template< class EdgeIt, class FaceIt>
void             tds.star_hole( Vertex_handle v,
                               EdgeIt edge_begin,
                               EdgeIt edge_end,
                               FaceIt face_begin,
                               FaceIt face_end)
```

same as above, recycling faces in the sequence $[face_begin, face_end[$.

```
void             tds.make_hole( Vertex_handle v, List_edges& hole)
```

removes the vertex v , and store in $hole$ the list of edges on the boundary of the hole.

```
Vertex_handle    tds.create_vertex()           adds a new vertex.
Face_handle      tds.create_face( Face_handle f1, int i1, Face_handle f2, int i2, Face_handle f3, int i3)
```

adds a face which is the neighbor $i1$ of $f1$, $i2$ of $f2$ and $i3$ of $f3$.

```
Face_handle      tds.create_face( Face_handle f1, int i1, Face_handle f2, int i2)
```

adds a face which is the neighbor $i1$ of $f1$, and the neighbor $i2$ of $f2$.

```
Face_handle      tds.create_face( Face_handle f1, int i1, Vertex_handle v)
```

adds a face which is the neighbor $i1$ of $f1$, and has v as vertex.

```
Face_handle      tds.create_face( Vertex_handle v1, Vertex_handle v2, Vertex_handle v3)
```

adds a face with vertices $v1$, $v2$ and $v3$.

<i>Face_handle</i>	<i>tds.create_face(Vertex_handle v1, Vertex_handle v2, Vertex_handle v3, Face_handle f1, Face_handle f2, Face_handle f3)</i>	adds a face with vertices <i>v1</i> , <i>v2</i> and <i>v3</i> , and neighbors <i>f1</i> , <i>f2</i> , <i>f3</i> .
<i>Face_handle</i>	<i>tds.create_face()</i>	adds a face whose vertices and neighbors are set to NULL.
<i>void</i>	<i>tds.delete_face(Face_handle)</i>	deletes a face.
<i>void</i>	<i>tds.delete_vertex(Vertex_handle)</i>	deletes a vertex.

└────────── advanced ─────────┘

Miscellaneous

<i>int</i>	<i>tds.ccw(int i)</i>	returns $i + 1$ modulo 3. <i>Precondition:</i> $0 \leq i \leq 2$.
<i>int</i>	<i>tds.cw(int i)</i>	returns $i + 2$ modulo 3. <i>Precondition:</i> $0 \leq i \leq 2$.
<i>bool</i>	<i>tds.is_valid()</i>	checks the combinatorial validity of the triangulation: call the <i>is_valid()</i> member function for each vertex and each face, checks the number of vertices and the Euler relation between numbers of vertices, faces and edges.
<i>size_type</i>	<i>tds.degree(Vertex_handle v)</i>	Returns the degree of <i>v</i> in the triangulation.

I/O

The information output in the *ostream* is: the dimension, the number of (finite) vertices, the number of (finite) faces. Then comes for each vertex, the non combinatorial information stored in that vertex if any. Then comes for each face, the indices of its vertices and the non combinatorial information (if any) stored in this face. Then comes for each face again the indices of the neighboring faces. The index of an item (vertex or face) the rank of this item in the output order. When dimension < 2, the same information is output for faces of maximal dimension instead of faces.

<i>void</i>	<i>tds.file_output(ostream& os, Vertex_handle v = Vertex_handle(), bool skip_first=false)</i>	writes <i>tds</i> into the stream <i>os</i> . If <i>v</i> is not a null handle, vertex <i>v</i> is output first or skipped if <i>skip_first</i> is true.
<i>Vertex_handle</i>	<i>tds.file_input(istream& is, bool skip_first=false)</i>	inputs <i>tds</i> from file and returns a pointer to the first input vertex. If <i>skip_first</i> is true, it is assumed that the first vertex has been omitted when output.

istream& *istream*& *is* >> *TriangulationDataStructure_3* & *tds*

reads a combinatorial triangulation from *is* and assigns it to
tds

ostream& *ostream*& *os* << *TriangulationDataStructure_3* *tds*

writes *tds* into the stream *os*

Has Models

CGAL::Triangulation_data_structure_2<*Vb*,*Fb*>

See Also

TriangulationDataStructure_2::Face

TriangulationDataStructure_2::Vertex

CGAL::Triangulation_2<*Traits*,*Tds*>

TriangulationDSFaceBase_2

Definition

The concept `TriangulationDSFaceBase_2` describes the requirements for the base face of a *Triangulation_data_structure_2* <Vb.Fb>.

Note that if the *Triangulation_data_structure_2* is plugged into a triangulation class, the face base class may have additional geometric requirements depending on the triangulation class.

At the base level, (see Sections 20.3 and 21.3), a face stores handles on its three vertices and on the three neighboring faces. The vertices and neighbors are indexed 0, 1 and 2. Neighbor i lies opposite to vertex i .

Since the *Triangulation_data_structure_2* is the class which defines the handle types, the face base class has to be somehow parameterized by the triangulation data structure. But since the *Triangulation_data_structure_2* itself is parameterized by the face and vertex base classes, there is a cycle in the definition of these classes. In order to break the cycle, the base classes for faces and vertices which are plugged in to instantiate a *Triangulation_data_structure_2* use a *void* as triangulation data structure parameter. Then, the *Triangulation_data_structure_2* uses a *rebind* mechanism (similar to the one specified in *std::allocator*) in order to plug itself as parameter in the face and vertex base classes. This mechanism requires that the base class provides a templated nested class *Rebind_TDS* that itself provides the subtype *Rebind_TDS<TDS>::Other* which is the *rebound* version of the base class. This *rebound* base class is the class that the *Triangulation_data_structure_2* actually uses as a base class for the class *Triangulation_data_structure_2::Face*.

Types

The concept `TriangulationDSFaceBase_2` has to provide the following types.

```
TriangulationDSFaceBase_2:: template <typename TDS2> struct Rebind_TDS;
```

This nested template class has to define a type *Other* which is the *rebound* face base, where the *Triangulation_data_structure_2* is actually plugged in. This type *Other* will be the actual base of the class *Triangulation_data_structure_2::Face*.

```
typedef TriangulationDataStructure_2 Triangulation_data_structure;
typedef TriangulationDataStructure_2::Vertex_handle Vertex_handle;
typedef TriangulationDataStructure_2::Face_handle Face_handle;
```

Creation

TriangulationDSFaceBase_2 *f*; default constructor.
TriangulationDSFaceBase_2 *f*(*Vertex_handle* v0, *Vertex_handle* v1, *Vertex_handle* v2);

Initializes the vertices with $v0$, $v1$, $v2$ and the neighbors with *Face_handle*()).

TriangulationDSFaceBase_2 *f*(*Vertex_handle* *v0*,
 Vertex_handle *v1*,
 Vertex_handle *v2*,
 Face_handle *n0*,
 Face_handle *n1*,
 Face_handle *n2*)

initializes the vertices with *v0*, *v1*, *v2* and the neighbors with
n0, *n1*, *n2*.

Access Functions

<i>int</i>	<i>f.dimension()</i>	returns the dimension.
<i>Vertex_handle</i>	<i>f.vertex(int i)</i>	returns the vertex <i>i</i> of <i>f</i> . Precondition: $0 \leq i \leq 2$.
<i>bool</i>	<i>f.has_vertex(Vertex_handle v)</i>	returns true if <i>v</i> is a vertex of <i>f</i> .
<i>bool</i>	<i>f.has_vertex(Vertex_handle v, int& i)</i>	as above, and sets <i>i</i> to the index of <i>v</i> in <i>f</i> .
<i>int</i>	<i>f.index(Vertex_handle v)</i>	returns the index of <i>v</i> in <i>f</i> .
<i>Face_handle</i>	<i>f.neighbor(int i)</i>	returns the neighbor <i>i</i> of <i>f</i> . Precondition: $0 \leq i \leq 2$.
<i>bool</i>	<i>f.has_neighbor(Face_handle n)</i>	returns true if <i>n</i> is a neighbor of <i>f</i> .
<i>bool</i>	<i>f.has_neighbor(Face_handle n, int& i)</i>	as above, and sets <i>i</i> to the index of <i>n</i> in <i>f</i> .
<i>int</i>	<i>f.index(const Face_handle n)</i>	returns the index of neighbor <i>n</i> in <i>f</i> .

Setting

<i>void</i>	<i>f.set_vertex(int i, Vertex_handle v)</i>	sets vertex <i>i</i> to <i>v</i> . Precondition: $0 \leq i \leq 2$.
<i>void</i>	<i>f.set_vertices()</i>	sets the vertices to <i>Vertex_handle()</i> .
<i>void</i>	<i>f.set_vertices(Vertex_handle v0, Vertex_handle v1, Vertex_handle v2)</i>	sets the vertices.
<i>void</i>	<i>f.set_neighbor(int i, Face_handle n)</i>	sets neighbors <i>i</i> to <i>n</i> . Precondition: $0 \leq i \leq 2$.
<i>void</i>	<i>f.set_neighbors()</i>	sets the neighbors to <i>Face_handle()</i> .
<i>void</i>	<i>f.set_neighbors(Face_handle n0, Face_handle n1, Face_handle n2)</i>	sets the neighbors.

Orientation

<i>void</i>	<i>f.reorient()</i>	Changes the orientation of <i>f</i> by exchanging <i>vertex(0)</i> with <i>vertex(1)</i> and <i>neighbor(0)</i> with <i>neighbor(1)</i> .
<i>void</i>	<i>f.ccw_permute()</i>	performs a counterclockwise permutation of the vertices and neighbors of <i>f</i> .
<i>void</i>	<i>f.cw_permute()</i>	performs a clockwise permutation of the ver- tices and neighbors of <i>f</i> .

Checking

bool *f.is_valid(bool verbose = false)*

performs any required test on a face.
If *verbose* is set to *true*, messages are printed to give a precise indication of the kind of invalidity encountered.

Various

*void** *f.for_compact_container()*
void&* *f.for_compact_container()*

These member functions are required by *Triangulation_data_structure_2* because it uses *Compact_container* to store its faces. See the documentation of *Compact_container* for the exact requirements.

Has Models

CGAL::Triangulation_ds_face_base_2<Tds>
CGAL::Triangulation_face_base_2<Traits,Fb>
CGAL::Regular_triangulation_face_base_2<Traits,Fb>
CGAL::Constrained_triangulation_face_base_2<Traits,Fb>
CGAL::Triangulation_face_base_with_info_2<Info,Traits,Fb>

See Also

TriangulationDSVertexBase_2
TriangulationDataStructure_2::Face
TriangulationFaceBase_2
Triangulation_data_structure_2<Vb,Fb>

TriangulationDataStructure_2::Face

Definition

The concept `TriangulationDataStructure_2::Face` describes the types used to store the faces face class of a *TriangulationDataStructure_2*, see page [1463](#). A `TriangulationDataStructure_2::Face` stores three pointers to its three vertices and three pointers to its three neighbors. The vertices are indexed 0,1, and 2 in counterclockwise order. The neighbor indexed i lies opposite to vertex i .

In degenerate cases, when the triangulation data structure stores a simplicial complex of dimension 0 and 1, the type `TriangulationDataStructure_2::Face` is used to store the faces of maximal dimension of the complex : i.e. a vertex in dimension 0, an edge in dimension 1. Only vertices and neighbors with index 0 are set in the first case, only vertices and neighbors with index 0 or 1 are set in the second case.

Types

The class `TriangulationDataStructure_2::Face` defines the same types as the triangulation data structure except the iterators and the circulators.

Creation

The methods *create_face* and *delete_face()* have to be used to define new faces and to delete non longer used faces.

Vertex Access Functions

<i>Vertex_handle</i>	<i>f.vertex(int i)</i>	returns the vertex i of f . <i>Precondition:</i> $0 \leq i \leq 2$.
<i>int</i>	<i>f.index(Vertex_handle v)</i>	returns the index of vertex v in f . <i>Precondition:</i> v is a vertex of f
<i>bool</i>	<i>f.has_vertex(Vertex_handle v)</i>	returns <i>true</i> if v is a vertex of f .
<i>bool</i>	<i>f.has_vertex(Vertex_handle v, int& i)</i>	returns <i>true</i> if v is a vertex of f , and computes the index i of v in f .

Neighbor Access Functions

The neighbor with index i is the neighbor which is opposite to the vertex with index i .

<i>Face_handle</i>	<i>f.neighbor(int i)</i>	returns the neighbor i of f . <i>Precondition:</i> $0 \leq i \leq 2$.
<i>int</i>	<i>f.index(Face_handle n)</i>	returns the index of face n . <i>Precondition:</i> n is a neighbor of f .
<i>bool</i>	<i>f.has_neighbor(Face_handle n)</i>	returns <i>true</i> if n is a neighbor of f .
<i>bool</i>	<i>f.has_neighbor(Face_handle n, int& i)</i>	returns <i>true</i> if n is a neighbor of f , and compute the index i of n .

Setting

<i>void</i>	<i>f.set_vertex(int i, Vertex_handle v)</i>	sets vertex <i>i</i> to be <i>v</i> . <i>Precondition:</i> $0 \leq i \leq 2$.
<i>void</i>	<i>f.set_neighbor(int i, Face_handle n)</i>	sets neighbor <i>i</i> to be <i>n</i> . <i>Precondition:</i> $0 \leq i \leq 2$.
<i>void</i>	<i>f.set_vertices()</i>	sets the vertices pointers to <i>NULL</i> .
<i>void</i>	<i>f.set_vertices(Vertex_handle v0, Vertex_handle v1, Vertex_handle v2)</i>	sets the vertices pointers.
<i>void</i>	<i>f.set_neighbors()</i>	sets the neighbors pointers to <i>NULL</i> .
<i>void</i>	<i>f.set_neighbors(Face_handle n0, Face_handle n1, Face_handle n2)</i>	sets the neighbors pointers.

Checking

<i>bool</i>	<i>f.is_valid()</i>	returns <i>true</i> if the function <i>is_valid()</i> of the base class returns <i>true</i> and if, for each index <i>i</i> , $0 \leq i < 3$, face <i>f</i> is a neighbor of its neighboring face <i>neighbor(i)</i> and shares with this neighbor the vertices <i>cw(i)</i> and <i>ccw(i)</i> in correct reverse order.
-------------	---------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Miscellaneous

<i>int</i>	<i>f.ccw(int i)</i>	Returns $i + 1$ modulo 3. <i>Precondition:</i> $0 \leq i \leq 2$.
<i>int</i>	<i>f.cw(int i)</i>	Returns $i + 2$ modulo 3. <i>Precondition:</i> $0 \leq i \leq 2$.

I/O

<i>istream&</i>	<i>istream& is >> &f</i>	Inputs any non combinatorial information possibly stored in the face.
<i>ostream&</i>	<i>ostream& os << f</i>	Outputs any non combinatorial information possibly stored in the face.

See Also

TriangulationDataStructure_2,
TriangulationDataStructure_2::Vertex,
TriangulationFaceBase_2.

TriangulationDSVertexBase_2

Definition

The concept `TriangulationDSVertexBase_2` describes the requirements for the vertex base class of a `Triangulation_data_structure_2<Vb,Fb>`.

Note that if the `Triangulation_data_structure_2` is plugged into a triangulation class, the vertex base class has additional geometric requirements depending on the triangulation class.

At the base level, provides access to one of its incident face through a *Face_handle*.

Since the `Triangulation_data_structure_2` is the class which defines the handle types, the vertex base class has to be somehow parameterized by the triangulation data structure. But since the `Triangulation_data_structure_2` itself is parameterized by the face and vertex base classes, there is a cycle in the definition of these classes. In order to break the cycle, the base classes for faces and vertices which are plugged in to instantiate a `Triangulation_data_structure_2` use a *void* as triangulation data structure parameter. Then, the `Triangulation_data_structure_2` uses a *rebind* mechanism (similar to the one specified in `std::allocator`) in order to plug itself as parameter in the face and vertex base classes. This mechanism requires that the base class provides a templated nested class *Rebind_TDS* that itself provides the subtype *Rebind_TDS<TDS2>::Other* which is the *rebound* version of the base class. This *rebound* base class is the class that the `Triangulation_data_structure_2` actually uses as a base class for the class of `Triangulation_data_structure_2::Vertex`.

Refines

`TriangulationDataStructure_2::Vertex`

Types

The concept `TriangulationDSVertexBase_2` has to provide the following types.

`TriangulationDSVertexBase_2::template <typename TDS2> struct Rebind_TDS;`

This nested template class has to define a type *Other* which is the *rebound* vertex base , where the actual `Triangulation_data_structure_2` is plugged in. This type *Other* will be the actual base of the class `Triangulation_data_structure_2::Vertex`.

<code>typedef TriangulationDataStructure_2</code>	<code>Triangulation_data_structure;</code>
<code>typedef TriangulationDataStructure_2::Vertex_handle</code>	<code>Vertex_handle;</code>
<code>typedef TriangulationDataStructure_2::Face_handle</code>	<code>Face_handle;</code>

Creation

<code>TriangulationDSVertexBase_2 v;</code>	default constructor.
<code>TriangulationDSVertexBase_2 v(Face_handle f);</code>	Constructs a vertex pointing to face <i>f</i> .

Various

```
void*          v.for_compact_container()
void*&         v.for_compact_container()
```

These member functions are required by *Triangulation_data_structure_2* because it uses *Compact_container* to store its faces. See the documentation of *Compact_container* for the exact requirements.

Has Models

```
CGAL::Triangulation_ds_vertex_base_2<Tds>
CGAL::Triangulation_vertex_base_2<Traits,Vb>
CGAL::Regular_triangulation_vertex_base_2<Traits,Vb>
CGAL::Triangulation_hierarchy_vertex_base_2<Vb>
CGAL::Triangulation_vertex_base_with_info_2<Info,Traits,vb>
```

See Also

```
TriangulationVertexBase_2
TriangulationDSFaceBase_2
TriangulationFaceBase_2
TriangulationDataStructure_2::Vertex
Triangulation_data_structure_2<Vb,Fb>
```

TriangulationDataStructure_2::Vertex

Definition

The concept `TriangulationDataStructure_2::Vertex` describes the type used by a *TriangulationDataStructure_2* to store the vertices, see page [1463](#).

Some of the requirements listed below are of geometric nature and are *optional* when using the triangulation data structure class alone. They became required when the triangulation data structure is plugged into a triangulation.

Types

The class `TriangulationDataStructure_2::Vertex` defines the same types as the triangulation data structure except the iterators.

TriangulationDataStructure_2::Vertex::Point *Optional for the triangulation data structure used alone.*

Creation

In order to obtain new vertices or destruct unused vertices, the user must call the *create_vertex()* and *delete_vertex()* methods of the triangulation data structure.

Access Functions

<i>Point</i>	<i>v.point()</i>	returns the geometric information of <i>v</i> .
<i>Face_handle</i>	<i>v.face()</i>	returns a face of the triangulation having <i>v</i> as vertex.

————— *advanced* —————

Setting

<i>void</i>	<i>v.set_point(Point p)</i>	sets the geometric information to <i>p</i> .
<i>void</i>	<i>v.set_face(Face_handle f)</i>	sets the incident face to <i>f</i> .

————— *advanced* —————

Checking

<i>bool</i>	<i>v.is_valid(bool verbose = false)</i>
-------------	-------------------------------------------

Checks the validity of the vertex. Must check that its incident face has this vertex. The validity of the base vertex is also checked. When *verbose* is set to *true*, messages are printed to give a precise indication on the kind of invalidity encountered.

I/O

istream& *istream*& *is* >> & *v* Inputs the non-combinatorial information possibly stored in the vertex.

ostream& *ostream*& *os* << *v* Outputs the non combinatorial operation possibly stored in the vertex.

Has Models

CGAL::Triangulation_ds_vertex_2<*Vb*,*Fb*>

See Also

TriangulationDataStructure_2

TriangulationDataStructure_2::Face

CGAL::Triangulation_data_structure_2<Vb,Fb>

Definition

The class *Triangulation_data_structure_2<Vb,Fb>* is a model for the *TriangulationDataStructure_2* concept. It can be used to represent an orientable 2D triangulation embedded in a space of any dimension.

```
#include <CGAL/Triangulation_data_structure_2.h>
```

Is Model for the Concepts

TriangulationDataStructure_2

Modifiers

In addition to the modifiers required by the *TriangulationDataStructure_2* concept, the *Triangulation_data_structure_2<Vb,Fb>* class supports also the modifiers below. Note also that the modifiers below guarantee the combinatorial validity of the resulting data structure.

<i>Vertex_handle</i>	<i>tds.join_vertices(Face_handle f, int i)</i>	Joins the vertices that are endpoints of the edge (f,i) . It returns a vertex handle to common vertex (see Fig. 21.6). <i>Precondition:</i> f must be different from <i>Face_handle()</i> and i must be 0, 1 or 2.
<i>Vertex_handle</i>	<i>tds.join_vertices(Edge e)</i>	Joins the vertices that are endpoints of the edge e . It returns a vertex handle to common vertex.
<i>Vertex_handle</i>	<i>tds.join_vertices(Edge_iterator eit)</i>	Joins the vertices that are endpoints of the edge $*eit$. It returns a vertex handle to common vertex.
<i>Vertex_handle</i>	<i>tds.join_vertices(Edges_circulator ec)</i>	Joins the vertices that are endpoints of the edge $*ec$. It returns a vertex handle to common vertex.

```
boost::tuples::tuple<Vertex_handle, Vertex_handle, Face_handle, Face_handle>
```

```
tds.split_vertex( Vertex_handle v, Face_handle f1, Face_handle f2 )
```

Splits the vertex v into two vertices $v1$ and $v2$. The common faces f and g of $v1$ and $v2$ are created after (in the counter-clockwise sense) the faces $f1$ and $f2$. The 4-tuple $(v1,v2,f,g)$ is returned (see Fig. 21.6).
Precondition: *dimension()* must be equal to 2, $f1$ and $f2$ must be different from *Face_handle()* and v must be a vertex of both $f1$ and $f2$.

<i>Vertex_handle</i>	<i>tds.insert_degree_2(Face_handle f, int i)</i>	Inserts a degree two vertex and two faces adjacent to it that have two common edges. The edge defined by the face handle f and the integer i is duplicated. It returns a handle to the vertex created (see Fig. 21.7).
----------------------	----------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

`void tds.remove_degree_2(Vertex_handle v)`

Removes a degree 2 vertex and the two faces adjacent to it. The two edges of the star of v that are not incident to it are collapsed (see Fig. 21.7).

Precondition: The degree of v must be equal to 2.

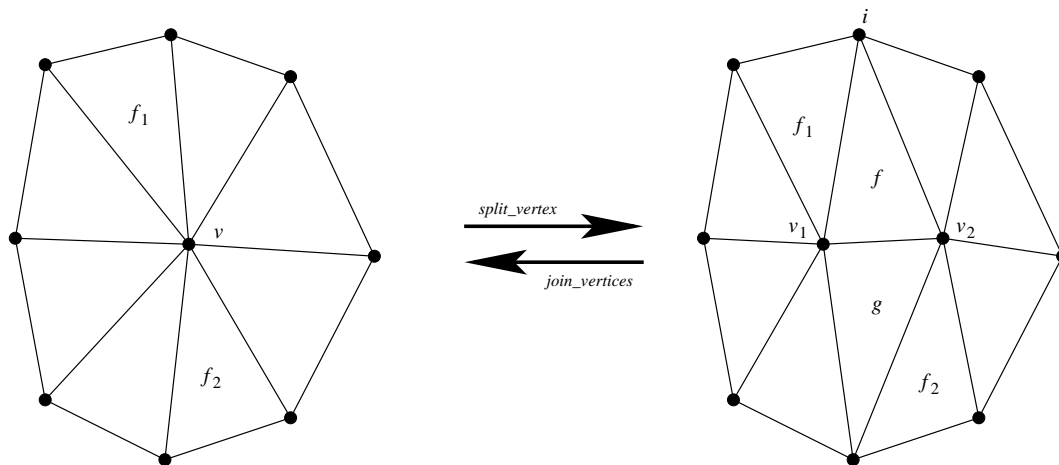


Figure 21.6: The join and split operations.

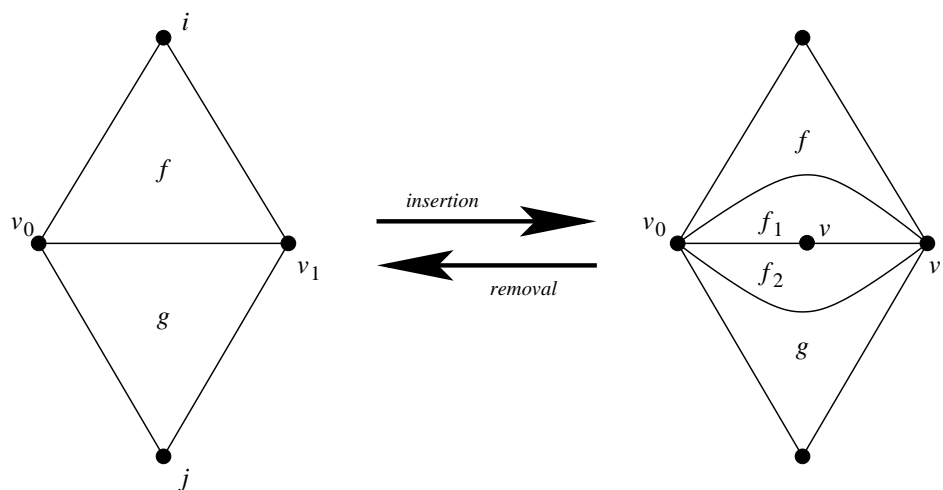


Figure 21.7: Insertion and removal of degree 2 vertices.

CGAL::Triangulation_ds_face_base_2<Tds>

Definition

The class *Triangulation_ds_face_base_2<Tds>* is a model for the concept *TriangulationDSFaceBase_2* to be used by *Triangulation_data_structure_2*.

```
#include <CGAL/Triangulation_ds_face_base_2.h>
```

Is Model for the Concepts

TriangulationDSFaceBase_2

See Also

CGAL::Triangulation_face_base_2<Traits,Fb>

CGAL::Triangulation_ds_vertex_base_2<Tds>

CGAL::Triangulation_ds_vertex_base_2<Tds>

Definition

The class *Triangulation_ds_vertex_base_2* can be used as the base vertex for a *Triangulation_data_structure_2*, it is a model of the concept *TriangulationDSVertexBase_2*.

This base class can be used directly or can serve as a base to derive other base classes with some additional attributes (a color for example) tuned for a specific application.

Note that if the *Triangulation_data_structure_2* is used as a parameter of a geometric triangulation, there are additional geometric requirements to be fulfilled by the vertex base class, and *Triangulation_ds_vertex_base_2* cannot be plugged in.

```
#include <CGAL/Triangulation_ds_vertex_base_2.h>
```

Is Model for the Concepts

TriangulationDSVertexBase_2

See Also

CGAL::Triangulation_vertex_base_2<Traits,Vb>

CGAL::Triangulation_ds_face_base_2<Tds>

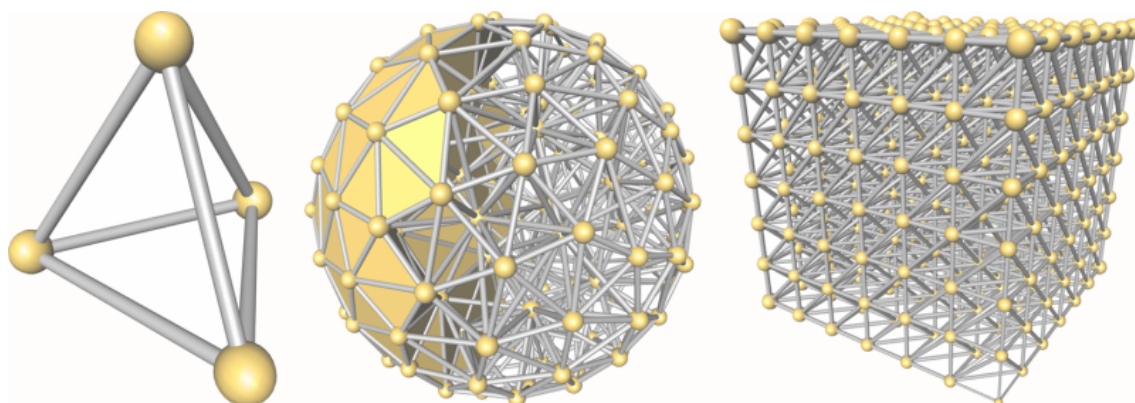
Chapter 22

3D Triangulations

Sylvain Pion and Monique Teillaud

Contents

22.1 Representation	1486
22.2 Delaunay Triangulation	1487
22.3 Regular Triangulation	1488
22.4 Triangulation Hierarchy	1488
22.5 Software Design	1489
22.5.1 The Geometric Traits Parameter	1489
22.5.2 The Triangulation Data Structure Parameter	1490
22.5.3 Flexibility of the Design	1490
22.5.4 Backward compatibility	1491
22.6 Examples	1493
22.6.1 Basic example	1493
22.6.2 Changing the vertex base	1494
22.6.3 Use of the Delaunay hierarchy	1497
22.6.4 Finding the cells in conflict with a point in a Delaunay triangulation	1498
22.6.5 Regular triangulation	1499
22.7 Design and Implementation History	1500



The basic 3D-triangulation class of CGAL is primarily designed to represent the triangulations of a set of points A in \mathbb{R}^3 . It is a partition of the convex hull of A into tetrahedra whose vertices are the points of A . Together with

the unbounded cell having the convex hull boundary as its frontier, the triangulation forms a partition of \mathbb{R}^3 . Its cells (3-faces) are such that two cells either do not intersect or share a common facet (2-face), edge (1-face) or vertex (0-face).

22.1 Representation

In order to deal only with tetrahedra, which is convenient for many applications, the unbounded cell can be subdivided into tetrahedra by considering that each convex hull facet is incident to an *infinite cell* having as fourth vertex an auxiliary vertex called the *infinite vertex*. In that way, each facet is incident to exactly two cells and special cases at the boundary of the convex hull are simple to deal with.

The class `Triangulation_3<TriangulationTraits_3, TriangulationDataStructure_3>` of CGAL implements this point of view and therefore considers the triangulation of the set of points as a set of finite and infinite tetrahedra. Notice that the infinite vertex has no significant coordinates and that no geometric predicate can be applied on it.

A triangulation is a collection of vertices and cells that are linked together through incidence and adjacency relations. Each cell gives access to its four incident vertices and to its four adjacent cells. Each vertex gives access to one of its incident cells.

The four vertices of a cell are indexed with 0, 1, 2 and 3 in positive orientation, the positive orientation being defined by the orientation of the underlying Euclidean space \mathbb{R}^3 (see Figure 22.1). The neighbors of a cell are also indexed with 0, 1, 2, 3 in such a way that the neighbor indexed by i is opposite to the vertex with the same index.

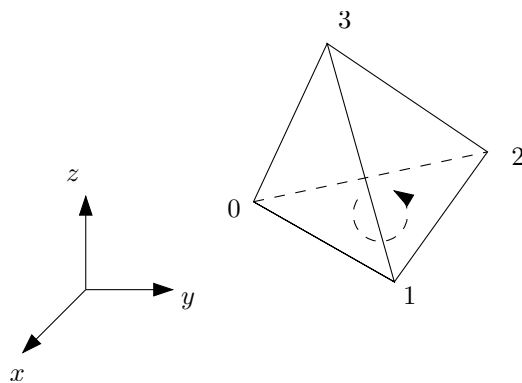


Figure 22.1: Orientation of a cell (3-dimensional case).

As in the underlying combinatorial triangulation (see Chapter 23), edges (1-faces) and facets (2-faces) are not explicitly represented: a facet is given by a cell and an index (the facet i of a cell c is the facet of c that is opposite to the vertex with index i) and an edge is given by a cell and two indices (the edge (i,j) of a cell c is the edge whose endpoints are the vertices of c with indices i and j). See Figure 23.1.

Degenerate Dimensions The class `Triangulation_3` can also deal with triangulations whose dimension d is less than 3. A triangulation of a set of points in \mathbb{R}^d covers the whole space \mathbb{R}^d and consists of cells having $d + 1$ vertices: some of them are infinite, they are obtained by linking the additional infinite vertex to each facet of the convex hull of the points.

- *dimension 2*: when a triangulation only contains coplanar points (which is the case when there are only three points), it consists of triangular faces.
- *dimension 1*: the triangulation contains only collinear points (which is the case when there are only two points), it consists of edges.
- *dimension 0*: the triangulation contains only one finite point.
- *dimension -1*: this is a convention to handle the case when the only vertex of the triangulation is the infinite one.

The same cell class is used in all cases: triangular faces in 2D can be considered as degenerate cells, having only three vertices (resp. neighbors) numbered (0, 1, 2); edges in 1D have only two vertices (resp. neighbors) numbered 0 and 1.

The implicit representation of facets (resp. edges) still holds for degenerate dimensions (*i.e.* dimensions < 3): in dimension 2, each cell has only one facet of index 3, and 3 edges (0, 1), (1, 2) and (2, 0); in dimension 1, each cell has one edge (0, 1).

Validity A triangulation of \mathbb{R}^3 is said to be *locally valid* iff

- (a)-(b) Its underlying combinatorial graph, the triangulation data structure, is *locally valid* (see Section 23.1 of Chapter 23)
- (c) Any cell has its vertices ordered according to positive orientation. See Figure 22.1.

When the triangulation is degenerated into a triangulation of dimension 2, the geometric validity reduces to:

(c-2D) For any two adjacent triangles (u, v, w_1) and (u, v, w_2) with common edge (u, v) , w_1 and w_2 lie on opposite sides of (u, v) in the plane.

When all the points are collinear, this condition becomes:

(c-1D) For any two adjacent edges (u, v) and (v, w) , u and w lie on opposite sides of the common vertex v on the line.

The *is_valid()* method provided in *Triangulation_3* checks the local validity of a given triangulation. This does not always ensure global validity [MNS⁺96, DLPT98] but it is sufficient for practical cases.

22.2 Delaunay Triangulation

The class *Delaunay_triangulation_3* represents a three-dimensional Delaunay triangulation.

Delaunay triangulations have the specific *empty sphere property*, that is, the circumscribing sphere of each cell of such a triangulation does not contain any other vertex of the triangulation in its interior. These triangulations are uniquely defined except in degenerate cases where five points are cospherical. Note however that the CGAL implementation computes a unique triangulation even in these cases.

This implementation is fully dynamic: it supports both insertions of points and vertex removal. The user is advised to use the class *Triangulation_hierarchy_3* in order to benefit from an increased efficiency for large data sets.

22.3 Regular Triangulation

The class *Regular_triangulation_3* implements incremental regular triangulations, also known as weighted Delaunay triangulations.

Let $S^{(w)}$ be a set of weighted points in \mathbb{R}^3 . Let $p^{(w)} = (p, w_p)$, $p \in \mathbb{R}^3$, $w_p \in \mathbb{R}$ and $z^{(w)} = (z, w_z)$, $z \in \mathbb{R}^3$, $w_z \in \mathbb{R}$ be two weighted points. A weighted point $p^{(w)} = (p, w_p)$ can also be seen as a sphere of center p and radius w_p . The *power product* between $p^{(w)}$ and $z^{(w)}$ is defined as

$$\Pi(p^{(w)}, z^{(w)}) = \|p - z\|^2 - w_p - w_z$$

where $\|p - z\|$ is the Euclidean distance between p and z . $p^{(w)}$ and $z^{(w)}$ are said to be *orthogonal* iff $\Pi(p^{(w)}, z^{(w)}) = 0$ (see Figure 22.2).

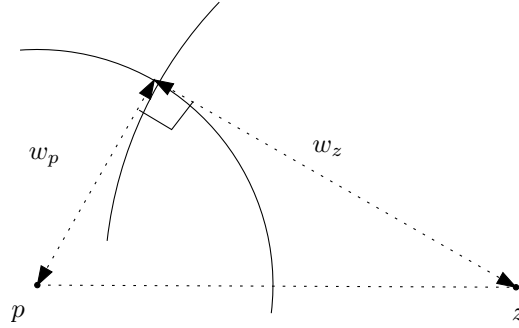


Figure 22.2: Orthogonal weighted points (picture in 2D).

Four weighted points have a unique common orthogonal weighted point called the *power sphere*. The weighted point orthogonal to three weighted points in the plane defined by these three points is called the *power circle*. The *power segment* will denote the weighted point orthogonal to two weighted points on the line defined by these two points.

A sphere $z^{(w)}$ is said to be *regular* if $\forall p^{(w)} \in S^{(w)}, \Pi(p^{(w)}, z^{(w)}) \geq 0$.

A triangulation of $S^{(w)}$ is *regular* if the power spheres of all simplices are regular.

The regular triangulation of $S^{(w)}$ is in fact the projection onto \mathbb{R}^3 of the convex hull of the four-dimensional points $(p, \|p - O\|^2 - w_p)$, for $p^{(w)} = (p, w_p) \in S^{(w)}$. Note that all points of $S^{(w)}$ do not necessarily appear as vertices of the regular triangulation. To know more about regular triangulations, see for example [ES96].

When all weights are 0, power spheres are nothing more than circumscribing spheres, and the regular triangulation is exactly the Delaunay triangulation.

22.4 Triangulation Hierarchy

The class *Triangulation_hierarchy_3* implements a triangulation augmented with a data structure that allows fast point location queries. Thus, it allows fast construction of the triangulation. As proved in [Dev02], this structure has an optimal behavior when it is built for Delaunay triangulations.

Note that, since the algorithms that are provided are randomized, the running time of constructing a triangulation with a hierarchy may be improved when shuffling the data points.

22.5 Software Design

The main classes *Triangulation_3*, *Delaunay_triangulation_3* and *Regular_triangulation_3* are connected to each other by the derivation diagram shown in Figure 22.3. This diagram also shows two other classes: *Triangulation_utils_3* (page 1597), which provides a set of tools operating on the indices of vertices in cells, and *Triangulation_hierarchy_3*, which implements a hierarchy of triangulations suitable for speeding up point location.

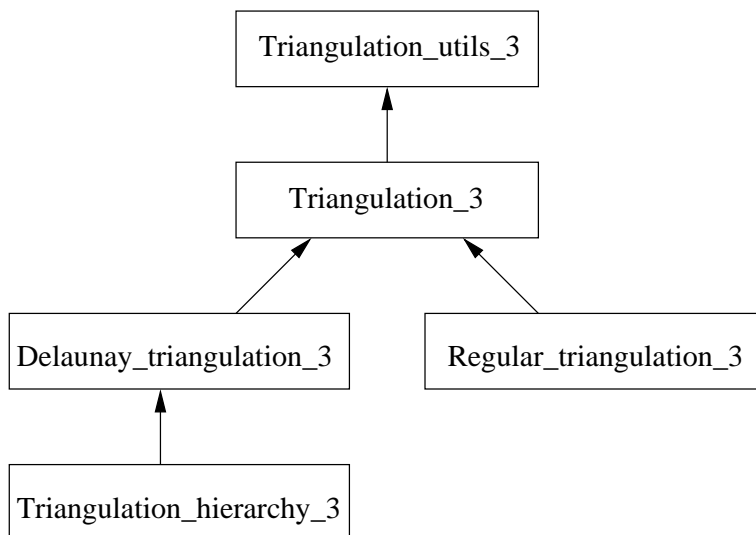


Figure 22.3: Derivation diagram of the 3D triangulation classes.

The three main classes (*Triangulation_3*, *Delaunay_triangulation_3* and *Regular_triangulation_3*) provide high-level geometric functionality such as location of a point in the triangulation [DPT02], insertion and possibly removal of a point [DT03], and are responsible for the geometric validity. They are built as layers on top of a triangulation data structure, which stores their combinatorial structure. This separation between the geometry and the combinatorics is reflected in the software design by the fact that these three triangulation classes take two template parameters :

- the **geometric traits** class, which provides the type of points to use as well as the elementary operations on them (predicates and constructions). The concepts for these parameters are described in more details in Section 22.5.1 and in page 1534.
- the **triangulation data structure** class, which stores their combinatorial structure, described in Section 23.2 of Chapter 23.

The class *Triangulation_hierarchy_3* is parameterized by a class, which at the moment can only be *Delaunay_triangulation_3*. It fetches its geometric traits from this parameter directly.

22.5.1 The Geometric Traits Parameter

The first template parameter of the triangulation class *Triangulation_3*<*TriangulationTraits_3*, *TriangulationDataStructure_3*> is the geometric traits class, described by the concept *TriangulationTraits_3*. It must define the types of the geometric objects (points, segments, triangles and tetrahedra) forming the

triangulation together with a few geometric predicates on these objects: orientation in space, orientation in case of coplanar points, order of collinear points.

In addition to the requirements described before, the geometric traits class of *Delaunay_triangulation_3* must define predicates to test for the *empty sphere property*. It is described by the concept *DelaunayTriangulationTraits_3*, which refines *TriangulationTraits_3*.

The kernels provided by CGAL: *Cartesian*, *Homogeneous*, *Simple_cartesian*, *Simple_homogeneous* and *Filtered_kernel* can all be used as models for the geometric traits parameter. They supply the user with all the functionalities described for the concepts *TriangulationTraits_3* (page 1534) and *DelaunayTriangulationTraits_3* (page 1536). In addition, the predefined kernels *Exact_predicates_inexact_constructions_kernel* (page ??) and *Exact_predicates_exact_constructions_kernel* (page ??) can also be used, the later being recommended when the dual construction is used.

In order to be used as the traits class for *Regular_triangulation_3*, a class must provide functions to compute the *power tests* (see Section 22.3). *Regular_triangulation_euclidean_traits_3<K,Weight>* is a traits class designed to be used by the class *Regular_triangulation_3<RegularTriangulationTraits_3, TriangulationDataStructure_3>*. It provides *Weighted_point*, a class for weighted points needed by the regular triangulation, which derives from the three dimensional point class *K::Point_3*. It supplies the user with all the functionalities described for the concept *RegularTriangulationTraits_3* (page 1539). It can be used as a traits class for *Regular_triangulation_3<RegularTriangulationTraits_3, TriangulationDataStructure_3>*.

Note that for regular triangulations, plugging a filtered kernel such as *Exact_predicates_inexact_constructions_kernel* or *Exact_predicates_exact_constructions_kernel* in *Regular_triangulation_euclidean_traits_3<K,Weight>* will in fact not provide exact predicates since the weighted points and the predicates on them are not defined in the CGAL kernels. To solve this, there is also another model of the traits concept, *Regular_triangulation_filtered_traits_3<FK>*, which is providing filtered predicates (exact and efficient). The argument *FK* must be a model of the *Kernel* concept, and it is also restricted to be a instance of the *Filtered_kernel* template.

22.5.2 The Triangulation Data Structure Parameter

The second template parameter of the main classes (*Triangulation_3*, *Delaunay_triangulation_3* and *Regular_triangulation_3*) is a triangulation data structure class. This class can be seen as a container for the cells and vertices maintaining incidence and adjacency relations (see Chapter 23). A model of this triangulation data structure is *Triangulation_data_structure_3* (page 1594), and it is described by the *TriangulationDataStructure_3* concept (page 1573). This model is itself parameterized by a vertex base and a cell base classes, which gives the possibility to customize the vertices and cells used by the triangulation data structure, and hence by the geometric triangulation using it. Depending on the kind of triangulation used, the requirements on the vertex and cell base classes vary, and are expressed by various concepts, following the refinement diagram shown in Figure 22.4.

A default value for the triangulation data structure parameter is provided in all the triangulation classes, so it need not be specified by the user unless he wants to use a different triangulation data structure or a different vertex or cell base class.

22.5.3 Flexibility of the Design

In order to satisfy as many uses as possible, a design has been selected that allows to exchange different parts to meet the users' needs, while still re-using a maximum of the provided functionalities. We have already seen that the main triangulation classes are parameterized by a geometric traits class and a triangulation data structure (TDS), so that each of them can be interchanged with alternate implementations.

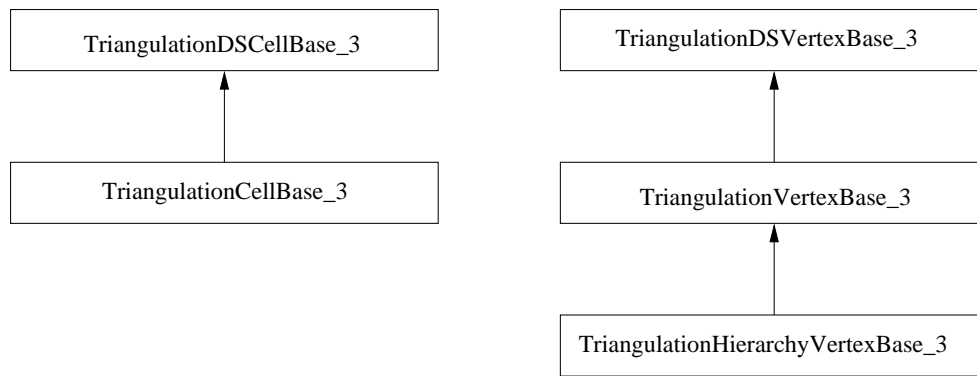


Figure 22.4: Concepts refinement hierarchy for the vertex and cell base classes parameters.

The most useful flexibility is the ability given to the user to add his own data in the vertices and cells by providing his own vertex and cell base classes to *Triangulation_data_structure_3*. The Figure 22.5 shows in more detail the flexibility that is provided, and the place where the user can insert his own vertex and/or cell base classes.

22.5.4 Backward compatibility

Starting with CGAL release 3.0, the design of the triangulation data structure has been changed in order to give the possibility to store handles (an entity akin to pointers) directly in the vertex and cell base classes. Previously, `void*` pointers were stored there instead, and later converted internally to handles, but this happened to be too restrictive for some uses.

The difference is visible to the user when he provides his own vertex or cell base class. Previously, something like the following had to be written:

```

...
template < class GT >
class My_vertex
  : public Triangulation_vertex_base<GT>
{
  typedef Triangulation_vertex_base<GT> Vb;
public:
  typedef typename Vb::Point          Point;

  My_vertex() {}
  My_vertex(const Point&p)              : Vb(p) {}
  My_vertex(const Point&p, void *c)    : Vb(p, c) {}
  ...
};

typedef Cartesian<double>                               GT;
typedef Triangulation_data_structure_3<My_vertex<GT>,
                                     Triangulation_cell_base_3<GT> > My_TDS;
typedef Triangulation_3<GT, My_TDS>                               Tr;
...

```

While now, there are three possibilities. The simplest one is to use the class *Triangulation_vertex_base_with_*

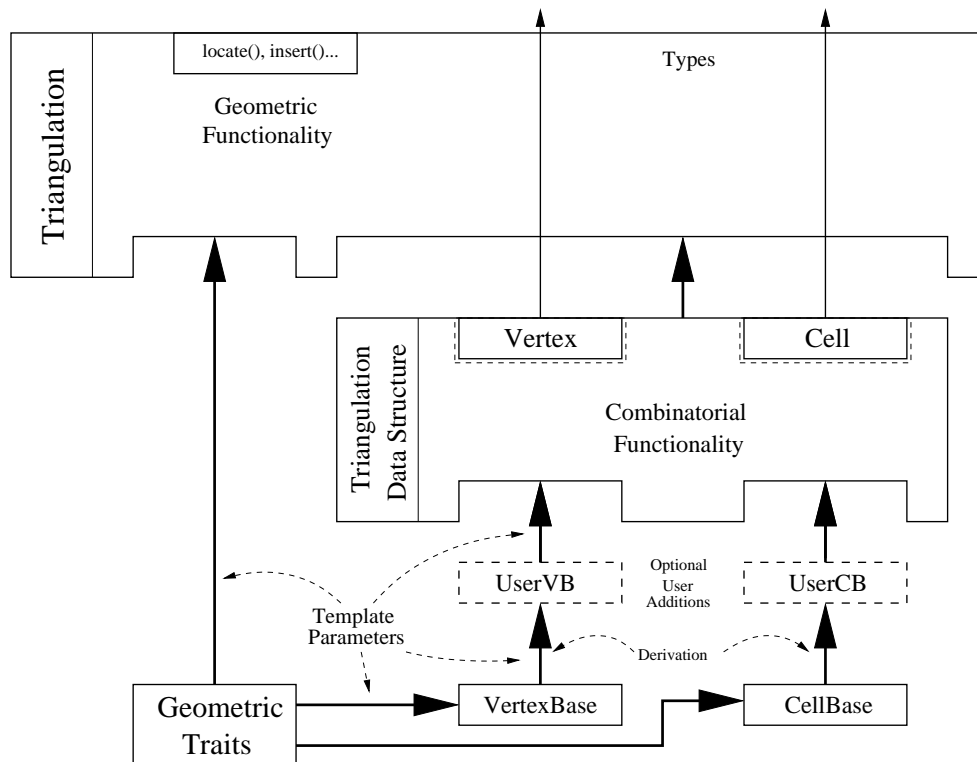


Figure 22.5: Triangulation software design.

info_3, and this approach is illustrated in a following subsection 22.6.2. The most complicated one, and probably useless for almost all cases, is to write a vertex base class from scratch, following the documented requirements. This is mostly useless because most of the time it is enough to derive from the models that CGAL provides, and add the desired features. In this case, when the user needs to access some type that depends on the triangulation data structure (typically handles), then he should write something like:

```

...
template < class GT, class Vb = Triangulation_vertex_base<GT> >
class My_vertex
: public Vb
{
public:
    typedef typename Vb::Point          Point;
    typedef typename Vb::Cell_handle    Cell_handle;

    template < class TDS2 >
    struct Rebind_TDS {
        typedef typename Vb::template Rebind_TDS<TDS2>::Other Vb2;
        typedef My_vertex<GT, Vb2>                               Other;
    };

    My_vertex() {}
    My_vertex(const Point&p) : Vb(p) {}
    My_vertex(const Point&p, Cell_handle c) : Vb(p, c) {}
...
};

```

```
... // The rest has not changed
```

The changes that need to be made are the following:

- *My_vertex* is now parameterized by the vertex base class it derives from.
- a nested template class *Rebind_TDS* must be defined.
- `void*` must be changed to *Cell_handle*, and you need to extract the *Cell_handle* type from the vertex base class that *My_vertex* derives from.

The situation is exactly similar for cell base classes. Section [23.2](#) provides more detailed information.

22.6 Examples

22.6.1 Basic example

This example shows the incremental construction of a 3D triangulation, the location of a point and how to perform elementary operations on indices in a cell. It uses the default parameter of the *Triangulation_3* class.

```
// examples/Triangulation_3/example_simple.C

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Triangulation_3.h>

#include <iostream>
#include <fstream>
#include <cassert>
#include <list>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Triangulation_3<K>      Triangulation;

typedef Triangulation::Cell_handle    Cell_handle;
typedef Triangulation::Vertex_handle Vertex_handle;
typedef Triangulation::Locate_type    Locate_type;
typedef Triangulation::Point          Point;

int main()
{
    // construction from a list of points :
    std::list<Point> L;
    L.push_front(Point(0,0,0));
    L.push_front(Point(1,0,0));
    L.push_front(Point(0,1,0));

    Triangulation T(L.begin(), L.end());
```

```

int n = T.number_of_vertices();

// insertion from a vector :
std::vector<Point> V(3);
V[0] = Point(0,0,1);
V[1] = Point(1,1,1);
V[2] = Point(2,2,2);

n = n + T.insert(V.begin(), V.end());

assert( n == 6 );           // 6 points have been inserted
assert( T.is_valid() );    // checking validity of T

Locate_type lt;
int li, lj;
Point p(0,0,0);
Cell_handle c = T.locate(p, lt, li, lj);
// p is the vertex of c of index li :
assert( lt == Triangulation::VERTEX );
assert( c->vertex(li)->point() == p );

Vertex_handle v = c->vertex( (li+1)&3 );
// v is another vertex of c
Cell_handle nc = c->neighbor(li);
// nc = neighbor of c opposite to the vertex associated with p
// nc must have vertex v :
int nli;
assert( nc->has_vertex( v, nli ) );
// nli is the index of v in nc

std::ofstream outFileT("output",std::ios::out);
// writing file output;
outFileT << T;

Triangulation Tl;
std::ifstream inFileT("output",std::ios::in);
// reading file output;
inFileT >> Tl;
assert( Tl.is_valid() );
assert( Tl.number_of_vertices() == T.number_of_vertices() );
assert( Tl.number_of_cells() == T.number_of_cells() );

return 0;
}

```

22.6.2 Changing the vertex base

The following two examples show how the user can plug his own vertex base in a triangulation. Changing the cell base is similar.

Adding a color

When the user doesn't need to add a type in a vertex which depends on the *TriangulationDataStructure_3* (e.g. a *Vertex_handle* or *Cell_handle*), then he can use the *Triangulation_vertex_base_with_info_3* class to add his own information easily in the vertices. The example below shows how to add a *CGAL::Color* this way.

```
// file: examples/Triangulation_3/example_color.C

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_3.h>
#include <CGAL/Triangulation_vertex_base_with_info_3.h>
#include <CGAL/IO/Color.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Triangulation_vertex_base_with_info_3<CGAL::Color, K> Vb;
typedef CGAL::Triangulation_data_structure_3<Vb> Tds;
typedef CGAL::Delaunay_triangulation_3<K, Tds> Delaunay;

typedef Delaunay::Point Point;

int main()
{
    Delaunay T;

    T.insert(Point(0,0,0));
    T.insert(Point(1,0,0));
    T.insert(Point(0,1,0));
    T.insert(Point(0,0,1));
    T.insert(Point(2,2,2));
    T.insert(Point(-1,0,1));

    // Set the color of finite vertices of degree 6 to red.
    Delaunay::Finite_vertices_iterator vit;
    for (vit = T.finite_vertices_begin(); vit != T.finite_vertices_end(); ++vit)
        if (T.degree(vit) == 6)
            vit->info() = CGAL::RED;

    return 0;
}
```

Adding handles

When the user needs to add a type in a vertex which depends on the *TriangulationDataStructure_3* (e.g. a *Vertex_handle* or *Cell_handle*), then he has to derive his own vertex base class, as the following example shows.

```
// file: examples/Triangulation_3/example_adding_handles.C

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_3.h>
#include <CGAL/Triangulation_vertex_base_3.h>
```

```

template < class GT, class Vb = CGAL::Triangulation_vertex_base_3<GT> >
class My_vertex_base
    : public Vb
{
public:
    typedef typename Vb::Vertex_handle    Vertex_handle;
    typedef typename Vb::Cell_handle      Cell_handle;
    typedef typename Vb::Point            Point;

    template < class TDS2 >
    struct Rebind_TDS {
        typedef typename Vb::template Rebind_TDS<TDS2>::Other    Vb2;
        typedef My_vertex_base<GT, Vb2>                          Other;
    };

    My_vertex_base() {}

    My_vertex_base(const Point& p)
        : Vb(p) {}

    My_vertex_base(const Point& p, Cell_handle c)
        : Vb(p, c) {}

    Vertex_handle    vh;
    Cell_handle      ch;
};

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Triangulation_data_structure_3<My_vertex_base<K> >    Tds;
typedef CGAL::Delaunay_triangulation_3<K, Tds>                      Delaunay;

typedef Delaunay::Vertex_handle    Vertex_handle;
typedef Delaunay::Point            Point;

int main()
{
    Delaunay T;

    Vertex_handle v0 = T.insert(Point(0,0,0));
    Vertex_handle v1 = T.insert(Point(1,0,0));
    Vertex_handle v2 = T.insert(Point(0,1,0));
    Vertex_handle v3 = T.insert(Point(0,0,1));
    Vertex_handle v4 = T.insert(Point(2,2,2));
    Vertex_handle v5 = T.insert(Point(-1,0,1));

    // Now we can link the vertices as we like.
    v0->vh = v1;
    v1->vh = v2;
    v2->vh = v3;
    v3->vh = v4;
    v4->vh = v5;

```



```

    v5->vh = v0;

    return 0;
}

```

22.6.3 Use of the Delaunay hierarchy

```

// file: examples/Triangulation_3/example_hierarchy.C

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_3.h>
#include <CGAL/Triangulation_hierarchy_3.h>

#include <cassert>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Triangulation_vertex_base_3<K> Vb;
typedef CGAL::Triangulation_hierarchy_vertex_base_3<Vb> Vbh;
typedef CGAL::Triangulation_data_structure_3<Vbh> Tds;
typedef CGAL::Delaunay_triangulation_3<K,Tds> Dt;
typedef CGAL::Triangulation_hierarchy_3<Dt> Dh;

typedef Dh::Vertex_iterator Vertex_iterator;
typedef Dh::Vertex_handle Vertex_handle;
typedef Dh::Point Point;

int main()
{
    Dh T;

    // insertion of points on a 3D grid
    std::vector<Vertex_handle> V;

    for (int z=0 ; z<5 ; z++)
        for (int y=0 ; y<5 ; y++)
            for (int x=0 ; x<5 ; x++)
                V.push_back(T.insert(Point(x,y,z)));

    assert( T.is_valid() );
    assert( T.number_of_vertices() == 125 );
    assert( T.dimension() == 3 );

    // removal of the vertices in random order
    std::random_shuffle(V.begin(), V.end());

    for (int i=0; i<125; ++i)
        T.remove(V[i]);

    assert( T.is_valid() );
    assert( T.number_of_vertices() == 0 );
}

```

```

    return 0;
}

```

22.6.4 Finding the cells in conflict with a point in a Delaunay triangulation

```

// file: examples/Triangulation_3/example_find_conflicts.C

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_3.h>
#include <CGAL/point_generators_3.h>

#include <vector>
#include <cassert>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Delaunay_triangulation_3<K>          Delaunay;
typedef Delaunay::Point                             Point;
typedef Delaunay::Cell_handle                       Cell_handle;
typedef Delaunay::Facet                             Facet;

int main()
{
    Delaunay T;
    CGAL::Random_points_in_sphere_3<Point> rnd;

    // First, make sure the triangulation is 3D.
    T.insert(Point(0,0,0));
    T.insert(Point(1,0,0));
    T.insert(Point(0,1,0));
    T.insert(Point(0,0,1));

    assert(T.dimension() == 3);

    // Inserts 100 random points if and only if their insertion
    // in the Delaunay tetrahedralization conflicts with
    // an even number of cells.
    for (int i = 0; i != 100; ++i) {
        Point p = *rnd++;

        // Locate the point
        Delaunay::Locate_type lt;
        int li, lj;
        Cell_handle c = T.locate(p, lt, li, lj);
        if (lt == Delaunay::VERTEX)
            continue; // Point already exists

        // Get the cells that conflict with p in a vector V,
        // and a facet on the boundary of this hole in f.
        std::vector<Cell_handle> V;
        Facet f;

        T.find_conflicts(p, c,

```

```

        CGAL::Oneset_iterator<Facet>(f), // Get one boundary facet
        std::back_inserter(V));        // Conflict cells in V

    if ((V.size() & 1) == 0) // Even number of conflict cells ?
        T.insert_in_hole(p, V.begin(), V.end(), f.first, f.second);
}

std::cout << "Final triangulation has " << T.number_of_vertices()
          << " vertices." << std::endl;

return 0;
}

```

22.6.5 Regular triangulation

This example shows the building of a regular triangulation. In this triangulation, points have an associated weight, and some points can be hidden and do not result in vertices in the triangulation. Another difference is that a specific traits class has to be used (at least at the moment).

```

// file: examples/Triangulation_3/example_regular.C

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Regular_triangulation_3.h>
#include <CGAL/Regular_triangulation_euclidean_traits_3.h>
#include <CGAL/Regular_triangulation_filtered_traits_3.h>
#include <cassert>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Regular_triangulation_filtered_traits_3<K> Traits;

typedef Traits::RT Weight;
typedef Traits::Bare_point Point;
typedef Traits::Weighted_point Weighted_point;

typedef CGAL::Regular_triangulation_3<Traits> Rt;

typedef Rt::Vertex_iterator Vertex_iterator;
typedef Rt::Vertex_handle Vertex_handle;

int main()
{
    Rt T;

    // insertion of points on a 3D grid
    std::vector<Vertex_handle> V;

    for (int z=0 ; z<5 ; z++)
        for (int y=0 ; y<5 ; y++)
            for (int x=0 ; x<5 ; x++) {
                Point p(x, y, z);

```

```

        Weight w = (x+y-z*y*x)*2.0; // let's say this is the weight.
        Weighted_point wp(p, w);
        V.push_back(T.insert(wp));
    }

    assert( T.is_valid() );
    assert( T.dimension() == 3 );

    std::cout << "Number of vertices : " << T.number_of_vertices() << std::endl;

    return 0;
}

```

22.7 Design and Implementation History

Monique Teillaud started to work on the 3D triangulation packages in 1997, following the design of the 2D triangulation packages. The notions of degenerate dimensions and infinite vertex were formalized [Tei99] and induced changes in the 2D triangulation packages. The packages were first released in CGAL 2.1. They contained basic functionalities on triangulations, Delaunay triangulations, regular triangulations.

A first version of removal of a vertex from a Delaunay triangulation was released in CGAL 2.2. However, this removal became really robust only in CGAL 2.3, after some research that allowed to deal with degenerate cases quite easily [DT03]. Andreas Fabri implemented this revised version of the removal, and a faster removal algorithm for CGAL 3.0.

In 2000, Sylvain Pion started working on these packages. He improved the efficiency of triangulations in CGAL 2.3 and 2.4 in several ways [BDP⁺02]: he implemented the Delaunay hierarchy [Dev02] in 2.3, he improved the memory footprint in 2.4 and 3.0, he also performed work on arithmetic filters [DP03] (see Support Library and Kernel) to improve the speed of triangulations. He changed the design in CGAL 3.0, allowing users to add handles in their own vertices and cells.

In 2005, Christophe Delage implemented the vertex removal function for regular triangulations, which allowed to release this functionality in CGAL 3.2.

The authors also wish to thank Jean-Daniel Boissonnat, Olivier Devillers and Mariette Yvinec for helpful discussions [BDTY00].

3D Triangulations

Reference Manual

Sylvain Pion and Monique Teillaud

A three-dimensional triangulation is a three-dimensional simplicial complex, pure connected and without singularities [BY98]. Its cells (3-faces) are such that two cells either do not intersect or share a common facet (2-face), edge (1-face) or vertex (0-face).

The basic 3D-triangulation class of CGAL is primarily designed to represent the triangulations of a set of points A in \mathbb{R}^3 . It can be viewed as a partition of the convex hull of A into tetrahedra whose vertices are the points of A . Together with the unbounded cell having the convex hull boundary as its frontier, the triangulation forms a partition of \mathbb{R}^3 .

In order to deal only with tetrahedra, which is convenient for many applications, the unbounded cell can be subdivided into tetrahedra by considering that each convex hull facet is incident to an *infinite cell* having as fourth vertex an auxiliary vertex called the *infinite vertex*. In that way, each facet is incident to exactly two cells and special cases at the boundary of the convex hull are simple to deal with.

A triangulation is a collection of vertices and cells that are linked together through incidence and adjacency relations. Each cell gives access to its four incident vertices and to its four adjacent cells. Each vertex gives access to one of its incident cells.

The four vertices of a cell are indexed with 0, 1, 2 and 3 in positive orientation, the positive orientation being defined by the orientation of the underlying Euclidean space \mathbb{R}^3 . The neighbors of a cell are also indexed with 0, 1, 2, 3 in such a way that the neighbor indexed by i is opposite to the vertex with the same index. See Figure 22.1.

22.8 Classified Reference Pages

Concepts

TriangulationTraits_3	page 1534
DelaunayTriangulationTraits_3	page 1536
RegularTriangulationTraits_3	page 1539
TriangulationCellBase_3	page 1546
TriangulationVertexBase_3	page 1547
TriangulationHierarchyVertexBase_3	page 1549

RegularTriangulationCellBase_3	page 1550
TriangulationDataStructure_3	page 1573
WeightedPoint	page 1559

Classes

Main Classes

<i>CGAL::Triangulation_3</i> < <i>TriangulationTraits_3</i> , <i>TriangulationDataStructure_3</i> >	page 1504
<i>CGAL::Delaunay_triangulation_3</i> < <i>DelaunayTriangulationTraits_3</i> , <i>TriangulationDataStructure_3</i> >	page 1520
<i>CGAL::Triangulation_hierarchy_3</i> < <i>Tr</i> >	page 1527
<i>CGAL::Regular_triangulation_3</i> < <i>RegularTriangulationTraits_3</i> , <i>TriangulationDataStructure_3</i> > ...	page 1528
<i>CGAL::Triangulation_cell_base_3</i> < <i>TriangulationTraits_3</i> , <i>TriangulationDSCellBase_3</i> >	page 1552
<i>CGAL::Triangulation_cell_base_with_info_3</i> < <i>Info</i> , <i>TriangulationTraits_3</i> , <i>TriangulationCellBase_3</i> > page 1553	
<i>CGAL::Triangulation_vertex_base_3</i> < <i>TriangulationTraits_3</i> , <i>TriangulationDSVertexBase_3</i> >	page 1554
<i>CGAL::Triangulation_vertex_base_with_info_3</i> < <i>Info</i> , <i>TriangulationTraits_3</i> , <i>TriangulationVertexBase_3</i> > page 1555	
<i>CGAL::Triangulation_hierarchy_vertex_base_3</i> < <i>TriangulationVertexBase_3</i> >	page 1556
<i>CGAL::Regular_triangulation_cell_base_3</i> < <i>Traits</i> , <i>Cb</i> >	page 1557

Traits Classes

<i>CGAL::Regular_triangulation_euclidean_traits_3</i> < <i>K</i> , <i>Weight</i> >	page 1542
<i>CGAL::Regular_triangulation_filtered_traits_3</i> < <i>FK</i> >	page 1545

Enums

<i>CGAL::Triangulation_3::Locate_type</i>	page 1558
-------------------------------------------------	---------------------------

22.9 Alphabetical List of Reference Pages

<i>DelaunayTriangulationTraits_3</i>	page 1536
<i>Delaunay_triangulation_3</i> < <i>DelaunayTriangulationTraits_3</i> , <i>TriangulationDataStructure_3</i> >	page 1520
<i>Locate_type</i>	page 1558
<i>RegularTriangulationCellBase_3</i>	page 1550
<i>RegularTriangulationTraits_3</i>	page 1539
<i>Regular_triangulation_3</i> < <i>RegularTriangulationTraits_3</i> , <i>TriangulationDataStructure_3</i> >	page 1528
<i>Regular_triangulation_cell_base_3</i> < <i>Traits</i> , <i>Cb</i> >	page 1557
<i>Regular_triangulation_euclidean_traits_3</i> < <i>K</i> , <i>Weight</i> >	page 1542
<i>Regular_triangulation_filtered_traits_3</i> < <i>FK</i> >	page 1545

<i>TriangulationCellBase_3</i>	page 1546
<i>TriangulationHierarchyVertexBase_3</i>	page 1549
<i>TriangulationTraits_3</i>	page 1534
<i>TriangulationVertexBase_3</i>	page 1547
<i>Triangulation_3</i> < <i>TriangulationTraits_3</i> , <i>TriangulationDataStructure_3</i> >	page 1504
<i>Triangulation_cell_base_3</i> < <i>TriangulationTraits_3</i> , <i>TriangulationDSCellBase_3</i> >	page 1552
<i>Triangulation_cell_base_with_info_3</i> < <i>Info</i> , <i>TriangulationTraits_3</i> , <i>TriangulationCellBase_3</i> >	page 1553
<i>Triangulation_hierarchy_3</i> < <i>Tr</i> >	page 1527
<i>Triangulation_hierarchy_vertex_base_3</i> < <i>TriangulationVertexBase_3</i> >	page 1556
<i>Triangulation_vertex_base_3</i> < <i>TriangulationTraits_3</i> , <i>TriangulationDSVertexBase_3</i> >	page 1554
<i>Triangulation_vertex_base_with_info_3</i> < <i>Info</i> , <i>TriangulationTraits_3</i> , <i>TriangulationVertexBase_3</i> > ..	page 1555
<i>WeightedPoint</i>	page 1559

CGAL::Triangulation_3<TriangulationTraits_3,TriangulationDataStructure_3>

Definition

The class *Triangulation_3* represents a 3-dimensional tetrahedralization of points.

```
#include <CGAL/Triangulation_3.h>
```

Parameters

The first template argument must be a model of the *TriangulationTraits_3* concept.

The second template argument must be a model of the *TriangulationDataStructure_3* concept. It has the default value *Triangulation_data_structure_3<Triangulation_vertex_base_3<TriangulationTraits_3>,Triangulation_cell_base_3<TriangulationTraits_3>>*.

Inherits From

Triangulation_utils_3

Types

The class *Triangulation_3* defines the following types:

<code>typedef TriangulationDataStructure_3</code>	<code>Triangulation_data_structure;</code>
<code>typedef TriangulationTraits_3</code>	<code>Geom_traits;</code>
<code>typedef TriangulationTraits_3::Point_3</code>	<code>Point;</code>
<code>typedef TriangulationTraits_3::Segment_3</code>	<code>Segment;</code>
<code>typedef TriangulationTraits_3::Triangle_3</code>	<code>Triangle;</code>
<code>typedef TriangulationTraits_3::Tetrahedron_3</code>	<code>Tetrahedron;</code>

Only vertices (0-faces) and cells (3-faces) are stored. Edges (1-faces) and facets (2-faces) are not explicitly represented and thus there are no corresponding classes (see Section 22.1).

<code>typedef TriangulationDataStructure_3::Vertex</code>	<code>Vertex;</code>
<code>typedef TriangulationDataStructure_3::Cell</code>	<code>Cell;</code>
<code>typedef TriangulationDataStructure_3::Facet</code>	<code>Facet;</code>
<code>typedef TriangulationDataStructure_3::Edge</code>	<code>Edge;</code>

The vertices and faces of the triangulations are accessed through *handles*, *iterators* and *circulators*. A handle is a type which supports the two dereference operators *operator** and *operator->*. The Handle concept is documented in the support library. Iterators and circulators are bidirectional and non-mutable. The edges and facets of the triangulation can also be visited through iterators and circulators which are bidirectional and non-mutable.

Iterators and circulators are convertible to the corresponding handles, thus the user can pass them directly as arguments to the functions.

<i>typedef TriangulationDataStructure_3::Vertex_handle</i>	<i>Vertex_handle;</i>	handle to a vertex
<i>typedef TriangulationDataStructure_3::Cell_handle</i>	<i>Cell_handle;</i>	handle to a cell
<i>typedef TriangulationDataStructure_3::size_type</i>	<i>size_type;</i>	Size type (an unsigned integral type)
<i>typedef TriangulationDataStructure_3::difference_type</i>	<i>difference_type;</i>	Difference type (a signed integral type)
<i>typedef TriangulationDataStructure_3::Cell_iterator</i>	<i>All_cells_iterator;</i>	iterator over cells
<i>typedef TriangulationDataStructure_3::Facet_iterator</i>	<i>All_facets_iterator;</i>	iterator over facets
<i>typedef TriangulationDataStructure_3::Edge_iterator</i>	<i>All_edges_iterator;</i>	iterator over edges
<i>typedef TriangulationDataStructure_3::Vertex_iterator</i>	<i>All_vertices_iterator;</i>	iterator over vertices
<i>Triangulation_3<TriangulationTraits_3,TriangulationDataStructure_3>:: Finite_cells_iterator</i>		iterator over finite cells
<i>Triangulation_3<TriangulationTraits_3,TriangulationDataStructure_3>:: Finite_facets_iterator</i>		iterator over finite facets
<i>Triangulation_3<TriangulationTraits_3,TriangulationDataStructure_3>:: Finite_edges_iterator</i>		iterator over finite edges
<i>Triangulation_3<TriangulationTraits_3,TriangulationDataStructure_3>:: Finite_vertices_iterator</i>		iterator over finite vertices
<i>Triangulation_3<TriangulationTraits_3,TriangulationDataStructure_3>:: Point_iterator</i>		iterator over the points corresponding to the finite vertices of the triangulation.
<i>typedef TriangulationDataStructure_3::Cell_circulator</i>	<i>Cell_circulator;</i>	circulator over all cells incident to a given edge
<i>typedef TriangulationDataStructure_3::Facet_circulator</i>	<i>Facet_circulator;</i>	circulator over all facets incident to a given edge

The triangulation class also defines the following enum type to specify which case occurs when locating a point in the triangulation.

```
enum Locate_type { VERTEX=0,
                  EDGE,
                  FACET,
                  CELL,
                  OUTSIDE_CONVEX_HULL,
                  OUTSIDE_AFFINE_HULL}
```

Creation

Triangulation_3<*TriangulationTraits_3*,*TriangulationDataStructure_3*> *t*(*TriangulationTraits_3* *traits* = *TriangulationTraits_3*())

Introduces a triangulation *t* having only one vertex which is the infinite vertex.

Triangulation_3<*TriangulationTraits_3*,*TriangulationDataStructure_3*> *t*(*Triangulation_3* *tr*);

Copy constructor. All vertices and faces are duplicated.

template < *class InputIterator* >
Triangulation_3<*TriangulationTraits_3*,*TriangulationDataStructure_3*> *t*(*InputIterator* *first*,
InputIterator *last*,
TriangulationTraits_3 *traits* = *TriangulationTraits_3*())

Introduces a triangulation *t* constructed by the repeated insertion of the iterator range [*first,last*) of value type *Point*.

Assignment

Triangulation_3 & *t* = *Triangulation_3* *tr*

The triangulation *tr* is duplicated, and modifying the copy after the duplication does not modify the original. The previous triangulation held by *t* is deleted.

void *t.swap*(*Triangulation_3* & *tr*)

The triangulations *tr* and *t* are swapped. *t.swap(tr)* should be preferred to *t = tr* or to *t(tr)* if *tr* is deleted after that. Indeed, there is no copy of cells and vertices, thus this method runs in constant time.

void *t.clear*() Deletes all finite vertices and all cells of *t*.

template < *class GT*, *class Tds* >
bool *Triangulation_3*<*GT*, *Tds*> *t1* == *Triangulation_3*<*GT*, *Tds*> *t2*

Equality operator. Returns true iff there exist a bijection between the vertices of *t1* and those of *t2* and a bijection between the cells of *t1* and those of *t2*, which preserve the geometry of the triangulation, that is, the points of each corresponding pair of vertices are equal, and the tetrahedra corresponding to each pair of cells are equal (up to a permutation of their vertices).

template < *class GT*, *class Tds* >
bool *Triangulation_3*<*GT*, *Tds*> *t1* != *Triangulation_3*<*GT*, *Tds*> *t2*

The opposite of *operator==*.

Access Functions

<i>TriangulationTraits_3</i>	<i>t.geom_traits()</i>	Returns a const reference to the geometric traits object.
<i>TriangulationDataStructure_3</i>	<i>t.tds()</i>	Returns a const reference to the triangulation data structure.

— *advanced* —

Non const access

The responsibility of keeping a valid triangulation belongs to the user when using advanced operations allowing a direct manipulation of the *tds*.

<i>TriangulationDataStructure_3</i> &	<i>t.tds()</i>	Returns a reference to the triangulation data structure.
---------------------------------------	----------------	----------------------------------------------------------

This method is mainly a help for users implementing their own triangulation algorithms.

— *advanced* —

<i>int</i>	<i>t.dimension()</i>	Returns the dimension of the affine hull.
<i>size_type</i>	<i>t.number_of_vertices()</i>	Returns the number of finite vertices.
<i>size_type</i>	<i>t.number_of_cells()</i>	Returns the number of cells or 0 if <i>t.dimension()</i> < 3.
<i>Vertex_handle</i>	<i>t.infinite_vertex()</i>	Returns the infinite vertex.
<i>Cell_handle</i>	<i>t.infinite_cell()</i>	Returns a cell incident to the infinite vertex.

Non-constant-time access functions

As previously said, the triangulation is a collection of cells that are either infinite or represent a finite tetrahedra, where an infinite cell is a cell incident to the infinite vertex. Similarly we call an edge (resp. facet) *infinite* if it is incident to the infinite vertex.

<i>size_type</i>	<i>t.number_of_facets()</i>	The number of facets. Returns 0 if <i>t.dimension()</i> < 2.
<i>size_type</i>	<i>t.number_of_edges()</i>	The number of edges. Returns 0 if <i>t.dimension()</i> < 1.
<i>size_type</i>	<i>t.number_of_finite_cells()</i>	The number of finite cells. Returns 0 if <i>t.dimension()</i> < 3.
<i>size_type</i>	<i>t.number_of_finite_facets()</i>	The number of finite facets. Returns 0 if <i>t.dimension()</i> < 2.
<i>size_type</i>	<i>t.number_of_finite_edges()</i>	The number of finite edges. Returns 0 if <i>t.dimension()</i> < 1.

Geometric access functions

<i>Tetrahedron</i>	<i>t.tetrahedron(const Cell_handle c)</i>	Returns the tetrahedron formed by the four vertices of <i>c</i> . <i>Precondition:</i> <i>t.dimension()</i> = 3 and the cell is finite.
<i>Triangle</i>	<i>t.triangle(const Cell_handle c, int i)</i>	Returns the triangle formed by the three vertices of facet (<i>c</i> , <i>i</i>). The triangle is oriented so that its normal points to the inside of cell <i>c</i> . <i>Precondition:</i> <i>t.dimension()</i> ≥ 2 and <i>i</i> ∈ {0,1,2,3} in dimension 3, <i>i</i> = 3 in dimension 2, and the facet is finite.

<i>Triangle</i>	<i>t.triangle(Facet f)</i>	<p>Same as the previous method for facet <i>f</i>. <i>Precondition:</i> <i>t.dimension()</i> ≥ 2 and the facet is finite.</p>
<i>Segment</i>	<i>t.segment(Edge e)</i>	<p>Returns the line segment formed by the vertices of <i>e</i>. <i>Precondition:</i> <i>t.dimension()</i> ≥ 1 and <i>e</i> is finite.</p>
<i>Segment</i>	<i>t.segment(const Cell_handle c, int i, int j)</i>	<p>Same as the previous method for edge (c, i, j). <i>Precondition:</i> As above and $i \neq j$. Moreover $i, j \in \{0, 1, 2, 3\}$ in dimension 3, $i, j \in \{0, 1, 2\}$ in dimension 2, $i, j \in \{0, 1\}$ in dimension 1.</p>

Tests for Finite and Infinite Vertices and Faces

<i>bool</i>	<i>t.is_infinite(const Vertex_handle v)</i>	<p><i>true</i>, iff vertex <i>v</i> is the infinite vertex.</p>
<i>bool</i>	<i>t.is_infinite(const Cell_handle c)</i>	<p><i>true</i>, iff <i>c</i> is incident to the infinite vertex. <i>Precondition:</i> <i>t.dimension()</i> = 3.</p>
<i>bool</i>	<i>t.is_infinite(const Cell_handle c, int i)</i>	<p><i>true</i>, iff the facet <i>i</i> of cell <i>c</i> is incident to the infinite vertex. <i>Precondition:</i> <i>t.dimension()</i> ≥ 2 and $i \in \{0, 1, 2, 3\}$ in dimension 3, $i = 3$ in dimension 2.</p>
<i>bool</i>	<i>t.is_infinite(Facet f)</i>	<p><i>true</i> iff facet <i>f</i> is incident to the infinite vertex. <i>Precondition:</i> <i>t.dimension()</i> ≥ 2.</p>
<i>bool</i>	<i>t.is_infinite(const Cell_handle c, int i, int j)</i>	<p><i>true</i>, iff the edge (i, j) of cell <i>c</i> is incident to the infinite vertex. <i>Precondition:</i> <i>t.dimension()</i> ≥ 1 and $i \neq j$. Moreover $i, j \in \{0, 1, 2, 3\}$ in dimension 3, $i, j \in \{0, 1, 2\}$ in dimension 2, $i, j \in \{0, 1\}$ in dimension 1.</p>
<i>bool</i>	<i>t.is_infinite(Edge e)</i>	<p><i>true</i> iff edge <i>e</i> is incident to the infinite vertex. <i>Precondition:</i> <i>t.dimension()</i> ≥ 1.</p>

Queries

<i>bool</i>	<i>t.is_vertex(Point p, Vertex_handle & v)</i>	<p>Tests whether <i>p</i> is a vertex of <i>t</i> by locating <i>p</i> in the triangulation. If <i>p</i> is found, the associated vertex <i>v</i> is given.</p>
<i>bool</i>	<i>t.is_vertex(Vertex_handle v)</i>	<p>Tests whether <i>v</i> is a vertex of <i>t</i>.</p>

bool *t.is_edge(Vertex_handle u, Vertex_handle v, Cell_handle & c, int & i, int & j)*

Tests whether (u,v) is an edge of t . If the edge is found, it gives a cell c having this edge and the indices i and j of the vertices u and v in c , in this order.
Precondition: u and v are vertices of t .

bool *t.is_facet(Vertex_handle u,*
 Vertex_handle v,
 Vertex_handle w,
 Cell_handle & c,
 int & i,
 int & j,
 int & k)

Tests whether (u,v,w) is a facet of t . If the facet is found, it computes a cell c having this facet and the indices i, j and k of the vertices u, v and w in c , in this order.
Precondition: u, v and w are vertices of t .

bool *t.is_cell(Cell_handle c)*

Tests whether c is a cell of t .

bool *t.is_cell(Vertex_handle u,*
 Vertex_handle v,
 Vertex_handle w,
 Vertex_handle x,
 Cell_handle & c,
 int & i,
 int & j,
 int & k,
 int & l)

Tests whether (u,v,w,x) is a cell of t . If the cell c is found, the method computes the indices i, j, k and l of the vertices u, v, w and x in c , in this order.
Precondition: u, v, w and x are vertices of t .

bool *t.is_cell(Vertex_handle u,*
 Vertex_handle v,
 Vertex_handle w,
 Vertex_handle x,
 Cell_handle & c)

Tests whether (u,v,w,x) is a cell of t and computes this cell c .
Precondition: u, v, w and x are vertices of t .

There is a method *has_vertex* in the cell class. The analogous methods for facets are defined here.

bool *t.has_vertex(Facet f, Vertex_handle v, int & j)*

If v is a vertex of f , then j is the index of v in the cell $f.first$, and the method returns *true*.
Precondition: $t.dimension()=3$

bool *t.has_vertex(Cell_handle c, int i, Vertex_handle v, int & j)*

Same for facet (c,i) . Computes the index j of v in c .

bool *t.has_vertex(Facet f, Vertex_handle v)*

bool *t.has_vertex(Cell_handle c, int i, Vertex_handle v)*

Same as the first two methods, but these two methods do not return the index of the vertex.

The following three methods test whether two facets have the same vertices.

bool *t.are_equal(Cell_handle c, int i, Cell_handle n, int j)*

bool *t.are_equal(Facet f, Facet g)*

bool *t.are_equal(Facet f, Cell_handle n, int j)*

For these three methods:

Precondition: $t.dimension()=3$.

Point location

The class *Triangulation_3<TriangulationTraits_3, TriangulationDataStructure_3>* provides two functions to locate a given point with respect to a triangulation. It provides also functions to test if a given point is inside a finite face or not. Note that the class *Delaunay_triangulation_3* also provides a *nearest_vertex()* function.

Cell_handle *t.locate(Point query, Cell_handle start = Cell_handle())*

If the point *query* lies inside the convex hull of the points, the cell that contains the query in its interior is returned. If *query* lies on a facet, an edge or on a vertex, one of the cells having *query* on its boundary is returned.

If the point *query* lies outside the convex hull of the points, an infinite cell with vertices $\{p, q, r, \infty\}$ is returned such that the tetrahedron $(p, q, r, query)$ is positively oriented (the rest of the triangulation lies on the other side of facet (p, q, r)).

Note that *locate* works even in degenerate dimensions: in dimension 2 (resp. 1, 0) the *Cell_handle* returned is the one that represents the facet (resp. edge, vertex) containing the query point.

The optional argument *start* is used as a starting place for the search.

Cell_handle *t.locate(Point query, Locate_type & lt, int & li, int & lj, Cell_handle start = Cell_handle())*

If *query* lies inside the affine hull of the points, the *k*-face (finite or infinite) that contains *query* in its interior is returned, by means of the cell returned together with *lt*, which is set to the locate type of the query (*VERTEX*, *EDGE*, *FACET*, *CELL*, or *OUTSIDE_CONVEX_HULL* if the cell is infinite and *query* lies strictly in it) and two indices *li* and *lj* that specify the *k*-face of the cell containing *query*.

If the *k*-face is a cell, *li* and *lj* have no meaning; if it is a facet (resp. vertex), *li* gives the index of the facet (resp. vertex) and *lj* has no meaning; if it is an edge, *li* and *lj* give the indices of its vertices.

If the point *query* lies outside the affine hull of the points, which can happen in case of degenerate dimensions, *lt* is set to *OUTSIDE_AFFINE_HULL*, and the cell returned has no meaning. As a particular case, if there is no finite vertex yet in the triangulation, *lt* is set to *OUTSIDE_AFFINE_HULL* and *locate* returns the default constructed handle.

The optional argument *start* is used as a starting place for the search.

Bounded_side *t.side_of_cell(Point p, Cell_handle c, Locate_type & lt, int & li, int & lj)*

Returns a value indicating on which side of the oriented boundary of *c* the point *p* lies. More precisely, it returns:

- *ON_BOUNDED_SIDE* if *p* is inside the cell. For an infinite cell this means that *p* lies strictly in the half space limited by its finite facet and not containing any other point of the triangulation.

- *ON_BOUNDARY* if *p* on the boundary of the cell. For an infinite cell this means that *p* lies on the *finite* facet. Then *lt* together with *li* and *lj* give the precise location on the boundary. (See the descriptions of the *locate* methods.)

- *ON_UNBOUNDED_SIDE* if *p* lies outside the cell. For an infinite cell this means that *p* does not satisfy either of the two previous conditions.

Precondition: t.dimension() = 3

Bounded_side *t.side_of_facet(Point p, Facet f, Locate_type & lt, int & li, int & lj)*

Returns a value indicating on which side of the oriented boundary of *f* the point *p* lies:

- *ON_BOUNDED_SIDE* if *p* is inside the facet. For an infinite facet this means that *p* lies strictly in the half plane limited by its finite edge and not containing any other point of the triangulation.

- *ON_BOUNDARY* if *p* is on the boundary of the facet. For an infinite facet this means that *p* lies on the finite edge. *lt*, *li* and *lj* give the precise location of *p* on the boundary of the facet. *li* and *lj* refer to indices in the degenerate cell *c* representing *f*.

- *ON_UNBOUNDED_SIDE* if *p* lies outside the facet. For an infinite facet this means that *p* does not satisfy either of the two previous conditions.

Precondition: t.dimension() = 2 and p lies in the plane containing the triangulation. f.second = 3 (in dimension 2 there is only one facet per cell).

Bounded_side *t.side_of_facet(Point p, Cell_handle c, Locate_type & lt, int & li, int & lj)*

Same as the previous method for the facet (*c*,3).

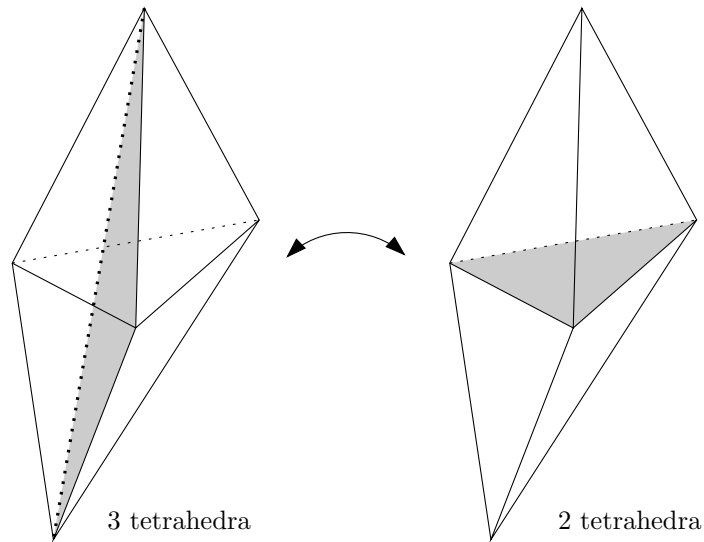


Figure 22.6: Flips.

Bounded_side $t.side_of_edge(Point\ p, Edge\ e, Locate_type\ \&\ lt, int\ \&\ li)$

Returns a value indicating on which side of the oriented boundary of e the point p lies:

- *ON_BOUNDED_SIDE* if p is inside the edge. For an infinite edge this means that p lies in the half line defined by the vertex and not containing any other point of the triangulation.
- *ON_BOUNDARY* if p equals one of the vertices, li give the index of the vertex in the cell storing e
- *ON_UNBOUNDED_SIDE* if p lies outside the edge. For an infinite edge this means that p lies on the other half line, which contains the other points of the triangulation.

Precondition: $t.dimension() = 1$ and p is collinear with the points of the triangulation. $e.second = 0$ and $e.third = 1$ (in dimension 1 there is only one edge per cell).

Bounded_side $t.side_of_edge(Point\ p, Cell_handle\ c, Locate_type\ \&\ lt, int\ \&\ li)$

Same as the previous method for edge $(c, 0, 1)$.

Flips

Two kinds of flips exist for a three-dimensional triangulation. They are reciprocal. To be flipped, an edge must be incident to three tetrahedra. During the flip, these three tetrahedra disappear and two tetrahedra appear. Figure 22.6(left) shows the edge that is flipped as bold dashed, and one of its three incident facets is shaded. On the right, the facet shared by the two new tetrahedra is shaded.

Flips are possible only under the following conditions:

- the edge or facet to be flipped is not on the boundary of the convex hull of the triangulation
- the five points involved are in convex position.

The following methods guarantee the validity of the resulting 3D triangulation.

Flips for a 2d triangulation are not implemented yet

```
bool      t.flip( Edge e)
bool      t.flip( Cell_handle c, int i, int j)
```

Before flipping, these methods check that edge $e=(c,i,j)$ is flippable (which is quite expensive). They return *false* or *true* according to this test.

```
void      t.flip_flippable( Edge e)
void      t.flip_flippable( Cell_handle c, int i, int j)
```

Should be preferred to the previous methods when the edge is known to be flippable.

Precondition: The edge is flippable.

```
bool      t.flip( Facet f)
bool      t.flip( Cell_handle c, int i)
```

Before flipping, these methods check that facet $f=(c,i)$ is flippable (which is quite expensive). They return *false* or *true* according to this test.

```
void      t.flip_flippable( Facet f)
void      t.flip_flippable( Cell_handle c, int i)
```

Should be preferred to the previous methods when the facet is known to be flippable.

Precondition: The facet is flippable.

Insertions

The following operations are guaranteed to lead to a valid triangulation when they are applied on a valid triangulation.

```
Vertex_handle  t.insert( Point p, Cell_handle start = Cell_handle())
```

Inserts point p in the triangulation and returns the corresponding vertex.

If point p coincides with an already existing vertex, this vertex is returned and the triangulation remains unchanged.

If point p lies in the convex hull of the points, it is added naturally: if it lies inside a cell, the cell is split into four cells, if it lies on a facet, the two incident cells are split into three cells, if it lies on an edge, all the cells incident to this edge are split into two cells.

If point p is strictly outside the convex hull but in the affine hull, p is linked to all visible points on the convex hull to form the new triangulation. See Figure 22.7.

If point p is outside the affine hull of the points, p is linked to all the points, and the dimension of the triangulation is incremented. All the points now belong to the boundary of the convex hull, so, the infinite vertex is linked to all the points to triangulate the new infinite face. See Figure 22.8. The optional argument *start* is used as a starting place for the search.

Vertex_handle *t.insert(Point p, Locate_type lt, Cell_handle loc, int li, int lj)*

Inserts point *p* in the triangulation and returns the corresponding vertex. Similar to the above *insert()* function, but takes as additional parameter the return values of a previous location query. See description of *locate()* above.

template < class InputIterator >
int *t.insert(InputIterator first, InputIterator last)*

Inserts the points in the range $[first, last)$. Returns the number of inserted points.

Precondition: The *value_type* of *first* and *last* is *Point*.

The previous methods are sufficient to build a whole triangulation. We also provide some other methods that can be used instead of *insert(p)* when the place where the new point *p* must be inserted is already known. They are also guaranteed to lead to a valid triangulation when they are applied on a valid triangulation.

Vertex_handle *t.insert_in_cell(Point p, Cell_handle c)*

Inserts point *p* in cell *c*. Cell *c* is split into 4 tetrahedra.

Precondition: *t.dimension()* = 3 and *p* lies strictly inside cell *c*.

Vertex_handle *t.insert_in_facet(Point p, Facet f)*

Inserts point *p* in facet *f*. In dimension 3, the 2 neighboring cells are split into 3 tetrahedra; in dimension 2, the facet is split into 3 triangles.

Precondition: *t.dimension()* ≥ 2 and *p* lies strictly inside face *f*.

Vertex_handle *t.insert_in_facet(Point p, Cell_handle c, int i)*

As above, insertion in facet (c, i) .

Precondition: As above and $i \in \{0, 1, 2, 3\}$ in dimension 3, $i = 3$ in dimension 2.

Vertex_handle *t.insert_in_edge(Point p, Edge e)*

Inserts *p* in edge *e*. In dimension 3, all the cells having this edge are split into 2 tetrahedra; in dimension 2, the 2 neighboring facets are split into 2 triangles; in dimension 1, the edge is split into 2 edges.

Precondition: *t.dimension()* ≥ 1 and *p* lies on edge *e*.

Vertex_handle *t.insert_in_edge(Point p, Cell_handle c, int i, int j)*

As above, inserts *p* in edge (i, j) of *c*.

Precondition: As above and $i \neq j$. Moreover $i, j \in \{0, 1, 2, 3\}$ in dimension 3, $i, j \in \{0, 1, 2\}$ in dimension 2, $i, j \in \{0, 1\}$ in dimension 1.

Vertex_handle *t.insert_outside_convex_hull(Point p, Cell_handle c)*

The cell *c* must be an infinite cell containing *p*.

Links *p* to all points in the triangulation that are visible from *p*. Updates consequently the infinite faces. See Figure 22.7.

Precondition: *t.dimension()* > 0 , *c*, and the *k*-face represented by *c* is infinite and contains *t*.

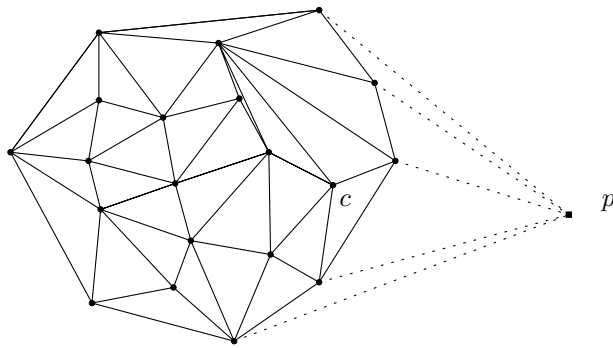


Figure 22.7: *insert_outside_convex_hull* (2-dimensional case).

Vertex_handle `t.insert_outside_affine_hull(Point p)`

p is linked to all the points, and the infinite vertex is linked to all the points (including p) to triangulate the new infinite face, so that all the points now belong to the boundary of the convex hull. See Figure 22.8.

This method can be used to insert the first point in an empty triangulation.

Precondition: $t.dimension() < 3$ and p lies outside the affine hull of the points.

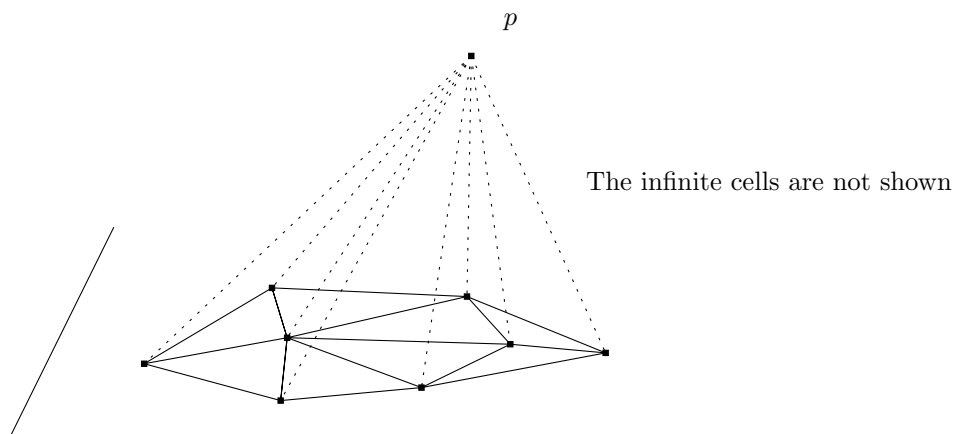


Figure 22.8: *insert_outside_affine_hull* (2-dimensional case).

Vertex_handle *t.insert_in_hole(Point p, CellIt cell_begin, CellIt cell_end, Cell_handle begin, int i)*

Creates a new vertex by starring a hole. It takes an iterator range [*cell_begin*; *cell_end*] of *Cell_handles* which specifies a hole: a set of connected cells (resp. facets in dimension 2) which is star-shaped wrt *p*. (*begin*, *i*) is a facet (resp. an edge) on the boundary of the hole, that is, *begin* belongs to the set of cells (resp. facets) previously described, and *begin*->*neighbor(i)* does not. Then this function deletes all the cells (resp. facets) describing the hole, creates a new vertex *v*, and for each facet (resp. edge) on the boundary of the hole, creates a new cell (resp. facet) with *v* as vertex. Then *v*->*set_point(p)* is called and *v* is returned.

This operation is equivalent to calling *tds().insert_in_hole(cell_begin, cell_end, begin, i); v->set_point(p)*.

Precondition: *t.dimension()* ≥ 2 , the set of cells (resp. facets in dimension 2) is connected, its boundary is connected, and *p* lies inside the hole, which is star-shaped wrt *p*.

Traversal of the Triangulation

The triangulation class provides several iterators and circulators that allow one to traverse it (completely or partially).

Cell, Face, Edge and Vertex Iterators

The following iterators allow the user to visit cells, facets, edges and vertices of the triangulation. These iterators are non-mutable, bidirectional and their value types are respectively *Cell*, *Facet*, *Edge* and *Vertex*. They are all invalidated by any change in the triangulation.

<i>Finite_vertices_iterator</i>	<i>t.finite_vertices_begin()</i>	Starts at an arbitrary finite vertex. Then ++ and -- will iterate over finite vertices. Returns <i>finite_vertices_end()</i> when <i>t.number_of_vertices()</i> = 0.
<i>Finite_vertices_iterator</i>	<i>t.finite_vertices_end()</i>	Past-the-end iterator
<i>Finite_edges_iterator</i>	<i>t.finite_edges_begin()</i>	Starts at an arbitrary finite edge. Then ++ and -- will iterate over finite edges. Returns <i>finite_edges_end()</i> when <i>t.dimension()</i> < 1.
<i>Finite_edges_iterator</i>	<i>t.finite_edges_end()</i>	Past-the-end iterator
<i>Finite_facets_iterator</i>	<i>t.finite_facets_begin()</i>	Starts at an arbitrary finite facet. Then ++ and -- will iterate over finite facets. Returns <i>finite_facets_end()</i> when <i>t.dimension()</i> < 2.
<i>Finite_facets_iterator</i>	<i>t.finite_facets_end()</i>	Past-the-end iterator
<i>Finite_cells_iterator</i>	<i>t.finite_cells_begin()</i>	Starts at an arbitrary finite cell. Then ++ and -- will iterate over finite cells. Returns <i>finite_cells_end()</i> when <i>t.dimension()</i> < 3.
<i>Finite_cells_iterator</i>	<i>t.finite_cells_end()</i>	Past-the-end iterator
<i>All_vertices_iterator</i>	<i>t.all_vertices_begin()</i>	Starts at an arbitrary vertex. Iterates over all vertices (even the infinite one). Returns <i>vertices_end()</i> when <i>t.number_of_vertices()</i> = 0.
<i>All_vertices_iterator</i>	<i>t.all_vertices_end()</i>	Past-the-end iterator

<i>All_edges_iterator</i>	<i>t.all_edges_begin()</i>	Starts at an arbitrary edge. Iterates over all edges (even infinite ones). Returns <i>edges_end()</i> when <i>t.dimension()</i> < 1.
<i>All_edges_iterator</i>	<i>t.all_edges_end()</i>	Past-the-end iterator
<i>All_facets_iterator</i>	<i>t.all_facets_begin()</i>	Starts at an arbitrary facet. Iterates over all facets (even infinite ones). Returns <i>facets_end()</i> when <i>t.dimension()</i> < 2.
<i>All_facets_iterator</i>	<i>t.all_facets_end()</i>	Past-the-end iterator
<i>All_cells_iterator</i>	<i>t.all_cells_begin()</i>	Starts at an arbitrary cell. Iterates over all cells (even infinite ones). Returns <i>cells_end()</i> when <i>t.dimension()</i> < 3.
<i>All_cells_iterator</i>	<i>t.all_cells_end()</i>	Past-the-end iterator
<i>Point_iterator</i>	<i>t.points_begin()</i>	Iterates over the points of the triangulation.
<i>Point_iterator</i>	<i>t.points_end()</i>	Past-the-end iterator

Cell and Facet Circulators

The following circulators respectively visit all cells or all facets incident to a given edge. They are non-mutable and bidirectional. They are invalidated by any modification of one of the cells traversed.

<i>Cell_circulator</i>	<i>t.incident_cells(Edge e)</i>	Starts at an arbitrary cell incident to <i>e</i> . <i>Precondition: t.dimension() = 3.</i>
<i>Cell_circulator</i>	<i>t.incident_cells(Cell_handle c, int i, int j)</i>	As above for edge (i,j) of <i>c</i> .
<i>Cell_circulator</i>	<i>t.incident_cells(Edge e, Cell_handle start)</i>	Starts at cell <i>start</i> . <i>Precondition: t.dimension() = 3 and start is incident to e.</i>
<i>Cell_circulator</i>	<i>t.incident_cells(Cell_handle c, int i, int j, Cell_handle start)</i>	As above for edge (i,j) of <i>c</i> .

The following circulators on facets are defined only in dimension 3, though facets are defined also in dimension 2: there are only two facets sharing an edge in dimension 2.

<i>Facet_circulator</i>	<i>t.incident_facets(Edge e)</i>	Starts at an arbitrary facet incident to <i>e</i> . <i>Precondition: t.dimension() = 3</i>
<i>Facet_circulator</i>	<i>t.incident_facets(Cell_handle c, int i, int j)</i>	As above for edge (i,j) of <i>c</i> .
<i>Facet_circulator</i>	<i>t.incident_facets(Edge e, Facet start)</i>	Starts at facet <i>start</i> . <i>Precondition: start is incident to e.</i>
<i>Facet_circulator</i>	<i>t.incident_facets(Edge e, Cell_handle start, int f)</i>	Starts at facet of index <i>f</i> in <i>start</i> .

Facet_circulator *t.incident_facets(Cell_handle c, int i, int j, Facet start)*

As above for edge (i,j) of c .

Facet_circulator *t.incident_facets(Cell_handle c, int i, int j, Cell_handle start, int f)*

As above for edge (i,j) of c and facet $(start,f)$.

int *t.mirror_index(Cell_handle c, int i)*

Returns the index of c in its i^{th} neighbor.

Precondition: $i \in \{0, 1, 2, 3\}$.

Vertex_handle *t.mirror_vertex(Cell_handle c, int i)*

Returns the vertex of the i^{th} neighbor of c that is opposite to c .

Precondition: $i \in \{0, 1, 2, 3\}$.

Facet *t.mirror_facet(Facet f)*

Returns the same facet viewed from the other adjacent cell.

Traversal of the incident cells and facets, and the adjacent vertices of a given vertex

template <class OutputIterator>

OutputIterator *t.incident_cells(Vertex_handle v, OutputIterator cells)*

Copies the *Cell_handles* of all cells incident to v to the output iterator *cells*. If $t.dimension() < 3$, then do nothing. Returns the resulting output iterator.

Precondition: $v \neq Vertex_handle(), t.is_vertex(v)$.

template <class OutputIterator>

OutputIterator *t.incident_facets(Vertex_handle v, OutputIterator facets)*

Copies the *Facets* incident to v to the output iterator *facets*. Returns the resulting output iterator.

Precondition: $t.dimension() = 3, v \neq Vertex_handle(), t.is_vertex(v)$.

template <class OutputIterator>

OutputIterator *t.incident_vertices(Vertex_handle v, OutputIterator vertices)*

Copies the *Vertex_handles* of all vertices incident to v to the output iterator *vertices*. If $t.dimension() < 2$, then do nothing. Returns the resulting output iterator.

Precondition: $v \neq Vertex_handle(), t.is_vertex(v)$.

size_type *t.degree(Vertex_handle v)*

Returns the degree of a vertex, that is, the number of incident vertices. The infinite vertex is counted.

Precondition: $v \neq Vertex_handle(), t.is_vertex(v)$.

Checking

The responsibility of keeping a valid triangulation belongs to the user when using advanced operations allowing a direct manipulation of cells and vertices. We provide the user with the following methods to help debugging.

bool *t.is_valid(bool verbose = false)*

Checks the combinatorial validity of the triangulation. Checks also the validity of its geometric embedding (see Section 22.1).

When *verbose* is set to true, messages describing the first invalidity encountered are printed.

bool *t.is_valid(Cell_handle c, bool verbose = false)*

Checks the combinatorial validity of the cell by calling the *is_valid* method of the *TriangulationDataStructure_3* cell class. Also checks the geometric validity of *c*, if *c* is finite. (See Section 1486.)

When *verbose* is set to *true*, messages are printed to give a precise indication of the kind of invalidity encountered.

└────────── *advanced* ─────────┘

I/O

CGAL provides an interface to Geomview for a 3D-triangulation. See the chapter on Geomview in the Support Library manual. `#include <CGAL/IO/Triangulation_geomview_ostream_3.h>`

istream& *istream& is >> Triangulation_3 &t*

Reads the underlying combinatorial triangulation from *is* by calling the corresponding input operator of the triangulation data structure class, and the non-combinatorial information by calling the corresponding input operators of the vertex and the cell classes. Assigns the resulting triangulation to *t*.

ostream& *ostream& os << Triangulation_3 t*

Writes the triangulation *t* into *os*.

The information in the *istream* is: the dimension, the number of finite vertices, the non-combinatorial information about vertices (point, etc), the number of cells, the indices of the vertices of each cell, plus the non-combinatorial information about each cell, then the indices of the neighbors of each cell, where the index corresponds to the preceding list of cells. When dimension < 3, the same information is stored for faces of maximal dimension instead of cells.

See Also

TriangulationDataStructure_3::Vertex
TriangulationDataStructure_3::Cell

CGAL::Delaunay_triangulation_3<DelaunayTriangulationTraits_3, TriangulationDataStructure_3>

Definition

The class *Delaunay_triangulation_3* represents a three-dimensional Delaunay triangulation.

The user is advised to use the class *Triangulation_hierarchy_3* rather than this basic Delaunay triangulation class: it offers the same functionalities but is much more efficient for large data sets.

```
#include <CGAL/Delaunay_triangulation_3.h>
```

Parameters

The first template argument must be a model of the *DelaunayTriangulationTraits_3* concept.

The second template argument must be a model of the *TriangulationDataStructure_3* concept. It has the default value *Triangulation_data_structure_3<Triangulation_vertex_base_3<DelaunayTriangulationTraits_3>, Triangulation_cell_base_3<DelaunayTriangulationTraits_3>>*.

Inherits From

Triangulation_3<DelaunayTriangulationTraits_3, TriangulationDataStructure_3>

Types

In addition to those inherited, the following types are defined, for use by the construction of the Voronoi diagram:

```
typedef DelaunayTriangulationTraits_3::Line_3      Line;
typedef DelaunayTriangulationTraits_3::Ray_3       Ray;
typedef DelaunayTriangulationTraits_3::Plane_3     Plane;
typedef DelaunayTriangulationTraits_3::Object_3    Object;
```

Creation

```
Delaunay_triangulation_3<DelaunayTriangulationTraits_3, TriangulationDataStructure_3> dt(
DelaunayTriangulationTraits_3 traits = DelaunayTriangulationTraits_3())
```

Creates an empty Delaunay triangulation, possibly specifying a traits class *traits*.

```
Delaunay_triangulation_3<DelaunayTriangulationTraits_3, TriangulationDataStructure_3> dt( Delaunay_
triangulation_3 dt1)
```

Copy constructor.


```

template < class InputIterator >
Delaunay_triangulation_3<DelaunayTriangulationTraits_3, TriangulationDataStructure_3> dt( InputIterator
first,
                                                    InputIterator
last,
                                                    DelaunayTriangulationTraits_
3 traits = DelaunayTriangulationTraits_3())

```

Creates a Delaunay triangulation of the points specified by the iterator range $[first, last)$ of value type *Point*, possibly specifying a traits class *traits*.

Operations

Insertion

The following methods overload the corresponding methods of triangulations to ensure the empty sphere property of Delaunay triangulations.

In the degenerate case when there are cospherical points, the Delaunay triangulation is known not to be uniquely defined. In this case, CGAL chooses a particular Delaunay triangulation using a symbolic perturbation scheme [DT03].

```
Vertex_handle dt.insert( Point p, Cell_handle start = Cell_handle())
```

Inserts point p in the triangulation and returns the corresponding vertex. Similar to the insertion in a triangulation, but ensures in addition the empty sphere property of all the created faces. The optional argument *start* is used as a starting place for the search.

```
Vertex_handle dt.insert( Point p, Locate_type lt, Cell_handle loc, int li, int lj)
```

Inserts point p in the triangulation and returns the corresponding vertex. Similar to the above *insert()* function, but takes as additional parameter the return values of a previous location query. See description of *Triangulation_3::locate()*.

The following method allows one to insert several points. It returns the number of inserted points.

```

template < class InputIterator >
int dt.insert( InputIterator first, InputIterator last)

```

Inserts the points in the iterator range $[first, last)$, of value type *Point*.

Point moving

Vertex_handle *dt.move_point(Vertex_handle v, Point p)*

Moves the point stored in *v* to *p*, while preserving the Delaunay property. This performs an action semantically equivalent to *remove(v)* followed by *insert(p)*, but is supposedly faster when the point has not moved much. Returns the handle to the new vertex.

Precondition: *v* is a finite vertex of the triangulation.

Removal

When a vertex *v* is removed from a triangulation, all the cells incident to *v* must be removed, and the polyhedral region consisting of all the tetrahedra that are incident to *v* must be retriangulated. So, the problem reduces to triangulating a polyhedral region, while preserving its boundary, or to compute a *constrained* triangulation. This is known to be sometimes impossible: the Schönhardt polyhedron cannot be triangulated [She98].

However, when dealing with Delaunay triangulations, the case of such polyhedra that cannot be retriangulated cannot happen, so CGAL proposes a vertex removal.

void *dt.remove(Vertex_handle v)*

Removes the vertex *v* from the triangulation.

Note that in CGAL 3.0 we have implemented a new algorithm for retriangulating the hole after the removal. In case that you experience problems when removing a vertex, the old code can be enabled with a *#define CGAL_DELAUNAY_3_OLD_REMOVE 1*.

Precondition: *v* is a finite vertex of the triangulation.

template < typename InputIterator >

int *dt.remove(InputIterator first, InputIterator beyond)*

Removes the vertices specified by the iterator range [*first*, *beyond*) of value type *Vertex_handle*. *remove()* is called over each element of the range. The number of vertices removed is returned.

Precondition: All vertices of the range are finite vertices of the triangulation.

Queries

Bounded_side *dt.side_of_sphere(Cell_handle c, Point p)*

Returns a value indicating on which side of the circumscribed sphere of *c* the point *p* lies. More precisely, it returns:

- *ON_BOUNDED_SIDE* if *p* is inside the sphere. For an infinite cell this means that *p* lies strictly either in the half space limited by its finite facet and not containing any other point of the triangulation, or in the interior of the disk circumscribing the *finite* facet.

- *ON_BOUNDARY* if *p* on the boundary of the sphere. For an infinite cell this means that *p* lies on the circle circumscribing the *finite* facet.

- *ON_UNBOUNDED_SIDE* if *p* lies outside the sphere. For an infinite cell this means that *p* does not satisfy either of the two previous conditions.

Precondition: *dt.dimension() = 3*.

Bounded_side *dt.side_of_circle(Facet f, Point p)*

Returns a value indicating on which side of the circumscribed circle of *f* the point *p* lies. More precisely, it returns:

- in dimension 3:

– For a finite facet, *ON_BOUNDARY* if *p* lies on the circle, *ON_UNBOUNDED_SIDE* when it lies in the exterior of the disk, *ON_BOUNDED_SIDE* when it lies in its interior.

– For an infinite facet, it considers the plane defined by the finite facet of the same cell, and does the same as in dimension 2 in this plane.

- in dimension 2:

– For a finite facet, *ON_BOUNDARY* if *p* lies on the circle, *ON_UNBOUNDED_SIDE* when it lies in the exterior of the disk, *ON_BOUNDED_SIDE* when it lies in its interior.

– For an infinite facet, *ON_BOUNDARY* if the point lies on the finite edge of *f* (endpoints included), *ON_BOUNDED_SIDE* for a point in the open half plane defined by *f* and not containing any other point of the triangulation, *ON_UNBOUNDED_SIDE* elsewhere.

Precondition: *dt.dimension()* ≥ 2 and in dimension 3, *p* is coplanar with *f*.

Bounded_side *dt.side_of_circle(Cell_handle c, int i, Point p)*

Same as the previous method for facet *i* of cell *c*.

Vertex_handle *dt.nearest_vertex(Point p, Cell_handle c = Cell_handle())*

Returns any nearest vertex to the point *p*, or the default constructed handle if the triangulation is empty. The optional argument *c* is a hint specifying where to start the search.

Precondition: *c* is a cell of *dt*.

Vertex_handle *dt.nearest_vertex_in_cell(Point p, Cell_handle c)*

Returns the vertex of the cell *c* that is nearest to *p*.

A point *p* is said to be in conflict with a cell *c* in dimension 3 (resp. a facet *f* in dimension 2) iff *dt.side_of_sphere(c, p)* (resp. *dt.side_of_circle(f, p)*) returns *ON_BOUNDED_SIDE*. The set of cells (resp. facets in dimension 2) which are in conflict with *p* is connected, and it forms a hole.

```
template <class OutputIteratorBoundaryFacets, class OutputIteratorCells>
std::pair<OutputIteratorBoundaryFacets, OutputIteratorCells>
```

```
    dt.find_conflicts( Point p,
                      Cell_handle c,
                      OutputIteratorBoundaryFacets bfit,
```

OutputIteratorCells cit)

Computes the conflict hole induced by p . The starting cell (resp. facet) c must be in conflict. Then this function returns respectively in the output iterators:

- *cit*: the cells (resp. facets) in conflict.
- *bfit*: the facets (resp. edges) on the boundary, that is, the facets (resp. edges) (t, i) where the cell (resp. facet) t is in conflict, but $t \rightarrow neighbor(i)$ is not.

This function can be used in conjunction with *insert_in_hole()* in order to decide the insertion of a point after seeing which elements of the triangulation are affected. Returns the pair composed of the resulting output iterators.

Precondition: $dt.dimension() \geq 2$, and c is in conflict with p .

template <class OutputIteratorBoundaryFacets, class OutputIteratorCells, class OutputIteratorInternalFacets>

Triple<OutputIteratorBoundaryFacets, OutputIteratorCells, OutputIteratorInternalFacets>

dt.find_conflicts(Point p,
Cell_handle c,
OutputIteratorBoundaryFacets bfit,
OutputIteratorCells cit,
OutputIteratorInternalFacets ifit)

Same as the other *find_conflicts()* function, except that it also computes the internal facets, i.e. the facets common to two cells which are in conflict with p . Then this function returns respectively in the output iterators:

- *cit*: the cells (resp. facets) in conflict.
- *bfit*: the facets (resp. edges) on the boundary, that is, the facets (resp. edges) (t, i) where the cell (resp. facet) t is in conflict, but $t \rightarrow neighbor(i)$ is not.
- *ifit*: the facets (resp. edges) inside the hole, that is, delimiting two cells (resp. facets) in conflict.

Returns the *Triple* composed of the resulting output iterators.

Precondition: $dt.dimension() \geq 2$, and c is in conflict with p .

template <class OutputIterator>
OutputIterator

dt.vertices_in_conflict(Point p, Cell_handle c, OutputIterator res)

Similar to *find_conflicts()*, but reports the vertices which are on the boundary of the conflict hole of p , in the output iterator *res*. Returns the resulting output iterator.

Precondition: $dt.dimension() \geq 2$, and c is in conflict with p .

A face (cell, facet or edge) is said to be a Gabriel face iff its smallest circumscribing sphere do not enclose any vertex of the triangulation. Any Gabriel face belongs to the Delaunay triangulation, but the reciprocal is not true. The following member functions test the Gabriel property of Delaunay faces.

bool dt.is_Gabriel(Cell_handle c, int i)
bool dt.is_Gabriel(Cell_handle c, int i, int j)
bool dt.is_Gabriel(Facet f)
bool dt.is_Gabriel(Edge e)

Voronoi diagram

CGAL offers several functionalities to display the Voronoi diagram of a set of points in 3D.

Note that the user should use a kernel with exact constructions in order to guarantee the computation of the Voronoi diagram (as opposed to computing the triangulation only, which requires only exact predicates).

Point *dt.dual(Cell_handle c)*

Returns the circumcenter of the four vertices of *c*.
Precondition: dt.dimension() = 3 and c is not infinite.

Object *dt.dual(Facet f)*

Returns the dual of facet *f*, which is
in dimension 3: either a segment, if the two cells incident to *f* are finite, or a ray, if one of them is infinite;
in dimension 2: a point.
Precondition: dt.dimension() ≥ 2 and f is not infinite.

Object *dt.dual(Cell_handle c, int i)*

same as the previous method for facet *(c,i)*.

template <class Stream>
Stream& *dt.draw_dual(Stream & os)*

Sends the set of duals to all the facets of *dt* into *os*.

————— *advanced* —————

Checking

bool *dt.is_valid(bool verbose = false)*

Checks the combinatorial validity of the triangulation and the validity of its geometric embedding (see Section 22.1). Also checks that all the circumscribing spheres (resp. circles in dimension 2) of cells (resp. facets in dimension 2) are empty.
When *verbose* is set to true, messages describing the first invalidity encountered are printed.

bool *dt.is_valid(Cell_handle c, bool verbose = false)*

Checks the combinatorial and geometric validity of the cell (see Section 22.1). Also checks that the circumscribing sphere (resp. circle in dimension 2) of cells (resp. facet in dimension 2) is empty.
When *verbose* is set to true, messages are printed to give a precise indication of the kind of invalidity encountered.

These methods are mainly a debugging help for the users of advanced features.

_____ *advanced* _____

See Also

CGAL::Triangulation_hierarchy_3.

CGAL::Triangulation_hierarchy_3<Tr>

Definition

The class *Triangulation_hierarchy_3* implements a triangulation augmented with a data structure which allows fast point location queries. As proved in [Dev02], this structure has an optimal behavior when it is built for Delaunay triangulations. It can however be used for other triangulations.

```
#include <CGAL/Triangulation_hierarchy_3.h>
```

Parameters

It is templated by a parameter which must be instantiated by one of the CGAL triangulation classes. *In the current implementation, only Delaunay_triangulation_3 is supported for Tr.*

Tr::Vertex has to be a model of the concept *TriangulationHierarchyVertexBase_3*.

Tr::Geom_traits has to be a model of the concept *DelaunayTriangulationTraits_3*.

Inherits From

Tr

Triangulation_hierarchy_3<Tr> offers exactly the same functionalities as *Tr*. Most of them (point location, insertion, removal...) are overloaded to improve their efficiency by using the hierarchic structure.

Note that, since the algorithms that are provided are randomized, the running time of constructing a triangulation with a hierarchy may be improved when shuffling the data points.

However, the I/O operations are not overloaded. So, writing a hierarchy into a file will lose the hierarchic structure and reading it from the file will result in an ordinary triangulation whose efficiency will be the same as *Tr*.

Implementation

The data structure is a hierarchy of triangulations. The triangulation at the lowest level is the original triangulation where operations and point location are to be performed. Then at each succeeding level, the data structure stores a triangulation of a small random sample of the vertices of the triangulation at the preceeding level. Point location is done through a top-down nearest neighbor query. The nearest neighbor query is first performed naively in the top level triangulation. Then, at each following level, the nearest neighbor at that level is found through a linear walk performed from the nearest neighbor found at the preceeding level. Because the number of vertices in each triangulation is only a small fraction of the number of vertices of the preceeding triangulation the data structure remains small and achieves fast point location queries on real data.

See Also

CGAL::Triangulation_hierarchy_vertex_base_3

CGAL::Delaunay_triangulation_3

CGAL::Regular_triangulation_3<RegularTriangulationTraits_3, TriangulationDataStructure_3>

Definition

Let $S^{(w)}$ be a set of weighted points in \mathbb{R}^3 . Let $p^{(w)} = (p, w_p)$, $p \in \mathbb{R}^3$, $w_p \in \mathbb{R}$ and $z^{(w)} = (z, w_z)$, $z \in \mathbb{R}^3$, $w_z \in \mathbb{R}$ be two weighted points. A weighted point $p^{(w)} = (p, w_p)$ can also be seen as a sphere of center p and radius w_p . The *power product* (or *power distance*) between $p^{(w)}$ and $z^{(w)}$ is defined as

$$\Pi(p^{(w)}, z^{(w)}) = \|p - z\|^2 - w_p - w_z$$

where $\|p - z\|$ is the Euclidean distance between p and z . $p^{(w)}$ and $z^{(w)}$ are said to be *orthogonal* if $\Pi(p^{(w)} - z^{(w)}) = 0$ (see Figure 22.2).

Four weighted points have a unique common orthogonal weighted point called the *power sphere*. A sphere $z^{(w)}$ is said to be *regular* if $\forall p^{(w)} \in S^{(w)}, \Pi(p^{(w)} - z^{(w)}) \geq 0$.

A triangulation of $S^{(w)}$ is *regular* if the power spheres of all simplices are regular.

```
#include <CGAL/Regular_triangulation_3.h>
```

Parameters

The first template argument must be a model of the *RegularTriangulationTraits_3* concept.

The second template argument must be a model of the *TriangulationDataStructure_3* concept. It has the default value *Triangulation_data_structure_3<Triangulation_vertex_base_3<RegularTriangulationTraits_3>, Triangulation_cell_base_3<RegularTriangulationTraits_3>>*.

Inherits From

Triangulation_3<RegularTriangulationTraits_3, TriangulationDataStructure_3>

Types

```
typedef RegularTriangulationTraits_3::Bare_point      Bare_point;      The type for points  $p$  of
                                                    weighted points  $p^{(w)} = (p, w_p)$ 

typedef RegularTriangulationTraits_3::Weighted_point_3 Weighted_point;
```

Creation

```
Regular_triangulation_3<RegularTriangulationTraits_3, TriangulationDataStructure_3>      rt(
RegularTriangulationTraits_3 traits = RegularTriangulationTraits_3())
```

Creates an empty regular triangulation, possibly specifying a traits class *traits*.

Regular_triangulation_3<*RegularTriangulationTraits_3*,*TriangulationDataStructure_3*> *rt*(*Regular_triangulation_3* *rt1*)

Copy constructor.

```
template < class InputIterator >
Regular_triangulation_3<RegularTriangulationTraits_3,TriangulationDataStructure_3> rt( InputIterator first,
                                           InputIterator last,
                                           RegularTriangulationTraits_3 traits = RegularTriangulationTraits_3())
```

Creates a regular triangulation of the points specified by the iterator range [*first,last*) of value type *Weighted_point*, possibly specifying a traits class *traits*.

Operations

Insertion

The following methods, which already exist in triangulations, are overloaded to ensure the property that all power spheres are regular.

Vertex_handle *rt.insert*(*Weighted_point* *p*, *Cell_handle* *start* = *Cell_handle*())

Inserts weighted point *p* in the triangulation. If this insertion creates a vertex, this vertex is returned. Otherwise, this method returns the default constructed handle. If *p* coincides with an existing vertex and has a greater weight, then *p* replaces that point and the triangulation is updated. The optional argument *start* is used as a starting place for the search.

Vertex_handle *rt.insert*(*Weighted_point* *p*, *Locate_type* *lt*, *Cell_handle* *loc*, *int* *li*, *int* *lj*)

Inserts weighted point *p* in the triangulation and returns the corresponding vertex. Similar to the above *insert*() function, but takes as additional parameter the return values of a previous location query. See description of *Triangulation_3::locate*().

The following method allows one to insert several points.

```
template < class InputIterator >
int                    rt.insert( InputIterator first, InputIterator last)
```

Inserts the weighted points in the range [*first, last*). It returns the difference of the number of vertices between after and before the insertions (it may be negative due to hidden points).

Precondition: The *value_type* of *first* and *last* is *Point*.

Removal

```
void      rt.remove( Vertex_handle v)
```

Removes the vertex v from the triangulation.

Queries

Let us remark that

$$\Pi(p^{(w)} - z^{(w)}) > 0$$

is equivalent to

p lies outside the sphere with center z and radius $\sqrt{w_p^2 + w_z^2}$.

This remark helps provide an intuition about the following predicates.

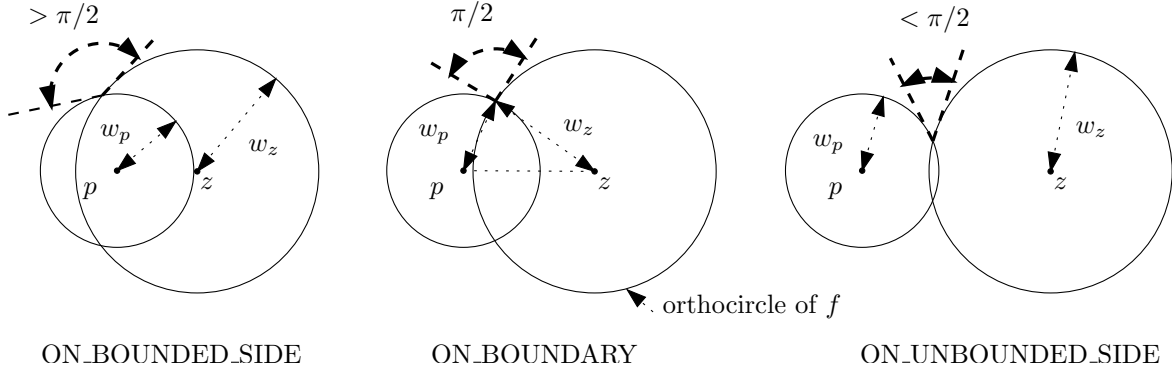


Figure 22.9: side_of_power_circle.

```
Bounded_side  rt.side_of_power_sphere( Cell_handle c, Weighted_point p)
```

Returns the position of the weighted point p with respect to the power sphere of c . More precisely, it returns:

- *ON_BOUNDED_SIDE* if $\Pi(p^{(w)} - z(c)^{(w)}) < 0$ where $z(c)^{(w)}$ is the power sphere of c . For an infinite cell this means either that p lies strictly in the half space limited by its finite facet and not containing any other point of the triangulation, or that the angle between p and the power circle of the *finite* facet of c is greater than $\pi/2$.
- *ON_BOUNDARY* if p is orthogonal to the power sphere of c i.e. $\Pi(p^{(w)} - z(c)^{(w)}) = 0$. For an infinite cell this means that p is orthogonal to the power circle of its *finite* facet.
- *ON_UNBOUNDED_SIDE* if $\Pi(p^{(w)} - z(c)^{(w)}) > 0$ i.e. the angle between the weighted point p and the power sphere of c is less than $\pi/2$ or if these two spheres do not intersect. For an infinite cell this means that p does not satisfy either of the two previous conditions.

Precondition: $rt.dimension() = 3$.

Bounded_side *rt.side_of_power_circle(Facet f, Weighted_point p)*

Returns the position of the point p with respect to the power circle of f .
 More precisely, it returns:
 — in dimension 3:
 – For a finite facet,
ON_BOUNDARY if p is orthogonal to the power circle in the plane of the facet,
ON_UNBOUNDED_SIDE when their angle is less than $\pi/2$,
ON_BOUNDED_SIDE when it is greater than $\pi/2$ (see Figure 22.9).
 – For an infinite facet, it considers the plane defined by the finite facet of the cell f_{first} , and does the same as in dimension 2 in this plane.
 — in dimension 2:
 – For a finite facet,
ON_BOUNDARY if p is orthogonal to the circle,
ON_UNBOUNDED_SIDE when the angle between p and the power circle of f is less than $\pi/2$, *ON_BOUNDED_SIDE* when it is greater than $\pi/2$.
 – For an infinite facet,
ON_BOUNDED_SIDE for a point in the open half plane defined by f and not containing any other point of the triangulation,
ON_UNBOUNDED_SIDE in the other open half plane.
 If the point p is collinear with the finite edge e of f , it returns:
ON_BOUNDED_SIDE if $\Pi(p^{(w)} - z(e)^{(w)}) < 0$, where $z(e)^{(w)}$ is the power segment of e in the line supporting e ,
ON_BOUNDARY if $\Pi(p^{(w)} - z(e)^{(w)}) = 0$,
ON_UNBOUNDED_SIDE if $\Pi(p^{(w)} - z(e)^{(w)}) > 0$.
Precondition: rt.dimension() \geq 2.

Bounded_side *rt.side_of_power_circle(Cell_handle c, int i, Weighted_point p)*

Same as the previous method for facet i of cell c .

Bounded_side *rt.side_of_power_segment(Cell_handle c, Weighted_point p)*

In dimension 1, returns
ON_BOUNDED_SIDE if $\Pi(p^{(w)} - z(c)^{(w)}) < 0$, where $z(c)^{(w)}$ is the power segment of the edge represented by c ,
ON_BOUNDARY if $\Pi(p^{(w)} - z(c)^{(w)}) = 0$,
ON_UNBOUNDED_SIDE if $\Pi(p^{(w)} - z(c)^{(w)}) > 0$.
Precondition: rt.dimension() = 1.

Vertex_handle

rt.nearest_power_vertex(Point p, Cell_handle c = Cell_handle())

Returns the vertex of the triangulation which is nearest to p with respect to the power distance. This means that the power of the query point p with respect to the weighted point in the returned vertex is smaller than the power of p with respect to the weighted point in any other vertex. Ties are broken arbitrarily. The default constructed handle is returned if the triangulation is empty. The optional argument c is a hint specifying where to start the search.

Precondition: c is a cell of rt .

Vertex_handle

rt.nearest_power_vertex_in_cell(Point p, Cell_handle c)

Returns the vertex of the cell c that is nearest to p with respect to the power distance.

A weighted point p is said to be in conflict with a cell c in dimension 3 (resp. with a facet f in dimension 2) if it has a negative power distance to the power sphere of c (resp. to the power circle of f). The set of cells (resp. facets in dimension 2) which are in conflict with p is connected.

template <class OutputIteratorBoundaryFacets, class OutputIteratorCells, class OutputIteratorInternalFacets>

Triple<OutputIteratorBoundaryFacets, OutputIteratorCells, OutputIteratorInternalFacets>

*rt.find_conflicts(const Weighted_point p,
Cell_handle c,
OutputIteratorBoundaryFacets bfit,
OutputIteratorCells cit,
OutputIteratorInternalFacets ifit)*

Compute the conflicts with p . The starting cell (resp. facet) c must be in conflict with p . Then this function returns respectively in the output iterators:

- *cit*: the cells (resp. facets) in conflict with p .
- *bfit*: the facets (resp. edges) on the boundary of the conflict zone, that is, the facets (resp. edges) (t, i) where the cell (resp. facet) t is in conflict, but $t \rightarrow neighbor(i)$ is not.
- *ifit*: the facets (resp. edges) inside the conflict zone, that facets incident to two cells (resp. facets) in conflict.

Returns the *Triple* composed of the resulting output iterators.

Precondition: $rt.dimension() \geq 2$, and c is in conflict with p .

In the weighted setting, a face (cell, facet, edge or vertex) is said to be a Gabriel face iff the smallest sphere orthogonal to the weighted points associated to its vertices, has a positive power product with the weighted point of any other vertex of the triangulation. Any weighted Gabriel face belongs to the regular triangulation, but the reciprocal is not true. The following member functions test the Gabriel property of the faces of the regular triangulation.

bool rt.is_Gabriel(Cell_handle c, int i)

bool rt.is_Gabriel(Cell_handle c, int i, int j)

bool *rt.is_Gabriel(Facet f)*
bool *rt.is_Gabriel(Edge e)*
bool *rt.is_Gabriel(Vertex_handle v)*

Power diagram

CGAL offers several functionalities to display the Power diagram of a set of points in 3D.

Note that the user should use a kernel with exact constructions in order to guarantee the computation of the Voronoi diagram (as opposed to computing the triangulation only, which requires only exact predicates).

Point *rt.dual(Cell_handle c)*

Returns the weighted circumcenter of the four vertices of *c*.
Precondition: rt.dimension()= 3 and c is not infinite.

Object *rt.dual(Facet f)* Returns the dual of facet *f*, which is
 in dimension 3: either a segment, if the two cells incident to *f* are finite,
 or a ray, if one of them is infinite;
 in dimension 2: a point.
Precondition: rt.dimension() ≥ 2 and f is not infinite.

Object *rt.dual(Cell_handle c, int i)*

same as the previous method for facet *(c,i)*.

template <class Stream>
Stream& *rt.draw_dual(Stream & os)*

Sends the set of duals to all the facets of *rt* into *os*.

————— *advanced* —————

Checking

bool *rt.is_valid(bool verbose = false)*

Checks the combinatorial validity of the triangulation and the validity of its geometric embedding (see Section 22.1). Also checks that all the power spheres (resp. power circles in dimension 2, power segments in dimension 1) of cells (resp. facets in dimension 2, edges in dimension 1) are regular. When *verbose* is set to true, messages describing the first invalidity encountered are printed.

This method is mainly a debugging help for the users of advanced features.

————— *advanced* —————

TriangulationTraits_3

Definition

The concept `TriangulationTraits_3` is the first template parameter of the class `Triangulation_3`. It defines the geometric objects (points, segments, triangles and tetrahedra) forming the triangulation together with a few geometric predicates and constructions on these objects : lexicographical comparison, orientation in case of coplanar points and orientation in space.

Types

`TriangulationTraits_3::Point_3` The point type. It must be *DefaultConstructible*, *CopyConstructible* and *Assignable*.

`TriangulationTraits_3::Segment_3` The segment type.

`TriangulationTraits_3::Tetrahedron_3` The tetrahedron type.

`TriangulationTraits_3::Triangle_3` The triangle type.

`TriangulationTraits_3::Construct_segment_3`

A constructor object that must provide the function operator
Segment_3 operator()(Point_3 p, Point_3 q),
which constructs a segment from two points.

`TriangulationTraits_3::Construct_triangle_3`

A constructor object that must provide the function operator
Triangle_3 operator()(Point_3 p, Point_3 q, Point_3 r),
which constructs a triangle from three points.

`TriangulationTraits_3::Construct_tetrahedron_3`

A constructor object that must provide the function operator
Tetrahedron_3 operator()(Point_3 p, Point_3 q, Point_3 r, Point_3 s),
which constructs a tetrahedron from four points.

`TriangulationTraits_3::Compare_xyz_3`

A predicate object that must provide the function operator
Comparison_result operator()(Point p, Point q),
which returns *EQUAL* if the two points are equal. Otherwise it must return a consistent order for any two points chosen in a same line.

`TriangulationTraits_3::Coplanar_orientation_3`

A predicate object that must provide the function operator
Orientation operator()(Point p, Point q, Point r),
which returns *COLLINEAR* if the points are collinear. Otherwise it must return a consistent orientation for any three points chosen in a same plane.

TriangulationTraits_3:: Orientation_3

A predicate object that must provide the function operator
Orientation operator()(Point p, Point q, Point r, Point s),
which returns POSITIVE, if *s* lies on the positive side of the oriented plane *h* defined
by *p*, *q*, and *r*, returns NEGATIVE if *s* lies on the negative side of *h*, and returns
COPLANAR if *s* lies on *h*.

Creation

<i>TriangulationTraits_3 traits;</i>	Default constructor.
<i>TriangulationTraits_3 traits(Triangulation_traits_3 tr);</i>	Copy constructor.

Operations

The following functions give access to the predicate and construction objects:

<i>Construct_tetrahedron_3</i>	<i>traits.construct_tetrahedron_3_object()</i>
<i>Construct_triangle_3</i>	<i>traits.construct_triangle_3_object()</i>
<i>Construct_segment_3</i>	<i>traits.construct_segment_3_object()</i>
<i>Compare_xyz_3</i>	<i>traits.compare_xyz_3_object()</i>
<i>Coplanar_orientation_3</i>	<i>traits.coplanar_orientation_3_object()</i>
<i>Orientation_3</i>	<i>traits.orientation_3_object()</i>

Has Models

CGAL::Exact_predicates_inexact_constructions_kernel (recommended)
CGAL::Exact_predicates_exact_constructions_kernel
CGAL::Filtered_kernel
CGAL::Cartesian
CGAL::Simple_cartesian
CGAL::Homogeneous
CGAL::Simple_homogeneous

DelaunayTriangulationTraits_3

Definition

The concept `DelaunayTriangulationTraits_3` is the first template parameter of the class `Delaunay_triangulation_3`. It defines the geometric objects (points, segments...) forming the triangulation together with a few geometric predicates and constructions on these objects.

Refines

TriangulationTraits_3

In addition to the requirements described for the traits class of *Triangulation_3*, the geometric traits class of a Delaunay triangulation must fulfill the following requirements:

Types

<code>DelaunayTriangulationTraits_3:: Line_3</code>	The line type.
<code>DelaunayTriangulationTraits_3:: Object_3</code>	The object type.
<code>DelaunayTriangulationTraits_3:: Plane_3</code>	The plane type.
<code>DelaunayTriangulationTraits_3:: Ray_3</code>	The ray type.

`DelaunayTriangulationTraits_3:: Coplanar_side_of_bounded_circle_3`

A predicate object that must provide the function operator `Bounded_side operator()(Point p, Point q, Point r, Point s)`, which determines the bounded side of the circle defined by p , q , and r on which s lies.

Precondition: p , q , r , and s are coplanar and p , q , and r are not collinear.

`DelaunayTriangulationTraits_3:: Side_of_oriented_sphere_3`

A predicate object that must provide the function operator `Oriented_side operator()(Point p, Point q, Point r, Point s, Point t)`, which determines on which side of the oriented sphere circumscribing p , q , r , s the point t lies.

The following additional predicate is required only when the *Triangulation_hierarchy_3* is used on top of *Delaunay_triangulation_3*:

`DelaunayTriangulationTraits_3:: Compare_distance_3`

A predicate object that must provide the function operator `Comparison_result operator()(Point p, Point q, Point r)`, which compares the distance between p and q to the distance between p and r .

In addition, only when the dual operations are used, the traits class must provide the following constructor objects:

DelaunayTriangulationTraits_3:: Construct_circumcenter_3

A constructor object that must provide the function operator
Point_3 operator()(Point_3 p, Point_3 q, Point_3 r, Point_3 s),
which constructs the circumcenter of four points.

Precondition: *p, q, r* and *s* must be non coplanar.

It must also provide the function operator
Point_3 operator()(Point_3 p, Point_3 q, Point_3 r),
which constructs the circumcenter of three points.

Precondition: *p, q* and *r* must be non collinear.

DelaunayTriangulationTraits_3:: Construct_object_3

A constructor object that must provide the function operators
Object_3 operator()(Point_3 p),
Object_3 operator()(Segment_3 s) and
Object_3 operator()(Ray_3 r)
that construct an object respectively from a point, a segment and a ray.

DelaunayTriangulationTraits_3:: Construct_perpendicular_line_3

A constructor object that must provide the function operator
Line_3 operator()(Plane_3 pl, Point_3 p),
which constructs the line perpendicular to *pl* passing through *p*.

DelaunayTriangulationTraits_3:: Construct_plane_3

A constructor object that must provide the function operator
Plane_3 operator()(Point_3 p, Point_3 q, Point_3 r),
which constructs the plane passing through *p, q* and *r*.
Precondition: *p, q* and *r* are non collinear.

DelaunayTriangulationTraits_3:: Construct_ray_3

A constructor object that must provide the function operator
Ray_3 operator()(Point_3 p, Line_3 l),
which constructs the ray starting at *p* with direction given by *l*.

Operations

The following functions give access to the predicate and construction objects:

<i>Coplanar_side_of_bounded_circle_3</i>	<i>traits.coplanar_side_of_bounded_circle_3_object()</i>
<i>Side_of_oriented_sphere_3</i>	<i>traits.side_of_oriented_sphere_3_object()</i>

When using the triangulation hierarchy, the traits must provide:

<i>Compare_distance_3</i>	<i>traits.compare_distance_3_object()</i>
---------------------------	-------------------------------------------

The following functions must be provided only if the methods of *DelaunayTriangulation_3* returning elements of the Voronoi diagram are instantiated:

<i>Construct_circumcenter_3</i>	<i>traits.construct_circumcenter_3_object()</i>
<i>Construct_object_3</i>	<i>traits.construct_object_3_object()</i>
<i>Construct_perpendicular_line_3</i>	<i>traits.construct_perpendicular_line_object()</i>
<i>Construct_plane_3</i>	<i>traits.construct_plane_3_object()</i>

Construct_ray_3

traits.construct_ray_3_object()

Has Models

CGAL::Exact_predicates_inexact_constructions_kernel (recommended)

CGAL::Exact_predicates_exact_constructions_kernel (recommended for Voronoi)

CGAL::Filtered_kernel

CGAL::Cartesian

CGAL::Simple_cartesian

CGAL::Homogeneous

CGAL::Simple_homogeneous

RegularTriangulationTraits_3

Definition

The concept `RegularTriangulationTraits_3` is the first template parameter of the class *Regular_triangulation_3*. It defines the geometric objects (points, segments...) forming the triangulation together with a few geometric predicates and constructions on these objects.

Refines

TriangulationTraits_3

In addition to the requirements described for the traits class of *Triangulation_3*, the geometric traits class of *Regular_triangulation_3* must fulfill the following requirements.

Types

<i>RegularTriangulationTraits_3:: Line_3</i>	The line type.
<i>RegularTriangulationTraits_3:: Object_3</i>	The object type.
<i>RegularTriangulationTraits_3:: Plane_3</i>	The plane type.
<i>RegularTriangulationTraits_3:: Ray_3</i>	The ray type.

We use here the same notation as in Section 22.3. To simplify notation, p will often denote in the sequel either the point $p \in \mathbb{R}^3$ or the weighted point $p^{(w)} = (p, w_p)$.

<i>RegularTriangulationTraits_3:: Weighted_point_3</i>	The weighted point type.
--------------------------------------------------------	--------------------------

RegularTriangulationTraits_3:: Power_test_3

A predicate object which must provide the following function operators:

Oriented_side operator()(*Weighted_point_3* *p*, *Weighted_point_3* *q*, *Weighted_point_3* *r*, *Weighted_point_3* *s*, *Weighted_point_3* *t*),

which performs the following:

Let $z(p, q, r, s)^{(w)}$ be the power sphere of the weighted points (p, q, r, s) . Returns

ON_ORIENTED_BOUNDARY if *t* is orthogonal to $z(p, q, r, s)^{(w)}$,

ON_NEGATIVE_SIDE if *t* lies outside the oriented sphere of center $z(p, q, r, s)$ and radius $\sqrt{w_{z(p,q,r,s)}^2 + w_t^2}$ (which is equivalent to $\Pi(t^{(w)}, z(p, q, r, s)^{(w)}) > 0$),

ON_POSITIVE_SIDE if *t* lies inside this oriented sphere.

Precondition: *p*, *q*, *r*, *s* are not coplanar. Note that with this definition, if all the points have a weight equal to 0, then $power_test(p, q, r, s, t) = side_of_oriented_sphere(p, q, r, s, t)$.

Oriented_side operator()(*Weighted_point_3* *p*, *Weighted_point_3* *q*, *Weighted_point_3* *r*, *Weighted_point_3* *t*),

which has an definition analogous to the previous method, for coplanar points, with the power circle $z(p, q, r)^{(w)}$.

Precondition: *p*, *q*, *r* are not collinear and *p*, *q*, *r*, *t* are coplanar. If all the points have a weight equal to 0, then $power_test(p, q, r, s, t) = side_of_oriented_circle(p, q, r, s, t)$.

Oriented_side operator()(*Weighted_point_3* *p*, *Weighted_point_3* *q*, *Weighted_point_3* *t*),

which is the same for collinear points, where $z(p, q)^{(w)}$ is the power segment of *p* and *q*.

Precondition: *p* and *q* have different Bare_points, and *p*, *q*, *t* are collinear. If all points have a weight equal to 0, then $power_test(p, q, t)$ gives the same answer as the kernel predicate $s(p, q).has_on(t)$ would give, where $s(p, q)$ denotes the segment with endpoints *p* and *q*.

Oriented_side operator()(*Weighted_point_3* *p*, *Weighted_point_3* *q*),

which is the same for equal points, that is when *p* and *q* have equal coordinates, then it returns the comparison of the weights (*ON_POSITIVE_SIDE* when *q* is heavier than *p*).

Precondition: *p* and *q* have equal Bare_points.

The following predicate is required if a call to *nearest_power_vertex* or *nearest_power_vertex_in_cell* is issued:

RegularTriangulationTraits_3:: Compare_power_distance_3

A predicate object that must provide the function operator

Comparison_result operator()(*Point_3* *p*, *Weighted_point_3* *q*, *Weighted_point_3* *r*),

which compares the power distance between *p* and *q* to the power distance between *p* and *r*.

In addition, only when the dual operations are used, the traits class must provide the following constructor objects:

RegularTriangulationTraits_3:: Construct_weighted_circumcenter_3

A constructor type. The operator() constructs the bare point which is the center of the smallest orthogonal sphere to the input weighted points.

Bare_point operator()(*Weighted_point_3* *p*, *Weighted_point_3* *q*, *Weighted_point_3* *r*, *Weighted_point_3* *s*);

RegularTriangulationTraits_3:: Construct_object_3

A constructor object that must provide the function operators
Object_3 operator()(Point_3 p),
Object_3 operator()(Segment_3 s) and
Object_3 operator()(Ray_3 r)
that construct an object respectively from a point, a segment and a ray.

RegularTriangulationTraits_3:: Construct_perpendicular_line_3

A constructor object that must provide the function operator
Line_3 operator()(Plane_3 pl, Point_3 p),
which constructs the line perpendicular to *pl* passing through *p*.

RegularTriangulationTraits_3:: Construct_plane_3

A constructor object that must provide the function operator
Plane_3 operator()(Point_3 p, Point_3 q, Point_3 r),
which constructs the plane passing through *p*, *q* and *r*.
Precondition: *p*, *q* and *r* are non collinear.

RegularTriangulationTraits_3:: Construct_ray_3

A constructor object that must provide the function operator
Ray_3 operator()(Point_3 p, Line_3 l),
which constructs the ray starting at *p* with direction given by *l*.

Operations

The following function gives access to the predicate object:

Power_test_3 *traits.power_test_3_object()*

The following functions must be provided only if the member functions of *Regular_triangulation_3* returning elements of the dual diagram are called:

Construct_weighted_circumcenter_3

traits.construct_weighted_circumcenter_3_object()

Construct_object_3 *traits.construct_object_3_object()*

Construct_perpendicular_line_3

traits.construct_perpendicular_line_object()

Construct_plane_3 *traits.construct_plane_3_object()*

Construct_ray_3 *traits.construct_ray_3_object()*

Has Models

CGAL::Regular_triangulation_euclidean_traits_3

CGAL::Regular_triangulation_filtered_traits_3.

CGAL::Regular_triangulation_euclidean_traits_3<K,Weight>

Definition

The class *Regular_triangulation_euclidean_traits_3*<K,Weight> is designed as a default traits class for the class *Regular_triangulation_3*<*RegularTriangulationTraits_3*,*TriangulationDataStructure_3*>. It provides *Weighted_point_3*, a class for weighted points, which derives from the three dimensional point class *K::Point_3*.

The first argument *K* must be a model of the *Kernel* concept.

The second argument *Weight* of the class *Regular_triangulation_euclidean_traits_3*<K,Weight> is in fact optional: if it is not provided, *K::RT* will be used.

The class is a model of the concept *RegularTriangulationTraits_3* but it also contains predicates and constructors on weighted points that are not required in the concept *RegularTriangulationTraits_3*.

Note that this template class is specialized for *CGAL::Exact_predicates_inexact_constructions_kernel*, so that it is as if *Regular_triangulation_filtered_traits_3* was used, i.e. you get filtered predicates automatically.

```
#include <CGAL/Regular_triangulation_euclidean_traits_3.h>
```

Is Model for the Concepts

RegularTriangulationTraits_3

Inherits From

K

Types

<code>typedef K::Point_3</code>	<code>Bare_point;</code>	The type for point p of a weighted point $p^{(w)} = (p, w_p)$.
<code>typedef Weighted_point <Bare_point, Weight></code>	<code>Weighted_point_3;</code>	The type for weighted points.

Types for predicate functors

Regular_triangulation_euclidean_traits_3<K,Weight>:: *Power_test_3*

A predicate type for power test. Belongs to the *RegularTriangulationTraits_3* concept.

Regular_triangulation_euclidean_traits_3<K,Weight>:: *Compare_power_distance_3*

A predicate type to compare power distance. Belongs to the *RegularTriangulationTraits_3* concept.

Regular_triangulation_euclidean_traits_3<K,Weight>:: In_smallest_orthogonal_sphere_3

A predicate type. The operator() takes weighted points as arguments and returns the sign of the power distance of the last one with respect to the smallest sphere orthogonal to the others.

Sign operator()(*Weighted_point_3* p, *Weighted_point_3* q, *Weighted_point_3* r, *Weighted_point_3* s, *Weighted_point_3* t) ;

Sign operator()(*Weighted_point_3* p, *Weighted_point_3* q, *Weighted_point_3* r, *Weighted_point_3* s) ;

Sign operator()(*Weighted_point_3* p, *Weighted_point_3* q, *Weighted_point_3* r) ;

Sign operator()(*Weighted_point_3* p, *Weighted_point_3* q) ;

Regular_triangulation_euclidean_traits_3<K,Weight>:: Side_of_bounded_orthogonal_sphere_3

A predicate type. The operator() is similar to the operator() of *In_smallest_orthogonal_sphere_3* except that the returned type is not a *Sign* but belongs to the enum *Bounded_side* (A *NEGATIVE*, *ccnNULL* and *POSITIVE*) corresponding respectively to *ON_BOUNDED_SIDE*, *ON_BOUNDARY* and *ON_UNBOUNDED_SIDE*)).

Bounded_side operator()(*Weighted_point_3* p, *Weighted_point_3* q, *Weighted_point_3* r, *Weighted_point_3* s, *Weighted_point_3* t) ;

Bounded_side operator()(*Weighted_point_3* p, *Weighted_point_3* q, *Weighted_point_3* r, *Weighted_point_3* s) ;

Bounded_side operator()(*Weighted_point_3* p, *Weighted_point_3* q, *Weighted_point_3* r) ;

Regular_triangulation_euclidean_traits_3<K,Weight>:: Does_simplex_intersect_dual_support_3

A predicate type. The operator() takes weighted points as arguments, considers the subspace of points with equal power distance with respect to its arguments and the intersection of this subspace with the affine hull of the bare points associated to the arguments. The operator() returns *ON_BOUNDED_SIDE*, *ON_BOUNDARY* or *ON_UNBOUNDED_SIDE* according to the position of this intersection with respect to the simplex formed by the bare points. This predicate is usefull for flow computations.

Bounded_side operator()(*Weighted_point_3* p, *Weighted_point_3* q, *Weighted_point_3* r, *Weighted_point_3* s) ;

Bounded_side operator()(*Weighted_point_3* p, *Weighted_point_3* q, *Weighted_point_3* r) ;

Bounded_side operator()(*Weighted_point_3* p, *Weighted_point_3* q) ;

Types for constructor functors

Regular_triangulation_euclidean_traits_3<K,Weight>:: Construct_weighted_circumcenter_3

A constructor type. The operator() constructs the bare point which is the center of the smallest orthogonal sphere to the input weighted points.

Bare_point operator()(*Weighted_point_3* p, *Weighted_point_3* q, *Weighted_point_3* r, *Weighted_point_3* s) ;

Bare_point operator()(*Weighted_point_3* p, *Weighted_point_3* q, *Weighted_point_3* r) ;

Bare_point operator()(*Weighted_point_3* p, *Weighted_point_3* q) ;

Regular_triangulation_euclidean_traits_3<K,Weight>:: Compute_power_product_3

A functor type. The operator() computes the power distance between its arguments.

FT operator()(*Weighted_point_3* p, *Weighted_point_3* q) ;

Regular_triangulation_euclidean_traits_3<K,Weight>:: *Compute_squared_radius_smallest_orthogonal_sphere_3*

A functor type. The operator() computes the squared radius of the smallest sphere orthogonal to the arguments.

FT operator() (*Weighted_point_3 p*, *Weighted_point_3 q*, *Weighted_point_3 r*, *Weighted_point_3 s*);

FT operator() (*Weighted_point_3 p*, *Weighted_point_3 q*, *Weighted_point_3 r*);

FT operator() (*Weighted_point_3 p*, *Weighted_point_3 q*);

Regular_triangulation_euclidean_traits_3<K,Weight>:: *Compute_critical_squared_radius_3*

A functor type. The operator() takes weighted points as arguments and computes the squared radius of the sphere centered in the last point and orthogonal to the other weighted points. The last argument is a weighted point but its weight does not matter. This construction is adhoc for pumping slivers. For robustness issue, a predicate to compare critical squared radii for a given last point should be needed.

FT operator() (*Weighted_point_3 p*, *Weighted_point_3 q*, *Weighted_point_3 r*, *Weighted_point_3 s*, *Weighted_point_3 t*);

Operations

The following functions give access to the predicate and constructor functors.

<i>Power_test_3</i>	<i>traits.power_test_3_object()</i>
<i>Compare_power_distance_3</i>	<i>traits.compare_power_distance_3_object()</i>
<i>In_smallest_orthogonal_sphere_3</i>	<i>traits.in_smallest_orthogonal_sphere_3_object()</i>
<i>Side_of_bounded_orthogonal_sphere_3</i>	<i>traits.side_of_bounded_orthogonal_sphere_3_object()</i>
<i>Does_simplex_intersect_dual_support_3</i>	<i>traits.does_simplex_intersect_dual_support_3_object()</i>
<i>Construct_weighted_circumcenter_3</i>	<i>traits.construct_weighted_circumcenter_3_object()</i>
<i>Compute_power_product_3</i>	<i>traits.compute_power_product_3_object()</i>
<i>Compute_squared_radius_smallest_orthogonal_sphere_3</i>	
	<i>traits.compute_squared_radius_smallest_orthogonal_sphere_3_object()</i>
<i>Compute_critical_squared_radius_3</i>	<i>traits.compute_critical_squared_radius_3_object()</i>

See Also

CGAL::Regular_triangulation_filtered_traits_3.

CGAL::Regular_triangulation_filtered_traits_3<FK>

Definition

The class *Regular_triangulation_filtered_traits_3<FK>* is designed as a traits class for the class *Regular_triangulation_3<RegularTriangulationTraits_3, TriangulationDataStructure_3>*. Its difference with *Regular_triangulation_euclidean_traits_3* is that it provides filtered predicates which are meant to be fast and exact.

The first argument *FK* must be a model of the *Kernel* concept, and it is also restricted to be an instance of the *Filtered_kernel* template.

```
#include <CGAL/Regular_triangulation_filtered_traits_3.h>
```

Is Model for the Concepts

RegularTriangulationTraits_3

Inherits From

Regular_triangulation_euclidean_traits_3<FK>

See Also

CGAL::Regular_triangulation_euclidean_traits_3.

TriangulationCellBase_3

Definition

The cell base required by the geometric triangulations does not store any geometric information, so only the requirements of the triangulation data structure apply. However, we provide this concept for symmetry with the vertex case.

Refines

TriangulationDSCellBase_3

Has Models

CGAL::Triangulation_cell_base_3

CGAL::Triangulation_cell_base_with_info_3

See Also

TriangulationVertexBase_3

TriangulationVertexBase_3

Definition

The vertex base used by the geometric triangulation must store a point. So we list here the additional requirements compared to a vertex base usable for the triangulation data structure.

Refines

TriangulationDSVertexBase_3

Types

TriangulationVertexBase_3::Point

Must be the same as the point type *TriangulationTraits_3::Point_3* defined by the geometric traits class of the triangulation.

Creation

TriangulationVertexBase_3 *v*(*Point* *p*);

Constructs a vertex whose geometric embedding is point *p*.

TriangulationVertexBase_3 *v*(*Point* *p*, *Cell_handle* *c*);

Constructs a vertex embedding the point *p* and pointing to cell *c*.

Access Functions

Point *v*.*point*()

Returns the point.

Setting

void *v*.*set_point*(*Point* *p*)

Sets the point.

I/O

istream& *istream& is* >> & *v*

Inputs the non-combinatorial information given by the vertex: the point and other possible information.

ostream& *ostream& os* << *v*

Outputs the non-combinatorial information given by the vertex: the point and other possible information.

Has Models

CGAL::Triangulation_vertex_base_3

CGAL::Triangulation_vertex_base_with_info_3

CGAL::Triangulation_hierarchy_vertex_base_3

See Also

TriangulationCellBase_3

TriangulationHierarchyVertexBase_3

TriangulationHierarchyVertexBase_3

Definition

The vertex base used by *Triangulation_hierarchy_3* must provide access to two vertex handles for linking between the levels of the hierarchy.

Refines

TriangulationVertexBase_3

Access Functions

<i>Vertex_handle</i>	<i>v.up()</i> <i>const</i>	Returns the <i>Vertex_handle</i> pointing to the level above.
<i>Vertex_handle</i>	<i>v.down()</i> <i>const</i>	Returns the <i>Vertex_handle</i> pointing to the level below.

Setting

<i>void</i>	<i>v.set_up(Vertex_handle v)</i>	Sets the <i>Vertex_handle</i> pointing to the level above to <i>v</i> .
<i>void</i>	<i>v.set_down(Vertex_handle v)</i>	Sets the <i>Vertex_handle</i> pointing to the level below to <i>v</i> .

Has Models

CGAL::Triangulation_hierarchy_vertex_base_3

RegularTriangulationCellBase_3

Definition

The regular triangulation of a set of weighted points does not necessarily have one vertex for each of the input points. Some of the input weighted points have no cell in the dual power diagrams and therefore do not correspond to a vertex of the regular triangulation. Those weighted points are said to be *hidden* points. A point which is hidden at a given time may appear later as a vertex of the regular triangulation upon removal of some other weighted point. Therefore, hidden points have to be stored somewhere. The regular triangulation stores those hidden points in its cells.

A hidden point can appear as a vertex of the triangulation only when the three-dimensional cell where its point component is located (the cell which hides it) is removed. Therefore we decided to store in each cell of a regular triangulation the list of hidden points that are located in the face. Thus points hidden by a face are easily reinserted in the triangulation when the face is removed.

The base cell of a regular triangulation has to be a model of the concept `RegularTriangulationCellBase_3`, which refines the concept `TriangulationCellBase_3` by adding in the cell a container to store hidden points.

Refines

TriangulationCellBase_3

Types

RegularTriangulationCellBase_3::Point Must be the same as the point type *TriangulationTraits_3::Point_3* defined by the geometric traits class of the triangulation.

Types

RegularTriangulationCellBase_3::Point_iterator
 Iterator of value type `Point`

Access Functions

Point_iterator *rcb.hidden_points_begin()*
 Returns an iterator pointing to the first hidden point.

Point_iterator *rcb.hidden_points_end()*
 Returns a past-the-end iterator.

Setting

void *rcb.hide_point(Point p)*

Adds p to the set of hidden points of the cell.

Has Models

CGAL::Regular_triangulation_cell_base_3

See Also

TriangulationCellBase_3

CGAL::Triangulation_cell_base_3<TriangulationTraits_3, TriangulationDSCellBase_3>

Definition

The class *Triangulation_cell_base_3* is a model of the concept *TriangulationCellBase_3*, the base cell of a 3D-triangulation.

This class can be used directly or can serve as a base to derive other classes with some additional attributes (a color for example) tuned for a specific application.

```
#include <CGAL/Triangulation_cell_base_3.h>
```

Parameters

The first template argument is the geometric traits class *TriangulationTraits_3*. It is actually not used by this class.

The second template argument is a combinatorial cell base class from which *Triangulation_cell_base_3* derives. It has the default value *Triangulation_ds_cell_base_3*<>.

Is Model for the Concepts

TriangulationCellBase_3

Inherits From

TriangulationDSCellBase_3

See Also

CGAL::Triangulation_ds_cell_base_3

CGAL::Triangulation_cell_base_with_info_3

CGAL::Triangulation_vertex_base_3

CGAL::Triangulation_cell_base_with_info_3<Info, TriangulationTraits_3, TriangulationCellBase_3>

Definition

The class *Triangulation_cell_base_with_info_3* is a model of the concept *TriangulationCellBase_3*, the base cell of a 3D-triangulation. It provides an easy way to add some user defined information in cells. Note that input/output operators discard this additional information.

```
#include <CGAL/Triangulation_cell_base_with_info_3.h>
```

Parameters

The first template argument is the information the user would like to add to a cell. It has to be *DefaultConstructible* and *Assignable*.

The second template argument is the geometric traits class *TriangulationTraits_3*. It is actually not used by this class.

The third template argument is a cell base class from which *Triangulation_cell_base_with_info_3* derives. It has the default value *Triangulation_cell_base_3<TriangulationTraits_3>*.

Is Model for the Concepts

TriangulationCellBase_3

Inherits From

TriangulationCellBase_3

Types

```
typedef Info          Info;
```

Access Functions

<code>const Info&</code>	<code>v.info() const</code>	Returns a const reference to the object of type <i>Info</i> stored in the cell.
<code>Info&</code>	<code>v.info()</code>	Returns a reference to the object of type <i>Info</i> stored in the cell.

See Also

CGAL::Triangulation_cell_base_3

CGAL::Triangulation_vertex_base_with_info_3

CGAL::Triangulation_vertex_base_3<TriangulationTraits_3, TriangulationDSVertexBase_3>

Definition

The class *Triangulation_vertex_base_3* is a model of the concept *TriangulationVertexBase_3*, the base vertex of a 3D-triangulation. This class stores a point.

This class can be used directly or can serve as a base to derive other classes with some additional attributes (a color for example) tuned for a specific application.

```
#include <CGAL/Triangulation_vertex_base_3.h>
```

Parameters

The first template argument is the geometric traits class *TriangulationTraits_3* which provides the point type, *Point_3*. Users of the geometric triangulations (Section 23.2 and Chapter 22) are strongly advised to use the same geometric traits class *TriangulationTraits_3* as the one used for *Triangulation_3*. This way, the point type defined by the base vertex is the same as the point type defined by the geometric traits class.

The second template argument is a combinatorial vertex base class from which *Triangulation_vertex_base_3* derives. It has the default value *Triangulation_ds_vertex_base_3*<>.

Is Model for the Concepts

TriangulationVertexBase_3

Inherits From

TriangulationDSVertexBase_3

Types

```
typedef TriangulationTraits_3::Point_3          Point;
```

See Also

CGAL::Triangulation_cell_base_3
CGAL::Triangulation_ds_vertex_base_3
CGAL::Triangulation_vertex_base_with_info_3
CGAL::Triangulation_hierarchy_vertex_base_3

CGAL::Triangulation_vertex_base_with_info_3<Info, TriangulationTraits_3, TriangulationVertexBase_3>

Definition

The class *Triangulation_vertex_base_with_info_3* is a model of the concept *TriangulationVertexBase_3*, the base vertex of a 3D-triangulation. It provides an easy way to add some user defined information in vertices. Note that input/output operators discard this additional information.

```
#include <CGAL/Triangulation_vertex_base_with_info_3.h>
```

Parameters

The first template argument is the information the user would like to add to a vertex. It has to be *DefaultConstructible* and *Assignable*.

The second template argument is the geometric traits class *TriangulationTraits_3* which provides the *Point_3*.

The third template argument is a vertex base class from which *Triangulation_vertex_base_with_info_3* derives. It has the default value *Triangulation_vertex_base_3<TriangulationTraits_3>*.

Is Model for the Concepts

TriangulationVertexBase_3

Inherits From

TriangulationVertexBase_3

Types

```
typedef Info          Info;
```

Access Functions

<i>const Info&</i>	<i>v.info() const</i>	Returns a const reference to the object of type <i>Info</i> stored in the vertex.
<i>Info&</i>	<i>v.info()</i>	Returns a reference to the object of type <i>Info</i> stored in the vertex.

See Also

CGAL::Triangulation_cell_base_with_info_3
CGAL::Triangulation_vertex_base_3

CGAL::Triangulation_hierarchy_vertex_base_3<TriangulationVertexBase_3>

Definition

This class is designed to be used as the vertex base class for *Triangulation_hierarchy_3*.

It inherits from its parameter *TriangulationVertexBase_3*, and adds the requirements in order to match the concept *TriangulationHierarchyVertexBase_3*, it does so by storing two *Vertex_handles*. This design allows to use either a vertex base class provided by CGAL, or a user customized vertex base with additional functionalities.

```
#include <CGAL/Triangulation_hierarchy_vertex_base_3.h>
```

Parameters

It is parameterized by a model of the concept *TriangulationVertexBase_3*.

Is Model for the Concepts

TriangulationHierarchyVertexBase_3

Inherits From

TriangulationVertexBase_3

See Also

CGAL::Triangulation_hierarchy_3

CGAL::Triangulation_vertex_base_3

CGAL::Triangulation_vertex_base_with_info_3

CGAL::Regular_triangulation_cell_base_3<Traits,Cb>

Definition

The class *Regular_triangulation_cell_base_3<Traits,Cb>* is a model of the concept *RegularTriangulationCellBase_3*. It is the default face base class of regular triangulations.

```
#include <CGAL/Regular_triangulation_cell_base_3.h>
```

Parameters

The template parameters *Traits* has to be a model of *RegularTriangulationTraits_3*.

The template parameter *Cb* has to be a model of *TriangulationCellBase_3*. By default, this parameter is instantiated by *CGAL::Triangulation_cell_base_3<Traits>*.

Is Model for the Concepts

RegularTriangulationCellBase_3

Inherits From

Cb

See Also

RegularTriangulationCellBase_3

RegularTriangulationTraits_3

CGAL::Regular_triangulation_3<Traits,Tds>

CGAL::Triangulation_3::Locate_type

Definition

The enum *Locate_type* is defined by *Triangulation_3* to specify which case occurs when locating a point in the triangulation.

```
enum Locate_type { VERTEX=0,  
                  EDGE,  
                  FACET,  
                  CELL,  
                  OUTSIDE_CONVEX_HULL,  
                  OUTSIDE_AFFINE_HULL }
```

See Also

CGAL::Triangulation_3

WeightedPoint

Definition

The concept `WeightedPoint` is needed by *Regular_triangulation_euclidean_traits_3*. It must fulfill the following requirements:

Types

<i>WeightedPoint::Point</i>	The point type
<i>WeightedPoint::Weight</i>	The weight type
<i>typedef Point::RT</i> <i>RT</i> ;	The ring type

Creation

WeightedPoint wp(Point p=Point(), Weight w = Weight(0));

Access Functions

<i>Point</i>	<i>point()</i>
<i>Weight</i>	<i>weight()</i>

Has Models

Weighted_point.

See Also

CGAL::Regular_triangulation_euclidean_traits_3
CGAL::Regular_triangulation_filtered_traits_3
CGAL::Regular_triangulation_3.

Chapter 23

3D Triangulation Data Structure

Sylvain Pion and Monique Teillaud

Contents

23.1 Representation	1561
23.2 Software Design	1564
23.2.1 Flexibility of the Design	1565
23.2.2 Cyclic Dependency	1566
23.2.3 Backward Compatibility	1567
23.3 Examples	1567
23.3.1 Incremental construction	1567
23.3.2 Cross-linking between a 2D and a 3D data structures	1569
23.4 Design and Implementation History	1570

A geometric triangulation has two aspects: the combinatorial structure, which gives the incidence and adjacency relations between faces, and the geometric information related to the position of vertices.

CGAL provides 3D geometric triangulations in which these two aspects are clearly separated. As described in Chapter 22, a geometric triangulation of a set of points in \mathbb{R}^d , $d \leq 3$ is a partition of the whole space \mathbb{R}^d into cells having $d + 1$ vertices. Some of them are infinite, they are obtained by linking an additional vertex at infinity to each facet of the convex hull of the points (see Section 22.1). The underlying combinatorial graph of such a triangulation without boundary of \mathbb{R}^d can be seen as a triangulation of the topological sphere S^d in \mathbb{R}^{d+1} .

This chapter deals with 3D-triangulation data structures, meant to maintain the combinatorial information for 3D-geometric triangulations. The reader interested in geometric triangulations of \mathbb{R}^3 is advised to read Chapter 22.

23.1 Representation

In CGAL, a 3D triangulation data structure is a container of cells (3-faces) and vertices (0-faces). Each cell gives access to its four incident vertices and to its four adjacent cells. Each vertex gives direct access to one of its incident cells, which is sufficient to retrieve all the incident cells when needed.

The four vertices of a cell are indexed with 0, 1, 2 and 3. The neighbors of a cell are also indexed with 0, 1, 2, 3 in such a way that the neighbor indexed by i is opposite to the vertex with the same index (see Figure 23.1).

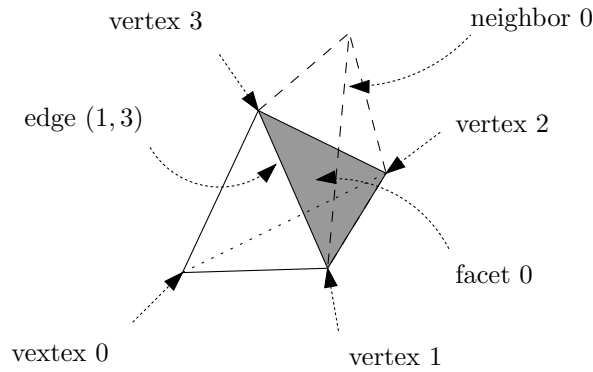


Figure 23.1: Representation.

Edges (1-faces) and facets (2-faces) are not explicitly represented: a facet is given by a cell and an index (the facet i of a cell c is the facet of c that is opposite to the vertex of index i) and an edge is given by a cell and two indices (the edge (i, j) of a cell c is the edge whose endpoints are the vertices of indices i and j of c).

Degenerate Dimensions As CGAL explicitly deals with all degenerate cases, a 3D-triangulation data structure in CGAL can handle the cases when the dimension of the triangulation is lower than 3.

Thus, a 3D-triangulation data structure can store a triangulation of a topological sphere S^d of \mathbb{R}^{d+1} , for any $d \in \{-1, 0, 1, 2, 3\}$.

Let us give, for each dimension, the example corresponding to the triangulation data structure having a minimal number of vertices, i.e. a simplex. These examples are illustrated by presenting their usual geometric embedding.

- *dimension 3*. The triangulation data structure consists of the boundary of a 4-dimensional simplex, which has 5 vertices. A geometric embedding consists in choosing one of these vertices to be infinite, thus four of the five 3-cells become infinite: the geometric triangulation has one finite tetrahedron remaining, each of its facets being incident to an infinite cell. See Figure 23.2.
- *dimension 2*. We have 4 vertices forming one 3-dimensional simplex, i.e. the boundary of a tetrahedron. The geometric embedding in the plane results from choosing one of these vertices to be infinite, then the geometric triangulation has one finite triangle whose edges are incident to the infinite triangles. See Figure 23.3.
- *dimension 1*. A 2-dimensional simplex (a triangle) has 3 vertices. The geometric embedding is an edge whose vertices are linked to an infinite point. See Figure 23.4.

The last three cases are defined uniquely:

- *dimension 0*. A 0-dimensional triangulation is combinatorially equivalent to the boundary of a 1-dimensional simplex (an edge), which consists of 2 vertices. One of them becomes infinite in the geometric embedding, and there is only one finite vertex remaining. The two vertices are adjacent.
- *dimension -1*. This dimension is a convention to represent a 0-dimensional simplex, that is a sole vertex, which will be geometrically embedded as an “empty” triangulation, having only one infinite vertex.
- *dimension -2*. This is also a convention. The triangulation data structure has no vertex. There is no associated geometric triangulation.

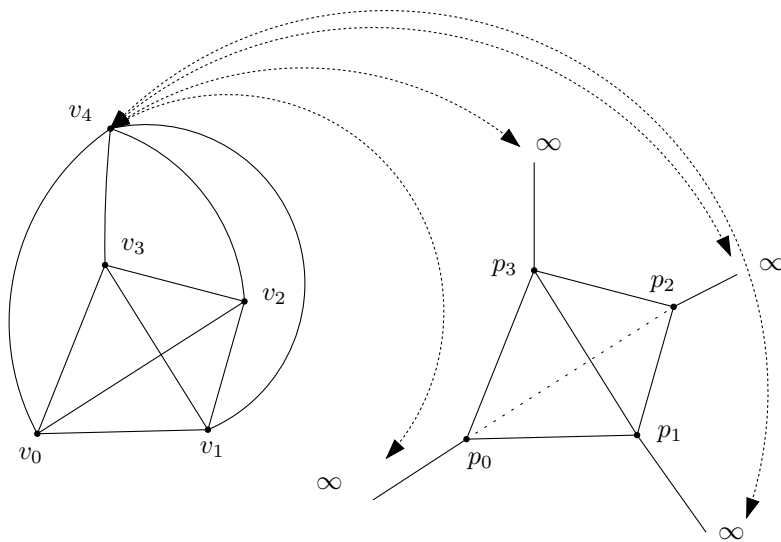


Figure 23.2: 4D simplex and a 3D geometric embedding.

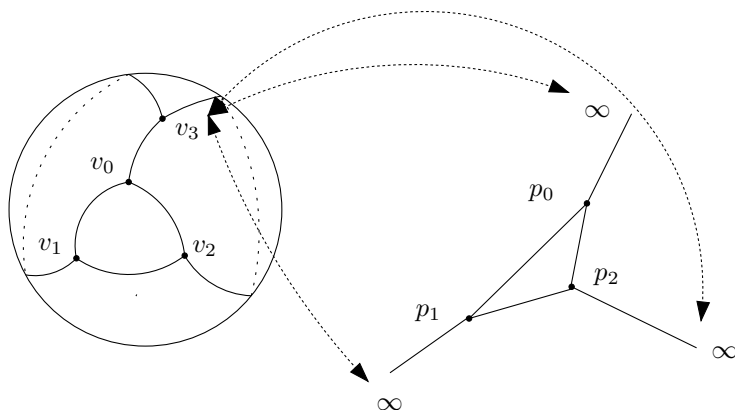


Figure 23.3: 3D simplex and a 2D geometric embedding.

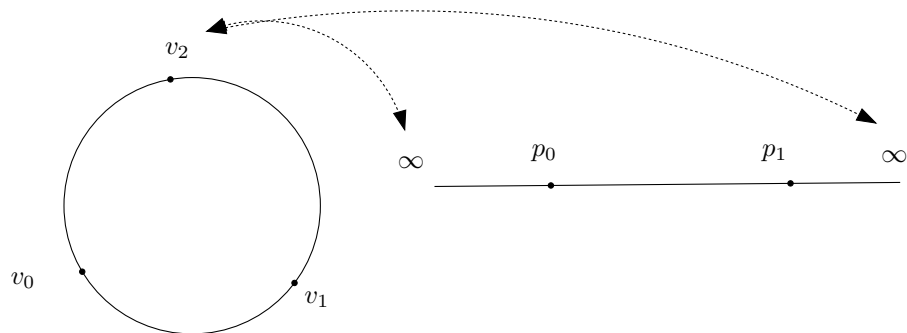


Figure 23.4: 2D simplex and a 1D geometric embedding.

Note that the notion of infinite vertex has no meaning for the triangulation data structure. The infinite vertex of the geometric embedding is a vertex that cannot be distinguished from the other vertices in the combinatorial triangulation.

The same cell class is used in all cases: triangular faces in 2D can be considered as degenerate cells, having only three vertices (resp. neighbors) numbered $(0, 1, 2)$; edges in 1D have only two vertices (resp. neighbors) numbered 0 and 1.

The implicit representation of facets (resp. edges) still holds for degenerate (< 3) dimensions : in dimension 2, each cell has only one facet of index 3, and 3 edges $(0, 1)$, $(1, 2)$ and $(2, 0)$; in dimension 1, each cell has one edge $(0, 1)$.

Validity A 3D combinatorial triangulation is said to be *locally valid* iff the following is true:

(a) When a cell c has a neighbor pointer to another cell c' , then reciprocally this cell c' has a neighbor pointer to c , and c and c' have three vertices in common. These cells are called adjacent.

(b) The cells have a coherent orientation: if two cells c_1 and c_2 are adjacent and share a facet with vertices u, v, w , then the vertices of c_1 are numbered $(v_0^1 = u, v_1^1 = v, v_2^1 = w, v_3^1)$, and the vertices of c_2 are numbered $(v_0^2 = v, v_1^2 = u, v_2^2 = w, v_3^2)$, up to positive permutations of $(0, 1, 2, 3)$. In other words, if we embed the triangulation in \mathbb{R}^3 , then the fourth vertices v_3^1 and v_3^2 of c_1 and c_2 see the common facet in opposite orientations. See Figure 23.5.

The set $\mathbf{\Sigma}_4$ of permutations of $(0, 1, 2, 3)$ has cardinality 24, and the set of positive permutations A_4 has cardinality 12. Thus, for a given orientation, there are up to 12 different orderings of the four vertices of a cell. Note that cyclic permutations are negative and so do not preserve the orientation of a cell.

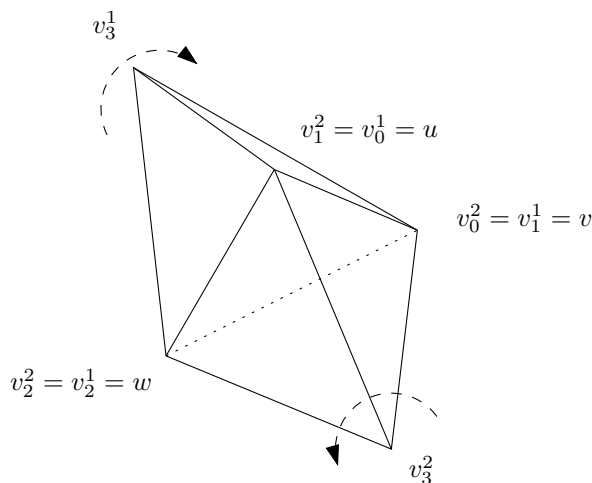


Figure 23.5: Coherent orientations of two cells (3-dimensional case).

The *is_valid()* method provided by *Triangulation_data_structure_3* checks the local validity of a given triangulation data structure.

23.2 Software Design

The 3D-triangulation data structure class of CGAL, *Triangulation_data_structure_3*, is designed to be used as a combinatorial layer upon which a geometric layer can be built [Ket98]. This geometric layer is typically one of

the 3D-triangulation classes of CGAL: *Triangulation_3*, *Delaunay_triangulation_3* and *Regular_triangulation_3*. This relation is described in more details in Chapter 22, where the Section 22.5 explains other important parts of the design related to the geometry.

We focus here on the design of the triangulation data structure (TDS) itself, which the Figure 23.6 illustrates.

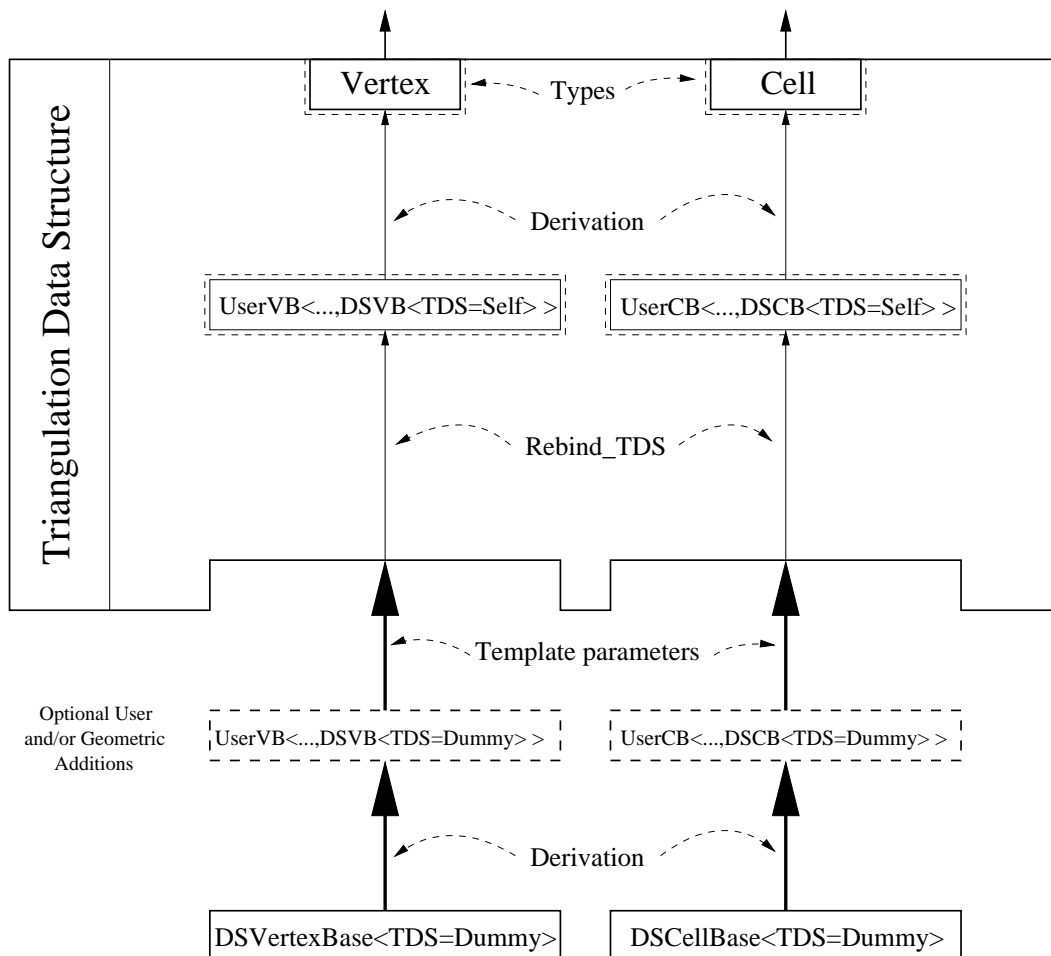


Figure 23.6: Triangulation Data Structure software design.

23.2.1 Flexibility of the Design

In order for the user to be able to add his own data in the vertices and cells, the design of the TDS is split into two layers:

- In the bottom layer, the (vertex and cell) base classes store elementary incidence and adjacency (and possibly geometric or other) information. These classes are parameterized by the TDS which provides the handle types. (They can also be parameterized by a geometric traits class or anything else.) A vertex stores a *Cell_handle*, and a cell stores four *Vertex_handles* and four *Cell_handles*.
- The middle layer is the TDS, which is purely combinatorial. It provides operations such as insertion of a new vertex in a given cell, on a 1 or 2-face. It also allows one, if the dimension of the triangulation is smaller than 3, to insert a vertex so that the dimension of the triangulation is increased by one. The TDS

is responsible for the combinatorial integrity of the eventual geometric triangulation built on top of it (the upper layer, see Chapter 22).

The user has several ways to add his own data in the vertex and cell base classes used by the TDS. He can either:

- use the classes *Triangulation_vertex_base_with_info* and *Triangulation_cell_base_with_info*, which allow to add one data member of a user provided type, and give access to it.
- derive his own classes from the default base classes *Triangulation_ds_vertex_base*, and *Triangulation_ds_cell_base* (or the geometric versions typically used by the geometric layer, *Triangulation_vertex_base*, and *Triangulation_cell_base*).
- write his own base classes following the requirements given by the concepts *TriangulationCellBase_3* and *TriangulationVertexBase_3* (described in page 1546 and page 1547).

23.2.2 Cyclic Dependency

Since adjacency relations are stored in the vertices and cells, it means that the vertex and cell base classes have to be able to store handles (an entity akin to pointers) to their neighbors in the TDS. This in turns means that the vertex and cell base classes have to know the types of these handles, which are provided by the TDS. So in a sense, the base classes are parameterized by the TDS, and the TDS is parameterized by the vertex and cell base classes ! This is a cycle which cannot be resolved easily.

The solution that we have chosen is similar to the mechanism used by the standard class *std::allocator*: the vertex and cell base classes are initially given a fake or dummy TDS template parameter, whose unique purpose is to provide the types that can be used by the vertex and cell base classes (such as handles). Then, inside the TDS itself, these base classes are *rebound* to the real TDS type, that is we obtain the same vertex and cell base classes, but parameterized with the real TDS instead of the dummy one. Rebinding is performed by a nested template class of the vertex and cell base classes (see code below), which provides a type which is the rebound vertex or cell base class¹.

Here is how it works, schematically:

```
template < class Vb, class Cb >
class TDS
{
    typedef TDS<Vb, Cb>    Self;

    // Rebind the vertex and cell base to the actual TDS (Self).
    typedef typename Vb::template Rebind_TDS<Self>::Other  VertexBase;
    typedef typename Cb::template Rebind_TDS<Self>::Other  CellBase;

    // ... further internal machinery leads to the final public types:
public:
    typedef ...  Vertex;
    typedef ...  Cell;
    typedef ...  Vertex_handle;
    typedef ...  Cell_handle;
};
```

¹It is logically equivalent to a mechanism that does not exist yet in the C++ language: *template typedef* or *template aliasing*

```

template < class TDS = ... > // The default is some internal type faking a TDS
class Triangulation_ds_vertex_base_3
{
public:
    template < class TDS2 >
    struct Rebind_TDS {
        typedef Triangulation_ds_vertex_base_3<TDS2>    Other;
    };
    ...
};

```

When derivation is used for the vertex or cell base classes, which is the case at the geometric level with *Triangulation_vertex_base_3*, then it gets slightly more involved because its base class has to be rebound as well:

```

template < class GT, class Vb = Triangulation_ds_vertex_base_3<> >
class Triangulation_vertex_base_3 : public Vb
{
public:
    template < class TDS2 >
    struct Rebind_TDS {
        typedef typename Vb::template Rebind_TDS<TDS2>::Other    Vb2;
        typedef Triangulation_vertex_base_3<GT, Vb2>             Other;
    };
    ...
};

```

23.2.3 Backward Compatibility

The rebinding scheme has been introduced in CGAL version 3.0. It is incompatible with the previous versions, for the cases where the user provides his own vertex or cell base class. In these cases, the user needs to add the rebind nested template class appropriately. More informations are given in [22.5.4](#).

23.3 Examples

23.3.1 Incremental construction

The following example shows how to construct a 3D triangulation data structure by inserting vertices.

```

// file: examples/Triangulation_3/example_tds.C

#include <CGAL/Triangulation_data_structure_3.h>
#include <iostream>
#include <fstream>
#include <cassert>
#include <vector>

typedef CGAL::Triangulation_data_structure_3<>    Tds;

```

```

typedef Tds::size_type          size_type;
typedef Tds::Cell_handle        Cell_handle;
typedef Tds::Vertex_handle      Vertex_handle;

int main()
{
    Tds T;

    assert( T.number_of_vertices() == 0 );
    assert( T.dimension() == -2 );
    assert( T.is_valid() );

    std::vector<Vertex_handle> PV(7);

    PV[0] = T.insert_increase_dimension();
    assert( T.number_of_vertices() == 1 );
    assert( T.dimension() == -1 );
    assert( T.is_valid() );

    // each of the following insertions of vertices increases the dimension
    for ( int i=1; i<5; i++ ) {
        PV[i] = T.insert_increase_dimension(PV[0]);
        assert( T.number_of_vertices() == (size_type) i+1 );
        assert( T.dimension() == i-1 );
        assert( T.is_valid() );
    }
    assert( T.number_of_cells() == 5 );

    // we now have a simplex in dimension 4

    // cell incident to PV[0]
    Cell_handle c = PV[0]->cell();
    int ind;
    bool check = c->has_vertex( PV[0], ind );
    assert( check );
    // PV[0] is the vertex of index ind in c

    // insertion of a new vertex in the facet opposite to PV[0]
    PV[5] = T.insert_in_facet(c, ind);

    assert( T.number_of_vertices() == 6 );
    assert( T.dimension() == 3 );
    assert( T.is_valid() );

    // insertion of a new vertex in c
    PV[6] = T.insert_in_cell(c);

    assert( T.number_of_vertices() == 7 );
    assert( T.dimension() == 3 );
    assert( T.is_valid() );

    std::ofstream oFileT("output_tds",std::ios::out);
    // writing file output_tds;

```



```

oFileT << T;

return 0;
}

```

23.3.2 Cross-linking between a 2D and a 3D data structures

This example program illustrates how to setup a 2D and a 3D triangulation data structures whose vertices respectively store vertex handles of the other one.

```

// file: examples/Triangulation_3/example_linking_2d_and_3d.C

#include <CGAL/Triangulation_data_structure_2.h>
#include <CGAL/Triangulation_data_structure_3.h>
#include <cassert>

// declare the 2D vertex base type, parametrized by some 3D TDS.
template < typename T3, typename Vb = CGAL::Triangulation_ds_vertex_base_2<> >
class My_vertex_2;

// declare the 3D vertex base type, parametrized by some 2D TDS.
template < typename T2, typename Vb = CGAL::Triangulation_ds_vertex_base_3<> >
class My_vertex_3;

// Then, we have to break the dependency cycle.

// we need to refer to a dummy 3D TDS.
typedef CGAL::Triangulation_ds_vertex_base_3<>::Triangulation_data_structure
    Dummy_tds_3;
// the 2D TDS, initially plugging a dummy 3D TDS in the vertex type
// (to break the dependency cycle).
typedef CGAL::Triangulation_data_structure_2<My_vertex_2<Dummy_tds_3> > TDS_2;
// the 3D TDS, here we can plug the 2D TDS directly.
typedef CGAL::Triangulation_data_structure_3<My_vertex_3<TDS_2> > TDS_3;

template < typename T3, typename Vb >
class My_vertex_2
    : public Vb
{
public:
    typedef typename Vb::Face_handle Face_handle;

    template <typename TDS2>
    struct Rebind_TDS {
        typedef typename Vb::template Rebind_TDS<TDS2>::Other Vb2;
        // we also have to break the cycle here by hardcoding TDS_3 instead of T3.
        typedef My_vertex_2<TDS_3, Vb2> Other;
    };

    My_vertex_2() {}

```

```

My_vertex_2(Face_handle f) : Vb(f) {}

// we store a vertex handle of the 3D TDS.
typename T3::Vertex_handle v3;
};

template < typename T2, typename Vb >
class My_vertex_3
: public Vb
{
public:
    typedef typename Vb::Cell_handle    Cell_handle;

    template <typename TDS2>
    struct Rebind_TDS {
        typedef typename Vb::template Rebind_TDS<TDS2>::Other    Vb2;
        typedef My_vertex_3<T2, Vb2>                               Other;
    };

    My_vertex_3() {}

    My_vertex_3(Cell_handle c) : Vb(c) {}

    // we store a vertex handle of the 2D TDS.
    typename T2::Vertex_handle v2;
};

int main() {
    TDS_2 t2;
    TDS_3 t3;

    TDS_2::Vertex_handle v2 = t2.insert_dim_up();
    TDS_3::Vertex_handle v3 = t3.insert_increase_dimension();

    v2->v3 = v3;
    v3->v2 = v2;

    assert(t2.is_valid());
    assert(t3.is_valid());
    return 0;
}

```

23.4 Design and Implementation History

Monique Teillaud introduced the triangulation of the topological sphere S^d in \mathbb{R}^{d+1} to manage the underlying graph of geometric triangulations and handle degenerate dimensions [Tei99].

Sylvain Pion improved the software in several ways, in particular regarding the memory management.

3D Triangulation Data Structure Reference Manual

Sylvain Pion and Monique Teillaud

The triangulation data structure is able to represent a triangulation of a topological sphere S^d of \mathbb{R}^{d+1} , for $d \in \{-1, 0, 1, 2, 3\}$. (See [23.1.](#))

The vertex class of a 3D-triangulation data structure must define a number of types and operations. The requirements that are of geometric nature are required only when the triangulation data structure is used as a layer for the geometric triangulation classes. (See [Section 23.2.](#))

The cell class of a triangulation data structure stores four handles to its four vertices and four handles to its four neighbors. The vertices are indexed 0, 1, 2, and 3 in a consistent order. The neighbor indexed i lies opposite to vertex i .

In degenerate dimensions, cells are used to store faces of maximal dimension: in dimension 2, each cell represents only one facet of index 3, and 3 edges (0,1), (1,2) and (2,0); in dimension 1, each cell represents one edge (0,1). (See [Section 23.1.](#))

23.5 Classified Reference Pages

Concepts

TriangulationDataStructure_3	page 1573
TriangulationDataStructure_3::Cell	page 1585
TriangulationDataStructure_3::Vertex	page 1587
TriangulationDSCellBase_3	page 1589
TriangulationDSVertexBase_3	page 1592

Classes

CGAL::Triangulation_data_structure_3<*TriangulationDSVertexBase_3*,*TriangulationDSCellBase_3*>
page [1594](#)

This class is a model for the concept of the 3D-triangulation data structure *TriangulationDataStructure_3*. It is templated by base classes for vertices and cells.

CGAL provides base vertex classes and base cell classes:

<i>CGAL::Triangulation_ds_cell_base_3</i> <>	page 1595
<i>CGAL::Triangulation_ds_vertex_base_3</i> <>	page 1596
<i>CGAL::Triangulation_cell_base_3</i> < <i>TriangulationTraits_3</i> , <i>TriangulationDSCellBase_3</i> >	page 1552
<i>CGAL::Triangulation_vertex_base_3</i> < <i>TriangulationTraits_3</i> , <i>TriangulationDSVertexBase_3</i> >	page 1554
<i>CGAL::Triangulation_hierarchy_vertex_base_3</i> < <i>TriangulationVertexBase_3</i> >	page 1556

Helper Classes

<i>CGAL::Triangulation_utils_3</i>	page 1597
------------------------------------------	---------------------------

It defines operations on the indices of vertices and neighbors within a cell of a triangulation.

23.6 Alphabetical List of Reference Pages

<i>Cell</i>	page 1585
<i>TriangulationDataStructure_3</i>	page 1573
<i>TriangulationDSCellBase_3</i>	page 1589
<i>TriangulationDSVertexBase_3</i>	page 1592
<i>Triangulation_data_structure_3</i> < <i>TriangulationDSVertexBase_3</i> , <i>TriangulationDSCellBase_3</i> >	page 1594
<i>Triangulation_ds_cell_base_3</i> <>	page 1595
<i>Triangulation_ds_vertex_base_3</i> <>	page 1596
<i>Triangulation_utils_3</i>	page 1597
<i>Vertex</i>	page 1587

TriangulationDataStructure_3

Definition

3D-triangulation data structures are meant to maintain the combinatorial information for 3D-geometric triangulations.

In CGAL, a triangulation data structure is a container of cells (3-faces) and vertices (0-faces). Each cell gives access to its four incident vertices and to its four adjacent cells. Each vertex gives direct access to one of its incident cells, which is sufficient to retrieve all the incident cells when needed.

The four vertices of a cell are indexed with 0, 1, 2 and 3. The neighbors of a cell are also indexed with 0, 1, 2, 3 in such a way that the neighbor indexed by i is opposite to the vertex with the same index (see Figure 23.1).

Edges (1-faces) and facets (2-faces) are not explicitly represented: a facet is given by a cell and an index (the facet i of a cell c is the facet of c that is opposite to the vertex of index i) and an edge is given by a cell and two indices (the edge (i,j) of a cell c is the edge whose endpoints are the vertices of indices i and j of c).

As CGAL explicitly deals with all degenerate cases, a 3D-triangulation data structure in CGAL can handle the cases when the dimension of the triangulation is lower than 3 (see Section 23.1).

Thus, a 3D-triangulation data structure can store a triangulation of a topological sphere S^d of \mathbb{R}^{d+1} , for any $d \in \{-1, 0, 1, 2, 3\}$.

The second template parameter of the basic triangulation class (see Chapter 22, page 1504) *Triangulation_3* is a triangulation data structure class. (See Chapter 23.)

To ensure all the **flexibility** of the class *Triangulation_3*, a model of a triangulation data structure must be templated by the base vertex and the base cell classes (see 23.1): *TriangulationDataStructure_3* < *TriangulationVertexBase_3*, *TriangulationCellBase_3* >. The optional functionalities related to geometry are compulsory for this use as a template parameter of *Triangulation_3*.

A class that satisfies the requirements for a triangulation data structure class must provide the following types and operations.

Types

<i>TriangulationDataStructure_3:: Vertex</i>	Vertex type
<i>TriangulationDataStructure_3:: Cell</i>	Cell type
<i>TriangulationDataStructure_3:: size_type</i>	Size type (unsigned integral type)
<i>TriangulationDataStructure_3:: difference_type</i>	Difference type (signed integral type)

Vertices and cells are usually manipulated via *handles*, which support the two dereference operators *operator** and *operator->*.

TriangulationDataStructure_3::Vertex_handle
TriangulationDataStructure_3::Cell_handle

Requirements for *Vertex* and *Cell* are described in *TriangulationDataStructure_3::Vertex* and *TriangulationDataStructure_3::Cell* (page 1587 and page 1585).

typedef Triple<Cell_handle, int, int> Edge; (c,i,j) is the edge of cell c whose vertices indices are i and j .
 (See Section 23.1.)
typedef std::pair<Cell_handle, int> Facet; (c,i) is the facet of c opposite to the vertex of index i . (See
 Section 23.1.)

The following iterators allow one to visit all the vertices, edges, facets and cells of the triangulation data structure. They are all bidirectional, non-mutable iterators.

TriangulationDataStructure_3::Cell_iterator
TriangulationDataStructure_3::Facet_iterator
TriangulationDataStructure_3::Edge_iterator
TriangulationDataStructure_3::Vertex_iterator

The following circulators allow us to visit all the cells and facets incident to a given edge. They are bidirectional and non-mutable.

TriangulationDataStructure_3::Facet_circulator
TriangulationDataStructure_3::Cell_circulator

Iterators and circulators are convertible to the corresponding handles, thus the user can pass them directly as arguments to the functions.

Creation

<i>TriangulationDataStructure_3 tds;</i>	Default constructor.
<i>TriangulationDataStructure_3 tds(tds1);</i>	Copy constructor. All vertices and cells are duplicated.
<i>TriangulationDataStructure_3& tds = tds1</i>	Assignment operator. All vertices and cells are duplicated, and the former data structure of <i>tds</i> is deleted.

Vertex_handle tds.copy_tds(tds1, Vertex_handle v = Vertex_handle())
tds1 is copied into *tds*. If $v \neq \text{Vertex_handle}()$, the vertex of *tds* corresponding to v is returned, otherwise *Vertex_handle()* is returned.
Precondition: The optional argument v is a vertex of *tds1*.

<i>void tds.swap(& tds1)</i>	Swaps <i>tds</i> and <i>tds1</i> . There is no copy of cells and vertices, thus this method runs in constant time. This method should be preferred to <i>tds=tds1</i> or <i>tds(tds1)</i> when <i>tds1</i> is deleted after that.
<i>void tds.clear()</i>	Deletes all cells and vertices. <i>tds</i> is reset as a triangulation data structure constructed by the default constructor.

Operations

Access Functions

<i>int</i>	<i>tds.dimension()</i>	The dimension of the triangulated topological sphere.
<i>size_type</i>	<i>tds.number_of_vertices()</i>	The number of vertices. Note that the triangulation data structure has one more vertex than an associated geometric triangulation, if there is one, since the infinite vertex is a standard vertex and is thus also counted.
<i>size_type</i>	<i>tds.number_of_cells()</i>	The number of cells. Returns 0 if <i>tds.dimension()</i> < 3.

Non constant-time access functions

<i>size_type</i>	<i>tds.number_of_facets()</i>	The number of facets. Returns 0 if <i>tds.dimension()</i> < 2.
<i>size_type</i>	<i>tds.number_of_edges()</i>	The number of edges. Returns 0 if <i>tds.dimension()</i> < 1.

┌────────── *advanced* ─────────┐

Setting

<i>void</i>	<i>tds.set_dimension(int n)</i>	Sets the dimension to <i>n</i> .
-------------	----------------------------------	----------------------------------

┌────────── *advanced* ─────────┐

Queries

<i>bool</i>	<i>tds.is_vertex(Vertex_handle v)</i>	Tests whether <i>v</i> is a vertex of <i>tds</i> .
<i>bool</i>	<i>tds.is_edge(Cell_handle c, int i, int j)</i>	Tests whether <i>(c,i,j)</i> is an edge of <i>tds</i> . Answers <i>false</i> when <i>dimension()</i> < 1 . <i>Precondition: i, j ∈ {0, 1, 2, 3}</i>
<i>bool</i>	<i>tds.is_edge(Vertex_handle u, Vertex_handle v, Cell_handle & c, int & i, int & j)</i>	Tests whether <i>(u,v)</i> is an edge of <i>tds</i> . If the edge is found, it computes a cell <i>c</i> having this edge and the indices <i>i</i> and <i>j</i> of the vertices <i>u</i> and <i>v</i> , in this order.
<i>bool</i>	<i>tds.is_edge(Vertex_handle u, Vertex_handle v)</i>	Tests whether <i>(u,v)</i> is an edge of <i>tds</i> .
<i>bool</i>	<i>tds.is_facet(Cell_handle c, int i)</i>	Tests whether <i>(c,i)</i> is a facet of <i>tds</i> . Answers <i>false</i> when <i>dimension()</i> < 2 . <i>Precondition: i ∈ {0, 1, 2, 3}</i>

bool *tds.is_facet*(*Vertex_handle* *u*,
 Vertex_handle *v*,
 Vertex_handle *w*,
 Cell_handle & *c*,
 int & *i*,
 int & *j*,
 int & *k*)

Tests whether (u,v,w) is a facet of *tds*. If the facet is found, it computes a cell *c* having this facet and the indices *i*, *j* and *k* of the vertices *u*, *v* and *w*, in this order.

bool *tds.is_cell*(*Cell_handle* *c*)

Tests whether *c* is a cell of *tds*. Answers *false* when *dimension()* < 3 .

bool *tds.is_cell*(*Vertex_handle* *u*,
 Vertex_handle *v*,
 Vertex_handle *w*,
 Vertex_handle *t*,
 Cell_handle & *c*,
 int & *i*,
 int & *j*,
 int & *k*,
 int & *l*)

Tests whether (u,v,w,t) is a cell of *tds*. If the cell *c* is found, it computes the indices *i*, *j*, *k* and *l* of the vertices *u*, *v*, *w* and *t* in *c*, in this order.

There is a method *has_vertex* in the cell class. The analogous methods for facets are defined here.

bool *tds.has_vertex*(*Facet* *f*, *Vertex_handle* *v*, *int* & *j*)

If *v* is a vertex of *f*, then *j* is the index of *v* in the cell *f.first*, and the method returns *true*.

Precondition: *tds.dimension()*=3

bool *tds.has_vertex*(*Cell_handle* *c*, *int* *i*, *Vertex_handle* *v*, *int* & *j*)

Same for facet (c,i) . Computes the index *j* of *v* in *c*.

bool *tds.has_vertex*(*Facet* *f*, *Vertex_handle* *v*)

bool *tds.has_vertex*(*Cell_handle* *c*, *int* *i*, *Vertex_handle* *v*)

Same as the first two methods, but these two methods do not return the index of the vertex.

The following three methods test whether two facets have the same vertices.

bool *tds.are_equal*(*Facet* *f*, *Facet* *g*)

bool *tds.are_equal*(*Cell_handle* *c*, *int* *i*, *Cell_handle* *n*, *int* *j*)

bool *tds.are_equal*(*Facet* *f*, *Cell_handle* *n*, *int* *j*)

For these three methods:

Precondition: *tds.dimension()*=3.

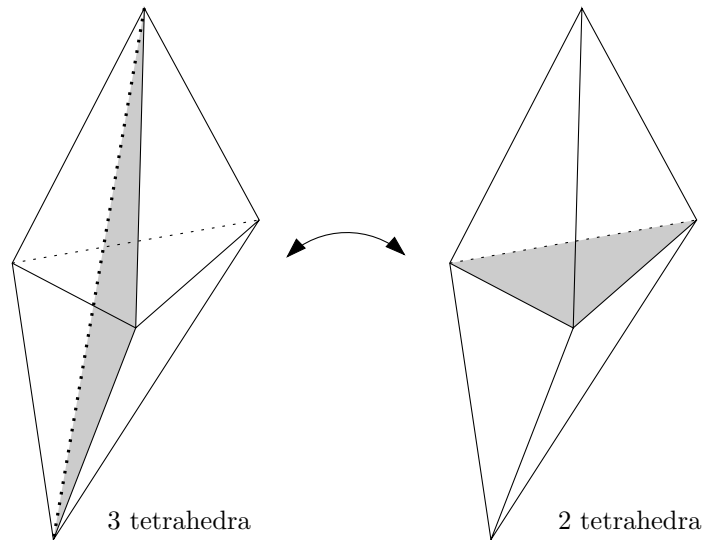


Figure 23.7: Flips.

Flips

Two kinds of flips exist for a three-dimensional triangulation. They are reciprocal. To be flipped, an edge must be incident to three tetrahedra. During the flip, these three tetrahedra disappear and two tetrahedra appear. Figure 23.7(left) shows the edge that is flipped as bold dashed, and one of its three incident facets is shaded. On the right, the facet shared by the two new tetrahedra is shaded.

The following methods guarantee the validity of the resulting 3D combinatorial triangulation. Moreover the flip operations do not invalidate the vertex handles, and only invalidate the cell handles of the affected cells.

Flips for a 2d triangulation are not implemented yet

```
bool    tds.flip( Edge e)
bool    tds.flip( Cell_handle c, int i, int j)
```

Before flipping, these methods check that edge $e=(c,i,j)$ is flip-pable (which is quite expensive). They return *false* or *true* according to this test.

```
void    tds.flip_flippable( Edge e)
void    tds.flip_flippable( Cell_handle c, int i, int j)
```

Should be preferred to the previous methods when the edge is known to be flippable.

Precondition: The edge is flippable.

```
bool    tds.flip( Facet f)
bool    tds.flip( Cell_handle c, int i)
```

Before flipping, these methods check that facet $f=(c,i)$ is flip-pable (which is quite expensive). They return *false* or *true* according to this test.

```
void    tds.flip_flippable( Facet f)
```

void *tds.flip_flippable(Cell_handle c, int i)*

Should be preferred to the previous methods when the facet is known to be flippable.

Precondition: The facet is flippable.

Insertions

The following modifier member functions guarantee the combinatorial validity of the resulting triangulation.

Vertex_handle *tds.insert_in_cell(Cell_handle c)*

Creates a new vertex, inserts it in cell c and returns its handle. The cell c is split into four new cells, each of these cells being formed by the new vertex and a facet of c .

Precondition: $tds.dimension() = 3$ and c is a cell of tds .

Vertex_handle *tds.insert_in_facet(Facet f)*

Creates a new vertex, inserts it in facet f and returns its handle. In dimension 3, the two incident cells are split into 3 new cells; in dimension 2, the facet is split into 3 facets.

Precondition: $tds.dimension() \geq 2$ and f is a facet of tds .

Vertex_handle *tds.insert_in_facet(Cell_handle c, int i)*

Creates a new vertex, inserts it in facet i of c and returns its handle.

Precondition: $tds.dimension() \geq 2$, $i \in \{0, 1, 2, 3\}$ in dimension 3, $i = 3$ in dimension 2 and (c, i) is a facet of tds .

Vertex_handle *tds.insert_in_edge(Edge e)*

Creates a new vertex, inserts it in edge e and returns its handle. In dimension 3, all the incident cells are split into 2 new cells; in dimension 2, the 2 incident facets are split into 2 new facets; in dimension 1, the edge is split into 2 new edges.

Precondition: $tds.dimension() \geq 1$ and e is an edge of tds .

Vertex_handle *tds.insert_in_edge(Cell_handle c, int i, int j)*

Creates a new vertex, inserts it in edge (i, j) of c and returns its handle.

Precondition: $tds.dimension() \geq 1$. $i \neq j$, $i, j \in \{0, 1, 2, 3\}$ in dimension 3, $i, j \in \{0, 1, 2\}$ in dimension 2, $i, j \in \{0, 1\}$ in dimension 1 and (c, i, j) is an edge of tds .

Vertex_handle *tds.insert_increase_dimension(Vertex_handle star = Vertex_handle())*

Transforms a triangulation of the sphere S^d of \mathbb{R}^{d+1} into the triangulation of the sphere S^{d+1} of \mathbb{R}^{d+2} by adding a new vertex v : v is linked to all the vertices to triangulate one of the two half-spheres of dimension $(d+1)$. Vertex *star* is used to triangulate the second halfsphere (when there is an associated geometric triangulation, *star* is in fact the vertex associated with its infinite vertex). See Figure 23.8.

The numbering of the cells is such that, if f was a face of maximal dimension in the initial triangulation, then (f, v) (in this order) is the corresponding face in the new triangulation. This method can be used to insert the first two vertices in an empty triangulation.

A handle to v is returned.

Precondition: *tds.dimension()* = $d < 3$. When *tds.number_of_vertices()* > 0, *star* \neq *Vertex_handle()* and *star* is a vertex of *tds*.

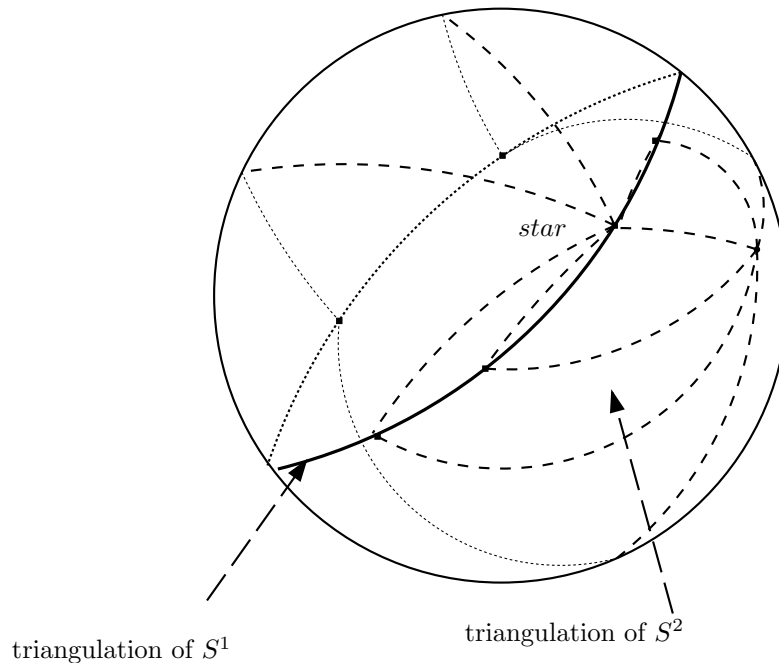


Figure 23.8: *insert_increase_dimension* (1-dimensional case).

Vertex_handle *tds.insert_in_hole(CellIt cell_begin, CellIt cell_end, Cell_handle begin, int i)*

Creates a new vertex by starring a hole. It takes an iterator range $[cell_begin; cell_end]$ of *Cell_handles* which specifies a set of connected cells (resp. facets in dimension 2) describing a hole. $(begin, i)$ is a facet (resp. an edge) on the boundary of the hole, that is, *begin* belongs to the set of cells (resp. facets) previously described, and *begin*->*neighbor(i)* does not. Then this function deletes all the cells (resp. facets) describing the hole, creates a new vertex *v*, and for each facet (resp. edge) on the boundary of the hole, creates a new cell (resp. facet) with *v* as vertex. *v* is returned.

Precondition: *tds.dimension()* ≥ 2 , the set of cells (resp. facets) is connected, and its boundary is connected.

Removal

void *tds.remove_decrease_dimension(Vertex_handle v, Vertex_handle w = v)*

This operation is the reciprocal of *insert_increase_dimension()*. It transforms a triangulation of the sphere S^d of \mathbb{R}^{d+1} into the triangulation of the sphere S^{d-1} of \mathbb{R}^d by removing the vertex *v*. Delete the cells incident to *w*, keep the others.

Precondition: *tds.dimension()* $= d \geq -1$. *tds.degree(v)* $= \text{degree}(w) = \text{tds.number_of_vertices()} - 1$.

Cell_handle *tds.remove_from_maximal_dimension_simplex(Vertex_handle v)*

Removes *v*. The incident simplices of maximal dimension incident to *v* are replaced by a single simplex of the same dimension. This operation is exactly the reciprocal to *tds.insert_in_cell(v)* in dimension 3, *tds.insert_in_facet(v)* in dimension 2, and *tds.insert_in_edge(v)* in dimension 1.

Precondition: *tds.degree(v)* $= \text{tds.dimension()} + 1$.

— advanced —

Other modifiers

The following modifiers can affect the validity of the triangulation data structure.

void *tds.reorient()* Changes the orientation of all cells of the triangulation data structure.

Precondition: *tds.dimension()* ≥ 1 .

Vertex_handle *tds.create_vertex(Vertex v = Vertex())*

Adds a copy of the vertex *v* to the triangulation data structure.

Creates a vertex which is a copy of the one pointed to by v and adds it to the triangulation data structure.

Adds a copy of the cell c to the triangulation data structure.

Creates a cell which is a copy of the one pointed to by c and adds it to the triangulation data structure.

Creates a cell and adds it into the triangulation data structure. Initializes the vertices of the cell, its neighbor handles being initialized with the default constructed handle.

Creates a cell, initializes its vertices and neighbors, and adds it into the triangulation data structure.

Removes the vertex from the triangulation data structure.
Precondition: The vertex is a vertex of *tds*.

Removes the cell from the triangulation data structure.
Precondition: The cell is a cell of *tds*.

Calls *delete_vertex* over an iterator range of value type *Vertex_handle*.

1581

void *tds.delete_cells(CellIt first, CellIt last)*

Calls *delete_cell* over an iterator range of value type *Cell_handle*.

└────────── *advanced* ─────────┘

Traversing the triangulation

Cell_iterator *tds.cells_begin()* Returns *cells_end()* when *tds.dimension()* < 3.
Cell_iterator *tds.cells_end()*
Cell_iterator *tds.raw_cells_begin()*

Low-level access to the cells, does not return *cells_end()* when *tds.dimension()* < 3.

Cell_iterator *tds.raw_cells_end()*
Facet_iterator *tds.facets_begin()* Returns *facets_end()* when *tds.dimension()* < 2.
Facet_iterator *tds.facets_end()*
Edge_iterator *tds.edges_begin()* Returns *edges_end()* when *tds.dimension()* < 1.
Edge_iterator *tds.edges_end()*
Vertex_iterator *tds.vertices_begin()*
Vertex_iterator *tds.vertices_end()*

Cell_circulator *tds.incident_cells(Edge e)*

Starts at an arbitrary cell incident to *e*.
Precondition: tds.dimension() = 3

Cell_circulator *tds.incident_cells(Cell_handle c, int i, int j)*

As above for edge (*i,j*) of *c*.

Cell_circulator *tds.incident_cells(Edge e, Cell_handle start)*

Starts at cell *start*.
Precondition: tds.dimension() = 3 and *start* is incident to *e*.

Cell_circulator *tds.incident_cells(Cell_handle c, int i, int j, Cell_handle start)*

As above for edge (*i,j*) of *c*.

The following circulators on facets are defined only in dimension 3, though facets are defined also in dimension 2: there are only two facets sharing an edge in dimension 2.

Facet_circulator *tds.incident_facets(Edge e)*

Starts at an arbitrary facet incident to *e*.
Precondition: tds.dimension() = 3

Facet_circulator *tds.incident_facets(Cell_handle c, int i, int j)*

As above for edge (*i,j*) of *c*.

Facet_circulator *tds.incident_facets(Edge e, Facet start)*

Starts at facet *start*.
Precondition: start is incident to *e*.

Facet_circulator *tds.incident_facets(Edge e, Cell_handle start, int f)*

Starts at facet of index f in *start*.

Facet_circulator *tds.incident_facets(Cell_handle c, int i, int j, Facet start)*

As above for edge (i,j) of *c*.

Facet_circulator *tds.incident_facets(Cell_handle c, int i, int j, Cell_handle start, int f)*

As above for edge (i,j) of *c* and facet $(start,f)$.

Traversal of the incident cells and facets, and the adjacent vertices of a given vertex

template <class OutputIterator>

OutputIterator *tds.incident_cells(Vertex_handle v, OutputIterator cells)*

Copies the *Cell_handles* of all cells (resp. facets in dimension 2) incident to v to the output iterator *cells*. If *tds.dimension()* < 2 , then do nothing. Returns the resulting output iterator.

Precondition: $v \neq \text{Vertex_handle}()$, *tds.is_vertex*(v).

template <class OutputIterator>

OutputIterator *tds.incident_facets(Vertex_handle v, OutputIterator facets)*

Copies the *Facets* incident to v to the output iterator *facets*. Returns the resulting output iterator.

Precondition: *tds.dimension()* $= 3$, $v \neq \text{Vertex_handle}()$, *tds.is_vertex*(v).

template <class OutputIterator>

OutputIterator *tds.incident_vertices(Vertex_handle v, OutputIterator vertices)*

Copies the *Vertex_handles* of all vertices incident to v to the output iterator *vertices*. If *tds.dimension()* < 2 , then do nothing. Returns the resulting output iterator.

Precondition: $v \neq \text{Vertex_handle}()$, *tds.is_vertex*(v).

size_type *tds.degree(Vertex_handle v)*

Returns the degree of a vertex, that is, the number of incident vertices.

Precondition: $v \neq \text{Vertex_handle}()$, *tds.is_vertex*(v).

int *tds.mirror_index(Cell_handle c, int i)*

Returns the index of c in its i^{th} neighbor.

Precondition: $i \in \{0, 1, 2, 3\}$.

Vertex_handle *tds.mirror_vertex(Cell_handle c, int i)*

Returns the vertex of the i^{th} neighbor of c that is opposite to c .

Precondition: $i \in \{0, 1, 2, 3\}$.

Facet *tds.mirror_facet(Facet f)*

Returns the same facet viewed from the other adjacent cell.

└────────── advanced ─────────┘

Checking

bool *tds.is_valid(bool verbose = false)*

Checks the combinatorial validity of the triangulation by checking the local validity of all its cells and vertices (see functions below). (See Section 23.1.) Moreover, the Euler relation is tested. When *verbose* is set to *true*, messages are printed to give a precise indication on the kind of invalidity encountered.

bool *tds.is_valid(Vertex_handle v, bool verbose = false)*

Checks the local validity of the adjacency relations of the triangulation. It also calls the *is_valid* member function of the vertex. When *verbose* is set to *true*, messages are printed to give a precise indication on the kind of invalidity encountered.

bool *tds.is_valid(Cell_handle c, bool verbose = false)*

Checks the local validity of the adjacency relations of the triangulation. It also calls the *is_valid* member function of the cell. When *verbose* is set to *true*, messages are printed to give a precise indication on the kind of invalidity encountered.

└────────── advanced ─────────┘

I/O

istream& *istream& is >> & tds* Reads a combinatorial triangulation from *is* and assigns it to *tds*

ostream& *ostream& os << tds* Writes *tds* into the stream *os*

The information stored in the *iostream* is: the dimension, the number of vertices, the number of cells, the indices of the vertices of each cell, then the indices of the neighbors of each cell, where the index corresponds to the preceding list of cells. When $\text{dimension} < 3$, the same information is stored for faces of maximal dimension instead of cells.

Has Models

CGAL::Triangulation_data_structure_3

See Also

TriangulationDataStructure_3::Vertex
TriangulationDataStructure_3::Cell

TriangulationDataStructure_3::Cell

Definition

The concept `Cell` stores four *Vertex_handles* to its four vertices and four *Cell_handles* to its four neighbors. The vertices are indexed 0, 1, 2, and 3 in consistent order. The neighbor indexed i lies opposite to vertex i .

In degenerate dimensions, cells are used to store faces of maximal dimension: in dimension 2, each cell represents only one facet of index 3, and 3 edges (0,1), (1,2) and (2,0); in dimension 1, each cell represents one edge (0,1). (See also Section [23.1](#).)

Types

The class `Cell` defines the following types.

```
typedef TriangulationDataStructure_3      Triangulation_data_structure;
typedef TriangulationDataStructure_3::Vertex_handle Vertex_handle;
typedef TriangulationDataStructure_3::Cell_handle   Cell_handle;
```

Creation

In order to obtain new cells or destruct unused cells, the user must call the *create_cell()* and *delete_cell()* methods of the triangulation data structure.

Operations

Access Functions

<i>Vertex_handle</i>	<i>c.vertex(int i)</i>	Returns the vertex i of c . <i>Precondition:</i> $i \in \{0, 1, 2, 3\}$.
<i>int</i>	<i>c.index(Vertex_handle v)</i>	Returns the index of vertex v in c . <i>Precondition:</i> v is a vertex of c .
<i>bool</i>	<i>c.has_vertex(Vertex_handle v)</i>	Returns <i>true</i> if v is a vertex of c .
<i>bool</i>	<i>c.has_vertex(Vertex_handle v, int & i)</i>	Returns <i>true</i> if v is a vertex of c , and computes its index i in c .
<i>Cell_handle</i>	<i>c.neighbor(int i)</i>	Returns the neighbor i of c . <i>Precondition:</i> $i \in \{0, 1, 2, 3\}$.
<i>int</i>	<i>c.index(Cell_handle n)</i>	Returns the index corresponding to neighboring cell n . <i>Precondition:</i> n is a neighbor of c .
<i>bool</i>	<i>c.has_neighbor(Cell_handle n)</i>	Returns <i>true</i> if n is a neighbor of c .
<i>bool</i>	<i>c.has_neighbor(Cell_handle n, int & i)</i>	Returns <i>true</i> if n is a neighbor of c , and computes its index i in c .

Setting

<i>void</i>	<i>c.set_vertex(int i, Vertex_handle v)</i>	Sets vertex <i>i</i> to <i>v</i> . <i>Precondition: $i \in \{0, 1, 2, 3\}$.</i>
<i>void</i>	<i>c.set_vertices(Vertex_handle v0, Vertex_handle v1, Vertex_handle v2, Vertex_handle v3)</i>	Sets the vertex pointers.
<i>void</i>	<i>c.set_neighbor(int i, Cell_handle n)</i>	Sets neighbor <i>i</i> to <i>n</i> . <i>Precondition: $i \in \{0, 1, 2, 3\}$.</i>
<i>void</i>	<i>c.set_neighbors(Cell_handle n0, Cell_handle n1, Cell_handle n2, Cell_handle n3)</i>	Sets the neighbors pointers.

Checking

<i>bool</i>	<i>c.is_valid(bool verbose = false, int level = 0)</i>	User defined local validity checking function.
-------------	---------------------------------------------------------	------------------------------------------------

See Also

TriangulationDataStructure_3::Vertex.

TriangulationDataStructure_3::Vertex

Definition

The concept Vertex represents the vertex class of a 3D-triangulation data structure. It must define the types and operations listed in this section. Some of these requirements are of geometric nature, they are *optional* when using the triangulation data structure class alone. They become compulsory when the triangulation data structure is used as a layer for the geometric triangulation class. (See Section 23.2.)

Types

Vertex:: Point *Optional for the triangulation data structure alone.*

The class Vertex defines types that are the same as some of the types defined by the triangulation data structure class *TriangulationDataStructure_3*.

```
typedef TriangulationDataStructure_3      Triangulation_data_structure;
typedef TriangulationDataStructure_3::Vertex_handle  Vertex_handle;
typedef TriangulationDataStructure_3::Cell_handle    Cell_handle;
```

Creation

In order to obtain new vertices or destruct unused vertices, the user must call the *create_vertex()* and *delete_vertex()* methods of the triangulation data structure.

Operations

Access Functions

<i>Cell_handle</i>	<i>v.cell()</i>	Returns a cell of the triangulation having <i>v</i> as vertex.
<i>Point</i>	<i>v.point()</i>	Returns the point stored in the vertex. <i>Optional for the triangulation data structure alone.</i>

Setting

<i>void</i>	<i>v.set_cell(Cell_handle c)</i>	Sets the incident cell to <i>c</i> .
<i>void</i>	<i>v.set_point(Point p)</i>	Sets the point to <i>p</i> . <i>Optional for the triangulation data structure alone.</i>

Checking

bool *v.is_valid(bool verbose = false)*

Checks the validity of the vertex. Must check that its incident cell has this vertex. The validity of the base vertex is also checked.

When *verbose* is set to *true*, messages are printed to give a precise indication on the kind of invalidity encountered.

See Also

TriangulationDataStructure_3::Cell.

TriangulationDSCellBase_3

Definition

At the base level (see Sections 22.5 and 23.2), a cell stores handles to its four vertices and to its four neighbor cells. The vertices and neighbors are indexed 0, 1, 2 and 3. Neighbor i lies opposite to vertex i .

Since the Triangulation data structure is the class which defines the handle types, the cell base class has to be somehow parameterized by the Triangulation data structure. But since it is itself parameterized by the cell and vertex base classes, there is a cycle in the definition of these classes. In order to break the cycle, the base classes for vertex and cell which are given as arguments for the Triangulation data structure use *void* as Triangulation data structure parameter, and the Triangulation data structure then uses a *rebind*-like mechanism (similar to the one specified in *std::allocator*) in order to put itself as parameter to the vertex and cell classes. The *rebound* base classes so obtained are the classes which are used as base classes for the final vertex and cell classes. More information can be found in Section 23.2.

Types

The concept TriangulationDSCellBase_3 has to provide the following types.

TriangulationDSCellBase_3::template <typename TDS2> struct Rebind_TDS;

This nested template class has to define a type *Other* which is the *rebound* cell, that is, the one whose *Triangulation_data_structure* will be the actually used one. The *Other* type will be the real base class of *Triangulation_data_structure_3::Cell*.

<i>typedef TriangulationDataStructure_3</i>	<i>Triangulation_data_structure;</i>
<i>typedef TriangulationDataStructure_3::Vertex_handle</i>	<i>Vertex_handle;</i>
<i>typedef TriangulationDataStructure_3::Cell_handle</i>	<i>Cell_handle;</i>

Creation

<i>TriangulationDSCellBase_3 c;</i>	Default constructor
<i>TriangulationDSCellBase_3 c(Vertex_handle v0, Vertex_handle v1, Vertex_handle v2, Vertex_handle v3);</i>	

Initializes the vertices with $v0$, $v1$, $v2$, $v3$. Neighbors are initialized to the default constructed handle.

*TriangulationDSCellBase_3 c(Vertex_handle v0,
Vertex_handle v1,
Vertex_handle v2,
Vertex_handle v3,
Cell_handle n0,
Cell_handle n1,
Cell_handle n2,*

Cell_handle n3)

Initializes the vertices with $v0$, $v1$, $v2$, $v3$ and the neighbors with $n0$, $n1$, $n2$, $n3$.

Access Functions

<i>Vertex_handle</i>	<i>c.vertex(int i)</i>	Returns the vertex i of c . <i>Precondition:</i> $i \in \{0, 1, 2, 3\}$.
<i>int</i>	<i>c.index(Vertex_handle v)</i>	Returns the index of v . <i>Precondition:</i> v is a vertex of c
<i>bool</i>	<i>c.has_vertex(Vertex_handle v)</i>	True iff v is a vertex of c .
<i>bool</i>	<i>c.has_vertex(Vertex_handle v, int & i)</i>	Returns <i>true</i> if v is a vertex of c , and computes its index i in c .
<i>Cell_handle</i>	<i>c.neighbor(int i)</i>	Returns the neighbor i of c . <i>Precondition:</i> $i \in \{0, 1, 2, 3\}$.
<i>int</i>	<i>c.index(Cell_handle n)</i>	Returns the index of cell n in c . <i>Precondition:</i> n is a neighbor of c .
<i>bool</i>	<i>c.has_neighbor(Cell_handle n)</i>	Returns <i>true</i> if n is a neighbor of c .
<i>bool</i>	<i>c.has_neighbor(Cell_handle n, int & i)</i>	Returns <i>true</i> if n is a neighbor of c , and computes its index i in c .

Setting

<i>void</i>	<i>c.set_vertex(int i, Vertex_handle v)</i>	Sets vertex i to v . <i>Precondition:</i> $i \in \{0, 1, 2, 3\}$.
<i>void</i>	<i>c.set_vertices()</i>	Sets the vertices to the default constructed handle.
<i>void</i>	<i>c.set_vertices(Vertex_handle v0, Vertex_handle v1, Vertex_handle v2, Vertex_handle v3)</i>	Sets the vertices.
<i>void</i>	<i>c.set_neighbor(int i, Cell_handle n)</i>	Sets neighbor i to n . <i>Precondition:</i> $i \in \{0, 1, 2, 3\}$.
<i>void</i>	<i>c.set_neighbors()</i>	Sets the neighbors to the default constructed handle.
<i>void</i>	<i>c.set_neighbors(Cell_handle n0, Cell_handle n1, Cell_handle n2, Cell_handle n3)</i>	Sets the neighbors.

Checking

<i>bool</i>	<i>c.is_valid(bool verbose = false, int level = 0)</i>	Performs any desired geometric test on a cell. When <i>verbose</i> is set to <i>true</i> , messages are printed to give a precise indication of the kind of invalidity encountered. <i>level</i> increases the level of testing.
-------------	---------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Various

*void** *c.for_compact_container()*
void&* *c.for_compact_container()*

These member functions are required by *Triangulation_data_structure_3* because it uses *Compact_container* to store its cells. See the documentation of *Compact_container* for the exact requirements.

void *c.set_in_conflict_flag(unsigned char f)*
unsigned char *c.get_in_conflict_flag()*

These functions are used internally to mark cells with a flag. The user is not encouraged to use them directly as they may change in the future.

I/O

<i>istream&</i>	<i>istream& is >> & c</i>	Inputs the possible non combinatorial information given by the cell.
<i>ostream&</i>	<i>ostream& os << c</i>	Outputs the possible non combinatorial information given by the cell.

Has Models

CGAL::Triangulation_ds_cell_base_3
CGAL::Triangulation_cell_base_3
CGAL::Triangulation_cell_base_with_info_3

See Also

TriangulationDSVertexBase_3
TriangulationVertexBase_3
TriangulationHierarchyVertexBase_3
TriangulationCellBase_3

TriangulationDSVertexBase_3

Definition

At the bottom level of 3D-triangulations (see Sections [22.5](#) and [23.2](#)), a vertex provides access to one of its incident cells through a handle.

Note that when you use the triangulation data structure as parameter of a geometric triangulation, the vertex base class has additional geometric requirements : it has to match the *TriangulationVertexBase_3* concept.

Since the Triangulation data structure is the class which defines the handle types, the vertex base class has to be somehow parameterized by the Triangulation data structure. But since it is itself parameterized by the cell and vertex base classes, there is a cycle in the definition of these classes. In order to break the cycle, the base classes for vertex and cell which are given as arguments for the Triangulation data structure use *void* as Triangulation data structure parameter, and the Triangulation data structure then uses a *rebind*-like mechanism (similar to the one specified in *std::allocator*) in order to put itself as parameter to the vertex and cell classes. The *rebound* base classes so obtained are the classes which are used as base classes for the final vertex and cell classes. More information can be found in Section [23.2](#).

Types

The class *TriangulationDSVertexBase_3* has to define the following types.

TriangulationDSVertexBase_3::template <typename TDS2> struct Rebind_TDS;

This nested template class has to define a type *Other* which is the *rebound* vertex, that is, the one whose *Triangulation_data_structure* will be the actually used one. The *Other* type will be the real base class of *Triangulation_data_structure_3::Vertex*.

<i>typedef TriangulationDataStructure_3</i>	<i>Triangulation_data_structure;</i>
<i>typedef TriangulationDataStructure_3::Vertex_handle</i>	<i>Vertex_handle;</i>
<i>typedef TriangulationDataStructure_3::Cell_handle</i>	<i>Cell_handle;</i>

Creation

<i>TriangulationDSVertexBase_3 v;</i>	Default constructor.
<i>TriangulationDSVertexBase_3 v(Cell_handle c);</i>	Constructs a vertex pointing to cell <i>c</i> .

Operations

Access Functions

<i>Cell_handle v.cell()</i>	Returns the pointer to an incident cell
-----------------------------	-----------------------------------------

Setting

void *v.set_cell(Cell_handle c)* Sets the incident cell.

Checking

bool *v.is_valid(bool verbose=false, int level=0)*

Performs any desired test on a vertex. Checks that the pointer to an incident cell is not the default constructed handle.

Various

*void** *v.for_compact_container()*
void&* *v.for_compact_container()*

These member functions are required by *Triangulation_data_structure_3* because it uses *Compact_container* to store its cells. See the documentation of *Compact_container* for the exact requirements.

I/O

istream& *istream& is >> & v* Inputs the non-combinatorial information given by the vertex.

ostream& *ostream& os << v* Outputs the non-combinatorial information given by the vertex.

Has Models

CGAL::Triangulation_ds_vertex_base_3
CGAL::Triangulation_vertex_base_3
CGAL::Triangulation_vertex_base_with_info_3
CGAL::Triangulation_hierarchy_vertex_base_3

See Also

TriangulationVertexBase_3
TriangulationHierarchyVertexBase_3
TriangulationDSCellBase_3
TriangulationCellBase_3

CGAL::Triangulation_data_structure_3<TriangulationDSVertexBase_3, TriangulationDSCellBase_3>

Definition

The class *Triangulation_data_structure_3* stores a 3D-triangulation data structure and provides the optional geometric functionalities to be used as a parameter for a 3D-geometric triangulation (see Chapter 22).

```
#include <CGAL/Triangulation_data_structure_3.h>
```

Parameters

It is parameterized by base classes for vertices and cells which have to match the requirements for the concepts *TriangulationDSCellBase_3* and *TriangulationDSVertexBase_3* respectively (see page 1589 and page 1592).

They have the default values *Triangulation_ds_vertex_base_3*<> and *Triangulation_ds_cell_base_3*<> respectively.

Is Model for the Concepts

TriangulationDataStructure_3

Inherits From

CGAL::Triangulation_utils_3

The class *Triangulation_utils_3* defines basic computations on indices of vertices and neighbors of cells.

See Also

CGAL::Triangulation_ds_vertex_base_3
CGAL::Triangulation_ds_cell_base_3
CGAL::Triangulation_vertex_base_with_info_3
CGAL::Triangulation_cell_base_with_info_3

CGAL::Triangulation_ds_cell_base_3<>

Definition

The class *Triangulation_ds_cell_base_3* is a model for the concept *TriangulationDSCellBase_3* to be used by *Triangulation_data_structure_3*.

```
#include <CGAL/Triangulation_ds_cell_base_3.h>
```

Is Model for the Concepts

TriangulationDSCellBase_3

See Also

CGAL::Triangulation_cell_base_3

CGAL::Triangulation_ds_vertex_base_3

CGAL::Triangulation_cell_base_with_info_3

CGAL::Triangulation_ds_vertex_base_3<>

Definition

The class *Triangulation_ds_vertex_base_3* can be used as the base vertex for a 3D-triangulation data structure, it is a model of the concept *TriangulationDSVertexBase_3*.

Note that if the triangulation data structure is used as a parameter of a geometric triangulation (Section [23.2](#) and Chapter [22](#)), then the vertex base class has to fulfill additional geometric requirements, i.e. it has to be a model of the concept *TriangulationVertexBase_3*.

This base class can be used directly or can serve as a base to derive other base classes with some additional attributes (a color for example) tuned for a specific application.

```
#include <CGAL/Triangulation_ds_vertex_base_3.h>
```

Is Model for the Concepts

TriangulationDSVertexBase_3

See Also

CGAL::Triangulation_vertex_base_3

CGAL::Triangulation_ds_cell_base_3

CGAL::Triangulation_vertex_base_with_info_3

CGAL::Triangulation_utils_3

Definition

The class *Triangulation_utils_3* defines operations on the indices of vertices and neighbors within a cell.

```
#include <CGAL/Triangulation_utils_3.h>
```

Operations

unsigned int *next_around_edge(unsigned int i, unsigned int j)*

In dimension 3, index of the neighbor *n* that is next to the current cell, when turning positively around an oriented edge whose endpoints are indexed *i* and *j*. According to the usual numbering of vertices and neighbors in a given cell, it is also the index of the vertex opposite to this neighbor *n*. (see Figure 23.9).

Precondition: $(i < 4) \ \&\& \ (j < 4) \ \&\& \ (i \neq j)$.

unsigned int *ccw(unsigned int i)*

Has a meaning only in dimension 2.

Computes the index of the vertex that is next to the vertex numbered *i* in counterclockwise direction. (see Figure 23.9).

Precondition: $i < 3$.

unsigned int *cw(unsigned int i)*

Same for clockwise.

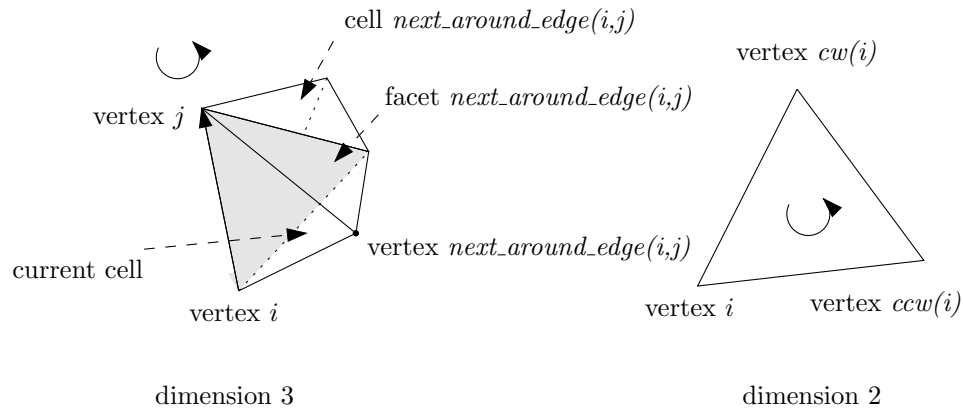


Figure 23.9: Operations on indices.

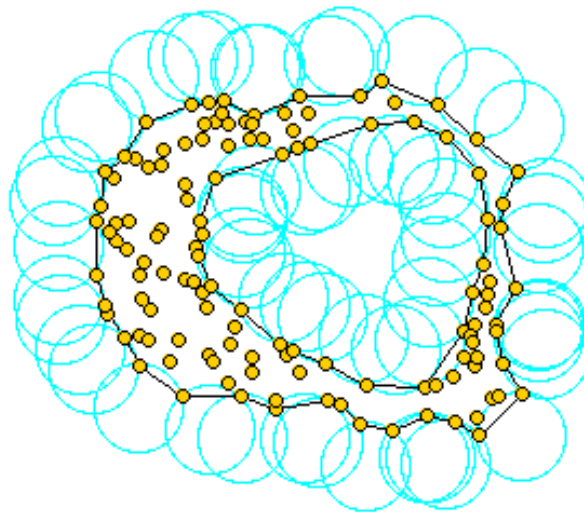
Chapter 24

2D Alpha Shapes

Tran Kai Frank Da

Contents

24.1 Definitions	1600
24.2 Functionality	1600
24.3 Concepts and Models	1601
24.4 Examples	1601
24.4.1 Example for Basic Alpha-Shapes	1601
24.4.2 Example for Basic Alpha-Shapes with Many Points	1602
24.4.3 Example for Weighted Alpha-Shapes	1602



Assume we are given a set S of points in 2D or 3D and we'd like to have something like “the shape formed by these points.” This is quite a vague notion and there are probably many possible interpretations, the α -shape being one of them. Alpha shapes can be used for shape reconstruction from a dense unorganized set of data points. Indeed, an α -shape is demarcated by a frontier, which is a linear approximation of the original shape [BB97].

As mentioned in Edelsbrunner's and Mücke's paper [EM94], one can intuitively think of an α -shape as the following. Imagine a huge mass of ice-cream making up the space \mathbb{R}^3 and containing the points as "hard" chocolate pieces. Using one of these sphere-formed ice-cream spoons we carve out all parts of the ice-cream block we can reach without bumping into chocolate pieces, thereby even carving out holes in the inside (eg. parts not reachable by simply moving the spoon from the outside). We will eventually end up with a (not necessarily convex) object bounded by caps, arcs and points. If we now straighten all "round" faces to triangles and line segments, we have an intuitive description of what is called the α -shape of S . Here's an example for this process in 2D (where our ice-cream spoon is simply a circle):

And what is α in the game? α is the squared radius of the carving spoon. A very small value will allow us to eat up all of the ice-cream except the chocolate points themselves. Thus we already see that the α -shape degenerates to the point-set S for $\alpha \rightarrow 0$. On the other hand, a huge value of α will prevent us even from moving the spoon between two points since it's way too large. So we will never spoon up ice-cream lying in the inside of the convex hull of S , and hence the α -shape for $\alpha \rightarrow \infty$ is the convex hull of S .¹

24.1 Definitions

We distinguish two versions of alpha shapes. *Basic alpha shapes* are based on the Delaunay triangulation. *Weighted alpha shapes* are based on its generalization, the regular triangulation, replacing the euclidean distance by the power to weighted points.

There is a close connection between alpha shapes and the underlying triangulations. More precisely, the α -complex of S is a subcomplex of this triangulation of S , containing the α -exposed k -simplices, $0 \leq k \leq d$. A simplex is α -exposed, if there is an open disk (resp. ball) of radius $\sqrt{\alpha}$ through the vertices of the simplex that does not contain any other point of S , for the metric used in the computation of the underlying triangulation. The corresponding α -shape is defined as the underlying interior space of the α -complex (see [EM94]).

In general, an α -complex is a non-connected and non-pure polytope, it means, that one k -simplex, $0 \leq k \leq d - 1$ is not necessary adjacent to a $(k + 1)$ -simplex.

The α -shapes of S form a discrete family, even though they are defined for all real numbers α with $0 \leq \alpha \leq \infty$. Thus, we can represent the entire family of α -shapes of S by the underlying triangulation of S . In this representation each k -simplex of the underlying triangulation is associated with an interval that specifies for which values of α the k -simplex belongs to the α -shape. Relying on this fact, the family of α -shapes can be computed efficiently and relatively easily. Furthermore, we can select an appropriate α -shape from a finite number of different α -shapes and corresponding α -values.

24.2 Functionality

The class `CGAL::Alpha_shape_2<Dt>` represents the family of α -shapes of points in a plane for *all* positive α . It maintains the underlying triangulation Dt which represents connectivity and order among squared radius of its faces. Each k -dimensional face of the Dt is associated with an interval that specifies for which values of α the face belongs to the α -shape. There are links between the intervals and the k -dimensional faces of the triangulation.

The class `CGAL::Alpha_shape_2<Dt>` provides functions to set and get the current α -value, as well as an iterator that enumerates the α -values where the α -shape changes.

¹ice cream, ice cream!!! The wording of this introductory paragraphs is borrowed from Kaspar Fischer's "Introduction to Alpha Shapes" which can be found at <http://n.ethz.ch/student/fischerk/alphashapes/as/index.html>. The picture has been taken from Walter Luh's homepage at <http://www.stanford.edu/~wluh/cs448b/alphashapes.html>.

It provides iterators to enumerate the vertices and edges that are in the α -shape, and functions that allow to classify vertices, edges and faces with respect to the α -shape. They can be in the interior of a face that belongs or does not belong to the α -shape. They can be singular/regular, that is be on the boundary of the α -shape, but not incident/incident to a triangle of the α -complex.

Finally, it provides a function to determine the α -value such that the α -shape satisfies the following two properties, or at least the second one if there is no such α that both are satisfied:

- (i) The number of components equals a number of your choice and
- (ii) all data points are either on the boundary or in the interior of the regularized version of the α -shape (no singular edges).

The current implementation is static, that is after its construction points cannot be inserted or removed.

24.3 Concepts and Models

We currently do not specify concepts for the underlying triangulation type. Models that work for a basic alpha-shape are the classes `CGAL::Delaunay_triangulation_2` and `CGAL::Triangulation_hierarchy_2` templated with a Delaunay triangulation. A model that works for a weighted alpha-shape is the class `CGAL::Regular_triangulation_2`.

The triangulation needs a geometric traits class as argument. The requirements of this class are described in the concept `CGAL::AlphaShapeTraits_2` for which the CGAL kernels and `CGAL::Weighted_alpha_shape_euclidean_traits_2` are models.

There are no requirements on the triangulation data structure. However it must be parameterized with vertex and face classes, which are model of the concepts `AlphaShapeVertex_2` and `AlphaShapeFace_2`, by default the classes `CGAL::Alpha_shape_vertex_base_2<Gt>` and `CGAL::Alpha_shape_face_base_2<Tf>`.

24.4 Examples

24.4.1 Example for Basic Alpha-Shapes

The basic alpha shape needs a Delaunay triangulation as underlying triangulation *Dt*. The Delaunay triangulation class is parameterized with a geometric and a triangulation data structure traits.

For the geometric traits class we can use a CGAL kernel.

For the triangulation data structure traits, we have to choose the vertex and face classes needed for alpha shapes, namely `CGAL::Alpha_shape_vertex_base_2<Gt, Dv>` and `CGAL::Alpha_shape_face_base_2<Gt, Df>`. As default vertex and face type they use `CGAL::Triangulation_vertex_base_2<Gt>` and `CGAL::Triangulation_face_base_2<Gt>` respectively.

The following code snippet shows how to obtain a basic alpha shape type.

```
typedef CGAL::Cartesian<double> K;

typedef CGAL::Alpha_shape_vertex_base_2<K> Av;
```

```

typedef CGAL::Triangulation_face_base_2<K> Tf;
typedef CGAL::Alpha_shape_face_base_2<K,Tf> Af;

typedef CGAL::Triangulation_default_data_structure_2<K,Av,Af> Tds;
typedef CGAL::Delaunay_triangulation_2<K,Tds> Dt;
typedef CGAL::Alpha_shape_2<Dt> Alpha_shape_2;

```

24.4.2 Example for Basic Alpha-Shapes with Many Points

When the input data set is huge, say more than 10.000 points, it pays off to use a triangulation hierarchy. It has the same API as the Delaunay triangulation and differs only in the types of the vertices and faces. Therefore, the only part that changes are the typedefs in the beginning.

```

typedef CGAL::Cartesian<double> K;

typedef CGAL::Alpha_shape_vertex_base_2<K> Avb;
typedef CGAL::Triangulation_hierarchy_vertex_base_2<Avb> Av;

typedef CGAL::Triangulation_face_base_2<K> Tf;
typedef CGAL::Alpha_shape_face_base_2<K,Tf> Af;

typedef CGAL::Triangulation_default_data_structure_2<K,Av,Af> Tds;
typedef CGAL::Delaunay_triangulation_2<K,Tds> Dt;
typedef CGAL::Triangulation_hierarchy_2<Dt> Ht;
typedef CGAL::Alpha_shape_2<Ht> Alpha_shape_2;

```

24.4.3 Example for Weighted Alpha-Shapes

A weighted alpha shape, needs a regular triangulation as underlying triangulation *Dt*, and it needs a particular face class, namely *CGAL::Regular_triangulation_face_base_2<Gt>*. Note that there is no special weighted alpha shape class.

```

typedef CGAL::Cartesian<double> K;
typedef CGAL::Weighted_alpha_shape_euclidean_traits_2<K> Gt;

typedef CGAL::Alpha_shape_vertex_base_2<Gt> Av;

typedef CGAL::Regular_triangulation_face_base_2<Gt> Rf;
typedef CGAL::Alpha_shape_face_base_2<Gt,Rf> Af;

typedef CGAL::Triangulation_default_data_structure_2<Gt,Av,Af> Tds;
typedef CGAL::Regular_triangulation_2<Gt,Tds> Rt;
typedef CGAL::Alpha_shape_2<Rt> Alpha_shape_2;

```

2D Alpha Shapes

Reference Manual

Tran Kai Frank Da

This chapter presents a framework for alpha shapes. The description is based on the articles [EM94, Ede92]. Alpha shapes are the generalization of the convex hull of a point set. Let S be a finite set of points in \mathbb{R}^d , $d = 2, 3$ and α a parameter with $0 \leq \alpha \leq \infty$. For $\alpha = \infty$, the α -shape is the convex hull of S . As α decreases, the α -shape shrinks and develops cavities, as soon as a sphere of radius $\sqrt{\alpha}$ can be put inside. Finally, for $\alpha = 0$, the α -shape is the set S itself.

We distinguish two versions of alpha shapes, one is based on the Delaunay triangulation and the other on its generalization, the regular triangulation, replacing the natural distance by the power to weighted points. The metric used determines an underlying triangulation of the alpha shape and thus, the version computed. The *basic alpha shape* (cf. 24.4.1) is associated with the Delaunay triangulation (cf. 20.5). The *weighted alpha shape* (cf. 24.4.3) is associated with the regular triangulation (cf. 20.6).

There is a close connection between alpha shapes and the underlying triangulations. More precisely, the α -complex of S is a subcomplex of this triangulation of S , containing the α -exposed k -simplices, $0 \leq k \leq d$. A simplex is α -exposed, if there is an open disk (resp. ball) of radius $\sqrt{\alpha}$ through the vertices of the simplex that does not contain any other point of S , for the metric used in the computation of the underlying triangulation. The corresponding α -shape is defined as the underlying interior space of the α -complex.

In general, an α -complex is a non-connected and non-pure polytope, it means, that one k -simplex, $0 \leq k \leq d - 1$ is not necessary adjacent to a $(k + 1)$ -simplex.

The α -shapes of S form a discrete family, even though they are defined for all real numbers α with $0 \leq \alpha \leq \infty$. Thus, we can represent the entire family of α -shapes of S by the underlying triangulation of S . In this representation each k -simplex of the underlying triangulation is associated with an interval that specifies for which values of α the k -simplex belongs to the α -shape. Relying on this result, the family of α -shapes can be computed efficiently and relatively easily. Furthermore, we can select an appropriate α -shape from a finite number of different α -shapes and corresponding α -values.

24.5 Classified Reference Pages

Concepts

AlphaShapeTraits_2	page 1614
AlphaShapeFace_2	page 1611
AlphaShapeVertex_2	page 1616

Classes

<i>CGAL::Alpha_shape_2<Dt></i>	page 1605
<i>CGAL::Weighted_alpha_shape_euclidean_traits_2<K></i>	page 1615
<i>CGAL::Alpha_shape_vertex_base_2<AlphaShapeTraits_2></i>	page 1617
<i>CGAL::Alpha_shape_face_base_2<AlphaShapeTraits_2, TriangulationFaceBase_2></i>	page 1613

24.6 Alphabetical List of Reference Pages

<i>AlphaShapeFace_2</i>	page 1611
<i>AlphaShapeTraits_2</i>	page 1614
<i>AlphaShapeVertex_2</i>	page 1616
<i>Alpha_shape_2<Dt></i>	page 1605
<i>Alpha_shape_face_base_2<AlphaShapeTraits_2, TriangulationFaceBase_2></i>	page 1613
<i>Alpha_shape_vertex_base_2<AlphaShapeTraits_2></i>	page 1617
<i>Weighted_alpha_shape_euclidean_traits_2<K></i>	page 1615

CGAL::Alpha_shape_2<Dt>

Definition

The class *Alpha_shape_2<Dt>* represents the family of α -shapes of points in a plane for *all* positive α . It maintains the underlying triangulation *Dt* which represents connectivity and order among its faces. Each *k*-dimensional face of the *Dt* is associated with an interval that specifies for which values of α the face belongs to the α -shape. There are links between the intervals and the *k*-dimensional faces of the triangulation.

Note that this class is at the same time used for *basic* and for *weighted* Alpha Shapes.

Inherits From

Dt

This class is the underlying triangulation class.

The modifying functions *insert* and *remove* will overwrite the inherited functions. At the moment, only the static version is implemented.

Types

Alpha_shape_2<Dt>:: Gt the alpha shape traits type.

it has to derive from a triangulation traits class. For example *Dt::Point* is a Point class.

typedef Gt::FT

FT; the number type for computation.

Alpha_shape_2<Dt>:: Alpha_iterator

A bidirectional and non-mutable iterator that allow to traverse the increasing sequence of different α -values.

Precondition: Its *value_type* is *FT*

Alpha_shape_2<Dt>:: Alpha_shape_vertices_iterator

A bidirectional and non-mutable iterator that allow to traverse the vertices which belongs to the α -shape for the current α .

Precondition: Its *value_type* is *Dt::Vertex_handle*

Alpha_shape_2<Dt>:: Alpha_shape_edges_iterator

A bidirectional and non-mutable iterator that allow to traverse the edges which belongs to the α -shape for the current α .

Precondition: Its *value_type* is *Dt::Edge*.

enum Classification_type { EXTERIOR, SINGULAR, REGULAR, INTERIOR};

Distinguishes the different cases for classifying a k -dimensional face of the underlying triangulation of the α -shape.

EXTERIOR if the face does not belong to the α -complex.

SINGULAR if the face belongs to the boundary of the α -shape, but is not incident to any 2-dimensional face of the α -complex

REGULAR if the face belongs to the boundary of the α -shape and is incident to a 2-dimensional face of the α -complex

INTERIOR if the face belongs to the α -complex, but does not belong to the boundary of the α -shape.

enum Mode { GENERAL, REGULARIZED};

In general, an alpha shape can be disconnected and contain many singular edges or vertices. Its regularized version is formed by the set of regular edges and their vertices.

Creation

Alpha_shape_2<Dt> A(FT alpha = 0, Mode m = GENERAL);

Introduces an empty α -shape A for a positive α -value $alpha$.

Precondition: $alpha \geq 0$.

template < class InputIterator >

Alpha_shape_2<Dt> A(InputIterator first, InputIterator last, FT alpha = 0, Mode m = GENERAL);

Initializes the family of alpha-shapes with the points in the range $[first, last)$ and introduces an α -shape A for a positive α -value $alpha$.

Precondition: The *value_type* of *first* and *last* is *Point*.
 $alpha \geq 0$.

Operations

template < class InputIterator >

int A.make_alpha_shape(InputIterator first, InputIterator last)

Initialize the family of alpha-shapes with the points in the range $[first, last)$. Returns the number of inserted points.

If the function is applied to a non-empty family of alpha-shape, it is cleared before initialization.

Precondition: The *value_type* of *first* and *last* is *Point*.

void A.clear() Clears the structure.

<i>FT</i>	<i>A.set_alpha(FT alpha)</i>	Sets the α -value to <i>alpha</i> . Returns the previous α -value. <i>Precondition: alpha \geq 0.</i>
<i>FT</i>	<i>A.get_alpha(void)</i>	Returns the current α -value.
<i>FT</i>	<i>A.get_nth_alpha(int n)</i>	Returns the <i>n</i> -th alpha-value, sorted in an increasing order. <i>Precondition: n < number of alphas.</i>
<i>int</i>	<i>A.number_of_alphas()</i>	Returns the number of different alpha-values.
<i>Mode</i>	<i>A.set_mode(Mode m = GENERAL)</i>	Sets <i>A</i> to its general or regularized version. Returns the previous mode.
<i>Mode</i>	<i>A.get_mode(void)</i>	Returns whether <i>A</i> is general or regularized.
<i>Alpha_shape_vertices_iterator</i>		
	<i>A.alpha_shape_vertices_begin()</i>	Starts at an arbitrary finite vertex which belongs to the α -shape for the current α .
<i>Alpha_shape_vertices_iterator</i>		
	<i>A.alpha_shape_vertices_end()</i>	Past-the-end iterator.
<i>Alpha_shape_edges_iterator</i>		
	<i>A.alpha_shape_edges_begin()</i>	Starts at an arbitrary finite edge which belongs to the α -shape for the current α . In regularised mode, edges are represented as a pair (f,i), where f is an interior face of the α -shape.
<i>Alpha_shape_edges_iterator</i>		
	<i>A.alpha_shape_edges_end()</i>	Past-the-end iterator.

Predicates

Classification_type

A.classify(Point p, FT alpha = get_alpha())

Locates a point p in the underlying triangulation and Classifies the associated k -face with respect to A .

Classification_type

A.classify(Face_handle f, FT alpha = get_alpha())

Classifies the face f of the underlying triangulation with respect to A .

Classification_type

A.classify(Edge e, FT alpha = get_alpha())

Classifies the edge e of the underlying triangulation with respect to A .

Classification_type

A.classify(Face_handle f, int i, FT alpha = get_alpha())

Classifies the edge of the face f opposite to the vertex with index i of the underlying triangulation with respect to A .

Classification_type

A.classify(Vertex_handle v, FT alpha = get_alpha())

Classifies the vertex v of the underlying triangulation with respect to A .

Traversal of the α -Values

Alpha_iterator

A.alpha_begin() Returns an iterator that allows to traverse the sorted sequence of α -values of the family of alpha shapes.

Alpha_iterator

A.alpha_end() Returns the corresponding past-the-end iterator.

Alpha_iterator

A.alpha_find(FT alpha)

Returns an iterator pointing to an element with α -value $alpha$, or the corresponding past-the-end iterator if such an element is not found.

Alpha_iterator

A.alpha_lower_bound(FT alpha)

Returns an iterator pointing to the first element with α -value not less than *alpha*.

Alpha_iterator

A.alpha_upper_bound(FT alpha)

Returns an iterator pointing to the first element with α -value greater than *alpha*.

Operations

int *A.number_of_solid_components(FT alpha = get_alpha())*

Returns the number of solid components of *A*, that is, the number of components of its regularized version.

Alpha_iterator

A.find_optimal_alpha(int nb_components)

Returns an iterator pointing to the first element with α -value such that *A* satisfies the following two properties:

nb_components equals the number of solid components and all data points are either on the boundary or in the interior of the regularized version of *A*.

If no such value is found, the iterator points to the first element with α -value such that *A* satisfies the second property.

I/O

The I/O operators are defined for *ostream*, and for the window stream provided by CGAL. The format for the *ostream* is an internal format.

#include <CGAL/IO/io.h>

ostream& *ostream& os << A* Inserts the alpha shape *A* for the current α -value into the stream *os*.
Precondition: The insert operator must be defined for *Point*.

#include <CGAL/IO/Window_stream.h>

#include <CGAL/IO/alpha_shapes_2_window_stream.h>

Window_stream&

Window_stream& *W* << *A*

Inserts the alpha shape *A* for the current α -value into the window stream *W*.

Precondition: The insert operator must be defined for *Point* and *Segment*.

Implementation

The set of intervals associated with the *k*-dimensional faces of the underlying triangulation are stored in *multimaps*.

The cross links between the intervals and the *k*-dimensional faces of the triangulation are realized using methods in the *k*-dimensional faces themselves.

A.alpha find uses linear search, while *A.alpha lower bound* and *A.alpha upper bound* use binary search. *A.number of solid components* performs a graph traversal and takes time linear in the number of faces of the underlying triangulation. *A.find optimal alpha* uses binary search and takes time $O(n \log n)$, where *n* is the number of points.

AlphaShapeFace_2

Definition

Refines

TriangulationFaceBase_2.

Types

AlphaShapeFace_2:: Interval_3

A container type to get (and put) the three special values $(\alpha_1, \alpha_2, \alpha_3)$ associated with an alpha shape edge.

AlphaShapeFace_2:: FT

A type to hold a coordinate type class. The type must provide a copy constructor, assignment, comparison operators, negation, multiplication, division and allow the declaration and initialization with a small integer constant (cf. requirements for number types). An obvious choice would be coordinate type of the point class

Creation

————— *advanced* —————

AlphaShapeFace_2 f;

default constructor.

AlphaShapeFace_2 f(Vertex_handle v0, Vertex_handle v1, Vertex_handle v2);

constructor setting the incident vertices.

*AlphaShapeFace_2 f(Vertex_handle v0,
Vertex_handle v1,
Vertex_handle v2,
Face_handle n0,
Face_handle n1,
Face_handle n2)*

constructor setting the incident vertices and the neighboring faces.

————— *advanced* —————

Access Functions

Interval_3 f.get_ranges(int i)

returns the interval associated with the edge indexed with i , which contains three alpha values $\alpha_1 \leq \alpha_2 \leq \alpha_3$, such as for α between α_1 and α_2 , the edge indexed with i is attached but singular, for α between α_2 and α_3 , the edge is regular, and for α greater than α_3 , the edge is interior.

<i>FT</i>	<i>f.get_alpha()</i>	return the alpha value, under which the alpha shape contains the face.
-----------	----------------------	------------------------------------------------------------------------

Modifiers

<div style="border-top: 1px solid black; width: 100%; position: relative;"> <div style="position: absolute; right: 0; top: -10px;"><i>advanced</i></div> </div>		
<i>void</i>	<i>f.set_ranges(int i, Interval_3 V)</i>	sets the interval associated with the edge indexed with <i>i</i> , which contains three alpha values $\alpha_1 \leq \alpha_2 \leq \alpha_3$, such as for α between α_1 and α_2 , the edge indexed with <i>i</i> is attached but singular, for α between α_2 and α_3 , the edge is regular, and for α greater than α_3 , the edge is interior.
<i>void</i>	<i>f.set_alpha(FT A)</i>	sets the alpha value, under which the alpha shape contains the face.
<div style="border-top: 1px solid black; width: 100%; position: relative;"> <div style="position: absolute; right: 0; top: -10px;"><i>advanced</i></div> </div>		

CGAL::Alpha_shape_face_base_2<AlphaShapeTraits_2, TriangulationFaceBase_2>

Definition

The class *Alpha_shape_face_base_2<AlphaShapeTraits_2, TriangulationFaceBase_2>* is the default model for the concept *AlphaShapeFace_2*.

```
#include <CGAL/Alpha_shape_face_base_2.h>
```

Is Model for the Concepts

AlphaShapeFace_2

Inherits From

TriangulationFaceBase_2

AlphaShapeTraits_2

Definition

A model of the concept AlphaShapeTraits_2 must provide the following predicate and operations in addition to the requirements for the underlying triangulation traits class. It means, the metric has to be euclidean for Delaunay triangulation or the power metric for regular triangulation.

Refines

TriangulationTraits_2

Types

AlphaShapeTraits_2::FT

A type to hold a coordinate type class. The type must provide a copy constructor, assignment, comparison operators, negation, multiplication, division and allow the declaration and initialization with a small integer constant (cf. requirements for number types).

Precondition: An obvious choice would be coordinate type of the point class.

Creation

Only a default constructor is required. Note that further constructors can be provided.

AlphaShapeTraits_2 t;

A default constructor.

Constructions by function objects

Compute_squared_radius_2

t.compute_squared_radius_2_object()

Returns an object, which has to be able to compute the squared radius of the circle of the points p_0 , p_1 , p_2 or the squared radius of smallest circle of the points p_0 , p_1 , as *FT* associated with *the metric used by Dt*.

Predicate by function object

Side_of_bounded_circle_2

t.side_of_bounded_circle_2_object()

Returns an object, which has to be able to compute the relative position of point *test* to the smallest circle of the points p_0 , p_1 , using *the same metric as Dt*.

CGAL::Weighted_alpha_shape_euclidean_traits_2<K>

Definition

The class *Weighted_alpha_shape_euclidean_traits_2<K>* is the default model for the concept *AlphaShapeTraits_2* for the regular version of Alpha Shapes. *K* must be a kernel.

```
#include <CGAL/Weighted_alpha_shape_euclidean_traits_2.h>
```

Refines

Regular_triangulation_euclidean_traits_2<K, typename K::FT>

Is Model for the Concepts

AlphaShapeTraits_2

AlphaShapeVertex_2

Definition

Refines

TriangulationVertexBase_2.

Types

AlphaShapeVertex_2:: FT

A type to hold a coordinate type class. The type must provide a copy constructor, assignment, comparison operators, negation, multiplication, division and allow the declaration and initialization with a small integer constant (cf. requirements for number types). An obvious choice would be coordinate type of the point class.

Creation

— *advanced* —

```
AlphaShapeVertex_2 v;
AlphaShapeVertex_2 v( Point p);
AlphaShapeVertex_2 v( Point p, Face_handle ff);
```

default constructor.
constructor setting the point.

constructor setting the point associated to and an incident face.

— *advanced* —

Access Functions

std::pair< FT, FT > v.get_range()

returns two alpha values $\alpha_1 \leq \alpha_2$, such as for α between α_1 and α_2 , the vertex is attached but singular, and for α upper α_2 , the vertex is regular.

Modifiers

— *advanced* —

```
void v.set_range( std::pair< FT, FT > I)
```

sets the alpha values $\alpha_1 \leq \alpha_2$, such as for α between α_1 and α_2 , the vertex is attached but singular, and for α upper α_2 , the vertex is regular.

— *advanced* —

CGAL::Alpha_shape_vertex_base_2<AlphaShapeTraits_2>

Definition

The class *Alpha_shape_vertex_base_2<AlphaShapeTraits_2>* is the default model for the concept *AlphaShapeVertex_2*.

```
#include <CGAL/Alpha_shape_vertex_base_2.h>
```

Is Model for the Concepts

AlphaShapeVertex_2

Inherits From

TriangulationVertexBase_2

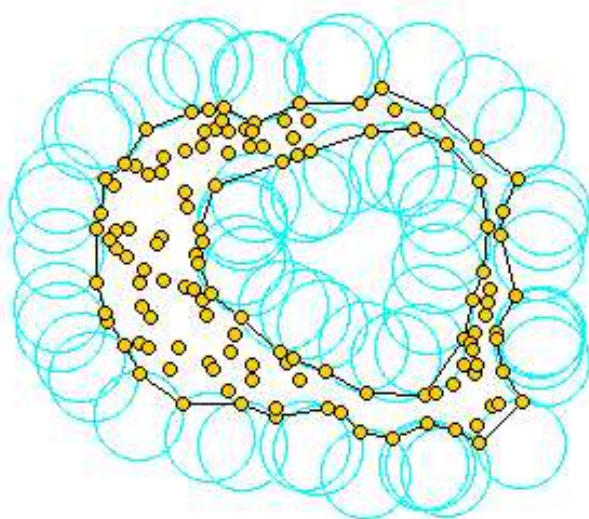
Chapter 25

3D Alpha Shapes

Tran Kai Frank Da and Mariette Yvinec

Contents

25.1 Definitions	1621
25.2 Functionality	1622
25.3 Concepts and Models	1622
25.4 Examples	1623
25.4.1 Example for Basic Alpha-Shapes	1623
25.4.2 Building Basic Alpha Shapes for Many Points	1624
25.4.3 Example for Weighted Alpha-Shapes	1625



Assume we are given a set S of points in 2D or 3D and we'd like to have something like “the shape formed by these points.” This is quite a vague notion and there are probably many possible interpretations, the alpha shape being one of them. Alpha shapes can be used for shape reconstruction from a dense unorganized set of data points. Indeed, an alpha shape is demarcated by a frontier, which is a linear approximation of the original shape [BB97].

As mentioned in Edelsbrunner's and Mücke's paper [EM94], one can intuitively think of an alpha shape as the following. Imagine a huge mass of ice-cream making up the space \mathbb{R}^3 and containing the points as “hard” chocolate pieces. Using one of those sphere-formed ice-cream spoons we carve out all parts of the ice-cream block we can reach without bumping into chocolate pieces, thereby even carving out holes in the inside (eg. parts not reachable by simply moving the spoon from the outside). We will eventually end up with a (not necessarily convex) object bounded by caps, arcs and points. If we now straighten all “round” faces to triangles and line segments, we have an intuitive description of what is called the alpha shape of S . Here's an example for this process in 2D (where our ice-cream spoon is simply a circle):

Alpha shapes depend on a parameter α from which they are named. What is α in the ice-cream game? α is the squared radius of the carving spoon. A very small value will allow us to eat up all of the ice-cream except the chocolate points themselves. Thus we already see that the alpha shape degenerates to the point-set S for $\alpha \rightarrow 0$. On the other hand, a huge value of α will prevent us even from moving the spoon between two points since it's way too large. So we will never spoon up ice-cream lying in the inside of the convex hull of S , and hence the alpha shape for $\alpha \rightarrow \infty$ is the convex hull of S .¹

25.1 Definitions

More precisely, the definition of alpha shapes is based on an underlying triangulation that may be a Delaunay triangulation in case of basic alpha shapes or a regular triangulation (cf. 22.3) in case of weighted alpha shapes.

Let us consider the basic case with a Delaunay triangulation. We first define the alpha complex of the set of points S . The alpha complex is a subcomplex of the Delaunay triangulation. For a given value of α , the alpha complex includes all the simplices in the Delaunay triangulation which have an empty circumsphere with squared radius equal or smaller than α . Here “empty” means that the open sphere does not include any points of S . The alpha shape is then simply the domain covered by the simplices of the alpha complex (see [EM94]).

In general, an alpha complex is a non-connected and non-pure complex. This means in particular that the alpha complex may have singular faces. For $0 \leq k \leq d - 1$, a k -simplex of the alpha complex is said to be singular if it is not a facet of a $(k + 1)$ -simplex of the complex. CGAL provides two versions of the alpha shapes. In the general mode, the alpha shapes correspond strictly to the above definition. The regularized mode provides a regularized version of the alpha shapes corresponding to the domain covered by a regularized version of the alpha complex where singular faces are removed.

The alpha shapes of a set of points S form a discrete family, even though they are defined for all real numbers α . The entire family of alpha shapes can be represented through the underlying triangulation of S . In this representation each k -simplex of the underlying triangulation is associated with an interval that specifies for which values of α the k -simplex belongs to the alpha complex. Relying on this fact, the family of alpha shapes can be computed efficiently and relatively easily. Furthermore, we can select the optimal value of α to get an alpha shape including all data points and having less than a given number of connected components. Also, the alpha-values allow to define a filtration on the faces of the triangulation of a set of points. In this filtration, the faces of the triangulation are output in increasing order of the alpha value for which they appear in the alpha complex. In case of equal alpha value lower dimensional faces are output first.

¹ice cream, ice cream!!! The wording of this introductory paragraph is borrowed from Kaspar Fischer's “Introduction to Alpha Shapes” which can be found at <http://n.ethz.ch/student/fischerk/alphashapes/as/index.html>. The picture has been taken from Walter Luh's homepage at <http://www.stanford.edu/~wluh/cs448b/alphashapes.html>.

The definition is analog in the case of weighed alpha shapes. The input set is now a set of weighted points (which can be regarded as spheres) and the underlying triangulation is the regular triangulation of this set. Two spheres, or two weighted points, with centers C_1, C_2 and radii r_1, r_2 are said to be orthogonal iff $C_1 C_2^2 = r_1^2 + r_2^2$ and suborthogonal iff $C_1 C_2^2 < r_1^2 + r_2^2$. For a given value of α the weighted alpha complex is formed with the simplices of the regular triangulation such that there is a sphere orthogonal to the weighted points associated with the vertices of the simplex and suborthogonal to all the other input weighted points. Once again the alpha shape is then defined as the domain covered by the alpha complex and arise in two versions general or regularized.

25.2 Functionality

The class `CGAL::Alpha_shape_3<Dt>` represents the whole family of alpha shapes for a given set of points. The class includes the underlying triangulation Dt of the set, and associates to each k -face of this triangulation an interval specifying for which values of α the face belongs to the alpha complex.

The class `CGAL::Alpha_shape_3<Dt>` provides functions to set and get the current α -value, as well as an iterator that enumerates the α values where the alpha shape changes.

The class provides member functions to classify for a given value of *alpha* the different faces of the triangulation as *EXTERIOR*, *SINGULAR*, *REGULAR* or *INTERIOR* with respect to the alpha shape. A k face on the boundary of the alpha complex is said to be *REGULAR* if it is a subface of the alpha complex which is a subface of some $k + 1$ face of the alpha complex and *SINGULAR* otherwise.

The class provides also output iterators to get for a given *alpha* value the vertices, edges, facets and cells of the different types (*EXTERIOR*, *SINGULAR*, *REGULAR* or *INTERIOR*).

Also the class has a filtration member function that, given an output iterator with `CGAL::object` as value type, outputs the faces of the triangulation according to the order of apparition in the alpha complex when alpha increases.

Finally, it provides a function to determine the smallest value α such that the alpha shape satisfies the following two properties

- (ii) all data points are either on the boundary or in the interior of the regularized version of the alpha shape (no singular faces).
- (i) The number of components is equal or less than a given number .

The current implementation is static, that is after its construction points cannot be inserted or removed.

25.3 Concepts and Models

We currently do not specify concepts for the underlying triangulation type. Models that work for a basic alpha-shape are the classes `CGAL::Delaunay_triangulation_3` and `CGAL::Triangulation_hierarchy_3` templated with a Delaunay triangulation. A model that works for a weighted alpha-shape is the class `CGAL::Regular_triangulation_3`.

The triangulation needs a geometric traits class as argument. The requirements of this class are described in the concept `CGAL::AlphaShapeTraits_3` for which the CGAL kernels are models in the non-weighted case, and for which the class `CGAL::Weighted_alpha_shape_euclidean_traits_3` is model in the weighted case.

The triangulation data structure of the triangulation with any has to be a model of the concept `CGAL::TriangulationDataStructure_3`. However it must be parameterized with vertex and cell classes, which

are model of the concepts *AlphaShapeVertex_3* and *AlphaShapeCell_3*. The package provides by default the classes *CGAL::Alpha_shape_vertex_base_3<Gt>* and *CGAL::Alpha_shape_cell_base_3<Gt>*.

25.4 Examples

25.4.1 Example for Basic Alpha-Shapes

This example builds a basic alpha shape using a Delaunay triangulation as underlying triangulation.

```
// examples/Alpha_shapes_3/example_alpha.C

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_3.h>
#include <CGAL/Alpha_shape_3.h>

#include <fstream>
#include <list>

struct K : CGAL::Exact_predicates_inexact_constructions_kernel {};

typedef CGAL::Alpha_shape_vertex_base_3<K>          Vb;
typedef CGAL::Alpha_shape_cell_base_3<K>           Fb;
typedef CGAL::Triangulation_data_structure_3<Vb,Fb> Tds;
typedef CGAL::Delaunay_triangulation_3<K,Tds>      Triangulation_3;
typedef CGAL::Alpha_shape_3<Triangulation_3>       Alpha_shape_3;

typedef K::Point_3 Point;
typedef Alpha_shape_3::Alpha_iterator Alpha_iterator;

int main()
{
    std::list<Point> lp;

    //read input
    std::ifstream is("../data/bunny_1000");
    int n;
    is >> n;
    std::cout << "Reading " << n << " points " << std::endl;
    Point p;
    for( ; n>0 ; n--) {
        is >> p;
        lp.push_back(p);
    }

    // compute alpha shape
    Alpha_shape_3 as(lp.begin(),lp.end());
    std::cout << "Alpha shape computed in REGULARIZED mode by default"
        << std::endl;

    // find optimal alpha value
    Alpha_iterator opt = as.find_optimal_alpha(1);
```

```

std::cout << "Optimal alpha value to get one connected component is "
    << *opt    << std::endl;
as.set_alpha(*opt);
assert(as.number_of_solid_components() == 1);
return 0;
}

```

25.4.2 Building Basic Alpha Shapes for Many Points

When many points are input in the alpha shape, say more than 10 000, it pays off to use a triangulation hierarchy as underlying triangulation (cf. [22.4](#)).

```

// examples/Alpha_shapes_3/example_big_alpha.C

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_3.h>
#include <CGAL/Triangulation_hierarchy_3.h>
#include <CGAL/Alpha_shape_3.h>

#include <fstream>
#include <list>

struct K : CGAL::Exact_predicates_inexact_constructions_kernel {};

typedef CGAL::Alpha_shape_vertex_base_3<K>          Vb;
typedef CGAL::Triangulation_hierarchy_vertex_base_3<Vb> Vbh;
typedef CGAL::Alpha_shape_cell_base_3<K>           Fb;
typedef CGAL::Triangulation_data_structure_3<Vbh,Fb> Tds;
typedef CGAL::Delaunay_triangulation_3<K,Tds>      Delaunay;
typedef CGAL::Triangulation_hierarchy_3<Delaunay>  Delaunay_hierarchy;
typedef CGAL::Alpha_shape_3<Delaunay_hierarchy>    Alpha_shape_3;

typedef K::Point_3 Point;
typedef Alpha_shape_3::Alpha_iterator Alpha_iterator;
typedef Alpha_shape_3::NT NT;

int main()
{
    Delaunay_hierarchy dt;
    std::ifstream is("../data/bunny_1000");
    int n;
    is >> n;
    Point p;
    std::cout << n << " points read" << std::endl;
    for( ; n>0 ; n--) {
        is >> p;
        dt.insert(p);
    }
    std::cout << "Delaunay computed." << std::endl;
}

```



```

// compute alpha shape
Alpha_shape_3 as(dt);
std::cout << "Alpha shape computed in REGULARIZED mode by default."
    << std::endl;

// find optimal alpha values
Alpha_shape_3::NT alpha_solid = as.find_alpha_solid();
Alpha_iterator opt = as.find_optimal_alpha(1);
std::cout << "Smallest alpha value to get a solid through data points is "
    << alpha_solid << std::endl;
std::cout << "Optimal alpha value to get one connected component is "
    << *opt << std::endl;
as.set_alpha(*opt);
assert(as.number_of_solid_components() == 1);
return 0;
}

```

25.4.3 Example for Weighted Alpha-Shapes

The following examples build a weighted alpha shape requiring a regular triangulation as underlying triangulation. The alpha shape is build in *GENERAL* mode.

```

// examples/Alpha_shapes_3/example_weight.C

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Weighted_alpha_shape_euclidean_traits_3.h>
#include <CGAL/Regular_triangulation_3.h>
#include <CGAL/Alpha_shape_3.h>
#include <list>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;

typedef CGAL::Weighted_alpha_shape_euclidean_traits_3<K> Gt;

typedef CGAL::Alpha_shape_vertex_base_3<Gt> Vb;
typedef CGAL::Alpha_shape_cell_base_3<Gt> Fb;
typedef CGAL::Triangulation_data_structure_3<Vb,Fb> Tds;
typedef CGAL::Regular_triangulation_3<Gt,Tds> Triangulation_3;
typedef CGAL::Alpha_shape_3<Triangulation_3> Alpha_shape_3;

typedef Alpha_shape_3::Cell_handle Cell_handle;
typedef Alpha_shape_3::Vertex_handle Vertex_handle;
typedef Alpha_shape_3::Facet Facet;
typedef Alpha_shape_3::Edge Edge;
typedef Gt::Weighted_point Weighted_point;
typedef Gt::Bare_point Bare_point;

int main()
{
    std::list<Weighted_point> lwp;

    //input : a small molecule

```

```

lwp.push_back(Weighted_point(Bare_point( 1, -1, -1), 4));
lwp.push_back(Weighted_point(Bare_point(-1, 1, -1), 4));
lwp.push_back(Weighted_point(Bare_point(-1, -1, 1), 4));
lwp.push_back(Weighted_point(Bare_point( 1, 1, 1), 4));
lwp.push_back(Weighted_point(Bare_point( 2, 2, 2), 1));

//build alpha_shape in GENERAL mode and set alpha=0
Alpha_shape_3 as(lwp.begin(), lwp.end(), 0, Alpha_shape_3::GENERAL);

//explore the 0-shape - It is dual to the boundary of the union.
std::list<Cell_handle> cells;
std::list<Facet> facets;
std::list<Edge> edges;
as.get_alpha_shape_cells(std::back_inserter(cells),
    Alpha_shape_3::INTERIOR);
as.get_alpha_shape_facets(std::back_inserter(facets),
    Alpha_shape_3::REGULAR);
as.get_alpha_shape_facets(std::back_inserter(facets),
    Alpha_shape_3::SINGULAR);
as.get_alpha_shape_edges(std::back_inserter(edges),
    Alpha_shape_3::SINGULAR);
std::cout << " The 0-shape has : " << std::endl;
std::cout << cells.size() << " interior tetrahedra" << std::endl;
std::cout << facets.size() << " boundary facets" << std::endl;
std::cout << edges.size() << " singular edges" << std::endl;
return 0;
}

```

3D Alpha Shapes

Reference Manual

Tran Kai Frank Da and Mariette Yvinec

Alpha shapes definition is based on an underlying triangulation that may be a Delaunay triangulation in case of basic alpha shapes or a regular triangulation (cf. [22.3](#)) in case of weighted alpha shapes.

Let us consider the basic case with a Delaunay triangulation. We first define the alpha complex of the set of points S . The alpha complex is a subcomplex of the Delaunay triangulation. For a given value of α , the alpha complex includes all the simplices in the Delaunay triangulation which have an empty circumsphere with squared radius equal or smaller than α . Here “empty” means that the open sphere do not include any points of S . The alpha shape is then simply the domain covered by the simplices of the alpha complex (see [\[EM94\]](#)).

In general, an alpha complex is a non-connected and non-pure complex. This means in particular that the alpha complex may have singular faces. For $0 \leq k \leq d - 1$, a k -simplex of the alpha complex is said to be singular if it is not a facet of a $(k + 1)$ -simplex of the complex. CGAL provides two versions of the alpha shapes. In the general mode, the alpha shapes correspond strictly to the above definition. The regularized mode provides a regularized version of the alpha shapes corresponding to the domain covered by a regularized version of the alpha complex where singular faces are removed.

The alpha shapes of a set of points S form a discrete family, even though they are defined for all real numbers α . The entire family of alpha shapes can be represented through the underlying triangulation of S . In this representation each k -simplex of the underlying triangulation is associated with an interval that specifies for which values of α the k -simplex belongs to the alpha complex. Relying on this fact, the family of alpha shapes can be computed efficiently and relatively easily. Furthermore, we can select the optimal value of α to get an alpha shape including all data points and having less than a given number of connected components.

The definition is analog in the case of weighted alpha shapes. The input set is now a set of weighted points (which can be regarded as spheres) and the underlying triangulation is the regular triangulation of this set. Two spheres, or two weighted points, with centers C_1, C_2 and radii r_1, r_2 are said to be orthogonal iff $C_1 C_2^2 = r_1^2 + r_2^2$ and suborthogonal iff $C_1 C_2^2 < r_1^2 + r_2^2$. For a given value of α the weighted alpha complex is formed with the simplices of the regular triangulation such that there is a sphere orthogonal to the weighted points associated with the vertices of the simplex and suborthogonal to all the other input weighted points. Once again the alpha shape is then defined as the domain covered by the alpha complex and arise in two versions general or regularized.

25.5 Classified Reference Pages

Concepts

<code>AlphaShapeTraits_3</code>	page 1631
<code>WeightedAlphaShapeTraits_3</code>	page 1643
<code>AlphaShapeCell_3</code>	page 1629
<code>AlphaShapeVertex_3</code>	page 1632

Classes

<code>CGAL::Alpha_status<NT></code>	page 1641
<code>CGAL::Alpha_shape_3<Dt></code>	page 1633
<code>CGAL::Weighted_alpha_shape_euclidean_traits_3<K></code>	page 1644
<code>CGAL::Alpha_shape_vertex_base_3<Traits,Vb></code>	page 1640
<code>CGAL::Alpha_shape_cell_base_3<Traits,Fb></code>	page 1639

25.6 Alphabetical List of Reference Pages

<code>AlphaShapeCell_3</code>	page 1629
<code>AlphaShapeTraits_3</code>	page 1631
<code>AlphaShapeVertex_3</code>	page 1632
<code>Alpha_shape_3<Dt></code>	page 1633
<code>Alpha_shape_cell_base_3<Traits,Fb></code>	page 1639
<code>Alpha_shape_vertex_base_3<Traits,Vb></code>	page 1640
<code>Alpha_status<NT></code>	page 1641
<code>WeightedAlphaShapeTraits_3</code>	page 1643
<code>Weighted_alpha_shape_euclidean_traits_3<K></code>	page 1644

AlphaShapeCell_3

Definition

This concept describes the requirements for the base cell of an alpha shape.

Refines

TriangulationCellBase_3.

AlphaShapeCell_3:: NT A number type. Must be the same as the number type used in the traits class of the triangulation underlying the alpha shape.

AlphaShapeCell_3:: Alpha_status_iterator An iterator with value type *CGAL::Alpha_status<NT>*.

Creation

AlphaShapeCell_3 f; default constructor.

AlphaShapeCell_3 f(Vertex_handle v0, Vertex_handle v1, Vertex_handle v2, Vertex_handle v3);

constructor setting the incident vertices.

*AlphaShapeCell_3 f(Vertex_handle v0,
Vertex_handle v1,
Vertex_handle v2,
Vertex_handle v3,
Cell_handle n0,
Cell_handle n1,
Cell_handle n2,
Cell_handle n3)*

constructor setting the incident vertices and the neighboring cells.

Access Functions

NT *f.get_alpha()* Returns the alpha value of the cell.

Alpha_status_iterator

f.get_facet_status(int i)

Returns an iterator on the *CGAL::Alpha_status<NT>* of the facet *i* of the cell.

Modifiers

void *f.set_alpha(NT alpha)*

Sets the critical value of the cell.

void *f.set_facet_status(int i, Alpha_status_iterator as)*

Sets the iterator pointing to the *CGAL::Alpha_status<NT>* of the facet *i* of the cell.

See Also

CGAL::Alpha_status

AlphaShapeTraits_3

Definition

The concept AlphaShapeTraits_3 describes the requirements for the geometric traits class of the underlying Delaunay triangulation of a basic alpha shape.

Generalizes

DelaunayTriangulationTraits_3

In addition to the requirements described in the concept *DelaunayTriangulationTraits_3*, the geometric traits class of a Delaunay triangulation plugged in a basic alpha shapes provides the following.

Types

AlphaShapeTraits_3::NT A number type compatible with the type used for the points coordinate.

AlphaShapeTraits_3::Compute_squared_radius_3

An object constructor able to compute the squared radius of the smallest circumsphere of 4 points p_0, p_1, p_2, p_3 , the squared radius of the smallest circumsphere of 3 points p_0, p_1, p_2 , and the squared radius of smallest circumsphere of the points p_0, p_1 .

Creation

AlphaShapeTraits_3 traits; Default constructor.

Access Functions

Compute_squared_radius_3 *traits.compute_squared_radius_3_object()*

Has Models

All CGAL kernels.

CGAL::Exact_predicates_inexact_constructions_kernel (recommended)

CGAL::Exact_predicates_exact_constructions_kernel

CGAL::Filtered_kernel

CGAL::Cartesian

CGAL::Simple_cartesian

CGAL::Homogeneous

CGAL::Simple_homogeneous

AlphaShapeVertex_3

Definition

This concept describe the requirements for the base vertex of an alpha shape.

Refines

TriangulationVertexBase_3.

Types

AlphaShapeVertex_3:: Point

Must be the same as the point type provided by the geometric traits class of the triangulation.

AlphaShapeVertex_3:: Alpha_status;

Must be *CGAL::Alpha_status<NT>* where *NT* is the number type used in the geometric traits class of the triangulation.

Creation

AlphaShapeVertex_3 v;

default constructor.

AlphaShapeVertex_3 v(Point p);

constructor setting the point associated to.

AlphaShapeVertex_3 v(Point p, Cell_handle c);

constructor setting the point associated to and an incident cell.

Modifiers

*Alpha_status**

v.get_alpha_status()

Returns a pointer the alpha status of the vertex.

See Also

CGAL::Alpha_status

CGAL::Alpha_shape_3<Dt>

Definition

The class *Alpha_shape_3<Dt>* represents the family of alpha shapes of points in the 3D space for *all* positive α . It maintains an underlying triangulation of the class *Dt* which represents connectivity and order among its faces. Each *k*-dimensional face of the *Dt* is associated with an interval that specifies for which values of α the face belongs to the alpha shape.

Note that this class is at the same time used for *basic* and for *weighted* Alpha Shapes.

Inherits From

Dt

This class is the underlying triangulation class.

The modifying functions *insert* and *remove* will overwrite the inherited functions. At the moment, only the static version is implemented.

Types

Alpha_shape_3<Dt>:: Gt the alpha shape traits type.

it has to derive from a triangulation traits class. For example *Dt::Point* is a Point class.

typedef Gt::FT

NT; the number type of alpha values.

Alpha_shape_3<Dt>:: Alpha_iterator

A bidirectional and non-mutable iterator that allow to traverse the increasing sequence of different alpha values.

Precondition: Its *value_type* is *NT*

enum Mode { GENERAL, REGULARIZED};

In GENERAL mode, the alpha complex can have singular faces, i. e. faces of dimension *k*, for $k = (0, 1, 2)$ that are not subfaces of a $k + 1$ face of the complex. In REGULARIZED mode, the complex is regularized, that is singular faces are dropped and the alpha complex includes only a subset of the tetrahedral cells of the triangulation and the subfaces of those cells.

enum Classification_type { EXTERIOR, SINGULAR, REGULAR, INTERIOR};

Enum to classify the faces of the underlying triangulation with respect to the alpha shape.

In GENERAL mode, for $k = (0, 1, 2)$, each k -dimensional simplex of the triangulation can be classified as EXTERIOR, SINGULAR, REGULAR or INTERIOR. In GENERAL mode a k simplex is REGULAR if it is on the boundary of the alpha complex and belongs to a $k + 1$ simplex in this complex and it is SINGULAR if it is a boundary simplex that is not included in a $k + 1$ simplex of the complex.

In REGULARIZED mode, for $k = (0, 1, 2)$ each k -dimensional simplex of the triangulation can be classified as EXTERIOR, REGULAR or INTERIOR, i.e. there is no singular faces. A k simplex is REGULAR if it is on the boundary of alpha complex and belongs to a tetrahedral cell of the complex.

Creation

Alpha_shape_3<Dt> A(FT alpha = 0, Mode m = REGULARIZED);

Introduces an empty alpha shape data structure A for and set the current alpha value to $alpha$ and the mode to m .

Alpha_shape_3<Dt> A(Dt& dt, NT alpha = 0, Mode m = REGULARIZED);

Build an alpha shape of mode m from the triangulation dt . Be careful that this operation destroy the triangulation.

template < class InputIterator >

Alpha_shape_3<Dt> A(InputIterator first, InputIterator last, FT alpha = 0, Mode m = REGULARIZED);

Build an alpha shape data structure in mode m for the points in the range $[first, last)$ and set the current alpha value to $alpha$.

Precondition: The *value_type* of *first* and *last* is *Point* (the type point of the underlying triangulation.)

Modifiers

template < class InputIterator >

int A.make_alpha_shape(InputIterator first, InputIterator last)

Initialize the alpha shape data structure for points in the range $[first, last)$. Returns the number of data points inserted in the underlying triangulation.

If the function is applied to a non-empty alpha shape data structure, it is cleared before initialization.

Precondition: The *value_type* of *first* and *last* is *Point*.

void A.clear() Clears the structure.

<i>FT</i>	<i>A.set_alpha(Coord_type alpha)</i>	Sets the α -value to <i>alpha</i> . Returns the previous α -value. <i>Precondition: alpha</i> ≥ 0 .
<i>Mode</i>	<i>A.set_mode(Mode m = REGULARIZED)</i>	Sets <i>A</i> in GENERAL or REGULARIZED. Returns the previous mode. Changing the mode of an alpha shape data structure entails a partial recomputation of the data structure.

Query Functions

<i>Mode</i>	<i>A.get_mode(void)</i>	Returns whether <i>A</i> is general or regularized.
<i>FT</i>	<i>A.get_alpha(void)</i>	Returns the current α -value.
<i>FT</i>	<i>A.get_nth_alpha(int n)</i>	Returns the <i>n</i> -th alpha-value, sorted in an increasing order. <i>Precondition: n</i> \leq number of alphas.
<i>int</i>	<i>A.number_of_alphas()</i>	Returns the number of different alpha-values.
<i>Classification_type</i>	<i>A.classify(Point p, FT alpha = get_alpha())</i>	Locates a point <i>p</i> in the underlying triangulation and Classifies the associated k-face with respect to <i>alpha</i> .
<i>Classification_type</i>	<i>A.classify(Cell_handle f, FT alpha = get_alpha())</i>	Classifies the cell <i>f</i> of the underlying triangulation with respect to <i>alpha</i> .
<i>Classification_type</i>	<i>A.classify(Facet f, FT alpha = get_alpha())</i>	Classifies the facet <i>e</i> of the underlying triangulation with respect to <i>alpha</i> .

Classification_type

A.classify(Cell_handle f, int i, FT alpha = get_alpha())

Classifies the facet of the cell *f* opposite to the vertex with index *i* of the underlying triangulation with respect to *alpha*.

Classification_type

A.classify(Edge e, FT alpha = get_alpha())

Classifies the edge *e* with respect to *alpha* .

Classification_type

A.classify(Vertex_handle v, FT alpha = get_alpha())

Classifies the vertex *v* of the underlying triangulation with respect to *alpha*.

template<class OutputIterator>
OutputIterator

A.get_alpha_shape_cells(OutputIterator it,
Classification_type type,
NT alpha = get_alpha())

Write the cells which are of type *type* for the alpha value *alpha* to the sequence pointed to by the output iterator *it*. Return past the end of the output sequence.

template<class OutputIterator>
OutputIterator

A.get_alpha_shape_facets(OutputIterator it,
Classification_type type,
NT alpha= get_alpha())

Write the facets which are of type *type* for the alpha value *alpha* to the sequence pointed to by the output iterator *it*. Return past the end of the output sequence.

template<class OutputIterator>
OutputIterator

A.get_alpha_shape_edges(OutputIterator it,
Classification_type type,
NT alpha = get_alpha())

Write the edges which are of type *type* for the alpha value *alpha* to the sequence pointed to by the output iterator *it*. Return past the end of the output sequence.

template<class OutputIterator>
OutputIterator

A.get_alpha_shape_vertices(OutputIterator it,
Classification_type type,
NT alpha get_alpha())

Write the vertices which are of type *type* for the alpha value *alpha* to the sequence pointed to by the output iterator *it*. Return past the end of the output sequence.

template<class OutputIterator>
OutputIterator

A.filtration(OutputIterator it)

Output all the faces of the triangulation in increasing order of the alpha value for which they appear in the alpha complex. In case of equal alpha value lower dimensional faces are output first. The value type of the OutputIterator has to be a CGAL::Object

Traversal of the α -Values

Alpha_iterator

A.alpha_begin() Returns an iterator that allows to traverse the sorted sequence of α -values of the family of alpha shapes.

Alpha_iterator

A.alpha_end() Returns the corresponding past-the-end iterator.

Alpha_iterator

A.alpha_find(FT alpha)

Returns an iterator pointing to an element with α -value *alpha*, or the corresponding past-the-end iterator if such an element is not found.

Alpha_iterator

A.alpha_lower_bound(FT alpha)

Returns an iterator pointing to the first element with α -value not less than *alpha*.

Alpha_iterator

A.alpha_upper_bound(FT alpha)

Returns an iterator pointing to the first element with α -value greater than *alpha*.

Operations

int *A.number_of_solid_components(FT alpha = get_alpha())*

Returns the number of solid components of *A*, that is, the number of components of its regularized version.

Alpha_iterator

A.find_optimal_alpha(int nb_components)

Returns an iterator pointing to smallest α value such that *A* satisfies the following two properties:

all data points are either on the boundary or in the interior of the regularized version of *A*.

The number of solid component of *A* is equal to or smaller than *nb_components*.

I/O

The I/O operators are defined for *iostream*, and for the window stream provided by CGAL. The format for the *iostream* is an internal format.

#include <CGAL/IO/io.h>

ostream& *ostream& os << A* Inserts the alpha shape *A* for the current alpha value into the stream *os*.
Precondition: The insert operator must be defined for *Point*.

#include <CGAL/IO/Geomview_stream.h>

#include <CGAL/IO/alpha_shape_geomview_ostream_3.h>

Geomview_stream&

Geomview_stream& W << A

Inserts the alpha shape *A* for the current alpha value into the Geomview stream *W*.

Precondition: The insert operator must be defined for *Point* and *Triangle*.

Implementation

In GENERAL mode, the alpha intervals of each triangulation face is computed and stored at initialization time. In REGULARIZED mode, the alpha shape intervals of edges are not stored nor computed at initialization. Edges are simply classified on the fly upon request. This allows to have much faster building of alpha shapes in REGULARIZED mode.

A.alpha find uses linear search, while *A.alpha lower bound* and *A.alpha upper bound* use binary search. *A.number of solid components* performs a graph traversal and takes time linear in the number of cells of the underlying triangulation. *A.find of optimal alpha* uses binary search and takes time $O(n \log n)$, where *n* is the number of points.

CGAL::Alpha_shape_cell_base_3<Traits,Fb>

Definition

The class *Alpha_shape_cell_base_3<Traits,Fb>* is the default model for the concept *AlphaShapeCell_3*.

The class has two parameters. The traits class *Traits* provides the number type for alpha values. The second parameter *Fb* is a base class instantiated by default with *CGAL::Triangulation_cell_base_3<Traits>*.

```
#include <CGAL/Alpha_shape_cell_base_3.h>
```

Is Model for the Concepts

AlphaShapeCellBase_3

Inherits From

Fb

CGAL::Alpha_shape_vertex_base_3<Traits,Vb>

Definition

The class *Alpha_shape_vertex_base_3<Traits,Vb>* is the default model for the concept *AlphaShapeVertex_3*.

The class has two parameters : the traits class *Traits* which provides the type for the points or the weighted points. The second parameter *Vb* is a base class instantiated by default with *CGAL::Triangulation_vertex_base_3<Traits>*.

```
#include <CGAL/Alpha_shape_vertex_base_3.h>
```

Is Model for the Concepts

AlphaShapeVertexBase_3

Inherits From

Vb

See Also

AlphaShapeCellBase_3

AlphaShapeVertexBase_3

WeightedAlphaShapeTraits_3

Definition

The concept `WeightedAlphaShapeTraits_3` describes the requirements for the geometric traits class of the underlying regular triangulation of a weighted alpha shape.

Generalizes

RegularTriangulationTraits_3

In addition to the requirements described in the concept *RegularTriangulationTraits_3*, the geometric traits class of a Regular triangulation plugged in a basic alpha shapes provides the following.

Types

WeightedAlphaShapeTraits_3::NT A number type compatible with the type used for the points coordinates.

WeightedAlphaShapeTraits_3::Compute_squared_radius_3

An object constructor able to compute the squared radius of the smallest sphere orthogonal to four weighted points p_0, p_1, p_2, p_3 , and the squared radius of the smallest sphere orthogonal to three weighted points p_0, p_1, p_2 , and the squared radius of smallest sphere orthogonal to two weighted points p_0, p_1 .

Creation

WeightedAlphaShapeTraits_3 wast; default constructor.

Access Functions

Compute_squared_radius_3

wast.compute_squared_radius_3_object()

Has Models

CGAL::Weighted_alpha_shape_euclidean_traits_3<K>,

CGAL::Weighted_alpha_shape_euclidean_traits_3<K>

Definition

The class *Weighted_alpha_shape_euclidean_traits_3<K>* is the default model for the concept *WeightedAlphaShapeTraits_3* of traits class for the underlying triangulation of a weighed alpha shapes. *K* must be a kernel.

```
#include <CGAL/Weighted_alpha_shape_euclidean_traits_3.h>
```

Refines

Regular_triangulation_euclidean_traits_3<K, typename K::FT>

Is Model for the Concepts

WeightedAlphaShapeTraits_3

Part VIII

Voronoi Diagrams

Chapter 26

2D Segment Delaunay Graphs

Menelaos Karavelas

Contents

26.1 Definitions	1647
26.2 Software Design	1649
26.3 The Geometric Traits	1652
26.4 The Segment Delaunay Graph Hierarchy	1653
26.5 Examples	1654
26.5.1 First Example	1654
26.5.2 Second Example	1655
26.5.3 Third Example	1656

This chapter describes the two-dimensional segment Delaunay graph package of CGAL. We start with a few definitions in Section [26.1](#). The software design of the 2D segment Delaunay graph package is described in Section [26.2](#). In Section [26.3](#) we discuss the geometric traits of the 2D segment Delaunay graph package and in Section [26.4](#) the segment Delaunay graph hierarchy, a data structure suitable for fast nearest neighbor queries, is briefly described.

26.1 Definitions

The 2D segment Delaunay graph package of CGAL is designed to compute the Delaunay graph of a set of possibly intersecting segments on the plane. Although we compute the Delaunay graph, we will often refer to its dual, the segment Voronoi diagram, since it is easier to explain and understand. The algorithm that has been implemented is incremental. The corresponding CGAL class is called *SegmentDelaunay_graph_2* <*SegmentDelaunayGraphTraits_2*, *SegmentDelaunayGraphStructure_2*> and will be discussed in more detail in the sequel. The interested reader may want to refer to the paper by Karavelas [[Kar04](#)] for the general idea as well as the details of the algorithm implemented.

Definitions. Before describing the details of the implementation we make a brief introduction to the theory of segment Delaunay graphs and segment Voronoi diagrams. The segment Voronoi diagram is defined over a set of non-intersecting sites, which can either be points or linear segments, which we assume that are given through their endpoints. The segment Voronoi diagram a subdivision of the plane into connected regions, called *cells*, associated with the sites. The dual graph of the segment Voronoi diagram is called the segment Delaunay graph.

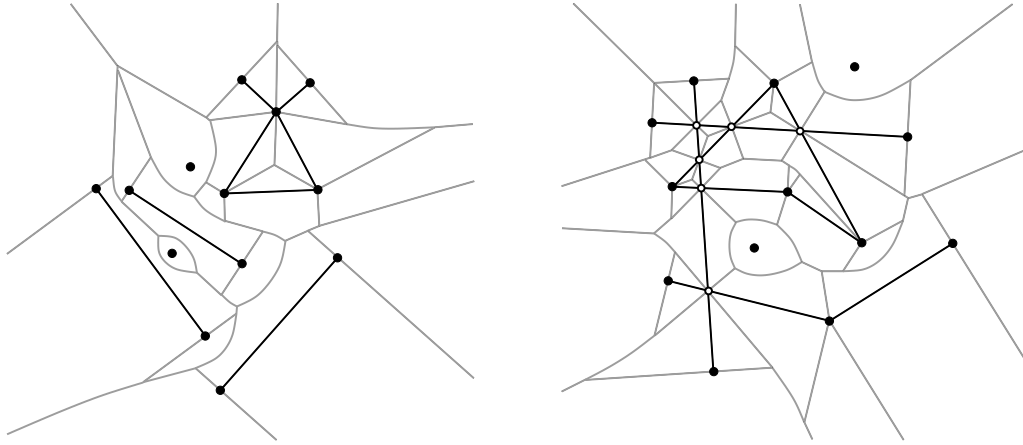


Figure 26.1: The segment Voronoi diagram for a set of weakly (left) and strongly (right) intersecting sites.

The cell of a site t_i is the locus of points on the plane that are closer to t_i than any other site t_j , $j \neq i$. The distance $\delta(x, t_i)$ of a point x in the plane to a site t_i is defined as the minimum of the Euclidean distances of x from the points in t_i . Hence, if t_i is a point p_i , then

$$\delta(x, t_i) = \|x - t_i\|,$$

whereas if t_i is a segment, then

$$\delta(x, t_i) = \min_{y \in t_i} \|x - y\|,$$

where $\|\cdot\|$ denotes the Euclidean norm. It can easily be seen that it is a generalization of the Voronoi diagram for points.

In many applications the restriction that sites are non-intersecting is too strict. Often we want to allow segments that touch at their endpoints, or even segments that overlap or intersect properly at their interior (for example, see Fig. 26.1). Allowing such configurations poses certain problems. More specifically, when we allow segments to touch at their endpoints we may end up with pairs of segments whose bisector is two-dimensional. If we allow pairs of segments that intersect properly at their interior, the interiors of their Voronoi cells are no longer simply connected. In both cases above the resulting Voronoi diagrams are no longer instances of abstract Voronoi diagrams (cf. [Kle89]), which has a direct consequence on the efficient computation of the corresponding Voronoi diagram. The remedy to these problems is to consider linear segments not as one object, but rather as three, namely the two endpoints and the interior. This choice guarantees that all bisectors in the Voronoi diagram are one-dimensional and that all Voronoi cells are simply connected. Moreover, we further distinguish between two cases, according to the type of intersecting pair that our input data set contains. A pair of sites is called *weakly intersecting* if they share a single common point and this common point does not lie in the interior of any of the two sites. A pair of sites is called *strongly intersecting* if they intersect and they either have more than one common point or their common point lies in the interior of at least one of the two sites. As it will be seen later the two cases have different representation (and thus storage) requirements, as well as they require a somehow different treatment on how the predicates are evaluated. Having made the distinction between weakly and strongly intersecting sites, and having said that segment sites are treated as three objects, we are now ready to precisely define the Delaunay graph we compute. Given a set \mathcal{S} of input sites, let $\mathcal{S}_{\mathcal{A}}$ be the set of points and (open) segments in the arrangement $\mathcal{A}(\mathcal{S})$ of \mathcal{S} . The 2D segment Delaunay graph package of CGAL computes the (triangulated) Delaunay graph that is dual to the Euclidean Voronoi diagram of the sites in the set $\mathcal{S}_{\mathcal{A}}$.

The segment Delaunay graph is uniquely defined once we have the segment Voronoi diagram. If all sites are in *general position*, then the Delaunay graph is a graph with triangular faces away from the convex hull of the set of sites. To unify our approach and handling of the Delaunay graph we add to the set of (finite) sites a fictitious site

at infinity, which we call the *site at infinity*. We can then connect all vertices of the outer face of the Delaunay graph to the site at infinity which gives us a graph with the property that all of its faces are now triangular. However, the Delaunay graph is not a triangulation for two main reasons: we cannot always embed it on the plane with straight line segments that yield a triangulation and, moreover, we may have two faces of the graph that have two edges in common, which is not allowed in a triangulation.

We would like to finish our brief introduction to the theory of segment Delaunay graphs and segment Voronoi diagrams by discussing the concept of general position. We say that a set of sites is in general position if no two triplets of sites have the same tritangent Voronoi circle. This statement is rather technical and it is best understood in the context of points. The equivalent statement for points is that we have no two triplets of points that define the same circumcircle, or equivalently that no four points are co-circular. The statement about general position made above is a direct generalization of the (much simpler to understand) statement about points. On the contrary, when we have sites in degenerate position, the Delaunay graph has faces with more than three edges on their boundary. We can get a triangulated version of the Delaunay graph by simply *triangulating* the corresponding faces in an arbitrary way. In fact the algorithm that has been implemented in CGAL has the property that it always returns a valid *triangulated* version of the segment Delaunay graph. By valid we mean that it contains the actual (non-triangulated) Delaunay graph, and whenever there are faces with more than three edges then they are triangulated. The way that they are triangulated depends on the order of insertion of the sites in the diagram.

One final remark has to be made with respect to the difference between the set of *input sites* and the set of *output sites*. The set of input sites consists of the closed sites that the user inserts in the diagram. Since segment sites are treated as three objects, internally our algorithm sees only points and open segments. As a result, from the point of view of the algorithm, the input sites have no real meaning. What has real meaning is the set of sites that correspond to cells of the Voronoi diagram and this is the set of output sites.

Degenerate Dimensions. The dimension of the segment Delaunay graph is in general 2. The exceptions to this rule are as follows:

- The dimension is -1 if the segment Delaunay graph contains no sites.
- The dimension is 0 if the segment Delaunay graph contains exactly one (output) site.
- The dimension is 1 if the segment Delaunay graph contains exactly two (output) sites.

26.2 Software Design

The 2D segment Delaunay graph class `SegmentDelaunay_graph_2<SegmentDelaunayGraphTraits_2, SegmentDelaunayGraphDataStructure_2>` follows the design of the triangulation package of CGAL. It is parametrized by two arguments:

- the **geometric traits** class. It provides the basic geometric objects involved in the algorithm, such as sites, points etc. It also provides the geometric predicates for the computation of the segment Delaunay graph, as well as some basic constructions that can be used, for example, to visualize the diagram. The geometric traits for the segment Delaunay graph will be discussed in more detail in the next section.
- the **segment Delaunay graph data structure**. This is essentially the same as the Apollonius graph data structure (discussed in Chapter 27.2), augmented with some additional operations that are specific to segment Voronoi diagrams. The corresponding concept is that of `SegmentDelaunayGraphDataStructure_2`, which in fact is a refinement of the `ApolloniusGraphDataStructure_2` concept. The class `Triangulation_data_structure_2<Vb, Fb>` is a model of the concept `SegmentDelaunayGraphDataStructure_2`. A default value for the corresponding template parameter is provided, so the user does not need to specify it.

Strongly Intersecting Sites and their Representation. As we have mentioned above, the segment Delaunay graph package of CGAL is designed to support the computation of the segment Voronoi diagram even when the input segment sites are intersecting. This choice poses certain issues for the design of the software package. The major concern is the representation of the subsegments that appear in the arrangement of these sites, because the sites in the arrangement are the ones over which the diagram is actually defined. A direct consequence of the choice of representation is the algebraic degree of the predicates involved in the computation of the segment Delaunay graph, as well as the storage requirements for the subsegments and points on intersection in the arrangement.

The case of weakly intersecting sites does not require any special treatment. We can simply represent points by their coordinates and segments by their endpoints. In the case of strongly intersecting sites, the obvious choice to use the afore-mentioned representation has severe disadvantages. Consider two strongly intersecting segments t_i and t_j , whose endpoints have homogeneous coordinates of size b . Their intersection point will have homogeneous coordinates of bit size $6b + O(1)$. This effect can be cascaded, which implies that after inserting k (input) segments we can arrive at having points of intersection whose bit sizes are exponential with respect to k , i.e., their homogeneous coordinates will have bit size $\Omega(2^k b)$. Not only the points of intersection, but also the adjacent subsegments will be represented by quantities of arbitrarily high bit size, and as a result we would not be able to give a bound on the bit sizes of the coordinates of the points of intersection. As a result, we would not be able to give a bound on the memory needed to store these coordinates. An equally important consequence is that we would also not be able to give a bound on the algebraic degree of the algebraic expressions involved in the evaluation of the predicates.

Such a behavior is obviously undesirable. For robustness, efficiency, and scalability purposes, it is critical that the bit size of the algebraic expressions in the predicates does not depend on the input size. For this reason, as well as for others to be discussed below, we decided to represent sites in an implicit manner, which somehow encodes the history of their construction. In particular, we exploit the fact that points of intersection always lie on two input segments, and that segments that are not part of the input are always supported by input segments.

For example, let us consider the configuration in Fig. 26.2. We assume that the segments $t_i = p_i q_i$, $i = 1, 2, 3$, are inserted in that order. Upon the insertion of t_2 , our algorithm will split the segment t_1 into the subsegments $p_1 s_1$ and $s_1 q_1$, then add s_1 , and finally insert the subsegments $p_2 s_1$ and $s_1 q_2$. How do we represent the five new sites? s_1 will be represented by its two defining segments t_1 and t_2 . The segment $p_1 s_1$ will be represented by two segments, a point, and a boolean. The first segment is t_1 , which is always the segment with the same support as the newly created segment. The second segment is t_2 and the point is p_1 . The boolean indicates whether the first endpoint of $p_1 s_1$ is an input point; in this case the boolean is equal to *true*. The segment $s_1 q_1$ will also be represented by two segments, a point, and a boolean, namely, t_1 (the supporting segment of $s_1 q_1$), t_2 and *false* (it is the second endpoint of $s_1 q_1$ that is an input point). Subsegments $p_2 s_2$ and $s_2 q_2$ are represented analogously. Consider now what happens when we insert t_3 . The point s_2 will again be represented by two segments, but not $s_1 q_1$ and t_3 . In fact, it will be represented by t_1 (the supporting segment of $s_1 q_1$) and t_3 . $s_2 q_1$ will be represented by two segments, a point, and a boolean (t_1 , t_3 and *false*), and similarly for $p_3 s_2$ and $s_2 q_3$. On the other hand, both endpoints of $s_1 s_2$ are non-input points. In such a case we represent the segment by three input segments. More precisely, $s_1 s_2$ is represented by the segments t_1 (the supporting segment of $s_1 q_1$), t_2 (it defines s_1 along with t_1) and t_3 (it defines s_2 along with t_1).

The five different presentations, two for points (coordinates; two input segments) and three for segments (two input points; two input segments, an input point and a boolean; three input segments), form a closed set of representations and thus represent any point of intersection or subsegment regardless of the number of input segments. Moreover, every point (input or intersection) has homogeneous coordinates of bit size at most $3b + O(1)$. The supporting lines of the segments (they are needed in some of the predicates) have coefficients which are always of bit size $2b + O(1)$. As a result, the bit size of the expressions involved in our predicates will always be $O(b)$, independently of the size of the input. The *SegmentDelaunayGraphSite_2* concept encapsulates the ideas presented above. A site is represented in this concept by up to four points and a boolean, or up to six points, depending on its type. The class *Segment_Delaunay_graph_site_2<K>* implements this concept.

Even this representation, however, has some degree of redundancy. The endpoint of a segment appears in both

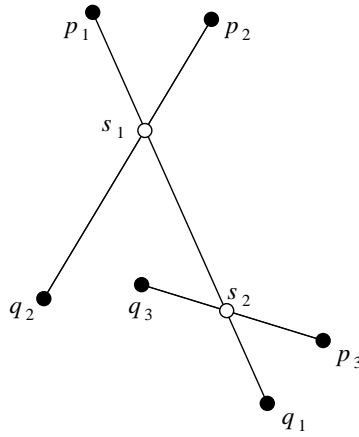


Figure 26.2: Site representation. The point s_1 is represented by the four points p_1 , q_1 , p_2 and q_2 . The segment p_1s_1 is represented by the points p_1 , q_1 , p_2 , q_2 and a boolean which is set to *true* to indicate that the first endpoint is not a point of intersection. The segment s_1s_2 is represented by the six points: p_1 , q_1 , p_2 , q_2 , p_3 and q_3 . The remaining (non-input) points and segments in the figure are represented similarly.

the representation of the (open) segment site as well as the representation of the point site itself. The situation becomes even worse in the presence of strongly intersecting sites: a point may appear in the representation of multiple subsegments and/or points of intersection. To avoid this redundancy, input points are stored in a container, and the various types of sites (input points and segments, points of intersection, subsegments with one or two points of intersection as endpoints) only store handles to the points in the container. This is achieved by the *Segment_Delaunay_graph_storage_site_2<Gt>* class which is a model of the corresponding concept: *SegmentDelaunayGraphStorageSite_2*. This concept enforces a site to be represented by up to 6 handles (which are very lightweight objects) instead of 6 points, which are, compared to handles of course, very heavy objects.

Optimizing Memory Allocation. There are applications where we know beforehand that the input consists of only weakly intersecting sites. In these cases the site representation described above poses a significant overhead in the memory requirements of our implementation: instead of representing sites with up to two points (or ultimately with up to two handles), we require sites to store six points (respectively, six handles). To avoid this overhead we have introduced two series of traits classes:

- One that supports the full-fledged sites, and is suitable when the input consists of strongly intersecting sites. This series consists of the *Segment_Delaunay_graph_traits_2<K,MTag>* and *Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>* classes.
- One that is customized for input that contain only weakly intersecting sites. This series consists of the *Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>* and *Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>* classes.

The advantages of having different traits classes are as follows:

- When the user chooses to use one of the traits classes in the second series we only store two handles per site. This implies a reduction by a factor of three in the memory allocated per site with respect to the first series of traits classes.
- In the case of the first series of traits classes, we can better exploit the knowledge that have strongly intersecting sites, in order to further apply geometric filters (see below) during the evaluation of the

predicates. On the contrary, if the second series of traits classes is used, we can avoid geometric filtering tests that have meaning only in the case of strongly intersecting sites.

26.3 The Geometric Traits

The predicates required for the computation of the segment Voronoi diagram are rather complicated. It is not the purpose of this document to discuss them in detail. The interested reader may refer to Burnikel's thesis [Bur96], where it is shown that in the case of weakly intersecting sites represented in homogeneous coordinates of bit size b , the maximum bit size of the algebraic expressions involved in the predicates is $40b + O(1)$. Given our site representation given above we can guarantee that even in the case of strongly intersecting sites, the algebraic degree of the predicates remains $O(b)$, independently of the size of the input. What we want to focus in the remainder of this section are the different kinds of filtering techniques that we have employed in our implementation.

Geometric Filtering. Our representation of sites is coupled very naturally, with what we call *geometric filtering*. The technique amounts to performing simple geometric tests exploiting the representation of our data, as well as the geometric structure inherent in our problem, in order to evaluate predicates in seemingly degenerate configurations. Geometric filtering can be seen as a preprocessing step before performing arithmetic filtering. Roughly speaking, by arithmetic filtering we mean that we first try to evaluate the predicates using a fixed-precision floating-point number type (such as `double`), and at the same time keep error bounds on the numerical errors of the computations we perform. If the numerical errors are too big and do not permit us to evaluate the predicate, we switch to an exact number type, and repeat the evaluation of the predicate. Geometric filtering can help by eliminating situations in which the arithmetic filter will fail, thus decreasing the number of times we need to evaluate a predicate using exact arithmetic.

To illustrate the application and effectiveness of this approach, let us consider a very simple example usage. Suppose we want to determine if two non-input points are identical (we assume here that the input sites are represented by *doubles*). In order to do that we need to compute their coordinates and compare them. If the two points are identical, the answer to our question using *double* arithmetic may be wrong (due to numerical errors), in which case we will have to reside to the more expensive exact computation. Instead, before testing the coordinates for equality, we can use the representation of the points to potentially answer the question. More specifically, and this is the geometric filtering part of the computation, we can first test if the defining segments of the two points are the same. If they are not, then we proceed to comparing their coordinates as usual. Testing the defining segments for equality does not involve any arithmetic operations on the input, but rather only comparisons on *doubles*. By performing this very simple test we avoid a numerically difficult computation, which could be performed thousands of times during the computation of a Delaunay graph.

Geometric filtering has been implemented in all our models of the *SegmentDelaunayGraphTraits_2* concept. These models are the classes: *Segment_Delaunay_graph_traits_2*< $K, MTag$ >, *Segment_Delaunay_graph_traits_without_intersections_2*< $K, MTag$ >, *Segment_Delaunay_graph_filtered_traits_2*< CK, CM, EK, EM, FK, FM > and *Segment_Delaunay_graph_filtered_traits_without_intersections_2*< CK, CM, EK, EM, FK, FM >.

Arithmetic Filtering. As mentioned above, performing computations with exact arithmetic can be very costly. For this reason we have devoted considerable effort in implementing different kinds of arithmetic filtering mechanisms. Presently, there two ways of performing arithmetic filtering for the predicates involved in the computation of segment Delaunay graphs:

1. The user can define his/her kernel using as number type, a number type of the form *CGAL::Filtered_exact*< CT, ET >. Then this kernel can be entered as the first template parameter in the *Segment_Delaunay_graph_2*< $K, MTag$ > or *Segment_Delaunay_graph_with_intersections_2*< $K, MTag$ > class.

2. The user can define up to three different kernels *CK*, *FK* and *EK* (default values are provided for most parameters). The first kernel *CK* is used only for constructions. The second kernel *FK* is the filtering kernel: the traits class will attempt to compute the predicates using this kernel. If the filtering kernel fails to successfully compute a predicate, the exact kernel *EK* will be used. These three kernels are then used in the *Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>* and *Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>* classes, which have been implemented using the *Filtered_predicate<EP,FP>* mechanism.

Our experience so far has shown that for all reasonable and valid values of the template parameters, the second method for arithmetic filtering is more efficient among the two.

Let's consider once more the classes *Segment_Delaunay_graph_2<K,MTag>* and *Segment_Delaunay_graph_with_intersections_2<K,MTag>*. The template parameter *MTag* provides another degree of freedom to the user, who can indicate the type of arithmetic operations to be used in the evaluation of the predicates. More specifically, in both classes, *MTag* can be *CGAL::Sqrt_field_tag*, in which case the predicates will be evaluated using all four basic arithmetic operations plus square roots; this requires, of course, that the number type used in the kernel *K* supports these operations exactly. The second choices are *CGAL::Field_tag* for the *Segment_Delaunay_graph_2<K,MTag>* class, and *CGAL::Ring_tag* for the *Segment_Delaunay_graph_with_intersections_2<K,MTag>* class. In the first case we indicate that we want the predicates to be computed using only the four basic arithmetic operations, whereas in the second case we evaluate the predicates using only ring operations. Again, for the predicates to be evaluated correctly, the number type used in the kernel *K* must support the corresponding operations exactly.

The semantics for the template parameters *CM*, *FM* and *EM* in the *Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>* and *Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>* classes are analogous. With each of these template parameters we can control the type of arithmetic operations that are going to be used in calculations involving each of the corresponding kernels *CK*, *FK* and *EK*. When the *Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>* is used the possible values for *CM*, *FM* and *EM* are *CGAL::Sqrt_field_tag* and *CGAL::Field_tag*, whereas if the *Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>* class is used, the possible values are *CGAL::Sqrt_field_tag* and *CGAL::Ring_tag*. The semantics are the same as in the case of the *Segment_Delaunay_graph_2<K,MTag>* and *Segment_Delaunay_graph_with_intersections_2<K,MTag>* classes.

26.4 The Segment Delaunay Graph Hierarchy

The *Segment_Delaunay_graph_hierarchy_2<SegmentDelaunayGraphTraits_2, SSTag, SegmentDelaunayGraphDataStructure_2>* class is the analogue of the *Triangulation_hierarchy_2* or the *Apollonius_graph_hierarchy_2* classes, applied to the segment Delaunay graph. It consists of a hierarchy of segment Delaunay graphs constructed in a manner analogous to the Delaunay hierarchy by Devillers [Dev02]. Unlike the triangulation hierarchy or the Apollonius graph hierarchy, the situation here is more complicated because of two factors: firstly, segments are treated as three objects instead of one (the two endpoints and the interior of the segments), and secondly, the presence of strongly intersecting sites complicates significantly the way the hierarchy is constructed. The interested reader may refer to the paper by Karavelas [Kar04] for the details of the construction of the hierarchy. Another alternative is to have a hybrid hierarchy that consists of the segment Delaunay graph at the bottom-most level and point Voronoi diagrams at all other levels. This choice seems to work very well in practice, primarily because it avoids the overhead of maintaining a Delaunay graph for segments at the upper levels of the hierarchy. However, it seems much less likely to be possible to give any theoretical guarantees for its performance, in contrast to the hierarchy with segment Delaunay graphs at all levels (cf. [Kar04]). The user can choose between the two types of hierarchies by means of the template parameter *SSTag*. If *SSTag* is set to *false* (which is also the default value), the upper levels of the hierarchy consist of point Delaunay graphs. If *SSTag* is set to *true*, we have segment Delaunay graphs at all levels of the hierarchy.

The class `Segment_Delaunay_graph_hierarchy_2<SegmentDelaunayGraphTraits_2, SSTag, SegmentDelaunayGraphDataStructure_2>` has exactly the same interface and functionality as the `Segment_Delaunay_graph_2<SegmentDelaunayGraphTraits_2, SegmentDelaunayGraphDataStructure_2>` class. Using the segment Delaunay graph hierarchy involves an additional cost in space and time for maintaining the hierarchy. Our experiments have shown that it usually pays off to use the hierarchy for inputs consisting of more than about 1,000 sites.

26.5 Examples

26.5.1 First Example

The following example shows to use the segment Delaunay graph traits in conjunction with the *Filtered_exact<CT,ET>* mechanism. In addition it shows how to use a few of the iterators provided by the `Segment_Delaunay_graph_2` class in order to count a few site-related quantities.

```
// file: sdg-count-sites.C
#include <CGAL/basic.h>

// standard includes
#include <iostream>
#include <fstream>
#include <cassert>

// define the exact number type
# include <CGAL/Quotient.h>
# include <CGAL/MP_Float.h>
typedef CGAL::Quotient<CGAL::MP_Float> ENT;

// define the kernels
#include <CGAL/Simple_cartesian.h>

typedef CGAL::Simple_cartesian<double> CK;
typedef CGAL::Simple_cartesian<ENT> EK;

// typedefs for the traits and the algorithm
#include <CGAL/Segment_Delaunay_graph_filtered_traits_2.h>
#include <CGAL/Segment_Delaunay_graph_2.h>

typedef CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,
/* The construction kernel allows for / and sqrt */ CGAL::Sqrt_field_tag,
EK,
/* The exact kernel supports field ops exactly */ CGAL::Field_tag> Gt;

typedef CGAL::Segment_Delaunay_graph_2<Gt> SDG2;

using namespace std;

int main() {
    ifstream ifs("data/sitesx.cin");
    assert( ifs );
```

```

SDG2          sdg;
SDG2::Site_2  site;

while ( ifs >> site ) { sdg.insert( site ); }

ifs.close();

assert( sdg.is_valid(true, 1) );
cout << endl << endl;

// print the number of input and output sites
cout << "# of input sites : " << sdg.number_of_input_sites() << endl;
cout << "# of output sites: " << sdg.number_of_output_sites() << endl;

unsigned int n_ipt(0), n_iseg(0), n_opt(0), n_oseg(0), n_ptx(0);

// count the number of input points and input segments
SDG2::Input_sites_iterator iit;
for (iit = sdg.input_sites_begin(); iit != sdg.input_sites_end(); ++iit)
{
    if ( iit->is_point() ) { n_ipt++; } else { n_iseg++; }
}

// count the number of output points and output segments, as well
// as the number of points that are points of intersection of pairs
// of strongly intersecting sites
SDG2::Output_sites_iterator oit;
for (oit = sdg.output_sites_begin(); oit != sdg.output_sites_end(); ++oit)
{
    if ( oit->is_segment() ) { n_oseg++; } else {
        n_opt++;
        if ( !oit->is_input() ) { n_ptx++; }
    }
}

cout << endl << "# of input segments: " << n_iseg << endl;
cout << "# of input points:      " << n_ipt << endl << endl;
cout << "# of output segments: " << n_oseg << endl;
cout << "# of output points:     " << n_opt << endl << endl;
cout << "# of intersection points: " << n_ptx << endl;

return 0;
}

```

26.5.2 Second Example

The following example shows how to use the segment Delaunay graph hierarchy along with the filtered traits class that supports intersecting sites.

```

// file: sdg-filtered-traits.C
#include <CGAL/basic.h>

```

```

// standard includes
#include <iostream>
#include <fstream>
#include <cassert>

// example that uses the filtered traits and
// the segment Delaunay graph hierarchy

// choose the kernel
#include <CGAL/Simple_cartesian.h>

struct Rep : public CGAL::Simple_cartesian<double> {};

// typedefs for the traits and the algorithm
#include <CGAL/Segment_Delaunay_graph_hierarchy_2.h>
#include <CGAL/Segment_Delaunay_graph_filtered_traits_2.h>

struct Gt
    : public CGAL::Segment_Delaunay_graph_filtered_traits_2<Rep> {};

typedef CGAL::Segment_Delaunay_graph_hierarchy_2<Gt>   SDG2;

int main()
{
    std::ifstream ifs("data/sites.cin");
    assert( ifs );

    SDG2          sdg;
    SDG2::Site_2  site;

    // read the sites and insert them in the segment Delaunay graph
    while ( ifs >> site ) {
        sdg.insert(site);
    }

    // validate the segment Delaunay graph
    assert( sdg.is_valid(true, 1) );

    return 0;
}

```

26.5.3 Third Example

The following example demonstrates how to recover the defining sites for the edges of the Voronoi diagram (which are the duals of the edges of the segment Delaunay graph computed).

```

// file: sdg-voronoi-vertices.C
#include <CGAL/basic.h>

```



```

// standard includes
#include <iostream>
#include <fstream>
#include <cassert>
#include <string>

// define the kernel
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Filtered_kernel.h>

typedef CGAL::Simple_cartesian<double>    CK;
typedef CGAL::Filtered_kernel<CK>        Kernel;

// typedefs for the traits and the algorithm
#include <CGAL/Segment_Delaunay_graph_traits_2.h>
#include <CGAL/Segment_Delaunay_graph_2.h>

typedef CGAL::Segment_Delaunay_graph_traits_2<Kernel>  Gt;
typedef CGAL::Segment_Delaunay_graph_2<Gt>             SDG2;

using namespace std;

int main()
{
    ifstream ifs("data/sites2.cin");
    assert( ifs );

    SDG2      sdg;
    SDG2::Site_2  site;

    // read the sites from the stream and insert them in the diagram
    while ( ifs >> site ) { sdg.insert( site ); }

    ifs.close();

    // validate the diagram
    assert( sdg.is_valid(true, 1) );
    cout << endl << endl;

    /*
    // now walk through the non-infinite edges of the segment Delaunay
    // graphs (which are dual to the edges in the Voronoi diagram) and
    // print the sites defining each Voronoi edge.
    //
    // Each oriented Voronoi edge (horizontal segment in the figure
    // below) is defined by four sites A, B, C and D.
    //
    //      \           /
    //      \         B /
    //      \       /   /
    //      C ----- D
    //      /       \ \
    //      /         A \
    //      /           \
    */

```

```

//
// The sites A and B define the (oriented) bisector on which the
// edge lies whereas the sites C and D, along with A and B define
// the two endpoints of the edge. These endpoints are the Voronoi
// vertices of the triples A, B, C and B, A, D.
// If one of these vertices is the vertex at infinity the string
// "infinite vertex" is printed; the corresponding Voronoi edge is
// actually a straight-line or parabolic ray.
// The sites below are printed in the order A, B, C, D.
*/

string inf_vertex("infinite vertex");
char vid[] = {'A', 'B', 'C', 'D'};

SDG2::Finite_edges_iterator eit = sdg.finite_edges_begin();
for (int k = 1; eit != sdg.finite_edges_end(); ++eit, ++k) {
    SDG2::Edge e = *eit;
    // get the vertices defining the Voronoi edge
    SDG2::Vertex_handle v[] = { e.first->vertex( sdg.ccw(e.second) ),
                                e.first->vertex( sdg.cw(e.second) ),
                                e.first->vertex( e.second ),
                                sdg.tds().mirror_vertex(e.first, e.second) };

    cout << "--- Edge " << k << " ---" << endl;
    for (int i = 0; i < 4; i++) {
        // check if the vertex is the vertex at infinity; if yes, print
        // the corresponding string, otherwise print the site
        if ( sdg.is_infinite(v[i]) ) {
            cout << vid[i] << ": " << inf_vertex << endl;
        } else {
            cout << vid[i] << ": " << v[i]->site() << endl;
        }
    }
    cout << endl;
}

return 0;
}

```

2D Segment Delaunay Graphs

Reference Manual

Menelaos Karavelas

CGAL provides the class `CGAL::Segment_Delaunay_graph_2<Gt,DS>` for computing the 2D Delaunay graph of segments and points. The two template parameters must be models of the `SegmentDelaunayGraphTraits_2` and `SegmentDelaunayGraphDataStructure_2` concepts. The first concept is related to the geometric objects and predicates associated with segment Delaunay graphs, whereas the second concept refers to the data structure used to represent the segment Delaunay graph, which is dual to the 2D Voronoi diagram of segments and points. The classes `Segment_Delaunay_graph_traits_2<K,MTag>`, `Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>`, `Segment_Delaunay_graph_filtered_traits_2<K,MTag>`, `Segment_Delaunay_graph_filtered_traits_without_intersections_2<K,MTag>` are models of the `SegmentDelaunayGraphTraits_2` concept, whereas the class `Triangulation_data_structure_2<Vb,Fb>` is a model of the `SegmentDelaunayGraphDataStructure_2` concept.

26.6 Classified Reference Pages

Concepts

<code>SegmentDelaunayGraphSite_2</code>	page 1669
<code>SegmentDelaunayGraphStorageSite_2</code>	page 1673
<code>SegmentDelaunayGraphDataStructure_2</code>	page 1678
<code>SegmentDelaunayGraphVertexBase_2</code>	page 1680
<code>SegmentDelaunayGraphTraits_2</code>	page 1683
<code>SegmentDelaunayGraphHierarchyVertexBase_2</code>	page 1696

Classes

<code>CGAL::Segment_Delaunay_graph_2<Gt,DS></code>	page 1661
<code>CGAL::Segment_Delaunay_graph_site_2<K></code>	page 1672
<code>CGAL::Segment_Delaunay_graph_storage_site_2<Gt></code>	page 1677
<code>CGAL::Segment_Delaunay_graph_vertex_base_2<Gt,SSTag></code>	page 1682
<code>CGAL::Segment_Delaunay_graph_traits_2<K,MTag></code>	page 1688
<code>CGAL::Segment_Delaunay_graph_traits_without_intersections_2<K,MTag></code>	page 1689
<code>CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM></code>	page 1690
<code>CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM></code>	page 1692
<code>CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,SSTag,DS></code>	page 1694
<code>CGAL::Segment_Delaunay_graph_hierarchy_vertex_base_2<Vbb></code>	page 1697

26.7 Alphabetical List of Reference Pages

<i>SegmentDelaunayGraphDataStructure_2</i>	page 1678
<i>SegmentDelaunayGraphHierarchyVertexBase_2</i>	page 1696
<i>SegmentDelaunayGraphSite_2</i>	page 1669
<i>SegmentDelaunayGraphStorageSite_2</i>	page 1673
<i>SegmentDelaunayGraphTraits_2</i>	page 1683
<i>SegmentDelaunayGraphVertexBase_2</i>	page 1680
<i>Segment_Delaunay_graph_2<Gt,DS></i>	page 1661
<i>Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM></i>	page 1690
<i>Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM></i>	page 1692
<i>Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS></i>	page 1694
<i>Segment_Delaunay_graph_hierarchy_vertex_base_2<Vbb></i>	page 1697
<i>Segment_Delaunay_graph_site_2<K></i>	page 1672
<i>Segment_Delaunay_graph_storage_site_2<Gt></i>	page 1677
<i>Segment_Delaunay_graph_traits_2<K,MTag></i>	page 1688
<i>Segment_Delaunay_graph_traits_without_intersections_2<K,MTag></i>	page 1689
<i>Segment_Delaunay_graph_vertex_base_2<Gt,SSTag></i>	page 1682

CGAL::Segment_Delaunay_graph_2<Gt,DS>

Definition

The class *Segment_Delaunay_graph_2*<*Gt*,*DS*> represents the segment Delaunay graph (which is the dual graph of the 2D segment Voronoi diagram). Currently it supports only insertions of sites. It is templated by two template arguments *Gt*, which must be a model of *SegmentDelaunayGraphTraits_2* and *DS*, which must be a model of *SegmentDelaunayGraphDataStructure_2*. The second template argument defaults to *CGAL::Triangulation_data_structure_2*< *CGAL::Segment_Delaunay_graph_vertex_base_2*<*Gt*>, *CGAL::Triangulation_face_base_2*<*Gt*> >.

```
#include <CGAL/Segment_Delaunay_graph_2.h>
```

Is Model for the Concepts

DelaunayGraph_2

Types

<code>typedef Gt</code>	<code>Geom_traits;</code>	A type for the geometric traits.
<code>typedef DS</code>	<code>Data_structure;</code>	A type for the underlying data structure.
<code>typedef Data_structure</code>	<code>Triangulation_data_structure;</code>	
		This type has been added so that the <i>Segment_Delaunay_graph_2</i> < <i>Gt</i> , <i>DS</i> > class is a model of the <i>DelaunayGraph_2</i> concept.
<code>typedef typename DS::size_type</code>	<code>size_type;</code>	Size type (an unsigned integral type)
<code>typedef typename Gt::Point_2</code>	<code>Point_2;</code>	A type for the point defined in the geometric traits.
<code>typedef typename Gt::Site_2</code>	<code>Site_2;</code>	A type for the segment Delaunay graph site, defined in the geometric traits.
<code>Segment_Delaunay_graph_2<Gt,DS>:: Point_container;</code>		A type for the container of points.
<code>typedef typename Point_container::iterator</code>	<code>Point_handle;</code>	A handle for points in the point container.

The vertices and faces of the segment Delaunay graph are accessed through *handles*, *iterators* and *circulators*. The iterators and circulators are all bidirectional and non-mutable. The circulators and iterators are assignable to the corresponding handle types, and they are also convertible to the corresponding handles. The edges of the segment Delaunay graph can also be visited through iterators and circulators, the edge circulators and iterators are also bidirectional and non-mutable. In the following, we call *infinite* any face or edge incident to the infinite vertex and the infinite vertex itself. Any other feature (face, edge or vertex) of the segment Delaunay graph is said to be *finite*. Some iterators (the *All* iterators) allow to visit finite or infinite features while the others (the *Finite* iterators) visit only finite features. Circulators visit both infinite and finite features.

<code>typedef typename DS::Edge</code>	<code>Edge;</code>	The edge type. The <i>Edge(f,i)</i> is the edge common to faces <i>f</i> and <i>f.neighbor(i)</i> . It is also the edge joining the vertices <i>vertex(cw(i))</i> and <i>vertex(ccw(i))</i> of <i>f</i> . <i>Precondition:</i> <i>i</i> must be 0, 1 or 2.
----------------------------------------	--------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<code>typedef typename DS::Vertex</code>	<code>Vertex;</code>	A type for a vertex.
<code>typedef typename DS::Face</code>	<code>Face;</code>	A type for a face.
<code>typedef typename DS::Vertex_handle</code>	<code>Vertex_handle;</code>	A type for a handle to a vertex.
<code>typedef typename DS::Face_handle</code>	<code>Face_handle;</code>	A type for a handle to a face.
<code>typedef typename DS::Vertex_circulator</code>	<code>Vertex_circulator;</code>	A type for a circulator over vertices incident to a given vertex.
<code>typedef typename DS::Face_circulator</code>	<code>Face_circulator;</code>	A type for a circulator over faces incident to a given vertex.
<code>typedef typename DS::Edge_circulator</code>	<code>Edge_circulator;</code>	A type for a circulator over edges incident to a given vertex.
<code>typedef typename DS::Vertex_iterator</code>	<code>All_vertices_iterator;</code>	A type for an iterator over all vertices.
<code>typedef typename DS::Face_iterator</code>	<code>All_faces_iterator;</code>	A type for an iterator over all faces.
<code>typedef typename DS::Edge_iterator</code>	<code>All_edges_iterator;</code>	A type for an iterator over all edges.
<code>Segment_Delaunay_graph_2<Gt,DS>::Finite_vertices_iterator;</code>		A type for an iterator over finite vertices.
<code>Segment_Delaunay_graph_2<Gt,DS>::Finite_faces_iterator;</code>		A type for an iterator over finite faces.
<code>Segment_Delaunay_graph_2<Gt,DS>::Finite_edges_iterator;</code>		A type for an iterator over finite edges.

In addition to iterators and circulators for vertices and faces, iterators for sites are provided. In particular there are iterators for the set of input sites and the set of output sites. The set of input sites is the set of sites inserted by the user using the *insert* methods of this class. If a site is inserted multiple times, every instance of this site will be reported. The set of output sites is the set of sites in the segment Delaunay graph. The value type of these iterators is *Site_2*.

<code>Segment_Delaunay_graph_2<Gt,DS>::Input_sites_iterator</code>	A type for a bidirectional iterator over all input sites.
<code>Segment_Delaunay_graph_2<Gt,DS>::Output_sites_iterator</code>	A type for a bidirectional iterator over all output sites (the sites in the Delaunay graph).

Creation

In addition to the default and copy constructors the following constructors are defined:

<code>Segment_Delaunay_graph_2<Gt,DS> sdg(Gt gt=Gt());</code>	Creates the segment Delaunay graph using <i>gt</i> as geometric traits.
----------------------------------------------------------------------	-------------------------------------------------------------------------

`template< class Input_iterator >`

<code>Segment_Delaunay_graph_2<Gt,DS> sdg(Input_iterator first, Input_iterator beyond, Gt gt=Gt());</code>	Creates the segment Delaunay graph using <i>gt</i> as geometric traits and inserts all sites in the range [<i>first</i> , <i>beyond</i>).
-------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------

Creates the segment Delaunay graph using *gt* as geometric traits and inserts all sites in the range [*first*, *beyond*).

Precondition: *Input_iterator* must be a model of *InputIterator*. The value type of *Input_iterator* must be either *Point_2* or *Site_2*.

Access Functions

<code>Geom_traits</code>	<code>sdg.geom_traits()</code>	Returns a reference to the segment Delaunay graph traits object.
--------------------------	--------------------------------	------------------------------------------------------------------

<i>int</i>	<i>sdg.dimension()</i>	Returns the dimension of the segment Delaunay graph. The dimension is -1 if the graph contains no sites, 0 if the graph contains one site, 1 if it contains two sites and 2 if it contains three or more sites.
<i>size_type</i>	<i>sdg.number_of_vertices()</i>	Returns the number of finite vertices of the segment Delaunay graph.
<i>size_type</i>	<i>sdg.number_of_faces()</i>	Returns the number of faces (both finite and infinite) of the segment Delaunay graph.
<i>size_type</i>	<i>sdg.number_of_input_sites()</i>	Return the number of input sites.
<i>size_type</i>	<i>sdg.number_of_output_sites()</i>	Return the number of output sites. This is equal to the number of vertices in the segment Delaunay graph.
<i>Face_handle</i>	<i>sdg.infinite_face()</i>	Returns a face incident to the <i>infinite_vertex</i> .
<i>Vertex_handle</i>	<i>sdg.infinite_vertex()</i>	Returns the <i>infinite_vertex</i> .
<i>Vertex_handle</i>	<i>sdg.finite_vertex()</i>	Returns a vertex distinct from the <i>infinite_vertex</i> . <i>Precondition:</i> The number of sites in the segment Delaunay graph must be at least one.
<i>Data_structure</i>	<i>sdg.data_structure()</i>	Returns a reference to the segment Delaunay graph data structure object.
<i>Data_structure</i>	<i>sdg.tds()</i>	Same as <i>data_structure()</i> . It has been added for compliance to the <i>DelaunayGraph_2</i> concept.
<i>Point_container</i>	<i>sdg.point_container()</i>	Returns a reference to the point container object.

Traversal of the segment Delaunay graph

A segment Delaunay graph can be seen as a container of faces and vertices. Therefore the *Segment_Delaunay_graph_2*<Gt,DS> class provides several iterators and circulators that allow to traverse it (completely or partially).

Face, Edge and Vertex Iterators

The following iterators allow respectively to visit finite faces, finite edges and finite vertices of the segment Delaunay graph. These iterators are non-mutable, bidirectional and their value types are respectively *Face*, *Edge* and *Vertex*. They are all invalidated by any change in the segment Delaunay graph.

<i>Finite_vertices_iterator</i>	<i>sdg.finite_vertices_begin()</i>	Starts at an arbitrary finite vertex.
<i>Finite_vertices_iterator</i>	<i>sdg.finite_vertices_end()</i>	Past-the-end iterator.
<i>Finite_edges_iterator</i>	<i>sdg.finite_edges_begin()</i>	Starts at an arbitrary finite edge.
<i>Finite_edges_iterator</i>	<i>sdg.finite_edges_end()</i>	Past-the-end iterator.
<i>Finite_faces_iterator</i>	<i>sdg.finite_faces_begin()</i>	Starts at an arbitrary finite face.
<i>Finite_faces_iterator</i>	<i>sdg.finite_faces_end()</i>	Past-the-end iterator.

The following iterators allow respectively to visit all (both finite and infinite) faces, edges and vertices of the segment Delaunay graph. These iterators are non-mutable, bidirectional and their value types are respectively *Face*, *Edge* and *Vertex*. They are all invalidated by any change in the segment Delaunay graph.

<i>All_vertices_iterator</i>	<i>sdg.all_vertices_begin()</i>	Starts at an arbitrary vertex.
<i>All_vertices_iterator</i>	<i>sdg.all_vertices_end()</i>	Past-the-end iterator.
<i>All_edges_iterator</i>	<i>sdg.all_edges_begin()</i>	Starts at an arbitrary edge.

<i>All_edges_iterator</i>	<i>sdg.all_edges_end()</i>	Past-the-end iterator.
<i>All_faces_iterator</i>	<i>sdg.all_faces_begin()</i>	Starts at an arbitrary face.
<i>All_faces_iterator</i>	<i>sdg.all_faces_end()</i>	Past-the-end iterator.

Site iterators

The following iterators allow respectively to visit all sites. These iterators are non-mutable, bidirectional and their value type is *Site_2*. They are all invalidated by any change in the segment Delaunay graph.

<i>Input_sites_iterator</i>	<i>sdg.input_sites_begin()</i>	Starts at an arbitrary input site.
<i>Input_sites_iterator</i>	<i>sdg.input_sites_end()</i>	Past-the-end iterator.
<i>Output_sites_iterator</i>	<i>sdg.output_sites_begin()</i>	Starts at an arbitrary output site.
<i>Output_sites_iterator</i>	<i>sdg.output_sites_end()</i>	Past-the-end iterator.

Face, Edge and Vertex Circulators

The *Segment_Delaunay_graph_2*<*Gt,DS*> class also provides circulators that allow to visit respectively all faces or edges incident to a given vertex or all vertices adjacent to a given vertex. These circulators are non-mutable and bidirectional. The operator *operator++* moves the circulator counterclockwise around the vertex while the *operator--* moves clockwise. A face circulator is invalidated by any modification of the face pointed to. An edge circulator is invalidated by any modification in one of the two faces incident to the edge pointed to. A vertex circulator is invalidated by any modification in any of the faces adjacent to the vertex pointed to.

<i>Face_circulator</i>	<i>sdg.incident_faces(Vertex_handle v)</i>	Starts at an arbitrary face incident to <i>v</i> .
<i>Face_circulator</i>	<i>sdg.incident_faces(Vertex_handle v, Face_handle f)</i>	Starts at face <i>f</i> . <i>Precondition</i> : Face <i>f</i> is incident to vertex <i>v</i> .
<i>Edge_circulator</i>	<i>sdg.incident_edges(Vertex_handle v)</i>	Starts at an arbitrary edge incident to <i>v</i> .
<i>Edge_circulator</i>	<i>sdg.incident_edges(Vertex_handle v, Face_handle f)</i>	Starts at the first edge of <i>f</i> incident to <i>v</i> , in counterclockwise order around <i>v</i> . <i>Precondition</i> : Face <i>f</i> is incident to vertex <i>v</i> .
<i>Vertex_circulator</i>	<i>sdg.incident_vertices(Vertex_handle v)</i>	Starts at an arbitrary vertex incident to <i>v</i> .
<i>Vertex_circulator</i>	<i>sdg.incident_vertices(Vertex_handle v, Face_handle f)</i>	Starts at the first vertex of <i>f</i> adjacent to <i>v</i> in counterclockwise order around <i>v</i> . <i>Precondition</i> : Face <i>f</i> is incident to vertex <i>v</i> .

Traversal of the Convex Hull

Applied on the *infinite_vertex* the above methods allow to visit the vertices on the convex hull and the infinite edges and faces. Note that a counterclockwise traversal of the vertices adjacent to the *infinite_vertex* is a clockwise traversal of the convex hull.

```
Vertex_circulator    sdg.incident_vertices( sdg.infinite_vertex())
Vertex_circulator    sdg.incident_vertices( sdg.infinite_vertex(), Face_handle f)
Face_circulator      sdg.incident_faces( sdg.infinite_vertex())
Face_circulator      sdg.incident_faces( sdg.infinite_vertex(), Face_handle f)
Edge_circulator      sdg.incident_edges( sdg.infinite_vertex())
Edge_circulator      sdg.incident_edges( sdg.infinite_vertex(), Face_handle f)
```

Predicates

The class *Segment_Delaunay_graph_2*<*Gt,DS*> provides methods to test the finite or infinite character of any feature.

```
bool    sdg.is_infinite( Vertex_handle v)      true, iff v is the infinite_vertex.
bool    sdg.is_infinite( Face_handle f)        true, iff face f is infinite.
bool    sdg.is_infinite( Face_handle f, int i)  true, iff edge (f,i) is infinite.
bool    sdg.is_infinite( Edge e)               true, iff edge e is infinite.
bool    sdg.is_infinite( Edge_circulator ec)    true, iff edge *ec is infinite.
```

Insertion

```
template< class Input_iterator >
size_type    sdg.insert( Input_iterator first, Input_iterator beyond)
```

Inserts the sites in the range [*first,beyond*). The number of additional sites inserted in the Delaunay graph is returned. *Input_iterator* must be a model of *InputIterator* and its value type must be either *Point_2* or *Site_2*.

```
template< class Input_iterator >
size_type    sdg.insert( Input_iterator first, Input_iterator beyond, Tag_false)
```

Same as the previous method. *Input_iterator* must be a model of *InputIterator* and its value type must be either *Point_2* or *Site_2*.

```
template< class Input_iterator >
size_type    sdg.insert( Input_iterator first, Input_iterator beyond, Tag_true)
```

Inserts the sites in the range [*first,beyond*) after performing a random shuffle on them. The number of additional sites inserted in the Delaunay graph is returned. *Input_iterator* must be a model of *InputIterator* and its value type must be either *Point_2* or *Site_2*.

```
Vertex_handle    sdg.insert( Point_2 p)
```

Inserts the point *p* in the segment Delaunay graph. If *p* has already been inserted, then the vertex handle of its already inserted copy is returned. If *p* has not been inserted yet, the vertex handle of *p* is returned.

Vertex_handle *sdg.insert(Point_2 p, Vertex_handle vnear)*

Inserts p in the segment Delaunay graph using the site associated with $vnear$ as an estimate for the nearest neighbor of p . The vertex handle returned has the same semantics as the vertex handle returned by the method *Vertex_handle insert(Point_2 p)*.

Vertex_handle *sdg.insert(Point_2 p1, Point_2 p2)*

Inserts the closed segment with endpoints $p1$ and $p2$ in the segment Delaunay graph. If the segment has already been inserted in the Delaunay graph then the vertex handle of its already inserted copy is returned. If the segment does not intersect any segment in the existing diagram, the vertex handle corresponding to its corresponding open segment is returned. Finally, if the segment intersects other segments in the existing Delaunay graph, the vertex handle to one of its open subsegments is returned.

Vertex_handle *sdg.insert(Point_2 p1, Point_2 p2, Vertex_handle vnear)*

Inserts the segment whose endpoints are $p1$ and $p2$ in the segment Delaunay graph using the site associated with $vnear$ as an estimate for the nearest neighbor of $p1$. The vertex handle returned has the same semantics as the vertex handle returned by the method *Vertex_handle insert(Point_2 p1, Point_2 p2)*.

Vertex_handle *sdg.insert(Site_2 s)*

Inserts the site s in the segment Delaunay graph. The vertex handle returned has the same semantics as the vertex handle returned by the methods *Vertex_handle insert(Point_2 p)* and *Vertex_handle insert(Point_2 p1, Point_2 p2)*, depending on whether s represents a point or a segment respectively.

Precondition: s.is_input() must be true.

Vertex_handle *sdg.insert(Site_2 s, Vertex_handle vnear)*

Inserts s in the segment Delaunay graph using the site associated with $vnear$ as an estimate for the nearest neighbor of s , if s is a point, or the first endpoint of s , if s is a segment. The vertex handle returned has the same semantics as the vertex handle returned by the method *Vertex_handle insert(Site_2 s)*.

Precondition: s.is_input() must be true.

Nearest neighbor location

Vertex_handle *sdg.nearest_neighbor(Point_2 p)*

Finds the nearest neighbor of the point p . In other words it finds the site whose segment Voronoi diagram cell contains p . Ties are broken arbitrarily and one of the nearest neighbors of p is returned. If there are no sites in the segment Delaunay graph *Vertex_handle()* is returned.

Vertex_handle *sdg.nearest_neighbor(Point_2 p, Vertex_handle vnear)*

Finds the nearest neighbor of the point p using the site associated with $vnear$ as an estimate for the nearest neighbor of p . Ties are broken arbitrarily and one of the nearest neighbors of p is returned. If there are no sites in the segment Delaunay graph *Vertex_handle()* is returned.

I/O

template < class Stream >
Stream& sdg.draw_dual(Stream& str) Draws the segment Voronoi diagram to the stream *str*. The following operators must be defined:

Stream& operator<<(Stream&, Gt::Segment_2)

Stream& operator<<(Stream&, Gt::Ray_2)

Stream& operator<<(Stream&, Gt::Line_2)

template < class Stream >
Stream& sdg.draw_skeleton(Stream& str)

Draws the segment Voronoi diagram to the stream *str*, except the edges of the diagram corresponding to a segment and its endpoints. The following operators must be defined:

Stream& operator<<(Stream&, Gt::Segment_2)

Stream& operator<<(Stream&, Gt::Ray_2)

Stream& operator<<(Stream&, Gt::Line_2)

template< class Stream >
Stream& sdg.draw_dual_edge(Edge e, Stream& str)

Draws the edge *e* of the segment Voronoi diagram to the stream *str*. The following operators must be defined:

Stream& operator<<(Stream&, Gt::Segment_2)

Stream& operator<<(Stream&, Gt::Ray_2)

Stream& operator<<(Stream&, Gt::Line_2)

Precondition: e must be a finite edge.

template< class Stream >
Stream& sdg.draw_dual_edge(Edge_circulator ec, Stream& str)

Draws the edge **ec* of the segment Voronoi diagram to the stream *str*. The following operators must be defined:

Stream& operator<<(Stream&, Gt::Segment_2)

Stream& operator<<(Stream&, Gt::Ray_2)

Stream& operator<<(Stream&, Gt::Line_2)

*Precondition: *ec must be a finite edge.*

template< class Stream >
Stream& sdg.draw_dual_edge(All_edges_iterator eit, Stream& str)

Draws the edge **eit* of the segment Voronoi diagram to the stream *str*. The following operators must be defined:

Stream& operator<<(Stream&, Gt::Segment_2)

Stream& operator<<(Stream&, Gt::Ray_2)

Stream& operator<<(Stream&, Gt::Line_2)

*Precondition: *eit must be a finite edge.*

template< class Stream >
Stream& sdg.draw_dual_edge(Finite_edges_iterator eit, Stream& str)

Draws the edge **eit* of the segment Voronoi diagram to the stream *str*. The following operators must be defined:

Stream& operator<<(Stream&, Gt::Segment_2)

Stream& operator<<(Stream&, Gt::Ray_2)

Stream& operator<<(Stream&, Gt::Line_2)

<code>void</code>	<code>sdg.file_output(std::ostream& os)</code>	Writes the current state of the segment Delaunay graph to an output stream. In particular, all sites in the diagram are written to the stream (represented through appropriate input sites), as well as the underlying combinatorial data structure.
<code>void</code>	<code>sdg.file_input(std::istream& is)</code>	Reads the state of the segment Delaunay graph from an input stream.
<code>std::ostream&</code>	<code>std::ostream& os << sdg</code>	Writes the current state of the segment Delaunay graph to an output stream.
<code>std::istream&</code>	<code>std::istream& is >> sdg</code>	Reads the state of the segment Delaunay graph from an input stream.

Validity check

<code>bool</code>	<code>sdg.is_valid(bool verbose = false, int level = 1)</code>	Checks the validity of the segment Delaunay graph. If <i>verbose</i> is <i>true</i> a short message is sent to <i>std::cerr</i> . If <i>level</i> is 0, only the data structure is validated. If <i>level</i> is 1, then both the data structure and the segment Delaunay graph are validated. Negative values of <i>level</i> always return true, and values greater than 1 are equivalent to <i>level</i> being 1.
-------------------	-----------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Miscellaneous

<code>void</code>	<code>sdg.clear()</code>	Clears all contents of the segment Delaunay graph.
<code>void</code>	<code>sdg.swap(other)</code>	The segment Delaunay graphs <i>other</i> and <i>sdg</i> are swapped. <i>sdg.swap(other)</i> should be preferred to <i>sdg= other</i> or to <i>sdg(other)</i> if <i>other</i> is deleted afterwards.

See Also

[DelaunayGraph_2](#)
[SegmentDelaunayGraphTraits_2](#)
[SegmentDelaunayGraphDataStructure_2](#)
[SegmentDelaunayGraphVertexBase_2](#)
[TriangulationFaceBase_2](#)
[CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>](#)
[CGAL::Segment_Delaunay_graph_traits_2<K,MTag>](#)
[CGAL::Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>](#)
[CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>](#)
[CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>](#)
[CGAL::Triangulation_data_structure_2<Vb,Fb>](#)
[CGAL::Segment_Delaunay_graph_vertex_base_2<Gt,SSTag>](#)
[CGAL::Triangulation_face_base_2<Gt>](#)

SegmentDelaunayGraphSite_2

Definition

The concept *SegmentDelaunayGraphSite_2* provides the requirements for the sites of a segment Delaunay graph.

Refines

DefaultConstructible

CopyConstructible

Assignable

Types

<i>SegmentDelaunayGraphSite_2:: Point_2</i>	The point type.
<i>SegmentDelaunayGraphSite_2:: Segment_2</i>	The segment type.
<i>SegmentDelaunayGraphSite_2:: FT</i>	The field number type.
<i>SegmentDelaunayGraphSite_2:: RT</i>	The ring number type.

Creation

In addition to the default and copy constructors the following static methods are available for constructing sites:

<i>SegmentDelaunayGraphSite_2</i>	<i>s.construct_site_2(Point_2 p)</i>	Constructs a site from a point: the site represents the point p .
<i>SegmentDelaunayGraphSite_2</i>	<i>s.construct_site_2(Point_2 p1, Point_2 p2)</i>	Constructs a site from two points: the site represents the (open) segment $(p1,p2)$.
<i>SegmentDelaunayGraphSite_2</i>	<i>s.construct_site_2(Point_2 p1, Point_2 p2, Point_2 q1, Point_2 q2)</i>	Constructs a site from four points: the site represents the point of intersection of the segments $(p1,p2)$ and $(q1,q2)$.
<i>SegmentDelaunayGraphSite_2</i>	<i>s.construct_site_2(Point_2 p1, Point_2 p2, Point_2 q1, Point_2 q2, bool b)</i>	Constructs a site from four points and a boolean: the site represents a segment. If b is <i>true</i> the endpoints are $p1$ and p_{\times} , otherwise p_{\times} and $p2$. p_{\times} is the point of intersection of the segments $(p1,p2),(q1,q2)$.
<i>SegmentDelaunayGraphSite_2</i>	<i>s.construct_site_2(Point_2 p1, Point_2 p2, Point_2 q1, Point_2 q2, Point_2 r1,</i>	

Point_2 r2)

Constructs a site from six points: the site represents the segment with endpoints the points of intersection of the pairs of segments $(p1,p2),(q1,q2)$ and $(p1,p2),(r1,r2)$.

Predicates

<i>bool</i>	<i>s.is_defined()</i>	Returns <i>true</i> if the site represents a valid point or segment.
<i>bool</i>	<i>s.is_point()</i>	Returns <i>true</i> if the site represents a point.
<i>bool</i>	<i>s.is_segment()</i>	Returns <i>true</i> if the site represents a segment.
<i>bool</i>	<i>s.is_input()</i>	Returns <i>true</i> if the site represents an input point or a segment defined by two input points. Returns <i>false</i> if it represents a point of intersection of two segments, or if it represents a segment, at least one endpoint of which is a point of intersection of two segments.
<i>bool</i>	<i>s.is_input(unsigned int i)</i>	Returns <i>true</i> if the <i>i</i> -th endpoint of the site is an input point. Returns <i>false</i> if the <i>i</i> -th endpoint of the site is the intersection of two segments. <i>Precondition:</i> <i>i</i> must be at most 1, and <i>s.is_segment()</i> must be <i>true</i> .

Access Functions

<i>Point_2</i>	<i>s.point()</i>	Returns the point represented by the site <i>s</i> . <i>Precondition:</i> <i>s.is_point()</i> must be <i>true</i> .
<i>Segment_2</i>	<i>s.segment()</i>	Returns the segment represented by the site <i>s</i> . <i>Precondition:</i> <i>s.is_segment()</i> must be <i>true</i> .
<i>Point_2</i>	<i>s.source()</i>	Returns the source endpoint of the segment. Note that this method can construct an inexact point if the number type used is inexact. <i>Precondition:</i> <i>s.is_segment()</i> must be <i>true</i> .
<i>Point_2</i>	<i>s.target()</i>	Returns the target endpoint of the segment. Note that this method can construct an inexact point if the number type used is inexact. <i>Precondition:</i> <i>s.is_segment()</i> must be <i>true</i> .
<i>SegmentDelaunayGraphSite_2</i>	<i>s.supporting_site()</i>	Returns a segment site object representing the segment that supports the segment represented by the site. Both endpoints of the returned site are input points. <i>Precondition:</i> <i>s.is_segment()</i> must be <i>true</i> .
<i>SegmentDelaunayGraphSite_2</i>	<i>s.supporting_site(unsigned int i)</i>	Returns a segment site object representing the <i>i</i> -th segment that supports the point of intersection represented by the site. Both endpoints of the returned site are input points. <i>Precondition:</i> <i>i</i> must be at most 1, <i>s.is_point()</i> must be <i>true</i> and <i>s.is_input()</i> must be <i>false</i> .
<i>SegmentDelaunayGraphSite_2</i>	<i>s.crossing_site(unsigned int i)</i>	Returns a segment site object representing the <i>i</i> -th segment that supports the <i>i</i> -th endpoint of the site which is not the supporting segment of the site. Both endpoints of the returned site are input points. <i>Precondition:</i> <i>i</i> must be at most 1, <i>s.is_segment()</i> must be <i>true</i> and <i>s.is_input(i)</i> must be <i>false</i> .

<i>SegmentDelaunayGraphSite_2</i>	<i>s.source_site()</i>	Returns a point site object representing the source point of the site. <i>Precondition: s.is_segment()</i> must be <i>true</i> .
<i>SegmentDelaunayGraphSite_2</i>	<i>s.target_site()</i>	Returns a point site object representing the target point of the site. <i>Precondition: s.is_segment()</i> must be <i>true</i> .
<i>Point_2</i>	<i>s.source_of_supporting_site()</i>	Returns the source point of the supporting site of the this site. <i>Precondition: is_segment()</i> must be <i>true</i> .
<i>Point_2</i>	<i>s.target_of_supporting_site()</i>	Returns the target point of the supporting site of the this site. <i>Precondition: is_segment()</i> must be <i>true</i> .
<i>Point_2</i>	<i>s.source_of_supporting_site(unsigned int i)</i>	Returns the source point of the <i>i</i> -th supporting site of the this site. <i>Precondition: is_point()</i> must be <i>true</i> , <i>is_input()</i> must be <i>false</i> and <i>i</i> must either be 0 or 1.
<i>Point_2</i>	<i>s.target_of_supporting_site(unsigned int i)</i>	Returns the target point of the <i>i</i> -th supporting site of the this site. <i>Precondition: is_point()</i> must be <i>true</i> , <i>is_input()</i> must be <i>false</i> and <i>i</i> must either be 0 or 1.
<i>Point_2</i>	<i>s.source_of_crossing_site(unsigned int i)</i>	Returns the source point of the <i>i</i> -th crossing site of the this site. <i>Precondition: is_segment()</i> must be <i>true</i> , <i>is_input(i)</i> must be <i>false</i> and <i>i</i> must either be 0 or 1.
<i>Point_2</i>	<i>s.target_of_crossing_site(unsigned int i)</i>	Returns the target point of the <i>i</i> -th supporting site of the this site. <i>Precondition: is_segment()</i> must be <i>true</i> , <i>is_input(i)</i> must be <i>false</i> and <i>i</i> must either be 0 or 1.

Has Models

CGAL::Segment_Delaunay_graph_site_2<K>

See Also

SegmentDelaunayGraphTraits_2

CGAL::Segment_Delaunay_graph_site_2<K>

CGAL::Segment_Delaunay_graph_traits_2<K,MTag>

CGAL::Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>

CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>

CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>

CGAL::Segment_Delaunay_graph_site_2<K>

Definition

The class *Segment_Delaunay_graph_site_2<K>* is a model for the concept *SegmentDelaunayGraphSite_2*. It is parametrized by a template parameter *K* which must be a model of the *Kernel* concept.

```
#include <CGAL/Segment_Delaunay_graph_site_2.h>
```

Is Model for the Concepts

SegmentDelaunayGraphSite_2

Types

The class *Segment_Delaunay_graph_site_2<K>* introduces the following type in addition to the types in the concept *SegmentDelaunayGraphSite_2*.

```
typedef K          Kernel;          A type for the template parameter K.
```

See Also

Kernel

SegmentDelaunayGraphSite_2

CGAL::Segment_Delaunay_graph_traits_2<K,MTag>

CGAL::Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>

CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>

CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>

SegmentDelaunayGraphStorageSite_2

Definition

The concept *SegmentDelaunayGraphStorageSite_2* provides the requirements for the storage sites of a segment Delaunay graph. The storage sites are sites that are used to store the information of a site in a more compact form (that uses less storage). This is achieved by storing handles to points instead of points.

Refines

DefaultConstructible
CopyConstructible
Assignable

Types

<i>SegmentDelaunayGraphStorageSite_2:: Site_2</i>		The site type.
<i>typedef typename std::set<typename Site_2::Point_2>::iterator</i>	<i>Point_handle;</i>	The type for a handle to a point.

Creation

In addition to the default and copy constructors, the following static methods should be available for constructing sites:

<i>SegmentDelaunayGraphStorageSite_2</i>	<i>ss.construct_storage_site_2(Point_handle hp)</i>	Constructs a storage site from a point handle. The storage site represents the point associated with the point handle <i>hp</i> .
<i>SegmentDelaunayGraphStorageSite_2</i>	<i>ss.construct_storage_site_2(Point_handle hp1, Point_handle hp2)</i>	Constructs a storage site from two point handles. The storage site represents the segment the endpoints of which are the points associated with the point handles <i>hp1</i> and <i>hp2</i> .
<i>SegmentDelaunayGraphStorageSite_2</i>	<i>ss.construct_storage_site_2(Point_handle hp1, Point_handle hp2, Point_handle hq1, Point_handle hq2)</i>	Constructs a storage site from four point handles. The storage site represents the point of intersection of the segments the endpoints of which are the points associated with the point handles <i>hp1</i> , <i>hp2</i> and <i>hq1</i> and <i>hq2</i> , respectively.

SegmentDelaunayGraphStorageSite_2 *ss.construct_storage_site_2(Point_handle hp1,*
Point_handle hp2,
Point_handle hq1,
Point_handle hq2,
bool b)

Constructs a site from four point handles and a boolean. The storage site represents a segment. If *b* is *true*, the first endpoint of the segment is the point associated with the handle *hp1* and the second endpoint is the point of intersection of the segments the endpoints of which are the point associated with the point handles *hp1*, *hp2* and *hq1*, *hq2*, respectively. If *b* is *false*, the first endpoint of the represented segment is the one mentioned above, whereas the second endpoint if the point associated with the point handle *hp2*.

SegmentDelaunayGraphStorageSite_2 *ss.construct_storage_site_2(Point_handle hp1,*
Point_handle hp2,
Point_handle hq1,
Point_handle hq2,
Point_handle hr1,
Point_handle hr2)

Constructs a storage site from six point handles. The storage site represents of segment the endpoints of which are points of intersection of two pairs of segments, the endpoints of which are *hp1*, *hp2/hq1*, *hq2* and *hp1*, *hp2/hr1*, *hr2*, respectively.

Predicates

<i>bool</i> <i>ss.is_defined()</i> <i>bool</i> <i>ss.is_point()</i> <i>bool</i> <i>ss.is_segment()</i> <i>bool</i> <i>ss.is_input()</i>	Returns <i>true</i> if the storage site represents a valid point or segment. Returns <i>true</i> if the storage site represents a point. Returns <i>true</i> if the storage site represents a segment. Returns <i>true</i> if the storage site represents an input point or a segment defined by two input points. Returns <i>false</i> if it represents a point of intersection of two segments, or if it represents a segment, at least one endpoint of which is a point of intersection of two segments.
<i>bool</i> <i>ss.is_input(unsigned int i)</i>	Returns <i>true</i> if the <i>i</i> -th endpoint of the corresponding site is an input point. Returns <i>false</i> if the <i>i</i> -th endpoint of the corresponding site is the intersection of two segments. <i>Precondition:</i> <i>i</i> must be at most 1, and <i>ss.is_segment()</i> must be <i>true</i> .

Access Functions

SegmentDelaunayGraphStorageSite_2 *ss.supporting_site()*

Returns a storage site object representing the segment that supports the segment represented by the storage site. The returned storage site represents a site, both endpoints of which are input points.
Precondition: *ss.is_segment()* must be *true*.

<i>SegmentDelaunayGraphStorageSite_2</i>	<i>ss.source_site()</i>	Returns a storage site that represents the first endpoint of the represented segment. <i>Precondition: ss.is_segment() must be true.</i>
<i>SegmentDelaunayGraphStorageSite_2</i>	<i>ss.target_site()</i>	Returns a storage site that represents the second endpoint of the represented segment. <i>Precondition: ss.is_segment() must be true.</i>
<i>SegmentDelaunayGraphStorageSite_2</i>	<i>ss.supporting_site(unsigned int i)</i>	Returns a storage site object representing the <i>i</i> -th segment that supports the point of intersection represented by the storage site. The returned storage site represents a site, both endpoints of which are input points. <i>Precondition: i must be at most 1, ss.is_point() must be true and ss.is_input() must be false.</i>
<i>SegmentDelaunayGraphStorageSite_2</i>	<i>ss.crossing_site(unsigned int i)</i>	Returns a storage site object representing the <i>i</i> -th segment that supports the <i>i</i> -th endpoint of the site which is not the supporting segment of the site. The returned storage site represents a site, both endpoints of which are input points. <i>Precondition: i must be at most 1, ss.is_segment() must be true and ss.is_input(i) must be false.</i>
<i>Site_2</i>	<i>ss.site()</i>	Returns the site represented by the storage site.
<i>Point_handle</i>	<i>ss.point()</i>	Returns a handle associated with the represented point. <i>Precondition: is_point() and is_input() must both be true.</i>
<i>Point_handle</i>	<i>ss.source_of_supporting_site()</i>	Returns a handle to the source point of the supporting site of the this site. <i>Precondition: is_segment() must be true.</i>
<i>Point_handle</i>	<i>ss.target_of_supporting_site()</i>	Returns a handle to the target point of the supporting site of the this site. <i>Precondition: is_segment() must be true.</i>
<i>Point_handle</i>	<i>ss.source_of_supporting_site(unsigned int i)</i>	Returns a handle to the source point of the <i>i</i> -th supporting site of the this site. <i>Precondition: is_point() must be true, is_input() must be false and i must either be 0 or 1.</i>
<i>Point_handle</i>	<i>ss.target_of_supporting_site(unsigned int i)</i>	Returns a handle to the target point of the <i>i</i> -th supporting site of the this site. <i>Precondition: is_point() must be true, is_input() must be false and i must either be 0 or 1.</i>

Point_handle

ss.source_of_crossing_site(unsigned int i)

Returns a handle to the source point of the *i*-th crossing site of the this site.

Precondition: *is_segment()* must be *true*, *is_input(i)* must be *false* and *i* must either be 0 or 1.

Point_handle

ss.target_of_crossing_site(unsigned int i)

Returns a handle to the target point of the *i*-th supporting site of the this site.

Precondition: *is_segment()* must be *true*, *is_input(i)* must be *false* and *i* must either be 0 or 1.

Has Models

CGAL::Segment_Delaunay_graph_storage_site_2<Gt>

See Also

SegmentDelaunayGraphTraits_2

CGAL::Segment_Delaunay_graph_site_2<K>

CGAL::Segment_Delaunay_graph_storage_site_2<Gt>

CGAL::Segment_Delaunay_graph_traits_2<K,MTag>

CGAL::Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>

CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>

CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>

CGAL::Segment_Delaunay_graph_storage_site_2<Gt>

Definition

The class *Segment_Delaunay_graph_storage_site_2<Gt>* is a model for the concept *SegmentDelaunayGraphStorageSite_2*. It is parametrized by a single template parameter *Gt*, which must be a model of the *SegmentDelaunayGraphTraits_2* concept.

```
#include <CGAL/Segment_Delaunay_graph_storage_site_2.h>
```

Is Model for the Concepts

SegmentDelaunayGraphStorageSite_2

Types

The class *Segment_Delaunay_graph_storage_site_2<Gt>* introduces the following type in addition to the types in the concept *SegmentDelaunayGraphStorageSite_2*.

typedef Gt *Geom_traits*; A type for the template parameter *Gt*.

See Also

SegmentDelaunayGraphSite_2

SegmentDelaunayGraphTraits_2

CGAL::Segment_Delaunay_graph_site_2<K>

CGAL::Segment_Delaunay_graph_traits_2<K,MTag>

CGAL::Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>

CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>

CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>

SegmentDelaunayGraphDataStructure_2

Definition

The concept *SegmentDelaunayGraphDataStructure_2* refines the concept *ApolloniusGraphDataStructure_2*. In addition it provides two methods for the merging of two vertices joined by an edge of the data structure, and the splitting of a vertex into two. The method that merges two vertices, called *join_vertices* identifies the two vertices and deletes their common two faces. The method that splits a vertex, called *split_vertex* introduces a new vertex that shares an edge and two faces with the old vertex (see figure below). Notice that the *join_vertices* and *split_vertex* operations are complementary, in the sense that one reverses the action of the other.

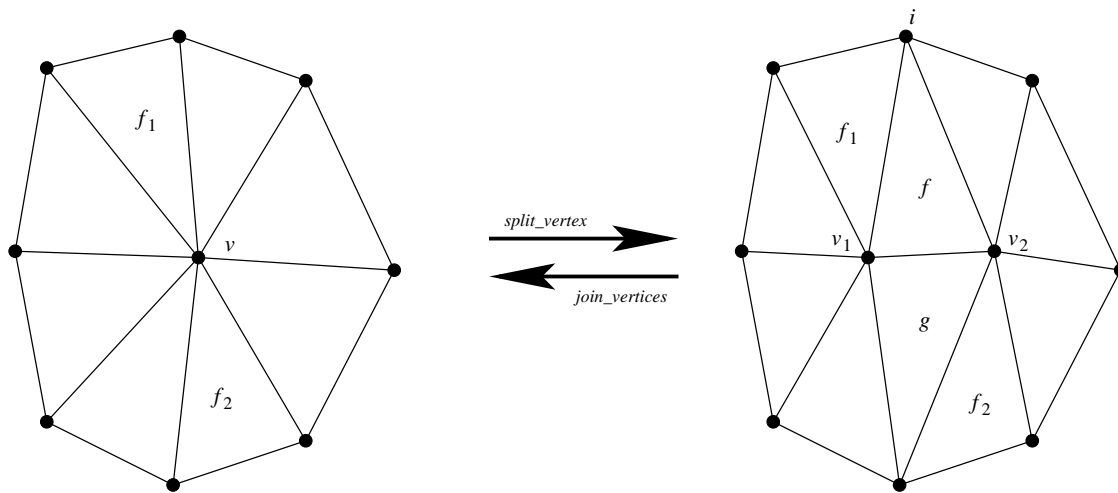


Figure 26.3: The join and split operations. Left to right: The vertex v is split into v_1 and v_2 . The faces f and g are inserted after f_1 and f_2 , respectively, in the counter-clockwise sense. The vertices v_1 , v_2 and the faces f and g are returned as a *boost tuple* in that order. Right to left: The edge (f, i) is collapsed, and thus the vertices v_1 and v_2 are joined. The vertex v is returned.

We only describe the additional requirements with respect to the *ApolloniusGraphDataStructure_2* concept.

Refines

ApolloniusGraphDataStructure_2

Modification

<p><i>Vertex_handle</i> <i>sdgds.join_vertices</i>(<i>Face_handle</i> f, <i>int</i> i)</p>	<p>Joins the vertices that are endpoints of the edge (f, i). It returns a vertex handle to common vertex.</p>
--------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------

boost::tuples::tuple<Vertex_handle, Vertex_handle, Face_handle, Face_handle>

sdgds.split_vertex(Vertex_handle v, Face_handle f1, Face_handle f2)

Splits the vertex v into two vertices $v1$ and $v2$. The common faces f and g of $v1$ and $v2$ are created after (in the counter-clockwise sense) the faces $f1$ and $f2$. The 4-tuple $(v1, v2, f, g)$ is returned (see Fig. [26.6](#)).

Has Models

CGAL::Triangulation_data_structure_2<Vb,Fb>

See Also

TriangulationDataStructure_2

ApolloniusGraphDataStructure_2

SegmentDelaunayGraphVertexBase_2

TriangulationFaceBase_2

SegmentDelaunayGraphVertexBase_2

Definition

The concept *SegmentDelaunayGraphVertexBase_2* describes the requirements for the vertex base class of the *SegmentDelaunayGraphDataStructure_2* concept. A vertex stores a site of the segment Delaunay graph and provides access to one of its incident faces through a *Face_handle*.

Refines

DefaultConstructible
CopyConstructible
Assignable

Types

<i>SegmentDelaunayGraphVertexBase_2::Geom_traits</i>	A type for the geometric traits that defines the site. <i>Precondition:</i> The type <i>Geom_traits</i> must define the type <i>Site_2</i> .
<i>SegmentDelaunayGraphVertexBase_2::Site_2</i>	A type for the site. This type must coincide with the type <i>Geom_traits::Site_2</i> .
<i>SegmentDelaunayGraphVertexBase_2::Storage_site_tag</i>	A type that indicates what kind of storage type to use. <i>Storage_site_tag</i> must either be <i>CGAL::Tag_true</i> or <i>CGAL::Tag_false</i> .
<i>SegmentDelaunayGraphVertexBase_2::Storage_site_2</i>	A type for the internal representation of sites. This type must satisfy the requirements of the concept <i>SegmentDelaunayGraphStorageSite_2</i> .
<i>SegmentDelaunayGraphVertexBase_2::Data_structure</i>	A type for the underlying data structure, to which the vertex belongs to.
<i>SegmentDelaunayGraphVertexBase_2::Vertex_handle</i>	A type for the vertex handle of the segment Delaunay graph data structure.
<i>SegmentDelaunayGraphVertexBase_2::Face_handle</i>	A type for the face handle of the segment Delaunay graph data structure.

Creation

In addition to the default and copy constructors and following constructors are required:

<i>SegmentDelaunayGraphVertexBase_2</i> <i>v</i> (<i>Storage_site_2</i> <i>ss</i>);	Constructs a vertex associated with the site represented by the storage site <i>ss</i> .
<i>SegmentDelaunayGraphVertexBase_2</i> <i>v</i> (<i>Storage_site_2</i> <i>ss</i> , <i>Face_handle</i> <i>f</i>);	Constructs a vertex associated with the site represented by the storage site <i>ss</i> , and pointing to the face associated with the face handle <i>f</i> .

Access Functions

<i>Storage_site_2</i>	<i>v.storage_site()</i>	Returns the storage site representing the site.
<i>Site_2</i>	<i>v.site()</i>	Returns the site.
<i>Face_handle</i>	<i>v.face()</i>	Returns a handle to an incident face.

Setting

<i>void</i>	<i>v.set_site(Storage_site_2 ss)</i>	Sets the storage site.
<i>void</i>	<i>v.set_face(Face_handle f)</i>	Sets the incident face.

Checking

<i>bool</i>	<i>v.is_valid(bool verbose, int level)</i>	Performs any required tests on a vertex.
-------------	---------------------------------------------	------------------------------------------

Has Models

CGAL::Segment_Delaunay_graph_vertex_base_2<Gt>.

See Also

SegmentDelaunayGraphDataStructure_2
SegmentDelaunayGraphTraits_2
SegmentDelaunayGraphSite_2
SegmentDelaunayGraphStorageSite_2
CGAL::Segment_Delaunay_graph_vertex_base_2<Gt>
CGAL::Segment_Delaunay_graph_site_2<K>
CGAL::Segment_Delaunay_graph_storage_site_2<Gt,SSTag>
CGAL::Triangulation_data_structure_2<Vb,Fb>

CGAL::Segment_Delaunay_graph_vertex_base_2<Gt,SSTag>

Definition

The class *Segment_Delaunay_graph_vertex_base_2<Gt,SSTag>* provides a model for the *SegmentDelaunayGraphVertexBase_2* concept which is the vertex base required by the *SegmentDelaunayGraphDataStructure_2* concept. The class *Segment_Delaunay_graph_vertex_base_2<Gt,SSTag>* has two template arguments, the first being the geometric traits of the segment Delaunay graph and should be a model of the concept *SegmentDelaunayGraphTraits_2*. The second template argument indicates whether or not to use the simple storage site that does not support intersecting segments, or the full storage site, that supports intersecting segments. The possible values are *CGAL::Tag_true* and *CGAL::Tag_false*. *CGAL::Tag_true* indicates that the full storage site is to be used, whereas *CGAL::Tag_false* indicates that the simple storage site is to be used.

```
#include <CGAL/Segment_Delaunay_graph_vertex_base_2.h>
```

Is Model for the Concepts

SegmentDelaunayGraphVertexBase_2

See Also

SegmentDelaunayGraphVertexBase_2

SegmentDelaunayGraphDataStructure_2

SegmentDelaunayGraphTraits_2

CGAL::Triangulation_data_structure_2<Vb,Fb>

CGAL::Segment_Delaunay_graph_traits_2<K,MTag>

CGAL::Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>

CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>

CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>

SegmentDelaunayGraphTraits_2

Definition

The concept *SegmentDelaunayGraphTraits_2* provides the traits requirements for the *Segment_Delaunay_graph_2*<Gt,DS> and *Segment_Delaunay_graph_hierarchy_2*<Gt,STag,DS> classes. In particular, it provides a type *Site_2*, which must be a model of the concept *SegmentDelaunayGraphSite_2*. It also provides constructions for sites and several function object types for the predicates.

Refines

DefaultConstructible
CopyConstructible
Assignable

Types

<i>SegmentDelaunayGraphTraits_2:: Intersections_tag</i>	Indicates or not whether the intersecting segments are to be supported. The tag must either be <i>CGAL::Tag_true</i> or <i>CGAL::Tag_false</i> .
<i>SegmentDelaunayGraphTraits_2:: Site_2</i>	A type for a site of the segment Delaunay graph. Must be a model of the concept <i>SegmentDelaunayGraphSite_2</i> .
<i>SegmentDelaunayGraphTraits_2:: Point_2</i>	A type for a point.
<i>SegmentDelaunayGraphTraits_2:: Line_2</i>	A type for a line. Only required if the segment Delaunay graph is inserted in a stream.
<i>SegmentDelaunayGraphTraits_2:: Ray_2</i>	A type for a ray. Only required if the segment Delaunay graph is inserted in a stream.
<i>SegmentDelaunayGraphTraits_2:: Segment_2</i>	A type for a segment. Only required if if the segment Delaunay graph is inserted in a stream.
<i>SegmentDelaunayGraphTraits_2:: FT</i>	A type for the field number type of sites, points, etc..
<i>SegmentDelaunayGraphTraits_2:: RT</i>	A type for the ring number type of sites, points, etc.
<i>SegmentDelaunayGraphTraits_2:: Arrangement_type</i>	An enumeration type that indicates the type of the arrangement of two sites. The possible values are <i>DISJOINT</i> , <i>IDENTICAL</i> , <i>CROSSING</i> , <i>TOUCHING_1</i> , <i>TOUCHING_2</i> , <i>TOUCHING_11</i> , <i>TOUCHING_12</i> , <i>TOUCHING_21</i> , <i>TOUCHING_22</i> , <i>OVERLAPPING_11</i> , <i>OVERLAPPING_12</i> , <i>OVERLAPPING_21</i> , <i>OVERLAPPING_22</i> , <i>INTERIOR</i> , <i>INTERIOR_1</i> , <i>INTERIOR_2</i> , <i>TOUCHING_11_INTERIOR_1</i> , <i>TOUCHING_11_INTERIOR_2</i> , <i>TOUCHING_12_INTERIOR_1</i> , <i>TOUCHING_12_INTERIOR_2</i> , <i>TOUCHING_21_INTERIOR_1</i> , <i>TOUCHING_21_INTERIOR_2</i> , <i>TOUCHING_22_INTERIOR_1</i> , <i>TOUCHING_22_INTERIOR_2</i> . A detailed description of the meaning of these values is shown the end of the reference manual for this concept. (to be done)

<i>SegmentDelaunayGraphTraits_2:: Object_2</i>	A type representing different types of objects in two dimensions, namely: <i>Point_2</i> , <i>Site_2</i> , <i>Line_2</i> , <i>Ray_2</i> and <i>Segment_2</i> .
<i>SegmentDelaunayGraphTraits_2:: Assign_2</i>	Must provide <i>template <class T> bool operator() (T& t, Object_2 o)</i> which assigns <i>o</i> to <i>t</i> if <i>o</i> was constructed from an object of type <i>T</i> . Returns <i>true</i> , if the assignment was possible.
<i>SegmentDelaunayGraphTraits_2:: Construct_object_2</i>	Must provide <i>template <class T> Object_2 operator() (T t)</i> that constructs an object of type <i>Object_2</i> that contains <i>t</i> and returns it.
<i>SegmentDelaunayGraphTraits_2:: Construct_svd_vertex_2</i>	A constructor for a point of the segment Voronoi diagram equidistant from three sites. Must provide <i>Point_2 operator() (Site_2 s1, Site_2 s2, Site_2 s3)</i> , which constructs a point equidistant from the sites <i>s1</i> , <i>s2</i> and <i>s3</i> .
<i>SegmentDelaunayGraphTraits_2:: Compare_x_2</i>	A predicate object type. Must provide <i>Comparison_result operator() (Site_2 s1, Site_2 s2)</i> , which compares the x-coordinates of the points represented by the sites <i>s1</i> and <i>s2</i> . <i>Precondition: s1 and s2 must be points.</i>
<i>SegmentDelaunayGraphTraits_2:: Compare_y_2</i>	A predicate object type. Must provide <i>Comparison_result operator() (Site_2 s1, Site_2 s2)</i> , which compares the y-coordinates of the points represented by the sites <i>s1</i> and <i>s2</i> . <i>Precondition: s1 and s2 must be points.</i>
<i>SegmentDelaunayGraphTraits_2:: Orientation_2</i>	A predicate object type. Must provide <i>Orientation operator() (Site_2 s1, Site_2 s2, Site_2 s3)</i> , which performs the usual orientation test for three points. <i>s1</i> , <i>s2</i> and <i>s3</i> . <i>Precondition: the sites s1, s2 and s3 must be points.</i>
<i>SegmentDelaunayGraphTraits_2:: Equal_2</i>	A predicate object type. Must provide <i>bool operator() (Site_2 s1, Site_2 s2)</i> , which determines if the points represented by the sites <i>s1</i> and <i>s2</i> are identical. <i>Precondition: s1 and s2 must be points.</i>
<i>SegmentDelaunayGraphTraits_2:: Are_parallel_2</i>	A predicate object type. Must provide <i>bool operator() (Site_2 s1, Site_2 s2)</i> , which determines if the segments represented by the sites <i>s1</i> and <i>s2</i> are parallel. <i>Precondition: s1 and s2 must be segments.</i>
<i>SegmentDelaunayGraphTraits_2:: Oriented_side_of_bisector_2</i>	A predicate object type. Must provide <i>Oriented_side operator() (Site_2 s1, Site_2 s2, Point_2 p)</i> , which returns the oriented side of the bisector of <i>s1</i> and <i>s2</i> that contains <i>p</i> . Returns <i>ON_POSITIVE_SIDE</i> if <i>p</i> lies in the half-space of <i>s1</i> (i.e., <i>p</i> is closer to <i>s1</i> than <i>s2</i>); returns <i>ON_NEGATIVE_SIDE</i> if <i>p</i> lies in the half-space of <i>s2</i> ; returns <i>ON_ORIENTED_BOUNDARY</i> if <i>p</i> lies on the bisector of <i>s1</i> and <i>s2</i> .

SegmentDelaunayGraphTraits_2:: Vertex_conflict_2

A predicate object type. Must provide *Sign operator()(Site_2 s1, Site_2 s2, Site_2 s3, Site_2 q)*, which returns the sign of the distance of q from the Voronoi circle of $s1, s2, s3$ (the Voronoi circle of three sites $s1, s2, s3$ is a circle co-tangent to all three sites, that touches them in that order as we walk on its circumference in the counter-clockwise sense).

Precondition: the Voronoi circle of $s1, s2, s3$ must exist.

Must also provide *Sign operator()(Site_2 s1, Site_2 s2, Site_2 q)*, which returns the sign of the distance of q from the bitangent line of $s1, s2$ (a degenerate Voronoi circle, with its center at infinity).

SegmentDelaunayGraphTraits_2:: Finite_edge_interior_conflict_2

A predicate object type. Must provide *bool operator()(Site_2 s1, Site_2 s2, Site_2 s3, Site_2 s4, Site_2 q, Sign sgn)*. The sites $s1, s2, s3$ and $s4$ define a Voronoi edge that lies on the bisector of $s1$ and $s2$ and has as endpoints the Voronoi vertices defined by the triplets $s1, s2, s3$ and $s1, s4$ and $s2$. The sign sgn is the common sign of the distance of the site q from the Voronoi circle of the triplets $s1, s2, s3$ and $s1, s4$ and $s2$. In case that sgn is equal to *NEGATIVE*, the predicate returns *true* if and only if the entire Voronoi edge is in conflict with q . If sgn is equal to *POSITIVE* or *ZERO*, the predicate returns *false* if and only if q is not in conflict with the Voronoi edge.

Precondition: the Voronoi vertices of $s1, s2, s3$, and $s1, s4, s2$ must exist.

Must also provide *bool operator()(Site_2 s1, Site_2 s2, Site_2 s3, Site_2 q, Sign sgn)*. The sites $s1, s2, s3$ and the site at infinity s_∞ define a Voronoi edge that lies on the bisector of $s1$ and $s2$ and has as endpoints the Voronoi vertices v_{123} and $v_{1\infty 2}$ defined by the triplets $s1, s2, s3$ and $s1, s_\infty$ and $s2$ (the second vertex is actually at infinity). The sign sgn is the common sign of the distance of the site q from the two Voronoi circles centered at the Voronoi vertices v_{123} and $v_{1\infty 2}$. In case that sgn is *NEGATIVE*, the predicate returns *true* if and only if the entire Voronoi edge is in conflict with q . If sgn is *POSITIVE* or *ZERO*, the predicate returns *false* if and only if q is not in conflict with the Voronoi edge.

Precondition: the Voronoi vertex v_{123} of $s1, s2, s3$ must exist.

Must finally provide *bool operator()(Site_2 s1, Site_2 s2, Site_2 q, Sign sgn)*. The sites $s1, s2$ and the site at infinity s_∞ define a Voronoi edge that lies on the bisector of $v_{12\infty}$ and $v_{1\infty 2}$ $s1$ and $s2$ and has as endpoints the Voronoi vertices defined by the triplets $s1, s2, s_\infty$ and $s1, s_\infty$ and $s2$ (both vertices are actually at infinity). The sign sgn denotes the common sign of the distance of the site q from the Voronoi circles centered at $v_{12\infty}$ and $v_{1\infty 2}$. If sgn is *NEGATIVE*, the predicate returns *true* if and only if the entire Voronoi edge is in conflict with q . If *POSITIVE* or *ZERO* is *false*, the predicate returns *false* if and only if q is not in conflict with the Voronoi edge.

SegmentDelaunayGraphTraits_2:: Infinite_edge_interior_conflict_2

A predicate object type. Must provide *bool operator()(Site_2 s1, Site_2 s2, Site_2 s3, Site_2 q, Sign sgn)*. The sites $s_\infty, s1, s2$ and $s3$ define a Voronoi edge that lies on the bisector of s_∞ and $s1$ and has as endpoints the Voronoi vertices $v_{\infty 12}$ and $v_{\infty 31}$ defined by the triplets $s_\infty, s1, s2$ and $s_\infty, s3$ and $s1$. The sign sgn is the common sign of the distances of q from the Voronoi circles centered at the vertices $v_{\infty 12}$ and $v_{\infty 31}$. If sgn is *NEGATIVE*, the predicate returns *true* if and only if the entire Voronoi edge is in conflict with q . If sgn is *POSITIVE* or *ZERO*, the predicate returns *false* if and only if q is not in conflict with the Voronoi edge.

SegmentDelaunayGraphTraits_2:: Oriented_side_2

A predicate object type. Must provide *Oriented_side operator()*(*Site_1 s1*, *Site_2 s2*, *Site_2 s3*, *Site_2 s*, *Site_2 p*). Determines the oriented side of the line ℓ that contains the point site p , where ℓ is the line that passes through the Voronoi vertex of the sites $s1$, $s2$, $s3$ and is perpendicular to the segment site s .

Precondition: s must be a segment and p must be a point.

SegmentDelaunayGraphTraits_2:: Arrangement_type_2

A predicate object type. Must provide *Arrangement_type operator()*(*Site_2 s1*, *Site_2 s2*) that returns the type of the arrangement of the two sites $s1$ and $s2$.

Access to predicate objects

<i>Compare_x_2</i>	<i>gt.compare_x_2_object()</i>
<i>Compare_y_2</i>	<i>gt.compare_y_2_object()</i>
<i>Orientation_2</i>	<i>gt.orientation_2_object()</i>
<i>Equal_2</i>	<i>gt.equal_2_object()</i>
<i>Are_parallel_2</i>	<i>gt.are_parallel_2_object()</i>
<i>Oriented_side_of_bisector_2</i>	<i>gt.oriented_side_of_bisector_test_2_object()</i>
<i>Vertex_conflict_2</i>	<i>gt.vertex_conflict_2_object()</i>
<i>Finite_edge_interior_conflict_2</i>	<i>gt.finite_edge_interior_conflict_2_object()</i>
<i>Infinite_edge_interior_conflict_2</i>	<i>gt.infinite_edge_interior_conflict_2_object()</i>
<i>Oriented_side_2</i>	<i>gt.oriented_side_2_object()</i>
<i>Arrangement_type_2</i>	<i>gt.arrangement_type_2_object()</i>

Access to constructor objects

<i>Construct_object_2</i>	<i>gt.construct_object_2_object()</i>
<i>Construct_svd_vertex_2</i>	<i>gt.construct_svd_vertex_2_object()</i>

Access to other objects

<i>Assign_2</i>	<i>gt.assign_2_object()</i>
-----------------	-----------------------------

Has Models

CGAL::Segment_Delaunay_graph_traits_2<K,MTag>
CGAL::Segment_Delaunay_traits_without_intersections_2<K,MTag>
CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>
CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>

See Also

SegmentDelaunayGraphSite_2
CGAL::Segment_Delaunay_graph_2<Gt,DS>

CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>
CGAL::Segment_Delaunay_graph_traits_2<K,MTag>
CGAL::Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>
CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>
CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>

CGAL::Segment_Delaunay_graph_traits_2<K, MTag>

Definition

The class *Segment_Delaunay_graph_traits_2<K, MTag>* provides a model for the *SegmentDelaunayGraphTraits_2* concept. This class has two template parameters. The first template parameter must be a model of the *Kernel* concept. The second template parameter corresponds to how predicates are evaluated. There are two possible values for *MTag*, namely *CGAL::Sqrt_field_tag* and *CGAL::Field_tag*. The first one must be used when the number type used in the representation supports the exact evaluation of signs of expressions involving all four basic operations and square roots, whereas the second one requires the exact evaluation of signs of field-type expressions, i.e., expressions involving additions, subtractions, multiplications and divisions. The default value for *MTag* is *CGAL::Field_tag*. The way the predicates are evaluated is discussed in [Bur96] and [Kar04] (the geometric filtering part).

```
#include <CGAL/Segment_Delaunay_graph_traits_2.h>
```

Is Model for the Concepts

SegmentDelaunayGraphTraits_2

Types

```
typedef CGAL::Tag_true    Intersections_tag;
```

The *Segment_Delaunay_graph_traits_2<K, MTag>* class introduces a few additional types with respect to the *SegmentDelaunayGraphTraits_2* concept. These are:

<code>typedef K</code>	<code>Kernel;</code>	A typedef for the template parameter <i>K</i> .
<code>typedef MTag</code>	<code>Method_tag;</code>	A typedef for the template parameter <i>MTag</i> .

See Also

Kernel
SegmentDelaunayGraphTraits_2
CGAL::Field_tag
CGAL::Sqrt_field_tag
CGAL::Segment_Delaunay_graph_2<Gt, DS>
CGAL::Segment_Delaunay_graph_hierarchy_2<Gt, STag, DS>
CGAL::Segment_Delaunay_graph_traits_without_intersections_2<K, MTag>
CGAL::Segment_Delaunay_graph_filtered_traits_2<CK, CM, EK, EM, FK, FM>
CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK, CM, EK, EM, FK, FM>

CGAL::Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>

Definition

The class *Segment_Delaunay_graph_traits_without_intersections_2*<*K*,*MTag*> provides a model for the *SegmentDelaunayGraphTraits_2* concept. This class has two template parameters. The first template parameter must be a model of the *Kernel* concept. The second template parameter corresponds to how predicates are evaluated. There are two possible values for *MTag*, namely *CGAL::Sqrt_field_tag* and *CGAL::Ring_tag*. The first one must be used when the number type used in the representation supports the exact evaluation of signs of expressions involving all four basic operations and square roots, whereas the second one requires the exact evaluation of signs of ring-type expressions, i.e., expressions involving only additions, subtractions and multiplications. The default value for *MTag* is *CGAL::Ring_tag*. The way the predicates are evaluated is discussed in [Bur96] and [Kar04] (the geometric filtering part).

```
#include <CGAL/Segment_Delaunay_graph_traits_2.h>
```

Is Model for the Concepts

SegmentDelaunayGraphTraits_2

Types

```
typedef CGAL::Tag_false    Intersections_tag;
```

The *Segment_Delaunay_graph_traits_without_intersections_2*<*K*,*MTag*> class introduces a few additional types with respect to the *SegmentDelaunayGraphTraits_2* concept. These are:

<code>typedef K</code>	<code>Kernel;</code>	A typedef for the template parameter <i>K</i> .
<code>typedef MTag</code>	<code>Method_tag;</code>	A typedef for the template parameter <i>MTag</i> .

See Also

Kernel
SegmentDelaunayGraphTraits_2
CGAL::Ring_tag
CGAL::Sqrt_field_tag
CGAL::Segment_Delaunay_graph_2<*Gt*,*DS*>
CGAL::Segment_Delaunay_graph_hierarchy_2<*Gt*,*S**Tag*,*DS*>
CGAL::Segment_Delaunay_graph_traits_2<*K*,*MTag*>
CGAL::Segment_Delaunay_graph_filtered_traits_2<*CK*,*CM*,*EK*,*EM*,*FK*,*FM*>
CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<*CK*,*CM*,*EK*,*EM*,*FK*,*FM*>

CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>

Definition

The class *Segment_Delaunay_graph_filtered_traits_2*<CK,CM,EK,EM,FK,FM> provides a model for the *SegmentDelaunayGraphTraits_2* concept.

The class *Segment_Delaunay_graph_filtered_traits_2*<CK,CM,EK,EM,FK,FM> uses the filtering technique [BBP01] to achieve traits for the *Segment_Delaunay_graph_2*<Gt,DS> class with efficient and exact predicates given an exact kernel *EK* and a filtering kernel *FK*. The geometric constructions associated provided by this class are equivalent to those provided by the traits class *Segment_Delaunay_graph_traits_2*<CK,CM>, which means that they may be inexact depending on the choice of the *CK* kernel.

This class has six template parameters. The first, third and fifth template parameters must be a models of the *Kernel* concept. The parameter *CK* is the construction kernel and it is the kernel that will be used for constructions. The parameter *FK* is the filtering kernel; this kernel will be used for performing the arithmetic filtering for the predicates involved in the computation of the segment Delaunay graph. Finally, the parameter *EK* is the exact kernel; this kernel will be used for computing the predicates if the filtering kernel fails to produce an answer.

The second, fourth and sixth template parameters correspond to how predicates are evaluated. There are two predefined possible values for these parameters, namely *CGAL::Sqrt_field_tag* and *CGAL::Field_tag*. The first one must be used when the number type used in the representation supports the exact evaluation of signs of expressions involving all four basic operations and square roots, whereas the second one requires that only field operations are exact. Finally, in order to get exact constructions *CM* must be set to *CGAL::Sqrt_field_tag* and the number type in *CK* must support operations involving divisions and square roots (as well as the other three basic operations of course). The way the predicates are evaluated is discussed in [Bur96] and [Kar04] (the geometric filtering part).

The default values for the template parameters are as follows: *CM* = *CGAL::Sqrt_field_tag* (it is assumed that *CGAL::Cartesian<double>* or *CGAL::Simple_cartesian<double>* will be the entry for the template parameter *CK*), *EM* = *CGAL::Field_tag*, *FK* = *CGAL::Simple_cartesian<CGAL::Interval_nt<false> >*, *FM* = *CGAL::Sqrt_field_tag*. If the GMP package is installed with CGAL, the template parameter *EK* has the default value: *EK* = *CGAL::Simple_cartesian<CGAL::Gmpq>*, otherwise its default value is *EK* = *CGAL::Simple_cartesian<CGAL::Quotient<CGAL::MP_Float> >*.

```
#include <CGAL/Segment_Delaunay_graph_filtered_traits_2.h>
```

Is Model for the Concepts

```
SegmentDelaunayGraphTraits_2
DefaultConstructible
CopyConstructible
Assignable
```

Types

```
typedef CGAL::Tag_true    Intersections_tag;
```

In addition to the types required by the *SegmentDelaunayGraphTraits_2* concept the class *Segment_Delaunay_graph_filtered_traits_2*<*CK,CM,EK,EM,FK,FM*> defines the following types:

```
typedef CK          Kernel;
typedef CK          Construction_kernel;
typedef FK          Filtering_kernel;
typedef EK          Exact_kernel;
typedef CM          Method_tag;
typedef CM          Construction_traits_method_tag;
typedef FM          Filtering_traits_method_tag;
typedef EM          Exact_traits_method_tag;
Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>:: Construction_traits;
```

A type for the segment Delaunay graph traits, where the kernel is *CK*.

```
Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>:: Filtering_traits;
```

A type for the segment Delaunay graph traits, where the kernel is *FK*.

```
Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>:: Exact_traits;
```

A type for the segment Delaunay graph traits, where the kernel is *EK*.

See Also

```
Kernel
SegmentDelaunayGraphTraits_2
CGAL::Field_tag
CGAL::Sqrt_field_tag
CGAL::Segment_Delaunay_graph_2<Gt,DS>
CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>
CGAL::Segment_Delaunay_graph_traits_2<K,MTag>
CGAL::Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>
CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>
```

CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>

Definition

The class *Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>* provides a model for the *SegmentDelaunayGraphTraits_2* concept.

The class *Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>* uses the filtering technique [BBP01] to achieve traits for the *Segment_Delaunay_graph_2<Gt,DS>* class with efficient and exact predicates given an exact kernel *EK* and a filtering kernel *FK*. The geometric constructions associated provided by this class are equivalent to those provided by the traits class *Segment_Delaunay_graph_traits_without_intersections_2<CK,CM>*, which means that they may be inexact, depending on the choice of the *CK* kernel.

This class has six template parameters. The first, third and fifth template parameters must be a models of the *Kernel* concept. The parameter *CK* is the construction kernel and it is the kernel that will be used for constructions. The parameter *FK* is the filtering kernel; this kernel will be used for performing the arithmetic filtering for the predicates involved in the computation of the segment Delaunay graph. Finally, the parameter *EK* is the exact kernel; this kernel will be used for computing the predicates if the filtering kernel fails to produce an answer.

The second, fourth and sixth template parameters correspond to how predicates are evaluated. There are two predefined possible values for these parameters, namely *CGAL::Sqrt_field_tag* and *CGAL::Ring_tag*. The first one must be used when the number type used in the representation supports the exact evaluation of signs of expressions involving all four basic operations and square roots, whereas the second requires the exact evaluation of signs of ring-type expressions, i.e., expressions involving only additions, subtractions and multiplications. Finally, in order to get exact constructions *CM* must be set to *CGAL::Sqrt_field_tag* and the number type in *CK* must support operations involving divisions and square roots (as well as the other three basic operations of course). The way the predicates are evaluated is discussed in [Bur96] and [Kar04] (the geometric filtering part).

The default values for the template parameters are as follows: *CM* = *CGAL::Sqrt_field_tag* (it is assumed that *CGAL::Cartesian<double>* or *CGAL::Simple_cartesian<double>* will be the entry for the template parameter *CK*), *EM* = *CGAL::Ring_tag*, *FK* = *CGAL::Simple_cartesian<CGAL::Interval_nt<false>>*, *FM* = *CGAL::Sqrt_field_tag*. If the GMP package is installed with CGAL, the template parameter *EK* has the default value: *EK* = *CGAL::Simple_cartesian<CGAL::Gmpq>*, otherwise its default value is *EK* = *CGAL::Simple_cartesian<CGAL::MP_Float>*.

```
#include <CGAL/Segment_Delaunay_graph_filtered_traits_2.h>
```

Is Model for the Concepts

```
SegmentDelaunayGraphTraits_2
DefaultConstructible
CopyConstructible
Assignable
```

Types

```
typedef CGAL::Tag_false    Intersections_tag;
```

In addition to the types required by the *SegmentDelaunayGraphTraits_2* concept the class *Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>* defines the following types:

```
typedef CK          Kernel;
typedef CK          Construction_kernel;
typedef FK          Filtering_kernel;
typedef EK          Exact_kernel;
typedef CM          Method_tag;
typedef CM          Construction_traits_method_tag;
typedef FM          Filtering_traits_method_tag;
typedef EM          Exact_traits_method_tag;
Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>:: Construction_
traits;
```

A type for the segment Delaunay graph traits, where the kernel is *CK*.

Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>:: Filtering_traits;

A type for the segment Delaunay graph traits, where the kernel is *FK*.

Segment_Delaunay_graph_filtered_traits_without_intersections_2<CK,CM,EK,EM,FK,FM>:: Exact_traits;

A type for the segment Delaunay graph traits, where the kernel is *EK*.

See Also

Kernel
SegmentDelaunayGraphTraits_2
CGAL::Ring_tag
CGAL::Sqrt_field_tag
CGAL::Segment_Delaunay_graph_2<Gt,DS>
CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>
CGAL::Segment_Delaunay_graph_traits_2<K,MTag>
CGAL::Segment_Delaunay_graph_traits_without_intersections_2<K,MTag>
CGAL::Segment_Delaunay_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>

CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>

Definition

We provide an alternative to the class *Segment_Delaunay_graph_2<Gt,DS>* for the incremental construction of the segment Delaunay graph. The *Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>* class maintains a hierarchy of Delaunay graphs. There are two possibilities as to how this hierarchy is constructed.

In the first case the bottom-most level of the hierarchy contains the full segment Delaunay graph. The upper levels of the hierarchy contain only points that are either point sites or endpoints of segment sites in the bottom-most Delaunay graph. A point that is in level i (either as an individual point or as the endpoint of a segment), is inserted in level $i + 1$ with probability $1/\alpha$ where $\alpha > 1$ is some constant. In the second case the upper levels of the hierarchy contains not only points but also segments. A site that is in level i , is in level $i + 1$ with probability $1/\beta$ where $\beta > 1$ is some constant.

The difference between the *Segment_Delaunay_graph_2<Gt,DS>* class and the *Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>* class (both versions of it) is on how the nearest neighbor location is done. Given a point p the location is done as follows: at the top most level we find the nearest neighbor of p as in the *Segment_Delaunay_graph_2<Gt,DS>* class. At every subsequent level i we use the nearest neighbor found at level $i + 1$ to find the nearest neighbor at level i . This is a variant of the corresponding hierarchy for points found in [Dev02]. The details are described in [Kar04].

The class has three template parameters. The first and third have essentially the same semantics as in the *Segment_Delaunay_graph_2<Gt,DS>* class. The first template parameter must be a model of the *SegmentDelaunayGraphTraits_2* concept. The third template parameter must be a model of the *SegmentDelaunayGraphDataStructure_2* concept. However, the vertex base class that is to be used in the segment Delaunay graph data structure must be a model of the *SegmentDelaunayGraphHierarchyVertexBase_2* concept. The third template parameter defaults to *Triangulation_data_structure_2<Segment_Delaunay_graph_hierarchy_vertex_base_2<Segment_Delaunay_graph_vertex_base_2<Gt>>, Triangulation_face_base_2<Gt>>*. The second template parameter controls whether or not segments are added in the upper levels of the hierarchy. It's possible values are *CGAL::Tag_true* and *CGAL::Tag_false*. If it is set to *CGAL::Tag_true*, segments are also inserted in the upper levels of the hierarchy. The value *CGAL::Tag_false* indicates that only points are to be inserted in the upper levels of the hierarchy. The default value for the second template parameter is *CGAL::Tag_false*.

The *Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>* class derives publicly from the *Segment_Delaunay_graph_2<Gt,DS>* class. The interface is the same with its base class. In the sequel only additional types and methods defined are documented.

```
#include <CGAL/Segment_Delaunay_graph_hierarchy_2.h>
```

Is Model for the Concepts

DefaultConstructible
CopyConstructible
Assignable

Inherits From

CGAL::Segment_Delaunay_graph_2<Gt,DS>

Types

Segment_Delaunay_graph_hierarchy_2<*Gt*,*S**Tag*,*DS*> introduces the following types in addition to those introduced by its base class *Segment_Delaunay_graph_2*<*Gt*,*DS*>.

```
typedef STag                               Segments_in_hierarchy_tag;  
                                           A type for the STag template parameter.  
typedef CGAL::Segment_Delaunay_graph_2<Gt,DS> Base;    A type for the base class.
```

Creation

In addition to the default and copy constructors, the following constructors are defined:

```
Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS> sdgh( Gt gt=Gt() );  
                                           Creates a hierarchy of segment Delaunay graphs using gt as geometric  
                                           traits.  
template< class Input_iterator >  
Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS> sdgh( Input_iterator first,  
                                           Input_iterator beyond,  
                                           Gt gt=Gt() )  
                                           Creates a segment Delaunay graph hierarchy using gt as geometric  
                                           traits and inserts all sites in the range [first, beyond). Input_iterator  
                                           must be a model of InputIterator. The value type of Input_iterator  
                                           must be either Point_2 or Site_2.
```

I/O

```
std::ostream&    std::ostream& os << svdh    Writes the current state of the segment Delaunay graph hierar-  
                                           chy to an output stream. In particular, all sites in the diagram  
                                           are written to the stream (represented through appropriate in-  
                                           put sites), as well as the underlying combinatorial hierarchical  
                                           data structure.  
std::istream&    std::istream& is >> svdh    Reads the state of the segment Delaunay graph hierarchy from  
                                           an input stream.
```

See Also

SegmentDelaunayGraphDataStructure_2
SegmentDelaunayGraphTraits_2
SegmentDelaunayGraphHierarchyVertexBase_2
CGAL::Segment_Delaunay_graph_2<*Gt*,*DS*>
CGAL::Triangulation_data_structure_2<*Vb*,*Fb*>
CGAL::Segment_Delaunay_graph_traits_2<*K*,*M**Tag*>
CGAL::Segment_Delaunay_graph_traits_without_intersections_2<*K*,*M**Tag*>
CGAL::Segment_Delaunay_graph_filtered_traits_2<*CK*,*CM*,*EK*,*EM*,*FK*,*FM*>
CGAL::Segment_Delaunay_graph_filtered_traits_without_intersections_2<*CK*,*CM*,*EK*,*EM*,*FK*,*FM*>
CGAL::Segment_Delaunay_graph_hierarchy_vertex_base_2<*Vbb*>

SegmentDelaunayGraphHierarchyVertexBase_2

Definition

The vertex of a segment Delaunay graph included in a segment Delaunay graph hierarchy has to provide some pointers to the corresponding vertices in the graphs of the next and preceeding levels. Therefore, the concept *SegmentDelaunayGraphHierarchyVertexBase_2* refines the concept *SegmentDelaunayGraphVertexBase_2*, by adding two vertex handles to the corresponding vertices for the next and previous level graphs.

Refines

SegmentDelaunayGraphVertexBase_2

Types

SegmentDelaunayGraphHierarchyVertexBase_2 does not introduce any types in addition to those of *SegmentDelaunayGraphVertexBase_2*.

Creation

The *SegmentDelaunayGraphHierarchyVertexBase_2* concept does not introduce any constructors in addition to those of the *SegmentDelaunayGraphVertexBase_2* concept.

Operations

<i>Vertex_handle</i>	<i>v.up()</i>	Returns a handle to the corresponding vertex of the next level segment Delaunay graph. If such a vertex does not exist <i>Vertex_handle()</i> is returned.
<i>Vertex_handle</i>	<i>v.down()</i>	Returns a handle to the corresponding vertex of the previous level segment Delaunay graph. If such a vertex does not exist <i>Vertex_handle()</i> is returned.
<i>void</i>	<i>v.set_up(Vertex_handle u)</i>	Sets the handle for the vertex of the next level segment Delaunay graph.
<i>void</i>	<i>v.set_down(Vertex_handle d)</i>	Sets the handle for the vertex of the previous level segment Delaunay graph.

Has Models

CGAL::Segment_Delaunay_graph_hierarchy_vertex_base_2<CGAL::Segment_Delaunay_graph_vertex_base_2<Gt,SSTag>>

See Also

SegmentDelaunayGraphDataStructure_2
SegmentDelaunayGraphVertexBase_2
CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,DS>
CGAL::Triangulation_data_structure_2<Vb,Fb>
CGAL::Segment_Delaunay_graph_vertex_base_2<Gt,SSTag>
CGAL::Segment_Delaunay_graph_hierarchy_vertex_base_2<Vbb>

CGAL::Segment_Delaunay_graph_hierarchy_vertex_base_2<Vbb>

Definition

The class *Segment_Delaunay_graph_hierarchy_vertex_base_2<Vbb>* provides a model for the *SegmentDelaunayGraphHierarchyVertexBase_2* concept, which is the vertex base required by the *Segment_Delaunay_graph_hierarchy_2<Gt,DS>* class. The class *Segment_Delaunay_graph_hierarchy_vertex_base_2<Vbb>* is templated by a class *Vbb* which must be a model of the *SegmentDelaunayGraphVertexBase_2* concept.

```
#include <CGAL/Segment_Delaunay_graph_hierarchy_vertex_base_2.h>
```

Inherits From

Vbb

Is Model for the Concepts

SegmentDelaunayGraphHierarchyVertexBase_2

See Also

SegmentDelaunayGraphVertexBase_2
SegmentDelaunayGraphHierarchyVertexBase_2
SegmentDelaunayGraphDataStructure_2
CGAL::Segment_Delaunay_graph_vertex_base_2<Gt,SSTag>
CGAL::Triangulation_data_structure_2<Vb,Fb>
CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>

Chapter 27

2D Apollonius Graphs (Delaunay Graphs of Disks)

Menelaos Karavelas and Mariette Yvinec

Contents

27.1 Definitions	1699
27.2 Software Design	1701
27.3 The Geometric Traits	1702
27.4 The Apollonius Graph Hierarchy	1705
27.5 Examples	1705
27.5.1 First Example	1705
27.5.2 Second Example	1706
27.5.3 Third Example	1708
27.5.4 Fourth Example	1709

This chapter describes the two-dimensional Apollonius graph of CGAL. We start with a few definitions in Section [27.1](#). The software design of the 2D Apollonius graph package is described in Section [27.2](#). In Section [27.3](#) we discuss the geometric traits of the 2D Apollonius graph package and in Section [27.4](#) the Apollonius graph hierarchy, a data structure suitable for fast nearest neighbor queries, is briefly described.

27.1 Definitions

The 2D Apollonius graph class of CGAL is designed to compute the dual of the *Apollonius diagram* or, as it is also known, the *Additively weighted Voronoi diagram*. The algorithm that has been implemented is dynamic, which means that we can perform insertions and deletions on line. The corresponding CGAL class is called `Apollonius_graph_2<ApolloniusGraphTraits_2, ApolloniusGraphDataStructure_2>` and will be discussed in more detail in the sequel. The interested reader may want to refer to the paper by Karavelas and Yvinec [[KY02](#)] for the general idea as well as the details of the algorithm implemented.

Before describing the details of the implementation we make a brief introduction to the theory of Apollonius diagrams. The Apollonius diagram is defined over a set of sites $P_i = (c_i, w_i)$, $i = 1, \dots, n$, where c_i is the point and w_i the weight of P_i . It is a subdivision of the plane into connected regions, called *cells*, associated with the

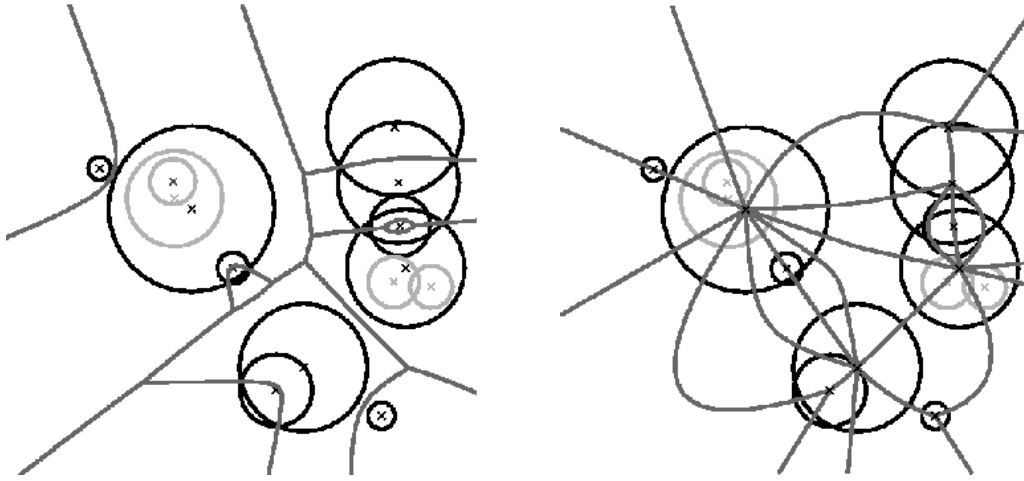


Figure 27.1: The Apollonius diagram (left) and its dual the Apollonius graph (right).

sites (see Fig. 27.1(left)). The cell of a site P_i is the locus of points on the plane that are closer to P_i than any other site P_j , $j \neq i$. The distance $\delta(x, P_i)$ of a point x in the plane to a site P_i is defined as:

$$\delta(x, P_i) = \|x - c_i\| - w_i,$$

where $\|\cdot\|$ denotes the Euclidean norm. It can easily be seen that it is a generalization of the Voronoi diagram for points, which can actually be obtained if all the weights w_i are equal. Unlike the case of points, however, it is possible that a site P_i might have an empty cell. This can also happen in the case of the power diagram, whose dual is the regular triangulation (see Section 20.6). If this is the case we call the site *hidden* (these are the black circles in Fig. 27.1). A site which is not hidden will be referred to as *visible*.

If all weights w_i are non-negative, the Apollonius diagram can be viewed as the Voronoi diagram of the set of circles $\{P_1, \dots, P_n\}$, where c_i is the center of the circle P_i and w_i its radius. If the weights are allowed to be negative, we need to go to 3D in order to explain what the Apollonius diagram means geometrically. We identify the 2D Euclidean plane with the xy -plane in 3D. Then the Voronoi diagram of a set of points can be seen as the vertical projection on the xy -plane of the lower envelope of a set of 3D cones defined as follows: for each point p in the set of 2D points we have a cone C_p whose apex is the point p . The axis of C_p is a line parallel to the z -axis passing through p , the angle of C_p is 45° and, finally C_p is facing in the positive z -direction (that is, C_p is contained in the positive z -halfspace). The Apollonius diagram corresponds to shifting the apexes of these cones in the z -direction by a quantity equal to the weight. Sites with negative weight will give rise to cones whose apex is in the negative z -halfspace and sites with positive weight will give rise to cones whose apex is in the positive z -halfspace. In a manner analogous to the case of points, the Apollonius diagram can then be defined as the vertical projection on the xy -plane of the lower envelope of the set of shifted cones. Notice that when all apexes are translated along the z -direction by the same amount, the projection of the lower envelope of the set of cones does not change. In particular, we can translate all cones by a large enough amount so that all apexes are in the positive z -halfspace. Algebraically, this means that the Apollonius diagram does not change if we add to all weights the same quantity, which in particular, implies that we can assume without loss of generality that all weights are positive. Given the observations above and in order to simplify our discussion of Apollonius diagrams, we will, from now on, assume that all weights are positive, and we will refer to the sites as circles.

The Apollonius diagram is a planar graph, and so is its dual, the Apollonius graph. There are many ways to embed it on the plane and one such way is shown in Fig. 27.1(right). The Apollonius graph is uniquely defined once we have the Apollonius diagram. If the circles are in *general position* (see precise definition below), then the Apollonius graph is a graph with triangular faces away from the convex hull of the set of circles (by

triangular we mean that every face has exactly three edges). Near the convex hull we may have some spikes (i.e., vertices of degree 1). To unify our approach and handling of the Apollonius graph we add to the set of (finite) circles a fictitious circle at infinity, which we call the *site at infinity*. We can then connect all vertices of the outer face of the Apollonius graph to the site at infinity which gives us a graph with the property that all of its faces are now triangular. However, the Apollonius graph is not a triangulation for two main reasons: we cannot always embed it on the plane with straight line segments that yield a triangulation and, moreover, we may have two faces of the graph that have two edges in common, which is not allowed in a triangulation. Both of these particularities appear when we consider the Apollonius graph of the set of circles in Fig. 27.1(right).

We would like to finish our brief introduction to the theory of Apollonius graphs by discussing the concept of general position. We say that a set of circles is in general position if no two triplets of circles have the same tritangent circle. This statement is rather technical and it is best understood in the context of points. The equivalent statement for points is that we have no two triplets of points that define the same circumcircle, or equivalently that no four points are co-circular. The statement about general position made above is a direct generalization of the (much simpler to understand) statement about points. On the contrary, when we have circles in degenerate position, the Apollonius graph has faces with more than three edges on their boundary. We can get a triangulated version of the graph by simply *triangulating* the corresponding faces in an arbitrary way. In fact the algorithm that has been implemented in CGAL has the property that it always returns a valid *triangulated* version of the Apollonius graph. By valid we mean that it contains the actual Apollonius graph (i.e., the actual dual of the Apollonius diagram) and whenever there are faces with more than three faces then they are triangulated. The way that they are triangulated depends on the order of insertion and deletion of the circles in the diagram.

One final point has to be made about hidden circles. First of all we would like to be more precise about our definition of hidden circles: we say that a circle is hidden if its cell has empty interior. This definition allows us to guarantee that all visible circles have cells that are two-dimensional regions. Geometrically the fact that a circle is hidden means that it is contained in the closure of the disk of another circle (see again Fig. 27.1). Note that a circle contained in the union of several disks, but not in the closure of any one of them, is not hidden.

Hidden circles pose an additional difficulty to our algorithm and software design. Since we allow circles to be inserted and deleted at wish, it is possible that a circle that was hidden at some point in time, may become visible at a later point in time; for example this can happen if we delete the circle that hides it. For this purpose we store hidden circles and have them reappear when they become visible. We will discuss this issue in detail below. For the time being it suffices to say that the user has the ability to control this behavior. More specifically it is possible to discard the circles that become hidden. This choice is totally natural when for example we expect to do only insertions, since in this case a circle that becomes hidden will never reappear. On the other hand if deletions are expected as well, then we lose the ability to have the hidden circles reappear.

Degenerate Dimensions. The dimension of the Apollonius graph is in general 2. The exceptions to this rule are as follows:

- The dimension is -1 if the Apollonius graph contains no circles.
- The dimension is 0 if the Apollonius graph contains exactly one visible circle.
- The dimension is 1 if the Apollonius graph contains exactly two visible circles.

27.2 Software Design

The 2D Apollonius graph class `Apollonius_graph_2<ApolloniusGraphTraits_2, ApolloniusGraphDataStructure_2>` follows the design of the triangulation package of CGAL. It is parametrized by two arguments:

- the **geometric traits** class. It provides the basic geometric objects involved in the algorithm, such as sites, points etc. It also provides the geometric predicates for the computation of the Apollonius graph, as well as some basic constructions that can be used, for example, to visualize the Apollonius graph or the Apollonius diagram. The geometric traits for the Apollonius graph will be discussed in more detail in the next section.
- the **Apollonius graph data structure**. This is essentially the same as the triangulation data structure (discussed in Chapter 21), augmented with some additional operations that are specific to Apollonius graphs. The corresponding concept is that of *ApolloniusGraphDataStructure_2*, which in fact is a refinement of the *TriangulationDataStructure_2* concept. The class *Triangulation_data_structure_2<Vb,Fb>* is a model of the concept *ApolloniusGraphDataStructure_2*. A default value for the corresponding template parameter is provided, so the user does not need to specify it.

Storing Hidden Sites. As we have already mentioned a circle is hidden if it is contained inside some visible circle. This creates a parent-child relationship between visible and hidden circles: the parent of a hidden circle is the visible circle that contains it. If more than one visible circles contain a hidden circle then the hidden circle can be assigned to any of the visible circles arbitrarily.

To store hidden circles we assign to every visible circle a list. This list comprises the hidden circles that are contained in the visible circle. The user can access the hidden circles associated with a visible circle through an iterator called *Hidden_sites_iterator*. This iterator is defined in the *ApolloniusGraphVertexBase_2* concept and is implemented by its model, the *Apollonius_graph_vertex_base_2<Gt,StoreHidden>* class. It is also possible to iterate through the entire set of hidden sites using an homonymous iterator defined by the *Apollonius_graph_2<Gt,Agds>* class.

Since storing hidden sites may not be of interest in some cases (e.g., for example this is the case if we only perform insertions in the Apollonius graph), the user has the possibility of controlling this behavior. More precisely, the class *Apollonius_graph_vertex_base_2<Gt,StoreHidden>* has two template parameters, the second of which is a boolean value. This value is by default *true* and it indicates that hidden sites should be stored. The user can indicate that hidden sites may be discarded by setting this value to *false*.

27.3 The Geometric Traits

The predicates required for the computation of the Apollonius graph are rather complicated. It is not the purpose of this document to discuss them in detail. The interested reader may refer to the papers by Karavelas and Emiris for the details [KE02, KE03]. However, we would like to give a brief overview of what they compute. There are several predicates needed by this algorithm. We will discuss the most important/complicated ones. It turns out that it is much easier to describe them in terms of the Apollonius diagram, rather than the Apollonius graph. Whenever it is applicable we will also describe their meaning in terms of the Apollonius graph.

The first two geometric predicates are called *Is_hidden_2* and *Oriented_side_of_bisector_2*. The first one involves two circles, say P_1 and P_2 . It determines if P_1 is hidden with respect to P_2 ; more precisely it checks whether the circle P_1 is contained in the closure of the disk defined by the circle P_2 . As its name indicates, it determines if a circle is hidden or not. The second predicate involves two circles P_1 and P_2 and a point q . It answers the question whether q is closer to P_1 or P_2 . Its name stems from the fact that answering the aforementioned question is equivalent to determining the oriented side of the bisector of P_1 and P_2 that contains the query point q . This predicate is used by the algorithm for closest neighbor queries for points.

The next geometric predicate is called *Vertex_conflict_2* and it involves four circles P_1 , P_2 , P_3 , and P_4 (see Fig. 27.3). The first three (red circles in Fig. 27.3) define a tritangent circle (yellow circle in Fig. 27.3). What we want to determine is the sign of the distance of the green circle from the yellow circle. The distance between

two circles $K_1 = (c_1, r_1)$ and $K_2 = (c_2, r_2)$ is defined as the distance of their centers minus their radii:

$$\delta(K_1, K_2) = \|c_1 - c_2\| - r_1 - r_2.$$

This predicate determines if a vertex in the Apollonius diagram (the center of the yellow circle) is destroyed when a new circle is inserted in the diagram (the green circle). In the Apollonius graph it tells us if a triangular face of the diagram is to be destroyed or not.

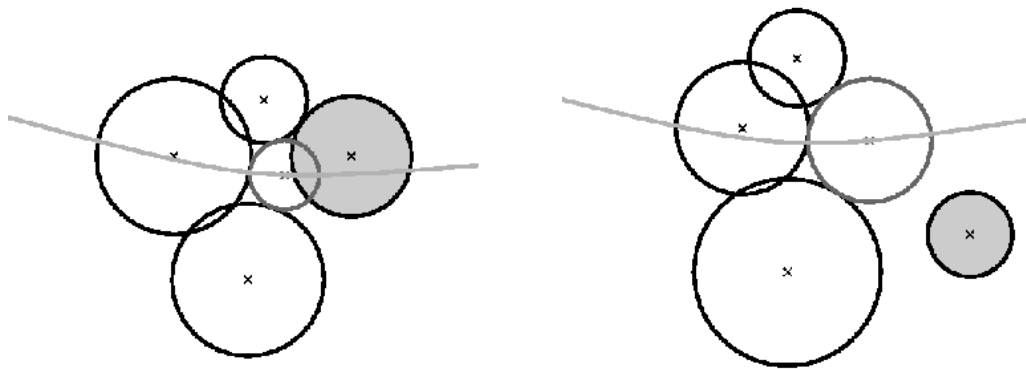


Figure 27.2: The *Vertex_conflict_2* predicate. The left-most, bottom-most and top-most circles define the tritangent circle in the middle. We want to determine the sign of the distance of the left-most circle from the one in the middle. The almost horizontal curve is the bisector of the top-most and bottom-most circles. Left: the predicate returns *CGAL::NEGATIVE*. Right: the predicate returns *CGAL::POSITIVE*.

What we essentially want to compute when we construct incrementally a Voronoi diagram, is whether the object to be inserted destroys an edge of the Voronoi diagram or not. In the case of points this is really easy and it amounts to the well known *incircle* test. In the case of circles the situation is more complicated. We can have six possible outcomes as to what portion of an edge of the Apollonius diagram the new circle destroys (see Fig. 27.3). The first two can be answered directly by the *Vertex_conflict_2* predicate evaluated for the two endpoints of the Apollonius diagram edge. This is due to the fact that the value of the *Vertex_conflict_2* predicate is different for the two endpoints. If the two values are the same then we need an additional test which determines if the interior of the Apollonius diagram edge is destroyed by the new circle. This is what the *Finite_edge_interior_conflict_2* and *Infinite_edge_interior_conflict_2* predicates do. In essence, it is the same predicate (same idea) applied to two different types of edges in the Apollonius diagram: a finite or an infinite edge. An edge is infinite if its dual edge in the Apollonius graph connects the site at infinity with the vertex corresponding to a (finite) circle; otherwise it is a finite edge.

The last predicate that we want to discuss is called *Is_degenerate_edge_2*. It tells us whether an edge in the Apollonius diagram is degenerate, that is if its two endpoints coincide. In the Apollonius graph such an edge corresponds to one of the additional edges that we use to triangulate the non-triangular faces.

The afore mentioned predicates are part of the *ApolloniusGraphTraits_2* concept of CGAL. CGAL also provides a model for this concept, the *Apollonius_graph_traits_2<K,Method_tag>* class. The first template parameter of this class must be a model of the *Kernel* concept. The second template parameter is a tag that indicates what operations are allowed in the computations that take place within the traits class. The two possible values of the *Method_tag* parameter are *CGAL::Ring_tag* and *CGAL::Sqrt_field_tag*. When *CGAL::Ring_tag* is used, only ring operations are used during the evaluation of the predicates, whereas if *CGAL::Sqrt_field_tag* is chosen, all four field operations, as well as square roots, are used during the predicate evaluation.

The *Apollonius_graph_traits_2<K,Method_tag>* class provides exact predicates if the number type in the kernel

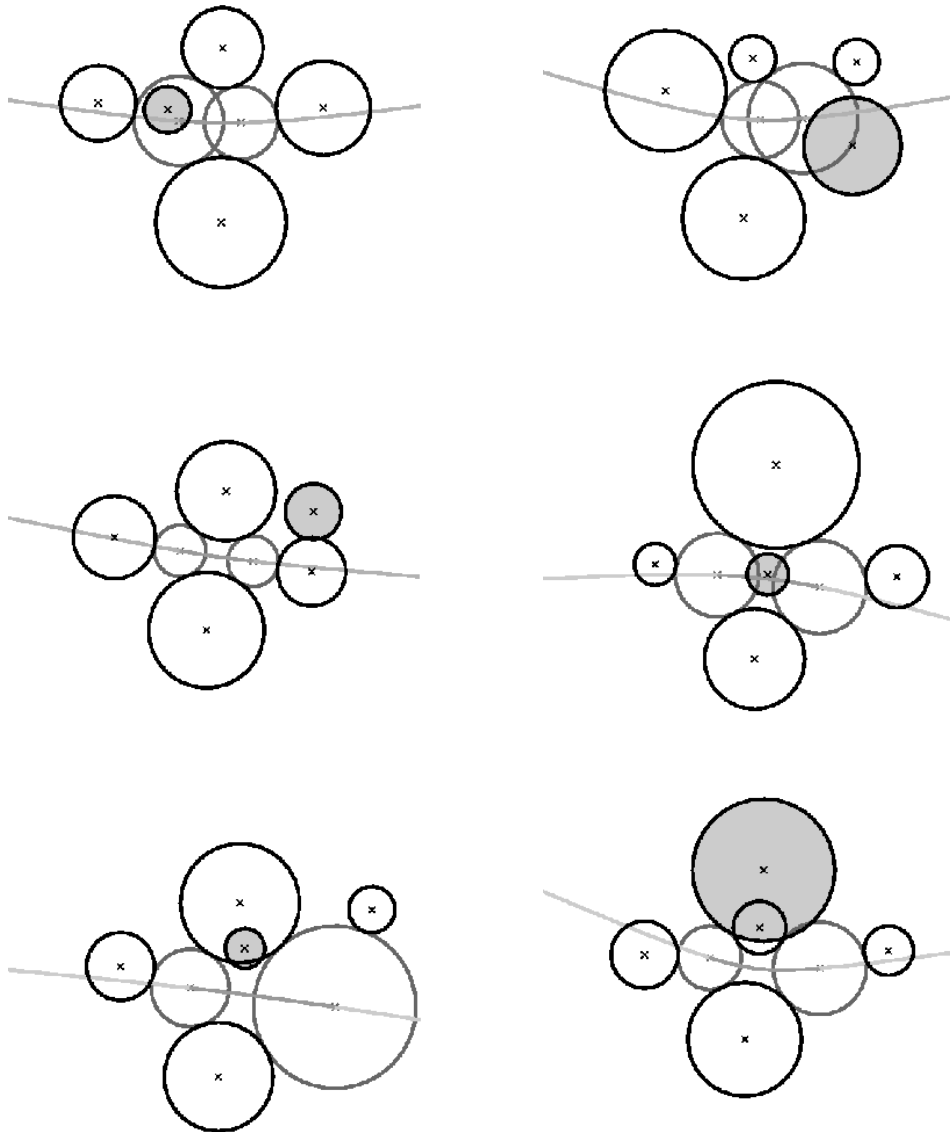


Figure 27.3: The 6 possible outcomes of the *Finite_edge_interior_conflict_2* predicate. Top left: only a neighborhood around the left-most endpoint of the edge will be destroyed. Top right: only a neighborhood around the right-most endpoint of the edge will be destroyed. Middle left: no portion of the edge is destroyed. Middle right: the entire edge will be destroyed. Bottom left: a neighborhood in the interior of the edge will be destroyed; the regions near the endpoints remain unaffected. Bottom right: The neighborhood around the two endpoints will be destroyed, but an interval in the interior of the edge will remain in the new diagram.

K is an exact number type. This is to be associated with the type of operations allowed for the predicate evaluation. For example `CGAL::MP_Float` as number type, with `CGAL::Ring_tag` as tag will give exact predicates, whereas `CGAL::MP_Float` with `CGAL::Sqrt_field_tag` will give inexact predicates.

Since using an exact number type may be too slow, the `Apollonius_graph_traits_2<K,Method_tag>` class is designed to support the dynamic filtering of CGAL through the `CGAL::Filtered_exact<CT,ET>` mechanism. In

particular if CT is an inexact number type that supports the operations denoted by the tag *Method_tag* and ET is an exact number type for these operations, then kernel with number type `CGAL::Filtered_exact<CT,ET>` will yield exact predicates for the Apollonius graph traits. To give a concrete example, `CGAL::Filtered_exact<double,CGAL::MP_Float>` with `CGAL::Ring_tag` will produce exact predicates.

Another possibility for fast and exact predicate evaluation is to use the `Apollonius_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>` class. This class is the analog of a filtered kernel. It takes a constructions kernel CK , a filtering kernel FK and an exact kernel EK , as well as the corresponding tags (CM , FM and EM , respectively). It evaluates the predicates by first using the filtering kernel, and if this fails the evaluation is performed using the exact kernel. The constructions are done using the kernel CK , which means that they are not necessarily exact. All template parameters except CK have default values, which are explained in the reference manual.

27.4 The Apollonius Graph Hierarchy

The `Apollonius_graph_hierarchy_2<ApolloniusGraphTraits_2,ApolloniusGraphDataStructure_2>` class is nothing but the equivalent of the `Triangulation_hierarchy_2` class, applied to the Apollonius graph. It consists of a series of Apollonius graphs constructed in a manner analogous to the Delaunay hierarchy by Devillers [Dev98]. The class `Apollonius_graph_hierarchy_2<ApolloniusGraphTraits_2,ApolloniusGraphDataStructure_2>` has exactly the same interface and functionality as the `Apollonius_graph_2<ApolloniusGraphTraits_2,ApolloniusGraphDataStructure_2>` class. Using the Apollonius graph hierarchy involves an additional cost in space and time for maintaining the hierarchy. Our experiments have shown that it usually pays off to use the hierarchy for inputs consisting of more than 1,000 circles. This threshold holds for both the construction of the Apollonius diagram itself, as well as for nearest neighbor queries.

27.5 Examples

27.5.1 First Example

```
// file: examples/Apollonius_graph_2/ag2_exact_traits.C

#include <CGAL/basic.h>

// standard includes
#include <iostream>
#include <fstream>
#include <cassert>

// the number type
#include <CGAL/MP_Float.h>

// example that uses an exact number type

typedef CGAL::MP_Float NT;

// choose the kernel
#include <CGAL/Simple_cartesian.h>

typedef CGAL::Simple_cartesian<NT> Kernel;
```

```

// typedefs for the traits and the algorithm

#include <CGAL/Apollonius_graph_2.h>
#include <CGAL/Apollonius_graph_traits_2.h>

typedef CGAL::Apollonius_graph_traits_2<Kernel> Traits;
typedef CGAL::Apollonius_graph_2<Traits> Apollonius_graph;

int main()
{
    std::ifstream ifs("data/sites.cin");
    assert( ifs );

    Apollonius_graph ag;
    Apollonius_graph::Site_2 site;

    // read the sites and insert them in the Apollonius graph
    while ( ifs >> site ) {
        ag.insert(site);
    }

    // validate the Apollonius graph
    assert( ag.is_valid(true, 1) );
    std::cout << std::endl;

    return 0;
}

```

27.5.2 Second Example

```

// file: examples/Apollonius_graph_2/ag2_exact_traits_sqrt.C

#include <CGAL/basic.h>

// standard includes
#include <iostream>
#include <fstream>
#include <cassert>

#if defined CGAL_USE_LEDA
#   include <CGAL/leda_real.h>
#elif defined CGAL_USE_CORE
#   include <CGAL/CORE_Expr.h>
#endif

#if defined CGAL_USE_LEDA

```

```

// If LEDA is present use leda_real as the exact number type
typedef leda_real NT;

#elif defined CGAL_USE_CORE
// Otherwise if CORE is present use CORE's Expr as the exact number type
typedef CORE::Expr NT;

#else

// Otherwise just use double. This may cause numerical errors but it
// is still worth doing it to show how to define correctly the traits
// class
typedef double NT;

#endif

#include <CGAL/Simple_cartesian.h>

typedef CGAL::Simple_cartesian<NT> Kernel;

// typedefs for the traits and the algorithm

#include <CGAL/Apollonius_graph_2.h>
#include <CGAL/Apollonius_graph_traits_2.h>

// the traits class is now going to assume that the operations
// +,-,*,/ and sqrt are supported exactly
typedef
CGAL::Apollonius_graph_traits_2<Kernel,CGAL::Sqrt_field_tag> Traits;

typedef CGAL::Apollonius_graph_2<Traits> Apollonius_graph;

int main()
{
    std::ifstream ifs("data/sites.cin");
    assert( ifs );

    Apollonius_graph ag;
    Apollonius_graph::Site_2 site;

    // read the sites and insert them in the Apollonius graph
    while ( ifs >> site ) {
        ag.insert(site);
    }

    // validate the Apollonius graph
    assert( ag.is_valid(true, 1) );
    std::cout << std::endl;

    return 0;
}

```

```
}
```

27.5.3 Third Example

```
// file: examples/Apollonius_graph_2/ag2_filtered_traits_no_hidden.C

#include <CGAL/basic.h>

// standard includes
#include <iostream>
#include <fstream>
#include <cassert>

#include <CGAL/basic.h>

// example that uses the filtered traits

// choose the representation
#include <CGAL/Simple_cartesian.h>

typedef CGAL::Simple_cartesian<double> Rep;

#include <CGAL/Apollonius_graph_2.h>
#include <CGAL/Triangulation_data_structure_2.h>
#include <CGAL/Apollonius_graph_vertex_base_2.h>
#include <CGAL/Triangulation_face_base_2.h>
#include <CGAL/Apollonius_graph_filtered_traits_2.h>

// typedef for the traits; the filtered traits class is used
typedef CGAL::Apollonius_graph_filtered_traits_2<Rep> Traits;

// typedefs for the algorithm

// With the second template argument in the vertex base class being
// false, we indicate that there is no need to store the hidden sites.
// One case where this is indeed not needed is when we only do
// insertions, like in the main program below.
typedef CGAL::Apollonius_graph_vertex_base_2<Traits,false> Vb;
typedef CGAL::Triangulation_face_base_2<Traits> Fb;
typedef CGAL::Triangulation_data_structure_2<Vb,Fb> Agds;
typedef CGAL::Apollonius_graph_2<Traits,Agds> Apollonius_graph;

int main()
{
    std::ifstream ifs("data/sites.cin");
    assert( ifs );

    Apollonius_graph ag;
    Apollonius_graph::Site_2 site;

    // read the sites and insert them in the Apollonius graph
    while ( ifs >> site ) {
```

```

    ag.insert(site);
}

// validate the Apollonius graph
assert( ag.is_valid(true, 1) );
std::cout << std::endl;

// now remove all sites
std::cout << "Removing all sites... " << std::flush;
while ( ag.number_of_vertices() > 0 ) {
    ag.remove( ag.finite_vertex() );
}
std::cout << "done!" << std::endl << std::endl;

return 0;
}

```

27.5.4 Fourth Example

```

// file: examples/Apollonius_graph_2/ag2_hierarchy.C

#include <CGAL/basic.h>

// standard includes
#include <iostream>
#include <fstream>
#include <cassert>

// example that uses the filtered traits

#include <CGAL/MP_Float.h>
#include <CGAL/Simple_cartesian.h>

// constructions kernel (inexact)
typedef CGAL::Simple_cartesian<double> CK;

// exact kernel
typedef CGAL::Simple_cartesian<CGAL::MP_Float> EK;

// typedefs for the traits and the algorithm

#include <CGAL/Apollonius_graph_hierarchy_2.h>
#include <CGAL/Apollonius_graph_filtered_traits_2.h>

// Type definition for the traits class.
// In this example we explicitly define the exact kernel. We also
// explicitly define what operations to use for the evaluation of the
// predicates and constructions, when the filtering and the exact
// kernels are used respectively.

```

```

// Note that the operations allowed for the filtering and the
// constructions (field operations plus square roots) are different
// from the operations allowed when the exact kernel is used (ring
// operations).
typedef CGAL::Sqrt_field_tag CM;
typedef CGAL::Ring_tag EM;
typedef CGAL::Apollonius_graph_filtered_traits_2<CK,CM,EK,EM> Traits;

// Now we use the Apollonius graph hierarchy.
// The hierarchy is faster for inputs consisting of about more than
// 1,000 sites
typedef CGAL::Apollonius_graph_hierarchy_2<Traits> Apollonius_graph;

int main()
{
    std::ifstream ifs("data/hierarchy.cin");
    assert( ifs );

    Apollonius_graph ag;
    Apollonius_graph::Site_2 site;

    // read the sites and insert them in the Apollonius graph
    while ( ifs >> site ) {
        ag.insert(site);
    }

    // validate the Apollonius graph
    assert( ag.is_valid(true, 1) );

    return 0;
}

```

2D Apollonius Graphs (Delaunay Graphs of Disks)

Reference Manual

Menelaos Karavelas and Mariette Yvinec

An Apollonius graph is the dual of the Apollonius diagram, also known as the *additively weighted Voronoi diagram*. It is essentially the Voronoi diagram of a set of disks, where the distance of a point of the plane from a disk is defined as the Euclidean distance of the point and the center of the circle, minus the radius of the disk.

CGAL provides the class `CGAL::Apollonius_graph_2<Gt,Agds>` for computing the 2D Apollonius graph. The two template parameters must be models of the `ApolloniusGraphTraits_2` and `ApolloniusGraphDataStructure_2` concepts. The first concept is related to the geometric objects and predicates associated with Apollonius graphs, whereas the second concept refers to the data structure used to represent the Apollonius graph. The classes `Apollonius_graph_traits_2<K,Method_tag>` and `Triangulation_data_structure_2<Vb,Fb>` are models of the afore-mentioned concepts.

27.6 Classified Reference Pages

Concepts

<code>ApolloniusSite_2</code>	page 1720
<code>ApolloniusGraphDataStructure_2</code>	page 1722
<code>ApolloniusGraphVertexBase_2</code>	page 1724
<code>ApolloniusGraphTraits_2</code>	page 1727

<code>ApolloniusGraphHierarchyVertexBase_2</code>	page 1736
---------------------------------------------------------	---------------------------

Classes

<code>CGAL::Apollonius_graph_2<Gt,Agds></code>	page 1713
<code>CGAL::Apollonius_site_2<K></code>	page 1721
<code>CGAL::Apollonius_graph_vertex_base_2<Gt,StoreHidden></code>	page 1726
<code>CGAL::Apollonius_graph_traits_2<K,Method_tag></code>	page 1731
<code>CGAL::Apollonius_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM></code>	page 1732

<i>CGAL::Apollonius_graph_hierarchy_2<Gt,Agds></i>	page 1733
<i>CGAL::Apollonius_graph_hierarchy_vertex_base_2<Agvb></i>	page 1738

27.7 Alphabetical List of Reference Pages

<i>ApolloniusGraphDataStructure_2</i>	page 1722
<i>ApolloniusGraphHierarchyVertexBase_2</i>	page 1736
<i>ApolloniusGraphTraits_2</i>	page 1727
<i>ApolloniusGraphVertexBase_2</i>	page 1724
<i>ApolloniusSite_2</i>	page 1720
<i>Apollonius_graph_2<Gt,Agds></i>	page 1713
<i>Apollonius_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM></i>	page 1732
<i>Apollonius_graph_hierarchy_2<Gt,Agds></i>	page 1733
<i>Apollonius_graph_hierarchy_vertex_base_2<Agvb></i>	page 1738
<i>Apollonius_graph_traits_2<K,Method_tag></i>	page 1731
<i>Apollonius_graph_vertex_base_2<Gt,StoreHidden></i>	page 1726
<i>Apollonius_site_2<K></i>	page 1721

CGAL::Apollonius_graph_2<Gt,Agds>

Definition

The class *Apollonius_graph_2*<*Gt*,*Agds*> represents the Apollonius graph. It supports insertions and deletions of sites. It is templated by two template arguments *Gt*, which must be a model of *ApolloniusGraphTraits_2*, and *Agds*, which must be a model of *ApolloniusGraphDataStructure_2*. The second template argument defaults to *CGAL::Triangulation_data_structure_2*< *CGAL::Apollonius_graph_vertex_base_2*<*Gt*,*true*>, *CGAL::Triangulation_face_base_2*<*Gt*> >.

```
#include <CGAL/Apollonius_graph_2.h>
```

Is Model for the Concepts

DelaunayGraph_2

Types

<i>typedef Agds</i>	<i>Data_structure;</i>	A type for the underlying data structure.
<i>typedef Data_structure</i>	<i>Triangulation_data_structure;</i>	Same as the <i>Data_structure</i> type. This type has been introduced in order for the <i>Apollonius_graph_2</i> < <i>Gt</i> , <i>Agds</i> > class to be a model of the <i>DelaunayGraph_2</i> concept.
<i>typedef Gt</i>	<i>Geom_traits;</i>	A type for the geometric traits.
<i>typedef Gt::Point_2</i>	<i>Point_2;</i>	A type for the point defined in the geometric traits.
<i>typedef Gt::Site_2</i>	<i>Site_2;</i>	A type for the Apollonius site, defined in the geometric traits.

The vertices and faces of the Apollonius graph are accessed through *handles*, *iterators* and *circulators*. The iterators and circulators are all bidirectional and non-mutable. The circulators and iterators are assignable to the corresponding handle types, and they are also convertible to the corresponding handles. The edges of the Apollonius graph can also be visited through iterators and circulators, the edge circulators and iterators are also bidirectional and non-mutable. In the following, we call *infinite* any face or edge incident to the infinite vertex and the infinite vertex itself. Any other feature (face, edge or vertex) of the Apollonius graph is said to be *finite*. Some iterators (the *All* iterators) allow to visit finite or infinite features while the others (the *Finite* iterators) visit only finite features. Circulators visit both infinite and finite features.

<i>typedef Data_structure::Edge</i>	<i>Edge;</i>	the edge type. The <i>Edge(f,i)</i> is the edge common to faces <i>f</i> and <i>f.neighbor(i)</i> . It is also the edge joining the vertices <i>vertex(cw(i))</i> and <i>vertex(ccw(i))</i> of <i>f</i> . <i>Precondition:</i> <i>i</i> must be 0, 1 or 2.
<i>typedef Data_structure::Vertex</i>	<i>Vertex;</i>	A type for a vertex.
<i>typedef Data_structure::Face</i>	<i>Face;</i>	A type for a face.
<i>typedef Data_structure::Vertex_handle</i>	<i>Vertex_handle;</i>	A type for a handle to a vertex.
<i>typedef Data_structure::Face_handle</i>	<i>Face_handle;</i>	A type for a handle to a face.
<i>typedef Data_structure::Vertex_circulator</i>	<i>Vertex_circulator;</i>	A type for a circulator over vertices incident to a given vertex.

<i>typedef Data_structure::Face_circulator</i>	<i>Face_circulator;</i>	A type for a circulator over faces incident to a given vertex.
<i>typedef Data_structure::Edge_circulator</i>	<i>Edge_circulator;</i>	A type for a circulator over edges incident to a given vertex.
<i>typedef Data_structure::Vertex_iterator</i>	<i>All_vertices_iterator;</i>	A type for an iterator over all vertices.
<i>typedef Data_structure::Face_iterator</i>	<i>All_faces_iterator;</i>	A type for an iterator over all faces.
<i>typedef Data_structure::Edge_iterator</i>	<i>All_edges_iterator;</i>	A type for an iterator over all edges.
<i>typedef Data_structure::size_type</i>	<i>size_type;</i>	An unsigned integral type.
<i>Apollonius_graph_2<Gt,Agds>:: Finite_vertices_iterator;</i>		A type for an iterator over finite vertices.
<i>Apollonius_graph_2<Gt,Agds>:: Finite_faces_iterator;</i>		A type for an iterator over finite faces.
<i>Apollonius_graph_2<Gt,Agds>:: Finite_edges_iterator;</i>		A type for an iterator over finite edges.

In addition to iterators and circulators for vertices and faces, iterators for sites are provided. In particular there are iterators for the entire set of sites, the hidden sites and the visible sites of the Apollonius graph.

<i>Apollonius_graph_2<Gt,Agds>:: Sites_iterator</i>	A type for an iterator over all sites.
<i>Apollonius_graph_2<Gt,Agds>:: Visible_sites_iterator</i>	A type for an iterator over all visible sites.
<i>Apollonius_graph_2<Gt,Agds>:: Hidden_sites_iterator</i>	A type for an iterator over all hidden sites.

Creation

Apollonius_graph_2<Gt,Agds> ag(Gt gt=Gt()); Creates an Apollonius graph using *gt* as geometric traits.

template< class Input_iterator >

Apollonius_graph_2<Gt,Agds> ag(Input_iterator first, Input_iterator beyond, Gt gt=Gt());

Creates an Apollonius graph using *gt* as geometric traits and inserts all sites in the range [*first*, *beyond*).

Precondition: *Input_iterator* must be a model of *InputIterator*. The value type of *Input_iterator* must be *Site_2*.

Apollonius_graph_2<Gt,Agds> ag(other); Copy constructor. All faces and vertices are duplicated. After the construction, *ag* and *other* refer to two different Apollonius graphs : if *other* is modified, *ag* is not.

Apollonius_graph_2<Gt,Agds> ag = other Assignment. If *ag* and *other* are the same object nothing is done. Otherwise, all the vertices and faces are duplicated. After the assignment, *ag* and *other* refer to different Apollonius graphs : if *other* is modified, *ag* is not.

Access Functions

<i>Geom_traits</i>	<i>ag.geom_traits()</i>	Returns a reference to the Apollonius graph traits object.
<i>Data_structure</i>	<i>ag.data_structure()</i>	Returns a reference to the underlying data structure.
<i>Data_structure</i>	<i>ag.tds()</i>	Same as <i>data_structure()</i> . This method has been added in compliance with the <i>DelaunayGraph_2</i> concept.
<i>int</i>	<i>ag.dimension()</i>	Returns the dimension of the Apollonius graph.
<i>size_type</i>	<i>ag.number_of_vertices()</i>	Returns the number of finite vertices.
<i>size_type</i>	<i>ag.number_of_visible_sites()</i>	Returns the number of visible sites.

<i>size_type</i>	<i>ag.number_of_hidden_sites()</i>	Returns the number of hidden sites.
<i>size_type</i>	<i>ag.number_of_faces()</i>	Returns the number of faces (both finite and infinite) of the Apollonius graph.
<i>Face_handle</i>	<i>ag.infinite_face()</i>	Returns a face incident to the <i>infinite_vertex</i> .
<i>Vertex_handle</i>	<i>ag.infinite_vertex()</i>	Returns the <i>infinite_vertex</i> .
<i>Vertex_handle</i>	<i>ag.finite_vertex()</i>	Returns a vertex distinct from the <i>infinite_vertex</i> . <i>Precondition:</i> The number of (visible) vertices in the Apollonius graph must be at least one.

Traversal of the Apollonius graph

An Apollonius graph can be seen as a container of faces and vertices. Therefore the Apollonius graph provides several iterators and circulators that allow to traverse it (completely or partially).

Face, Edge and Vertex Iterators

The following iterators allow respectively to visit finite faces, finite edges and finite vertices of the Apollonius graph. These iterators are non-mutable, bidirectional and their value types are respectively *Face*, *Edge* and *Vertex*. They are all invalidated by any change in the Apollonius graph.

<i>Finite_vertices_iterator</i>	<i>ag.finite_vertices_begin()</i>	Starts at an arbitrary finite vertex.
<i>Finite_vertices_iterator</i>	<i>ag.finite_vertices_end()</i>	Past-the-end iterator.
<i>Finite_edges_iterator</i>	<i>ag.finite_edges_begin()</i>	Starts at an arbitrary finite edge.
<i>Finite_edges_iterator</i>	<i>ag.finite_edges_end()</i>	Past-the-end iterator.
<i>Finite_faces_iterator</i>	<i>ag.finite_faces_begin()</i>	Starts at an arbitrary finite face.
<i>Finite_faces_iterator</i>	<i>ag.finite_faces_end()</i>	Past-the-end iterator.

The following iterators allow respectively to visit all (both finite and infinite) faces, edges and vertices of the Apollonius graph. These iterators are non-mutable, bidirectional and their value types are respectively *Face*, *Edge* and *Vertex*. They are all invalidated by any change in the Apollonius graph.

<i>All_vertices_iterator</i>	<i>ag.all_vertices_begin()</i>	Starts at an arbitrary vertex.
<i>All_vertices_iterator</i>	<i>ag.all_vertices_end()</i>	Past-the-end iterator.
<i>All_edges_iterator</i>	<i>ag.all_edges_begin()</i>	Starts at an arbitrary edge.
<i>All_edges_iterator</i>	<i>ag.all_edges_end()</i>	Past-the-end iterator.
<i>All_faces_iterator</i>	<i>ag.all_faces_begin()</i>	Starts at an arbitrary face.
<i>All_faces_iterator</i>	<i>ag.all_faces_end()</i>	Past-the-end iterator.

Site iterators

The following iterators allow respectively to visit all sites, the visible sites and the hidden sites. These iterators are non-mutable, bidirectional and their value type is *Site_2*. They are all invalidated by any change in the Apollonius graph.

<i>Sites_iterator</i>	<i>ag.sites_begin()</i>	Starts at an arbitrary site.
-----------------------	-------------------------	------------------------------

<i>Sites_iterator</i>	<i>ag.sites_end()</i>	Past-the-end iterator.
<i>Visible_sites_iterator</i>	<i>ag.visible_sites_begin()</i>	Starts at an arbitrary visible site.
<i>Visible_sites_iterator</i>	<i>ag.visible_sites_end()</i>	Past-the-end iterator.
<i>Hidden_sites_iterator</i>	<i>ag.hidden_sites_begin()</i>	Starts at an arbitrary hidden site.
<i>Hidden_sites_iterator</i>	<i>ag.hidden_sites_end()</i>	Past-the-end iterator.

Face, Edge and Vertex Circulators

The Apollonius graph also provides circulators that allow to visit respectively all faces or edges incident to a given vertex or all vertices adjacent to a given vertex. These circulators are non-mutable and bidirectional. The operator *operator++* moves the circulator counterclockwise around the vertex while the *operator--* moves clockwise. A face circulator is invalidated by any modification of the face pointed to. An edge circulator is invalidated by any modification in one of the two faces incident to the edge pointed to. A vertex circulator is invalidated by any modification in any of the faces adjacent to the vertex pointed to.

<i>Face_circulator</i>	<i>ag.incident_faces(Vertex_handle v)</i>	Starts at an arbitrary face incident to <i>v</i> .
<i>Face_circulator</i>	<i>ag.incident_faces(Vertex_handle v, Face_handle f)</i>	Starts at face <i>f</i> . Precondition: Face <i>f</i> is incident to vertex <i>v</i> .
<i>Edge_circulator</i>	<i>ag.incident_edges(Vertex_handle v)</i>	Starts at an arbitrary edge incident to <i>v</i> .
<i>Edge_circulator</i>	<i>ag.incident_edges(Vertex_handle v, Face_handle f)</i>	Starts at the first edge of <i>f</i> incident to <i>v</i> , in counterclockwise order around <i>v</i> . Precondition: Face <i>f</i> is incident to vertex <i>v</i> .
<i>Vertex_circulator</i>	<i>ag.incident_vertices(Vertex_handle v)</i>	Starts at an arbitrary vertex incident to <i>v</i> .
<i>Vertex_circulator</i>	<i>ag.incident_vertices(Vertex_handle v, Face_handle f)</i>	Starts at the first vertex of <i>f</i> adjacent to <i>v</i> in counterclockwise order around <i>v</i> . Precondition: Face <i>f</i> is incident to vertex <i>v</i> .

Traversal of the Convex Hull

Applied on the *infinite_vertex* the above functions allow to visit the vertices on the convex hull and the infinite edges and faces. Note that a counterclockwise traversal of the vertices adjacent to the *infinite_vertex* is a clockwise traversal of the convex hull.

<i>Vertex_circulator</i>	<i>ag.incident_vertices(ag.infinite_vertex())</i>
<i>Vertex_circulator</i>	<i>ag.incident_vertices(ag.infinite_vertex(), Face_handle f)</i>
<i>Face_circulator</i>	<i>ag.incident_faces(ag.infinite_vertex())</i>

<i>Face_circulator</i>	<i>ag.incident_faces(ag.infinite_vertex(), Face_handle f)</i>
<i>Edge_circulator</i>	<i>ag.incident_edges(ag.infinite_vertex())</i>
<i>Edge_circulator</i>	<i>ag.incident_edges(ag.infinite_vertex(), Face_handle f)</i>

Predicates

The class *Apollonius_graph_2<Gt,Agds>* provides methods to test the finite or infinite character of any feature.

<i>bool</i>	<i>ag.is_infinite(Vertex_handle v)</i>	<i>true</i> , iff <i>v</i> is the <i>infinite_vertex</i> .
<i>bool</i>	<i>ag.is_infinite(Face_handle f)</i>	<i>true</i> , iff face <i>f</i> is infinite.
<i>bool</i>	<i>ag.is_infinite(Face_handle f, int i)</i>	<i>true</i> , iff edge (<i>f</i> , <i>i</i>) is infinite.
<i>bool</i>	<i>ag.is_infinite(Edge e)</i>	<i>true</i> , iff edge <i>e</i> is infinite.
<i>bool</i>	<i>ag.is_infinite(Edge_circulator ec)</i>	<i>true</i> , iff edge <i>*ec</i> is infinite.

Insertion

```
template< class Input_iterator >
unsigned int    ag.insert( Input_iterator first, Input_iterator beyond)
```

Inserts the sites in the range [*first*,*beyond*). The number of sites in the range [*first*, *beyond*) is returned.

Precondition: *Input_iterator* must be a model of *InputIterator* and its value type must be *Site_2*.

<i>Vertex_handle</i>	<i>ag.insert(Site_2 s)</i>	Inserts the site <i>s</i> in the Apollonius graph. If <i>s</i> is visible then the vertex handle of <i>s</i> is returned, otherwise <i>Vertex_handle(NULL)</i> is returned.
----------------------	-----------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<i>Vertex_handle</i>	<i>ag.insert(Site_2 s, Vertex_handle vnear)</i>
----------------------	--------------------------------------------------

Inserts *s* in the Apollonius graph using the site associated with *vnear* as an estimate for the nearest neighbor of the center of *s*. If *s* is visible then the vertex handle of *s* is returned, otherwise *Vertex_handle(NULL)* is returned.

Removal

<i>void</i>	<i>ag.remove(Vertex_handle v)</i>	Removes the site associated to the vertex handle <i>v</i> from the Apollonius graph. <i>Precondition:</i> <i>v</i> must correspond to a valid finite vertex of the Apollonius graph.
-------------	------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Nearest neighbor location

<i>Vertex_handle</i>	<i>ag.nearest_neighbor(Point_2 p)</i>	Finds the nearest neighbor of the point <i>p</i> . In other words it finds the site whose Apollonius cell contains <i>p</i> . Ties are broken arbitrarily and one of the nearest neighbors of <i>p</i> is returned. If there are no visible sites in the Apollonius diagram <i>Vertex_handle(NULL)</i> is returned.
----------------------	----------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Vertex_handle *ag.nearest_neighbor(Point_2 p, Vertex_handle vnear)*

Finds the nearest neighbor of the point p using the site associated with $vnear$ as an estimate for the nearest neighbor of p . Ties are broken arbitrarily and one of the nearest neighbors of p is returned. If there are no visible sites in the Apollonius diagram *Vertex_handle(NULL)* is returned.

Access to the dual

The *Apollonius_graph_2* class provides access to the duals of the faces of the graph. The dual of a face of the Apollonius graph is a site. If the originating face is infinite, its dual is a site with center at infinity (or equivalently with infinite weight), which means that it can be represented geometrically as a line. If the originating face is finite, its dual is a site with finite center and weight. In the following three methods the returned object is assignable to either *Site_2* or *Gt::Line_2*, depending on whether the corresponding face of the Apollonius graph is finite or infinite, respectively.

<i>Gt::Object_2</i> <i>ag.dual(Face_handle f)</i>	Returns the dual corresponding to the face handle f . The returned object can be assignable to one of the following: <i>Site_2</i> , <i>Gt::Line_2</i> .
<i>Gt::Object_2</i> <i>ag.dual(All_faces_iterator it)</i>	Returns the dual of the face to which it points to. The returned object can be assignable to one of the following: <i>Site_2</i> , <i>Gt::Line_2</i> .
<i>Gt::Object_2</i> <i>ag.dual(Finite_faces_iterator it)</i>	Returns the dual of the face to which it points to. The returned object can be assignable to one of the following: <i>Site_2</i> , <i>Gt::Line_2</i> .

I/O

<pre>template< class Stream > Stream& ag.draw_primal(Stream& str)</pre>	<p>Draws the Apollonius graph to the stream str. <i>Precondition:</i> The following operators must be defined: <i>Stream& operator<<(Stream&, Gt::Segment_2),</i> <i>Stream& operator<<(Stream&, Gt::Ray_2).</i></p>
<pre>template < class Stream > Stream& ag.draw_dual(Stream& str)</pre>	<p>Draws the dual of the Apollonius graph, i.e., the Apollonius diagram, to the stream str. <i>Precondition:</i> The following operators must be defined: <i>Stream& operator<<(Stream&, Gt::Segment_2),</i> <i>Stream& operator<<(Stream&, Gt::Ray_2),</i> <i>Stream& operator<<(Stream&, Gt::Line_2).</i></p>
<pre>template< class Stream > Stream& ag.draw_primal_edge(Edge e, Stream& str)</pre>	<p>Draws the edge e of the Apollonius graph to the stream str. <i>Precondition:</i> The following operators must be defined: <i>Stream& operator<<(Stream&, Gt::Segment_2),</i> <i>Stream& operator<<(Stream&, Gt::Ray_2).</i></p>
<pre>template< class Stream ></pre>	

<i>Stream&</i>	<i>ag.draw_dual_edge(Edge e, Stream& str)</i>	Draws the dual of the edge <i>e</i> to the stream <i>str</i> . The dual of <i>e</i> is an edge of the Apollonius diagram. <i>Precondition:</i> The following operators must be defined: <i>Stream& operator<<(Stream&, Gt::Segment_2),</i> <i>Stream& operator<<(Stream&, Gt::Ray_2),</i> <i>Stream& operator<<(Stream&, Gt::Line_2).</i>
<i>void</i>	<i>ag.file_output(std::ostream& os)</i>	Writes the current state of the Apollonius graph to an output stream. In particular, all visible and hidden sites are written as well as the underlying combinatorial data structure.
<i>void</i>	<i>ag.file_input(std::istream& is)</i>	Reads the state of the Apollonius graph from an input stream.
<i>std::ostream&</i>	<i>std::ostream& os << ag</i>	Writes the current state of the Apollonius graph to an output stream.
<i>std::istream&</i>	<i>std::istream& is >> ag</i>	Reads the state of the Apollonius graph from an input stream.

Validity check

<i>bool</i>	<i>ag.is_valid(bool verbose = false, int level = 1)</i>	Checks the validity of the Apollonius graph. If <i>verbose</i> is <i>true</i> a short message is sent to <i>std::cerr</i> . If <i>level</i> is 0, only the data structure is validated. If <i>level</i> is 1, then both the data structure and the Apollonius graph are validated. Negative values of <i>level</i> always return true, and values greater than 1 are equivalent to <i>level</i> being 1.
-------------	----------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Miscellaneous

<i>void</i>	<i>ag.clear()</i>	Clears all contents of the Apollonius graph.
<i>void</i>	<i>ag.swap(other)</i>	The Apollonius graphs <i>other</i> and <i>ag</i> are swapped. <i>ag.swap(other)</i> should be preferred to <i>ag= other</i> or to <i>ag(other)</i> if <i>other</i> is deleted afterwards.

See Also

[DelaunayGraph_2](#)
[ApolloniusGraphTraits_2](#)
[ApolloniusGraphDataStructure_2](#)
[ApolloniusGraphVertexBase_2](#)
[TriangulationFaceBase_2](#)
[CGAL::Apollonius_graph_traits_2<K,Method_tag>](#)
[CGAL::Apollonius_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>](#)
[CGAL::Triangulation_data_structure_2<Vb,Fb>](#)
[CGAL::Apollonius_graph_vertex_base_2<Gt,StoreHidden>](#)
[CGAL::Triangulation_face_base_2<Gt>](#)

ApolloniusSite_2

Definition

The concept *ApolloniusSite_2* provides the requirements for an Apollonius site class.

Types

<i>ApolloniusSite_2:: Point_2</i>	The point type.
<i>ApolloniusSite_2:: FT</i>	The field number type.
<i>ApolloniusSite_2:: RT</i>	The ring number type.
<i>ApolloniusSite_2:: Weight</i>	The weight type.
<i>Precondition:</i> It must be the same as <i>FT</i> .	

Creation

<i>ApolloniusSite_2</i> <i>s</i> (<i>Point_2</i> <i>p</i> = <i>Point_2</i> (), <i>Weight</i> <i>w</i> = <i>Weight</i> (0));	
<i>ApolloniusSite_2</i> <i>s</i> (<i>other</i>);	Copy constructor.

Access Functions

<i>Point_2</i>	<i>s.point()</i>	Returns the center of the Apollonius site.
<i>Weight</i>	<i>s.weight()</i>	Returns the weight of the Apollonius site.

See Also

ApolloniusGraphTraits_2
CGAL::Apollonius_site_2<K>
CGAL::Apollonius_graph_traits_2<K,Method_tag>
CGAL::Apollonius_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>

CGAL::Apollonius_site_2<K>

Definition

The class *Apollonius_site_2*<*K*> is a model for the concept *ApolloniusSite_2*. It is parametrized by a template parameter *K* which must be a model of the *Kernel* concept.

```
#include <CGAL/Apollonius_site_2.h>
```

Is Model for the Concepts

ApolloniusSite_2

Types

The class *Apollonius_site_2*<*K*> does not introduce any types in addition to the concept *ApolloniusSite_2*.

Creation

```
Apollonius_site_2<K> s( Point_2 p=Point_2(), Weight w= Weight(0));  
Apollonius_site_2<K> s( other);           Copy constructor.
```

I/O

The I/O operators are defined for *iostream*.

```
std::ostream&      std::ostream& os << s      Inserts the Apollonius site s into the stream os.  
Precondition: The insert operator must be defined for Point_2 and  
Weight.
```

```
std::istream&      std::istream& is >> s      Reads an Apollonius site from the stream is and assigns it to s.  
Precondition: The extract operator must be defined for Point_2 and  
Weight.
```

The information output in the *iostream* is: the point of the Apollonius site and its weight.

```
#include <CGAL/IO/Qt_widget_Apollonius_site_2.h>
```

```
Qt_widget&         Qt_widget& w << s           Inserts the Apollonius site s into the Qt_widget stream w.  
Precondition: The insert operator must be defined for K::Circle_2.
```

See Also

Kernel
ApolloniusSite_2
CGAL::Qt_widget
CGAL::Apollonius_graph_traits_2<*K*,*Method_tag*>
CGAL::Apollonius_graph_filtered_traits_2<*CK*,*CM*,*EK*,*EM*,*FK*,*FM*>

ApolloniusGraphDataStructure_2

Definition

The concept *ApolloniusGraphDataStructure_2* refines the concept *TriangulationDataStructure_2*. In addition it provides two methods for the insertion and removal of a degree 2 vertex in the data structure. The insertion method adds a new vertex to the specified edge, thus creating two new edges. Moreover, it creates two new faces that have the two newly created edges in common (see figure below). The removal method performs the reverse operation.

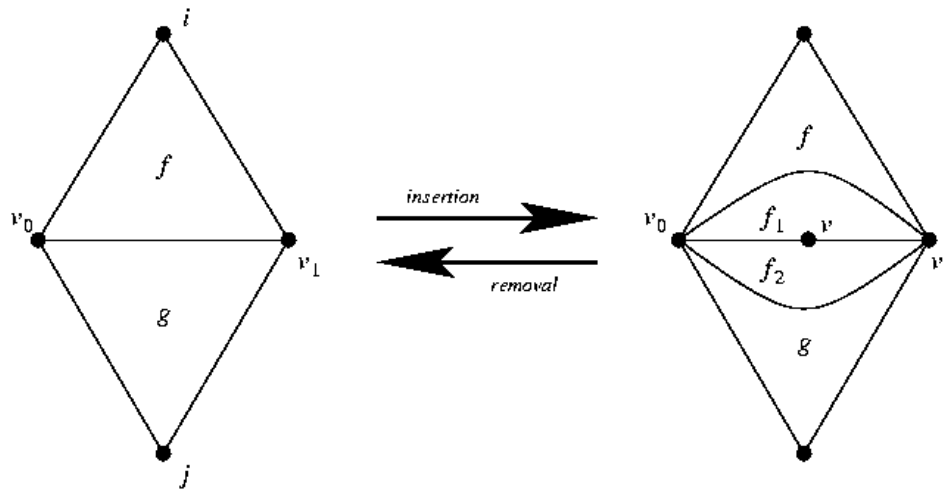


Figure 27.4: Insertion and removal of degree 2 vertices. Left to right: The edge (f,i) is replaced by two edges by means of inserting a vertex v on the edge. The faces f_1 and f_2 are created. Right to left: the faces f_1 and f_2 are destroyed. The vertex v is deleted and its two adjacent edges are merged.

We only describe the additional requirements with respect to the *TriangulationDataStructure_2* concept.

Refines

TriangulationDataStructure_2

Insertion

Vertex_handle `agds.insert_degree_2(Face_handle f, int i)`

Inserts a degree two vertex and two faces adjacent to it that have two common edges. The edge defined by the face handle f and the integer i is duplicated. It returns a handle to the vertex created.

Removal

void *agds.remove_degree_2(Vertex_handle v)* Removes a degree 2 vertex and the two faces adjacent to it. The two edges of the star of v that are not incident to it are collapsed.
Precondition: The degree of v must be equal to 2.

Has Models

CGAL::Triangulation_data_structure_2<Vb,Fb>

See Also

TriangulationDataStructure_2
ApolloniusGraphVertexBase_2
TriangulationFaceBase_2

ApolloniusGraphVertexBase_2

Definition

The concept *ApolloniusGraphVertexBase_2* describes the requirements for the vertex base class of the *ApolloniusGraphDataStructure_2* concept. A vertex stores an Apollonius site and provides access to one of its incident faces through a *Face_handle*. In addition, it maintains a container of sites. The container stores the hidden sites related to the vertex.

Types

<i>ApolloniusGraphVertexBase_2:: Geom_traits</i>	A type for the geometric traits that defines the site stored. <i>Precondition:</i> The type <i>Geom_traits</i> must define the type <i>Site_2</i> .
<i>ApolloniusGraphVertexBase_2:: Store_hidden</i>	A boolean that indicates if hidden sites are actually stored or not. Its value is <i>true</i> if hidden sites are stored, <i>false</i> otherwise.
<i>ApolloniusGraphVertexBase_2:: Site_2</i>	A type for the site stored. <i>Precondition:</i> This type must coincide with the type <i>Geom_traits::Site_2</i> .
<i>ApolloniusGraphVertexBase_2:: Apollonius_graph_data_structure_2</i>	A type for the Apollonius graph data structure, to which the vertex belongs to.
<i>ApolloniusGraphVertexBase_2:: Vertex_handle</i>	A type for the vertex handle of the Apollonius graph data structure.
<i>ApolloniusGraphVertexBase_2:: Face_handle</i>	A type for the face handle of the Apollonius graph data structure.
<i>ApolloniusGraphVertexBase_2:: Hidden_sites_iterator</i>	An iterator that iterates over the hidden sites in the hidden sites container of the vertex. <i>Precondition:</i> Must be a model of <i>Iterator</i> .

Creation

<i>ApolloniusGraphVertexBase_2 v;</i>	Default constructor.
<i>ApolloniusGraphVertexBase_2 v(Site_2 s);</i>	Constructs a vertex associated with the Apollonius site <i>s</i> and embedded at the center of <i>s</i> .
<i>ApolloniusGraphVertexBase_2 v(Site_2 s, Face_handle f);</i>	Constructs a vertex associated with the site <i>s</i> , embedded at the center of <i>s</i> , and pointing to the face associated with the face handle <i>f</i> .

Access Functions

<i>Site_2</i>	<i>v.site()</i>	Returns the Apollonius site.
<i>Face_handle</i>	<i>v.face()</i>	Returns a handle to an incident face.
<i>unsigned int</i>	<i>v.number_of_hidden_sites()</i>	Returns the number of hidden sites in the hidden sites container.

<i>Hidden_sites_iterator</i>	<i>v.hidden_sites_begin()</i>	Starts at an arbitrary hidden site.
<i>Hidden_sites_iterator</i>	<i>v.hidden_sites_end()</i>	Past-the-end iterator.

Setting and unsetting

<i>void</i>	<i>v.set_site(Site_2 s)</i>	Sets the Apollonius site.
<i>void</i>	<i>v.set_face(Face_handle f)</i>	Sets the incident face.
<i>void</i>	<i>v.add_hidden_site(Site_2 s)</i>	Adds a hidden site to the container of hidden sites.
<i>void</i>	<i>v.clear_hidden_sites_container()</i>	Clears the container of hidden sites.

Checking

bool

<i>v.is_valid(bool verbose, int level)</i>	Performs any required tests on a vertex.
---------------------------------------------	------------------------------------------

Has Models

CGAL::Apollonius_graph_vertex_base_2<Gt,StoreHidden>.

See Also

ApolloniusGraphDataStructure_2
ApolloniusGraphTraits_2
CGAL::Apollonius_graph_vertex_base_2<Gt,StoreHidden>

CGAL::Apollonius_graph_vertex_base_2<Gt,StoreHidden>

Definition

The class *Apollonius_graph_vertex_base_2<Gt,StoreHidden>* provides a model for the *ApolloniusGraphVertexBase_2* concept which is the vertex base required by the *ApolloniusGraphDataStructure_2* concept. The class *Apollonius_graph_vertex_base_2<Gt,StoreHidden>* has two template arguments, the first being the geometric traits of the Apollonius graph and should be a model of the concept *ApolloniusGraphTraits_2*. The second is a boolean which controls whether hidden sites are actually stored. Such a control is important if the user is not interested in hidden sites and/or if only insertions are made, in which case no hidden site can become visible. If *StoreHidden* is set to *true*, hidden sites are stored, otherwise they are discarded. By default *StoreHidden* is set to *true*.

```
#include <CGAL/Apollonius_graph_vertex_base_2.h>
```

Is Model for the Concepts

ApolloniusGraphVertexBase_2

Creation

<i>Apollonius_graph_vertex_base_2<Gt,StoreHidden></i> vb;	Default constructor.
<i>Apollonius_graph_vertex_base_2<Gt,StoreHidden></i> vb(<i>Site_2 s</i>);	Constructs a vertex associated with the site <i>s</i> and embedded at the center of <i>s</i> .
<i>Apollonius_graph_vertex_base_2<Gt,StoreHidden></i> vb(<i>Site_2 s</i> , <i>Face_handle f</i>);	Constructs a vertex associated with the site <i>s</i> , embedded at the center of <i>s</i> , and pointing to the face associated with the face handle <i>f</i> .

See Also

ApolloniusGraphVertexBase_2
ApolloniusGraphDataStructure_2
ApolloniusGraphTraits_2
CGAL::Triangulation_data_structure_2<Vb,Fb>
CGAL::Apollonius_graph_traits_2<K,Method_tag>
CGAL::Apollonius_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>

ApolloniusGraphTraits_2

Definition

The concept *ApolloniusGraphTraits_2* provides the traits requirements for the *Apollonius_graph_2* class. In particular, it provides a type *Site_2*, which must be a model of the concept *ApolloniusSite_2*. It also provides constructions for sites and several function object types for the predicates.

Types

<i>ApolloniusGraphTraits_2:: Point_2</i>	A type for a point.
<i>ApolloniusGraphTraits_2:: Site_2</i>	A type for an Apollonius site. Must be a model of the concept <i>ApolloniusSite_2</i> .
<i>ApolloniusGraphTraits_2:: Line_2</i>	A type for a line. Only required if access to the dual of the Apollonius graph is required or if the primal or dual diagram are inserted in a stream.
<i>ApolloniusGraphTraits_2:: Ray_2</i>	A type for a ray. Only required if access to the dual of the Apollonius graph is required or if the primal or dual diagram are inserted in a stream.
<i>ApolloniusGraphTraits_2:: Segment_2</i>	A type for a segment. Only required if access to the dual of the Apollonius graph is required or if the primal or dual diagram are inserted in a stream.
<i>ApolloniusGraphTraits_2:: Object_2</i>	A type representing different types of objects in two dimensions, namely: <i>Point_2</i> , <i>Site_2</i> , <i>Line_2</i> , <i>Ray_2</i> and <i>Segment_2</i> .
<i>ApolloniusGraphTraits_2:: FT</i>	A type for the field number type of sites.
<i>ApolloniusGraphTraits_2:: RT</i>	A type for the ring number type of sites.
<i>ApolloniusGraphTraits_2:: Assign_2</i>	Must provide <i>template <class T> bool operator() (T& t, Object_2 o)</i> which assigns <i>o</i> to <i>t</i> if <i>o</i> was constructed from an object of type <i>T</i> . Returns <i>true</i> , if the assignment was possible.
<i>ApolloniusGraphTraits_2:: Construct_object_2</i>	Must provide <i>template <class T> Object_2 operator() (T t)</i> that constructs an object of type <i>Object_2</i> that contains <i>t</i> and returns it.
<i>ApolloniusGraphTraits_2:: Construct_Apollonius_vertex_2</i>	A constructor for a point of the Apollonius diagram equidistant from three sites. Must provide <i>Point_2 operator() (Site_2 s1, Site_2 s2, Site_2 s3)</i> , which constructs a point equidistant from the sites <i>s1</i> , <i>s2</i> and <i>s3</i> .
<i>ApolloniusGraphTraits_2:: Construct_Apollonius_site_2</i>	A constructor for a dual Apollonius site (a site whose center is a vertex of the Apollonius diagram and its weight is the common distance of its center from the three defining sites). Must provide <i>Site_2 operator() (Site_2 s1, Site_2 s2, Site_2 s3)</i> , which constructs a dual site whose center <i>c</i> is equidistant from <i>s1</i> , <i>s2</i> and <i>s3</i> , and its weight is equal to the (signed) distance of <i>c</i> from <i>s1</i> (or <i>s2</i> or <i>s3</i>). Must also provide <i>Line_2 operator() (Site_2 s1, Site_2 s2)</i> , which constructs a line bitangent to <i>s1</i> and <i>s2</i> . This line is the dual site of <i>s1</i> , <i>s2</i> and the site at infinity; it can be viewed as a dual Apollonius site whose center is at infinity and its weight is infinite.

ApolloniusGraphTraits_2:: Compare_x_2

A predicate object type. Must provide *Comparison_result operator()(Site_2 s1, Site_2 s2)*, which compares the x-coordinates of the centers of *s1* and *s2*.

ApolloniusGraphTraits_2:: Compare_y_2

A predicate object type. Must provide *Comparison_result operator()(Site_2 s1, Site_2 s2)*, which compares the y-coordinates of the centers of *s1* and *s2*.

ApolloniusGraphTraits_2:: Compare_weight_2

A predicate object type. Must provide *Comparison_result operator()(Site_2 s1, Site_2 s2)*, which compares the weights of *s1* and *s2*.

ApolloniusGraphTraits_2:: Orientation_2

A predicate object type. Must provide *Orientation operator()(Site_2 s1, Site_2 s2, Site_2 s3)*, which performs the usual orientation test for the centers of the three sites *s1*, *s2* and *s3*.

Must also provide *Orientation operator()(Site_2 s1, Site_2 s2, Site_2 s3, Site_2 p1, Site_2 p2)*, which performs the usual orientation test for the Apollonius vertex of *s1*, *s2*, *s3* and the centers of *p1* and *p2*.

Precondition: the Apollonius vertex of *s1*, *s2* and *s3* must exist.

ApolloniusGraphTraits_2:: Is_hidden_2

A predicate object type. Must provide *bool operator()(Site_2 s1, Site_2 s2)*, which returns *true* if the circle corresponding to *s2* is contained in the closure of the disk corresponding to *s1*, *false* otherwise.

ApolloniusGraphTraits_2:: Oriented_side_of_bisector_2

A predicate object type. Must provide *Oriented_side operator()(Site_2 s1, Site_2 s2, Point_2 p)*, which returns the oriented side of the bisector of *s1* and *s2* that contains *p*. Returns *ON_POSITIVE_SIDE* if *p* lies in the half-space of *s1* (i.e., *p* is closer to *s1* than *s2*); returns *ON_NEGATIVE_SIDE* if *p* lies in the half-space of *s2*; returns *ON_ORIENTED_BOUNDARY* if *p* lies on the bisector of *s1* and *s2*.

ApolloniusGraphTraits_2:: Vertex_conflict_2

A predicate object type. Must provide *Sign operator()(Site_2 s1, Site_2 s2, Site_2 s3, Site_2 q)*, which returns the sign of the distance of *q* from the dual Apollonius site of *s1*, *s2*, *s3*.

Precondition: the dual Apollonius site of *s1*, *s2*, *s3* must exist.

Must also provide *Sign operator()(Site_2 s1, Site_2 s2, Site_2 q)*, which returns the sign of the distance of *q* from the bitangent line of *s1*, *s2* (a degenerate dual Apollonius site, with its center at infinity).

ApolloniusGraphTraits_2:: Finite_edge_interior_conflict_2

A predicate object type. Must provide *bool operator()(Site_2 s1, Site_2 s2, Site_2 s3, Site_2 s4, Site_2 q, bool b)*. The sites *s1*, *s2*, *s3* and *s4* define an Apollonius edge that lies on the bisector of *s1* and *s2* and has as endpoints the Apollonius vertices defined by the triplets *s1*, *s2*, *s3* and *s1*, *s4* and *s2*. The boolean *b* denotes if the two Apollonius vertices are in conflict with the site *q* (in which case *b* should be *true*, otherwise *false*). In case that *b* is *true*, the predicate returns *true* if and only if the entire Apollonius edge is in conflict with *q*. If *b* is *false*, the predicate returns *false* if and only if *q* is not in conflict with the Apollonius edge.

Precondition: the Apollonius vertices of *s1*, *s2*, *s3*, and *s1*, *s4*, *s2* must exist.

Must also provide *bool operator()(Site_2 s1, Site_2 s2, Site_2 s3, Site_2 q, bool b)*. The sites *s1*, *s2*, *s3* and the site at infinity s_∞ define an Apollonius edge that lies on the bisector of *s1* and *s2* and has as endpoints the Apollonius vertices defined by the triplets *s1*, *s2*, *s3* and *s1*, s_∞ and *s2* (the second Apollonius vertex is actually at infinity). The boolean *b* denotes if the two Apollonius vertices are in conflict with the site *q* (in which case *b* should be *true*, otherwise *false*). In case that *b* is *true*, the predicate returns *true* if and only if the entire Apollonius edge is in conflict with *q*. If *b* is *false*, the predicate returns *false* if and only if *q* is not in conflict with the Apollonius edge.

Precondition: the Apollonius vertex of *s1*, *s2*, *s3* must exist.

Must finally provide *bool operator()(Site_2 s1, Site_2 s2, Site_2 q, bool b)*. The sites *s1*, *s2* and the site at infinity s_∞ define an Apollonius edge that lies on the bisector of *s1* and *s2* and has as endpoints the Apollonius vertices defined by the triplets *s1*, *s2*, s_∞ and *s1*, s_∞ and *s2* (both Apollonius vertices are actually at infinity). The boolean *b* denotes if the two Apollonius vertices are in conflict with the site *q* (in which case *b* should be *true*, otherwise *false*). In case that *b* is *true*, the predicate returns *true* if and only if the entire Apollonius edge is in conflict with *q*. If *b* is *false*, the predicate returns *false* if and only if *q* is not in conflict with the Apollonius edge.

ApolloniusGraphTraits_2:: Infinite_edge_interior_conflict_2

A predicate object type. Must provide *bool operator()(Site_2 s1, Site_2 s2, Site_2 s3, Site_2 q, bool b)*. The sites s_∞ , *s1*, *s2* and *s3* define an Apollonius edge that lies on the bisector of s_∞ and *s1* and has as endpoints the Apollonius vertices defined by the triplets s_∞ , *s1*, *s2* and s_∞ , *s3* and *s1*. The boolean *b* denotes if the two Apollonius vertices are in conflict with the site *q* (in which case *b* should be *true*, otherwise *false*). In case that *b* is *true*, the predicate returns *true* if and only if the entire Apollonius edge is in conflict with *q*. If *b* is *false*, the predicate returns *false* if and only if *q* is not in conflict with the Apollonius edge.

ApolloniusGraphTraits_2:: Is_degenerate_edge_2

A predicate object type. Must provide *bool operator()(Site_2 s1, Site_2 s2, Site_2 s3, Site_2 s4)*. It returns *true* if the Apollonius edge defined by *s1*, *s2*, *s3* and *s4* is degenerate, *false* otherwise. An Apollonius edge is called degenerate if its two endpoints coincide.

Precondition: the Apollonius vertices of *s1*, *s2*, *s3*, and *s1*, *s4*, *s2* must exist.

Creation

ApolloniusGraphTraits_2 *gt*;
ApolloniusGraphTraits_2 *gt*(*other*);

Default constructor.
Copy constructor.

ApolloniusGraphTraits_2 *gt = other* Assignment operator.

Access to predicate objects

<i>Compare_x_2</i>	<i>gt.compare_x_2_object()</i>
<i>Compare_y_2</i>	<i>gt.compare_y_2_object()</i>
<i>Compare_weight_2</i>	<i>gt.compare_weight_2_object()</i>
<i>Orientation_2</i>	<i>gt.orientation_2_object()</i>
<i>Is_hidden_2</i>	<i>gt.is_hidden_2_object()</i>
<i>Oriented_side_of_bisector_2</i>	<i>gt.oriented_side_of_bisector_test_2_object()</i>
<i>Vertex_conflict_2</i>	<i>gt.vertex_conflict_2_object()</i>
<i>Finite_edge_interior_conflict_2</i>	<i>gt.finite_edge_interior_conflict_2_object()</i>
<i>Infinite_edge_interior_conflict_2</i>	<i>gt.infinite_edge_interior_conflict_2_object()</i>
<i>Is_degenerate_edge_2</i>	<i>gt.is_degenerate_edge_2_object()</i>

Access to constructor objects

<i>Construct_object_2</i>	<i>gt.construct_object_2_object()</i>
<i>Construct_Apollonius_vertex_2</i>	<i>gt.construct_Apollonius_vertex_2_object()</i>
<i>Construct_Apollonius_site_2</i>	<i>gt.construct_Apollonius_site_2_object()</i>

Access to other objects

<i>Assign_2</i>	<i>gt.assign_2_object()</i>
-----------------	-----------------------------

Has Models

CGAL::Apollonius_graph_traits_2<*K*,*Method_tag*>
CGAL::Apollonius_graph_filtered_traits_2<*CK*,*CM*,*EK*,*EM*,*FK*,*FM*>

See Also

CGAL::Apollonius_graph_2<*Gt*,*Agds*>
CGAL::Apollonius_graph_traits_2<*K*,*Method_tag*>
CGAL::Apollonius_graph_filtered_traits_2<*CK*,*CM*,*EK*,*EM*,*FK*,*FM*>

CGAL::Apollonius_graph_traits_2<K,Method_tag>

Definition

The class *Apollonius_graph_traits_2*<*K*,*Method_tag*> provides a model for the *ApolloniusGraphTraits_2* concept. This class has two template parameters. The first template parameter must be a model of the *Kernel* concept. The second template parameter corresponds to how predicates are evaluated. There are two predefined possible values for *Method_tag*, namely *CGAL::Sqrt_field_tag* and *CGAL::Ring_tag*. The first one must be used when the number type used in the representation supports the exact evaluation of signs of expressions involving all four basic operations and square roots, whereas the second one requires the exact evaluation of signs of ring-type expressions, i.e., expressions involving only additions, subtractions and multiplications. The default value for *Method_tag* is *CGAL::Ring_tag*. The way the predicates are evaluated is discussed in [KE02, KE03].

```
#include <CGAL/Apollonius_graph_traits_2.h>
```

Is Model for the Concepts

ApolloniusGraphTraits_2

Creation

<i>Apollonius_graph_traits_2</i> < <i>K</i> , <i>Method_tag</i> > traits;	Default constructor.
<i>Apollonius_graph_traits_2</i> < <i>K</i> , <i>Method_tag</i> > traits(other);	Copy constructor.
<i>Apollonius_graph_traits_2</i> < <i>K</i> , <i>Method_tag</i> > traits = other	
	Assignment operator.

See Also

Kernel
ApolloniusGraphTraits_2
CGAL::Ring_tag
CGAL::Sqrt_field_tag
CGAL::Apollonius_graph_2<*Gt*,*Agds*>
CGAL::Apollonius_graph_filtered_traits_2<*CK*,*CM*,*EK*,*EM*,*FK*,*FM*>

CGAL::Apollonius_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>

Definition

The class *Apollonius_graph_filtered_traits_2*<CK,CM,EK,EM,FK,FM> provides a model for the *ApolloniusGraphTraits_2* concept.

The class *Apollonius_graph_filtered_traits_2*<CK,CM,EK,EM,FK,FM> uses the filtering technique [BBP01] to achieve traits for the *Apollonius_graph_2*<Gt,Agds> class with efficient and exact predicates given an exact kernel *EK* and a filtering kernel *FK*. The geometric constructions associated provided by this class are equivalent to those provided by the traits class *Apollonius_graph_traits_2*<CK,CM>, which means that they may be inexact.

This class has six template parameters. The first, third and fifth template parameters must be a models of the *Kernel* concept. The second, fourth and sixth template parameters correspond to how predicates are evaluated. There are two predefined possible values for *Method_tag*, namely *CGAL::Sqrt_field_tag* and *CGAL::Ring_tag*. The first one must be used when the number type used in the representation supports the exact evaluation of signs of expressions involving all four basic operations and square roots, whereas the second one requires the exact evaluation of signs of ring-type expressions, i.e., expressions involving only additions, subtractions and multiplications. The way the predicates are evaluated is discussed in [KE02, KE03].

The default values for the template parameters are as follows: *CM* = *CGAL::Ring_tag*, *EK* = *CGAL::Simple_cartesian*<*CGAL::MP_Float*>, *EM* = *CM*, *FK* = *CGAL::Simple_cartesian*<*CGAL::Interval_nt*<false> >, *FM* = *CM*.

```
#include <CGAL/Apollonius_graph_filtered_traits_2.h>
```

Is Model for the Concepts

ApolloniusGraphTraits_2

Creation

<i>Apollonius_graph_filtered_traits_2</i> <CK,CM,EK,EM,FK,FM> traits;	Default
<i>Apollonius_graph_filtered_traits_2</i> <CK,CM,EK,EM,FK,FM> traits(other);	constructor.
	Copy
<i>Apollonius_graph_filtered_traits_2</i> <CK,CM,EK,EM,FK,FM> traits = other	constructor.
	Assignment
	operator.

See Also

Kernel
ApolloniusGraphTraits_2
CGAL::Ring_tag
CGAL::Sqrt_field_tag
CGAL::Apollonius_graph_2<Gt,Agds>
CGAL::Apollonius_graph_traits_2<K,Method_tag>

CGAL::Apollonius_graph_hierarchy_2<Gt,Agds>

Definition

We provide an alternative to the class *Apollonius_graph_2<Gt,Agds>* for the dynamic construction of the Apollonius graph. The *Apollonius_graph_hierarchy_2<Gt,Agds>* class maintains a hierarchy of Apollonius graphs. The bottom-most level of the hierarchy contains the full Apollonius diagram. A site that is in level i , is in level $i + 1$ with probability $1/\alpha$ where $\alpha > 1$ is some constant. The difference between the *Apollonius_graph_2<Gt,Agds>* class and the *Apollonius_graph_hierarchy_2<Gt,Agds>* is on how the nearest neighbor location is done. Given a point p the location is done as follows: at the top most level we find the nearest neighbor of p as in the *Apollonius_graph_2<Gt,Agds>* class. At every subsequent level i we use the nearest neighbor found at level $i + 1$ to find the nearest neighbor at level i . This is a variant of the corresponding hierarchy for points found in [Dev98]. The class has two template parameters which have essentially the same meaning as in the *Apollonius_graph_2<Gt,Agds>* class. The first template parameter must be a model of the *ApolloniusGraphTraits_2* concept. The second template parameter must be a model of the *ApolloniusGraphDataStructure_2* concept. However, the vertex base class that is to be used in the Apollonius graph data structure must be a model of the *ApolloniusGraphHierarchyVertexBase_2* concept. The second template parameter defaults to *Triangulation_data_structure_2< Apollonius_graph_hierarchy_vertex_base_2< Apollonius_graph_vertex_base_2<Gt,true> >, Triangulation_face_base_2<Gt> >*.

The *Apollonius_graph_hierarchy_2<Gt,Agds>* class derives publicly from the *Apollonius_graph_2<Gt,Agds>* class. The interface is the same with its base class. In the sequel only the methods overridden are documented.

```
#include <CGAL/Apollonius_graph_hierarchy_2.h>
```

Inherits From

CGAL::Apollonius_graph_2<Gt,Agds>

Types

Apollonius_graph_hierarchy_2<Gt,Agds> does not introduce other types than those introduced by its base class *Apollonius_graph_2<Gt,Agds>*.

Creation

Apollonius_graph_hierarchy_2<Gt,Agds> *agh*(*Gt* *gt*=*Gt*()); Creates an hierarchy of Apollonius graphs using *gt* as geometric traits.

template< class Input_iterator >
Apollonius_graph_hierarchy_2<Gt,Agds> *agh*(*Input_iterator* *first*, *Input_iterator* *beyond*, *Gt* *gt*=*Gt*()); Creates an Apollonius graph hierarchy using *gt* as geometric traits and inserts all sites in the range [*first*, *beyond*).

Apollonius_graph_hierarchy_2<Gt,Agds> *agh*(*other*); Copy constructor. All faces, vertices and inter-level pointers are duplicated. After the construction, *agh* and *other* refer to two different Apollonius graph hierarchies: if *other* is modified, *agh* is not.

<i>Apollonius_graph_hierarchy_2</i> < <i>Gt,Agds</i> >	<i>agh</i> = <i>other</i>	Assignment. All faces, vertices and inter-level pointers are duplicated. After the construction, <i>agh</i> and <i>other</i> refer to two different Apollonius graph hierarchies: if <i>other</i> is modified, <i>agh</i> is not.
--------------------------------------------------------	---------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Insertion

```
template< class Input_iterator >
```

```
unsigned int    agh.insert( Input_iterator first, Input_iterator beyond)
```

Inserts the sites in the range [*first*,*beyond*). The number of sites in the range [*first*, *beyond*) is returned.

Precondition: *Input_iterator* must be a model of *InputIterator* and its value type must be *Site_2*.

<i>Vertex_handle</i>	<i>agh.insert(Site_2 s)</i>	Inserts the site <i>s</i> in the Apollonius graph hierarchy. If <i>s</i> is visible then the vertex handle of <i>s</i> is returned, otherwise <i>Vertex_handle(NULL)</i> is returned.
----------------------	------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<i>Vertex_handle</i>	<i>agh.insert(Site_2 s, Vertex_handle vnear)</i>	
----------------------	---------------------------------------------------	--

Inserts *s* in the Apollonius graph hierarchy using the site associated with *vnear* as an estimate for the nearest neighbor of the center of *s*. If *s* is visible then the vertex handle of *s* is returned, otherwise *Vertex_handle(NULL)* is returned. A call to this method is equivalent to *agh.insert(s)*; and it has been added for the sake of conformity with the interface of the *Apollonius_graph_2*<*Gt,Agds*> class.

Removal

<i>void</i>	<i>agh.remove(Vertex_handle v)</i>	Removes the site associated to the vertex handle <i>v</i> from the Apollonius graph hierarchy. <i>Precondition:</i> <i>v</i> must correspond to a valid finite vertex of the Apollonius graph hierarchy.
-------------	-------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Nearest neighbor location

<i>Vertex_handle</i>	<i>agh.nearest_neighbor(Point p)</i>	Finds the nearest neighbor of the point <i>p</i> . In other words it finds the site whose Apollonius cell contains <i>p</i> . Ties are broken arbitrarily and one of the nearest neighbors of <i>p</i> is returned. If there are no visible sites in the Apollonius diagram <i>Vertex_handle(NULL)</i> is returned.
----------------------	---------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<i>Vertex_handle</i>	<i>agh.nearest_neighbor(Point p, Vertex_handle vnear)</i>	
----------------------	------------------------------------------------------------	--

Finds the nearest neighbor of the point *p*. If there are no visible sites in the Apollonius diagram *Vertex_handle(NULL)* is returned. A call to this method is equivalent to *agh.nearest_neighbor(p)*; and it has been added for the sake of conformity with the interface of the *Apollonius_graph_2*<*Gt,Agds*> class.

I/O

<code>void</code>	<code>agh.file_output(std::ostream& os)</code>	Writes the current state of the Apollonius graph hierarchy to an output stream. In particular, all visible and hidden sites are written as well as the underlying combinatorial hierarchical data structure.
<code>void</code>	<code>agh.file_input(std::istream& is)</code>	Reads the state of the Apollonius graph hierarchy from an input stream.
<code>std::ostream&</code>	<code>std::ostream& os << agh</code>	Writes the current state of the Apollonius graph hierarchy to an output stream.
<code>std::istream&</code>	<code>std::istream& is >> agh</code>	Reads the state of the Apollonius graph hierarchy from an input stream.

Validity check

<code>bool</code>	<code>agh.is_valid(bool verbose = false, int level = 1)</code>	Checks the validity of the Apollonius graph hierarchy. If <i>verbose</i> is <i>true</i> a short message is sent to <code>std::cerr</code> . If <i>level</i> is 0, the data structure at all levels is validated, as well as the inter-level pointers. If <i>level</i> is 1, then the data structure at all levels is validated, the inter-level pointers are validated and all levels of the Apollonius graph hierarchy are also validated. Negative values of <i>level</i> always return <i>true</i> , and values greater than 1 are equivalent to <i>level</i> being 1.
-------------------	-----------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Miscellaneous

<code>void</code>	<code>agh.clear()</code>	Clears all contents of the Apollonius graph hierarchy.
<code>void</code>	<code>agh.swap(other)</code>	The Apollonius graph hierarchies <i>other</i> and <i>agh</i> are swapped. <code>agh.swap(other)</code> should be preferred to <code>agh= other</code> or to <code>agh(other)</code> if <i>other</i> is deleted afterwards.

See Also

`ApolloniusGraphDataStructure_2`
`ApolloniusGraphTraits_2`
`ApolloniusGraphHierarchyVertexBase_2`
`CGAL::Apollonius_graph_2<Gt,Agds>`
`CGAL::Triangulation_data_structure_2<Vb,Fb>`
`CGAL::Apollonius_graph_traits_2<K,Method_tag>`
`CGAL::Apollonius_graph_filtered_traits_2<CK,CM,EK,EM,FK,FM>`
`CGAL::Apollonius_graph_hierarchy_vertex_base_2<Agvb>`

ApolloniusGraphHierarchyVertexBase_2

Definition

The vertex of an Apollonius graph included in an Apollonius graph hierarchy has to provide some pointers to the corresponding vertices in the graphs of the next and preceeding levels. Therefore, the concept *ApolloniusGraphHierarchyVertexBase_2* refines the concept *ApolloniusGraphVertexBase_2*, by adding two vertex handles to the corresponding vertices for the next and previous level graphs.

Refines

ApolloniusGraphVertexBase_2

Types

ApolloniusGraphHierarchyVertexBase_2 does not introduce any types in addition to those of *ApolloniusGraphVertexBase_2*.

Creation

<i>ApolloniusGraphHierarchyVertexBase_2</i> <i>v</i> ;	Default constructor.
<i>ApolloniusGraphHierarchyVertexBase_2</i> <i>v</i> (<i>Site_2</i> <i>s</i>);	Constructs a vertex associated with the site <i>s</i> and embedded at the center of <i>s</i> .
<i>ApolloniusGraphHierarchyVertexBase_2</i> <i>v</i> (<i>Site_2</i> <i>s</i> , <i>Face_handle</i> <i>f</i>);	Constructs a vertex associated with the site <i>s</i> , embedded at the center of <i>s</i> , and pointing to face <i>f</i> .

Operations

<i>Vertex_handle</i> <i>v.up</i> ()	Returns a handle to the corresponding vertex of the next level Apollonius graph. If such a vertex does not exist <i>Vertex_handle(NULL)</i> is returned.
<i>Vertex_handle</i> <i>v.down</i> ()	Returns a handle to the corresponding vertex of the previous level Apollonius graph.
<i>void</i> <i>v.set_up</i> (<i>Vertex_handle</i> <i>u</i>)	Sets the handle for the vertex of the next level Apollonius graph.
<i>void</i> <i>v.set_down</i> (<i>Vertex_handle</i> <i>d</i>)	Sets the handle for the vertex of the previous level Apollonius graph;

Has Models

CGAL::Apollonius_graph_hierarchy_vertex_base_2<*CGAL::Apollonius_graph_vertex_base_2*<*Gt*,*StoreHidden*> >

See Also

ApolloniusGraphDataStructure_2

ApolloniusGraphVertexBase_2

CGAL::Apollonius_graph_hierarchy_2<Gt,Agds>

CGAL::Triangulation_data_structure_2<Vb,Fb>

CGAL::Apollonius_graph_vertex_base_2<Gt,StoreHidden>

CGAL::Apollonius_graph_hierarchy_vertex_base_2<Agvb>

CGAL::Apollonius_graph_hierarchy_vertex_base_2<Agvb>

Definition

The class *Apollonius_graph_hierarchy_vertex_base_2<Agvb>* provides a model for the *ApolloniusGraphHierarchyVertexBase_2* concept, which is the vertex base required by the *Apollonius_graph_hierarchy_2<Gt,Agds>* class. The class *Apollonius_graph_hierarchy_vertex_base_2<Agvb>* is templated by a class *Agvb* which must be a model of the *ApolloniusGraphVertexBase_2* concept.

```
#include <CGAL/Apollonius_graph_hierarchy_vertex_base_2.h>
```

Inherits From

Agvb

Is Model for the Concepts

ApolloniusGraphHierarchyVertexBase_2

Creation

<i>Apollonius_graph_hierarchy_vertex_base_2<Agvb></i> <i>hvb</i> ;	Default constructor.
<i>Apollonius_graph_hierarchy_vertex_base_2<Agvb></i> <i>hvb</i> (<i>Site_2</i> <i>s</i>);	Constructs a vertex associated with the site <i>s</i> and embedded at the center of <i>s</i> .
<i>Apollonius_graph_hierarchy_vertex_base_2<Agvb></i> <i>hvb</i> (<i>Site_2</i> <i>s</i> , <i>Face_handle</i> <i>f</i>);	Constructs a vertex associated with the site <i>s</i> , embedded at the center of <i>s</i> , and pointing to the face associated with the face handle <i>f</i> .

See Also

ApolloniusGraphVertexBase_2
ApolloniusGraphHierarchyVertexBase_2
CGAL::Apollonius_graph_hierarchy_vertex_base_2<Gt,StoreHidden>

Chapter 28

2D Voronoi Diagram Adaptor

Menelaos Karavelas

Contents

28.1 Introduction	1739
28.2 Software Design	1741
28.3 The Adaptation Traits	1742
28.4 The Adaptation Policy	1743
28.4.1 Efficiency Considerations	1745
28.5 Examples	1745

This chapter describes an adaptor that adapts two-dimensional triangulated Delaunay graphs to the corresponding Voronoi diagrams. We start with a few definitions and a description of the issues that this adaptor addresses in Section 28.1. The software design of the Voronoi diagram adaptor package is described in Section 28.2. In Section 28.3 we discuss the traits required for performing the adaptation, and finally in Section 28.5 we present a few examples using this adaptor.

28.1 Introduction

A Voronoi diagram is typically defined for a set of objects, also called sites in the sequel, that lie in some space Σ and a distance function that measures the distance of a point x in Σ from an object in the object set. In this package we are interested in planar Voronoi diagrams, so in the sequel the space Σ will be the space \mathbb{R}^2 . Let $\mathcal{S} = \{S_1, S_2, \dots, S_n\}$ be our set of sites and let $\delta(x, S_i)$ denote the distance of a point $x \in \mathbb{R}^2$ from the site S_i . Given two sites S_i and S_j , the set V_{ij} of points that are closer to S_i than to S_j with respect to the distance function $\delta(x, \cdot)$ is simply the set:

$$V_{ij} = \{x \in \mathbb{R}^2 : \delta(x, S_i) < \delta(x, S_j)\}.$$

We can then define the set V_i of points on the plane that are closer to S_i than to any other object in \mathcal{S} as:

$$V_i = \bigcap_{i \neq j} V_{ij}.$$

The set V_i is said to be the *Voronoi cell* or *Voronoi face* of the site S_i . The locus of points on the plane that are equidistant from exactly two sites S_i and S_j is called a *Voronoi bisector*. A point that is equidistant to three or more objects in \mathcal{S} is called a *Voronoi vertex*. A simply connected subset of a Voronoi bisector is called a *Voronoi*

edge. The collection of Voronoi faces, edges and vertices is called the *Voronoi diagram* of the set S with respect to the distance function $\delta(x, \cdot)$, and it turns out that it is a subdivision of the plane, i.e., it is a planar graph.

We typically think of faces as 2-dimensional objects, edges as 1-dimensional objects and vertices as 0-dimensional objects. However, this may not be the case for several combinations of sites and distance functions (for example points in \mathbb{R}^2 under the L_1 or the L_∞ distance can produce 2-dimensional Voronoi edges). We call a Voronoi diagram *nice* if no such artifacts exist, i.e., if all vertices edges and faces are 0-, 1- and 2-dimensional, respectively.

Even nice Voronoi diagrams can end up being not so nice. The cell of a site can in general consist of several disconnected components. Such a case can happen, for example, when we consider weighted points $Q_i = (p_i, \lambda_i)$, where $p_i \in \mathbb{R}^2$, $\lambda_i \in \mathbb{R}$, and the distance function is the Euclidean distance multiplied by the weight of each site, i.e., $\delta_M(x, Q_i) = \lambda_i \|x - p_i\|$, where $\|\cdot\|$ denotes the Euclidean norm. In this package we are going to restrict ourselves to nice Voronoi diagrams that have the property that the Voronoi cell of each site is a simply connected region of the plane. We are going to call such Voronoi diagrams *simple Voronoi diagrams*. Examples of simple Voronoi diagrams include the usual Euclidean Voronoi diagram of points, the Euclidean Voronoi diagram of a set of disks on the plane (i.e., the Apollonius diagram), the Euclidean Voronoi diagram of a set of disjoint convex objects on the plane, or the power or (Laguerre) diagram for a set of circles on the plane. In fact every instance of an *abstract Voronoi diagram* in the sense of Klein [Kle89] is a simple Voronoi diagram in our setting. In the sequel when we refer to Voronoi diagrams we will refer to simple Voronoi diagrams.

In many cases we are not really interested in computing the Voronoi diagram itself, but rather its dual graph, called the *Delaunay graph*. In general the Delaunay graph is a planar graph, each face of which consists of at least three edges. Under the non-degeneracy assumption that no point on the plane is equidistant, under the distance function, to more than three sites, the Delaunay graph is a planar graph with triangular faces. In certain cases this graph can actually be embedded with straight line segments in which case we talk about a triangulation. This is the case, for example, for the Euclidean Voronoi diagram of points, or the power diagram of a set of circles. The dual graphs are, respectively, the Delaunay triangulation and the regular triangulation of the corresponding site sets. Graphs of non-constant non-uniform face complexity can be undesirable in many applications, so typically we end up triangulating the non-triangular faces of the Delaunay graph. Intuitively this amounts to imposing an implicit or explicit perturbation scheme during the construction of the Delaunay graph, that perturbs the input sites in such a way so as not to have degenerate configurations.

Choosing between computing the Voronoi diagram or the (triangulated) Delaunay graph is a major decision while implementing an algorithm. It heavily affects the design and choice of the different data structures involved. Although in theory the two approaches are entirely equivalent, it is not so straightforward to go from one representation to the other. The objective of this package is to provide a generic way of going from triangulated Delaunay graphs to planar subdivisions represented through a DCEL data structure. The goal is to provide an adaptor that gives the look and feel of a DCEL data structure, although internally it keeps a graph data structure representing triangular graphs.

The adaptation might seem straightforward at a first glance, and more or less this is the case; after all one graph is the dual of the other. The situation becomes complicated whenever we want to treat artifacts of the representation used. Suppose for example that we have a set of sites that contains subsets of sites in degenerate positions. The computed triangulated Delaunay graph has triangular faces that may be the result of an implicit or explicit perturbation scheme. The dual of such a triangulated Delaunay graph is a Voronoi diagram that has all its vertices of degree 3, and for that purpose we are going to call it a *degree-3 Voronoi diagram* in order to distinguish it from the true Voronoi diagram of the input sites. A degree-3 Voronoi diagram can have degenerate features, namely Voronoi edges of zero length, and/or Voronoi faces of zero area. Although we can potentially treat such artifacts, they are nonetheless artifacts of the algorithm we used and do not correspond to the true geometry of the Voronoi diagram.

The manner that we treat such issues in this package in a generic way is by defining an *adaptation policy*. The adaptation policy is responsible for determining which features in the degree-3 Voronoi diagram are to be rejected and which not. The policy to be used can vary depending on the application or the intended usage of

the resulting Voronoi diagram. What we care about is that firstly the policy itself is consistent and, secondly, that the adaptation is also done in a consistent manner. The latter is the responsibility of the adaptor provided by this package, whereas the former is the responsibility of the implementor of a policy.

In this package we currently provide two types of adaptation policies. The first one is the simplest: we reject no feature of the degree-3 Voronoi diagram; we call such a policy an *identity policy* since the Voronoi diagram produced is identical to the degree-3 Voronoi diagram. The second type of policy eliminates the degenerate features from the degree-3 Voronoi diagram yielding the true geometry of the Voronoi diagram of the input sites; we call such policies *degeneracy removal policies*.

Delaunay graphs can be mutable or non-mutable. By mutable we mean that sites can be inserted or removed at any time, in an entirely on-line fashion. By non-mutable we mean that once the Delaunay graph has been created, no changes, with respect to the set of sites defining it, are allowed. If the Delaunay graph is a non-mutable one, then the Voronoi diagram adaptor is a non-mutable adaptor as well.

If the Delaunay graph is mutable then the question of whether the Voronoi diagram adaptor is also mutable is slightly more complex to answer. As long as the adaptation policy used does not maintain a state, the Voronoi diagram adaptor is a mutable one; this is the case, for example, with our identity policy or the degeneracy removal policies. If, however, the adaptor maintains a state, then whether it is mutable or non-mutable really depends on whether its state can be updated after every change in the Delaunay graph. Such policies are our caching degeneracy removal policies: some of them result in mutable adaptors others result in non-mutable ones. In Section 28.4 we discuss the issue in more detail.

28.2 Software Design

The *Voronoi_diagram_2*<DG,AT,AP> class is parameterized by three template parameters. The first one must be a model of the *DelaunayGraph_2* concept. It corresponds to the API required by an object representing a Delaunay graph. All classes of CGAL that represent Delaunay diagrams are models of this concept, namely, Delaunay triangulations, regular triangulations, Apollonius graphs and segment Delaunay graphs. The second template parameter must be a model of the *AdaptationTraits_2* concept. We discuss this concept in detail in Section 28.3. The third template parameter must be model of the *AdaptationPolicy_2* concept, which we discuss in detail in Section 28.4.

The *Voronoi_diagram_2*<DG,AT,AP> class has been intentionally designed to provide an API similar to the arrangements class in CGAL: Voronoi diagrams are special cases of arrangements after all. The API of the two classes, however, could not be identical. The reason is that arrangements in CGAL do not yet support more than one unbounded faces, or equivalently, cannot handle unbounded curves. On the contrary, a Voronoi diagram defined over at least two generating sites, has at least two unbounded faces.

On a more technical level, the *Voronoi_diagram_2*<DG,AT,AP> class imitates the representation of the Voronoi diagram (seen as a planar subdivision) by a DCEL (Doubly Connected Edge List) data structure. We have vertices (the Voronoi vertices), halfedges (oriented versions of the Voronoi edges) and faces (the Voronoi cells). In particular, we can basically perform every operation we can perform in a standard DCEL data structure:

- go from a halfedge to its next and previous in the face;
- go from one face to an adjacent one through a halfedge and its twin (opposite) halfedge;
- walk around the boundary of a face;
- enumerate/traverse the halfedges incident to a vertex
- from a halfedge, access the adjacent face;

- from a face, access an adjacent halfedges;
- from a halfedges, access its source and target vertices;
- from a vertex, access an incident halfedge.

In addition to the above possibilities for traversal, we can also traverse the following features through iterators:

- the vertices of the Voronoi diagram;
- the edges or halfedges of the Voronoi diagram;
- the faces of the Voronoi diagram;
- the bounded faces of the Voronoi diagram;
- the bounded halfedges of the Voronoi diagram;
- the unbounded faces of the Voronoi diagram;
- the unbounded halfedges of the Voronoi diagram;
- the sites defining the Voronoi diagram.

Finally, depending on the adaptation traits passed to the Voronoi diagram adaptor, we can perform point location queries, namely given a point p we can determine the feature of the Voronoi diagram (vertex, edge, face) on which p lies.

28.3 The Adaptation Traits

The *AdaptationTraits_2* concept defines the types and functor required by the adaptor in order to access geometric information in the Delaunay graph that is needed by the *Voronoi_diagram_2<DG,AT,AP>* class. In particular, it provides functors for accessing sites in the Delaunay graph and constructing Voronoi vertices from their dual faces in the Delaunay graph. Finally, it defines a tag that indicates whether nearest site queries are to be supported by the Voronoi diagram adaptor. If such queries are to be supported, a functor is required.

Given a query point, the nearest site functor should return information related to how many and which sites of the Voronoi diagram are at equal and minimal distance from the query point. In particular, if the query point is closest to a single site, the vertex handle of the Delaunay graph corresponding to this site is returned. If the query point is closest to exactly two site, the edge of the Delaunay graph that is dual to the Voronoi edges on which the query point lies is returned. If three (or more) sites are closest to the query point, then the query point coincides with a vertex in the Voronoi diagram, and the face handle of the face in the Delaunay graph that is dual to the Voronoi vertex is returned. This way of abstracting the point location mechanism allows for multiple different point location strategies, which are passed to the Voronoi diagram adaptor through different models of the *AdaptationTraits_2* concept. The point location and nearest sites queries of the *Voronoi_diagram_2<DG,AT,AP>* class use internally this nearest site query functor.

In this package we provide four adaptation traits classes, all of which support nearest site queries:

- The *Apollonius_graph_adaptation_traits_2<AG2>* class: it provides the adaptation traits for Apollonius graphs.
- The *Delaunay_triangulation_adaptation_traits_2<DT2>* class: it provides the adaptation traits for Delaunay triangulations.

- The *Regular_triangulation_adaptation_traits_2<RT2>* class: it provides the adaptation traits for regular triangulations.
- The *Segment_Delaunay_graph_adaptation_traits_2<SDG2>* class: it provides the adaptation traits for segment Delaunay graphs.

28.4 The Adaptation Policy

As mentioned above, when we perform the adaptation of a triangulated Delaunay graph to a Voronoi diagram, a question that arises is whether we want to eliminate certain features of the Delaunay graph when we construct its Voronoi diagram representation (such features could be the Voronoi edges of zero length or, for the Voronoi diagram of a set of segments forming a polygon, all edges outside the polygon). The manner that we treat such issues in this package in a generic way is by defining an adaptation policy. The adaptation policy is responsible for determining which features in the degree-3 Voronoi diagram are to be rejected and which not. The policy to be used can vary depending on the application or the intended usage of the resulting Voronoi diagram.

The concept *AdaptationPolicy_2* defines the requirements on the predicate functors that determine whether a feature of the triangulated Delaunay graph should be rejected or not. More specifically it defines an *Edge_rejector* and a *Face_rejector* functor that answer the question: “should this edge (face) of the Voronoi diagram be rejected?”. In addition to the edge and face rejectors the adaptation policy defines a tag, the *Has_inserter* tag. This tag is either set to *CGAL::Tag_true* or to *CGAL::Tag_false*. Semantically it determines if the adaptor is allowed to insert sites in an on-line fashion (on-line removals are not yet supported). In the former case, i.e., when on-line site insertions are allowed, an additional functor is required, the *Site_inserter* functor. This functor takes a reference to a Delaunay graph and a site, and inserts the site in the Delaunay graph. Upon successful insertion, a handle to the vertex representing the site in the Delaunay graph is returned.

We have implemented two types of policies that provide two different ways for answering the question of which features of the Voronoi diagram to keep and which to discard. The first one is called the *identity policy* and corresponds to the *Identity_policy_2<DG,VT>* class. This policy is in some sense the simplest possible one, since it does not reject any feature of the Delaunay graph. The Voronoi diagram provided by the adaptor is the true dual (from the graph-theoretical point of view) of the triangulated Delaunay graph adapted. This policy assumes that the Delaunay graph adapted allows for on-line insertions, and the *Has_inserter* tag is set to *CGAL::Tag_true*. A default site inserter functor is also provided.

The second type of policy we provide is called *degeneracy removal policy*. If the set of sites defining the triangulated Delaunay graph contains subsets of sites in degenerate configurations, the graph-theoretical dual of the triangulated Delaunay graph has edges and potentially faces that are geometrically degenerate. By that we mean that the dual of the triangulated Delaunay graph can have Voronoi edges of zero length or Voronoi faces/cells of zero area. Such features may not be desirable and ideally we would like to eliminate them. The degeneracy removal policies eliminate exactly these features and provide a Voronoi diagram where all edges have non-zero length and all cells have non-zero area. More specifically, in these policies the *Edge_rejector* and *Face_rejector* functors reject the edges and vertices of the Delaunay graph that correspond to dual edges and faces that have zero length and area, respectively. In this package we provide four degeneracy removal policies, namely:

- The *Apollonius_graph_degeneracy_removal_policy_2<AG2>* class: it provides an adaptation policy for removing degeneracies when adapting an Apollonius graph to an Apollonius diagram.
- The *Delaunay_triangulation_degeneracy_removal_policy_2<DT2>* class: it provides an adaptation policy for removing degeneracies when adapting a Delaunay triangulation to a point Voronoi diagram.
- The *Regular_triangulation_degeneracy_removal_policy_2<RT2>* class: it provides an adaptation policy for removing degeneracies when adapting a regular triangulation to a power diagram

- The *Segment_Delaunay_graph_degeneracy_removal_policy_2<SDG2>* class: it provides an adaptation policy for removing degeneracies when adapting a segment Delaunay graph to a segment Voronoi diagram.

A variation of the degeneracy removal policies are the *caching degeneracy removal policies*. In these policies we cache the results of the edge and face rejectors. In particular, every time we want to determine, for example, if an edge of the Delaunay graph has, as dual edge in the Voronoi diagram, an edge of zero length, we check if the result has already been computed. If yes, we simply return the outcome. If not, we perform the necessary geometric tests, compute the answer, cache it and return it. Such a policy really pays off when we have a lot of degenerate data in our input set of sites. Verifying whether a Voronoi edge is degenerate or not implies computing the outcome of a predicate in a possibly degenerate or near degenerate configuration, which is typically very costly (compared to computing the same predicate in a generic configuration). To avoid this cost every single time we want to check if a Voronoi edge is degenerate or not, we compute the result of the geometric predicate the first time the adaptor asks for it, and simply lookup the answer in the future. In this package we provide four caching degeneracy removal policies, one per degeneracy removal policy mentioned above. Intentionally, we have not indicated the value of the *Has_inserter* tag for the degeneracy removal and caching degeneracy removal policies. The issue is discussed in detail in the sequel.

We raised the question above, as to whether the adaptor is a mutable or non-mutable one, in the sense of whether we can add/remove sites in an on-line fashion. The answer to this question depends on: (1) whether the Delaunay graph adapted allows for on-line insertions/removals and (2) whether the adaptation policies maintains a state and whether this state is easily maintainable when we want to allow for on-line modifications.

The way we indicate if we allow on-line insertions of sites is via the *Has_inserter* tag (as mentioned, on-line removals are currently not supported). The *Has_inserter* tag has two possible values, namely, *CGAL::Tag_true* and *CGAL::Tag_false*. The value *CGAL::Tag_true* indicates that the Delaunay graph allows for on-line insertions, whereas the value *CGAL::Tag_false* indicates the opposite. Note that these values *do not* indicate if the Delaunay graph supports on-line insertions, but rather whether the Voronoi diagram adaptor should be able to perform on-line insertions or not. This delicate point will become clearer below.

Let us consider the various scenarios. If the Delaunay graph is non-mutable, the Voronoi diagram adaptor cannot perform on-line insertions of sites. In this case not only degeneracy removal policies, but rather every single adaptation policy for adapting the Delaunay graph in question should have the *Has_inserter* tag set to *CGAL::Tag_false*.

If the Delaunay graph is mutable, i.e., on-line site insertions are allowed, we can choose between two types of adaptation policies, those that allow these on-line insertions and those that do not. In the former case the *Has_inserter* tag should be set to *CGAL::Tag_true*, whereas in the latter to *CGAL::Tag_false*. In other words, even if the Delaunay graph is mutable, we can choose (by properly determining the value of the *Has_inserter* tag) if the adaptor should be mutable as well. At a first glance it may seem excessive to restrict existing functionality. There are situations, however, where such a choice is necessary.

Consider a caching degeneracy removal policy. If we do not allow for on-line insertions then the cached quantities are always valid since the Voronoi diagram never changes. If we allow for on-line insertions the Voronoi diagram can change, which implies that the results of the edge and faces degeneracy testers that we have cached are no longer valid or relevant. In these cases, we need to somehow update these cached results, and ideally we would like to do this in an efficient manner. The inherent dilemma in the above discussion is whether the Voronoi diagram adaptor should be able to perform on-line insertions of sites. The answer to this question in this framework is given by the *Has_inserter* tag. If the tag is set to *CGAL::Tag_false* the adaptor cannot insert sites on-line, whereas if the tag is set to *CGAL::Tag_true* the adaptor can add sites on-line. In other words, the *Has_inserter* tag determines how the Voronoi diagram adaptor should behave, and this is enough from the adaptor's point of view.

From the point of view of a policy writer the dilemma is still there: should the policy allow for on-line insertions or not? The answer really depends on what are the consequences of such a choice. For a policy that

has no state, such as our degeneracy removal policies, it is natural to set the *Has_inserter* tag to *CGAL::Tag_true*. For our caching degeneracy removal policies, our choice was made on the grounds of whether we can update the cached results efficiently when insertions are performed. For CGAL's Apollonius graphs, Delaunay triangulation and regular triangulations it is possible to ask what are the edges and faces of the Delaunay graph that are to be destroyed when a query site is inserted. This is done via the *get_conflicts* method provided by these classes. Using the outcome of the *get_conflicts* method the site inserter can first update the cached results (i.e., indicate which are invalidated) and then perform the actual insertion. Such a method does not yet exist for segment Delaunay graphs. We have thus chosen to support on-line insertions for all non-caching degeneracy removal policies. The caching degeneracy removal policy for segment Delaunay graphs does not support on-line insertions, whereas the remaining three caching degeneracy removal policies support on-line insertions.

28.4.1 Efficiency Considerations

One last item that merits some discussion are the different choices from the point of view of time- and space-efficiency.

As far as the Voronoi diagram adaptor is concerned, only a copy of the adaptation traits and a copy of the adaptation policy are stored in it. The various adaptation traits classes we provide are empty classes (i.e., they do not store anything). The major time and space efficiency issues arise from the various implementations of the adaptation policies. Clearly, the identity policy has no dominant effect on neither the time or space efficiency. The costs when choosing this policy are due to the underlying Delaunay graph.

The non-caching degeneracy removal policies create a significant time overhead since every time we want to access a feature of the Voronoi diagram, we need to perform geometric tests in order to see if this feature or one of its neighboring ones has been rejected. Such a policy is acceptable if we know we are away from degeneracies or for small input sizes. In the case of the segment Delaunay graph, it is also the only policy we provide that at the same time removes degeneracies and allows for on-line insertion of sites. Caching policies seem to be the best choice for moderate to large input sizes (1000 sites and more). They do not suffer from the problem of dealing with degenerate configurations, but since they cache the results, they increase the space requirements by linear additive factor. To conclude, if the user is interested in getting a Voronoi diagram without degenerate features and knows all sites in advance, the best course of action is to insert all sites at construction time and use a caching degeneracy removal policy. This strategy avoids the updates of the cached results after each individual insertion, due to the features of the Voronoi diagram destroyed because of the site inserted.

28.5 Examples

In this section we present an example that shows how to perform point location queries.

```
// examples/Voronoi_diagram_2/point_location.C

#include <CGAL/basic.h>

// standard includes
#include <iostream>
#include <fstream>
#include <cassert>

// includes for defining the Voronoi diagram adaptor
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
```

```

#include <CGAL/Voronoi_diagram_2.h>
#include <CGAL/Delaunay_triangulation_adaptation_traits_2.h>
#include <CGAL/Delaunay_triangulation_adaptation_policies_2.h>

// typedefs for defining the adaptor
typedef CGAL::Exact_predicates_inexact_constructions_kernel      K;
typedef CGAL::Delaunay_triangulation_2<K>                        DT;
typedef CGAL::Delaunay_triangulation_adaptation_traits_2<DT>     AT;
typedef CGAL::Delaunay_triangulation_caching_degeneracy_removal_policy_2<DT> AP;
typedef CGAL::Voronoi_diagram_2<DT,AT,AP>                        VD;

// typedef for the result type of the point location
typedef AT::Site_2        Site_2;
typedef AT::Point_2       Point_2;

typedef VD::Locate_result  Locate_result;
typedef VD::Vertex_handle  Vertex_handle;
typedef VD::Face_handle    Face_handle;
typedef VD::Halfedge_handle Halfedge_handle;
typedef VD::Ccb_halfedge_circulator Ccb_halfedge_circulator;

void print_endpoint(Halfedge_handle e, bool is_src) {
    std::cout << "\t";
    if ( is_src ) {
        if ( e->has_source() ) std::cout << e->source()->point() << std::endl;
        else std::cout << "point at infinity" << std::endl;
    } else {
        if ( e->has_target() ) std::cout << e->target()->point() << std::endl;
        else std::cout << "point at infinity" << std::endl;
    }
}

int main()
{
    std::ifstream ifs("data/datal.dt.cin");
    assert( ifs );

    VD vd;

    Site_2 t;
    while ( ifs >> t ) { vd.insert(t); }
    ifs.close();

    assert( vd.is_valid() );

    std::ifstream ifq("data/queries1.dt.cin");
    assert( ifq );

    Point_2 p;
    while ( ifq >> p ) {
        std::cout << "Query point (" << p.x() << ", " << p.y()
            << ") lies on a Voronoi " << std::flush;

        Locate_result lr = vd.locate(p);
    }
}

```

```

if ( Vertex_handle* v = boost::get<Vertex_handle>(&lr) ) {
    std::cout << "vertex." << std::endl;
    std::cout << "The Voronoi vertex is:" << std::endl;
    std::cout << "\t" << (*v)->point() << std::endl;
} else if ( Halfedge_handle* e = boost::get<Halfedge_handle>(&lr) ) {
    std::cout << "edge." << std::endl;
    std::cout << "The source and target vertices "
        << "of the Voronoi edge are:" << std::endl;
    print_endpoint(*e, true);
    print_endpoint(*e, false);
} else if ( Face_handle* f = boost::get<Face_handle>(&lr) ) {
    std::cout << "face." << std::endl;
    std::cout << "The vertices of the Voronoi face are"
        << " (in counterclockwise order):" << std::endl;
    Ccb_halfedge_circulator ec_start = (*f)->outer_ccb();
    Ccb_halfedge_circulator ec = ec_start;
    do {
        print_endpoint(ec, false);
    } while ( ++ec != ec_start );
}
std::cout << std::endl;
}
ifq.close();

return 0;
}

```


2D Voronoi Diagram Adaptor Reference Manual

Menelaos Karavelas

CGAL provides the class `CGAL::Voronoi_diagram_2<DG,AT,AP>` for adapting the various (triangulated) Delaunay graphs to Voronoi diagrams according to some adaptation policy. In particular, the class `CGAL::Voronoi_diagram_2<DG,AT,AP>` provides an API for the duals of (triangulated) Delaunay graphs, that makes them look like planar subdivisions. The adaptation policy is responsible for deciding which edges and faces of these duals should be eliminated. This is especially important when, for instance, we want to eliminate degenerate features in the Voronoi diagram that are the result of the fact that Delaunay graphs are always triangulated and are due to degenerate configurations in the generating data.

The three template parameters must be models of the *DelaunayGraph_2*, *AdaptationTraits_2* and *AdaptationPolicy_2* concepts, respectively. The first concept is related to the Delaunay graphs that are to be adapted, whereas the second one is responsible for manipulating/accessing in a unified way the geometry of a specific Voronoi diagram as well as for performing nearest site queries. The third template parameter corresponds to the chosen adaptation policy and provides the necessary types and functors needed for performing this adaptation.

28.6 Classified Reference Pages

Concepts

DelaunayGraph_2.....	page 1764
AdaptationTraits_2.....	page 1768
AdaptationPolicy_2.....	page 1770

Classes

<code>CGAL::Voronoi_diagram_2<DG,AT,AP></code>	page 1751
<code>CGAL::Voronoi_diagram_2<DG,AT,AP>::Halfedge</code>	page 1758
<code>CGAL::Voronoi_diagram_2<DG,AT,AP>::Vertex</code>	page 1760
<code>CGAL::Voronoi_diagram_2<DG,AT,AP>::Face</code>	page 1762
<code>CGAL::Apollonius_graph_adaptation_traits_2<AG2></code>	page 1772
<code>CGAL::Delaunay_triangulation_adaptation_traits_2<DT2></code>	page 1773
<code>CGAL::Regular_triangulation_adaptation_traits_2<RT2></code>	page 1774
<code>CGAL::Segment_Delaunay_graph_adaptation_traits_2<SDG2></code>	page 1775
<code>CGAL::Identity_policy_2<DG,AT></code>	page 1776

<i>CGAL::Apollonius_graph_degeneracy_removal_policy_2<AG2></i>	page 1777
<i>CGAL::Apollonius_graph_caching_degeneracy_removal_policy_2<AG2></i>	page 1781
<i>CGAL::Delaunay_triangulation_degeneracy_removal_policy_2<DT2></i>	page 1778
<i>CGAL::Delaunay_triangulation_caching_degeneracy_removal_policy_2<DT2></i>	page 1782
<i>CGAL::Regular_triangulation_degeneracy_removal_policy_2<RT2></i>	page 1779
<i>CGAL::Regular_triangulation_caching_degeneracy_removal_policy_2<RT2></i>	page 1783
<i>CGAL::Segment_Delaunay_graph_degeneracy_removal_policy_2<SDG2></i>	page 1780
<i>CGAL::Segment_Delaunay_graph_caching_degeneracy_removal_policy_2<SDG2></i>	page 1784

28.7 Alphabetical List of Reference Pages

<i>AdaptationPolicy_2</i>	page 1770
<i>AdaptationTraits_2</i>	page 1768
<i>Apollonius_graph_adaptation_traits_2<AG2></i>	page 1772
<i>Apollonius_graph_caching_degeneracy_removal_policy_2<AG2></i>	page 1781
<i>Apollonius_graph_degeneracy_removal_policy_2<AG2></i>	page 1777
<i>DelaunayGraph_2</i>	page 1764
<i>Delaunay_triangulation_adaptation_traits_2<DT2></i>	page 1773
<i>Delaunay_triangulation_caching_degeneracy_removal_policy_2<DT2></i>	page 1782
<i>Delaunay_triangulation_degeneracy_removal_policy_2<DT2></i>	page 1778
<i>Face</i>	page 1762
<i>Halfedge</i>	page 1758
<i>Identity_policy_2<DG,AT></i>	page 1776
<i>Regular_triangulation_adaptation_traits_2<RT2></i>	page 1774
<i>Regular_triangulation_caching_degeneracy_removal_policy_2<RT2></i>	page 1783
<i>Regular_triangulation_degeneracy_removal_policy_2<RT2></i>	page 1779
<i>Segment_Delaunay_graph_adaptation_traits_2<SDG2></i>	page 1775
<i>Segment_Delaunay_graph_caching_degeneracy_removal_policy_2<SDG2></i>	page 1784
<i>Segment_Delaunay_graph_degeneracy_removal_policy_2<SDG2></i>	page 1780
<i>Vertex</i>	page 1760
<i>Voronoi_diagram_2<DG,AT,AP></i>	page 1751

CGAL::Voronoi_diagram_2<DG,AT,AP>

Definition

The class *Voronoi_diagram_2<DG,AT,AP>* provides an adaptor that enables us to view a triangulated Delaunay graph as their dual subdivision, the Voronoi diagram. The class *Voronoi_diagram_2<DG,AT,AP>* is designed to provide an API that is similar to that of CGAL's arrangements.

The first template parameter of the *Voronoi_diagram_2<DG,AT,AP>* class corresponds to the triangulated Delaunay graph and must be a model of the *DelaunayGraph_2* concept. The second template parameter must be a model of the *AdaptationTraits_2* concept. The third template parameter must be a model of the *AdaptationPolicy_2* concept. The third template parameter defaults to *CGAL::Identity_policy_2<DG,AT>*.

```
#include <CGAL/Voronoi_diagram_2.h>
```

Refines

DefaultConstructible, *CopyConstructible*, *Assignable*

Types

<i>typedef DG</i>	<i>Delaunay_graph;</i>	A type for the dual Delaunay graph.
<i>typedef AT</i>	<i>Adaptation_traits;</i>	A type for the adaptation traits needed by the Voronoi diagram adaptor.
<i>typedef AP</i>	<i>Adaptation_policy;</i>	A type for the adaptation policy used.
<i>typedef Adaptation_traits::Point_2</i>	<i>Point_2;</i>	A type a point.
<i>typedef Adaptation_traits::Site_2</i>	<i>Site_2;</i>	A type for the sites of the Voronoi diagram.
<i>typedef Delaunay_graph::size_type</i>	<i>size_type;</i>	A type for sizes.
<i>typedef Delaunay_graph::Geom_traits</i>	<i>Delaunay_geom_traits;</i>	A type for the geometric traits of the Delaunay graph.
<i>typedef Delaunay_graph::Vertex_handle</i>	<i>Delaunay_vertex_handle;</i>	A type for the vertex handles of the Delaunay graph.
<i>typedef Delaunay_graph::Face_handle</i>	<i>Delaunay_face_handle;</i>	A type for the face handles of the Delaunay graph.
<i>typedef Delaunay_graph::Edge</i>	<i>Delaunay_edge;</i>	A type for the edges of the Delaunay graph.
<i>Voronoi_diagram_2<DG,AT,AP>:: Halfedge</i>	A type for the halfedges of the Voronoi diagram.	
<i>Voronoi_diagram_2<DG,AT,AP>:: Vertex</i>	A type for the vertices of the Voronoi diagram.	
<i>Voronoi_diagram_2<DG,AT,AP>:: Face</i>	A type for the faces of the Voronoi diagram.	

The vertices, edges and faces of the Voronoi diagram are accessed through *handles*, *iterators* and *circulators*. The iterators and circulators are all bidirectional and non-mutable. The circulators and iterators are assignable to the corresponding handle types, and they are also convertible to the corresponding handles.

Voronoi_diagram_2<DG,AT,AP>:: Halfedge_handle Handle for halfedges.

<i>Voronoi_diagram_2<DG,AT,AP>:: Vertex_handle</i>	Handle for vertices.
<i>Voronoi_diagram_2<DG,AT,AP>:: Face_handle</i>	Handle for faces.
<i>Voronoi_diagram_2<DG,AT,AP>:: Edge_iterator</i>	A type for an iterator over Voronoi edges. Edges are considered non-oriented. Its value type is <i>Halfedge</i> .
<i>Voronoi_diagram_2<DG,AT,AP>:: Halfedge_iterator</i>	A type for an iterator over Voronoi halfedges. Halfedges are oriented and come in pairs. Its value type is <i>Halfedge</i> .
<i>Voronoi_diagram_2<DG,AT,AP>:: Face_iterator</i>	A type for an iterator over Voronoi faces. Its value type is <i>Face</i> .
<i>Voronoi_diagram_2<DG,AT,AP>:: Vertex_iterator</i>	A type for an iterator over Voronoi vertices. Its value type is <i>Vertex</i> .
<i>Voronoi_diagram_2<DG,AT,AP>:: Halfedge_around_vertex_circulator</i>	A type for a circulator over the halfedges that have a common vertex as their target. Its value type is <i>Halfedge</i> .
<i>Voronoi_diagram_2<DG,AT,AP>:: Ccb_halfedge_circulator</i>	A type for a circulator over the halfedges on the boundary of a Voronoi face. Its value type of is <i>Halfedge</i> .
<i>Voronoi_diagram_2<DG,AT,AP>:: Unbounded_faces_iterator</i>	A type for an iterator over the unbounded faces of the Voronoi diagram. Its value type is <i>Face</i> .
<i>Voronoi_diagram_2<DG,AT,AP>:: Bounded_faces_iterator</i>	A type for an iterator over the bounded faces of the Voronoi diagram. Its value type is <i>Face</i> .
<i>Voronoi_diagram_2<DG,AT,AP>:: Unbounded_halfedges_iterator</i>	A type for an iterator over the unbounded halfedges of the Voronoi diagram. Its value type is <i>Halfedge</i> .
<i>Voronoi_diagram_2<DG,AT,AP>:: Bounded_halfedges_iterator</i>	A type for an iterator over the bounded halfedges of the Voronoi diagram. Its value type is <i>Halfedge</i> .
<i>Voronoi_diagram_2<DG,AT,AP>:: Site_iterator</i>	A type for an iterator over the sites of the Voronoi diagram. Its value type is <i>Site_2</i> .
<i>typedef boost::variant<Face_handle,Halfedge_handle,Vertex_handle></i>	<i>Locate_result;</i> The result type of the point location queries.

Creation

```
Voronoi_diagram_2<DG,AT,AP> vd( Adaptation_traits at = Adaptation_traits(),
                                Adaptation_policy ap = Adaptation_policy(),
                                Delaunay_geom_traits gt = Delaunay_geom_traits())
```

Creates a Voronoi diagram using *at* as adaptation traits and *ap* as adaptation policy; the underlying Delaunay graph is created using *gt* as geometric traits.


```
Voronoi_diagram_2<DG,AT,AP> vd( Delaunay_graph dg,
                                bool swap_dg = false,
                                Adaptation_traits at = Adaptation_traits(),
                                Adaptation_policy ap = Adaptation_policy())
```

Creates a Voronoi diagram from the Delaunay graph *dg* and using *at* as adaptation traits and *ap* as adaptation policy. The Delaunay graph *dg* is fully copied if *swap_dg* is set to *false*, or swapped with the one stored internally if *swap_dg* is set to *true*.

```
template<class Iterator>
Voronoi_diagram_2<DG,AT,AP> vd( Iterator first,
                                Iterator beyond,
                                Adaptation_traits at = Adaptation_traits(),
                                Adaptation_policy ap = Adaptation_policy(),
                                Delaunay_geom_traits gt = Delaunay_geom_traits())
```

Creates a Voronoi diagram using as sites the sites in the iterator range [*first*, *beyond*), *at* as adaptation traits and *ap* as adaptation policy; the underlying Delaunay graph is created using *gt* as geometric traits. *Iterator* must be a model of the *InputIterator* concept and its value type must be *Site_2*.

Access Methods

<i>Delaunay_graph</i>	<i>vd.dual()</i>	Returns a const reference to the dual graph, i.e., the Delaunay graph.
<i>Halfedge_handle</i>	<i>vd.dual(Delaunay_edge e)</i>	Returns a handle to the halfedge in the Voronoi diagram that is dual to the edge <i>e</i> in the Delaunay graph.
<i>Face_handle</i>	<i>vd.dual(Delaunay_vertex_handle v)</i>	Returns a handle to the face in the Voronoi diagram that is dual to the vertex corresponding to the vertex handle <i>v</i> in the Delaunay graph.
<i>Vertex_handle</i>	<i>vd.dual(Delaunay_face_handle f)</i>	Returns a handle to the vertex in the Voronoi diagram that is dual to the face corresponding to the face handle <i>f</i> in the Delaunay graph.
<i>Adaptation_traits</i>	<i>vd.adaptation_traits()</i>	Returns a reference to the Voronoi traits.
<i>Adaptation_policy</i>	<i>vd.adaptation_policy()</i>	Returns a reference to the adaptation policy.
<i>size_type</i>	<i>vd.number_of_vertices()</i>	Returns the number of Voronoi vertices.
<i>size_type</i>	<i>vd.number_of_faces()</i>	Returns the number of Voronoi faces (bounded and unbounded).
<i>size_type</i>	<i>vd.number_of_halfedges()</i>	Returns the number of halfedges (bounded and unbounded) in the Voronoi diagram. This is always an even number.
<i>size_type</i>	<i>vd.number_of_connected_components()</i>	Returns the number of connected components of the Voronoi skeleton.
<i>Face_handle</i>	<i>vd.unbounded_face()</i>	Returns one of the unbounded faces of the Voronoi diagram. If no unbounded faces exist (this can happen if the number of sites is zero) the default constructed face handle is returned.

<i>Face_handle</i>	<i>vd.bounded_face()</i>	Returns one of the bounded faces of the Voronoi diagram. If no bounded faces exist the default constructed face handle is returned.
<i>Halfedge_handle</i>	<i>vd.unbounded_halfedge()</i>	Returns one of the unbounded halfedges of the Voronoi diagram. If no unbounded halfedges exist the default constructed halfedge handle is returned.
<i>Halfedge_handle</i>	<i>vd.bounded_halfedge()</i>	Returns one of the bounded halfedges of the Voronoi diagram. If no bounded halfedges exist the default constructed halfedge handle is returned.

Traversal of the Voronoi diagram

A Voronoi diagram can be seen as a container of faces, vertices and halfedges. Therefore the Voronoi diagram provides several iterators and circulators that allow to traverse it.

Iterators

The following iterators allow respectively to visit the faces (all or only the unbounded/bounded ones), edges, halfedges (all or only the unbounded/bounded ones) and vertices of the Voronoi diagram. These iterators are non-mutable, bidirectional and their value types are respectively *Face*, *Halfedge*, *Halfedge* and *Vertex*. All iterators are convertible to the corresponding handles and are invalidated by any change in the Voronoi diagram.

<i>Face_iterator</i>	<i>vd.faces_begin()</i>	Starts at an arbitrary Voronoi face.
<i>Face_iterator</i>	<i>vd.faces_end()</i>	Past-the-end iterator.
<i>Unbounded_faces_iterator</i>	<i>vd.unbounded_faces_begin()</i>	Starts at an arbitrary unbounded Voronoi face.
<i>Unbounded_faces_iterator</i>	<i>vd.unbounded_faces_end()</i>	Past-the-end iterator.
<i>Bounded_faces_iterator</i>	<i>vd.bounded_faces_begin()</i>	Starts at an arbitrary bounded Voronoi face.
<i>Bounded_faces_iterator</i>	<i>vd.bounded_faces_end()</i>	Past-the-end iterator.
<i>Edge_iterator</i>	<i>vd.edges_begin()</i>	Starts at an arbitrary Voronoi edge.
<i>Edge_iterator</i>	<i>vd.edges_end()</i>	Past-the-end iterator.
<i>Halfedge_iterator</i>	<i>vd.halfedges_begin()</i>	Starts at an arbitrary Voronoi halfedge.
<i>Halfedge_iterator</i>	<i>vd.halfedges_end()</i>	Past-the-end iterator.
<i>Unbounded_halfedges_iterator</i>	<i>vd.unbounded_halfedges_begin()</i>	Starts at an arbitrary unbounded Voronoi edge.
<i>Unbounded_halfedges_iterator</i>	<i>vd.unbounded_halfedges_end()</i>	Past-the-end iterator.
<i>Bounded_halfedges_iterator</i>	<i>vd.bounded_halfedges_begin()</i>	Starts at an arbitrary bounded Voronoi edge.
<i>Bounded_halfedges_iterator</i>	<i>vd.bounded_halfedges_end()</i>	Past-the-end iterator.

<i>Vertex_iterator</i>	<i>vd.vertices_begin()</i>	Starts at an arbitrary Voronoi vertex.
<i>Vertex_iterator</i>	<i>vd.vertices_end()</i>	Past-the-end iterator.

The following iterator provides access to the sites that define the Voronoi diagram. Its value type is *Site_2*. It is invalidated by any change in the Voronoi diagram.

<i>Site_iterator</i>	<i>vd.sites_begin()</i>	Starts at an arbitrary site.
<i>Site_iterator</i>	<i>vd.sites_end()</i>	Past-the-end iterator.

Circulators

The Voronoi diagram adaptor also provides circulators that allow to visit all halfedges whose target is a given vertex – this is the *Halfedge_around_vertex_circulator*, as well as all halfedges on the boundary of a Voronoi face – this is the *Ccb_halfedge_circulator*. These circulators are non-mutable and bidirectional. The operator *operator++* moves the former circulator counterclockwise around the vertex while the *operator--* moves clockwise. The latter circulator is moved by the operator *operator++* to the next halfedge on the boundary in the counterclockwise sense, while *operator--* moves clockwise. When the *Ccb_halfedge_circulator* is defined over an infinite Voronoi face *f*, then applying *operator++* to a circulator corresponding to a halfedge whose target is not finite moves to the next infinite (or semi-infinite) halfedge of *f* in the counterclockwise sense. Similarly, applying *operator++* to a circulator corresponding to a halfedge whose source is not finite, moves to the previous infinite (or semi-infinite) halfedge of *f* in the clockwise sense. The *Halfedge_around_vertex_circulator* circulator is invalidated by any modification in the faces adjacent to the vertex over which it is defined. The *Ccb_halfedge_circulator* is invalidated by any modification in the face over which it is defined.

<i>Ccb_halfedge_circulator</i>	<i>vd.ccb_halfedges(Face_handle f)</i>	Returns a circulator over the halfedges on the boundary of <i>f</i> . The circulator is initialized to an arbitrary halfedge on the boundary of the Voronoi face <i>f</i> .
--------------------------------	-----------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<i>Ccb_halfedge_circulator</i>	<i>vd.ccb_halfedges(Face_handle f, Halfedge_handle h)</i>	Returns a circulator over the halfedges on the boundary of <i>f</i> . The circulator is initialized with the halfedge <i>h</i> . <i>Precondition:</i> The halfedge <i>h</i> must lie on the boundary of <i>f</i> .
--------------------------------	------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<i>Halfedge_around_vertex_circulator</i>	<i>vd.incident_halfedges(Vertex_handle v)</i>	Returns a circulator over the halfedges whose target is the Voronoi vertex <i>v</i> . The circulator is initialized to an arbitrary halfedge incident to <i>v</i> .
------------------------------------------	------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------

<i>Halfedge_around_vertex_circulator</i>	<i>vd.incident_halfedges(Vertex_handle v, Halfedge_handle h)</i>	Returns a circulator over the halfedges whose target is the Voronoi vertex <i>v</i> . The circulator is initialized with the halfedge <i>h</i> . <i>Precondition:</i> The vertex <i>v</i> must be the target vertex of the halfedge <i>h</i> .
------------------------------------------	-------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Insertion

Face_handle *vd.insert(Site_2 t)* Inserts the site *t* in the Voronoi diagram. A handle to the face corresponding to the Voronoi face of *t* in the Voronoi diagram is returned. If *t* has an empty Voronoi cell, the default constructed face handle is returned. This method is supported only if *Voronoi_traits::Has_inserter* is set to *CGAL::Tag_true*.

template<class Iterator>
size_type *vd.insert(Iterator first, Iterator beyond)*

Inserts, in the Voronoi diagram, the sites in the iterator range [*first*, *beyond*). The value type of *Iterator* must be *Site_2*. The number of sites in the iterator range is returned. This method is supported only if *Voronoi_traits::Has_inserter* is set to *CGAL::Tag_true*.

Queries

Locate_result *vd.locate(Point_2 p)* Performs point location for the query point *p*. In other words, the face, halfedge or vertex of the Voronoi diagram is found on which the point *p* lies. This method is supported only if *Voronoi_traits::Has_nearest_site_2* is set to *CGAL::Tag_true*.
Precondition: The Voronoi diagram must contain at least one face.

I/O

void *vd.file_output(std::ostream& os)* Writes the current state of the Voronoi diagram to the output stream *os*.
The following operator must be defined:
std::ostream& operator<<(std::ostream&, Delaunay_graph)

void *vd.file_input(std::istream& is)* Reads the current state of the Voronoi diagram from the input stream *is*.
The following operator must be defined:
std::istream& operator>>(std::istream&, Delaunay_graph)

std::ostream& *std::ostream& os << vd* Writes the current state of the Voronoi diagram to the output stream *os*.
The following operator must be defined:
std::ostream& operator<<(std::ostream&, Delaunay_graph)

std::istream& *std::istream& is >> vd* Reads the current state of the Voronoi diagram from the input stream *is*.
The following operator must be defined:
std::istream& operator>>(std::istream&, Delaunay_graph)

Validity check

bool *vd.is_valid()* Checks the validity of the dual Delaunay graph and the Voronoi diagram adaptor.

Miscellaneous

`void vd.clear()` Clears all contents of the Voronoi diagram.
`void vd.swap(other)` The Voronoi diagrams *other* and *vd* are swapped. *vd.swap(other)* should be preferred to *vd= other* or to *vd(other)* if *other* is deleted afterwards.

See Also

DelaunayGraph_2
AdaptationTraits_2
AdaptationPolicy_2
CGAL::Voronoi_diagram_2<DG,AT,AP>::Face
CGAL::Voronoi_diagram_2<DG,AT,AP>::Halfedge
CGAL::Voronoi_diagram_2<DG,AT,AP>::Vertex
CGAL::Delaunay_triangulation_2<Traits,Tds>
CGAL::Regular_triangulation_2<Traits,Tds>
CGAL::Triangulation_hierarchy_2<Tr> provided that *Tr* is a model of *DelaunayGraph_2*
CGAL::Segment_Delaunay_graph_2<Gt,DS>
CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>
CGAL::Apollonius_graph_2<Gt,Agds>
CGAL::Apollonius_graph_hierarchy_2<Gt,Agds>
CGAL::Apollonius_graph_adaptation_traits_2<AG2>
CGAL::Delaunay_triangulation_adaptation_traits_2<DT2>
CGAL::Regular_triangulation_adaptation_traits_2<RT2>
CGAL::Segment_Delaunay_graph_adaptation_traits_2<SDG2>
CGAL::Identity_policy_2<DG,AT>
CGAL::Apollonius_graph_degeneracy_removal_policy_2<AG2>
CGAL::Apollonius_graph_caching_degeneracy_removal_policy_2<AG2>
CGAL::Delaunay_triangulation_degeneracy_removal_policy_2<DT2>
CGAL::Delaunay_triangulation_caching_degeneracy_removal_policy_2<DT2>
CGAL::Regular_triangulation_degeneracy_removal_policy_2<RT2>
CGAL::Regular_triangulation_caching_degeneracy_removal_policy_2<RT2>
CGAL::Segment_Delaunay_graph_degeneracy_removal_policy_2<SDG2>
CGAL::Segment_Delaunay_graph_caching_degeneracy_removal_policy_2<SDG2>

CGAL::Voronoi_diagram_2<DG,AT,AP>::Halfedge

Definition

The class *Halfedge* is the class provided by the *Voronoi_diagram_2<DG,AT,AP>* class for Voronoi halfedges. Below we present its interface.

Is Model for the Concepts

DefaultConstructible, *CopyConstructible*, *Assignable*, *EqualityComparable*, *LessThanComparable*

Types

<i>Halfedge::Vertex</i>		A type for the vertices of the Voronoi diagram.
<i>Halfedge::Face</i>		A type for the faces of the Voronoi diagram.
<i>Halfedge::Vertex_handle</i>		Handle for the vertices of the Voronoi diagram.
<i>Halfedge::Face_handle</i>		Handle for the faces of the Voronoi diagram.
<i>Halfedge::Halfedge_handle</i>		Handle for the halfedges of the Voronoi diagram.
<i>Halfedge::Ccb_halfedge_circulator</i>		A type for a bidirectional circulator over the halfedges of the boundary of a Voronoi face. The value type of the circulator is <i>CGAL::Voronoi_diagram_2<DG,AT,AP>::Halfedge</i> and is convertible to <i>Halfedge_handle</i> .
<i>Halfedge::Delaunay_graph</i>		A type for the Delaunay graph. It is a model of the <i>DelaunayGraph_2</i> concept.
<i>typedef Delaunay_graph::Edge</i>	<i>Delaunay_edge;</i>	A type for the dual edge in the Delaunay graph.
<i>typedef Delaunay_graph::Vertex_handle</i>	<i>Delaunay_vertex_handle;</i>	A type for vertex handles in the Delaunay graph.

Access Methods

<i>Halfedge_handle</i>	<i>e.twin()</i>	Returns the twin halfedge.
<i>Halfedge_handle</i>	<i>e.opposite()</i>	Same as <i>e.twin()</i> .
<i>Halfedge_handle</i>	<i>e.next()</i>	Returns the next halfedge in the counterclockwise sense around the boundary of the face that <i>e</i> is incident to.
<i>Halfedge_handle</i>	<i>e.previous()</i>	Returns the previous halfedge in the counterclockwise sense around the boundary of the adjacent face.

<i>Face_handle</i>	<i>e.face()</i>	Returns the face that <i>e</i> is incident to.
<i>Vertex_handle</i>	<i>e.source()</i>	Returns the source vertex of <i>e</i> . <i>Precondition:</i> The source vertex must exist, i.e., <i>has_source()</i> must return <i>true</i> .
<i>Vertex_handle</i>	<i>e.target()</i>	Returns the target vertex of <i>e</i> . <i>Precondition:</i> The target vertex must exist, i.e., <i>has_target()</i> must return <i>true</i> .
<i>Ccb_halfedge_circulator</i>	<i>e.ccb()</i>	Returns a bidirectional circulator to traverse the halfedges on the boundary of the Voronoi face containing <i>e</i> . The circulator is initialized to <i>e</i> . Applying <i>operator++</i> (resp. <i>operator--</i>) to this circulator returns the next halfedge on the boundary of the face containing <i>e</i> in the counterclockwise (resp. clockwise) sense.
<i>Delaunay_edge</i>	<i>e.dual()</i>	Returns the corresponding dual edge in the Delaunay graph.

In the four methods below we consider Voronoi halfedges to be “parallel” to the *x*-axis, oriented from left to right.

<i>Delaunay_vertex_handle</i>	<i>e.up()</i>	Returns a handle to the vertex in the Delaunay graph corresponding to the defining site above the Voronoi edge.
<i>Delaunay_vertex_handle</i>	<i>e.down()</i>	Returns a handle to the vertex in the Delaunay graph corresponding to the defining site below the Voronoi edge.
<i>Delaunay_vertex_handle</i>	<i>e.left()</i>	Returns a handle to the vertex in the Delaunay graph corresponding to the defining site to the left of the Voronoi edge. <i>Precondition:</i> <i>has_source()</i> must be <i>true</i> .
<i>Delaunay_vertex_handle</i>	<i>e.right()</i>	Returns a handle to the vertex in the Delaunay graph corresponding to the defining site to the right of the Voronoi edge. <i>Precondition:</i> <i>has_target()</i> must be <i>true</i> .

Predicate Methods

<i>bool e.has_source()</i>	Returns <i>true</i> iff the halfedge corresponds to a bisecting segment or a bisecting ray oriented appropriately so that its apex is its source.
<i>bool e.has_target()</i>	Returns <i>true</i> iff the halfedge corresponds to a bisecting segment or a bisecting ray oriented appropriately so that its apex is its target.
<i>bool e.is_unbounded()</i>	Returns <i>true</i> iff the source or the target of the halfedge does not exist, i.e., if either of <i>has_source()</i> or <i>has_target()</i> return <i>false</i> .
<i>bool e.is_bisector()</i>	Returns <i>true</i> iff the Voronoi edge is an entire bisector.
<i>bool e.is_segment()</i>	Returns <i>true</i> iff the Voronoi edge has both a source and a target Voronoi vertex.
<i>bool e.is_ray()</i>	Returns <i>true</i> iff the Voronoi edge has either a source or a target Voronoi vertex, but not both; in other words it is a bisecting ray.
<i>bool e.is_valid()</i>	Returns <i>true</i> if the following conditions are met: the halfedge is not a rejected edge with respect to the chosen adaptation policy; the twin edge of its twin edge is itself; its adjacent face is not a rejected face with respect to the chosen adaptation policy; its source and target vertices are valid (provided they exist, of course); the previous of its next halfedge is itself and the next of its previous halfedge is itself.

See Also

CGAL::Voronoi_diagram_2<DG,AT,AP>
CGAL::Voronoi_diagram_2<DG,AT,AP>::Vertex
CGAL::Voronoi_diagram_2<DG,AT,AP>::Face
DelaunayGraph_2

CGAL::Voronoi_diagram_2<DG,AT,AP>::Vertex

Definition

The class *Vertex* is the Voronoi vertex class provided by the class *Voronoi_diagram_2<DG,AT,AP>* class. Below we present its interface.

Is Model for the Concepts

DefaultConstructible, *CopyConstructible*, *Assignable*, *EqualityComparable*, *LessThanComparable*

Types

<i>Vertex:: Halfedge</i>	A type for the halfedges of the Voronoi diagram.
<i>Vertex:: Face</i>	A type for the faces of the Voronoi diagram.
<i>Vertex:: Vertex_handle</i>	Handle for the vertices of the Voronoi diagram.
<i>Vertex:: Face_handle</i>	Handle for the faces of the Voronoi diagram.
<i>Vertex:: Halfedge_handle</i>	Handle for the halfedges of the Voronoi diagram.
<i>Vertex:: Point_2</i>	A type for the point represented by the vertex.
<i>Vertex:: size_type</i>	A type for sizes.
<i>Vertex:: Halfedge_around_vertex_circulator</i>	A type for a bidirectional circulator that allows to traverse all incident halfedges, i.e., all halfedges that have the vertex as their target. The value type of the circulator is <i>CGAL::Voronoi_diagram_2<DG,AT,AP>::Halfedge</i> and is convertible to <i>Halfedge_handle</i> .
<i>Vertex:: Delaunay_graph</i>	A type for the Delaunay graph. It is a model of the <i>DelaunayGraph_2</i> concept.
<i>typedef Delaunay_graph::Face_handle Delaunay_face_handle;</i>	A type for the handle of the dual face.
<i>typedef Delaunay_graph::Vertex_handle Delaunay_vertex_handle;</i>	A type for the vertex handles in the Delaunay graph.

Access Methods

<i>Halfedge_handle</i>	<i>v.halfedge()</i>	Returns an incident halfedge that has <i>v</i> as its target.
<i>size_type</i>	<i>v.degree()</i>	Returns the in-degree of the vertex, i.e. the number of halfedges that have <i>v</i> as their target.
<i>Point_2</i>	<i>v.point()</i>	Returns the point represented by the vertex.
<i>Delaunay_face_handle</i>	<i>v.dual()</i>	Returns a handle to the corresponding dual face in the Delaunay graph.

Delaunay_vertex_handle *v.site(unsigned int i)* Returns a handle to the vertex in the Delaunay graph corresponding to the $(i + 1)$ -th generating site of the Voronoi vertex.

Precondition: *i* must be smaller or equal to 2.

Halfedge_around_vertex_circulator *v.incident_halfedges()* Returns a bidirectional circulator that allows the traversal of the halfedges that have *v* as their target. Applying *operator++* (resp. *operator--*) to this circulator returns the next incident halfedge in the counterclockwise (resp. clockwise) sense.

Predicate Methods

bool v.is_incident_edge(Halfedge_handle e) Returns *true* if the halfedge *e* is incident to *v*.
bool v.is_incident_face(Face_handle e) Returns *true* if the face *f* is incident to *v*.
bool v.is_valid() Returns *true* if the following conditions are met: the dual face is not an infinite face; all incident halfedges have the vertex as their target.

See Also

CGAL::Voronoi_diagram_2<DG,AT,AP>
CGAL::Voronoi_diagram_2<DG,AT,AP>::Halfedge
CGAL::Voronoi_diagram_2<DG,AT,AP>::Face
DelaunayGraph_2

CGAL::Voronoi_diagram_2<DG,AT,AP>::Face

Definition

The class *Face* is the class provided by the *Voronoi_diagram_2<DG,AT,AP>* class for Voronoi faces. Below we present its interface.

Is Model for the Concepts

DefaultConstructible, *CopyConstructible*, *Assignable*, *EqualityComparable*, *LessThanComparable*

Types

<i>Face:: Vertex</i>	A type for the vertices of the Voronoi diagram.
<i>Face:: Halfedge</i>	A type for the halfedges of the Voronoi diagram.
<i>Face:: Vertex_handle</i>	Handle for the vertices of the Voronoi diagram.
<i>Face:: Face_handle</i>	Handle for the faces of the Voronoi diagram.
<i>Face:: Halfedge_handle</i>	Handle for the halfedges of the Voronoi diagram.
<i>Face:: Ccb_halfedge_circulator</i>	A type for a bidirectional circulator over the halfedges on the boundary of the face. The value type of the circulator is <i>CGAL::Voronoi_diagram_2<DG,AT,AP>::Halfedge</i> , and is convertible to <i>Halfedge_handle</i> .
<i>Face:: Delaunay_graph</i>	A type for the Delaunay graph. It is a model of the <i>DelaunayGraph_2</i> concept.
<i>typedef Delaunay_graph::Vertex_handle</i>	<i>Delaunay_vertex_handle</i> ; A type for the handle of the dual vertex.

Access Methods

<i>Halfedge_handle</i>	<i>f.halfedge()</i>	Returns an incident halfedge on the boundary of <i>f</i> .
<i>Ccb_halfedge_circulator</i>	<i>f.ccb()</i>	Returns a bidirectional circulator for traversing the halfedges on the boundary of <i>f</i> . The halfedges are traversed in counterclockwise order.
<i>Delaunay_vertex_handle</i>	<i>f.dual()</i>	Returns a handle to the corresponding dual vertex in the Delaunay graph.

Predicate Methods

<i>bool</i>	<i>f.is_unbounded()</i>	Returns <i>true</i> iff the face is an unbounded face in the Voronoi diagram.
<i>bool</i>	<i>f.is_halfedge_on_ccb(Halfedge e)</i>	Returns <i>true</i> iff <i>e</i> is a halfedge of the boundary of <i>f</i> .

bool *f.is_valid()*

Returns *true* iff the following conditions are met: the face is not rejected by the chosen adaptation policy; all its adjacent halfedges do not have zero length; all its adjacent halfedges return the face as their adjacent face.

See Also

CGAL::Voronoi_diagram_2<DG,AT,AP>

CGAL::Voronoi_diagram_2<DG,AT,AP>::Vertex

CGAL::Voronoi_diagram_2<DG,AT,AP>::Halfedge

DelaunayGraph_2

DelaunayGraph_2

Definition

The concept *DelaunayGraph_2* defines the requirements for the first template parameter of the *Voronoi_diagram_2*<DG,AT,AP> class. The *DelaunayGraph_2* concept essentially defines the requirements that a class representing a Delaunay graph must obey so that the Voronoi diagram adaptor can adapt it.

Refines

DefaultConstructible, *CopyConstructible*, *Assignable*

Types

<i>DelaunayGraph_2:: size_type</i>	A type for sizes.
<i>DelaunayGraph_2:: Geom_traits</i>	A type for the geometric traits associated with the Delaunay graph.
<i>DelaunayGraph_2:: Triangulation_data_structure</i>	A type for the underlying triangulation data structure. It must be a model of the concept <i>TriangulationDataStructure_2</i> .
<i>DelaunayGraph_2:: Vertex</i>	A type for the vertices of the Delaunay graph.
<i>DelaunayGraph_2:: Face</i>	A type for the faces of the Delaunay graph.
<i>typedef std::pair<Face_handle,int> Edge;</i>	The type of the edges of the Delaunay graph.
<i>DelaunayGraph_2:: Vertex_handle</i>	Handle to the vertices of the Delaunay graph.
<i>DelaunayGraph_2:: Face_handle</i>	Handle to the faces of the Delaunay graph.

The following iterators and circulators must be defined. All iterators and circulators must be assignable and convertible to their corresponding handles.

<i>DelaunayGraph_2:: All_edges_iterator</i>	A type for an iterator over all edges of the Delaunay graph. Its value type must be <i>Edge</i> .
<i>DelaunayGraph_2:: Finite_edges_iterator</i>	A type for an iterator over the finite edges of the Delaunay graph. Its value type must be <i>Edge</i> .
<i>DelaunayGraph_2:: All_faces_iterator</i>	A type for an iterator over all faces of the Delaunay graph. Its value type must be <i>Face</i> .
<i>DelaunayGraph_2:: Finite_faces_iterator</i>	A type for an iterator over the finite faces of the Delaunay graph. Its value type must be <i>Face</i> .
<i>DelaunayGraph_2:: All_vertices_iterator</i>	A type for an iterator over all vertices of the Delaunay graph. Its value type must be <i>Vertex</i> .
<i>DelaunayGraph_2:: Finite_vertices_iterator</i>	A type for an iterator over the finite vertices of the Delaunay graph. Its value type must be <i>Vertex</i> .
<i>DelaunayGraph_2:: Face_circulator</i>	A type for a circulator over the adjacent faces of a vertex of the Delaunay graph. Its value type must be <i>Face</i> .
<i>DelaunayGraph_2:: Vertex_circulator</i>	A type for a circulator over the adjacent vertices of a vertex of the Delaunay graph. Its value type must be <i>Vertex</i> .
<i>DelaunayGraph_2:: Edge_circulator</i>	A type for a circulator over the adjacent edges of a vertex of the Delaunay graph. Its value type must be <i>Edge</i> .

Creation

In addition to the default and copy constructors, as well as the assignment operator, the following constructors are required.

<i>DelaunayGraph_2</i> <i>dg</i> (<i>Geom_traits</i> <i>gt</i>);	Constructor that takes an instance of the geometric traits.
<i>template<class It></i> <i>DelaunayGraph_2</i> <i>dg</i> (<i>It</i> <i>first</i> , <i>It</i> <i>beyond</i>);	Constructor that takes an iterator range. The value type of the iterator must be the type of the sites of the Delaunay graph.
<i>template<class It></i> <i>DelaunayGraph_2</i> <i>dg</i> (<i>It</i> <i>first</i> , <i>It</i> <i>beyond</i> , <i>Geom_traits</i> <i>gt</i>);	Constructor that takes an iterator range and an instance of the geometric traits. The value type of the iterator must be the type of the sites of the Delaunay graph.

Access methods

<i>Triangulation_data_structure</i>	<i>dg.tds()</i>	Returns a reference to the underlying triangulation data structure.
<i>Geom_traits</i>	<i>dg.geom_traits()</i>	Returns a reference to the geometric traits object.
<i>Vertex_handle</i>	<i>dg.infinite_vertex()</i>	Returns a handle to the infinite vertex.
<i>Vertex_handle</i>	<i>dg.finite_vertex()</i>	Returns a handle to a finite vertex, provided there exists one.
<i>Face_handle</i>	<i>dg.infinite_face()</i>	Returns a handle to a face incident to the infinite vertex.
<i>int</i>	<i>dg.dimension()</i>	Returns the dimension of the Delaunay graph.
<i>size_type</i>	<i>dg.number_of_vertices()</i>	Returns the number of finite vertices.
<i>size_type</i>	<i>dg.number_of_faces()</i>	Returns the number of faces (both finite and infinite).

Traversal of the Delaunay graph

A model of the *DelaunayGraph_2* concept must provide several iterators and circulators that allow to traverse it (completely or partially). All iterators and circulators must be convertible to the corresponding handles.

Face, Edge and Vertex Iterators

The following iterators must allow, respectively, to visit finite faces, finite edges and finite vertices of the Delaunay graph. These iterators must be non-mutable, bidirectional and their value types are respectively *Face*, *Edge* and *Vertex*.

<i>Finite_vertices_iterator</i>	<i>dg.finite_vertices_begin()</i>	Starts at an arbitrary finite vertex.
<i>Finite_vertices_iterator</i>	<i>dg.finite_vertices_end()</i>	Past-the-end iterator.
<i>Finite_edges_iterator</i>	<i>dg.finite_edges_begin()</i>	Starts at an arbitrary finite edge.
<i>Finite_edges_iterator</i>	<i>dg.finite_edges_end()</i>	Past-the-end iterator.
<i>Finite_faces_iterator</i>	<i>dg.finite_faces_begin()</i>	Starts at an arbitrary finite face.

<i>Finite_faces_iterator</i>	<i>dg.finite_faces_end()</i>	Past-the-end iterator.
------------------------------	------------------------------	------------------------

The following iterators must allow, respectively, to visit all (both finite and infinite) faces, edges and vertices of the Delaunay graph. These iterators are non-mutable, bidirectional and their value types are respectively *Face*, *Edge* and *Vertex*.

<i>All_vertices_iterator</i>	<i>dg.all_vertices_begin()</i>	Starts at an arbitrary vertex.
<i>All_vertices_iterator</i>	<i>dg.all_vertices_end()</i>	Past-the-end iterator.
<i>All_edges_iterator</i>	<i>dg.all_edges_begin()</i>	Starts at an arbitrary edge.
<i>All_edges_iterator</i>	<i>dg.all_edges_end()</i>	Past-the-end iterator.
<i>All_faces_iterator</i>	<i>dg.all_faces_begin()</i>	Starts at an arbitrary face.
<i>All_faces_iterator</i>	<i>dg.all_faces_end()</i>	Past-the-end iterator.

Face, Edge and Vertex Circulators

A model of the *DelaunayGraph_2* concept must also provide circulators that allow to visit, respectively, all faces or edges incident to a given vertex or all vertices adjacent to a given vertex. These circulators are non-mutable and bidirectional. The operator *operator++* must move the circulator counterclockwise around the vertex while the *operator--* must move the circulator clockwise.

<i>Face_circulator</i>	<i>dg.incident_faces(Vertex_handle v)</i>	Starts at an arbitrary face incident to <i>v</i> .
<i>Face_circulator</i>	<i>dg.incident_faces(Vertex_handle v, Face_handle f)</i>	Starts at face <i>f</i> . <i>Precondition:</i> Face <i>f</i> must be incident to vertex <i>v</i> .
<i>Edge_circulator</i>	<i>dg.incident_edges(Vertex_handle v)</i>	Starts at an arbitrary edge incident to <i>v</i> .
<i>Edge_circulator</i>	<i>dg.incident_edges(Vertex_handle v, Face_handle f)</i>	Starts at the first edge of <i>f</i> incident to <i>v</i> , in counterclockwise order around <i>v</i> . <i>Precondition:</i> Face <i>f</i> must be incident to vertex <i>v</i> .
<i>Vertex_circulator</i>	<i>dg.incident_vertices(Vertex_handle v)</i>	Starts at an arbitrary vertex incident to <i>v</i> .
<i>Vertex_circulator</i>	<i>dg.incident_vertices(Vertex_handle v, Face_handle f)</i>	Starts at the first vertex of <i>f</i> adjacent to <i>v</i> in counterclockwise order around <i>v</i> . <i>Precondition:</i> Face <i>f</i> must be incident to vertex <i>v</i> .

Predicates

A model of the *DelaunayGraph_2* concept must provide methods to test the finite or infinite character of any feature.

<i>bool</i>	<i>dg.is_infinite(Vertex_handle v)</i>	<i>true</i> , iff <i>v</i> is the <i>infinite_vertex</i> .
<i>bool</i>	<i>dg.is_infinite(Face_handle f)</i>	<i>true</i> , iff face <i>f</i> is infinite.
<i>bool</i>	<i>dg.is_infinite(Face_handle f, int i)</i>	<i>true</i> , iff edge (<i>f</i> , <i>i</i>) is infinite.
<i>bool</i>	<i>dg.is_infinite(Edge e)</i>	<i>true</i> , iff edge <i>e</i> is infinite.
<i>bool</i>	<i>dg.is_infinite(Edge_circulator ec)</i>	<i>true</i> , iff edge <i>*ec</i> is infinite.

Validity check

<i>bool</i>	<i>dg.is_valid(bool verbose = false)</i>	Checks the validity of the Delaunay graph. If <i>verbose</i> is <i>true</i> a short message is sent to <i>std::cerr</i> .
-------------	-------------------------------------------	---------------------------------------------------------------------------------------------------------------------------

Miscellaneous

<i>void</i>	<i>dg.clear()</i>	Clears all contents of the Delaunay graph.
<i>void</i>	<i>dg.swap(other)</i>	The Delaunay graphs <i>other</i> and <i>dg</i> are swapped. <i>dg.swap(other)</i> should be preferred to <i>dg= other</i> or to <i>dg(other)</i> if <i>other</i> is deleted afterwards.

Has Models

CGAL::Delaunay_triangulation_2<Traits,Tds>
CGAL::Regular_triangulation_2<Traits,Tds>
CGAL::Triangulation_hierarchy_2<Tr> provided that *Tr* is a model of *DelaunayGraph_2*
CGAL::Segment_Delaunay_graph_2<Gt,DS>
CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>
CGAL::Apollonius_graph_2<Gt,Agds>
CGAL::Apollonius_graph_hierarchy_2<Gt,Agds>

See Also

AdaptationTraits_2
AdaptationPolicy_2
CGAL::Voronoi_diagram_2<DG,AT,AP>

AdaptationTraits_2

Definition

The concept *AdaptationTraits_2* defines the functors required for accessing geometric information in the Delaunay graph that is needed by the *Voronoi_diagram_2<DG,AT,AP>* class. It optionally defines a functor for performing nearest site queries. A tag is provided for determining whether this functor is defined or not.

Refines

DefaultConstructible, *CopyConstructible*, *Assignable*

Types

<i>AdaptationTraits_2:: Point_2</i>	A type for a point.	
<i>AdaptationTraits_2:: Site_2</i>	A type for the sites of the Voronoi diagram.	
<i>AdaptationTraits_2:: Delaunay_graph</i>	A type for the triangulated Delaunay graph. The type <i>Delaunay_graph</i> must be a model of the <i>DelaunayGraph_2</i> concept.	
<i>typedef Delaunay_graph::Edge</i>	<i>Delaunay_edge;</i>	The type of the edges of the Delaunay graph
<i>typedef Delaunay_graph::Face_handle</i>	<i>Delaunay_face_handle;</i>	The type of the face handles of the Delaunay graph
<i>typedef Delaunay_graph::Vertex_handle</i>	<i>Delaunay_vertex_handle;</i>	The type of the vertex handles of the Delaunay graph.
<i>AdaptationTraits_2:: Access_site_2</i>	<p>A type for a functor that accesses the site associated with a vertex. The functor should be a model of the concepts <i>DefaultConstructible</i>, <i>CopyConstructible</i>, <i>Assignable</i> and <i>AdaptableFunctor</i> (with one argument). The functor must provide the following operator:</p> <p style="text-align: center;"><i>result_type operator()(Delaunay_vertex_handle v)</i></p> <p>where the result type <i>result_type</i> must be either <i>Site_2</i> or <i>const Site_2&</i>.</p>	
<i>AdaptationTraits_2:: Construct_Voronoi_point_2</i>	<p>A type for a functor that constructs the dual point of a (triangular) face in the Delaunay graph. This point is the Voronoi vertex of the three sites defining the face in the Delaunay graph. The functor must be a model of the concepts <i>DefaultConstructible</i>, <i>CopyConstructible</i>, <i>Assignable</i>, <i>AdaptableFunctor</i> (with one argument). It must provide the following operator:</p> <p style="text-align: center;"><i>Point_2 operator()(Delaunay_face_handle f)</i> .</p> <p>The face handle <i>f</i> must not correspond to an infinite face.</p>	
<i>AdaptationTraits_2:: Has_nearest_site_2</i>	<p>A tag for determining if the adaptation traits class provides a functor for performing nearest site queries. This tag is equal to either <i>CGAL::Tag_true</i> (a nearest site query functor is available) or <i>CGAL::Tag_false</i> (a nearest site query functor is not available).</p>	

AdaptationTraits_2::Nearest_site_2

A type for a functor that performs nearest site queries. Semantically, the result of the query is either a face, edge or vertex of the Delaunay graph. It is a face if the query point has at least three closest sites; the returned face has closest sites as vertices. It is an edge if the query point is equidistant to exactly two vertices of the Delaunay graph, which are the source and target vertices of the edge. In all other cases, the search result is a vertex, namely, the unique vertex of the Delaunay graph closest to the query point. The functor must be a model of the concepts *DefaultConstructible*, *CopyConstructible*, *Assignable*, *AdaptableFunctor* (with two arguments). It must provide the following operator:

```
result_type operator()(Delaunay_graph dg, Point_2 p)
where the result type result_type is boost::variant<
Delaunay_vertex_handle,Delaunay_edge,Delaunay_
face_handle>.
```

This type is required only if *Has_nearest_site_2* is equal to *CGAL::Tag_true*.

Access to objects

<i>Access_site_2</i>	<i>at.access_site_2_object()</i>
<i>Construct_Voronoi_point_2</i>	<i>at.construct_Voronoi_point_2_object()</i>
<i>Nearest_site_2</i>	<i>at.nearest_site_2_object()</i>

This method is required only if *Has_nearest_site_2* is equal to *CGAL::Tag_true*.

Has Models

CGAL::Apollonius_graph_adaptation_traits_2<AG2>
CGAL::Delaunay_triangulation_adaptation_traits_2<DT2>
CGAL::Regular_triangulation_adaptation_traits_2<RT2>
CGAL::Segment_Delaunay_graph_adaptation_traits_2<SDG2>

See Also

DelaunayGraph_2
CGAL::Voronoi_diagram_2<DG,AT,AP>

AdaptationPolicy_2

Definition

The concept *AdaptationPolicy_2* defines the requirements on the predicate functors that determine whether a feature of the triangulated Delaunay graph should be rejected or not. It also provides a functor for inserting sites in the Delaunay graph. The last functor is optional and a tag determines whether it is provided or not. Note that while the first two functors do not modify the Delaunay graph they take as an argument, the last ones does.

Refines

DefaultConstructible, *CopyConstructible*, *Assignable*

Types

<i>AdaptationPolicy_2:: Site_2</i>	A type for the sites of the Voronoi diagram.
<i>AdaptationPolicy_2:: Delaunay_graph</i>	A type for the triangulated Delaunay graph. The type <i>Delaunay_graph</i> must be a model of the <i>DelaunayGraph_2</i> concept.
<i>typedef Delaunay_graph::Vertex_handle</i>	<i>Delaunay_vertex_handle;</i>
<i>typedef Delaunay_graph::Face_handle</i>	<i>Delaunay_face_handle;</i>
<i>typedef Delaunay_graph::Edge</i>	<i>Delaunay_edge;</i>
<i>typedef Delaunay_graph::All_edges_iterator</i>	<i>All_Delaunay_edges_iterator;</i>
<i>typedef Delaunay_graph::Finite_edges_iterator</i>	<i>Finite_Delaunay_edges_iterator;</i>
<i>typedef Delaunay_graph::Edge_circulator</i>	<i>Delaunay_edge_circulator;</i>
<i>AdaptationPolicy_2:: Edge_rejector</i>	<p>A type for the predicate functor that is responsible for rejecting an edge of the Delaunay graph (or equivalently rejecting its dual edge in the Voronoi diagram). It must be model of the concepts <i>DefaultConstructible</i>, <i>CopyConstructible</i>, <i>Assignable</i>, and <i>AdaptableFunctor</i> (with two arguments). It must provide the following operators:</p> <pre> bool operator()(Delaunay_graph dg, Delaunay_edge e) bool operator()(Delaunay_graph dg, Delaunay_face_handle f, int i) bool operator()(Delaunay_graph dg, Delaunay_edge_circulator ec) bool operator()(Delaunay_graph dg, All_Delaunay_edges_iterator eit) bool operator()(Delaunay_graph dg, Finite_Delaunay_edges_iterator eit) </pre> <p>The functor returns <i>true</i> iff the edge is rejected.</p>
<i>AdaptationPolicy_2:: Face_rejector</i>	<p>A type for the predicate functor that is responsible for rejecting a vertex of the Delaunay graph (or equivalently its dual face in the Voronoi diagram – hence the name of the functor). It must be model of the concepts <i>DefaultConstructible</i>, <i>CopyConstructible</i>, <i>Assignable</i>, <i>AdaptableFunctor</i> (with two arguments). It must provide the following operator:</p> <pre> bool operator()(Delaunay_graph dg, Delaunay_vertex_handle v) </pre> <p>The functor returns <i>true</i> iff the face is rejected.</p>
<i>AdaptationPolicy_2:: Has_inserter</i>	A tag for determining if the adaptation policy class provides a functor for inserting sites in the Delaunay graph. This tag is equal to either <i>CGAL::Tag_true</i> (a site inserter functor is available) or <i>CGAL::Tag_false</i> (a site inserter functor is not available).

CGAL::Apollonius_graph_adaptation_traits_2<AG2>

Definition

The class *Apollonius_graph_adaptation_traits_2*<AG2> provides a model for the *AdaptationTraits_2* concept. The template parameter of the *Apollonius_graph_adaptation_traits_2*<AG2> class must be a model of the *DelaunayGraph_2* concept, and in particular it has the semantics of a (triangulated) 2D Apollonius graph.

```
#include <CGAL/Apollonius_graph_adaptation_traits_2.h>
```

Is Model for the Concepts

AdaptationTraits_2

Types

```
typedef CGAL::Tag_true    Has_nearest_site_2;
```

See Also

AdaptationTraits_2

DelaunayGraph_2

CGAL::Voronoi_diagram_2<DG,AT,AP>

CGAL::Apollonius_graph_2<Gt,Agds>

CGAL::Apollonius_graph_hierarchy_2<Gt,Agds>

CGAL::Delaunay_triangulation_adaptation_traits_2<DT2>

Definition

The class *Delaunay_triangulation_adaptation_traits_2<DT2>* provides a model for the *AdaptationTraits_2* concept. The template parameter of the *Delaunay_triangulation_adaptation_traits_2<DT2>* class must be a model of the *DelaunayGraph_2* concept, and in particular it has the semantics of a 2D Delaunay triangulation.

```
#include <CGAL/Delaunay_triangulation_adaptation_traits_2.h>
```

Is Model for the Concepts

AdaptationTraits_2

Types

```
typedef CGAL::Tag_true    Has_nearest_site_2;
```

See Also

AdaptationTraits_2

DelaunayGraph_2

CGAL::Voronoi_diagram_2<DG,AT,AP>

CGAL::Delaunay_triangulation_2<Traits,Tds>

CGAL::Regular_triangulation_adaptation_traits_2<RT2>

Definition

The class *Regular_triangulation_adaptation_traits_2<RT2>* provides a model for the *AdaptationTraits_2* concept. The template parameter of the *Regular_triangulation_adaptation_traits_2<RT2>* class must be a model of the *DelaunayGraph_2* concept, and in particular it has the semantics of a 2D regular triangulation.

```
#include <CGAL/Regular_triangulation_adaptation_traits_2.h>
```

Is Model for the Concepts

AdaptationTraits_2

Types

```
typedef CGAL::Tag_true    Has_nearest_site_2;
```

See Also

AdaptationTraits_2

DelaunayGraph_2

Voronoi_diagram_2<DG,AT,AP>

CGAL::Regular_triangulation_2<Traits,Tds>

CGAL::Segment_Delaunay_graph_adaptation_traits_2<SDG2>

Definition

The class *Segment_Delaunay_graph_adaptation_traits_2<SDG2>* provides a model for the *AdaptationTraits_2* concept. The template parameter of the *Segment_Delaunay_graph_adaptation_traits_2<SDG2>* class must be a model of the *DelaunayGraph_2* concept, and in particular it has the semantics of the 2D (triangulated) segment Delaunay graph.

```
#include <CGAL/Segment_Delaunay_graph_adaptation_traits_2.h>
```

Is Model for the Concepts

AdaptationTraits_2

Types

```
typedef CGAL::Tag_true    Has_nearest_site_2;
```

See Also

AdaptationTraits_2

DelaunayGraph_2

CGAL::Voronoi_diagram_2<DG,AT,AP>

CGAL::Segment_Delaunay_graph_2<Gt,DS>

CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>

CGAL::Identity_policy_2<DG,AT>

Definition

The class *Identity_policy_2<DG,AT>* provides a model for the *AdaptationPolicy_2* concept. The first template parameter of the *Identity_policy_2<DG,AT>* class must be a model of the *DelaunayGraph_2* concept, whereas as the second template parameter must be a model of the *AdaptationTraits_2* concept. This policy rejects no edge and no face of the Delaunay graph, thus giving a Voronoi diagram which is the true dual of the triangulation Delaunay graph. The Voronoi diagram created with this adaptation policy may have degenerate features, such as Voronoi edges of zero length, or Voronoi faces of zero area. This policy assumes that the Delaunay graph, that is adapted, allows for site insertions through an *insert* method that takes as argument an object of type *AT::Site_2*. The site inserter functor provided by this policy uses the afore-mentioned *insert* method.

```
#include <CGAL/Identity_policy_2.h>
```

Is Model for the Concepts

AdaptationPolicy_2

Types

```
typedef CGAL::Tag_true    Has_inserter;
```

See Also

AdaptationTraits_2

DelaunayGraph_2

CGAL::Voronoi_diagram_2<DG,AT,AP>

CGAL::Apollonius_graph_degeneracy_removal_policy_2<AG2>

Definition

The class *Apollonius_graph_degeneracy_removal_policy_2<AG2>* provides a model for the *AdaptationPolicy_2* concept. The template parameter of the *Apollonius_graph_degeneracy_removal_policy_2<AG2>* class must be a model of the *DelaunayGraph_2* concept, and in particular it has the semantics of a (triangulated) 2D Apollonius graph. This policy results in a Voronoi diagram that has no degenerate features, i.e., it has no Voronoi edges of zero length and no Voronoi faces of zero area.

```
#include <CGAL/Apollonius_graph_adaptation_policies_2.h>
```

Is Model for the Concepts

AdaptationPolicy_2

Types

```
typedef CGAL::Tag_true    Has_inserter;
```

See Also

AdaptationTraits_2

DelaunayGraph_2

CGAL::Apollonius_graph_caching_degeneracy_removal_policy_2<AG2>

CGAL::Voronoi_diagram_2<DG,AT,AP>

CGAL::Apollonius_graph_2<Gt,Agds>

CGAL::Apollonius_graph_hierarchy_2<Gt,Agds>

CGAL::Delaunay_triangulation_degeneracy_removal_policy_2<DT2>

Definition

The class *Delaunay_triangulation_degeneracy_removal_policy_2<DT2>* provides a model for the *AdaptationPolicy_2* concept. The template parameter of the *Delaunay_triangulation_degeneracy_removal_policy_2<DT2>* class must be a model of the *DelaunayGraph_2* concept, and in particular it has the semantics of a (triangulated) 2D Delaunay triangulation. This policy results in a Voronoi diagram that has no degenerate features, i.e., it has no Voronoi edges of zero length.

```
#include <CGAL/Delaunay_triangulation_adaptation_policies_2.h>
```

Is Model for the Concepts

AdaptationPolicy_2

Types

```
typedef CGAL::Tag_true    Has_inserter;
```

See Also

AdaptationTraits_2

DelaunayGraph_2

CGAL::Delaunay_triangulation_caching_degeneracy_removal_policy_2<DT2>

CGAL::Voronoi_diagram_2<DG,AT,AP>

CGAL::Delaunay_triangulation_2<Traits,Tds>

CGAL::Regular_triangulation_degeneracy_removal_policy_2<RT2>

Definition

The class *Regular_triangulation_degeneracy_removal_policy_2<RT2>* provides a model for the *AdaptationPolicy_2* concept. The template parameter of the *Regular_triangulation_degeneracy_removal_policy_2<RT2>* class must be a model of the *DelaunayGraph_2* concept, and in particular it has the semantics of a (triangulated) 2D regular triangulation. This policy results in a power diagram that has no degenerate features, i.e., it has no Voronoi edges of zero length.

```
#include <CGAL/Regular_triangulation_adaptation_policies_2.h>
```

Is Model for the Concepts

AdaptationPolicy_2

Types

```
typedef CGAL::Tag_true    Has_inserter;
```

See Also

AdaptationTraits_2

DelaunayGraph_2

CGAL::Regular_triangulation_caching_degeneracy_removal_policy_2<RT2>

CGAL::Voronoi_diagram_2<DG,AT,AP>

CGAL::Regular_triangulation_2<Traits,Tds>

CGAL::Segment_Delaunay_graph_degeneracy_removal_policy_2<SDG2>

Definition

The class *Segment_Delaunay_graph_degeneracy_removal_policy_2<SDG2>* provides a model for the *AdaptationPolicy_2* concept. The template parameter of the *Segment_Delaunay_graph_degeneracy_removal_policy_2<SDG2>* class must be a model of the *DelaunayGraph_2* concept, and in particular it has the semantics of a (triangulated) 2D segment Delaunay graphs. This policy results in a Voronoi diagram that has no degenerate features, i.e., it has no Voronoi edges of zero length and no Voronoi faces of zero area.

```
#include <CGAL/Segment_Delaunay_graph_adaptation_policies_2.h>
```

Is Model for the Concepts

AdaptationPolicy_2

Types

```
typedef CGAL::Tag_true    Has_inserter;
```

See Also

AdaptationTraits_2

DelaunayGraph_2

CGAL::Segment_Delaunay_graph_caching_degeneracy_removal_policy_2<SDG2>

CGAL::Voronoi_diagram_2<DG,AT,AP>

CGAL::Segment_Delaunay_graph_2<Gt,DS>

CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>

CGAL::Apollonius_graph_caching_degeneracy_removal_policy_2<AG2>

Definition

The class *Apollonius_graph_caching_degeneracy_removal_policy_2<AG2>* provides a model for the *AdaptationPolicy_2* concept. The template parameter of the *Apollonius_graph_caching_degeneracy_removal_policy_2<AG2>* class must be a model of the *DelaunayGraph_2* concept, and in particular it has the semantics of a (triangulated) 2D Apollonius graph. This policy caches the results of the edge and face rejectors and results in a Voronoi diagram that has no degenerate features, i.e., no Voronoi edges of zero length and no Voronoi faces of zero area.

```
#include <CGAL/Apollonius_graph_adaptation_policies_2.h>
```

Is Model for the Concepts

AdaptationPolicy_2

Types

```
typedef CGAL::Tag_true    Has_inserter;
```

See Also

AdaptationTraits_2
DelaunayGraph_2
CGAL::Apollonius_graph_degeneracy_removal_policy_2<AG2>
CGAL::Voronoi_diagram_2<DG,AT,AP>
CGAL::Apollonius_graph_2<Gt,Agds>
CGAL::Apollonius_graph_hierarchy_2<Gt,Agds>

CGAL::Delaunay_triangulation_caching_degeneracy_removal_policy_2<DT2>

Definition

The class *Delaunay_triangulation_caching_degeneracy_removal_policy_2<DT2>* provides a model for the *AdaptationPolicy_2* concept. The template parameter of the *Delaunay_triangulation_caching_degeneracy_removal_policy_2<DT2>* class must be a model of the *DelaunayGraph_2* concept, and in particular it has the semantics of a (triangulated) 2D Delaunay triangulation. This policy caches the results of the edge and face rejectors and results in a Voronoi diagram that has no degenerate features, i.e., no Voronoi edges of zero length.

```
#include <CGAL/Delaunay_triangulation_adaptation_policies_2.h>
```

Is Model for the Concepts

AdaptationPolicy_2

Types

```
typedef CGAL::Tag_true    Has_inserter;
```

See Also

AdaptationTraits_2

DelaunayGraph_2

CGAL::Delaunay_triangulation_degeneracy_removal_policy_2<DT2>

CGAL::Voronoi_diagram_2<DG,AT,AP>

CGAL::Delaunay_triangulation_2<Traits,Tds>

CGAL::Regular_triangulation_caching_degeneracy_removal_policy_2<RT2>

Definition

The class *Regular_triangulation_caching_degeneracy_removal_policy_2<RT2>* provides a model for the *AdaptationPolicy_2* concept. The template parameter of the *Regular_triangulation_caching_degeneracy_removal_policy_2<RT2>* class must be a model of the *DelaunayGraph_2* concept, and in particular it has the semantics of a (triangulated) 2D regular triangulation. This policy caches the results of the edge and face rejectors and results in a Voronoi diagram that has no degenerate features, i.e., no Voronoi edges of zero length.

```
#include <CGAL/Regular_triangulation_adaptation_policies_2.h>
```

Is Model for the Concepts

AdaptationPolicy_2

Types

```
typedef CGAL::Tag_true    Has_inserter;
```

See Also

AdaptationTraits_2

DelaunayGraph_2

CGAL::Regular_triangulation_degeneracy_removal_policy_2<RT2>

CGAL::Voronoi_diagram_2<DG,AT,AP>

CGAL::Regular_triangulation_2<Traits,Tds>

CGAL::Segment_Delaunay_graph_caching_degeneracy_removal_policy_2<SDG2>

Definition

The class *Segment_Delaunay_graph_caching_degeneracy_removal_policy_2<SDG2>* provides a model for the *AdaptationPolicy_2* concept. The template parameter of the *Segment_Delaunay_graph_caching_degeneracy_removal_policy_2<SDG2>* class must be a model of the *DelaunayGraph_2* concept, and in particular it has the semantics of a (triangulated) 2D segment Delaunay graph. This policy caches the results of the edge and face rejectors and results in a Voronoi diagram that has no degenerate features, i.e., no Voronoi edges of zero length and no Voronoi faces of zero area.

```
#include <CGAL/Segment_Delaunay_graph_adaptation_policies_2.h>
```

Is Model for the Concepts

AdaptationPolicy_2

Types

```
typedef CGAL::Tag_false    Has_inserter;
```

See Also

AdaptationTraits_2

DelaunayGraph_2

CGAL::Segment_Delaunay_graph_degeneracy_removal_policy_2<SDG2>

CGAL::Voronoi_diagram_2<DG,AT,AP>

CGAL::Segment_Delaunay_graph_2<Gt,DS>

CGAL::Segment_Delaunay_graph_hierarchy_2<Gt,STag,DS>

Part IX

Meshing

Chapter 29

2D Conforming Triangulations and Meshes

Laurent Rineau

Contents

29.1 Conforming Triangulations	1787
29.1.1 Definitions	1787
29.1.2 Building Conforming Triangulations	1788
29.1.3 Example: Making a Triangulation Conforming Delaunay and Then Conforming Gabriel	1788
29.2 Meshes	1791
29.2.1 Definitions	1791
29.2.2 Shape and Size Criteria	1791
29.2.3 The Meshing Algorithm	1793
29.2.4 Building Meshes	1794
29.2.5 Example Using the Global Function	1794
29.2.6 Example Using the Class <i>Delaunay_mesher_2<CDT></i>	1795
29.2.7 Example Using Seeds	1796

This package implements Shewchuk’s algorithm [She00] to construct conforming triangulations and 2D meshes. Conforming triangulations will be described in Section 29.1 and meshes in Section 29.2.

29.1 Conforming Triangulations

29.1.1 Definitions

A triangulation is a *Delaunay triangulation* if the circumscribing circle of any facet of the triangulation contains no vertex in its interior. A *constrained Delaunay triangulation* is a constrained triangulation which is as much Delaunay as possible. The circumscribing circle of any facet of a constrained Delaunay triangulation contains in its interior no data point *visible* from the facet.

An edge is said to be a *Delaunay edge* if it is inscribed in an empty circle (containing no data point in its interior). This edge is said to be a *Gabriel edge* if its diametrical circle is empty.

A constrained Delaunay triangulation is said to be a *conforming Delaunay triangulation* if every constrained edge is a Delaunay edge. Because any edge in a constrained Delaunay triangulation is either a Delaunay edge or a constrained edge, a conforming Delaunay triangulation is in fact a Delaunay triangulation. The only difference is that some of the edges are marked as constrained edges.

A constrained Delaunay triangulation is said to be a *conforming Gabriel triangulation* if every constrained edge is a Gabriel edge. The Gabriel property is stronger than the Delaunay property and each Gabriel edge is a Delaunay edge. Conforming Gabriel triangulations are thus also conforming Delaunay triangulations.

Any constrained Delaunay triangulation can be refined into a conforming Delaunay triangulation or into a conforming Gabriel triangulation by adding vertices, called *Steiner vertices*, on constrained edges until they are decomposed into subconstraints small enough to be Delaunay or Gabriel edges.

29.1.2 Building Conforming Triangulations

Constrained Delaunay triangulations can be refined into conforming triangulations by the two following global functions:

```
template<class CDT> void make_conforming_Delaunay_2 (CDT& t) and
template<class CDT> void make_conforming_Gabriel_2 (CDT& t).
```

In both cases, the template parameter *CDT* must be instantiated by a constrained Delaunay triangulation class. Such a class must be a model of the concept *ConstrainedDelaunayTriangulation_2*.

The geometric traits of the constrained Delaunay triangulation used to instantiate the parameter *CDT* has to be a model of the concept *ConformingDelaunayTriangulationTraits_2*.

The constrained Delaunay triangulation *t* is passed by reference and is refined into a conforming Delaunay triangulation or into a conforming Gabriel triangulation by adding vertices. The user is advised to make a copy of the input triangulation in the case where the original triangulation has to be preserved for other computations

The algorithm used by *make_conforming_Delaunay_2* and *make_conforming_Gabriel_2* builds internal data structures that would be computed twice if the two functions are called consecutively on the same triangulation. In order to avoid these data to be constructed twice, the advanced user can use the class *Triangulation_conformer_2<CDT>* to refine a constrained Delaunay triangulation into a conforming Delaunay triangulation and then into a conforming Gabriel triangulation. For additional control of the refinement algorithm, this class also provides separate functions to insert one Steiner point at a time.

29.1.3 Example: Making a Triangulation Conforming Delaunay and Then Conforming Gabriel

This example inserts several segments into a constrained Delaunay triangulation, makes it conforming Delaunay, and then conforming Gabriel. At each step, the number of vertices of the triangulation is printed.

```
// file: examples/Mesh_2/conforming.C

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Constrained_Delaunay_triangulation_2.h>
#include <CGAL/Triangulation_conformer_2.h>

#include <iostream>
```

```

struct K : public CGAL::Exact_predicates_inexact_constructions_kernel {};
typedef CGAL::Constrained_Delaunay_triangulation_2<K> CDT;
typedef CDT::Point Point;
typedef CDT::Vertex_handle Vertex_handle;

int main()
{
    CDT cdt;

    // construct a constrained triangulation
    Vertex_handle
        va = cdt.insert(Point( 5., 5.)),
        vb = cdt.insert(Point(-5., 5.)),
        vc = cdt.insert(Point( 4., 3.)),
        vd = cdt.insert(Point( 5.,-5.)),
        ve = cdt.insert(Point( 6., 6.)),
        vf = cdt.insert(Point(-6., 6.)),
        vg = cdt.insert(Point(-6.,-6.)),
        vh = cdt.insert(Point( 6.,-6.));

    cdt.insert_constraint(va,vb);
    cdt.insert_constraint(vb,vc);
    cdt.insert_constraint(vc,vd);
    cdt.insert_constraint(vd,va);
    cdt.insert_constraint(ve,vf);
    cdt.insert_constraint(vf,vg);
    cdt.insert_constraint(vg,vh);
    cdt.insert_constraint(vh,ve);

    std::cout << "Number of vertices before: "
                << cdt.number_of_vertices() << std::endl;

    // make it conforming Delaunay
    CGAL::make_conforming_Delaunay_2(cdt);

    std::cout << "Number of vertices after make_conforming_Delaunay_2: "
                << cdt.number_of_vertices() << std::endl;

    // then make it conforming Gabriel
    CGAL::make_conforming_Gabriel_2(cdt);

    std::cout << "Number of vertices after make_conforming_Gabriel_2: "
                << cdt.number_of_vertices() << std::endl;
}

```

See figures [29.1](#), [29.2](#) and [29.3](#).

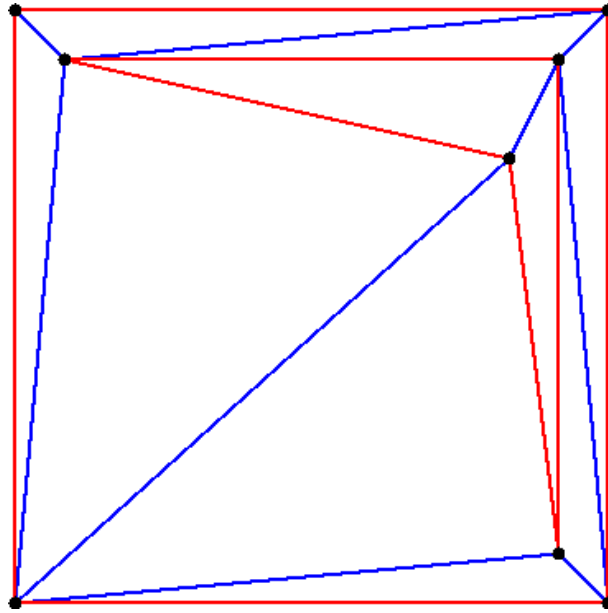


Figure 29.1: Initial triangulation.

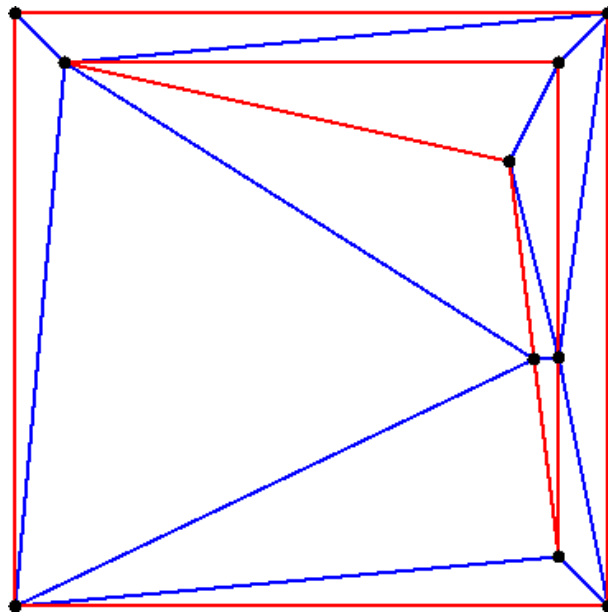


Figure 29.2: The corresponding conforming Delaunay triangulation.

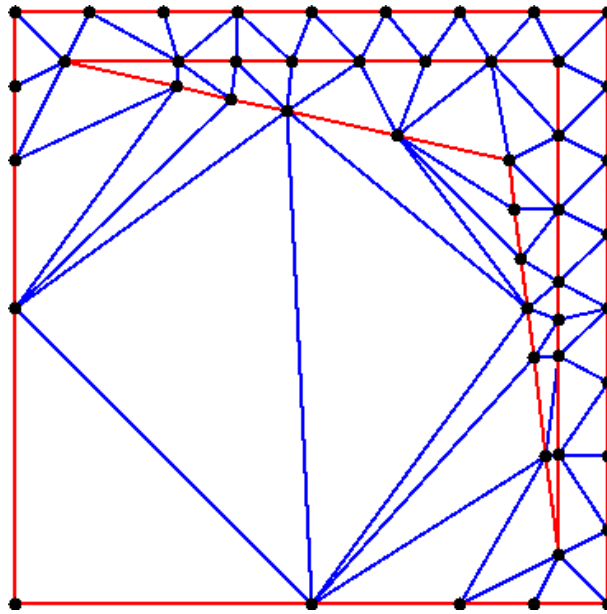


Figure 29.3: The corresponding conforming Gabriel triangulation.

29.2 Meshes

29.2.1 Definitions

A mesh is a partition of a given region into simplices whose shapes and sizes satisfy several criteria.

The domain is the region that the user wants to mesh. It has to be a bounded region of the plane. The domain is defined by a *planar straight line graph*, PSLG for short, which is a set of segments such that two segments in the set are either disjoint or share an endpoint. The segments of the PSLG are constraints that will be represented by a union of edges in the mesh. The PSLG can also contain isolated points that will appear as vertices of the mesh.

The segments of the PSLG are either segments of the boundary or internal constraints. The segments of the PSLG have to cover the boundary of the domain.

The PSLG divides the plane into several connected components. By default, the domain is the union of the bounded connected components. The user can override this default by providing a set of seed points. Either seed points mark components to be meshed or they mark components not to be meshed (holes).

See figures 29.4 and 29.5 for an example of a domain defined without using seed points, and a possible mesh of it. See figure 29.6 for another domain defined with the same PSLG and two seed points used to define holes. In the corresponding mesh (figure 29.7), these two holes are triangulated but not meshed.

29.2.2 Shape and Size Criteria

The shape criterion for triangles is a lower bound B on the ratio between the circumradius and the shortest edge length. Such a bound implies a lower bound of $\arcsin \frac{1}{2B}$ on the minimum angle of the triangle and an upper

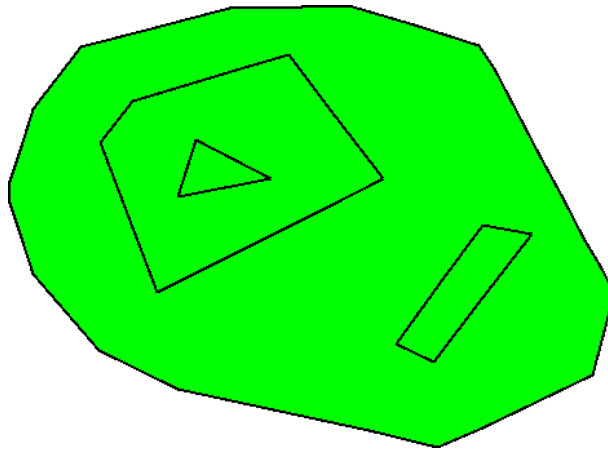


Figure 29.4: A domain defined without seed points.

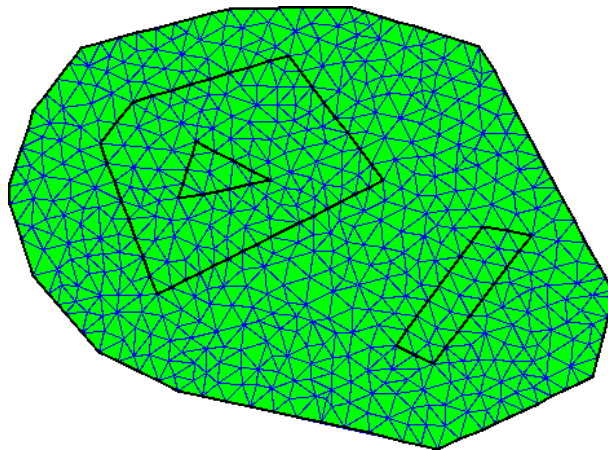


Figure 29.5: A mesh of the domain defined without seed points.

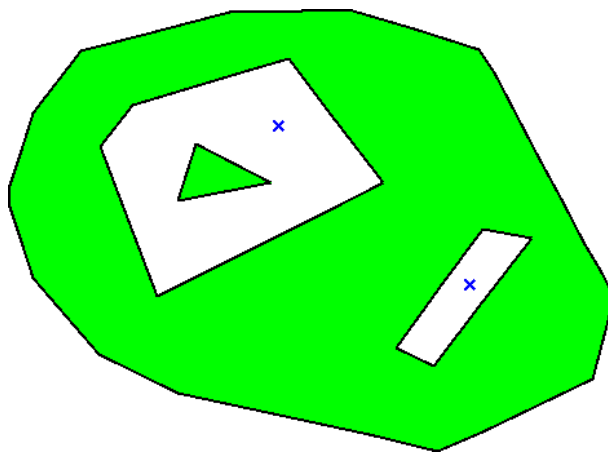


Figure 29.6: A domain with two seeds points defining holes.

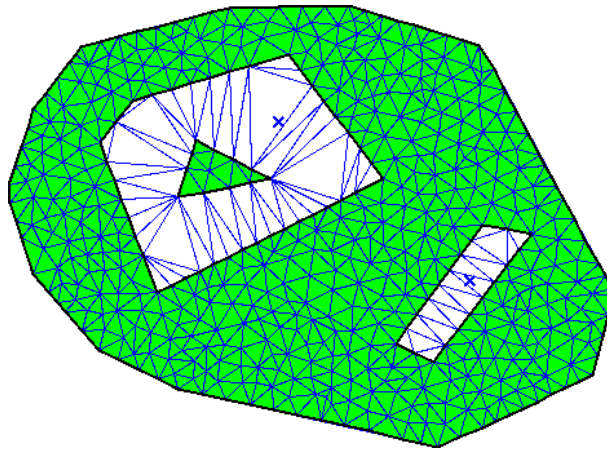


Figure 29.7: A mesh of the domain with two seeds defining holes.

bound of $\pi - 2 * \arcsin \frac{1}{2B}$ on the maximum angle. Unfortunately, the termination of the algorithm is guaranteed only if $B \geq \sqrt{2}$, which corresponds to a lower bound of 20.7 degrees over the angles.

The size criterion can be any criterion that tends to prefer small triangles. For example, the size criterion can be an upper bound on the length of longest edge of triangles, or an upper bound on the radius of the circumcircle. The size bound can vary over the domain. For example, the size criterion could impose a small size for the triangles intersecting a given line.

Both types of criteria are defined in an object *criteria* passed as parameter of the meshing functions.

29.2.3 The Meshing Algorithm

The input to a meshing problem is a PSLG and a set of seeds describing the domain to be meshed, and a set of size and shape criteria. The algorithm implemented in this package starts with a constrained Delaunay triangulation of the input PSLG and produces a mesh using the Delaunay refinement method. This method inserts new vertices to the triangulation, as far as possible from other vertices, and stops when the criteria are satisfied.

If all angles between incident segments of the input PSLG are greater than 60 degrees and if the bound on the circumradius/edge ratio is greater than $\sqrt{2}$, the algorithm is guaranteed to terminate with a mesh satisfying the size and shape criteria.

If some input angles are smaller than 60 degrees, the algorithm will end up with a mesh in which some triangles violate the criteria near small input angles. This is unavoidable since small angles formed by input segments cannot be suppressed. Furthermore, it has been shown ([She00]), that some domains with small input angles cannot be meshed with angles even smaller than the small input angles. Note that if the domain is a polygonal region, the resulting mesh will satisfy size and shape criteria except for the small input angles. In addition, the algorithm may succeed in producing meshes with a lower angle bound greater than 20.7 degrees, but there is no such guarantee.

29.2.4 Building Meshes

Meshes are obtained from constrained Delaunay triangulations by calling the global function `template<class CDT, class Criteria> void refine_Delaunay_mesh_2 (CDT &t, typename CDT::Geom_traits gt)`. The template parameter `CDT` must be instantiated by a constrained Delaunay triangulation class, which is a model of the concept `ConstrainedDelaunayTriangulation_2`. In order to override the domain, a version of this function has two more arguments that define a sequence of seed points.

The geometric traits class of `CDT` has to be a model of the concept `DelaunayMeshTraits_2`. This concept refines the concept `ConformingDelaunayTriangulationTraits_2` adding the geometric predicates and constructors. The template parameter `Criteria` must be a model of `MeshingCriteria_2`. This concept defines criteria that the triangles have to satisfy. CGAL provides two models for this concept:

- `Delaunay_mesh_criteria_2<CDT>`, that defines a shape criterion that bounds the minimum angle of triangles,
- `Delaunay_mesh_size_criteria_2<CDT>`, that adds to the previous criterion a bound on the maximum edge length.

If the function `refine_Delaunay_mesh_2` is called several times on the same triangulation with different criteria, the algorithm rebuilds the internal data structure used for meshing at every call. In order to avoid rebuild the data structure at every call, the advanced user can use the class `Delaunay_mesher_2<CDT>`. This class provides also step by step functions. Those functions insert one vertex at a time.

Any object of type `Delaunay_mesher_2<CDT>` is constructed from a reference to a `CDT`, and has several member functions to define the domain to be meshed and to mesh the `CDT`. See the example given below and the reference manual for details. Note that the `CDT` should not be externally modified during the life time of the `Delaunay_mesher_2<CDT>` object.

29.2.5 Example Using the Global Function

The following example inserts several segments into a constrained triangulation and then meshes it using the global function `refine_Delaunay_mesh_2`. The size and shape criteria are the default ones provided by the criteria class `Delaunay_mesh_criteria_2<K>`. No seeds are given, meaning that the mesh domain covers the whole plane except the unbounded component.

```
// file: examples/Mesh_2/mesh_global.C

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Constrained_Delaunay_triangulation_2.h>
#include <CGAL/Delaunay_mesher_2.h>
#include <CGAL/Delaunay_mesh_face_base_2.h>
#include <CGAL/Delaunay_mesh_size_criteria_2.h>

#include <iostream>

struct K : public CGAL::Exact_predicates_inexact_constructions_kernel {};
typedef CGAL::Triangulation_vertex_base_2<K> Vb;
typedef CGAL::Delaunay_mesh_face_base_2<K> Fb;
typedef CGAL::Triangulation_data_structure_2<Vb, Fb> Tds;
typedef CGAL::Constrained_Delaunay_triangulation_2<K, Tds> CDT;
```

```

typedef CGAL::Delaunay_mesh_size_criteria_2<CDT> Criteria;

typedef CDT::Vertex_handle Vertex_handle;
typedef CDT::Point Point;

int main()
{
    CDT cdt;

    Vertex_handle va = cdt.insert(Point(-4,0));
    Vertex_handle vb = cdt.insert(Point(0,-1));
    Vertex_handle vc = cdt.insert(Point(4,0));
    Vertex_handle vd = cdt.insert(Point(0,1));
    cdt.insert(Point(2, 0.6));

    cdt.insert_constraint(va, vb);
    cdt.insert_constraint(vb, vc);
    cdt.insert_constraint(vc, vd);
    cdt.insert_constraint(vd, va);

    std::cout << "Number of vertices: " << cdt.number_of_vertices() << std::endl;

    std::cout << "Meshing the triangulation..." << std::endl;
    CGAL::refine_Delaunay_mesh_2(cdt, Criteria(0.125, 0.5));

    std::cout << "Number of vertices: " << cdt.number_of_vertices() << std::endl;
}

```

29.2.6 Example Using the Class *Delaunay_mesher_2*<CDT>

This example uses the class *Delaunay_mesher_2*<CDT> and calls the *refine_mesh()* member function twice, changing the size and shape criteria in between. In such a case, using twice the global function *refine_Delaunay_mesh_2* would be less efficient, because some internal structures needed by the algorithm would be built twice.

```

// file: examples/Mesh_2/mesh_class.C

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Constrained_Delaunay_triangulation_2.h>
#include <CGAL/Delaunay_mesher_2.h>
#include <CGAL/Delaunay_mesh_face_base_2.h>
#include <CGAL/Delaunay_mesh_size_criteria_2.h>

#include <iostream>

struct K : public CGAL::Exact_predicates_inexact_constructions_kernel {};
typedef CGAL::Triangulation_vertex_base_2<K> Vb;
typedef CGAL::Delaunay_mesh_face_base_2<K> Fb;
typedef CGAL::Triangulation_data_structure_2<Vb, Fb> Tds;
typedef CGAL::Constrained_Delaunay_triangulation_2<K, Tds> CDT;
typedef CGAL::Delaunay_mesh_size_criteria_2<CDT> Criteria;
typedef CGAL::Delaunay_mesher_2<CDT, Criteria> Mesher;

```

```

typedef CDT::Vertex_handle Vertex_handle;
typedef CDT::Point Point;

int main()
{
    CDT cdt;

    Vertex_handle va = cdt.insert(Point(-4,0));
    Vertex_handle vb = cdt.insert(Point(0,-1));
    Vertex_handle vc = cdt.insert(Point(4,0));
    Vertex_handle vd = cdt.insert(Point(0,1));
    cdt.insert(Point(2, 0.6));

    cdt.insert_constraint(va, vb);
    cdt.insert_constraint(vb, vc);
    cdt.insert_constraint(vc, vd);
    cdt.insert_constraint(vd, va);

    std::cout << "Number of vertices: " << cdt.number_of_vertices() << std::endl;

    std::cout << "Meshing the triangulation with default criterias..."
               << std::endl;

    Mesher mesher(cdt);
    mesher.refine_mesh();

    std::cout << "Number of vertices: " << cdt.number_of_vertices() << std::endl;

    std::cout << "Meshing with new criterias..." << std::endl;
    // 0.125 is the default shape bound. It corresponds to about 20.6 degree.
    // 0.5 is the upper bound on the length of the longest edge.
    // See reference manual for Delaunay_mesh_size_traits_2<K>.
    mesher.set_criteria(Criteria(0.125, 0.5));
    mesher.refine_mesh();

    std::cout << "Number of vertices: " << cdt.number_of_vertices() << std::endl;
}

```

29.2.7 Example Using Seeds

This example uses the global function *refine_Delaunay_mesh_2* but defines a domain by using one seed. The size and shape criteria are the default ones provided by the criteria class *Delaunay_mesh_criteria_2<K>*.

```

// file: examples/Mesh_2/mesh_with_seeds.C

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Constrained_Delaunay_triangulation_2.h>
#include <CGAL/Delaunay_mesher_2.h>
#include <CGAL/Delaunay_mesh_face_base_2.h>
#include <CGAL/Delaunay_mesh_size_criteria_2.h>

#include <iostream>

```

```

struct K : public CGAL::Exact_predicates_inexact_constructions_kernel {};
typedef CGAL::Triangulation_vertex_base_2<K> Vb;
typedef CGAL::Delaunay_mesh_face_base_2<K> Fb;
typedef CGAL::Triangulation_data_structure_2<Vb, Fb> Tds;
typedef CGAL::Constrained_Delaunay_triangulation_2<K, Tds> CDT;
typedef CGAL::Delaunay_mesh_size_criteria_2<CDT> Criteria;

typedef CDT::Vertex_handle Vertex_handle;
typedef CDT::Point Point;

int main()
{
    CDT cdt;
    Vertex_handle va = cdt.insert(Point(2,0));
    Vertex_handle vb = cdt.insert(Point(0,2));
    Vertex_handle vc = cdt.insert(Point(-2,0));
    Vertex_handle vd = cdt.insert(Point(0,-2));

    cdt.insert_constraint(va, vb);
    cdt.insert_constraint(vb, vc);
    cdt.insert_constraint(vc, vd);
    cdt.insert_constraint(vd, va);

    va = cdt.insert(Point(3,3));
    vb = cdt.insert(Point(-3,3));
    vc = cdt.insert(Point(-3,-3));
    vd = cdt.insert(Point(3,0-3));

    cdt.insert_constraint(va, vb);
    cdt.insert_constraint(vb, vc);
    cdt.insert_constraint(vc, vd);
    cdt.insert_constraint(vd, va);

    std::list<Point> list_of_seeds;

    list_of_seeds.push_back(Point(0, 0));

    std::cout << "Number of vertices: " << cdt.number_of_vertices() << std::endl;

    std::cout << "Meshing the domain..." << std::endl;
    CGAL::refine_Delaunay_mesh_2(cdt, list_of_seeds.begin(), list_of_seeds.end(),
                                Criteria());

    std::cout << "Number of vertices: " << cdt.number_of_vertices() << std::endl;
}

```


2D Conforming Triangulations and Meshes

Reference Manual

Laurent Rineau

29.3 Classified Reference Pages

Concepts

ConformingDelaunayTriangulationTraits_2	page 1801
DelaunayMeshTraits_2	page 1804
MeshingCriteria_2	page 1814
DelaunayMeshFaceBase_2	page 1803

Classes

CGAL::Triangulation_conformer_2<CDT>	page 1817
CGAL::Delaunay_mesher_2<CDT, Criteria>	page 1805
CGAL::Delaunay_mesh_face_base_2<Traits, Fb>	page 1809
CGAL::Delaunay_mesh_criteria_2<CDT>	page 1808
CGAL::Delaunay_mesh_size_criteria_2<CDT>	page 1810
CGAL::Mesh_2::Face_badness	page 1811

Global functions

CGAL::make_conforming_Delaunay_2	page 1812
CGAL::make_conforming_Gabriel_2	page 1813
CGAL::refine_Delaunay_mesh_2	page 1816

29.4 Alphabetical List of Reference Pages

ConformingDelaunayTriangulationTraits_2	page 1801
DelaunayMeshFaceBase_2	page 1803

<i>DelaunayMeshTraits_2</i>	page 1804
<i>Delaunay_mesher_2</i> <CDT, Criteria>	page 1805
<i>Delaunay_mesh_criteria_2</i> <CDT>	page 1808
<i>Delaunay_mesh_face_base_2</i> <Traits, Fb>	page 1809
<i>Delaunay_mesh_size_criteria_2</i> <CDT>	page 1810
<i>make_conforming_Delaunay_2</i>	page 1812
<i>make_conforming_Gabriel_2</i>	page 1813
<i>MeshingCriteria_2</i>	page 1814
<i>Mesh_2::Face_badness</i>	page 1811
<i>refine_Delaunay_mesh_2</i>	page 1816
<i>Triangulation_conformer_2</i> <CDT>	page 1817

ConformingDelaunayTriangulationTraits_2

Definition

The concept `ConformingDelaunayTriangulationTraits_2` refines the concept `ConstrainedDelaunayTriangulationTraits_2` by providing a numeric field type *FT*, a type *Vector_2* and several constructors on *Vector_2*, *Point_2*, and a predicate on angles. The field type has to be a model of the concept *SqrtFieldType*. This field type and the constructors are used by the conforming algorithm to compute Steiner points on constrained edges.

Refines

DelaunayTriangulationTraits_2

Types

ConformingDelaunayTriangulationTraits_2:: FT

The field type. It must be a model of *SqrtFieldType*, that is must be a number type supporting the operations $+$, $-$, $*$, $/$, and $\sqrt{\cdot}$.

ConformingDelaunayTriangulationTraits_2:: Vector_2

The vector type.

ConformingDelaunayTriangulationTraits_2:: Construct_vector_2

Constructor object. Must provide the operator *Vector_2 operator()(Point a, Point b)* that computes the vector $b - a$.

ConformingDelaunayTriangulationTraits_2:: Construct_scaled_vector_2

Constructor object. Must provide the operator *Vector_2 operator()(Vector_2 v, FT scale)* that computes the vector $scale \cdot \mathbf{v}$.

ConformingDelaunayTriangulationTraits_2:: Construct_translated_point_2

Constructor object. Must provide the operator *Point_2 operator()(Point_2 p, Vector_2 v)* that computes the point $p + \mathbf{v}$.

ConformingDelaunayTriangulationTraits_2:: Construct_midpoint_2

Constructor object. Must provide the operator *Point_2 operator()(Point_2 a, Point_2 b)* that computes the midpoint of the segment ab .

ConformingDelaunayTriangulationTraits_2:: Compute_squared_distance_2

Constructor object. Must provide the operator *FT operator()(Point_2 a, Point_2 b)* that computes the squared distance between *a* and *b*.

ConformingDelaunayTriangulationTraits_2:: Angle_2

Predicate object. Must provide the operator *CGAL::Angle operator()(Point_2 p, Point_2 q, Point_2 r)* that returns OB-TUSE, RIGHT or ACUTE depending on the angle formed by the three points *p*, *q*, *r* (*q* being the vertex of the angle).

Access to predicate and constructor objects

<i>Construct_vector_2</i>	<i>traits.construct_vector_2_object()</i>
<i>Construct_scaled_vector_2</i>	<i>traits.construct_scaled_vector_2_object()</i>
<i>Construct_translated_point_2</i>	<i>traits.construct_translated_point_2_object()</i>
<i>Constructor_midpoint_2</i>	<i>traits.construct_midpoint_2_object()</i>
<i>Compute_squared_distance_2</i>	<i>traits.compute_squared_distance_2_object()</i>
<i>Angle_2</i>	<i>traits.angle_2_object()</i>

Has Models

Any model of *Kernel* concept. In particular, all CGAL kernels.

DelaunayMeshFaceBase_2

Definition

The concept `DelaunayMeshFaceBase.2` refines the concept `TriangulationFaceBase.2`. It adds two functions giving access to a boolean marker, that indicates if the face is in the meshing domain or not.

Refines

ConstrainedTriangulationFaceBase_2

Access Functions

<i>bool</i>	<i>f.is_in_domain()</i>	returns true if this face is in the domain to be refined.
-------------	-------------------------	-----------------------------------------------------------

```
void f.set_in_domain( const bool b)
```

sets if this face is in the domain.

Has Models

$$\text{Delaunay_mesh_face_base_2}\langle \text{Traits}, \text{Fb} \rangle$$

DelaunayMeshTraits_2

Definition

The concept `DelaunayMeshTraits_2` refines the concept `ConformingDelaunayTriangulationTraits_2`. It provides a construction object `Construct_circumcenter_2`.

Refines

`ConformingDelaunayTriangulationTraits_2`

Types

`DelaunayMeshTraits_2::Construct_circumcenter_2`

Constructor object. Must provide an operator `Point_2 operator()(Point_2 p, Point_2 q, Point_2 r)`; that computes the center of the circle passing through the points p , q and r .
Precondition: p , q and r are not collinear.

Access to predicate and constructor objects

`Construct_circumcenter_2` `traits.construct_circumcenter_2_object()`

Has Models

Any model of the *Kernel* concept. In particular, all CGAL kernels.

CGAL::Delaunay_mesher_2<CDT, Criteria>

This class implements a 2D mesh generator.

Parameters

The template parameter *CDT* should be a model of the concept *ConstrainedDelaunayTriangulation_2*, and type *CDT::Face* should be a model of the concept *MeshFaceBase_2*.

The geometric traits class of the instance of *CDT* has to be a model of the concept *DelaunayMeshTraits_2*.

The template parameter *Criteria* should be a model of the concept *MeshingCriteria_2*. This traits class defines the shape and size criteria for the triangles of the mesh. *Criteria::Face_handle* has to be the same as *CDT::Face_handle*.

Using this class

The constructor of the class *Delaunay_mesher_2<CDT, Criteria>* takes a reference to a *CDT* as an argument. A call to the refinement method *refine_mesh()* will refine the constrained Delaunay triangulation into a mesh satisfying the size and shape criteria specified in the traits class. Note that if, during the life time of the *Delaunay_mesher_2<CDT, Criteria>* object, the triangulation is externally modified, any further call to its member methods may crash. Consider constructing a new *Delaunay_mesher_2<CDT, Criteria>* object if the triangulation has been modified.

Meshing domain

The domain to be mesh is defined by the constrained edges and a set of seed points. The constrained edges divides the plane into several connected components. The mesh domain is either the union of the bounded connected components including at least one seed, or the union of the bounded connected components that do not contain any seed. Note that the unbounded component of the plane is never meshed.

```
#include <CGAL/Delaunay_mesher_2.h>
```

Types

```
typedef CDT::Geom_traits
```

Geom_traits; the geometric traits class.

```
Delaunay_mesher_2<CDT, Criteria>:: Seeds_iterator
```

const iterator over defined seeds. Its value type is *Geom_traits::Point_2*.

Creation

```
Delaunay_mesher_2<CDT, Criteria> mesher( CDT& t, Criteria criteria = Criteria());
```

Create a new mesher, working on *t*, with meshing criteria *criteria*.

Seeds functions

The following functions are used to define seeds.

<i>void</i>	<i>mesher.clear_seeds()</i>	Sets seeds to the empty set. All finite connected components of the constrained triangulation will be refined.
-------------	-----------------------------	----------------------------------------------------------------------------------------------------------------

<i>template<class InputIterator></i>		
<i>void</i>	<i>mesher.set_seeds(InputIterator begin, InputIterator end, const bool mark=false)</i>	
		Sets seeds to the sequence <i>[begin, end]</i> . If <i>mark=true</i> , the mesh domain is the union of the bounded connected components including at least one seed. If <i>mark=false</i> , the domain is the union of the bounded components including no seed. Note that the unbounded component of the plane is never meshed.
		<i>Requirement:</i> The <i>value_type</i> of <i>begin</i> and <i>end</i> is <i>Geom_traits::Point_2</i> .

<i>Seeds_const_iterator</i>	<i>mesher.seeds_begin()</i>	Start of the seeds sequence.
<i>Seeds_const_iterator</i>	<i>mesher.seeds_end()</i>	Past the end of the seeds sequence.

Meshing methods

<i>void</i>	<i>mesher.refine_mesh()</i>	Refines the constrained Delaunay triangulation into a mesh satisfying the criteria defined by the traits.
-------------	-----------------------------	-----------------------------------------------------------------------------------------------------------

<i>Criteria</i>	<i>mesher.get_criteria()</i>	Returns a const reference to the criteria traits object.
-----------------	------------------------------	----------------------------------------------------------

<i>void</i>	<i>mesher.set_criteria(Criteria criteria)</i>	
		Assigns <i>criteria</i> to the criteria traits object.

— *advanced* —

The function *set_criteria* scans all faces to recalculate the list of *bad faces*, that are faces not conforming to the meshing criteria. This function actually has an optional argument that permits to prevent this recalculation. The filling of the list of bad faces can then be done by a call to *set_bad_faces*.

<i>void</i>	<i>mesher.set_criteria(Criteria criteria, bool recalculate_bad_faces)</i>	
		Assigns <i>criteria</i> to the criteria traits object. If <i>recalculate_bad_faces</i> is <i>false</i> , the list of bad faces is let empty and the function <i>set_bad_faces</i> should be called before <i>refine_mesh</i> .

template <class InputIterator>

void *mesher.set_bad_faces(InputIterator begin, InputIterator end)*

This method permits to set the list of bad triangles directly, from the sequence [begin, end], so that the algorithm will not scan the whole set of triangles to find bad ones. To use if there is a non-naive way to find bad triangles.

Requirement: The *value_type* of *begin* and *end* is *Face_handle*.

_____ *advanced* _____

_____ *advanced* _____

Step by step operations

The *Delaunay_mesher_2*<*CDT*, *Criteria*> class allows, for debugging or demos, to play the meshing algorithm step by step, using the following methods.

void *mesher.init()*

This method must be called just before the first call to the following step by step refinement method, that is when all vertices and constrained edges have been inserted into the constrained Delaunay triangulation. It must be called again before any subsequent calls to the step by step refinement method if new vertices or constrained edges have been inserted since the last call.

bool *mesher.is_refinement_done()*

Tests if the step by step refinement algorithm is done. If it returns *true*, the following calls to *step_by_step_refine_mesh* will not insert any points, until some new constrained segments or points are inserted in the triangulation and *init* is called again.

bool *mesher.step_by_step_refine_mesh()*

Applies one step of the algorithm, by inserting one point, if the algorithm is not done. Returns *false* iff no point has been inserted because the algorithm is done.

_____ *advanced* _____

CGAL::Delaunay_mesh_criteria_2<CDT>

Definition

The class *Delaunay_mesh_criteria_2<CDT>* is a model for the *MeshingCriteria_2* concept. The shape criterion on triangles is given by a bound B such that for good triangles $\frac{r}{l} \leq B$ where l is the shortest edge length and r is the circumradius of the triangle. By default, $B = \sqrt{2}$, which is the best bound one can use with the guarantee that the refinement algorithm will terminate. The upper bound B is related to a lower bound α_{min} on the minimum angle in the triangle:

$$\sin \alpha_{min} = \frac{1}{2B}$$

so $B = \sqrt{2}$ corresponds to $\alpha_{min} \geq 20.7$ degrees.

```
#include <CGAL/Delaunay_mesh_criteria_2.h>
```

Is Model for the Concepts

MeshingCriteria_2

Creation

Delaunay_mesh_criteria_2<CDT> traits; Default constructor. $B = \sqrt{2}$.

Delaunay_mesh_criteria_2<CDT> traits(double $b = 0.125$);

Construct a traits class with bound $B = \sqrt{\frac{1}{4b}}$.

CGAL::Delaunay_mesh_face_base_2<Traits, Fb>

Definition

The class *Delaunay_mesh_face_base_2<Traits, Fb>* is a model for the concept *DelaunayMeshFaceBase_2*.

This class can be used directly or it can serve as a base to derive other classes with some additional attributes (a color for example) tuned to a specific application.

```
#include <CGAL/Delaunay_mesh_face_base_2.h>
```

Parameters

- The first parameter *Traits* is the geometric traits class. It must be the same as the one used for the Delaunay mesh.
- The second parameter *Fb* is the base class from which *Delaunay_mesh_face_base_2<Traits, Fb>* derives. It must be a model of the *TriangulationFaceBase_2* concept.

Inherits From

Fb

Is Model for the Concepts

DelaunayMeshFaceBase_2

CGAL::Delaunay_mesh_size_criteria_2<CDT>

Definition

The class *Delaunay_mesh_size_criteria_2<CDT>* is a model for the *MeshingCriteria_2* concept. The shape criterion on triangles is given by a bound B such that for good triangles $\frac{l}{r} \leq B$ where l is the shortest edge length and r is the circumradius of the triangle. By default, $B = \sqrt{2}$, which is the best bound one can use with the guarantee that the refinement algorithm will terminate. The upper bound B is related to a lower bound α_{min} on the minimum angle in the triangle:

$$\sin \alpha_{min} = \frac{1}{2B}$$

so $B = \sqrt{2}$ corresponds to $\alpha_{min} \geq 20.7$ degrees.

This traits class defines also a size criteria: all segments of all triangles must be shorter than a bound S .

```
#include <CGAL/Delaunay_mesh_size_criteria_2.h>
```

Is Model for the Concepts

MeshingCriteria_2

Creation

Delaunay_mesh_size_criteria_2<CDT> traits; Default constructor. $B = \sqrt{2}$. No bound on size

Delaunay_mesh_size_criteria_2<CDT> traits(double $b = 0.125$, double $S = 0$);

Construct a traits class with bound $B = \sqrt{\frac{1}{4b}}$. If $S \neq 0$, the size bound is S . If $S = 0$, there is no bound on size.

CGAL::Mesh_2::Face_badness

```
#include <CGAL/Mesh_2/Face_badness.h>
```

```
enum Mesh_2::Face_badness { NOT_BAD, BAD, IMPERATIVELY_BAD};
```

CGAL::make_conforming_Delaunay_2

```
#include <CGAL/Triangulation_conformer_2.h>
```

```
template<class CDT>  
void make_conforming_Delaunay_2( CDT &t)
```

Refines the constrained Delaunay triangulation t into a conforming Delaunay triangulation. After a call to this function, all edges of t are Delaunay edges.

Requirement: The template parameter CDT should be a model of the concept *ConstrainedDelaunayTriangulation_2*. The geometric traits class of into the constrained Delaunay triangulation must be a model of *ConformingDelaunayTriangulationTraits_2*.

CGAL::make_conforming_Gabriel_2

```
#include <CGAL/Triangulation_conformer_2.h>
```

```
template<class CDT>  
void make_conforming_Gabriel_2( CDT &t)
```

Refines the constrained Delaunay triangulation t into a conforming Gabriel triangulation. After a call to this function, all constrained edges of t have the *Gabriel property*: the circle that has e as diameter does not contain any vertex from the triangulation.

Requirement: The template parameter CDT should be a model of the concept *ConstrainedDelaunayTriangulation_2*. The geometric traits class of the constrained Delaunay triangulation must be a model of *ConformingDelaunayTriangulationTraits_2*.

MeshingCriteria_2

Definition

The concept `MeshingCriteria_2` defines the meshing criteria to be used in the algorithm. It provides a predicate `Is_bad` that tests a triangle according to criteria. The return type of `Is_bad` is an enum `Mesh_2::Face_badness`.

The possible values of `Mesh_2::Face_badness` are `NOT_BAD`, `BAD` and `IMPERATIVELY_BAD`. If the predicate returns `BAD`, the triangle is marked as bad and the algorithm will try to destroy it. If the predicate returns `IMPERATIVELY_BAD`, the algorithm will destroy the triangle unconditionally during its execution.

The termination of the algorithm is guaranteed when criteria are shape criteria corresponding to a bound on smallest angles not less than 20.7 degrees (this corresponds to a radius-edge ratio bound not less than $\sqrt{2}$). Any size criteria that are satisfied by small enough tetrahedra can be added to the set of criteria without compromising the termination.

Note that, in the presence of input angles smaller than 60 degrees, some bad shaped triangles can appear in the finale mesh in the neighboring of small angles. To achieve termination and the respect of size criteria everywhere, the `Is_bad` predicate has to return `IMPERATIVELY_BAD` when size criteria are not satisfied, and `BAD` when shape criteria are not satisfied.

`MeshingCriteria_2` also provides a type `Quality` designed to code a quality measure for triangles. The type `Quality` must be *less-than comparable* as the meshing algorithm will order bad triangles by quality, to split those with smallest quality first. The predicate `Is_bad` computes the quality of the triangle as a by-product.

Types

<code>MeshingCriteria_2:: Face_handle</code>	Handle to a face of the triangulation.
<code>MeshingCriteria_2:: Quality</code>	Default constructible, copy constructible, assignable, and less-than comparable type.
<code>MeshingCriteria_2:: Is_bad</code>	Predicate object. Must provide two operators. The first operator <code>Mesh_2::Face_badness operator()(Face_handle fh, Quality& q)</code> returns <code>NOT_BAD</code> if it satisfies the desired criteria for mesh triangles, <code>BAD</code> if it does not, and <code>IMPERATIVELY_BAD</code> if it does not and should be refined unconditionally. In addition, this operator assigns to <code>q</code> a value measuring the quality of the triangle pointed by <code>fh</code> . The second operator <code>Mesh_2::Face_badness operator()(Quality q)</code> returns <code>NOT_BAD</code> if <code>q</code> is the quality of a good triangle, <code>BAD</code> if the <code>q</code> represents a poor quality, and <code>IMPERATIVELY_BAD</code> if <code>q</code> represents the quality of a bad triangle that should be refined unconditionally.

Access to predicate and constructor objects

`Is_bad` `traits.is_bad_object()`

Has Models

Delaunay_mesh_criteria_2<CDT>

Delaunay_mesh_size_criteria_2<CDT>

CGAL::refine_Delaunay_mesh_2

```
template<class CDT, class Criteria>
void refine_Delaunay_mesh_2( CDT &t, Criteria criteria = Criteria())
```

Refines the default domain defined by a constrained Delaunay triangulation without seeds into a mesh satisfying the criteria defined by the traits *criteria*. The domain of the mesh covers all the connected components of the plane defined by the constrained edges of *t*, except for the unbounded component.

Precondition: The template parameter *CDT* must be a model of the concept *ConstrainedDelaunayTriangulation_2*. The geometric traits class of the constrained Delaunay triangulation must be a model of *DelaunayMeshTraits_2*.

Requirement: The face of the constrained Delaunay triangulation must be a model of the concept *DelaunayMeshFaceBase_2*. *Criteria* must be a model of the concept *MeshingCriteria_2* and *CDT::Face_handle* must be the same as *Criteria::Face_handle*.

```
template <class CDT, class Criteria, class InputIterator>
void refine_Delaunay_mesh_2( CDT& t,
                             InputIterator begin,
                             InputIterator end,
                             Criteria criteria = Criteria(),
                             bool mark = false)
```

Refines the default domain defined by a constrained Delaunay triangulation into a mesh satisfying the criteria defined by the traits *criteria*. The sequence [*begin*, *end*] gives a set of seeds points, that defines the domain to be meshed as follows. The constrained edges of *t* partition the plane into connected components. If *mark=true*, the mesh domain is the union of the bounded connected components including at least one seed. If *mark=false*, the domain is the union of the bounded components including no seed. Note that the unbounded component of the plane is never meshed.

Requirement: The *value_type* of *begin* and *end* is *CDT::Geom_traits::Point_2*.

CGAL::Triangulation_conformer_2<CDT>

The class *Triangulation_conformer_2<CDT>* is an auxiliary class of *Delaunay_mesher_2<CDT>*. It permits to refine a constrained Delaunay triangulation into a conforming Delaunay or conforming Gabriel triangulation. For standard needs, consider using the global functions *make_conforming_Gabriel_2* and *make_conforming_Delaunay_2*.

Parameters

The template parameter *CDT* should be a model of the concept *ConstrainedDelaunayTriangulation_2*.

The geometric traits class of the instance of *CDT* has to be a model of the concept *ConformingDelaunayTriangulationTraits_2*.

Using this class

The constructor of the class *Triangulation_conformer_2<CDT>* takes a reference to a *CDT* as an argument. A call to the method *make_conforming_Delaunay()* or *make_conforming_Gabriel()* will refine this constrained Delaunay triangulation into a conforming Delaunay or conforming Gabriel triangulation. Note that if, during the life time of the *Triangulation_conformer_2<CDT>* object, the triangulation is externally modified, any further call to its member methods may lead to undefined behavior. Consider reconstructing a new *Triangulation_conformer_2<CDT>* object if the triangulation has been modified.

The conforming methods insert points into constrained edges, thereby splitting them into several sub-constraints. You have access to the initial inserted constraints if you instantiate the template parameter by a *CGAL::Constrained_triangulation_plus_2<CDT>*.

```
#include <CGAL/Triangulation_conformer_2.h>
```

Creation

```
Triangulation_conformer_2<CDT> m( CDT& t);
```

Create a new conforming maker, working on *t*.

Operations

Conforming methods

```
void m.make_conforming_Delaunay()
```

Refines the triangulation into a conforming Delaunay triangulation. After a call to this method, all triangles fulfill the Delaunay property, that is the empty circle property.

void *m.make_conforming_Gabriel()*

Refines the triangulation into a conforming Gabriel triangulation. After a call to this method, all constrained edges *e* have the *Gabriel property*: the circle with diameter *e* does not contain any vertex of the triangulation.

Checking

The following methods verify that the constrained triangulation is conforming Delaunay or conforming Gabriel. These methods scan the whole triangulation and their complexity is proportional to the number of edges.

bool *m.is_conforming_Delaunay()*

Returns *true* iff all triangles fulfill the Delaunay property.

bool *m.is_conforming_Gabriel()*

Returns *true* iff all constrained edges have the Gabriel property: their circumsphere is empty.

— *advanced* —

Step by step operations

The *Triangulation_conformer_2<CDT>* class allows, for debugging or demos, to play the conforming algorithm step by step, using the following methods. They exist in two versions, depending on whether you want the triangulation to be conforming Delaunay or conforming Gabriel, respectively. Any call to a *step_by_step_conforming_XX* function requires a previous call to the corresponding function *init_XX* and Gabriel and Delaunay methods can not be mixed between two calls of *init_XX*.

void *m.init_Delaunay()*

The method must be called after all points and constrained segments are inserted and before any call to the following methods. If some points or segments are then inserted in the triangulation, this method must be called again.

bool *m.step_by_step_conforming_Delaunay()*

Applies one step of the algorithm, by inserting one point, if the algorithm is not done. Returns *false* iff no point has been inserted because the algorithm is done.

void *m.init_Gabriel()*

Analog to *init_Delaunay* for Gabriel conforming.

bool *m.step_by_step_conforming_Gabriel()*

Analog to *step_by_step_conforming_Delaunay()* for Gabriel conforming.

bool

m.is_conforming_done()

Tests if the step by step conforming algorithm is done. If it returns *true*, the following calls to *step_by_step_conforming_XX* will not insert any points, until some new constrained segments or points are inserted in the triangulation and *init_XX* is called again.

└────────── *advanced* ─────────┘

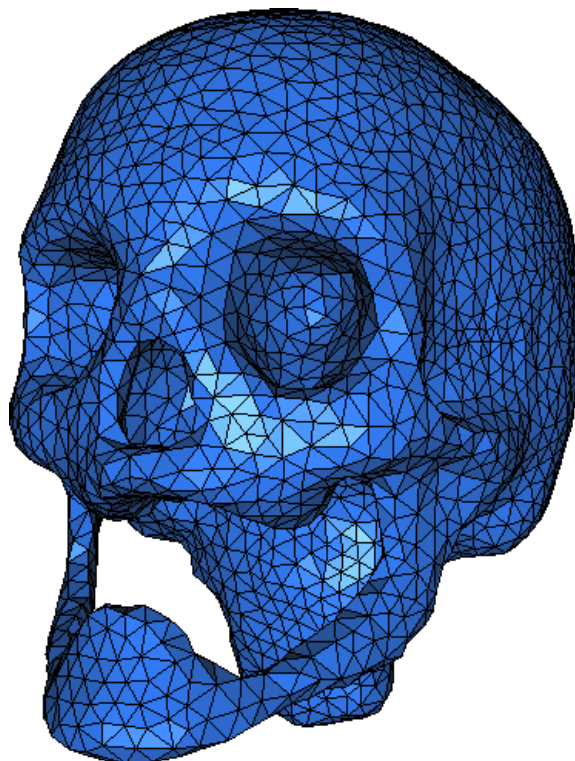
Chapter 30

3D Surface Mesher

Laurent Rineau and Mariette Yvinec

Contents

30.1 Introduction	1822
30.2 The Surface Mesher Interface	1822
30.3 Examples	1823
30.3.1 Meshing an implicit surface	1823
30.3.2 Meshing a surface defined as a gray level in a 3D image	1825
30.4 Meshing Criteria, Guarantees and Variations	1826
30.5 Design and Implementation History	1827



30.1 Introduction

This package provides a function template to compute a triangular mesh approximating a surface.

The meshing algorithm requires to know the surface to be meshed only through an oracle able to tell whether a given segment, line or ray intersects the surface or not and to compute an intersection point if any. This feature makes the package generic enough to be applied in a wide variety of situations. For instance, it can be used to mesh implicit surfaces described as the zero level set of some function. It may also be used in the field of medical imaging to mesh surfaces described as a gray level set in a three dimensional image.

The meshing algorithm is based on the notion of the restricted Delaunay triangulation. Basically the algorithm computes a set of sample points on the surface, and extract an interpolating surface mesh from the three dimensional triangulation of these sample points. Points are iteratively added to the sample, as in a Delaunay refinement process, until some size and shape criteria on mesh elements are satisfied.

The size and shape criteria guide the behaviour of the refinement process and control its termination. They also condition the size and shape of the elements in the final mesh. Naturally, those criteria can be customized to satisfy the user needs. The *Surface mesher* package offers a set of standard criteria that can be scaled through three numerical values. Also the user can also plug in its own set of refinement criteria.

There is no restriction on the topology and number of components of the surface provided that the oracle (or the user) is able to provide one initial sample point on each connected component. If the surface is smooth enough, and if the size criteria are small enough, the algorithm guarantees that the output mesh is homeomorphic to the surface, and is within a small bounded distance (Hausdorff or even Frechet distance) from the surface. The algorithm can also be used for non smooth surfaces but then there is no guarantee.

30.2 The Surface Mesher Interface

The meshing process is launched through a call to a function template. There are two overloaded versions of the meshing function whose signatures are the following:

```
template <class SurfaceMeshC2T3, class Surface, class Criteria, class Tag >
void      make_surface_mesh( SurfaceMeshC2T3& c2t3,
                           Surface surface,
                           Criteria criteria,
                           Tag)
```

```
template <class SurfaceMeshC2T3, class SurfaceMeshTraits, class Criteria, class Tag >
void      make_surface_mesh( SurfaceMeshC2T3& c2t3,
                           SurfaceMeshTraits::Surface_3 surface,
                           SurfaceMeshTraits traits,
                           Criteria criteria,
                           Tag)
```

The template parameter *SurfaceMeshC2T3* stands for a data structure type that is used to store the surface mesh. This type is required to be a model of the concept *SurfaceMeshComplex_2InTriangulation_3*. Such a data structure has a pointer to a three dimensional triangulation and encodes the surface mesh as a subset of facets in this triangulation. An argument of type *SurfaceMeshC2T3* is passed by reference to the meshing function. This argument holds the output mesh at the end of the process.

The template parameter *Surface* stands for the surface type. This type has to be a model of the concept *Surface_3*.

The knowledge on the surface, required by the surface mesher is encapsulated in a traits class. Actually, the mesher accesses the surface to be meshed through this traits class only. The traits class is required to be a model of the concept *SurfaceMeshTraits_3*. The difference between the two overloaded versions of *make_surface_mesh* can be explained as follows

- In the first overloaded version of *make_surface_mesh*, the surface type is given as template parameter (*Surface*) and the *surface* to be meshed is passed as parameter to the mesher. In that case the surface mesher traits type is automatically generated from the surface type by an auxiliary class called the *Surface_mesh_traits_generator_3*.
- In the second overloaded version of *make_surface_mesh*, the surface mesher traits type is provided by the template parameter *SurfaceMeshTraits_3* and the surface type is obtained from this traits type. Both a surface and a traits are passed to the mesher as arguments.

The first overloaded version can be used whenever the surface type either provides a nested type *Surface::Surface_mesher_traits_3* that is a model of *SurfaceMeshTraits_3* or is a surface type for which a specialization of the traits generator *Surface_mesh_traits_generator_3<Surface>* is provided. Currently, the library provides partial specializations of *Surface_mesher_traits_generator_3<Surface>* for implicit surfaces (*Implicit_surface_3<Traits, Function>*) and grey level images (*Gray_level_image_3<FT, Point>*).

The parameter *criteria* handles the description of the size and shape criteria driving the meshing process. The template parameter *Criteria* has to be instantiated by a model of the concept *MeshCriteria*.

The parameter *Tag* is a tag whose type influences the behavior of the meshing algorithm. For instance, this parameter can be used to enforce the manifold property of the output mesh while avoiding an over-refinement of the mesh. Further details on this subject are given in Section 30.4.

A call to *make_surface_mesh(c2t3, surface, criteria, tag)* launches the meshing process with an initial set of points which is the union of two subsets: the set of vertices in the initial triangulation pointed to by *c2t3*, and a set of points provided by the *Compute_initial_points()* functor of the traits class. This initial set of points is required to include at least one point on each connected component of the surface to be meshed.

30.3 Examples

30.3.1 Meshing an implicit surface

The first code example meshes a sphere given as the zero level set of a function $\mathbb{R}^3 \longrightarrow \mathbb{R}$. More precisely, the surface to be meshed is created by the constructor of the class *Implicit_surface_3<Kernel, Function>* from a pointer to the function (*sphere_function*) and a bounding sphere.

The default meshing criteria are determined by three numerical values:

- *angular_bound* is a lower bound in degrees for the angles of mesh facets.
- *radius_bound* is an upper bound on the radii of surface Delaunay balls. A surface Delaunay ball is a ball circumscribing a mesh facet and centered on the surface.
- *distance_bound* is an upper bound for the distance between the circumcenter of a mesh facet and the center of a surface Delaunay ball of this facet.

Given this surface type, the surface mesher will use an automatically generated traits class.

The resulting mesh is shown on figure 30.1.

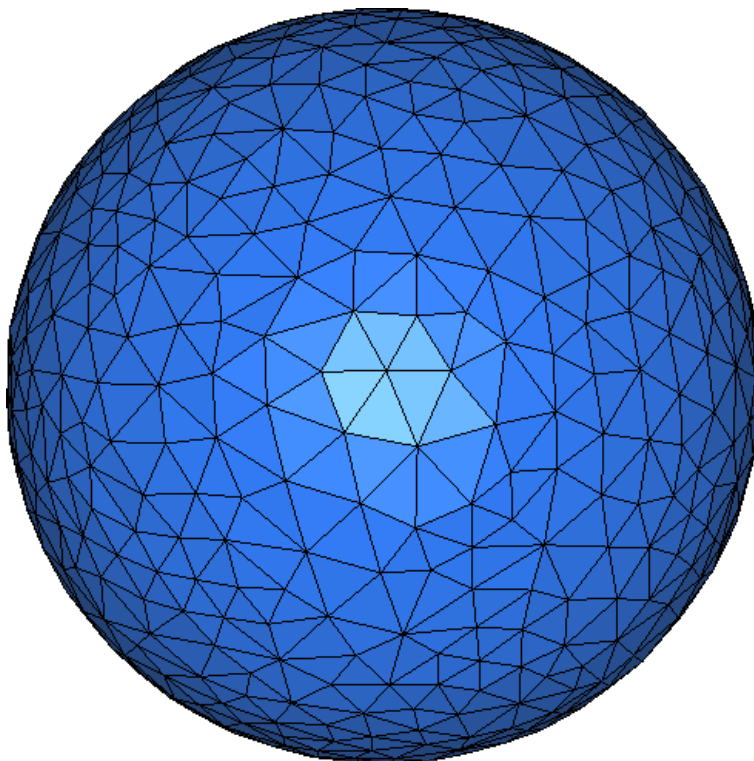


Figure 30.1: Surface mesh of a sphere

```
// file examples/Surface_mesher/implicit_surface_mesher.C
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/make_surface_mesh.h>
#include <CGAL/Implicit_surface_3.h>

struct Kernel : public CGAL::Exact_predicates_inexact_constructions_kernel {};
typedef CGAL::Surface_mesh_vertex_base_3<Kernel> Vb;
typedef CGAL::Surface_mesh_cell_base_3<Kernel> Cb;
typedef CGAL::Triangulation_data_structure_3<Vb, Cb> Tds;
typedef CGAL::Delaunay_triangulation_3<Kernel, Tds> Tr;
typedef CGAL::Surface_mesh_complex_2_in_triangulation_3<Tr> C2t3;
typedef Kernel::Sphere_3 Sphere_3;
typedef Kernel::Point_3 Point_3;
typedef Kernel::FT FT;

typedef FT (*Function) (Point_3);

typedef CGAL::Implicit_surface_3<Kernel, Function> Surface_3;

FT sphere_function (Point_3 p) {
    const FT x2=p.x()*p.x(), y2=p.y()*p.y(), z2=p.z()*p.z();
    return x2+y2+z2-1;
}
```



```

}

int main(int, char **) {
    Tr tr;          // 3D-Delaunay triangulation
    C2t3 c2t3 (tr); // 2D-complex in 3D-Delaunay triangulation

    // defining the surface
    Surface_3 surface(sphere_function,          // pointer to function
                     Sphere_3(CGAL::ORIGIN, 2.)); // bounding sphere

    // defining meshing criteria
    CGAL::Surface_mesh_default_criteria_3<Tr> criteria(30., // angular bound
                                                       0.1, // radius bound
                                                       0.1); // distance bound

    // meshing surface
    make_surface_mesh(c2t3, surface, criteria, CGAL::Non_manifold_tag());

    std::cout << "Final number of points: " << tr.number_of_vertices() << "\n";
}

```

30.3.2 Meshing a surface defined as a gray level in a 3D image

In this example the surface to be meshed is defined as the locus of points with a given gray level in a 3D image. The code is quite similar to the previous example.

The main difference with the previous code is that the function used to define the surface is an object of type `CGAL::Gray_level_image_3` created from an image file and a numerical value that is the gray value of the level one wishes to mesh.

Note that surface, which is still an object of type `Implicit_surface_3` is now, defined by three parameters that are the function, the bounding sphere and a numerical value called *the precision*. This precision, whose value is relative to the bounding sphere radius, is used in the intersection computation. This parameter has a default which was used in the previous example. Also note that the center of the bounding sphere is required to be internal a point where the function has a negative value.

The chosen iso-value of this 3D image corresponds to a head skull. The resulting mesh is the introducing picture of this chapter.

```

// file examples/Surface_mesher/3d_image_surface_mesher.C
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/make_surface_mesh.h>
#include <CGAL/Gray_level_image_3.h>
#include <CGAL/Implicit_surface_3.h>

struct Kernel : public CGAL::Exact_predicates_inexact_constructions_kernel {};
typedef CGAL::Surface_mesh_vertex_base_3<Kernel> Vb;
typedef CGAL::Surface_mesh_cell_base_3<Kernel> Cb;
typedef CGAL::Triangulation_data_structure_3<Vb, Cb> Tds;
typedef CGAL::Delaunay_triangulation_3<Kernel, Tds> Tr;
typedef CGAL::Surface_mesh_complex_2_in_triangulation_3<Tr> C2t3;

typedef CGAL::Gray_level_image_3<Kernel::FT, Kernel::Point_3> Gray_level_image;

```

```

typedef CGAL::Implicit_surface_3<Kernel, Gray_level_image> Surface_3;

int main(int, char **) {
    Tr tr;          // 3D-Delaunay triangulation
    C2t3 c2t3 (tr); // 2D-complex in 3D-Delaunay triangulation

    // the 'function' is a 3D gray level image
    Gray_level_image image("ImageIO/data/skull_2.9.inr.gz", 2.9);

    // Carefully choosen bounding sphere: the center must be inside the
    // surface defined by 'image' and the radius must be high enough so that
    // the sphere actually bounds the whole image.
    Kernel::Point_3 bounding_sphere_center(122., 102., 117.);
    Kernel::FT bounding_sphere_squared_radius = 200.*200.*2.;
    Kernel::Sphere_3 bounding_sphere(bounding_sphere_center,
                                     bounding_sphere_squared_radius);

    // definition of the surface, with 10-2 as relative precision
    Surface_3 surface(image, bounding_sphere, 1e-2);

    // defining meshing criteria
    CGAL::Surface_mesh_default_criteria_3<Tr> criteria(30.,
                                                       5.,
                                                       5.);

    // meshing surface, with the "manifold without boundary" algorithm
    make_surface_mesh(c2t3, surface, criteria, CGAL::Manifold_tag());

    std::cout << "Final number of points: " << tr.number_of_vertices() << "\n";
}

```

30.4 Meshing Criteria, Guarantees and Variations

The guarantees on the output mesh depend on the mesh criteria. Theoretical guarantees are given in [BO05]. First, the meshing algorithm is proved to terminate if the angular bound is not smaller than 30 degrees. Furthermore, the output mesh is guaranteed to be homeomorphic to the surface, and there is a guaranteed bound on the distance (Hausdorff and even Frechet distance) between the mesh and the surface if the radius bound is everywhere smaller than the ε times the local feature size. Here ε is a constant that has to be less than 0.16, and the local feature size $lfs(x)$ is defined on each point x of the surface as the distance from x to the medial axis. Note that the radius bound need not be uniform, although it is a uniform bound in the default criteria.

Naturally, such a theoretical guarantee can be only achieved for smooth surfaces that have a finite, non zero reach value. (The reach of a surface is the minimum value of local feature size on this surface).

The value of the local feature size on any point of the surface or its minimum on the surface it usually unknown although it can sometimes be guessed. Also it happens frequently that setting the meshing criteria so as to fulfill the theoretical conditions yields an over refined mesh. On the other hand, when the size criteria are relaxed, no homeomorphism with the input surface is guaranteed, and the output mesh is not even guaranteed to be manifold. To remedy this problem and give a more flexible meshing algorithm, the function template *make_surface_mesh* has a tag template parameter allowing to slightly change the behavior of the refinement process. This feature allows, for instance, to run the meshing algorithm with a relaxed size criteria, more coherent with the size of the mesh expected by the user, and still have a guarantee that the output mesh forms a manifold

surface. The function *make_surface_mesh* has specialized versions for the following tag types:

Manifold_tag: the output mesh is guaranteed to be a manifold surface without boundary.

Manifold_with_boundary_tag: the output mesh is guaranteed to be manifold but may have boundaries.

Non_manifold_tag: the algorithm relies on the given criteria and guarantees nothing else.

30.5 Design and Implementation History

The algorithm implemented in this package is mainly based on the work of Boissonnat and Oudot [BO05]

The meshing algorithm is implemented using the design of mesher levels described in [RY06].

David Rey, Steve Oudot and Andreas Fabri have participated in the development of this package.

3D Surface Mesher

Reference Manual

Laurent Rineau and Mariette Yvinec

The surface mesher package offers a function template which builds a triangular mesh approximating a surface.

The meshing algorithms requires to know the surface to be meshed through an oracle that mainly can tell whether a given segment, ray or line intersects the surface or not and can compute the intersections point if any. The oracle is represented by a traits class which can be passed to the meshing function or automatically generated for certain types of surfaces. The current implementation provides traits classes to mesh implicit surfaces as well as surfaces described as a gray level in a three dimensional image.

The output mesh conforms to some size and shape criteria which are customizable. The criteria are passed to the mesher through a parameter whose type is a model of the concept *SurfaceMeshCriteria_3*.

The meshing algorithm is a Delaunay refinement process which is mainly guided by the criteria. The output mesh may offer some guarantees, as being manifold, homeomorphic to the surface or within a given Hausdorff distance. However, these guarantees depend on the quality of the input surface (smoothness, with or without boundary, manifold or not), the type and values of the given criteria. The behavior of the refinement process can also be influenced through a tag, which allows for instance to enforce the manifold property of the output mesh while avoiding an over-refinement of the mesh.

30.6 Classified Reference Pages

Concepts

SurfaceMeshComplex_2InTriangulation_3	page 1847
SurfaceMeshTraits_3	page 1854
SurfaceMeshCriteria_3	page 1852
SurfaceMeshCellBase_3	page 1844
SurfaceMeshVertexBase_3	page 1864
SurfaceMeshTriangulation_3	page 1857
ImplicitFunction	page 1832
ImplicitSurfaceTraits_3	page 1834

Classes

<i>CGAL::Surface_mesh_complex_2_in_triangulation_3<Tr></i>	page 1846
<i>CGAL::Surface_mesh_vertex_base_3<Gt,Vb></i>	page 1863
<i>CGAL::Surface_mesh_cell_base_3<Gt,Cb></i>	page 1843
<i>CGAL::Surface_mesh_default_criteria_3<Tr></i>	page 1853
<i>CGAL::Surface_mesh_traits_generator_3<Surface></i>	page 1856
<i>CGAL::Implicit_surface_3<Traits, Function></i>	page 1833
<i>CGAL::Gray_level_image_3<FT, Point></i>	page 1831

Tag Classes

<i>CGAL::Manifold_tag</i>	page 1839
<i>CGAL::Manifold_with_boundary_tag</i>	page 1840
<i>CGAL::Non_manifold_tag</i>	page 1841

Function Templates

<i>CGAL::make_surface_mesh</i>	page 1837
--------------------------------------	-----------

30.7 Alphabetical List of Reference Pages

<i>Gray_level_image_3<FT, Point></i>	page 1831
<i>ImplicitFunction</i>	page 1832
<i>ImplicitSurfaceTraits_3</i>	page 1834
<i>Implicit_surface_3<Traits, Function></i>	page 1833
<i>make_surface_mesh</i>	page 1837
<i>Manifold_tag</i>	page 1839
<i>Manifold_with_boundary_tag</i>	page 1840
<i>Non_manifold_tag</i>	page 1841
<i>SurfaceMeshCellBase_3</i>	page 1844
<i>SurfaceMeshComplex_2InTriangulation_3</i>	page 1847
<i>SurfaceMeshCriteria_3</i>	page 1852
<i>SurfaceMeshTraits_3</i>	page 1854
<i>SurfaceMeshTriangulation_3</i>	page 1857
<i>SurfaceMeshVertexBase_3</i>	page 1864
<i>Surface_3</i>	page 1842
<i>Surface_mesh_cell_base_3<Gt,Cb></i>	page 1843
<i>Surface_mesh_complex_2_in_triangulation_3<Tr></i>	page 1846
<i>Surface_mesh_default_criteria_3<Tr></i>	page 1853
<i>Surface_mesh_traits_generator_3<Surface></i>	page 1856
<i>Surface_mesh_vertex_base_3<Gt,Vb></i>	page 1863

CGAL::Gray_level_image_3<FT, Point>

Definition

A 3D gray image is a tri-dimensional array that associates a scalar value to each triple of integer (x, y, z) in the range of the image. A trilinear interpolation algorithm provides a map $f : \mathbb{R}^3 \rightarrow \mathbb{R}$.

The class *Gray_level_image_3<FT, Point>* is a 3D gray image loader and a model of the concept *ImplicitFunction*. An object of the class *Gray_level_image_3<FT, Point>* is created with a parameter *iso* and then its *operator()* implements the function *sign of (f(p) - iso)*, for $p \in \mathbb{R}^3$. Plugging such a function in the creation of the *Implicit_surface_3* object given as parameter to *make_surface_mesh* yields a mesh approximating the level with value *iso* in the input 3D gray image.

Gray_level_image_3<FT, Point> provides an interface with an auxiliary library called *ImageIO*. An executable that uses *Gray_level_image_3<FT, Point>* must be linked with the *ImageIO* library. This library is shipped with CGAL in the *examples/Surface_mesher/* subdirectory.

The library *ImageIO* and therefore *Gray_level_image_3<FT, Point>* support several types of 3D images: IN-RIMAGE (extension *.inr[.gz]*), GIS (extension *.dim*, of *.ima[.gz]*), and ANALYZE (extension *.hdr*, or *.img[.gz]*).

```
#include <CGAL/Gray_level_image_3.h>
```

Is Model for the Concepts

ImplicitFunction

Types

Gray_level_image_3<FT, Point>:: FT the numerical type *FT*

Gray_level_image_3<FT, Point>:: Point the point type *Point*

Creation

```
Gray_level_image_3<FT, Point> image( const char* filename, FT iso_value);
```

filename is the path to a file of a type supported by *ImageIO*.
iso_value is an isovalue of *f*.

See Also

ImplicitFunction,
Implicit_surface_3<Traits, Function>,
make_surface_mesh

The concept `ImplicitFunction` describes a function object whose `operator()` computes the values of a function $f: \mathbb{R}^3 \rightarrow \mathbb{R}$.

<i>ImplicitFunction::FT</i>	Number type
-----------------------------	-------------

Operations

<i>FT</i>	<i>function(Point p)</i>	Returns the value $f(p)$, where $p \in \mathbb{R}^3$.
-----------	---------------------------	---------------------------------------------------------

Gray_level_image_function,
any pointer to a function of type $FT(*) (Point)$.

```
Implicit_surface_3<Traits, Function>,
make_surface_mesh
```


CGAL::Implicit_surface_3<Traits, Function>

Definition

The class *Implicit_surface_3*<*Traits*, *Function*> implements a surface described as the zero level set of a function $f : \mathbb{R}^3 \longrightarrow \mathbb{R}$.

For this type of surface, the library provides a partial specialization of the surface mesher traits generator: *Surface_mesh_traits_generator_3*<*Implicit_surface_3*<*Traits*, *Function*> >, that provides a traits class, model of the concept *SurfaceMeshTraits_3*, to be used by the surface mesher.

The parameter *Traits* is a traits class that has to be implemented with a model of *ImplicitSurfaceTraits_3*. Actually, this traits class implements the oracle needed by the surface mesher: the types, predicates and constructors provided in *Traits* are passed by the surface mesher traits generator to the generated the traits class used by the surface mesher.

The template parameter *Function* stands for a model of the concept *ImplicitFunction*. The number type *Function::FT* has to match the type *Traits::FT*.

```
#include <CGAL/Implicit_surface_3.h>
```

Creation

```
Implicit_surface_3<Traits, Function> surface( Function f,
                                             Sphere_3 bounding_sphere,
                                             FT error_bound = FT(1e-3))
```

f is the object of type *Function* that represents the implicit surface.

bounding_sphere is a bounding sphere of the implicit surface. The evaluation of *f* at the center *c* of this sphere must be negative: $f(c) < 0$.

error_bound is a relative error bound used to compute intersection points between the implicit surface and query segments. This bound is used in the default generated traits class. In this traits class, the intersection points between the surface and segments/rays/line are constructed by dichotomy. The dichotomy is stopped when the size of the intersected segment is less than the product of *error_bound* by the radius of *bounding_sphere*.

See Also

make_surface_mesh,
Surface_mesh_traits_generator_3<*Surface*>,
ImplicitSurfaceTraits,
ImplicitFunction.

ImplicitSurfaceTraits_3

Definition

The concept `ImplicitSurfaceTraits_3` describes the requirements of the traits class to be plugged as *Traits* in `Implicit_surface_3<Traits, Function>`.

When `make_surface_mesh` is called with a surface of type `Implicit_surface_3<Traits,Function>`, the surface mesher traits generator generates automatically a traits class that is a model of `SurfaceMeshTraits_3`. Actually, the concept `ImplicitSurfaceTraits_3` provides the types, predicates and constructors that are passed to the generated model of `SurfaceMeshTraits_3`.

Types

`ImplicitSurfaceTraits_3:: FT` The numerical type. It must be model of *SqrtFieldNumber-Type* and constructible from a *double*.

`ImplicitSurfaceTraits_3:: Point_3` The point type. This point type must have a constructor `Point_3(FT, FT, FT)`.

`ImplicitSurfaceTraits_3:: Line_3` The line type.

`ImplicitSurfaceTraits_3:: Ray_3` The ray type.

`ImplicitSurfaceTraits_3:: Segment_3` The segment type.

`ImplicitSurfaceTraits_3:: Vector_3` The vector type.

`ImplicitSurfaceTraits_3:: Sphere_3` The sphere type.

`ImplicitSurfaceTraits_3:: Compute_scalar_product_3`

A function object that provides the operator
`FT operator()(Vector_3 v, Vector_3 w)` which returns the scalar (inner) product of the two vectors *v* and *w*.

`ImplicitSurfaceTraits_3:: Compute_squared_distance_3`

A function object that provides the operator
`FT operator()(Point_3, Point_3)` which returns the squared distance between two points.

`ImplicitSurfaceTraits_3:: Compute_squared_radius_3`

A function object providing the operator
`FT operator()(const Sphere_3& s)` which returns the squared radius of *s*.

`ImplicitSurfaceTraits_3:: Construct_center_3`

A function object providing the operator
`Point_3 operator()(const Sphere_3& s)` which computes the center of the sphere *s*.

ImplicitSurfaceTraits_3:: Construct_midpoint_3

A function object providing the operator

Point_3 operator()(const Point_3& p, const Point_3& q) which computes the midpoint of the segment pq .

ImplicitSurfaceTraits_3:: Construct_point_on_3

A function object providing the following operators:

Point_3 operator()(const Line_3& l, int i); which returns an arbitrary point on l . It holds $point(i) == point(j)$, iff $i=j$. Furthermore, is directed from $point(i)$ to $point(j)$, for all $i < j$.

Point_3 operator()(const Ray_3& r, int i); which returns a point on r . $point(0)$ is the source, $point(i)$, with $i > 0$, is different from the source.

Precondition: $i \geq 0$.

Point_3 operator()(const Segment_3& s, int i); which returns source or target of s : $point(0)$ returns the source of s , $point(1)$ returns the target of s . The parameter i is taken modulo 2, which gives easy access to the other end point.

ImplicitSurfaceTraits_3:: Construct_segment_3

A function object providing the operators

Segment_3 operator()(const Point_3& p, const Point_3& q); which returns a segment with source p and target q . It is directed from the source towards the target.

ImplicitSurfaceTraits_3:: Construct_scaled_vector_3

A function object providing the operator

Vector_3 operator()(const Vector_3& v, const FT& scale) which returns the vector v scaled by a factor $scale$.

ImplicitSurfaceTraits_3:: Construct_translated_point_3

A function object providing the operator

Point_3 operator()(const Point_3& p, const Vector_3& v) which returns the point obtained by translating p by the vector v .

ImplicitSurfaceTraits_3:: Construct_vector_3

A function object providing the operator

Vector_3 operator()(const Point_3& a, const Point_3& b) which returns the vector $b-a$.

ImplicitSurfaceTraits_3:: Has_on_bounded_side_3

A function object providing the operator

bool operator()(const Sphere_3& s, const Point_3& p); which returns true iff p lies on the bounded side of s .

Operations

The following functions give access to the predicate and construction objects:

<i>Compute_scalar_product_3</i>	<i>traits.compute_scalar_product_3_object()</i>
<i>Compute_squared_distance_3</i>	<i>traits.compute_squared_distance_3_object()</i>
<i>Compute_squared_radius_3</i>	<i>traits.compute_squared_radius_3_object()</i>
<i>Construct_center_3</i>	<i>traits.construct_center_3_object()</i>
<i>Construct_midpoint_3</i>	<i>traits.construct_midpoint_3_object()</i>
<i>Construct_point_on_3</i>	<i>traits.construct_point_on_3_object()</i>
<i>Construct_scaled_vector_3</i>	<i>traits.construct_scaled_vector_3_object()</i>
<i>Construct_segment_3</i>	<i>traits.construct_segment_3_object()</i>
<i>Construct_translated_point_3</i>	<i>traits.construct_translated_point_3_object()</i>
<i>Construct_vector_3</i>	<i>traits.construct_vector_3_object()</i>
<i>Has_on_bounded_side_3</i>	<i>traits.has_on_bounded_side_3_object()</i>

Has Models

Any CGAL Kernel.

See Also

Implicit_surface_3<Traits, Function>,
make_surface_mesh

CGAL::make_surface_mesh

```
#include <CGAL/make_surface_mesh.h>
```

Definition

The function *make_surface_mesh* is a surface mesher, that is a function to build a two dimensional mesh approximating a surface.

The library provides two overloaded version of this function:

```
template <class SurfaceMeshC2T3, class Surface, class Criteria, class Tag >
void      make_surface_mesh( SurfaceMeshC2T3& c2t3,
                           Surface surface,
                           Criteria criteria,
                           Tag,
                           int initial_number_of_points = 20)
```

```
template <class SurfaceMeshC2T3, class SurfaceMeshTraits, class Criteria, class Tag >
void      make_surface_mesh( SurfaceMeshC2T3& c2t3,
                           SurfaceMeshTraits::Surface_3 surface,
                           SurfaceMeshTraits traits,
                           Criteria criteria,
                           Tag,
                           int initial_number_of_points = 20)
```

Parameters

The template parameter *SurfaceMeshC2T3* is required to be a model of the concept *SurfaceMeshComplex_2InTriangulation_3*, a data structure able to represent a two dimensional complex embedded in a three dimensional triangulation. The argument *c2t3* of type *SurfaceMeshC2T3*, passed by reference to the surface mesher, is used to maintain the current approximating mesh and it stores the final mesh at the end of the procedure. The type *SurfaceMeshC2T3* is in particular required to provide a type *SurfaceMeshC2T3::Triangulation_3* for the three dimensional triangulation embedding the surface mesh. The vertex and cell base classes of the triangulation *SurfaceMeshC2T3::Triangulation_3* are required to be models of the concepts *SurfaceMeshVertexBase_3* and *SurfaceMeshCellBase_3* respectively.

The template parameter *Surface* stands for the surface type. This type has to be a model of the concept *Surface_3*.

The knowledge on the surface, required by the surface mesher is encapsulated in a traits class. Actually, the mesher accesses the surface to be meshed through this traits class only. The traits class is required to be a model of the concept *SurfaceMeshTraits_3*.

In the first version of *make_surface_mesh* the surface type is a template parameter *Surface* and the surface mesher traits type is automatically generated from the surface type through the class *Surface_mesh_traits_generator_3<Surface>*.

The difference between the two overloaded versions of *make_surface_mesh* can be explained as follows

- In the first overloaded version of *make_surface_mesh*, the surface type is given as template parameter (*Surface*) and the *surface* to be meshed is passed as parameter to the mesher. In that case the surface mesher traits type is automatically generated from the surface type by an auxiliary class called the *Surface_mesh_traits_generator_3*.
- In the second overloaded version of *make_surface_mesh*, the surface mesher traits type is provided by the template parameter *SurfaceMeshTraits_3* and the surface type is obtained from this traits type. Both the surface and the traits are passed to the mesher as arguments.

The first overloaded version can be used whenever the surface type either provides a nested type *Surface::Surface_mesher_traits_3* that is a model of *SurfaceMeshTraits_3* or is a surface type for which a specialization of the traits generator *Surface_mesh_traits_generator_3<Surface>* is provided. Currently, the library provides partial specializations of *Surface_mesher_traits_generator_3<Surface>* for implicit surfaces (*Implicit_surface_3<Traits, Function>*) and grey level images (*Gray_level_image_3<FT, Point>*).

The template parameter *Criteria* has to be a model of the concept *SurfaceMeshCriteria_3*. The argument of type *Criteria* passed to the surface mesher specifies the size and shape requirements on the output surface mesh.

The template parameter *Tag* is a tag whose type affects the behaviour of the meshing algorithm. The function *make_surface_mesh* has specialized versions for the following tag types:

- *Manifold_tag*: the output mesh is guaranteed to be a manifold surface without boundary.
- *Manifold_with_boundary_tag*: the output mesh is guaranteed to be manifold but may have boundaries.
- *Non_manifold_tag*: the algorithm relies on the given criteria and guarantees nothing else.

The Delaunay refinement process is started with an initial set of points which is the union of two sets: the set of vertices in the initial triangulation pointed to by the *c2t3* argument and a set of points provided by the traits class. The optional parameter *initial_number_of_points* allows to monitor the number of points in this second set. (This parameter is passed to the *operator()* of the constructor object *Construct_initial_points* in the traits class.) The meshing algorithm requires that the initial set of points includes at least one point on each connected components of the surface to be meshed. one.

See Also

SurfaceMeshComplex_2InTriangulation_3
SurfaceMeshCellBase_3
SurfaceMeshVertexBase_3
Surface_3
SurfaceMeshCriteria_3

CGAL::Manifold_tag

Definition

The class *Manifold_tag* is a tag class used to monitor the surface meshing algorithm. When instantiated with the tag *Manifold_tag* the function template *make_surface_mesh* ensures that the output mesh is a manifold surface without boundary.

```
#include <CGAL/make_surface_mesh.h>
```

See Also

make_surface_mesh

Manifold_with_boundary_tag

Non_manifold_tag

CGAL::Manifold_with_boundary_tag

Definition

The class *Manifold_with_boundary_tag* is a tag class used to monitor the surface meshing algorithm. When instantiated with the tag *Manifold_with_boundary_tag*, the function template *make_surface_mesh* ensures that the output mesh is a manifold surface but it may have boundaries.

```
#include <CGAL/make_surface_mesh.h>
```

See Also

make_surface_mesh

Manifold_tag

Non_manifold_tag

CGAL::Non_manifold_tag

Definition

The class *Non_manifold_tag* is a tag class used to monitor the surface meshing algorithm. When instantiated with the tag *Non_manifold_tag* the function template *make_surface_mesh* does not ensure that the output mesh is a manifold surface. The manifold property of output mesh may nevertheless result from the choice of appropriate meshing criteria.

```
#include <CGAL/make_surface_mesh.h>
```

See Also

make_surface_mesh

Manifold_tag

Manifold_with_boundary_tag

Surface_3

Definition

The concept `Surface_3` describes the types of surfaces to be meshed. The surface types are required to be copy constructible and assignable.

Types

In addition, surface types are required

- either to provide a nested type: `Surface_3:: Surface_mesher_traits_3` a model of `SurfaceMesherTraits_3`
- or to be a surface type for which a specialization of the traits generator `Surface_mesh_traits_generator_3<Surface>` exists.

Has Models

`Implicit_surface_3<Traits, Function>`

See Also

`make_surface_mesh,`
`SurfaceMeshTraits_3`
`Surface_mesh_traits_generator_3<Surface>`

CGAL::Surface_mesh_cell_base_3<Gt,Cb>

Definition

The class *Surface_mesh_cell_base_3<Gt,Cb>* is a model of the concept *SurfaceMeshCellBase_3*. It is designed to serve as vertex base class in a triangulation class *Tr* plugged in a *Surface_mesh_complex_2_in_triangulation_3<Tr>* class.

The first template parameter is the geometric traits class.

The second template parameter is a base class. It has to be a model of the concept *TriangulationCellBase_3* and defaults to *Triangulation_cell_base_3 <GT>*.

```
#include <CGAL/Surface_mesh_cell_base_3.h>
```

Is Model for the Concepts

SurfaceMeshCellBase_3

Inherits From

Cb

See Also

SurfaceMeshComplex_2InTriangulation_3
Surface_mesh_complex_2_in_triangulation_3<Tr>
SurfaceMeshTriangulation_3
make_surface_mesh

SurfaceMeshCellBase_3

Definition

The concept `SurfaceMeshCellBase_3` describes the cell base type of the three dimensional triangulation used to embed the surface mesh.

More precisely, the first template parameter `SurfaceMeshC2T3` of the surface mesher `make_surface_mesh` is a model of the concept `SurfaceMeshComplex_2InTriangulation_3` which describes a data structure to store a pure two dimensional complex embedded in a three dimensional triangulation. In particular, the type `SurfaceMeshC2T3` is required to provide a three dimensional triangulation type `SurfaceMeshC2T3::Triangulation_3`. The concept `SurfaceMeshCellBase_3` describes the cell base type required in this triangulation type.

Generalizes

TriangulationCellBase_3

The concept `SurfaceMeshCellBase_3` adds four markers to mark the facets of the triangulation that belong to the two dimensional complex, and four markers that are helpers used in some operations to mark for instance the facets that have been visited.

This concept also provides storage for the center of a Delaunay surface ball. Given a surface and a 3D Delaunay triangulation, a Delaunay surface ball is a ball circumscribed to a facet of the triangulation and centered on the surface and empty of triangulation vertices. Such a ball does exist when the facet is part of the restriction to the surface of a three dimensional triangulation. In the following we call *surface center* of a facet, the center of its biggest Delaunay surface ball.

Types

SurfaceMeshCellBase_3::Point

The point type, required to match the point type of the three dimensional triangulation in which the surface mesh is embedded.

Creation

Operations

bool *cell.is_facet_on_surface(int i)*

returns *true*, if *facet(i)* is in the 2D complex.

void *cell.set_facet_on_surface(int i, bool b)*

Sets *facet(i)* as part of the 2D complex, if *b* is *true*, and *NOT_IN_COMPLEX*, otherwise.

bool *cell.is_facet_visited(int i)*

Returns *true*, if *facet(i)* has been visited, *false* otherwise.

void *cell.set_facet_visited(int i, bool b)*

Marks *facet(i)* as visited, if *b* is *true*, and non visited otherwise.

Point *cell.get_facet_surface_center(int i)*

Returns a const reference to the surface center of *facet(i)*.

void *cell.set_facet_surface_center(int i, Point p)*

Sets point *p* as the surface center of *facet(i)*.

Has Models

Surface_mesh_cell_base_3<Gt,Vb>

See Also

SurfaceMeshTriangulation_3

SurfaceMeshComplex_2InTriangulation_3

Surface_mesh_complex_2_in_triangulation_3<Tr>

make_surface_mesh

CGAL::Surface_mesh_complex_2_in_triangulation_3<Tr>

Definition

The class *Surface_mesh_complex_2_in_triangulation_3<Tr>* implements a data structure to store the restricted Delaunay triangulation used by the surface mesher. The restricted Delaunay triangulation is stored as a two dimensional complex embedded in a three dimensional triangulation.

The class *Surface_mesh_complex_2_in_triangulation_3<Tr>* is a model of the concept *SurfaceMeshComplex_2InTriangulation_3* and can be plugged as the template parameter *C2T3* in the function template *make_surface_mesh*.

The template parameter *Tr* has to be instantiated with a model of the concept *SurfaceMeshTriangulation_3*. (Any three dimensional triangulation of CGAL is a model of *Triangulation_3* provided that its vertex and cell base class be models of the concept *SurfaceMeshVertexBase_3* and *SurfaceMeshCellBase_3* respectively.)

```
#include <CGAL/Surface_mesh_complex_2_in_triangulation_3.h>
```

Is Model for the Concepts

SurfaceMeshComplex_2InTriangulation_3

See Also

make_surface_mesh

SurfaceMeshTriangulation_3

SurfaceMeshComplex_2InTriangulation_3

Definition

The concept `SurfaceMeshComplex_2InTriangulation_3` describes a data structure designed to represent a two dimensional pure complex embedded in a three dimensional triangulation.

A *complex* is a set C of faces such that:

- any subface of a face in C is a face of C
- two faces of C are disjoint or share a common subface

The complex is *two dimensional*, if its faces have dimension at most two. It is *pure* if any face in the complex is a subface of some face of maximal dimension. Thus, a two dimensional pure complex is a set of facets together with their edges and vertices. A two dimensional pure complex embedded in a three dimensional triangulation is a subset of the facets of this triangulation, together with their edges and vertices.

The concept `SurfaceMeshComplex_2InTriangulation_3` is particularly suited to handle surface meshes obtained as the restriction to a surface of a three dimensional Delaunay triangulation. A model of this concept is a type to be plugged as first template parameter in the function template `make_surface_mesh`.

The concept `SurfaceMeshComplex_2InTriangulation_3` is a simplification of more general concepts called respectively `PureComplex2InTriangulation3` and `Complex2InTriangulation3`. `PureComplex2InTriangulation3` is designed to represent a pure complex and `Complex2InTriangulation3` is designed to represent any two dimensional complex. Both concepts include member functions to analyse the complex, e.g. find its size, number of connected components, genus etc...

Types

`SurfaceMeshComplex_2InTriangulation_3` provides the following types.

`SurfaceMeshComplex_2InTriangulation_3::Triangulation`

The type of the embedding 3D triangulation. Must be a model of `SurfaceMeshTriangulation_3`.

`typedef Triangulation::Vertex_handle`

`Vertex_handle;`

The type of the embedding triangulation vertex handles.

`typedef Triangulation::Cell_handle`

`Cell_handle;`

The type of the embedding triangulation cell handles.

`typedef Triangulation::Facet`

`Facet;`

The type of the embedding triangulation facets.

`typedef Triangulation::Edge`

`Edge;`

The type of the embedding triangulation edges.

`typedef Triangulation::size_type`

`size_type;`

Size type (an unsigned integral type)

enum Face_status { NOT_IN_COMPLEX, BOUNDARY, REGULAR, SINGULAR};

A type to describe the status of a face (facet, edge, or vertex) with respect to the 2D pure complex. A *NOT_IN_COMPLEX* face does not belong to the 2D complex. Facets can only be *NOT_IN_COMPLEX* or *REGULAR* depending on whether they belong to the 2D complex or not. Edges and vertices can be *NOT_IN_COMPLEX*, *BOUNDARY*, *REGULAR* or *SINGULAR*. An edge in the complex is *BOUNDARY*, *REGULAR*, or *SINGULAR*, if it is incident to respectively 1, 2, or 3 or more facets in the complex. The status of a vertex is determined by the adjacency graph of the facets of the 2D complex incident to that vertex. The vertex of the 2D complex is *BOUNDARY*, if this adjacency graph is a simple path, it is *REGULAR* if the adjacency graph is cyclic, and *SINGULAR* in any other case.

SurfaceMeshComplex_2InTriangulation_3:: Facet_iterator

An iterator type to visit the facets of the 2D complex.

SurfaceMeshComplex_2InTriangulation_3:: Edge_iterator

An iterator type to visit the edges of the 2D complex.

SurfaceMeshComplex_2InTriangulation_3:: Vertex_iterator

An iterator type to visit vertices of the 2D complex.

SurfaceMeshComplex_2InTriangulation_3:: Boundary_edges_iterator

An iterator type to visit the boundary edges of the 2D complex.

Creation

SurfaceMeshComplex_2InTriangulation_3 c2t3(Triangulation& t3);

Builds an empty 2D complex embedded in the triangulation *t3*

template < class FacetSelector>

SurfaceMeshComplex_2InTriangulation_3 c2t3(Triangulation& t3, FacetSelector select);

Builds a 2D complex embedded in the triangulation *t3*, including in the 2D complex the facets of *t3* for which the predicate *select* returns *true*.

The type *FacetSelector* must be a function object with an operator to select facets: *bool operator()(Facet f);*

Member access

Triangulation & *c2t3.triangulation()* Returns the reference to the triangulation.

Modifications

void *c2t3.add_to_complex(Facet f)*
Adds facet *f* to the 2D complex.

void *c2t3.add_to_complex(Cell_handle c, int i)*
Adds facet *(c,i)* to the 2D complex.

void *c2t3.remove_from_complex(Facet f)*
Removes facet *f* from the 2D complex.

void *c2t3.remove_from_complex(Cell_handle c, int i)*
Removes facet *(c,i)* from the 2D complex.

Queries

Queries on the status of individual face with respect to the 2D complex.

size_type *c2t3.number_of_facets()*
Returns the number of facets that belong to the 2D complex.

Face_status *c2t3.face_status(Facet f)*
Returns the status of the facet *f* with respect to the 2D complex.

Face_status *c2t3.face_status(Cell_handle c, int i)*
Returns the status of the facet *(c,i)* with respect to the 2D complex.

Face_status *c2t3.face_status(Edge e)*
Returns the status of edge *e* in the 2D complex.

Face_status *c2t3.face_status(Cell_handle c, int i, int j)*
Returns the status of edge *(c,i,j)* in the 2D complex.

Face_status *c2t3.face_status(Vertex_handle v)*
Returns the status of vertex *v* in the 2D complex.

bool *c2t3.is_in_complex(Facet f)*
Returns *true*, if the facet *f* belongs to the 2D complex.

<i>bool</i>	<i>c2t3.is_in_complex(Cell_handle c, int i)</i>	Returns <i>true</i> , if the facet (c,i) belongs to the 2D complex.
<i>bool</i>	<i>c2t3.is_in_complex(Edge e)</i>	Returns <i>true</i> , if the edge e belongs to the 2D complex.
<i>bool</i>	<i>c2t3.is_in_complex(Cell_handle c, int i, int j)</i>	Returns <i>true</i> , if the edge (c,i,j) belongs to the 2D complex.
<i>bool</i>	<i>c2t3.is_in_complex(Vertex_handle v)</i>	Returns <i>true</i> , if the vertex v belongs to the 2D complex.
<i>bool</i>	<i>c2t3.is_regular_or_boundary_for_vertices(Vertex_handle v)</i>	Returns <i>true</i> if the status of vertex v is <i>REGULAR</i> or <i>BOUNDARY</i> . <i>Precondition:</i> All the edges of the complex incident to v are <i>REGULAR</i> or <i>BOUNDARY</i> .

Traversal of the complex

The data structure provides iterators to visit the facets, edges and vertices of the complex. All those iterators are bidirectional and non mutable.

<i>Facet_iterator</i>	<i>c2t3.facets_begin()</i>	Returns an iterator with value type <i>Facet</i> to visit the facets of the 2D complex.
<i>Facet_iterator</i>	<i>c2t3.facets_end()</i>	Returns the past the end iterator for the above iterator.
<i>Edge_iterator</i>	<i>c2t3.edges_begin()</i>	Returns an iterator with value type <i>Edge</i> to visit the edges of the 2D complex which are not isolated.
<i>Edge_iterator</i>	<i>c2t3.edges_end()</i>	Returns the past the end iteror for the above iterator.
<i>Boundary_edges_iterator</i>	<i>c2t3.boundary_edges_begin()</i>	Returns an iterator with value type <i>Edge</i> to visit the boundary edges of the complex.
<i>Boundary_edges_iterator</i>	<i>c2t3.boundary_edges_end()</i>	Returns the past the end iterator for the above iterator.
<i>Vertex_iterator</i>	<i>c2t3.vertices_begin()</i>	Returns an iterator with value type <i>Vertex_handle</i> to visit the vertices of the 2D complex.
<i>Vertex_iterator</i>	<i>c2t3.vertices_end()</i>	Returns the past the end iterator for the above iterator.

template <class OutputIterator>

OutputIterator *c2t3.incident_facets(Vertex_handle v, OutputIterator facets)*

Copies the *Facets* of the complex incident to *v* to the output iterator *facets*. Returns the resulting output iterator.

Precondition: *c2t3.triangulation().dimension() = 3*, *v* \neq *Vertex_handle()*, *c2t3.triangulation().is_vertex(v)*.

Has Models

Surface_mesh_complex_2_in_triangulation_3<Tr>

See Also

make_surface_mesh.

SurfaceMeshCriteria_3

Definition

The Delaunay refinement process involved in the function template *make_surface_mesh* is guided by a set of refinement criteria. The concept *SurfaceMeshCriteria_3* describes the type which handles those criteria. It corresponds to the requirements for the template parameter *Criteria* of the surface mesher function *make_surface_mesh*<*SurfaceMeshC2T3*,*Surface*,*Criteria*,*Tag*>.

Typically the meshing criteria are a set of elementary criteria, each of which has to be met by the facets of the final mesh. The meshing algorithm eliminates in turn *bad* facets, i.e., facets that do not meet all the criteria.

The size and quality of the final mesh depends on the order according to which bad facets are handled. Therefore, the meshing algorithm needs to be able to quantify the facet qualities and to compare the qualities of different faces. The concept *SurfaceMeshCriteria_3* defines a type *Quality* designed to measure the quality of a mesh facet. Typically this quality is a multicomponent variable. Each component corresponds to one criterion and measures how much the facet deviates from meeting this criterion. Then, the comparison operator on qualities is just a lexicographical comparison. The meshing algorithm handles facets with lowest quality first. The qualities are computed by a function *is_bad*(*Facet* *f*, *Quality*& *q*).

Types

SurfaceMeshCriteria_3::Facet

The type of facets. This type has to match the *Facet* type in the triangulation type used by the mesher function. (This triangulation type is the type *SurfaceMeshC2T3::Triangulation* provided by the the model of *SurfaceMeshComplex_2InTriangulation_3* plugged as first template parameter of *make_surface_mesh*).

SurfaceMeshCriteria_3::Quality

Default constructible, copy constructible, assignable, and less-than comparable type.

Operations

bool *criteria.is_bad*(*Facet* *f*, *Quality*& *q*)

Assigns the quality of the facet *f* to *q*, and returns *true* if *f* does not meet the criteria.

Has Models

Surface_mesh_default_criteria_3<*Tr*>

See Also

make_surface_mesh<*SurfaceMeshC2T3*,*Surface*,*Criteria*,*Tag*>

CGAL::Surface_mesh_default_criteria_3<Tr>

Definition

The class *Surface_mesh_default_criteria_3<Tr>* implements the most commonly used combination of meshing criteria. It involves mainly three criteria which are in order:

- a lower bound on the minimum angle in degrees of the surface mesh facets.
- an upper bound on the radius of surface Delaunay balls. A surface Delaunay ball is a ball circumscribing a facet, centered on the surface and empty of vertices. Such a ball exists for each facet of the current surface mesh. Indeed the current surface mesh is the Delaunay triangulation of the current sampling restricted to the surface which is just the set of facets in the three dimensional Delaunay triangulation of the sampling that have a Delaunay surface ball.
- an upper bound on the center-center distances of the surface mesh facets. The center-center distance of a surface mesh facet is the distance between the facet circumcenter and the center of its surface Delaunay ball.

```
#include <CGAL/Surface_mesh_default_criteria_3.h>
```

Is Model for the Concepts

SurfaceMeshCriteria_3

Types

```
typedef Tr::FT          FT;          The numerical type.
```

Creation

```
Surface_mesh_default_criteria_3<Tr> criteria( FT angle_bound, FT radius_bound, FT distance_bound);
```

Returns a *Surface_mesh_default_criteria_3<Tr>* with *angle_bound*, *radius_bound*, *distance_bound* as bounds for the minimum facet angle in degrees, the radius of the surface Delaunay balls and the center-center distances respectively.

See Also

make_surface_mesh

SurfaceMeshTraits_3

Definition

The concept `SurfaceMeshTraits_3` describes the knowledge that is required on the surface to be meshed. A model of this concept implements an oracle that is able to tell whether a segment (or a ray, or a line) intersects the surface or not and to compute some intersection points if any. The concept `SurfaceMeshTraits_3` also includes a constructor able to provide a small set of initial points on the surface.

Types

<code>SurfaceMeshTraits_3:: Point_3</code>	The type of points. This type is required to match the point type of the three dimensional embedding triangulation <code>C2T3::Triangulation_3</code> .
<code>SurfaceMeshTraits_3:: Segment_3</code>	The type of segments.
<code>SurfaceMeshTraits_3:: Ray_3</code>	The type of rays.
<code>SurfaceMeshTraits_3:: Line_3</code>	The type of lines.
<code>SurfaceMeshTraits_3:: Surface_3</code>	The surface type.

`SurfaceMeshTraits_3:: Intersect_3`

A model of this type provides the operator
`CGAL::object operator()(Surface_3 surface, Type1 type1)`
 to compute the intersection of the surface with an object of type `Type1` which may be `Segment_3`, `Ray_3` or `Line_3`.

`SurfaceMeshTraits_3:: Construct_initial_points`

A model of this type provides the following operators to construct initial points on the surface

```
template <class OutputIteratorPoints>
OutputIteratorPoints operator()(OutputIteratorPoints pts);
which outputs a set of points on the surface.
template <class OutputIteratorPoints>
OutputIteratorPoints operator() (OutputIteratorPoints pts,
int n);
which outputs a set of n points on the surface.
```

Operations

The following functions give access to the construction objects:

```
Intersect_3          traits.intersect_3_object()
Construct_initial_points
                    traits.construct_initial_points_object()
```

Has Models

`Surface_mesh_traits_3<Surface>`

See Also

make_surface_mesh

CGAL::Surface_mesh_traits_generator_3<Surface>

Definition

The class *Surface_mesh_traits_generator_3<Surface>* provides provides a type *Type* , that is a model of the concept *SurfaceMeshTraits_3* for the surface type *Surface*.

The type *Surface* is required to be a model of the concept *Surface_3*, which means that it is copy constructible and assignable. In addition, a *Surface* type is required

- either to provide a nested type *Surface::Surface_mesher_traits_3* that is a model of *SurfaceMeshTraits_3*
- or to be a surface type for which a specialization of the traits generator *Surface_mesh_traits_generator_3<Surface>* exists.

Currently, the library provides partial specializations of the traits generator for implicit surfaces (*Implicit_surface_3<Traits, Function>*) and grey level images (*Gray_level_image_3<FT, Point>*).

```
#include <CGAL/Surface_mesh_traits_generator_3.h>
```

```
Surface_mesh_traits_generator_3<Surface>:: Type
```

A model of the concept *SurfaceMeshTraits_3*.

See Also

SurfaceMeshTraits_3 *make_surface_mesh*

SurfaceMeshTriangulation_3

Definition

The concept `SurfaceMeshTriangulation_3` describes the triangulation type used by the surface mesher `make_surface_mesh` to represent the three dimensional triangulation embedding the surface mesh. Thus, this concept describes the requirements for the triangulation type `SurfaceMeshC2T3::Triangulation` nested in the model of `SurfaceMeshComplex2InTriangulation3` plugged as the template parameter `SurfaceMeshC2T3` of `make_surface_mesh`. It also describes the requirements for the triangulation type plugged in the class `Surface_mesh_complex_2_in_triangulation_3<Tr>`.

Types

`SurfaceMeshTriangulation_3::Point` The point type. It must be `DefaultConstructible`, `CopyConstructible` and `Assignable`.

`Vertices` and `cells` of the triangulation are manipulated via handles, which support the two dereference operators `operator*` and `operator->`.

`SurfaceMeshTriangulation_3::Vertex_handle` Handle to a data representing a *vertex*. *Vertex_handle* must be a model of *Handle* and its *value type* must be model of `TriangulationDataStructure_3::Vertex`.

`SurfaceMeshTriangulation_3::Cell_handle` Handle to a data representing a *cell*. *Cell_handle* must be a model of *Handle* and its *value type* must be model of `TriangulationDataStructure_3::Cell`.

`typedef CGAL::Triple<Cell_handle, int, int>`

`Edge;` The edge type.
`typedef std::pair<Cell_handle, int>`

`Facet;` The facet type.

The following iterators allow one to visit all finite vertices, edges and facets of the triangulation.

`SurfaceMeshTriangulation_3::Finite_vertices_iterator`
 Iterator over finite vertices

`SurfaceMeshTriangulation_3::Finite_edges_iterator`
 Iterator over finite edges

`SurfaceMeshTriangulation_3::Finite_facets_iterator`
 Iterator over finite facets

`SurfaceMeshTriangulation_3::Geom_traits` The geometric traits class. Must be a model of `DelaunayTriangulationTraits_3`.

Creation

<i>SurfaceMeshTriangulation_3</i> <i>t</i> ;	default constructor.
<i>SurfaceMeshTriangulation_3</i> <i>t</i> (<i>tr</i>);	Copy constructor. All vertices and faces are duplicated.

Assignment

<i>SurfaceMeshTriangulation_3</i> &		
<i>t</i> = <i>tr</i>		The triangulation <i>tr</i> is duplicated, and modifying the copy after the duplication does not modify the original. The previous triangulation held by <i>t</i> is deleted.
<i>void</i>	<i>t.clear()</i>	Deletes all finite vertices and all cells of <i>t</i> .

Access Functions

<i>int</i>	<i>t.dimension()</i>	Returns the dimension of the affine hull.
<i>DelaunayTriangulationTraits_3</i>	<i>t.geom_traits()</i>	Returns a const reference to a model of <i>DelaunayTriangulationTraits_3</i> .

Voronoi diagram

<i>Object</i>	<i>t.dual(Facet f)</i>	Returns the dual of facet <i>f</i> , which is in dimension 3: either a segment, if the two cells incident to <i>f</i> are finite, or a ray, if one of them is infinite; in dimension 2: a point.
---------------	-------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Queries

A point *p* is said to be in conflict with a cell *c* in dimension 3 (resp. a facet *f* in dimension 2) iff *t.side_of_sphere(c, p)* (resp. *t.side_of_circle(f, p)*) returns *ON_BOUNDED_SIDE*. The set of cells (resp. facets in dimension 2) which are in conflict with *p* is connected, and it forms a hole.

template <class OutputIteratorBoundaryFacets, class OutputIteratorCells, class OutputIteratorInternalFacets>

Triple<OutputIteratorBoundaryFacets, OutputIteratorCells, OutputIteratorInternalFacets>

t.find_conflicts(Point p,
Cell_handle c,
OutputIteratorBoundaryFacets bfit,
OutputIteratorCells cit,

OutputIteratorInternalFacets ifit)

Computes the conflict hole induced by p . The starting cell (resp. facet) c must be in conflict. Then this function returns respectively in the output iterators:

- *cit*: the cells (resp. facets) in conflict.
- *bfit*: the facets (resp. edges) on the boundary, that is, the facets (resp. edges) (t, i) where the cell (resp. facet) t is in conflict, but $t \rightarrow neighbor(i)$ is not.
- *ifit*: the facets (resp. edges) inside the hole, that is, delimiting two cells (resp facets) in conflict.

Returns the *Triple* composed of the resulting output iterators.

The following iterators allow the user to visit facets, edges and vertices of the triangulation.

Finite_vertices_iterator

t.finite_vertices_begin()

Starts at an arbitrary finite vertex. Then ++ and -- will iterate over finite vertices. Returns *finite_vertices_end()* when *t.number_of_vertices()* = 0.

Finite_vertices_iterator

t.finite_vertices_end()

Past-the-end iterator

Finite_edges_iterator

t.finite_edges_begin()

Starts at an arbitrary finite edge. Then ++ and -- will iterate over finite edges. Returns *finite_edges_end()* when *t.dimension()* < 1.

Finite_edges_iterator

t.finite_edges_end()

Past-the-end iterator

Finite_facets_iterator

t.finite_facets_begin()

Starts at an arbitrary finite facet. Then ++ and -- will iterate over finite facets. Returns *finite_facets_end()* when *t.dimension()* < 2.

Finite_facets_iterator

t.finite_facets_end()

Past-the-end iterator

template <class OutputIterator>

OutputIterator *t*.incident_cells(Vertex_handle v, OutputIterator cells)

Copies the *Cell_handles* of all cells incident to v to the output iterator *cells*. If *t.dimension()* < 3, then do nothing. Returns the resulting output iterator.

Precondition: $v \neq \text{Vertex_handle}()$, $t.is_vertex(v)$.

template <class OutputIterator>

OutputIterator *t*.incident_cells(Vertex_handle v, OutputIterator cells)

Copies the *Cell_handles* of all cells incident to v to the output iterator *cells*. If *t.dimension()* < 3, then do nothing. Returns the resulting output iterator.

<i>bool</i>	<i>t.is_vertex(Point p, Vertex_handle & v)</i>	Tests whether <i>p</i> is a vertex of <i>t</i> by locating <i>p</i> in the triangulation. If <i>p</i> is found, the associated vertex <i>v</i> is given.
<i>bool</i>	<i>t.is_edge(Vertex_handle u, Vertex_handle v, Cell_handle & c, int & i, int & j)</i>	Tests whether <i>(u,v)</i> is an edge of <i>t</i> . If the edge is found, it gives a cell <i>c</i> having this edge and the indices <i>i</i> and <i>j</i> of the vertices <i>u</i> and <i>v</i> in <i>c</i> , in this order. <i>Precondition: u and v are vertices of t.</i>
<i>bool</i>	<i>t.is_infinite(const Vertex_handle v)</i>	<i>true</i> , iff vertex <i>v</i> is the infinite vertex.
<i>bool</i>	<i>t.is_infinite(const Cell_handle c)</i>	<i>true</i> , iff <i>c</i> is incident to the infinite vertex. <i>Precondition: t.dimension() = 3.</i>
<i>Facet</i>	<i>t.mirror_facet(Facet f)</i>	Returns the same facet viewed from the other adjacent cell.
<i>int</i>	<i>t.vertex_triple_index(const int i, const int j)</i>	Return the indexes of the <i>j</i> th vertex of the facet of a cell opposite to vertex <i>i</i> .

Point location

<i>Cell_handle</i>	<i>t.locate(Point query, Cell_handle start = Cell_handle())</i>	<p>If the point <i>query</i> lies inside the convex hull of the points, the cell that contains the query in its interior is returned. If <i>query</i> lies on a facet, an edge or on a vertex, one of the cells having <i>query</i> on its boundary is returned.</p> <p>If the point <i>query</i> lies outside the convex hull of the points, an infinite cell with vertices $\{p, q, r, \infty\}$ is returned such that the tetrahedron $(p, q, r, query)$ is positively oriented (the rest of the triangulation lies on the other side of facet (p, q, r)).</p> <p>Note that <i>locate</i> works even in degenerate dimensions: in dimension 2 (resp. 1, 0) the <i>Cell_handle</i> returned is the one that represents the facet (resp. edge, vertex) containing the query point.</p> <p>The optional argument <i>start</i> is used as a starting place for the search.</p>
--------------------	------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

```

Cell_handle      t.locate( Point query,
                          Locate_type & lt,
                          int & li,
                          int & lj,
                          Cell_handle start = Cell_handle())

```

If *query* lies inside the affine hull of the points, the *k*-face (finite or infinite) that contains *query* in its interior is returned, by means of the cell returned together with *lt*, which is set to the locate type of the query (*VERTEX*, *EDGE*, *FACET*, *CELL*, or *OUTSIDE_CONVEX_HULL* if the cell is infinite and *query* lies strictly in it) and two indices *li* and *lj* that specify the *k*-face of the cell containing *query*.

If the *k*-face is a cell, *li* and *lj* have no meaning; if it is a facet (resp. vertex), *li* gives the index of the facet (resp. vertex) and *lj* has no meaning; if it is an edge, *li* and *lj* give the indices of its vertices.

If the point *query* lies outside the affine hull of the points, which can happen in case of degenerate dimensions, *lt* is set to *OUTSIDE_AFFINE_HULL*, and the cell returned has no meaning. As a particular case, if there is no finite vertex yet in the triangulation, *lt* is set to *OUTSIDE_AFFINE_HULL* and *locate* returns the default constructed handle.

The optional argument *start* is used as a starting place for the search.

```

template <class CellIt>
Vertex_handle      t.insert_in_hole( Point p, CellIt cell_begin, CellIt cell_end, Cell_handle begin, int i)

```

Creates a new vertex by starring a hole. It takes an iterator range [*cell_begin*; *cell_end*] of *Cell_handles* which specifies a hole: a set of connected cells (resp. facets in dimension 2) which is star-shaped wrt *p*. (*begin*, *i*) is a facet (resp. an edge) on the boundary of the hole, that is, *begin* belongs to the set of cells (resp. facets) previously described, and *begin*->*neighbor(i)* does not. Then this function deletes all the cells (resp. facets) describing the hole, creates a new vertex *v*, and for each facet (resp. edge) on the boundary of the hole, creates a new cell (resp. facet) with *v* as vertex. Then *v*->*set_point(p)* is called and *v* is returned.

Precondition: *t.dimension()* ≥ 2 , the set of cells (resp. facets in dimension 2) is connected, its boundary is connected, and *p* lies inside the hole, which is star-shaped wrt *p*.

Has Models

Any 3D Delaunay triangulation class of CGAL

See Also

```

Triangulation_3<TriangulationTraits_3,TriangulationDataStructure_3>
Delaunay_triangulation_3<DelaunayTriangulationTraits_3,TriangulationDataStructure_3>

```

SurfaceMeshComplex2InTriangulation3
Surface_mesh_complex_2_in_triangulation_3<Tr>
make_surface_mesh

CGAL::Surface_mesh_vertex_base_3<Gt,Vb>

Definition

The class *Surface_mesh_vertex_base_3<Gt,Vb>* is a model of the concept *SurfaceMeshVertexBase_3*. It is designed to serve as vertex base class in a triangulation class *Tr* plugged in a *Surface_mesh_complex_2_in_triangulation_3<Tr>* class.

The first template parameter is the geometric traits class.

The second template parameter is a base class. It has to be a model of the concept *TriangulationVertexBase_3* and defaults to *Triangulation_vertex_base_3 <GT>*.

```
#include <CGAL/Surface_mesh_vertex_base_3.h>
```

Is Model for the Concepts

SurfaceMeshVertexBase_3

Inherits From

Vb

See Also

SurfaceMeshComplex_2InTriangulation_3
Surface_mesh_complex_2_in_triangulation_3<Tr>
SurfaceMeshTriangulation_3
make_surface_mesh

SurfaceMeshVertexBase_3

Definition

The concept `SurfaceMeshVertexBase_3` describes the vertex base type of the three dimensional triangulation used to embed the surface mesh.

More precisely, the first template parameter *SurfaceMeshC2T3* of the surface mesher *make_surface_mesh* is a model of the concept *SurfaceMeshComplex_2InTriangulation_3* which describes a data structure to store a pure two dimensional complex embedded in a three dimensional triangulation. In particular, the type *SurfaceMeshC2T3* is required to provide a three dimensional triangulation type *SurfaceMeshC2T3::Triangulation_3*. The concept `SurfaceMeshVertexBase_3` describes the vertex base type required in this triangulation type.

Generalizes

TriangulationVertexBase_3

The surface mesher algorithm issues frequent queries about the status of the vertices with respect to the two dimensional complex that represents the current surface approximation. The class `SurfaceMeshVertexBase_3` offers a caching mechanism to answer more efficiently these queries. The caching mechanism includes two cached integers, which, when they are valid, store respectively the number of complex facets incident to the vertex and the number of connected components of the adjacency graph of those facets.

Creation

Operations

<i>bool</i>	<i>vb.is_c2t3_cache_valid()</i>	Returns <i>true</i> if the cache is valid.
<i>void</i>	<i>vb.invalidate_c2t3_cache()</i>	Invalidates the cash.
<i>int</i>	<i>vb.cached_number_of_incident_facets()</i>	Returns the cached number of facets of the complex incident to the vertex.
<i>int</i>	<i>vb.cached_number_of_components()</i>	This method concerns the adjacency graph of the facets of the complex incident to the vertex and returns a cached value for the number of connected components this graph.

Has Models

Surface_mesh_vertex_base_3<Gt,Vb>

See Also

SurfaceMesherComplex_2InTriangulation_3

Surface_mesh_complex_2_in_triangulation_3<Tr>.

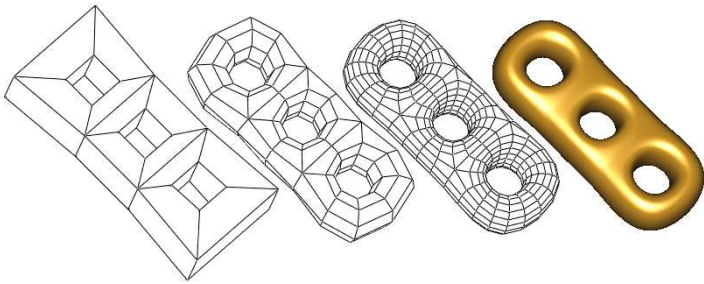
Chapter 31

3D Surface Subdivision Methods

Le-Jeng Andy Shiue

Contents

31.1 Introduction	1867
31.2 Subdivision Method	1868
31.3 A Quick Example: Catmull-Clark Subdivision	1869
31.4 Catmull-Clark Subdivision	1870
31.5 Refinement Host	1873
31.6 Geometry Policy	1874
31.7 The Four Subdivision Methods	1876
31.8 Other Subdivision Methods	1877



31.1 Introduction

Subdivision methods are simple yet powerful ways to generate smooth surfaces from arbitrary polyhedral meshes. Unlike spline-based surfaces (e.g NURBS) or other numeric-based modeling techniques, users of subdivision methods do not need the mathematical knowledge of the subdivision methods. The natural intuition of the geometry suffices to control the subdivision methods.

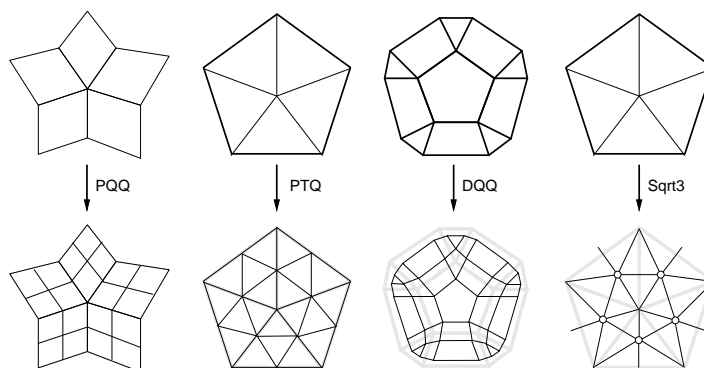
Subdivision_method_3, designed to work on the class *Polyhedron_3*, aims to be easy to use and to extend. *Subdivision_method_3* is not a class, but a namespace which contains four popular subdivision methods and their refinement functions. These include Catmull-Clark, Loop, Doo-Sabin and $\sqrt{3}$ subdivisions. Variations of these methods can be easily extended by substituting the geometry computation of the refinement host.

31.2 Subdivision Method

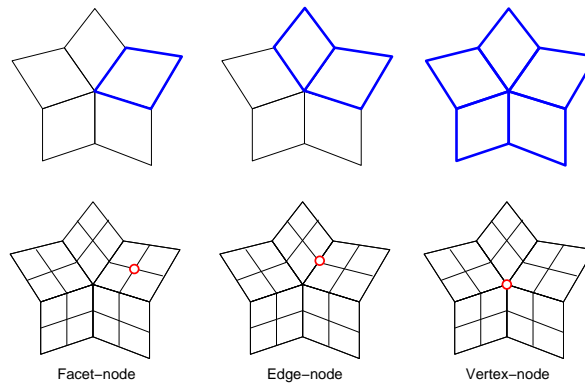
In this chapter, we explain some fundamentals of subdivision methods. We focus only on the topics that help you to understand the design of the package. [WW02] has details on subdivision methods. Some terminology introduced in this section will be used again in later sections. If you are only interested in using a specific subdivision method, Section 31.3 gives a quick tutorial on Catmull-Clark subdivision.

A subdivision method recursively refines a coarse mesh and generates an ever closer approximation to a smooth surface. The coarse mesh can have arbitrary shape, but it has to be a 2-manifold. In a 2-manifold, every interior point has a neighborhood homeomorphic to a 2D disk. Subdivision methods on non-manifolds have been developed, but are not considered in *Subdivision_method_3*. The chapter teaser shows the steps of Catmull-Clark subdivision on a CAD model. The coarse mesh is repeatedly refined by a quadrissection pattern, and new points are generated to approximate a smooth surface.

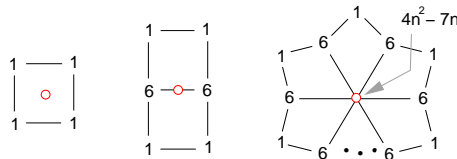
Many refinement patterns are used in practice. *Subdivision_method_3* supports the four most popular patterns, and each of them is used by Catmull-Clark[CC78], Loop, Doo-Sabin and $\sqrt{3}$ subdivision (left to right in the figure). We name these patterns by their topological characteristics instead of the associated subdivision methods. PQQ indicates the *Primal Quadrateral Quadrissection*. PTQ indicates the *Primal Triangle Quadrissection*. DQQ indicates the *Dual Quadrateral Quadrissection*. $\sqrt{3}$ indicates the converging speed of the triangulation toward the subdivision surface.



The figure demonstrates these four refinement patterns on the 1-disk of a valence-5 vertex/facet. Refined meshes are shown below the source meshes. Points on the refined mesh are generated by averaging neighbor points on the source mesh. A graph, called *stencil*, determines the source neighborhood whose points contribute to the position of a refined point. A refinement pattern usually defines more than one stencil. For example, the PQQ refinement has a vertex-node stencil, which defines the 1-ring of an input vertex; an edge-node stencil, which defines the 1-ring of an input edge; and a facet-node stencil, which defines an input facet. The stencils of the PQQ refinement are shown in the following figure. The blue neighborhoods in the top row indicate the corresponding stencils of the refined nodes in red.



Stencils with weights are called *geometry masks*. A subdivision method defines a geometry mask for each stencil, and generates new points by averaging source points weighted by the mask. Geometry masks are carefully chosen to meet requirements of certain surface smoothness and shape quality. The geometry masks of Catmull-Clark subdivision are shown below.



The weights shown here are unnormalized, and n is the valence of the vertex. The generated point, in red, is computed by a summation of the weighted points. For example, a Catmull-Clark facet-node is computed by the summation of $1/4$ of each point on its stencil.

A stencil can have an unlimited number of geometry masks. For example, a facet-node of PQQ refinement may be computed by the summation of $1/5$ of each stencil node instead of $1/4$. Although it is legal in *Subdivision_method_3* to have any kind of geometry mask, the result surfaces may be odd, not smooth, or not even exist. [WW02] explains the details on designing masks for a quality subdivision surface.

31.3 A Quick Example: Catmull-Clark Subdivision

Assuming you are familiar with *Polyhedron_3*, you can integrate *Subdivision_method_3* into your program without much effort.

```
// file: examples/Subdivision_method_3/CatmullClark_subdivision.C

#include <CGAL/Cartesian.h>
#include <CGAL/Subdivision_method_3.h>
#include <iostream>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>

typedef CGAL::Cartesian<double>          Kernel;
typedef CGAL::Polyhedron_3<Kernel>      Polyhedron;
```

```

using namespace std;
using namespace CGAL;

int main(int argc, char** argv) {
    if (argc != 2) {
        cout << "Usage: CatmullClark_subdivision d < filename" << endl;
        cout << "          d: the depth of the subdivision (0 < d < 10)" << endl;
        cout << "          filename: the input mesh (.off)" << endl;
        return 0;
    }

    int d = argv[1][0] - '0';

    Polyhedron P;
    cin >> P; // read the .off

    Subdivision_method_3::CatmullClark_subdivision(P,d);

    cout << P; // write the .off

    return 0;
}

```

This example demonstrates the use of the Catmull-Clark subdivision method on a *Polyhedron_3*. The polyhedron is restricted in the Cartesian space, where most subdivision applications are designed to work. There is only one line deserving a detailed explanation:

```
Subdivision_method_3::CatmullClark_subdivision(P,d);
```

Subdivision_method_3 specifies the namespace of our subdivision functions. *CatmullClark_subdivision(P,d)* computes the Catmull-Clark subdivision surface of the polyhedron *P* after *d* iterations of the refinements. The polyhedron *P* is passed by reference, and is modified (i.e. subdivided) by the subdivision function.

This example shows how to subdivide a simple *Polyhedron_3* with *Subdivision_method_3*. An application-defined polyhedron might use a specialized kernel and/or a specialized internal container. There are two major restrictions on the application-defined polyhedron to work with *Subdivision_method_3*.

- *Point_3* is type-defined by the kernel. Without *Point_3* and the associated operations being defined, *Subdivision_method_3* can not know how to compute and store the new vertex points.
- The primitives (such as vertices, halfedges and facets) in the internal container are sequentially ordered (e.g. *std::vector* and *std::list*). This implies that the iterators traverse the primitives in the order of their creations/insertions.

Section 31.5 gives detailed explanations on those two restrictions.

31.4 Catmull-Clark Subdivision

Subdivision_method_3 is designed to allow customization of the subdivision methods. This section explains the implementation of the Catmull-Clark subdivision function in *Subdivision_method_3*. The implementation demonstrates the customization of the PQQ refinement to Catmull-Clark subdivision.

When a subdivision method is developed, a refinement pattern is chosen, and then a set of geometry masks are developed to position the new points. There are three key components to implement a subdivision method:

- a mesh data structure that can represent arbitrary 2-manifolds,
- a process that refines the mesh data structure,
- and the geometry masks that compute the new points.

E. Catmull and J. Clark picked the PQQ refinement for their subdivision method, and developed a set of geometry masks to generate (or more precisely, to approximate) the B-spline surface from the control mesh. *Subdivision_method_3* provides a function that glues all three components of the Catmull-Clark subdivision method.

```
template <class Polyhedron_3, template <typename> class Mask>
void PQQ(Polyhedron_3& p, Mask<Polyhedron_3> mask, int depth)
```

Polyhedron_3 is a generic mesh data structure for arbitrary 2-manifolds. *PQQ()*, which refines the control mesh *p*, is a *refinement host* that uses a policy class *Mask<Polyhedron_3>* as part of its geometry computation. During the refinement, *PQQ()* computes and assigns new points by cooperating with the *mask*. To implement Catmull-Clark subdivision, *Mask*, the *geometry policy*, has to realize the geometry masks of Catmull-Clark subdivision. *depth* specifies the iterations of the refinement on the control mesh.

To implement the geometry masks, we need to know how a refinement host communicates with its geometry masks. The PQQ refinement defines three stencils, and hence three geometry masks are required for Catmull-Clark subdivision. The following class defines the interfaces of the stencils for the PQQ refinement.

```
template <class Polyhedron_3>
class PQQ_stencil_3 {
    void facet_node(Facet_handle facet, Point_3& pt);
    void edge_node(Halfedge_handle edge, Point_3& pt);
    void vertex_node(Vertex_handle vertex, Point_3& pt);
};
```

Each class function in *PQQ_stencil_3* computes a new point based on the neighborhood of the primitive handle, and assigns the new point to *Point_3& pt*.

We realize each class function with the geometry masks of Catmull-Clark subdivision.

```
template <class Polyhedron_3>
class CatmullClark_mask_3 {
    void facet_node(Facet_handle facet, Point_3& pt) {
        Halfedge_around_facet_circulator hcir = facet->facet_begin();
        int n = 0;
        Point_3 p(0,0,0);
        do {
            p = p + (hcir->vertex()->point() - ORIGIN);
            ++n;
        } while (++hcir != facet->facet_begin());
        pt = ORIGIN + (p - ORIGIN)/FT(n);
    }
};
```

```

void edge_node(Halfedge_handle edge, Point_3& pt) {
    Point_3 p1 = edge->vertex()->point();
    Point_3 p2 = edge->opposite()->vertex()->point();
    Point_3 f1, f2;
    facet_node(edge->facet(), f1);
    facet_node(edge->opposite()->facet(), f2);
    pt = Point_3((p1[0]+p2[0]+f1[0]+f2[0])/4,
                  (p1[1]+p2[1]+f1[1]+f2[1])/4,
                  (p1[2]+p2[2]+f1[2]+f2[2])/4 );
}

void vertex_node(Vertex_handle vertex, Point_3& pt) {
    Halfedge_around_vertex_circulator vcir = vertex->vertex_begin();
    int n = circulator_size(vcir);

    FT Q[] = {0.0, 0.0, 0.0}, R[] = {0.0, 0.0, 0.0};
    Point_3& S = vertex->point();

    Point_3 q;
    for (int i = 0; i < n; i++, ++vcir) {
        Point_3& p2 = vcir->opposite()->vertex()->point();
        R[0] += (S[0]+p2[0])/2;
        R[1] += (S[1]+p2[1])/2;
        R[2] += (S[2]+p2[2])/2;
        facet_node(vcir->facet(), q);
        Q[0] += q[0];
        Q[1] += q[1];
        Q[2] += q[2];
    }
    R[0] /= n;    R[1] /= n;    R[2] /= n;
    Q[0] /= n;    Q[1] /= n;    Q[2] /= n;

    pt = Point_3((Q[0] + 2*R[0] + S[0]*(n-3))/n,
                  (Q[1] + 2*R[1] + S[1]*(n-3))/n,
                  (Q[2] + 2*R[2] + S[2]*(n-3))/n );
}
};

```

This example shows the default implementation of Catmull-Clark masks in *Subdivision_method_3*. This default implementation assumes the *types* (such as *Point_3* and *Facet_handle*) are defined within *Polyhedron_3*. *CatmullClark_mask_3* is designed to work on a *Polyhedron_3* with the *Cartesian* kernel. You may need to rewrite the geometry computation to match the kernel geometry of your application.

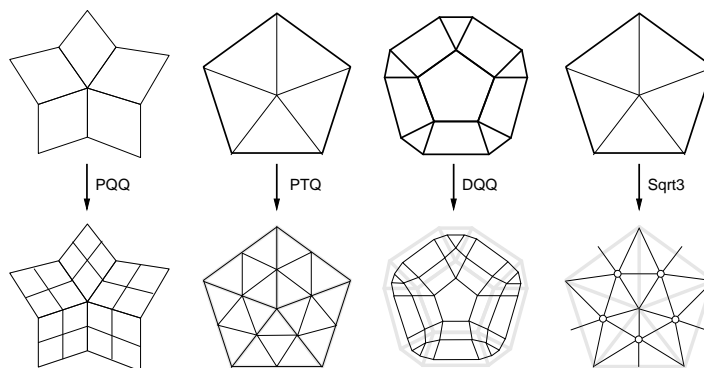
To invoke the Catmull-Clark subdivision method, we call *PQQ()* with the Catmull-Clark masks we just defined.

```
PQQ(p, CatmullClark_mask_3<Polyhedron_3>(), depth);
```

Loop, Doo-Sabin and $\sqrt{3}$ subdivisions are implemented in the similar process: pick a refinement host and implement the geometry policy. The key of developing your own subdivision method is implementing the right combination of the refinement host and the geometry policy. It is explained in the next two sections.

31.5 Refinement Host

A refinement host is a template function of a polyhedron class and a geometry mask class. It refines the input polyhedron, and computes new points through the geometry masks. *Subdivision_method_3* supports four refinement hosts: primal quadrilateral quadrisection (PQQ), primal triangle quadrisection (PTQ), dual quadrilateral quadrisection (DQQ) and $\sqrt{3}$ triangulation. Respectively, they are used by Catmull-Clark, Loop, Doo-Sabin and $\sqrt{3}$ subdivision.



```
namespace Subdivision_method_3 {
    template <class Polyhedron_3, template <typename> class Mask>
    void PQQ(Polyhedron_3& p, Mask<Polyhedron_3> mask, int step);

    template <class Polyhedron_3, template <typename> class Mask>
    void PTQ(Polyhedron_3& p, Mask<Polyhedron_3> mask, int step);

    template <class Polyhedron_3, template <typename> class Mask>
    void DQQ(Polyhedron_3& p, Mask<Polyhedron_3> mask, int step);

    template <class Polyhedron_3, template <typename> class Mask>
    void Sqrt3(Polyhedron_3& p, Mask<Polyhedron_3> mask, int step)
}
```

The polyhedron class is a specialization of *Polyhedron_3*, and the mask is a policy class realizing the geometry masks of the subdivision method.

A refinement host refines the input polyhedron, maintains the stencils (i.e., the mapping between the control mesh and the refined mesh), and calls the geometry masks to compute the new points. In *Subdivision_method_3*, refinements are implemented as a sequence of connectivity operations (mainly Euler operations). The order of the connectivity operations plays a key role when maintaining stencils. By matching the order of the source submeshes to the refined vertices, no flag in the primitives is required to register the stencils. It avoids the data dependency of the refinement host on the polyhedron class. To make the ordering trick work, the polyhedron class must have a sequential container, such as a vector or a linked-list, as the internal storage. A sequential container guarantees that the iterators of the polyhedron always traverse the primitives in the order of their insertions. Non-sequential structures such as trees or maps do not provide the required ordering, and hence can not be used with *Subdivision_method_3*.

Although *Subdivision_method_3* does not require flags to support the refinements and the stencils, it still needs to know how to compute and store the geometry data (i.e. the points). *Subdivision_method_3* expects that the typename *Point_3* is defined in the geometry kernel of the polyhedron (i.e. the *Polyhedron_3::Traits::Kernel*).

A point of the type *Point_3* is returned by the geometry policy and is then assigned to the new vertex. The geometry policy is explained in next section.

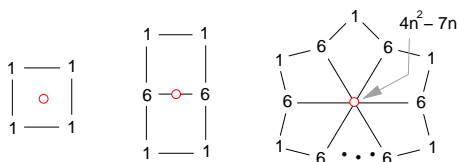
Refinement hosts *PQQ* and *DQQ* work on a general polyhedron, and *PTQ* and *Sqrt3* work on a triangulated polyhedron. The result of *PTQ* and *Sqrt3* on a non-triangulated polyhedron is undefined. *Subdivision_method_3* does not verify the precondition of the mesh characteristics before the refinement.

For details of the refinement implementation, interested users should refer to [SP05].

31.6 Geometry Policy

A geometry policy defines a set of geometry masks. Each geometry mask is realized as a member function that computes new points of the subdivision surface.

Each geometry mask receives a primitive handle (e.g. *Halfedge_handle*) of the control mesh, and returns a *Point_3* to the subdivided vertex. The function collects the vertex neighbors of the primitive handle (i.e. nodes on the stencil), and computes the new point based on the neighbors and the mask (i.e. the stencil weights).



This figure shows the geometry masks of Catmull-Clark subdivision. The weights shown here are unnormalized, and n is the valence of the vertex. The new points are computed by the summation of the weighted points on their stencils. Following codes show an implementation of the geometry mask of the facet-node. The complete listing of a Catmull-Clark geometry policy is in the Section 31.4.

```
template <class Polyhedron_3>
class CatmullClark_mask_3 {
    void facet_node(Facet_handle facet, Point_3& pt) {
        Halfedge_around_facet_circulator hcir = facet->facet_begin();
        int n = 0;
        Point_3 p(0,0,0);
        do {
            p = p + (hcir->vertex()->point() - ORIGIN);
            ++n;
        } while (++hcir != facet->facet_begin());
        pt = ORIGIN + (p - ORIGIN)/FT(n);
    }
}
```

In this example, the computation is based on the assumption that the *Point_3* is the *CGAL::Point_3*. It is an assumption, but not a restriction. You are allowed to use any point class as long as it is defined as the *Point_3* in your polyhedron. You may need to modify the geometry policy to support the computation and the assignment of the specialized point. This extension is not unusual in graphics applications. For example, you might want to subdivide the texture coordinates for your subdivision surface.

The refinement host of Catmull-Clark subdivision requires three geometry masks for polyhedrons without open boundaries: a vertex-node mask, an edge-node mask, and a facet-node mask. To support polyhedrons with boundaries, a border-node mask is also required. The border-node mask for Catmull-Clark subdivision is listed below, where *ept* returns the new point splitting *edge* and *vpt* returns the new point on the vertex pointed by *edge*.

```
void border_node(Halfedge_handle edge, Point_3& ept, Point_3& vpt) {
    Point_3& ep1 = edge->vertex()->point();
    Point_3& ep2 = edge->opposite()->vertex()->point();
    ept = Point_3((ep1[0]+ep2[0])/2, (ep1[1]+ep2[1])/2, (ep1[2]+ep2[2])/2);

    Halfedge_around_vertex_circulator vcir = edge->vertex_begin();
    Point_3& vp1 = vcir->opposite()->vertex()->point();
    Point_3& vp0 = vcir->vertex()->point();
    Point_3& vp_1 = (--vcir)->opposite()->vertex()->point();
    vpt = Point_3((vp_1[0] + 6*vp0[0] + vp1[0])/8,
                  (vp_1[1] + 6*vp0[1] + vp1[1])/8,
                  (vp_1[2] + 6*vp0[2] + vp1[2])/8 );
}
```

The mask interfaces of all four refinement hosts are listed below. *DQQ_stencil_3* and *Sqrt3_stencil_3* do not have the border-node stencil because the refinement hosts of DQQ and $\sqrt{3}$ refinements do not support global boundaries in the current release. This might be changed in the future releases.

```
template <class Polyhedron_3>
class PQQ_stencil_3 {
    void facet_node(Facet_handle, Point_3&);
    void edge_node(Halfedge_handle, Point_3&);
    void vertex_node(Vertex_handle, Point_3&);

    void border_node(Halfedge_handle, Point_3&, Point_3&);
};
```

```
template <class Polyhedron_3>
class PTQ_stencil_3 {
    void edge_node(Halfedge_handle, Point_3&);
    void vertex_node(Vertex_handle, Point_3&);

    void border_node(Halfedge_handle, Point_3&, Point_3&);
};
```

```
template <class Polyhedron_3>
class DQQ_stencil_3 {
public:
    void corner_node(Halfedge_handle edge, Point_3& pt);
};
```

```
template <class Polyhedron_3>
class Sqrt3_stencil_3 {
public:
    void vertex_node(Vertex_handle vertex, Point_3& pt);
};
```

The source codes of *CatmullClark_mask_3*, *Loop_mask_3*, *DooSabin_mask_3*, and *Sqrt3_mask_3* are the best sources of learning these stencil interfaces.

31.7 The Four Subdivision Methods

Subdivision_method_3 supports Catmull-Clark, Loop, Doo-Sabin and $\sqrt{3}$ subdivisions by specializing their respective refinement hosts. They are designed to work on a *Polyhedron_3*. If your application uses a polyhedron with a specialized geometry kernel, you need to specialize the refinement host with a geometry policy based on that kernel.

```
namespace Subdivision_method_3 {
    template <class Polyhedron_3>
    void CatmullClark_subdivision(Polyhedron_3& p, int step = 1) {
        PQQ(p, CatmullClark_mask_3<Polyhedron_3>(), step);
    }

    template <class Polyhedron_3>
    void Loop_subdivision(Polyhedron_3& p, int step = 1) {
        PTQ(p, Loop_mask_3<Polyhedron_3>(), step);
    }

    template <class Polyhedron_3>
    void DooSabin_subdivision(Polyhedron_3& p, int step = 1) {
        DQQ(p, DooSabin_mask_3<Polyhedron_3>(), step);
    }

    template <class Polyhedron_3>
    void Sqrt3_subdivision(Polyhedron_3& p, int step = 1) {
        Sqrt3(p, Sqrt3_mask_3<Polyhedron_3>(), step);
    }
}
```

The following example demonstrates the use of Doo-Sabin subdivision on a polyhedral mesh.

```
// file: examples/Subdivision_method_3/DooSabin_subdivision.C

#include <CGAL/Cartesian.h>
#include <CGAL/Subdivision_method_3.h>

#include <iostream>

#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>

typedef CGAL::Cartesian<double>          Kernel;
typedef CGAL::Polyhedron_3<Kernel>      Polyhedron;

using namespace std;
using namespace CGAL;
```

```

int main(int argc, char **argv) {
    if (argc != 2) {
        cout << "Usage: DooSabin_subdivision d < filename" << endl;
        cout << "          d: the depth of the subdivision (0 < d < 10)" << endl;
        cout << "          filename: the input mesh (.off)" << endl;
        return 0;
    }

    int d = argv[1][0] - '0';

    Polyhedron P;
    cin >> P; // read the .off

    Subdivision_method_3::DooSabin_subdivision(P,d);

    cout << P; // write the .off

    return 0;
}

```

31.8 Other Subdivision Methods

Subdivision_method_3 supports four practical subdivision methods on a Cartesian *Polyhedron_3*. More subdivision methods can be supported through the specialization of refinement hosts with custom geometry masks. The following example develops a subdivision method generating an improved Loop subdivision surface.

```

// file: examples/Subdivision_method_3/Customized_subdivision.C

#include <CGAL/Cartesian.h>
#include <CGAL/Subdivision_method_3.h>

#include <cstdio>
#include <iostream>

#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>

typedef CGAL::Cartesian<double>          Kernel;
typedef CGAL::Polyhedron_3<Kernel>      Polyhedron;

using namespace std;
using namespace CGAL;

// =====
template <class Poly>
class WLoop_mask_3 {
    typedef Poly                                Polyhedron;

    typedef typename Polyhedron::Vertex_iterator    Vertex_iterator;
    typedef typename Polyhedron::Halfedge_iterator  Halfedge_iterator;
    typedef typename Polyhedron::Facet_iterator     Facet_iterator;

```

```

typedef typename Polyhedron::Halfedge_around_facet_circulator
                        Halfedge_around_facet_circulator;
typedef typename Polyhedron::Halfedge_around_vertex_circulator
                        Halfedge_around_vertex_circulator;

typedef typename Polyhedron::Traits Traits;
typedef typename Traits::Kernel Kernel;

typedef typename Kernel::FT FT;
typedef typename Kernel::Point_3 Point;
typedef typename Kernel::Vector_3 Vector;

public:
void edge_node(Halfedge_iterator eitr, Point& pt) {
    Point& p1 = eitr->vertex()->point();
    Point& p2 = eitr->opposite()->vertex()->point();
    Point& f1 = eitr->next()->vertex()->point();
    Point& f2 = eitr->opposite()->next()->vertex()->point();

    pt = Point((3*(p1[0]+p2[0])+f1[0]+f2[0])/8,
               (3*(p1[1]+p2[1])+f1[1]+f2[1])/8,
               (3*(p1[2]+p2[2])+f1[2]+f2[2])/8 );
}

void vertex_node(Vertex_iterator vitr, Point& pt) {
    float R[] = {0.0, 0.0, 0.0};
    Point& S = vitr->point();

    Halfedge_around_vertex_circulator vcir = vitr->vertex_begin();
    int n = circulator_size(vcir);
    for (int i = 0; i < n; i++, ++vcir) {
        Point& p = vcir->opposite()->vertex()->point();
        R[0] += p[0]; R[1] += p[1]; R[2] += p[2];
    }
    if (n == 6) {
        pt = Point((10*S[0]+R[0])/16, (10*S[1]+R[1])/16, (10*S[2]+R[2])/16);
    } else if (n == 3) {
        double B = (5.0/8.0 - std::sqrt(3+2*std::cos(6.283/n))/64.0)/n;
        double A = 1-n*B;
        pt = Point((A*S[0]+B*R[0]), (A*S[1]+B*R[1]), (A*S[2]+B*R[2]));
    } else {
        double B = 3.0/8.0/n;
        double A = 1-n*B;
        pt = Point((A*S[0]+B*R[0]), (A*S[1]+B*R[1]), (A*S[2]+B*R[2]));
    }
}

void border_node(Halfedge_iterator eitr, Point& ept, Point& vpt) {
    Point& ep1 = eitr->vertex()->point();
    Point& ep2 = eitr->opposite()->vertex()->point();
    ept = Point((ep1[0]+ep2[0])/2, (ep1[1]+ep2[1])/2, (ep1[2]+ep2[2])/2);

    Halfedge_around_vertex_circulator vcir = eitr->vertex_begin();
    Point& vp1 = vcir->opposite()->vertex()->point();
    Point& vp0 = vcir->vertex()->point();

```

```

    Point& vp_1 = (--vcir)->opposite()->vertex()->point();
    vpt = Point((vp_1[0] + 6*vp0[0] + vp1[0])/8,
                (vp_1[1] + 6*vp0[1] + vp1[1])/8,
                (vp_1[2] + 6*vp0[2] + vp1[2])/8 );
}
};

int main(int argc, char **argv) {
    if (argc != 2) {
        cout << "Usage: Customized_subdivision d < filename" << endl;
        cout << "          d: the depth of the subdivision (0 < d < 10)" << endl;
        cout << "          filename: the input mesh (.off)" << endl;
        return 0;
    }

    int d = argv[1][0] - '0';

    Polyhedron P;
    cin >> P; // read the .off

    Subdivision_method_3::PTQ(P, WLoop_mask_3<Polyhedron>(), d);

    cout << P; // write the .off

    return 0;
}

```

The points generated by the geometry mask are semantically required to converge to a smooth surface. This is the requirement imposed by the theory of the subdivision surface. *Subdivision_method_3* does not enforce this requirement, nor will it verify the smoothness of the subdivided mesh. *Subdivision_method_3* guarantees the topological properties of the subdivided mesh. A genus- n 2-manifold is assured to be subdivided into a genus- n 2-manifold. But when specialized with ill-designed geometry masks, *Subdivision_method_3* may generate a surface that is odd, not smooth, or not even exist.

3D Surface Subdivision Methods

Reference Manual

Le-Jeng Andy Shiue

Subdivision methods recursively refine the control mesh (i.e. the input mesh) and generate points approximating the limit surface. Designed to work on the class *Polyhedron_3*, *Subdivision_method_3* aims to be easy to use and to extend. *Subdivision_method_3* is not a class, but a namespace which consists of four popular subdivision methods and their refinement hosts. Supported subdivision methods include Catmull-Clark, Loop, Doo-Sabin and $\sqrt{3}$ subdivisions. Their respective refinement hosts are PQQ, PTQ, DQQ and $\sqrt{3}$ refinements. Variations of those methods can be easily extended by substituting the geometry computation of the refinement host.

31.9 Classified Reference Pages

Concepts

<i>PQQMask_3</i>	page 1886
<i>PTQMask_3</i>	page 1887
<i>DQQMask_3</i>	page 1888
<i>Sqrt3Mask_3</i>	page 1889

Classes

<i>CGAL::Subdivision_method_3</i>	page 1883
<i>CGAL::CatmullClark_mask_3<Polyhedron_3></i>	page 1890
<i>CGAL::Loop_mask_3<Polyhedron_3></i>	page 1892
<i>CGAL::DooSabin_mask_3<Polyhedron_3></i>	page 1893
<i>CGAL::Sqrt3_mask_3<Polyhedron_3></i>	page 1894

31.10 Alphabetical List of Reference Pages

<i>CatmullClark_mask_3<Polyhedron_3></i>	page 1890
<i>DooSabin_mask_3<Polyhedron_3></i>	page 1893
<i>DQQMask_3</i>	page 1888
<i>Loop_mask_3<Polyhedron_3></i>	page 1892

<i>PQQMask_3</i>	page 1886
<i>PTQMask_3</i>	page 1887
<i>Sqrt3Mask_3</i>	page 1889
<i>Sqrt3_mask_3<Polyhedron_3></i>	page 1894
<i>Subdivision_method_3</i>	page 1883

CGAL::Subdivision_method_3

Definition

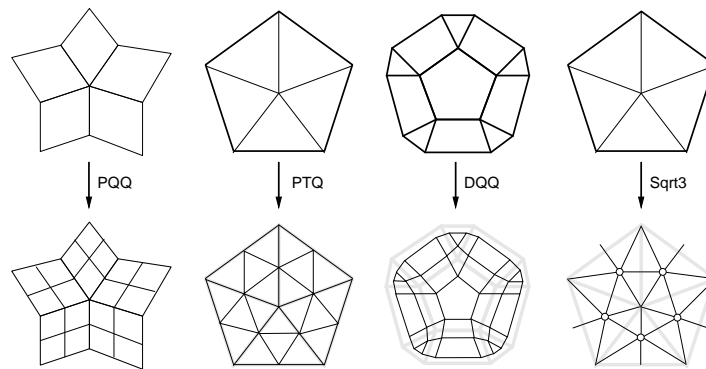
A subdivision method recursively refines a coarse mesh and generates an ever closer approximation to a smooth surface. *Subdivision_method_3* consists of four subdivision methods and their refinement hosts. Each refinement host is a template function of a polyhedron class and a geometry policy class. It refines the connectivity of the control mesh and computes the geometry of the refined mesh. The geometry computation is dedicated to the custom geometry policy. A geometry policy consists of functions that compute the new point based on the subdivision stencil. A stencil defines the footprint (a submesh of the control mesh) of a new point.

The four supported refinement hosts are the primal quadrilateral quadrissection (PQQ), the primal triangle quadri-section (PTQ), the dual quadrilateral quadrissection (DQQ), and the $\sqrt{3}$ triangulation. These refinements are respectively used in Catmull-Clark, Loop, Doo-Sabin and $\sqrt{3}$ subdivision.

```
#include <CGAL/Subdivision_method_3.h>
```

Refinement Host

A refinement host is a template function of a polyhedron class and a geometry mask class. It refines the input polyhedron, and computes new points through the geometry masks. *Subdivision_method_3* supports four refinement hosts: *PQQ*, *PTQ*, *DQQ* and *Sqrt3*.



```
template <class Polyhedron_3, template <typename> class Mask>
void PQQ( Polyhedron_3& p, Mask<Polyhedron_3> mask, int step = 1)
```

applies the PQQ refinement on the control mesh *p* *step* times. The geometry of the refined mesh is computed by the geometry policy *mask*. This function overwrites the control mesh *p* with the refined mesh.

```
template <class Polyhedron_3, template <typename> class Mask>
void PTQ( Polyhedron_3& p, Mask<Polyhedron_3> mask, int step = 1)
```

applies the PTQ refinement on the control mesh *p* *step* times, where *p* contains only triangle facets. The geometry of the refined mesh is computed by the geometry policy *mask*. This function overwrites the control mesh *p* with the refined mesh. The result of a non-triangle mesh *p* is undefined.

```
template <class Polyhedron_3, template <typename> class Mask>
void DQQ( Polyhedron_3& p, Mask<Polyhedron_3> mask, int step = 1)
```

applies the DQQ refinement on the control mesh p $step$ times. The geometry of the refined mesh is computed by the geometry policy $mask$. This function overwrites the control mesh p with the refined mesh.

```
template <class Polyhedron_3, template <typename> class Mask>
void Sqrt3( Polyhedron_3& p, Mask<Polyhedron_3> mask, int step = 1)
```

applies the $\sqrt{3}$ triangulation on the control mesh p $step$ times, where p contains only triangle facets. The geometry of the refined mesh is computed by the geometry policy $mask$. This function overwrites the control mesh p with the refined mesh. The result of a non-triangle mesh p is undefined.

Subdivision Method

```
template <class Polyhedron_3>
void CatmullClark_subdivision( Polyhedron_3& p, int step = 1)
```

applies Catmull-Clark subdivision $step$ times on the control mesh p . This function overwrites the control mesh p with the subdivided mesh.

```
template <class Polyhedron_3>
void Loop_subdivision( Polyhedron_3& p, int step = 1)
```

applies Loop subdivision $step$ times on the control mesh p . This function overwrites the control mesh p with the subdivided mesh.

```
template <class Polyhedron_3>
void DooSabin_subdivision( Polyhedron_3& p, int step = 1)
```

applies Doo-Sabin subdivision $step$ times on the control mesh p . This function overwrites the control mesh p with the subdivided mesh.

```
template <class Polyhedron_3>
void Sqrt3_subdivision( Polyhedron_3& p, int step = 1)
```

applies $\sqrt{3}$ subdivision $step$ times on the control mesh p . This function overwrites the control mesh p with the subdivided mesh.

See Also

CGAL::CatmullClark_mask_3<Polyhedron_3> page [1890](#)
CGAL::Loop_mask_3<Polyhedron_3> page [1892](#)
CGAL::Sqrt3_mask_3<Polyhedron_3> page [1894](#)

Example

This example program subdivides a polyhedral mesh with Catmull-Clark subdivision.

```
// file: examples/Subdivision_method_3/CatmullClark_subdivision.C

#include <CGAL/Cartesian.h>
#include <CGAL/Subdivision_method_3.h>
#include <iostream>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>

typedef CGAL::Cartesian<double>          Kernel;
typedef CGAL::Polyhedron_3<Kernel>      Polyhedron;

using namespace std;
using namespace CGAL;

int main(int argc, char** argv) {
    if (argc != 2) {
        cout << "Usage: CatmullClark_subdivision d < filename" << endl;
        cout << "      d: the depth of the subdivision (0 < d < 10)" << endl;
        cout << "      filename: the input mesh (.off)" << endl;
        return 0;
    }

    int d = argv[1][0] - '0';

    Polyhedron P;
    cin >> P; // read the .off

    Subdivision_method_3::CatmullClark_subdivision(P,d);

    cout << P; // write the .off

    return 0;
}
```

PQQMask_3

Required member functions for the PQQMask_3 concept. This policy concept of geometric computations is used in `CGAL::Subdivision_method_3::PQQ<Polyhedron_3, Mask>`.

Operations

`void mask.facet_node(Facet_handle facet, Point_3& pt)`

computes the facet-point pt based on the neighborhood of the facet f .

`void mask.edge_node(Edge_handle e, Point_3& pt)`

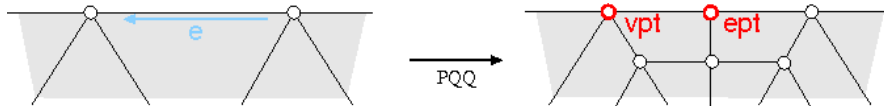
computes the edge-point pt based on the neighborhood of the edge e .

`void mask.vertex_node(Vertex_handle v, Point_3& pt)`

computes the vertex-point pt based on the neighborhood of the vertex v .

`void mask.border_node(Halfedge_handle e, Point_3& ept, Point_3& vpt)`

computes the edge-point ept and the vertex-point vpt based on the neighborhood of the border edge e .



Has Models

`CGAL::CatmullClark_mask_3<Polyhedron_3>` page [1890](#)

See Also

`CGAL::Subdivision_method_3` page [1883](#)

PTQMask_3

Required member functions for the PTQMask_3 concept. This policy concept of geometric computations is used in `CGAL::Subdivision_method_3::PTQ<Polyhedron_3, Mask>`.

Operations

`void mask.edge_node(Edge_handle e, Point_3& pt)`

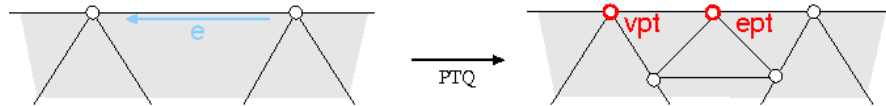
computes the edge-point pt based on the neighborhood of the edge e .

`void mask.vertex_node(Vertex_handle v, Point_3& pt)`

computes the vertex-point pt based on the neighborhood of the vertex v .

`void mask.border_node(Halfedge_handle e, Point_3& ept, Point_3& vpt)`

computes the edge-point ept and the vertex-point vpt based on the neighborhood of the border edge e .



Has Models

`CGAL::Loop_mask_3<Polyhedron_3>` page [1892](#)

See Also

`CGAL::Subdivision_method_3` page [1883](#)

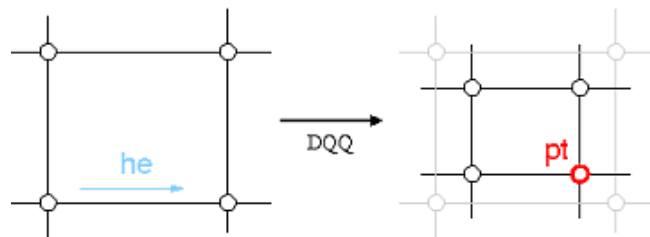
DQQMask_3

Required member functions for the DQQMask_3 concept. This policy concept of geometric computations is used in *CGAL::Subdivision_method_3::DQQ<Polyhedron_3, Mask>*.

Operations

void *mask.corner_node(Halfedge_handle he, Point_3& pt)*

computes the subdivided point *pt* based on the neighborhood of the vertex pointed by the halfedge *he*.



Has Models

CGAL::DooSabin_mask_3<Polyhedron_3> page [1893](#)

See Also

CGAL::Subdivision_method_3 page [1883](#)

Sqrt3Mask_3

Required member functions for the Sqrt3Mask_3 concept. This policy concept of geometric computations is used in *CGAL::Subdivision_method_3::Sqrt3<Polyhedron_3, Mask>*.

Operations

void *mask.facet_node(Facet_handle f, Point_3& pt)*

computes the subdivided point *pt* based on the neighborhood of the facet *f*.

void *mask.vertex_node(Vertex_handle v, Point& pt)*

computes the subdivided point *pt* based on the neighborhood of the vertex *v*.

Has Models

CGAL::Sqrt3_mask_3<Polyhedron_3> page [1894](#)

See Also

CGAL::Subdivision_method_3 page [1883](#)

CGAL::CatmullClark_mask_3<Polyhedron_3>

Definition

A stencil determines a source neighborhood whose points contribute to the position of a refined point. The geometry mask of a stencil specifies the computation on the nodes of the stencil. *CatmullClark_mask_3<Polyhedron_3>* implements the geometry masks of Catmull-Clark subdivision on a *Polyhedron_3<Cartesian>*.

```
#include <CGAL/Subdivision_mask_3.h>
```

Parameters

The full template declaration of *CatmullClark_mask_3<Polyhedron_3>* states one template parameter:

```
template < class Polyhedron_3> class CatmullClark_mask_3;
```

The only parameter requires a *Polyhedron_3* as the argument. The *Polyhedron_3* should be specialized with the *Cartesian* kernel, which defines the *Point_3* for the vertices.

Creation

```
CatmullClark_mask_3<Polyhedron_3> CC;      default constructor.
```

Stencil functions

```
void CC.facet_node( Facet_handle f, Point_3& pt)
```

computes the Catmull-Clark facet-point *pt* of the facet *f*.

```
void CC.edge_node( Edge_handle e, Point_3& pt)
```

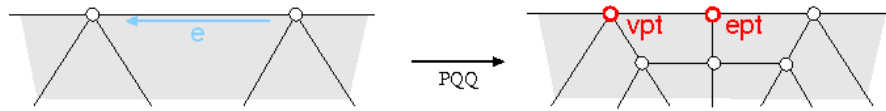
computes the Catmull-Clark edge-point *pt* of the edge *e*.

```
void CC.vertex_node( Vertex_handle v, Point_3& pt)
```

computes the Catmull-Clark vertex-point *pt* of the vertex *v*.

```
void CC.border_node( Halfedge_handle e, Point_3& ept, Point_3& vpt)
```

computes the Catmull-Clark edge-point *ept* and the Catmull-Clark vertex-point *vpt* of the border edge *e*.



See Also

CGAL::Subdivision_method_3 page [1883](#)

CGAL::Loop_mask_3<Polyhedron_3>

Definition

A stencil determines a source neighborhood whose points contribute to the position of a refined point. The geometry mask of a stencil specifies the computation on the nodes of the stencil. *Loop_mask_3<Polyhedron_3>* implements the geometry masks of Loop subdivision on a triangulated *Polyhedron_3<Cartesian>*.

```
#include <CGAL/Subdivision_mask_3.h>
```

Parameters

The full template declaration of *Loop_mask_3<Polyhedron_3>* states one template parameter:

```
template < class Polyhedron_3> class Loop_mask_3;
```

The only parameter requires a *Polyhedron_3* as the argument. The *Polyhedron_3* should be specialized with the *Cartesian* kernel, which defines the *Point_3* for the vertices.

Creation

```
Loop_mask_3<Polyhedron_3> L;           default constructor.
```

Stencil functions

```
void L.edge_node( Edge_handle e, Point_3& pt)

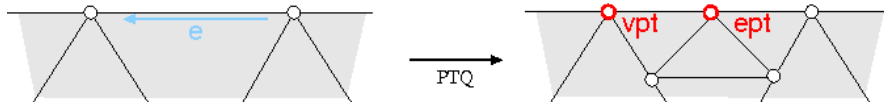
    computes the Loop edge-point pt of the edge e.
```

```
void L.vertex_node( Vertex_handle v, Point_3& pt)

    computes the Loop vertex-point pt of the vertex v.
```

```
void L.border_node( Halfedge_handle e, Point_3& ept, Point_3& vpt)

    computes the Loop edge-point ept and the Loop vertex-point vpt of the border edge e.
```



See Also

CGAL::Subdivision_method_3 page [1883](#)

CGAL::DooSabin_mask_3<Polyhedron_3>

Definition

A stencil determines a source neighborhood whose points contribute to the position of a refined point. The geometry mask of a stencil specifies the computation on the nodes of the stencil. *DooSabin_mask_3<Polyhedron_3>* implements the geometry masks of Doo-Sabin subdivision on a *Polyhedron_3<Cartesian>*.

```
#include <CGAL/Subdivision_mask_3.h>
```

Parameters

The full template declaration of *DooSabin_mask_3<Polyhedron_3>* states one template parameter:

```
template < class Polyhedron_3> class DooSabin_mask_3;
```

The only parameter requires a *Polyhedron_3* as the argument. The *Polyhedron_3* should be specialized with the *Cartesian* kernel, which defines the *Point_3* for the vertices.

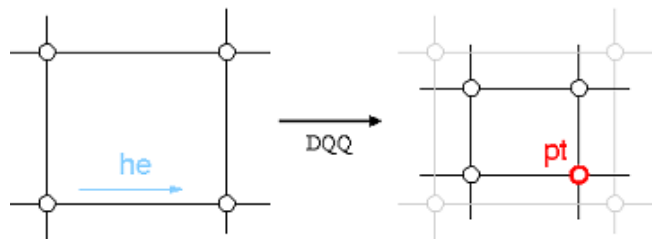
Creation

```
DooSabin_mask_3<Polyhedron_3> DS;          default constructor.
```

Stencil functions

```
void DS.corner_node( Halfedge_handle he, Point_3& pt)
```

computes the Doo-Sabin point *pt* of the vertex pointed by the halfedge *he*.



See Also

CGAL::Subdivision_method_3 page [1883](#)

CGAL::Sqrt3_mask_3<Polyhedron_3>

Definition

A stencil determines a source neighborhood whose points contribute to the position of a refined point. The geometry mask of a stencil specifies the computation on the nodes of the stencil. *Sqrt3_mask_3<Polyhedron_3>* implements the geometry masks of $\sqrt{3}$ subdivision on a triangulated *Polyhedron_3<Cartesian>*.

```
#include <CGAL/Subdivision_mask_3.h>
```

Parameters

The full template declaration of *Sqrt3_mask_3<Polyhedron_3>* states one template parameter:

```
template < class Polyhedron_3> class Sqrt3_mask_3;
```

The only parameter requires a *Polyhedron_3* as the argument. The *Polyhedron_3* should be specialized with the *Cartesian* kernel, which defines the *Point_3* for the vertices.

Creation

```
Sqrt3_mask_3<Polyhedron_3> S;           default constructor.
```

Stencil functions

```
void    S.facet_node( Facet_handle f, Point_3& pt)
           computes the  $\sqrt{3}$  facet-point pt of the facet f.
```

```
void    S.vertex_node( Vertex_handle v, Point& pt)
           computes the  $\sqrt{3}$  vertex-point pt of the vertex v.
```

See Also

CGAL::Subdivision_method_3 page [1883](#)

Chapter 32

Planar Parameterization of Triangulated Surface Meshes

Laurent Saboret, Pierre Alliez and Bruno Lévy

Contents

32.1 Introduction	1896
32.2 Basics	1897
32.2.1 Default Surface Parameterization	1897
32.2.2 Input Mesh for parameterize()	1898
32.2.3 Default Parameterization Example	1898
32.2.4 Enhanced parameterize() function	1900
32.2.5 Introduction to the Package Concepts	1901
32.3 Surface Parameterization Methods	1902
32.3.1 Fixed Border Surface Parameterizations	1902
32.3.2 Free Border Surface Parameterizations	1906
32.3.3 Discrete Authalic Parameterization Example	1907
32.3.4 Square Border Arc Length Parameterization Example	1907
32.4 Sparse Linear Algebra	1908
32.4.1 List of Solvers	1908
32.4.2 TAUCS Solver Example	1909
32.5 Cutting a Mesh	1909
32.5.1 Computing a Cut Graph	1909
32.5.2 Applying a Cut	1910
32.5.3 Cutting a Mesh Example	1910
32.6 Output	1914
32.6.1 EPS Output Example	1914
32.7 Complexity and Guarantees	1920
32.7.1 Parameterization Methods and Guarantees	1920
32.7.2 Precision	1921
32.7.3 Algorithmic Complexity	1921
32.8 Software Design	1922
32.8.1 Global Function parameterize()	1922
32.8.2 No Common Parameterization Algorithm	1922
32.8.3 Fixed_border_parameterizer_3 Class	1924
32.8.4 Border Parameterizations	1925

32.8.5	ParameterizationMesh_3 and ParameterizationPatchableMesh_3 Concepts	1925
32.8.6	SparseLinearAlgebraTraits_d Concept	1926
32.8.7	Cutting a Mesh	1926
32.9	Extending the Package and Reusing Code	1926
32.9.1	Reusing Mesh Adaptors	1926
32.9.2	Reusing Sparse Linear Algebra	1926
32.9.3	Adding New Parameterization Methods	1926
32.9.4	Adding New Border Parameterization Methods	1927
32.9.5	Mesh Cutting	1927

32.1 Introduction

Parameterizing a surface amounts to finding a one-to-one mapping from a suitable domain to the surface. A good mapping is the one which minimizes either angle distortions (conformal parameterization) or area distortions (equiareal parameterization) in some sense. In this package, we focus on parameterizing triangulated surfaces which are homeomorphic to a disk, and on piecewise linear mappings onto a planar domain.

Although the main motivation behind the first parameterization methods was the application to texture mapping, it is now frequently used for mapping more sophisticated modulation signals (such as normal, transparency, reflection or light modulation maps), fitting scattered data, re-parameterizing spline surfaces, repairing CAD models, approximating surfaces and remeshing.

This CGAL package implements some of the state-of-the-art surface parameterization methods, such as least squares conformal maps, discrete conformal map, discrete authalic parameterization, Floater mean value coordinates or tutte barycentric mapping. These methods mainly distinguish by the distortion they minimize (angles vs. areas), by the constrained border onto the planar domain (convex polygon vs. free border) and by the guarantees provided in terms of bijective mapping.

The package proposes currently an interface with `CGAL::Polyhedron_3<Traits>` data structure.

Since parameterizing meshes require efficient representation of sparse matrices and efficient iterative or direct linear solvers, we provide a unified interface to state-of-the-art linear solver libraries (TAUCS), and propose a separate package devoted to OpenNL sparse linear solver.

Note that linear solvers commonly use double precision floating point numbers. Therefore, this package is intended to be used with a CGAL Cartesian kernel with doubles.

The intended audience of this package is researchers, developers or students developing algorithms around parameterization of triangle meshes for geometry processing as well as for signal mapping on triangulated surfaces.

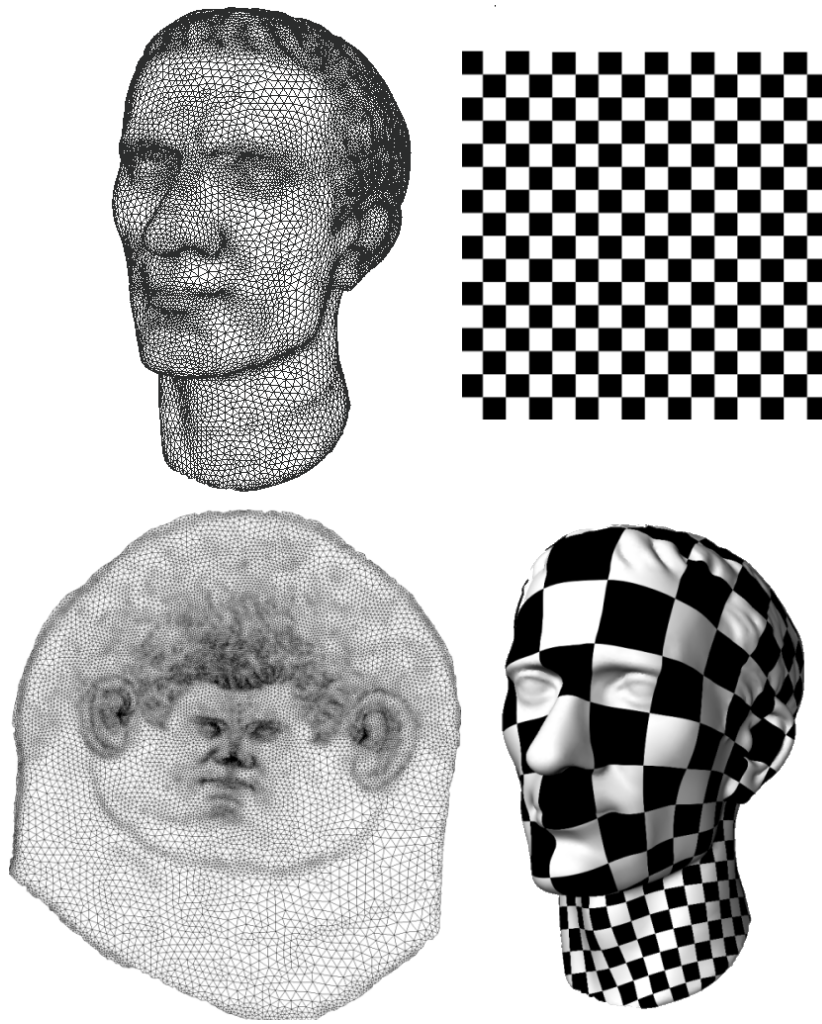


Figure 32.1: Texture mapping via Least Squares Conformal Maps parameterization. Top: original mesh and texture. Bottom: parameterized mesh (left: parameter space, right: textured mesh).

32.2 Basics

32.2.1 Default Surface Parameterization

From the user point of view, the simplest entry point to this package is the following function:

Parameterizer_traits_3<ParameterizationMesh_3>::Error_code

parameterize(ParameterizationMesh_3 & mesh)

Compute a one-to-one mapping from a 3D triangle surface mesh to a 2D circle, using Floater Mean Value Coordinates algorithm. A one-to-one piecewise linear mapping is guaranteed. The result is a pair of (u,v) parameter coordinates for each vertex of the input mesh.

Preconditions: mesh must be a triangle mesh surface with one connected component.

The function *CGAL::parameterize()* applies a default surface parameterization method: Floater Mean Value Coordinates [Flo03a], with an arc-length circular border parameterization, and using OpenNL sparse linear solver [Lev05]. The *ParameterizationMesh_3* concept defines the input meshes handled by *CGAL::parameterize()*. See Section 32.2.2. The result is stored into the (u,v) fields of the mesh halfedges.

Note: *CGAL::Parameterizer_traits_3<ParameterizationMesh_3>* is the (pure virtual) superclass of all surface parameterizations and defines the error codes.

32.2.2 Input Mesh for parameterize()

The input meshes handled *directly* by *CGAL::parameterize()* must be models of *ParameterizationMesh_3*, triangulated, 2-manifold, oriented, and homeomorphic to discs (possibly with holes).

Note: *ParameterizationMesh_3* is a general concept to access a polyhedral mesh. It is optimized for the *Surface_mesh_parameterization* package only in the sense that it defines the accessors to fields specific to the parameterization domain (*index*, *u*, *v*, *is_parameterized*). The extra constraints needed by the surface parameterization methods (triangulated, 2-manifold, homeomorphic to a disc) are not part of the concept and are checked at runtime.

This package provides a model of the *ParameterizationMesh_3* concept to access *CGAL::Polyhedron_3<Traits>* :

CGAL::Parameterization_polyhedron_adaptor_3<Polyhedron_3_>

We will see later that *CGAL::parameterize()* can support *indirectly* meshes that are not topological disks.

32.2.3 Default Parameterization Example

Simple_parameterization.C applies the default parameterization to a *CGAL::Polyhedron_3<Traits>* mesh (must be a topological disk). Eventually, it extracts the result from halfedges and prints it.

```
// Simple_parameterization.C

#include <CGAL/Cartesian.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>
#include <CGAL/Parameterization_polyhedron_adaptor_3.h>
#include <CGAL/parameterize.h>
```

```

#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <fstream>

// -----
// Private types
// -----

typedef CGAL::Cartesian<double>          Kernel;
typedef CGAL::Polyhedron_3<Kernel>      Polyhedron;

// -----
// main()
// -----

int main(int argc, char * argv[])
{
    std::cerr << "PARAMETERIZATION" << std::endl;
    std::cerr << "  Floater parameterization" << std::endl;
    std::cerr << "  Circle border" << std::endl;
    std::cerr << "  OpenNL solver" << std::endl;

    //*****
    // decode parameters
    //*****

    if (argc-1 != 1)
    {
        std::cerr << "Usage: " << argv[0] << " input_file.off" << std::endl;
        return(EXIT_FAILURE);
    }

    // File name is:
    const char* input_filename = argv[1];

    //*****
    // Read the mesh
    //*****

    // Read the mesh
    std::ifstream stream(input_filename);
    if(!stream)
    {
        std::cerr << "FATAL ERROR: cannot open file " << input_filename << std::endl;
        return EXIT_FAILURE;
    }
    Polyhedron mesh;
    stream >> mesh;

    //*****
    // Create Polyhedron adaptor

```

```

// Note: no cutting => we support only
// meshes that are topological disks
//*****

typedef CGAL::Parameterization_polyhedron_adaptor_3<Polyhedron>
                                     Parameterization_polyhedron_adaptor;
Parameterization_polyhedron_adaptor mesh_adaptor(mesh);

//*****
// Floater Mean Value Coordinates parameterization
// (defaults are circular border and OpenNL solver)
//*****

// Type that defines the error codes
typedef CGAL::Parameterizer_traits_3<Parameterization_polyhedron_adaptor>
                                     Parameterizer;

Parameterizer::Error_code err = CGAL::parameterize(mesh_adaptor);
if (err != Parameterizer::OK)
    std::cerr << "FATAL ERROR: " << Parameterizer::get_error_message(err) << std::endl;

//*****
// Output
//*****

if (err == Parameterizer::OK)
{
    // Raw output: dump (u,v) pairs
    Polyhedron::Vertex_const_iterator pVertex;
    for (pVertex = mesh.vertices_begin();
         pVertex != mesh.vertices_end();
         pVertex++)
    {
        // (u,v) pair is stored in any halfedge
        double u = mesh_adaptor.info(pVertex->halfedge())->uv().x();
        double v = mesh_adaptor.info(pVertex->halfedge())->uv().y();
        std::cout << "(u,v) = (" << u << ", " << v << ")" << std::endl;
    }
}

return (err == Parameterizer::OK) ? EXIT_SUCCESS : EXIT_FAILURE;
}

```

32.2.4 Enhanced parameterize() function

This package provides a second *CGAL::parameterize()* entry point where the user can specify a parameterization method:

Parameterizer_traits_3<ParameterizationMesh_3>::Error_code

parameterize(ParameterizationMesh_3 & mesh,

ParameterizerTraits_3 parameterizer)

Compute a one-to-one mapping from a 3D triangle surface 'mesh' to a simple 2D domain. The mapping is piecewise linear on the triangle mesh. The result is a pair (u,v) of parameter coordinates for each vertex of the input mesh. One-to-one mapping may be guaranteed or not, depending on the chosen *ParameterizerTraits_3* algorithm.

Preconditions: 'mesh' must be a triangle surface mesh with one connected component, and the mesh border must be mapped onto a convex polygon (for fixed border parameterizations).

32.2.5 Introduction to the Package Concepts

The *ParameterizerTraits_3* concept

This CGAL package implements some of the state-of-the-art surface parameterization methods, such as Least Squares Conformal Maps, Discrete Conformal Map, Discrete Authalic Parameterization, Floater Mean Value Coordinates or Tutte Barycentric Mapping. These methods are provided as models of the *ParameterizerTraits_3* concept. See Section 32.3.

Each of these surface parameterization methods is templated by the input mesh type, a border parameterization and a solver:

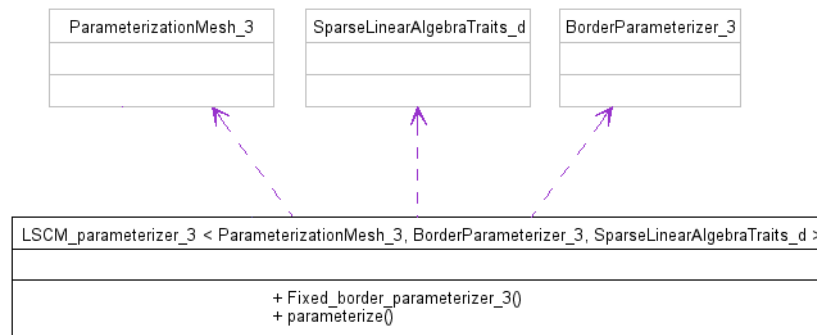


Figure 32.2: A parameterizer UML class diagram (simplified).

The *BorderParameterizer_3* concept

Parameterization methods for borders are used as traits classes modifying the behavior of *ParameterizerTraits_3* models. They are provided as models of the *BorderParameterizer_3* concept. See Sections 32.3.1 and 32.3.2.

The *SparseLinearAlgebraTraits_d* concept

This package solves sparse linear systems using solvers which are models of *SparseLinearAlgebraTraits_d*. See Section 32.4.

The `ParameterizationMesh_3` and `ParameterizationPatchableMesh_3` Concepts

As described in Section 32.2.2 the input meshes handled by `CGAL::parameterize()` must be models of the `ParameterizationMesh_3` concept. The surface parameterization methods provided by this package only support surfaces which are homeomorphic to disks, possibly with holes. Nevertheless meshed with arbitrary topology and number of connected components can be parameterized, provided that the user specifies a *cut graph* (an oriented list of vertices) which is the border of a topological disc. If no cut graph is specified as input, the longest border of the input mesh is taken by default, the others being considered as holes.

For this purpose, the `CGAL::Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>` class is responsible for *virtually* cutting a patch into a `ParameterizationPatchableMesh_3` mesh. The resulting patch is a topological disk (if the input cutting path is correct) and provides a `ParameterizationMesh_3` interface. It can be used as parameter for the function `CGAL::parameterize()`.

`ParameterizationPatchableMesh_3` inherits from `ParameterizationMesh_3`, thus is a concept for a 3D surface mesh. `ParameterizationPatchableMesh_3` adds the ability to support patches and virtual seams. *Patches* are a subset of a 3D mesh. *Virtual seams* behave as if the surface was cut along a cut graph. More information is provided in Section 32.5.

32.3 Surface Parameterization Methods

This CGAL package implements some of the state-of-the-art surface parameterization methods, such as Least Squares Conformal Maps, Discrete Conformal Map, Discrete Authalic Parameterization, Floater Mean Value Coordinates or Tutte Barycentric Mapping. These methods are provided as models of the `ParameterizerTraits_3` concept.

32.3.1 Fixed Border Surface Parameterizations

Fixed Border Surface Parameterizations need a set of constraints: two (u,v) coordinates for each vertex along the border. Such border parameterizations are described in Section 32.3.1.

Tutte Barycentric Mapping

`CGAL::Barycentric_mapping_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>`

The Barycentric Mapping parameterization method has been introduced by Tutte [Tut63]. In parameter space, each vertex is placed at the barycenter of its neighbors to achieve the so-called convex combination condition. This algorithm amounts to solve one sparse linear solver for each set of parameter coordinates, with a `#vertices` x `#vertices` sparse and symmetric positive definite matrix (if the border vertices are eliminated from the linear system). A coefficient (i, j) of the matrix is set to 1 for an edge linking the vertex v_i to the vertex v_j , to minus the degree of the vertex v_i for a diagonal element, and to 0 for any other matrix entry. Although a bijective mapping is guaranteed when the border is convex, this method does not minimize angles nor areas distortion.

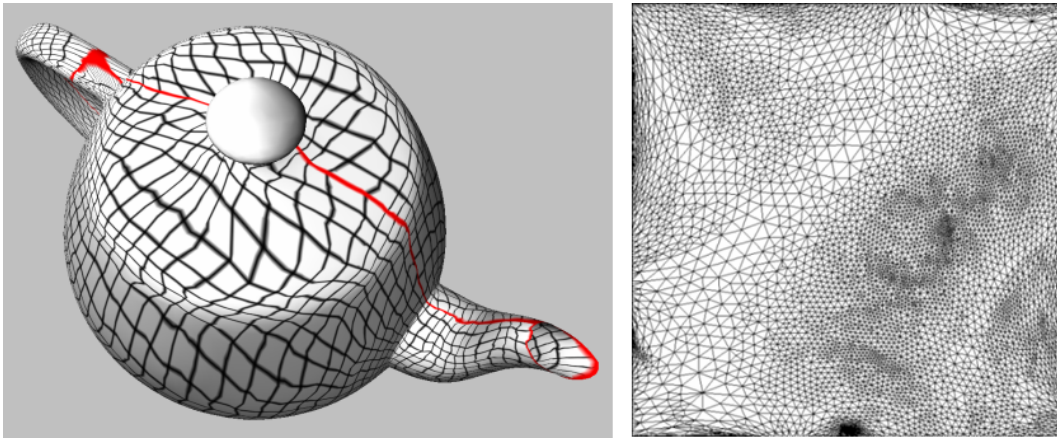


Figure 32.3: Left: Tutte barycentric mapping parameterization (the red line depicts the cut graph). Right: parameter space.

Discrete Conformal Map

CGAL::Discrete_conformal_map_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*, *SparseLinearAlgebraTraits_d*>

Discrete conformal map parameterization has been introduced by Eck et al. to the graphics community [EDD⁺95]. It attempts to lower angle deformation by minimizing a discrete version of the Dirichlet energy as derived by Pinkall and Polthier [PP93]. A one-to-one mapping is guaranteed only when the two following conditions are fulfilled: the barycentric mapping condition (each vertex in parameter space is a convex combination of its neighboring vertices), and the border is convex. This method solves two $\#vertices \times \#vertices$ sparse linear systems. The matrix (the same for both systems) is sparse and symmetric definite positive (if the border vertices are eliminated from the linear system and if the mesh contains no hole), thus can be efficiently solved using dedicated linear solvers.

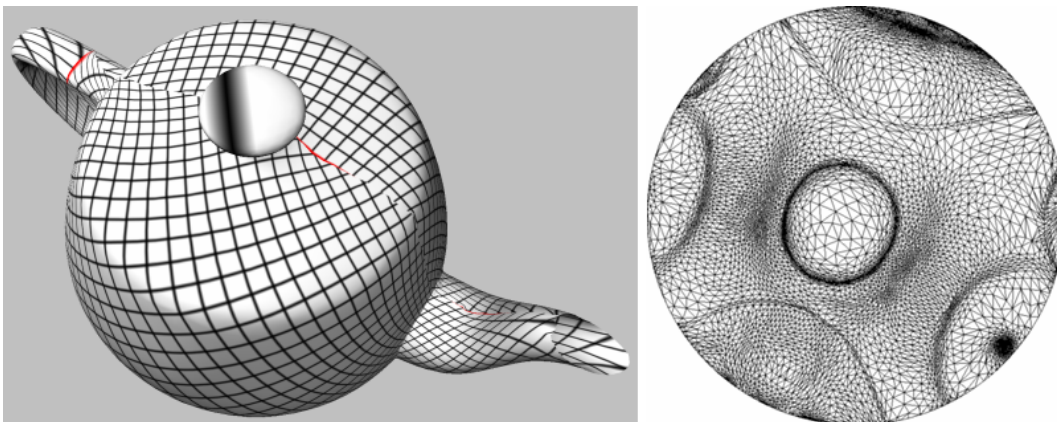


Figure 32.4: Left: discrete conformal map. Right: parameter space.

Floater Mean Value Coordinates

CGAL::Mean_value_coordinates_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>

The mean value coordinates parameterization method has been introduced by Floater [Flo03a]. Each vertex in parameter space is optimized so as to be a convex combination of its neighboring vertices. The barycentric coordinates are this time unconditionally positive, by deriving an application of the mean theorem for harmonic functions. This method is in essence an approximation of the discrete conformal maps, with a guaranteed one-to-one mapping when the border is convex. This method solves two $\#vertices \times \#vertices$ sparse linear systems. The matrix (the same for both systems) is asymmetric.

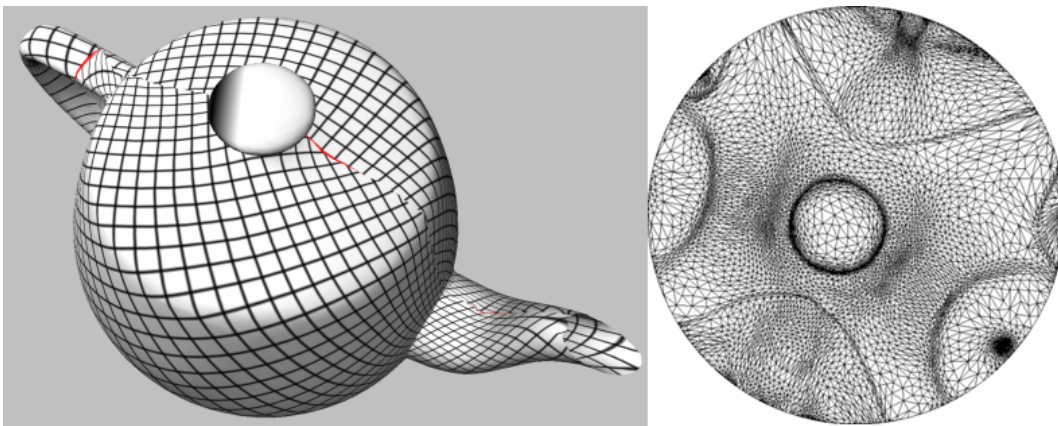


Figure 32.5: Floater Mean Value Coordinates

Discrete Authalic parameterization

CGAL::Discrete_authalic_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>

The discrete authalic parameterization method has been introduced by Desbrun et al. [DMA02]. It corresponds to a weak formulation of an area-preserving method, and in essence locally minimizes the area distortion. A one-to-one mapping is guaranteed only if the convex combination condition is fulfilled and the border is convex. This method solves two $\#vertices \times \#vertices$ sparse linear systems. The matrix (the same for both systems) is asymmetric.

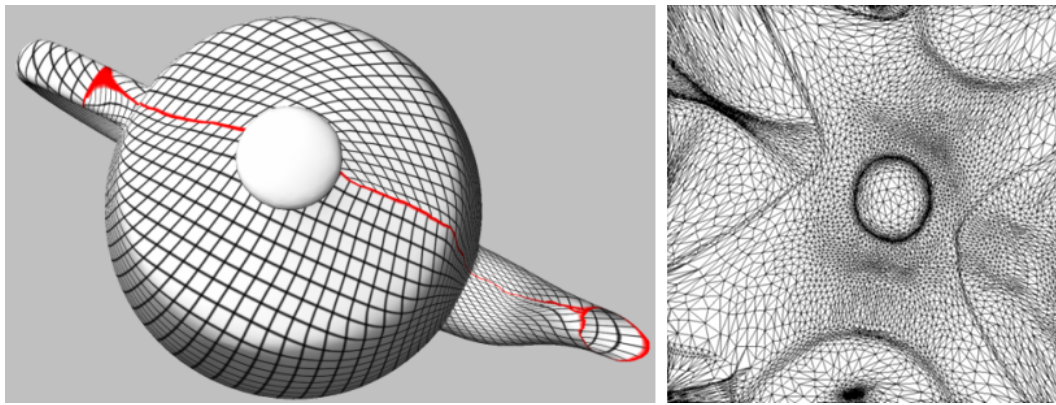


Figure 32.6: Discrete Authalic Parameterization

Border Parameterizations for Fixed Methods

Parameterization methods for borders are used as traits classes modifying the behavior of *ParameterizerTraits_3* models. They are provided as models of the *BorderParameterizer_3* concept. Border parameterizations for fixed border surface parameterizations are a family of methods to define a set of constraints, namely two u, v coordinates for each vertex along the border.

- The user can select a border parameterization among two commonly used methods: uniform or arc-length parameterization.

Usage:

Uniform border parameterization is more stable, although it gives poor visual results. The arc-length border parameterization is used by default.

- One convex shape specified by one shape among two standard ones: a circle or a square.

Usage:

The circular border parameterization is used by default as it corresponds to the simplest convex shape. The square border parameterization is commonly used for texture mapping.

```
CGAL::Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3>
CGAL::Circular_border_uniform_parameterizer_3<ParameterizationMesh_3>
CGAL::Square_border_arc_length_parameterizer_3<ParameterizationMesh_3>
CGAL::Square_border_uniform_parameterizer_3<ParameterizationMesh_3>
```

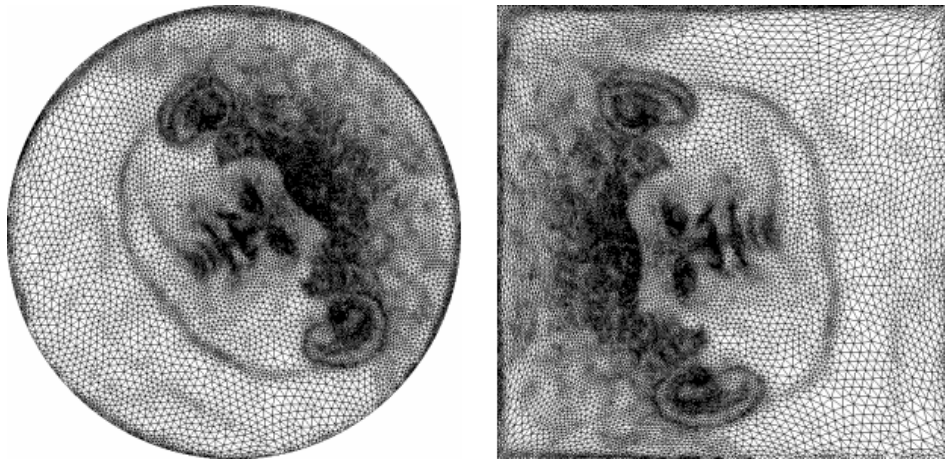


Figure 32.7: Left: Julius Cesar mask parameterization with Authalic/circular border. Right: Julius Cesar mask's image with Floater/square border.

32.3.2 Free Border Surface Parameterizations

Least Squares Conformal Maps

CGAL::LSCM_parameterizer_3<*ParameterizationMesh_3*,
SparseLinearAlgebraTraits_d>

BorderParameterizer_3,

The Least Squares Conformal Maps (LSCM) parameterization method has been introduced by Lévy et al. [LPRM02]. It corresponds to a conformal method with a free border (at least two vertices have to be constrained to obtain a unique solution), which allows further lowering of the angle distortion. A one-to-one mapping is not guaranteed by this method. It solves a $(2 \times \#triangles) \times \#vertices$ sparse linear system in the least squares sense, which implies solving a symmetric matrix.

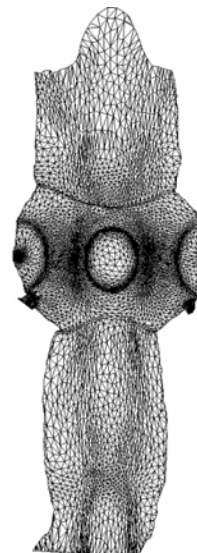
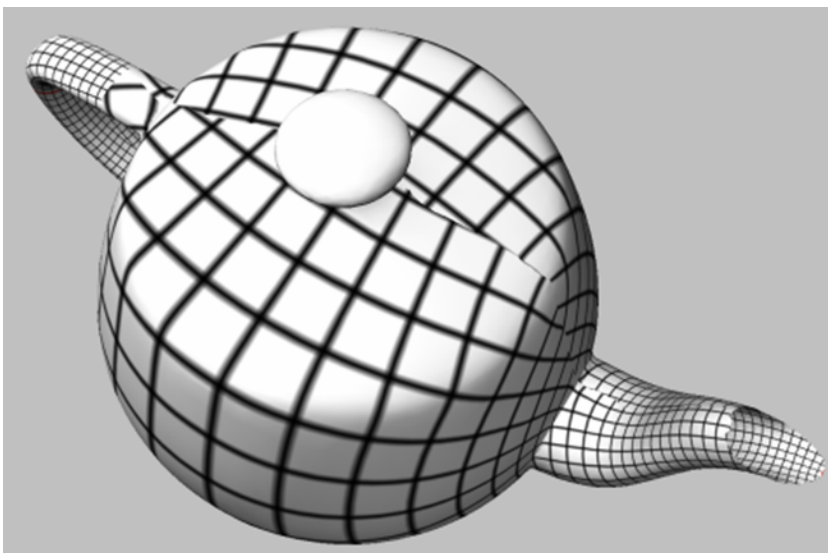


Figure 32.8: Least squares conformal maps.

Border Parameterizations for Free Methods

Parameterization methods for borders are used as traits classes modifying the behavior of *ParameterizerTraits_3* models. They are provided as models of the *BorderParameterizer_3* concept. The border parameterizations associated to free border surface parameterization methods define only two constraints: the pinned vertices.

- *CGAL::Two_vertices_parameterizer_3<ParameterizationMesh_3>*

Usage:

CGAL::Two_vertices_parameterizer_3<ParameterizationMesh_3> is the default free border parameterization, and is the only one available in the current version of this package.

32.3.3 Discrete Authalic Parameterization Example

Authalic_parameterization.C computes a Discrete Authalic parameterization over a *CGAL::Polyhedron_3<Traits>* mesh. Specifying a specific surface parameterization instead of the default one means using the second parameter of *CGAL::parameterize()*. The differences with the first example *Simple_parameterization.C* are:

```
#include <CGAL/Discrete_authalic_parameterizer_3.h>

...

//*****
// Discrete Authalic Parameterization
//*****

typedef CGAL::Discrete_authalic_parameterizer_3<Parameterization_polyhedron_adaptor>
        Parameterizer;

Parameterizer::Error_code err = CGAL::parameterize(mesh_adaptor, Parameterizer());

...
```

32.3.4 Square Border Arc Length Parameterization Example

Square_border_parameterization.C computes a Floater mean value coordinates parameterization with a square border arc length parameterization. Specifying a specific border parameterization instead of the default one means using the second parameter of *CGAL::Mean_value_coordinates_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>*. The differences with the first example *Simple_parameterization.C* are:

```
#include <CGAL/Square_border_parameterizer_3.h>
```

```

...

//*****
// Floater Mean Value Coordinates parameterization
// with square border
//*****

// Square border parameterizer
typedef CGAL::Square_border_arc_length_parameterizer_3<Parameterization_polyhedron_adaptor>
        Border_parameterizer;

// Floater Mean Value Coordinates parameterizer with square border
typedef CGAL::Mean_value_coordinates_parameterizer_3<Parameterization_polyhedron_adaptor,
        Border_parameterizer>
        Parameterizer;

Parameterizer::Error_code err = CGAL::parameterize(mesh_adaptor, Parameterizer());

...

```

32.4 Sparse Linear Algebra

Parameterizing triangle meshes requires both efficient representation of sparse matrices and efficient iterative or direct linear solvers. We provide links to standard libraries (TAUCS) and include a separate package devoted to OpenNL sparse linear solver.

32.4.1 List of Solvers

We provide an interface to several sparse linear solvers, as models of the *SparseLinearAlgebraTraits_d* concept:

- OpenNL [\[Lev05\]](#) is shipped with CGAL. This is the default solver.

OpenNL::DefaultLinearSolverTraits<COEFFTYPE, MATRIX, VECTOR, SOLVER> in OpenNL package
OpenNL::SymmetricLinearSolverTraits<COEFFTYPE, MATRIX, VECTOR, SOLVER> in OpenNL package

Usage:

OpenNL (in the version shipped with CGAL) is a lightweight sparse linear solver. It does not support large systems, but it is highly portable and supports exact number types.

- TAUCS is a state-of-the-art direct solver for sparse symmetric matrices. It also includes an out-of-core general sparse solver.

CGAL::Taucs_solver_traits<T>
CGAL::Taucs_symmetric_solver_traits<T>

Usage:

TAUCS is very robust and supports large systems. On the other hand, it is not available on all platforms supported by CGAL and does not support exact number types.

Install:

TAUCS can be downloaded from http://www.tau.ac.il/~stoledo/taucs/2.2/taucs_full.zip.

32.4.2 TAUCS Solver Example

Taucs_parameterization.C computes the default parameterization method (Floater mean value coordinates with a circular border), but specifically instantiates the TAUCS solver. Specifying a specific solver instead of the default one (OpenNL) means using the third parameter of *CGAL::Mean_value_coordinates_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>*. The differences with the first example *Simple_parameterization.C* are:

```
#include <CGAL/Taucs_solver_traits.h>

...

//*****
// Floater Mean Value Coordinates parameterization
// (circular border) with TAUCS solver
//*****

// Circular border parameterizer (the default)
typedef CGAL::Circular_border_arc_length_parameterizer_3<Parameterization_polyhedron_adaptor>
                                                    Border_parameterizer;

// TAUCS solver
typedef CGAL::Taucs_solver_traits<double> Solver;

// Floater Mean Value Coordinates parameterization
// (circular border) with TAUCS solver
typedef CGAL::Mean_value_coordinates_parameterizer_3<Parameterization_polyhedron_adaptor,
                                                    Border_parameterizer,
                                                    Solver>
                                                    Parameterizer;

Parameterizer::Error_code err = CGAL::parameterize(mesh_adaptor, Parameterizer());

...
```

32.5 Cutting a Mesh

32.5.1 Computing a Cut Graph

All surface parameterization methods proposed in this package only deal with meshes which are homeomorphic (topologically equivalent) to discs. Nevertheless meshes with arbitrary topology and number of connected components can be parameterized, provided that the user specifies a cut graph (an oriented list of vertices),

which is the border of a topological disc. If no cut graph is provided as input, the longest border already in the input mesh is taken as default border, all other borders being considered as holes. Note that only the inside part (i.e., one connected component) of the given border is parameterized.

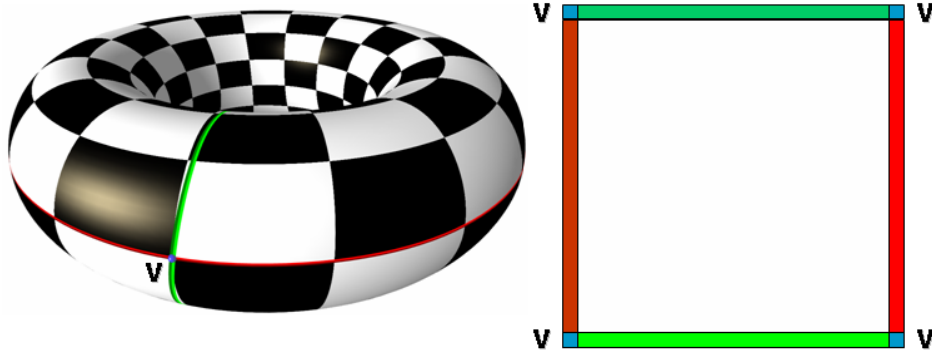


Figure 32.9: Cut Graph

This package does not provide any algorithm to transform an arbitrary mesh into a topological disk, the user being responsible for generating such a cut graph. Nevertheless, we provide in *polyhedron_ex_parameterization.C* a simple cutting algorithm for the sake of completeness.

32.5.2 Applying a Cut

The surface parameterization classes in this package only *directly* support surfaces which are homeomorphic to disks (models of *ParameterizationMesh_3*). This software design simplifies the implementation of all new parameterization methods.

The *CGAL::Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>* class is responsible for *virtually* cutting a patch in a *ParameterizationPatchableMesh_3* mesh. The resulting patch is a topological disk (if the cut graph is correct) and provides a *ParameterizationMesh_3* interface. It can be used as parameter of *CGAL::parameterize()*.

ParameterizationPatchableMesh_3 inherits from concept *ParameterizationMesh_3*, thus is a concept for a 3D surface mesh. *ParameterizationPatchableMesh_3* adds the ability to support patches and virtual seams. *Patches* are a subset of a 3D mesh. *Virtual seams* behave exactly as if the surface was cut along a certain graph.

The *ParameterizationMesh_3* interface with the Polyhedron is both a model of *ParameterizationMesh_3* and *ParameterizationPatchableMesh_3*:

CGAL::Parameterization_polyhedron_adaptor_3<Polyhedron_3_>

Note that this class is a decorator which adds *on the fly* the necessary fields to unmodified CGAL data structures (using STL maps). For better performances, it is recommended to use CGAL data structures enriched with the proper fields. See *Polyhedron_ex* class in *polyhedron_ex_parameterization.C* example.

32.5.3 Cutting a Mesh Example

Mesh_cutting_parameterization.C virtually cuts a *CGAL::Polyhedron_3<Traits>* mesh to make it a topological disk, then applies the default parameterization:

```

// Mesh_cutting_parameterization.C

#include <CGAL/Cartesian.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>
#include <CGAL/Parameterization_polyhedron_adaptor_3.h>
#include <CGAL/parameterize.h>
#include <CGAL/Parameterization_mesh_patch_3.h>

#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <fstream>

// -----
// Private types
// -----

typedef CGAL::Cartesian<double>          Kernel;
typedef CGAL::Polyhedron_3<Kernel>      Polyhedron;

// Polyhedron adaptor
typedef CGAL::Parameterization_polyhedron_adaptor_3<Polyhedron>
    Parameterization_polyhedron_adaptor;

// Type describing a border or seam as a vertex list
typedef std::list<Parameterization_polyhedron_adaptor::Vertex_handle>
    Seam;

// -----
// Private functions
// -----

// If the mesh is a topological disk, extract its longest border,
// else compute a very simple cut to make it homeomorphic to a disk.
// Return the border of this region (empty on error)
//
// CAUTION: this cutting algorithm is very naive. Write your own!
static Seam cut_mesh(Parameterization_polyhedron_adaptor& mesh_adaptor)
{
    // Helper class to compute genus or extract borders
    typedef CGAL::Parameterization_mesh_feature_extractor<Parameterization_polyhedron_adaptor>
        Mesh_feature_extractor;

    Seam seam;          // returned list

    // Get reference to Polyhedron_3 mesh
    Polyhedron& mesh = mesh_adaptor.get_adapted_mesh();

    // Extract mesh borders and compute genus
    Mesh_feature_extractor feature_extractor(mesh_adaptor);
    int nb_borders = feature_extractor.get_nb_borders();

```

```

int genus = feature_extractor.get_genus();

// If mesh is a topological disk
if (genus == 0 && nb_borders > 0)
{
    // Pick the longest border
    seam = feature_extractor.get_longest_border();
}
else // if mesh is NOT a topological disk, create a virtual cut
{
    const int CUT_LENGTH = 6;

    // Build consecutive halfedges array
    Polyhedron::Halfedge_handle seam_halfedges[CUT_LENGTH];
    seam_halfedges[0] = mesh.halfedges_begin();
    if (seam_halfedges[0] == NULL)
        return seam; // return empty list
    int i;
    for (i=1; i<CUT_LENGTH; i++)
    {
        seam_halfedges[i] = seam_halfedges[i-1]->next()->opposite()->next();
        if (seam_halfedges[i] == NULL)
            return seam; // return empty list
    }

    // Convert halfedges array to two-ways vertices list
    for (i=0; i<CUT_LENGTH; i++)
        seam.push_back(seam_halfedges[i]->vertex());
    for (i=CUT_LENGTH-1; i>=0; i--)
        seam.push_back(seam_halfedges[i]->opposite()->vertex());
}

return seam;
}

// -----
// main()
// -----

int main(int argc, char * argv[])
{
    std::cerr << "PARAMETERIZATION" << std::endl;
    std::cerr << " Floater parameterization" << std::endl;
    std::cerr << " Circle border" << std::endl;
    std::cerr << " OpenNL solver" << std::endl;
    std::cerr << " Very simple cut if model is not a topological disk" << std::endl;

    //*****
    // decode parameters
    //*****

    if (argc-1 != 1)
    {

```



```

        std::cerr << "Usage: " << argv[0] << " input_file.off" << std::endl;
        return (EXIT_FAILURE);
    }

    // File name is:
    const char* input_filename = argv[1];

    //*****
    // Read the mesh
    //*****

    // Read the mesh
    std::ifstream stream(input_filename);
    if(!stream)
    {
        std::cerr << "FATAL ERROR: cannot open file " << input_filename << std::endl;
        return EXIT_FAILURE;
    }
    Polyhedron mesh;
    stream >> mesh;

    //*****
    // Create Polyhedron adaptor
    //*****

    Parameterization_polyhedron_adaptor mesh_adaptor(mesh);

    //*****
    // Virtually cut mesh
    //*****

    // The parameterization methods support only meshes that
    // are topological disks => we need to compute a "cutting" of the mesh
    // that makes it homeomorphic to a disk
    Seam seam = cut_mesh(mesh_adaptor);
    if (seam.empty())
    {
        std::cerr << "FATAL ERROR: an unexpected error occurred while cutting the shape" << std::endl;
        return EXIT_FAILURE;
    }

    // Create a second adaptor that virtually "cuts" the mesh following the 'seam' path
    typedef CGAL::Parameterization_mesh_patch_3<Parameterization_polyhedron_adaptor>
        Mesh_patch_polyhedron;
    Mesh_patch_polyhedron mesh_patch(mesh_adaptor, seam.begin(), seam.end());

    //*****
    // Floater Mean Value Coordinates parameterization
    //*****

    // Type that defines the error codes
    typedef CGAL::Parameterizer_traits_3<Mesh_patch_polyhedron>
        Parameterizer;

```

```

Parameterizer::Error_code err = CGAL::parameterize(mesh_patch);
if (err != Parameterizer::OK)
    std::cerr << "FATAL ERROR: " << Parameterizer::get_error_message(err) << std::endl;

//*****
// Output
//*****

if (err == Parameterizer::OK)
{
    // Raw output: dump (u,v) pairs
    Polyhedron::Vertex_const_iterator pVertex;
    for (pVertex = mesh.vertices_begin();
        pVertex != mesh.vertices_end();
        pVertex++)
    {
        // (u,v) pair is stored in any halfedge
        double u = mesh_adaptor.info(pVertex->halfedge())->uv().x();
        double v = mesh_adaptor.info(pVertex->halfedge())->uv().y();
        std::cout << "(u,v) = (" << u << ", " << v << ")" << std::endl;
    }
}

return (err == Parameterizer::OK) ? EXIT_SUCCESS : EXIT_FAILURE;
}

```

32.6 Output

Parameterization methods compute (u, v) fields for each vertex of the input mesh, with the seam vertices being virtually duplicated (thanks to *CGAL::Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>*). To support this duplication, *CGAL::Parameterization_polyhedron_adaptor_3<Polyhedron_3>* stores the result in the (u, v) fields of the input mesh halfedges. A (u, v) pair is computed for each inner vertex (i.e. its halfedges share the same (u, v) pair), while a (u, v) pair is computed for each border halfedge. The user has to iterate over the mesh halfedges to get the result. Note that (u, v) fields do not exist in *CGAL::Polyhedron_3<Traits>*, thus the output traversal is specific to the way the (u, v) fields are implemented by the adaptor.

32.6.1 EPS Output Example

Complete_parameterization_example.C is a complete parameterization example which outputs the resulting parameterization to a EPS file. It gets the (u, v) fields computed by a parameterization method over a *CGAL::Polyhedron_3<Traits>* mesh with a *CGAL::Parameterization_polyhedron_adaptor_3<Polyhedron_3>* adaptor:

```

// Complete_parameterization_example.C

#include <CGAL/basic.h> // include basic.h before testing #defines

#ifdef CGAL_USE_TAUCS

```

```

#include <CGAL/Cartesian.h>
#include <CGAL/Polyhedron_3.h>
#include <CGAL/IO/Polyhedron_iostream.h>
#include <CGAL/Parameterization_polyhedron_adaptor_3.h>
#include <CGAL/parameterize.h>
#include <CGAL/Discrete_authalic_parameterizer_3.h>
#include <CGAL/Square_border_parameterizer_3.h>
#include <CGAL/Parameterization_mesh_patch_3.h>

#include <CGAL/Taucs_solver_traits.h>

#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <fstream>

// -----
// Private types
// -----

typedef CGAL::Cartesian<double>          Kernel;
typedef CGAL::Polyhedron_3<Kernel>      Polyhedron;

// Polyhedron adaptor
typedef CGAL::Parameterization_polyhedron_adaptor_3<Polyhedron>
        Parameterization_polyhedron_adaptor;

// Type describing a border or seam as a vertex list
typedef std::list<Parameterization_polyhedron_adaptor::Vertex_handle>
        Seam;

// -----
// Private functions
// -----

// If the mesh is a topological disk, extract its longest border,
// else compute a very simple cut to make it homeomorphic to a disk.
// Return the border of this region (empty on error)
//
// CAUTION: this cutting algorithm is very naive. Write your own!
static Seam cut_mesh(Parameterization_polyhedron_adaptor& mesh_adaptor)
{
    // Helper class to compute genus or extract borders
    typedef CGAL::Parameterization_mesh_feature_extractor<Parameterization_polyhedron_adaptor>
            Mesh_feature_extractor;

    Seam seam;          // returned list

    // Get reference to Polyhedron_3 mesh
    Polyhedron& mesh = mesh_adaptor.get_adapted_mesh();

```

```

// Extract mesh borders and compute genus
Mesh_feature_extractor feature_extractor(mesh_adaptor);
int nb_borders = feature_extractor.get_nb_borders();
int genus = feature_extractor.get_genus();

// If mesh is a topological disk
if (genus == 0 && nb_borders > 0)
{
    // Pick the longest border
    seam = feature_extractor.get_longest_border();
}
else // if mesh is NOT a topological disk, create a virtual cut
{
    const int CUT_LENGTH = 6;

    // Build consecutive halfedges array
    Polyhedron::Halfedge_handle seam_halfedges[CUT_LENGTH];
    seam_halfedges[0] = mesh.halfedges_begin();
    if (seam_halfedges[0] == NULL)
        return seam; // return empty list
    int i;
    for (i=1; i<CUT_LENGTH; i++)
    {
        seam_halfedges[i] = seam_halfedges[i-1]->next()->opposite()->next();
        if (seam_halfedges[i] == NULL)
            return seam; // return empty list
    }

    // Convert halfedges array to two-ways vertices list
    for (i=0; i<CUT_LENGTH; i++)
        seam.push_back(seam_halfedges[i]->vertex());
    for (i=CUT_LENGTH-1; i>=0; i--)
        seam.push_back(seam_halfedges[i]->opposite()->vertex());
}

return seam;
}

// Dump parameterized mesh to an eps file
static bool write_file_eps(const Parameterization_polyhedron_adaptor& mesh_adaptor,
                           const char *pFilename,
                           double scale = 500.0)
{
    const Polyhedron& mesh = mesh_adaptor.get_adapted_mesh();

    std::ofstream out(pFilename);
    if(!out)
        return false;
    CGAL::set_ascii_mode(out);

    // compute bounding box
    double xmin,xmax,ymin,ymax;
    xmin = ymin = xmax = ymax = 0;

```

```

Polyhedron::Halfedge_const_iterator pHalfedge;
for (pHalfedge = mesh.halfedges_begin();
    pHalfedge != mesh.halfedges_end();
    pHalfedge++)
{
    double x1 = scale * mesh_adaptor.info(pHalfedge->prev())->uv().x();
    double y1 = scale * mesh_adaptor.info(pHalfedge->prev())->uv().y();
    double x2 = scale * mesh_adaptor.info(pHalfedge)->uv().x();
    double y2 = scale * mesh_adaptor.info(pHalfedge)->uv().y();
    xmin = std::min(xmin,x1);
    xmin = std::min(xmin,x2);
    xmax = std::max(xmax,x1);
    xmax = std::max(xmax,x2);
    ymax = std::max(ymax,y1);
    ymax = std::max(ymax,y2);
    ymin = std::min(ymin,y1);
    ymin = std::min(ymin,y2);
}

out << "%!PS-Adobe-2.0 EPSF-2.0" << std::endl;
out << "%BoundingBox: " << int(xmin+0.5) << " "
    << int(ymin+0.5) << " "
    << int(xmax+0.5) << " "
    << int(ymax+0.5) << std::endl;
out << "%HiResBoundingBox: " << xmin << " "
    << ymin << " "
    << xmax << " "
    << ymax << std::endl;

out << "%EndComments" << std::endl;
out << "gsave" << std::endl;
out << "0.1 setlinewidth" << std::endl;

// color macros
out << std::endl;
out << "% RGB color command - r g b C" << std::endl;
out << "/C { setrgbcolor } bind def" << std::endl;
out << "/white { 1 1 1 C } bind def" << std::endl;
out << "/black { 0 0 0 C } bind def" << std::endl;

// edge macro -> E
out << std::endl;
out << "% Black stroke - x1 y1 x2 y2 E" << std::endl;
out << "/E {moveto lineto stroke} bind def" << std::endl;
out << "black" << std::endl << std::endl;

// for each halfedge
for (pHalfedge = mesh.halfedges_begin();
    pHalfedge != mesh.halfedges_end();
    pHalfedge++)
{
    double x1 = scale * mesh_adaptor.info(pHalfedge->prev())->uv().x();
    double y1 = scale * mesh_adaptor.info(pHalfedge->prev())->uv().y();
    double x2 = scale * mesh_adaptor.info(pHalfedge)->uv().x();
    double y2 = scale * mesh_adaptor.info(pHalfedge)->uv().y();

```

```

        out << x1 << " " << y1 << " " << x2 << " " << y2 << " E" << std::endl;
    }

    /* Emit EPS trailer. */
    out << "grestore" << std::endl;
    out << std::endl;
    out << "showpage" << std::endl;

    return true;
}

// -----
// main()
// -----

int main(int argc, char * argv[])
{
    std::cerr << "PARAMETERIZATION" << std::endl;
    std::cerr << " Discrete Authalic Parameterization" << std::endl;
    std::cerr << " Square border" << std::endl;
    std::cerr << " TAUCS solver" << std::endl;
    std::cerr << " Very simple cut if model is not a topological disk" << std::endl;
    std::cerr << " Output: EPS" << std::endl;

    //*****
    // decode parameters
    //*****

    if (argc-1 != 2)
    {
        std::cerr << "Usage: " << argv[0] << " input_file.off output_file.eps" << std::endl;
        return(EXIT_FAILURE);
    }

    // File names are:
    const char* input_filename = argv[1];
    const char* output_filename = argv[2];

    //*****
    // Read the mesh
    //*****

    // Read the mesh
    std::ifstream stream(input_filename);
    if(!stream)
    {
        std::cerr << "FATAL ERROR: cannot open file " << input_filename << std::endl;
        return EXIT_FAILURE;
    }
    Polyhedron mesh;
    stream >> mesh;

    //*****

```

```

// Create Polyhedron adaptor
//*****

Parameterization_polyhedron_adaptor mesh_adaptor(mesh);

//*****
// Virtually cut mesh
//*****

// The parameterization methods support only meshes that
// are topological disks => we need to compute a "cutting" of the mesh
// that makes it homeomorphic to a disk
Seam seam = cut_mesh(mesh_adaptor);
if (seam.empty())
{
    std::cerr << "FATAL ERROR: an unexpected error occurred while cutting the shape" << std::endl;
    return EXIT_FAILURE;
}

// Create a second adaptor that virtually "cuts" the mesh following the 'seam' path
typedef CGAL::Parameterization_mesh_patch_3<Parameterization_polyhedron_adaptor>
                                         Mesh_patch_polyhedron;
Mesh_patch_polyhedron mesh_patch(mesh_adaptor, seam.begin(), seam.end());

//*****
// Discrete Authalic Parameterization (square border)
// with TAUCS solver
//*****

// Border parameterizer
typedef CGAL::Square_border_arc_length_parameterizer_3<Mesh_patch_polyhedron>
                                         Border_parameterizer;

// TAUCS solver
typedef CGAL::Taucs_solver_traits<double> Solver;

// Discrete Authalic Parameterization (square border)
// with TAUCS solver
typedef CGAL::Discrete_authalic_parameterizer_3<Mesh_patch_polyhedron,
                                         Border_parameterizer,
                                         Solver> Parameterizer;

Parameterizer::Error_code err = CGAL::parameterize(mesh_patch, Parameterizer());
if (err != Parameterizer::OK)
    std::cerr << "FATAL ERROR: " << Parameterizer::get_error_message(err) << std::endl;

//*****
// Output
//*****

// Write Postscript file
if (err == Parameterizer::OK)
{
    if ( ! write_file_eps(mesh_adaptor, output_filename) )
    {

```

```

        std::cerr << "FATAL ERROR: cannot write file " << output_filename << std::endl;
        return EXIT_FAILURE;
    }
}

return (err == Parameterizer::OK) ? EXIT_SUCCESS : EXIT_FAILURE;
}

#else // CGAL_USE_TAUCS

#include <iostream>
#include <stdlib.h>
#include <stdio.h>

// -----
// Empty main() if TAUCS is not installed
// -----

int main(int argc, char * argv[])
{
    std::cerr << "Skip test as TAUCS is not installed" << std::endl;
    return EXIT_SUCCESS;
}

#endif // CGAL_USE_TAUCS

```

32.7 Complexity and Guarantees

32.7.1 Parameterization Methods and Guarantees

- Fixed boundaries
 - One-to-one mapping

Tutte's theorem guarantees a one-to-one mapping provided that the weights are all positive and the border convex. It is the case for Tutte barycentric mapping and Floater mean value coordinates. It is not always the case for discrete conformal map (cotangents) and discrete authalic parameterization.
 - Non-singularity of the matrix

Geshorgin's theorem guarantees the convergence of the solver if the matrix is diagonal dominant. This is the case with positive weights (Tutte barycentric mapping and Floater mean value coordinates).
- Free boundaries
 - One-to-one mapping

No guarantee is provided by LSCM (both global overlaps and triangle flips can occur).
 - Non-singularity of the matrix

For LSCM, the matrix of the system is the Gramm matrix of a matrix with maximal rank, and is therefore non-singular (Gramm theorem).

32.7.2 Precision

Two algorithms of this package construct the sparse linear system(s) using trigonometric functions, and are thus incompatible with exact arithmetic:

- Floater mean value coordinates
- Circular border parameterization

On the other hand, linear solvers commonly use double precision floating point numbers.

OpenNL's BICGSTAB solver (accessible through the *OpenNL::DefaultLinearSolverTraits*<*COEFFTYPE*, *MATRIX*, *VECTOR*, *SOLVER*> interface) is the only solver supported by this package which computes exact results, when used with an exact arithmetic. This package is intended to be used mainly with a CGAL Cartesian kernel with doubles.

OpenNL's BICGSTAB Solver with an Exact Arithmetic

The BICGSTAB conjugate gradient is in disguise a direct solver. In a nutshell, it computes a vector basis orthogonal with respect to the matrix, and the coordinates of the solution in this vector basis. Each iteration computes one component of the basis and one coordinate, therefore the algorithm converges to the solution in n iterations, where n is the dimension of the matrix. More precisely, it is shown to converge in k iteration, where k is the number of distinct eigenvalues of the matrix.

Solvers with a Floating Point Arithmetic

OpenNL's BICGSTAB example:

When inexact numerical types are used (e.g. doubles), accumulated errors slow down convergence (in practice, it requires approximately $5k$ iterations to converge). The required number of iterations depends on the eigenvalues of the matrix, and these eigenvalues depend on the shape of the triangles. The optimum is when the triangles are equilateral (then the solver converges in less than 10 iterations). The worst case is obtained when the mesh has a large number of skinny triangles (near-singular Jacobian matrix of the triangle). In this case, the spectrum of the matrix is wide (many different eigenvalues), and the solver requires nearly $5n$ iterations to converge.

32.7.3 Algorithmic Complexity

In this package, we focus on piecewise linear mappings onto a planar domain. All surface parameterization methods are based on solving one (or two) sparse linear system(s). The algorithmic complexity is dominated by the resolution of the sparse linear system(s).

OpenNL's BICGSTAB example:

At each iteration, the operation of highest complexity is the product between the sparse-matrix and a vector. The sparse matrix has a fixed number of non-zero coefficients per row, therefore the matrix / vector product has $O(n)$ complexity. Since convergence is reached after k iterations, the complexity is $O(k.n)$ (where k is the number of distinct eigenvalues of the matrix). Therefore, best case complexity is $O(n)$ (equilateral triangles), and worst case complexity is $O(n^2)$ (skinny triangles).

32.8 Software Design

32.8.1 Global Function `parameterize()`

This package's entry point is:

```
// Compute a one-to-one mapping from a 3D triangle surface 'mesh' to a
// 2D circle, using Floater Mean Value Coordinates algorithm.
// A one-to-one mapping is guaranteed.
template <class ParameterizationMesh_3>
typename Parameterizer_traits_3<ParameterizationMesh_3>::Error_code
parameterize(ParameterizationMesh_3& mesh) // 3D mesh, model of ParameterizationMesh_3 concept
{
    Mean_value_coordinates_parameterizer_3<ParameterizationMesh_3> parameterizer;
    return parameterizer.parameterize(mesh);
}

// Compute a one-to-one mapping from a 3D triangle surface 'mesh' to a
// simple 2D domain.
// One-to-one mapping may be guaranteed or not,
// depending on the chosen ParameterizerTraits_3 algorithm.
template <class ParameterizationMesh_3, class ParameterizerTraits_3>
typename Parameterizer_traits_3<ParameterizationMesh_3>::Error_code
parameterize(ParameterizationMesh_3& mesh,          // 3D mesh, model of ParameterizationMesh_3
             ParameterizerTraits_3 parameterizer) // Parameterization method for 'mesh'
{
    return parameterizer.parameterize(mesh);
}
```

You may notice that these global functions simply call the `parameterize()` method of a *ParameterizerTraits_3* object. The purpose of these global functions is:

- to be consistent with other CGAL algorithms that are also provided as global functions, e.g. *CGAL::convex_hull_2()*,
- to provide a default parameterization method (Floater Mean Value Coordinates), which wouldn't be possible with a direct call to an object's method.

You may also wonder why there is not just one *CGAL::parameterize()* function with a default *ParameterizerTraits_3* argument equal to *CGAL::Mean_value_coordinates_parameterizer_3<ParameterizationMesh_3>*. The reason is simply that this is not allowed by the C++ standard (see [C++98], paragraph 14.1/9).

32.8.2 No Common Parameterization Algorithm

ParameterizerTraits_3 models modify the behavior of the global function *CGAL::parameterize()* - hence the *Traits* in the name. On the other hand, *ParameterizerTraits_3* models do not modify the behavior of a common parameterization algorithm - as you might expect.

In this package, we focus on triangulated surfaces that are homeomorphic to a disk and on piecewise linear mappings onto planar domains. A consequence is that the skeleton of all parameterization methods of this package is the same:

- Allocate a sparse linear system $A.X = B$
- Parameterize the mesh border and initialize B
- Parameterize the inner points of the mesh and set A coefficients
- Solve the system

It is tempting to make the parameterization method a traits class that modifies the behavior of a common parameterization algorithm. On the other hand, there are several differences among methods:

- Fixed border methods need to parameterize all border vertices, while free border methods parameterize only two vertices.
- Some methods create symmetric definite positive systems, which may be solved more efficiently than general systems.
- Most parameterization methods use two $\#vertices \times \#vertices$ systems, where Least Squares Conformal Maps uses one $(2 * \#triangles) \times \#vertices$ system.
- Most parameterization methods invert the A matrix, when Least Squares Conformal Maps solves the system in the least squares sense.

Therefore, the software design chosen is:

- Each *ParameterizerTraits_3* model implements its own version of the parameterization algorithm as a `parameterize()` method.
- Each *ParameterizerTraits_3* model has template arguments defining the border parameterization and sparse linear solver to use, with default values adapted to the method.
- Code factorization is achieved using a class hierarchy and (few) virtual methods.

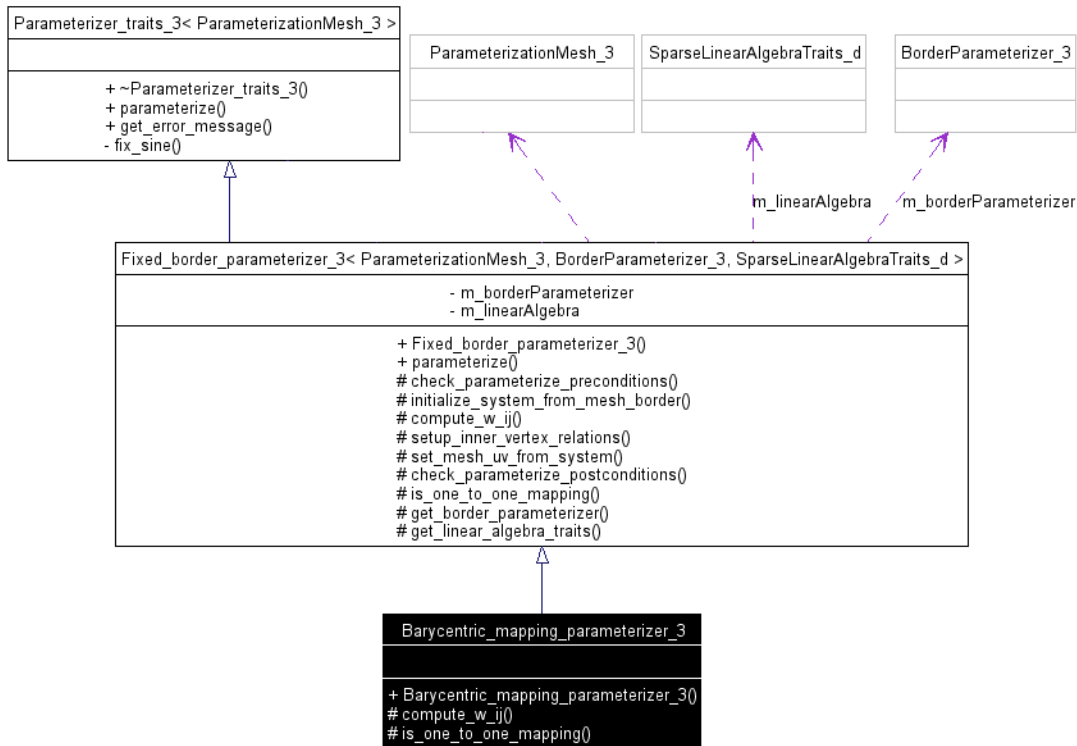


Figure 32.10: A parameterizer UML class diagram (main types and methods only)

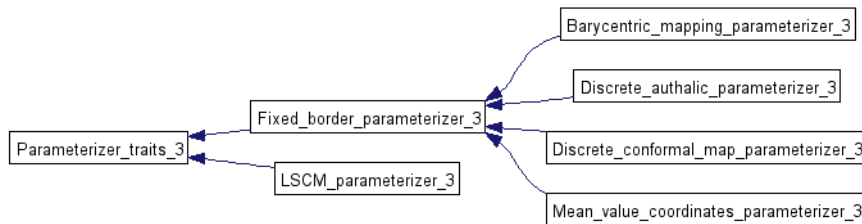


Figure 32.11: Surface parameterizer classes hierarchy

Note: *CGAL::Parameterizer_traits_3<ParameterizationMesh_3>* is the (pure virtual) superclass of all surface parameterization classes.

32.8.3 Fixed_border_parameterizer_3 Class

Linear fixed border parameterization algorithms are very close. They mainly differ by the energy that they try to minimize, i.e. by the value of the w_{ij} coefficient of the A matrix, for v_i and v_j neighbor vertices of the mesh [FH05]. One consequence is that most of the code of the fixed border methods is factorized in the *CGAL::Fixed_border_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>* class.

Subclasses:

- must provide *BorderParameterizer_3* and *SparseLinearAlgebraTraits_d* default template parameters that make sense,
- must implement *compute_w_ij()* to compute $w_{ij} = (i, j)$ coefficient of matrix A for v_j neighbor vertex of v_i ,
- may implement an optimized version of *is_one_to_one_mapping()*.

See `CGAL::Barycentric_mapping_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>` class as an example.

32.8.4 Border Parameterizations

Border Parameterizations are models of the *BorderParameterizer_3* concept. To simplify the implementation, *BorderParameterizer_3* models know only the *ParameterizationMesh_3* mesh class. They do not know the parameterization algorithm or the sparse linear solver used.

32.8.5 ParameterizationMesh_3 and ParameterizationPatchableMesh_3 Concepts

All parameterization methods are templated by the kind of mesh they are applied on. The mesh type must be a model of *ParameterizationMesh_3*.

The purpose of such a model is to:

1. Support several kind of meshes.
2. Hide the implementation of extra fields specific to the parameterization domain (*index*, *u*, *v*, *is_parameterized*).
3. Handle in the mesh type the complexity of *virtually* cutting a mesh to make it homeomorphic to a disk (instead of duplicating this code in each parameterization method).

Two options are possible for 1) and 2):

- Pass to all classes and methods a mesh pointer, a traits class to manipulate it, and accessors to the extra field arrays. This is the choice of the Boost Graph Library with *boost::graph_traits<>* and the property maps.
- Pass to all classes and methods an object that points to the actual mesh and knows how to access to its fields. This is the Adaptor concept [GHJV95].

The current design of this package uses the second option, which is simpler. Of course, we may decide at some point to switch to the first one to reach a deeper integration of CGAL with Boost.

Point 3) is solved by class `CGAL::Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>`, which takes care of *virtually* cutting a patch in a *ParameterizationPatchableMesh_3* mesh, to make it appear as a topological disk with a *ParameterizationMesh_3* interface. *ParameterizationPatchableMesh_3* inherits from concept *ParameterizationMesh_3* and adds the ability to support patches and virtual seams.

This mainly means that:

- vertices can be tagged as inside or outside the patch to parameterize,
- the fields specific to parameterizations (*index*, *u*, *v*, *is_parameterized*) can be set *per corner* (which is a more general way of saying *per half-edge*).

32.8.6 SparseLinearAlgebraTraits_d Concept

This package solves sparse linear systems using solvers which are models of *SparseLinearAlgebraTraits_d*.

SparseLinearAlgebraTraits_d is a sub-concept of the *LinearAlgebraTraits_d* concept in *Kernel_d*. The goal is to adapt easily code written for dense matrices to sparse ones, and vice-versa.

32.8.7 Cutting a Mesh

In this package, we focus on triangulated surfaces that are homeomorphic to a disk.

Computing a cutting path that transforms a closed mesh of arbitrary genus into a topological disk is a research topic on its own. This package does not intend to cover this topic at the moment.

32.9 Extending the Package and Reusing Code

32.9.1 Reusing Mesh Adaptors

ParameterizationMesh_3 defines a concept to access to a general polyhedral mesh. It is optimized for the *Surface_mesh_parameterization* package only in the sense that it defines the accessors to fields specific to the parameterization domain (*index*, *u*, *v*, *is_parameterized*).

It may be easily generalized.

32.9.2 Reusing Sparse Linear Algebra

The *SparseLinearAlgebraTraits_d* concept and the traits classes for OpenNL and TAUCS are independent of the rest of the *Surface_mesh_parameterization* package, and may be reused by CGAL developers for other purposes.

32.9.3 Adding New Parameterization Methods

Implementing a new fixed border linear parameterization is easy. Most of the code of the fixed border methods is factorized in the *CGAL::Fixed_border_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>* class. Subclasses must mainly implement a *compute_w_ij()* method which computes each $w_{ij} = (i, j)$ coefficient of the matrix *A* for v_j neighboring vertices of v_i .

Although implementing a new free border linear parameterization method is more challenging, the Least Squares Conformal Maps parameterization method provides a good starting point.

Implementing *non* linear parameterizations is a natural extension to this package, although only the mesh adaptors can be reused.

32.9.4 Adding New Border Parameterization Methods

Implementing a new border parameterization method is easy. Square, circular and two-points border parameterizations are good starting points.

32.9.5 Mesh Cutting

Obviously, this package would benefit of having robust algorithms which transform arbitrary meshes into topological disks.

Planar Parameterization of Triangulated Surface Meshes

Reference Manual

Laurent Saboret, Pierre Alliez and Bruno Lévy

Parameterizing a surface amounts to finding a one-to-one mapping from a suitable domain to the surface. A good mapping is the one which minimizes either angle or area distortions in some sense. In this package, we focus on triangulated surfaces that are homeomorphic to a disk and on piecewise linear mappings into a planar domain.

32.10 Classified Reference Pages

Main Function

CGAL::parameterize page [1979](#)

Concepts

ParameterizerTraits_3 page [1981](#)

BorderParameterizer_3 page [1936](#)

ParameterizationMesh_3 page [1958](#)

ParameterizationPatchableMesh_3 page [1962](#)

SparseLinearAlgebraTraits_d page [1986](#)

Surface Parameterization Methods

This CGAL package implements some of the state-of-the-art parameterization methods:

- Fixed border:
 - Tutte Barycentric Mapping [[Tut63](#)]. One-to-one mapping is guaranteed for convex border.
 - Floater Mean Value Coordinates [[Flo03a](#)]. One-to-one mapping is guaranteed for convex border.

- Discrete Conformal Map [EDD⁺95]. Conditionally guaranteed if all weights are positive and border is convex.
- Discrete Authalic parameterization [DMA02]. Conditionally guaranteed if all weights are positive and border is convex.
- Free border:
 - Least Squares Conformal Maps [LPRM02].

<i>CGAL::Parameterizer_traits_3<ParameterizationMesh_3></i>	page 1983
<i>CGAL::Fixed_border_parameterizer_3<ParameterizationMesh_3, SparseLinearAlgebraTraits_d></i>	<i>BorderParameterizer_3,</i> page 1947
<i>CGAL::Barycentric_mapping_parameterizer_3<ParameterizationMesh_3, SparseLinearAlgebraTraits_d></i>	<i>BorderParameterizer_3,</i> page 1934
<i>CGAL::Discrete_authalic_parameterizer_3<ParameterizationMesh_3, SparseLinearAlgebraTraits_d></i>	<i>BorderParameterizer_3,</i> page 1943
<i>CGAL::Discrete_conformal_map_parameterizer_3<ParameterizationMesh_3, SparseLinearAlgebraTraits_d></i>	<i>BorderParameterizer_3,</i> page 1945
<i>CGAL::LSCM_parameterizer_3<ParameterizationMesh_3, SparseLinearAlgebraTraits_d></i>	<i>BorderParameterizer_3,</i> page 1951
<i>CGAL::Mean_value_coordinates_parameterizer_3<ParameterizationMesh_3, SparseLinearAlgebraTraits_d></i>	<i>BorderParameterizer_3,</i> page 1956

Border Parameterization Methods

Border parameterization methods define a set of constraints (a constraint specifies two (u,v) coordinates for each instance of a vertex along the border).

This package implements all common border parameterization methods:

- For fixed border methods:
 - the user can select a border parameterization among two common methods: uniform or arc-length parameterizations.
 - one convex shape specified by:
 - * one shape among a set of standard ones (circle, square).
- For free border methods: at least two constraints (the pinned vertices).

<i>CGAL::Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3></i>	page 1937
<i>CGAL::Circular_border_uniform_parameterizer_3<ParameterizationMesh_3></i>	page 1941
<i>CGAL::Square_border_arc_length_parameterizer_3<ParameterizationMesh_3></i>	page 1987
<i>CGAL::Square_border_uniform_parameterizer_3<ParameterizationMesh_3></i>	page 1991
<i>CGAL::Two_vertices_parameterizer_3<ParameterizationMesh_3></i>	page 2001

Mesh

The general definition of input meshes handled *directly* by *CGAL::parameterize()* is:

- Model of *ParameterizationMesh_3*.
- Triangulated.
- 2-manifold.
- Oriented.
- Homeomorphic to a disc (may have holes).

This package provides a model of the *ParameterizationMesh_3* concept to access *CGAL::Polyhedron_3<Traits>* :
CGAL::Parameterization_polyhedron_adaptor_3<Polyhedron_3_>

Fortunately, the meshes supported *indirectly* by the package can be of any genus and have any number of connected components. If it is not a topological disc, the input mesh has to come with a description of a cutting path (an oriented list of vertices) which is the border of a topological disc. If no cutting path is given as input, we assume that the surface border is the longest border already in the input mesh (the other borders will be considered as holes).

The *CGAL::Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>* class is responsible for *virtually* cutting a patch in a *ParameterizationPatchableMesh_3* mesh. The resulting patch is a topological disk (if the input cutting path is correct) and provides a *ParameterizationMesh_3* interface. It can be used as parameter of *CGAL::parameterize()*.

Note that this way the user is responsible for cutting a closed mesh of arbitrary genus (even a topological disc with an intricate seam cut), as long as this condition is fulfilled.

The package provides an interface with *CGAL::Polyhedron_3<Traits>*:
CGAL::Parameterization_polyhedron_adaptor_3<Polyhedron_3_> page [1972](#)

Output

A (u, v) pair is computed for each inner vertex (i.e. its halfedges share the same (u, v) pair), while a (u, v) pair is computed for each border halfedge. The user has to iterate over the mesh halfedges to get the result.

Sparse Linear Algebra

Since parameterizing meshes requires efficient representation of sparse matrices and efficient iterative or direct linear solvers, we provide an interface to several sparse linear solvers:

- OpenNL (Bruno Lévy) is shipped with CGAL. This is the default solver.
- TAUCS is a state-of-the-art direct solver for sparse symmetric matrices. It also includes an out-of-core general solver.

OpenNL::DefaultLinearSolverTraits<COEFFTYPE, MATRIX, VECTOR, SOLVER> in OpenNL package
OpenNL::SymmetricLinearSolverTraits<COEFFTYPE, MATRIX, VECTOR, SOLVER> in OpenNL package
CGAL::Taucs_solver_traits<T> page [1995](#)
CGAL::Taucs_symmetric_solver_traits<T> page [1998](#)

Helper Classes

CGAL::Parameterization_mesh_feature_extractor<*ParameterizationMesh_3*> page 1965

Assertions

The assertion flags for the package use *SURFACE_MESH_PARAMETERIZATION* in their names (e.g., *CGAL_SURFACE_MESH_PARAMETERIZATION_NO_ASSERTIONS*).

For *fixed* border parameterizations:

- Preconditions:
 - check that the border is mapped onto a convex polygon.
 - check that the input mesh is triangular (expensive check).
 - check that the input mesh is a surface with one connected component (expensive check).
- Postconditions:
 - check one-to-one mapping.

For *free* border parameterizations:

- Preconditions:
 - check that the input mesh is triangular (expensive check).
 - check that the input mesh is a surface with one connected component (expensive check).
- Postconditions:
 - check one-to-one mapping.

Expensive checking is off by default. It can be enabled by defining *CGAL_SURFACE_MESH_PARAMETERIZATION_CHECK_EXPENSIVE*.

32.11 Alphabetical List of Reference Pages

<i>Barycentric_mapping_parameterizer_3</i> < <i>ParameterizationMesh_3</i> , <i>SparseLinearAlgebraTraits_d</i> >	<i>BorderParameterizer_3</i> , page 1934
<i>BorderParameterizer_3</i>	page 1936
<i>Circular_border_arc_length_parameterizer_3</i> < <i>ParameterizationMesh_3</i> >	page 1937
<i>Circular_border_parameterizer_3</i> < <i>ParameterizationMesh_3</i> >	page 1939
<i>Circular_border_uniform_parameterizer_3</i> < <i>ParameterizationMesh_3</i> >	page 1941
<i>Discrete_authalic_parameterizer_3</i> < <i>ParameterizationMesh_3</i> , <i>SparseLinearAlgebraTraits_d</i> >	<i>BorderParameterizer_3</i> , page 1943
<i>Discrete_conformal_map_parameterizer_3</i> < <i>ParameterizationMesh_3</i> , <i>SparseLinearAlgebraTraits_d</i> >	<i>BorderParameterizer_3</i> , page 1945
<i>Fixed_border_parameterizer_3</i> < <i>ParameterizationMesh_3</i> , <i>SparseLinearAlgebraTraits_d</i> >	<i>BorderParameterizer_3</i> , page 1947

<i>LSCM_parameterizer_3</i> < <i>ParameterizationMesh_3</i> , <i>BorderParameterizer_3</i> , <i>SparseLinearAlgebraTraits_d</i> >	
page	1951
<i>Matrix</i>	page 1954
<i>Mean_value_coordinates_parameterizer_3</i> < <i>ParameterizationMesh_3</i> , <i>BorderParameterizer_3</i> , <i>SparseLinearAlgebraTraits_d</i> >	page 1956
<i>ParameterizationMesh_3</i>	page 1958
<i>ParameterizationPatchableMesh_3</i>	page 1962
<i>Parameterization_mesh_feature_extractor</i> < <i>ParameterizationMesh_3</i> >	page 1965
<i>Parameterization_mesh_patch_3</i> < <i>ParameterizationPatchableMesh_3</i> >	page 1967
<i>Parameterization_polyhedron_adaptor_3</i> < <i>Polyhedron_3</i> >	page 1972
<i>ParameterizerTraits_3</i>	page 1981
<i>Parameterizer_traits_3</i> < <i>ParameterizationMesh_3</i> >	page 1983
<i>parameterize</i>	page 1979
<i>SparseLinearAlgebraTraits_d</i>	page 1986
<i>Square_border_arc_length_parameterizer_3</i> < <i>ParameterizationMesh_3</i> >	page 1987
<i>Square_border_parameterizer_3</i> < <i>ParameterizationMesh_3</i> >	page 1989
<i>Square_border_uniform_parameterizer_3</i> < <i>ParameterizationMesh_3</i> >	page 1991
<i>Taucs_matrix</i> < <i>T</i> >	page 1993
<i>Taucs_solver_traits</i> < <i>T</i> >	page 1995
<i>Taucs_symmetric_matrix</i> < <i>T</i> >	page 1997
<i>Taucs_symmetric_solver_traits</i> < <i>T</i> >	page 1998
<i>Taucs_vector</i> < <i>T</i> >	page 2000
<i>Two_vertices_parameterizer_3</i> < <i>ParameterizationMesh_3</i> >	page 2001
<i>Vector</i>	page 2003

CGAL::Barycentric_mapping_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>

Definition

The class *Barycentric_mapping_parameterizer_3* implements Tutte Barycentric Mapping algorithm [Tut63]. This algorithm is also called *Tutte Uniform Weights* by other authors.

One-to-one mapping is guaranteed if the surface's border is mapped to a convex polygon.

This class is a Strategy [GHJV95] called by the main parameterization algorithm *Fixed_border_parameterizer_3::parameterize()*. *Barycentric_mapping_parameterizer_3*:

- provides default *BorderParameterizer_3* and *SparseLinearAlgebraTraits_d* template parameters that make sense.
- implements *compute_w_ij()* to compute $w_{ij} = (i,j)$ coefficient of matrix A for j neighbor vertex of i based on Tutte Barycentric Mapping method.
- implements an optimized version of *is_one_to_one_mapping()*.

```
#include <CGAL/Barycentric_mapping_parameterizer_3.h>
```

Is Model for the Concepts

Model of the *ParameterizerTraits_3* concept.

Design Pattern

Barycentric_mapping_parameterizer_3<*ParameterizationMesh_3*, ...> class is a Strategy [GHJV95]: it implements a strategy of surface parameterization for models of *ParameterizationMesh_3*.

Parameters

The full template declaration is:

```
template<
class ParameterizationMesh_3,
class BorderParameterizer_3 = Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3>,
class SparseLinearAlgebraTraits_d = OpenNL::DefaultLinearSolverTraits<typename ParameterizationMesh_3::NT>>
class Barycentric_mapping_parameterizer_3;
```

Parameters: *ParameterizationMesh_3* 3D surface mesh.

BorderParameterizer_3 Strategy to parameterize the surface border.

SparseLinearAlgebraTraits_d Traits class to solve a sparse linear system. Note: the system is NOT symmetric because *Fixed_border_parameterizer_3* does not remove (yet) border vertices from the system.

Creation

Barycentric_mapping_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d> *param*(*Border_param* *border_param* = *Border_param*()),

LA sparse_la = *Sparse_LA*()

Constructor.

Parameters: *border_param* Object that maps the surface's border to 2D space.

sparse_la Traits object to access a sparse linear system.

Operations

virtual NT *param.compute_w_ij*(*Adaptor mesh*,
Vertex_const_handle main_vertex_v_i,
Vertex_around_vertex_const_circulator neighbor_vertex_v_j)

Compute $w_{ij} = (i,j)$ coefficient of matrix A for j neighbor vertex of i. Tutte Barycentric Mapping algorithm is the most simple one: $w_{ij} = 1$ for j neighbor vertex of i.

virtual bool *param.is_one_to_one_mapping*(*Adaptor mesh*, *Matrix A*, *Vector Bu*, *Vector Bv*)

Check if 3D \rightarrow 2D mapping is one-to-one. Theorem: one-to-one mapping is guaranteed if all w_{ij} coefficients are > 0 (for j vertex neighbor of i) and if the surface border is mapped onto a 2D convex polygon. All w_{ij} coefficients = 1 (for j vertex neighbor of i), thus mapping is guaranteed if the surface border is mapped onto a 2D convex polygon.

See Also

CGAL::Parameterizer_traits_3<*ParameterizationMesh_3*>.....page [1983](#)
CGAL::Fixed_border_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d>.....page [1947](#)
CGAL::Discrete_authalic_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d>.....page [1943](#)
CGAL::Discrete_conformal_map_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d>.....page [1945](#)
CGAL::LSCM_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d>.....page [1951](#)
CGAL::Mean_value_coordinates_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d>.....page [1956](#)

BorderParameterizer_3

Definition

BorderParameterizer_3 is a concept of class that parameterizes a given type of mesh, 'Adaptor', which is a model of the *ParameterizationMesh_3* concept.

Implementation note: To simplify the implementation, *BorderParameterizer_3* models know only the *ParameterizationMesh_3* class. They do not know the parameterization algorithm requirements or the kind of sparse linear system used.

Design Pattern

BorderParameterizer_3 models are Strategies [GHJV95]: they implement a strategy of border parameterization for models of *ParameterizationMesh_3*.

Types

BorderParameterizer_3::Adaptor
BorderParameterizer_3::Error_code

Export *ParameterizationMesh_3* template parameter.
 The various errors detected by this package.

Creation

Construction and destruction are undefined.

Operations

Error_code *bp.parameterize_border(Adaptor& mesh)*

Assign to mesh's border vertices a 2D position (ie a (u,v) pair) on border's shape. Mark them as *parameterized*. Return false on error.

bool *bp.is_border_convex()*

Indicate if border's shape is convex.

Has Models

CGAL::Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3> page [1937](#)
CGAL::Circular_border_uniform_parameterizer_3<ParameterizationMesh_3> page [1941](#)
CGAL::Square_border_arc_length_parameterizer_3<ParameterizationMesh_3> page [1987](#)
CGAL::Square_border_uniform_parameterizer_3<ParameterizationMesh_3> page [1991](#)
CGAL::Two_vertices_parameterizer_3<ParameterizationMesh_3> page [2001](#)

See Also

ParameterizerTraits_3 page [1981](#)
ParameterizationMesh_3 page [1958](#)

CGAL::Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3>

Definition

Circular_border_arc_length_parameterizer_3 is the default border parameterizer for fixed border parameterization methods.

This class parameterizes the border of a 3D surface onto a circle, with an arc-length parameterization: (u,v) values are proportional to the length of border edges. *Circular_border_parameterizer_3* implements most of the border parameterization algorithm. This class implements only *compute_edge_length()* to compute a segment's length.

```
#include <CGAL/Circular_border_parameterizer_3.h>
```

Is Model for the Concepts

Model of the *BorderParameterizer_3* concept.

Design Pattern

BorderParameterizer_3 models are Strategies [GHJV95]: they implement a strategy of border parameterization for models of *ParameterizationMesh_3*

Parameters

The full template declaration is:

```
template<
class ParameterizationMesh_3>
class Circular_border_arc_length_parameterizer_3;
```

Types

Creation

```
Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3> bp;
```

default constructor.

Operations

```
virtual double bp.compute_edge_length( Adaptor mesh,
                                       Vertex_const_handle source,
                                       Vertex_const_handle target)
```

Compute the length of an edge. Arc-length border parameterization: (u,v) values are proportional to the length of border edges.

See Also

<i>CGAL::Circular_border_uniform_parameterizer_3<ParameterizationMesh_3></i>	page 1941
<i>CGAL::Square_border_arc_length_parameterizer_3<ParameterizationMesh_3></i>	page 1987
<i>CGAL::Square_border_uniform_parameterizer_3<ParameterizationMesh_3></i>	page 1991
<i>CGAL::Two_vertices_parameterizer_3<ParameterizationMesh_3></i>	page 2001

Example

See *Taucs_parameterization.C* example.

CGAL::Circular_border_parameterizer_3<ParameterizationMesh_3>

Definition

This is the base class of strategies that parameterize the border of a 3D surface onto a circle. *Circular_border_parameterizer_3* is a pure virtual class, thus cannot be instantiated. It implements most of the algorithm. Sub-classes just have to implement *compute_edge_length()* to compute a segment's length.

Implementation note: To simplify the implementation, *BorderParameterizer_3* models know only the *ParameterizationMesh_3* class. They do not know the parameterization algorithm requirements or the kind of sparse linear system used.

```
#include <CGAL/Circular_border_parameterizer_3.h>
```

Is Model for the Concepts

Model of the *BorderParameterizer_3* concept (although you cannot instantiate this class).

Design Pattern

BorderParameterizer_3 models are Strategies [GHJV95]: they implement a strategy of border parameterization for models of *ParameterizationMesh_3*.

Parameters

The full template declaration is:

```
template<
class ParameterizationMesh_3>
class Circular_border_parameterizer_3;
```

Types

Circular_border_parameterizer_3<ParameterizationMesh_3>::Adaptor

Export *ParameterizationMesh_3* template parameter.

Creation

Circular_border_parameterizer_3<ParameterizationMesh_3> bp;

default constructor.

Operations

Parameterizer_traits_3<Adaptor>::Error_code

bp.parameterize_border(Adaptor& mesh)

Assign to mesh's border vertices a 2D position (ie a (u,v) pair) on border's shape. Mark them as *parameterized*.

bool

bp.is_border_convex()

Indicate if border's shape is convex.

virtual double

*bp.compute_edge_length(Adaptor mesh,
Vertex_const_handle source,
Vertex_const_handle target)*

Compute the length of an edge.

See Also

CGAL::Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3> page [1937](#)

CGAL::Circular_border_uniform_parameterizer_3<ParameterizationMesh_3> page [1941](#)

CGAL::Circular_border_uniform_parameterizer_3<ParameterizationMesh_3>

Definition

This class parameterizes the border of a 3D surface onto a circle in a uniform manner: points are equally spaced. *Circular_border_parameterizer_3* implements most of the border parameterization algorithm. This class implements only *compute_edge_length()* to compute a segment's length.

```
#include <CGAL/Circular_border_parameterizer_3.h>
```

Is Model for the Concepts

Model of the *BorderParameterizer_3* concept.

Design Pattern

BorderParameterizer_3 models are Strategies [GHJV95]: they implement a strategy of border parameterization for models of *ParameterizationMesh_3*

Parameters

The full template declaration is:

```
template<
class ParameterizationMesh_3>
class Circular_border_uniform_parameterizer_3;
```

Types

Creation

```
Circular_border_uniform_parameterizer_3<ParameterizationMesh_3> bp;
```

default constructor.

Operations

```
virtual double bp.compute_edge_length( Adaptor mesh,
                                       Vertex_const_handle source,
                                       Vertex_const_handle target)
```

Compute the length of an edge. Uniform border parameterization: points are equally spaced.

See Also

<i>CGAL::Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3></i>	page 1937
<i>CGAL::Square_border_arc_length_parameterizer_3<ParameterizationMesh_3></i>	page 1987
<i>CGAL::Square_border_uniform_parameterizer_3<ParameterizationMesh_3></i>	page 1991
<i>CGAL::Two_vertices_parameterizer_3<ParameterizationMesh_3></i>	page 2001

CGAL::Discrete_authalic_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>

Definition

The class *Discrete_authalic_parameterizer_3* implements the Discrete Authalic Parameterization algorithm [DMA02]. This method is sometimes called DAP or just *Authalic parameterization*.

DAP is a weak area-preserving parameterization. It is a compromise between area-preserving and angle-preserving.

One-to-one mapping is guaranteed if surface's border is mapped onto a convex polygon.

This class is a Strategy [GHV95] called by the main parameterization algorithm *Fixed_border_parameterizer_3::parameterize()*. *Discrete_authalic_parameterizer_3*:

- provides default *BorderParameterizer_3* and *SparseLinearAlgebraTraits_d* template parameters that make sense.
- implements *compute_w_ij()* to compute $w_{ij} = (i, j)$ coefficient of matrix A for j neighbor vertex of i based on Discrete Authalic Parameterization algorithm.

```
#include <CGAL/Discrete_authalic_parameterizer_3.h>
```

Is Model for the Concepts

Model of the *ParameterizerTraits_3* concept.

Design Pattern

Discrete_authalic_parameterizer_3<ParameterizationMesh_3, ...> class is a Strategy [GHV95]: it implements a strategy of surface parameterization for models of *ParameterizationMesh_3*.

Parameters

The full template declaration is:

```
template<
class ParameterizationMesh_3,
class BorderParameterizer_3 = Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3>,
class SparseLinearAlgebraTraits_d = OpenNL::DefaultLinearSolverTraits<typename ParameterizationMesh_3::NT>>
class Discrete_authalic_parameterizer_3;
```

Types

Creation

Discrete_authalic_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d> param(*Border_param* *border_param* = *Border_param*()),

LA sparse_la = *Sparse_LA*()

Constructor.

Parameters: *border_param* Object that maps the surface's border to 2D space.
sparse_la Traits object to access a sparse linear system.

Operations

virtual NT *param.compute_w_ij*(*Adaptor mesh*,
Vertex_const_handle *main_vertex_v_i*,
Vertex_around_vertex_const_circulator *neighbor_vertex_v_j*)

Compute $w_{ij} = (i, j)$ coefficient of matrix A for j neighbor vertex of i.

See Also

CGAL::Parameterizer_traits_3<*ParameterizationMesh_3*>page [1983](#)
CGAL::Fixed_border_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d>page [1947](#)
CGAL::Barycentric_mapping_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d>page [1934](#)
CGAL::Discrete_conformal_map_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d>page [1945](#)
CGAL::LSCM_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d>page [1951](#)
CGAL::Mean_value_coordinates_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d>page [1956](#)

Example

See *Authalic_parameterization.C* example.

CGAL::Discrete_conformal_map_parameterizer_3< ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>

Definition

The class *Discrete_conformal_map_parameterizer_3* implements the Discrete Conformal Map (DCM) parameterization [EDD⁺95]. This algorithm is also called *Discrete Conformal Parameterization (DCP)*, *Discrete Harmonic Map* or *Fixed Conformal Parameterization* by other authors.

This is a conformal parameterization, i.e. it attempts to preserve angles.

One-to-one mapping is guaranteed if surface's border is mapped onto a convex polygon.

This class is a Strategy [GHJV95] called by the main parameterization algorithm *Fixed_border_parameterizer_3::parameterize()*. *Discrete_conformal_map_parameterizer_3*:

- provides default *BorderParameterizer_3* and *SparseLinearAlgebraTraits_d* template parameters that make sense.
- implements *compute_w_ij()* to compute $w_{ij} = (i, j)$ coefficient of matrix A for j neighbor vertex of i based on Discrete Conformal Map method.

```
#include <CGAL/Discrete_conformal_map_parameterizer_3.h>
```

Is Model for the Concepts

Model of the *ParameterizerTraits_3* concept.

Design Pattern

Discrete_conformal_map_parameterizer_3<*ParameterizationMesh_3*, ...> class is a Strategy [GHJV95]: it implements a strategy of surface parameterization for models of *ParameterizationMesh_3*.

Parameters

The full template declaration is:

```
template<
class ParameterizationMesh_3,
class BorderParameterizer_3 = Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3>,
class SparseLinearAlgebraTraits_d = OpenNL::DefaultLinearSolverTraits<typename ParameterizationMesh_3::NT>>
class Discrete_conformal_map_parameterizer_3;
```

Parameters: *ParameterizationMesh_3* 3D surface mesh.

BorderParameterizer_3 Strategy to parameterize the surface border.

SparseLinearAlgebraTraits_d Traits class to solve a sparse linear system. Note: the system is NOT symmetric because *Fixed_border_parameterizer_3* does not remove (yet) border vertices from the system.

Types

Creation

Discrete_conformal_map_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d> *param*(*Border_param* *border_param* = *Border_param*() ,

LA sparse_la = *Sparse_LA*())

Constructor.

Parameters: *border_param* Object that maps the surface's border to 2D space.

sparse_la Traits object to access a sparse linear system.

Operations

virtual NT *param.compute_w_ij*(*Adaptor mesh*,
Vertex_const_handle *main_vertex_v_i*,
Vertex_around_vertex_const_circulator *neighbor_vertex_v_j*)

Compute $w_{ij} = (i,j)$ coefficient of matrix A for j neighbor vertex of i.

See Also

CGAL::Parameterizer_traits_3<*ParameterizationMesh_3*>page [1983](#)
CGAL::Fixed_border_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d>page [1947](#)
CGAL::Barycentric_mapping_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d>page [1934](#)
CGAL::Discrete_authalic_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d>page [1943](#)
CGAL::LSCM_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d>page [1951](#)
CGAL::Mean_value_coordinates_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d>page [1956](#)

CGAL::Fixed_border_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>

Definition

The class *Fixed_border_parameterizer_3* is the base class of fixed border parameterization methods (Tutte, Floater, ...).

One-to-one mapping is guaranteed if surface's border is mapped onto a convex polygon.

This class is a pure virtual class, thus cannot be instantiated. Anyway, it implements most of the parameterization algorithm *parameterize()*. Subclasses are Strategies [GHJV95] that modify the behavior of this algorithm:

- They provide *BorderParameterizer_3* and *SparseLinearAlgebraTraits_d* template parameters that make sense.
- They implement *compute_w_ij()* to compute $w_{ij} = (i, j)$ coefficient of matrix A for j neighbor vertex of i.
- They may implement an optimized version of *is_one_to_one_mapping()*.

Todo *Fixed_border_parameterizer_3* should remove border vertices from the linear systems in order to have a symmetric definite positive matrix for Tutte Barycentric Mapping and Discrete Conformal Map algorithms.

```
#include <CGAL/Fixed_border_parameterizer_3.h>
```

Is Model for the Concepts

Model of the *ParameterizerTraits_3* concept (although you cannot instantiate this class).

Design Pattern

Fixed_border_parameterizer_3<*ParameterizationMesh_3*, ...> class is a Strategy [GHJV95]: it implements (part of) a strategy of surface parameterization for models of *ParameterizationMesh_3*.

Parameters

The full template declaration is:

```
template<
class ParameterizationMesh_3,
class BorderParameterizer_3 = Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3>,
class SparseLinearAlgebraTraits_d = OpenNL::DefaultLinearSolverTraits<typename ParameterizationMesh_3::NT>>
class Fixed_border_parameterizer_3;
```

Types

Fixed_border_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d>::*Border_param*

Export *BorderParameterizer_3* template parameter.

Fixed_border_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d>::*Sparse_LA*

Export *SparseLinearAlgebraTraits_d* template parameter.

Creation

Fixed_border_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d> param(*Border_param* *border_param* = *Border_param*() ,

LA *sparse_la* = *Sparse_LA*())

Constructor.

Parameters: *border_param* Object that maps the surface's border to 2D space
sparse_la Traits object to access a sparse linear system

Operations

Parameterizer_traits_3<*Adaptor*>::*Error_code*

param.parameterize(*Adaptor*& *mesh*)

Compute a one-to-one mapping from a triangular 3D surface 'mesh' to a piece of the 2D space. The mapping is linear by pieces (linear in each triangle). The result is the (u,v) pair image of each vertex of the 3D surface. Preconditions:

- 'mesh' must be a surface with one connected component.
- 'mesh' must be a triangular mesh.
- the mesh border must be mapped onto a convex polygon.

Parameterizer_traits_3<*Adaptor*>::*Error_code*

param.check_parameterize_preconditions(*Adaptor*& *mesh*)

Check *parameterize*() preconditions:

- 'mesh' must be a surface with one connected component.
- 'mesh' must be a triangular mesh.
- the mesh border must be mapped onto a convex polygon.

void *param.initialize_system_from_mesh_border(Matrix& A,*
Vector& Bu,
Vector& Bv,
Adaptor mesh)

Initialize A, Bu and Bv after border parameterization. Fill the border vertices' lines in both linear systems: $u = constant$ and $v = constant$. Preconditions:

- vertices must be indexed.
- A, Bu and Bv must be allocated.
- border vertices must be parameterized.

virtual NT *param.compute_w_ij(Adaptor mesh,*
Vertex_const_handle main_vertex_v_i,
Vertex_around_vertex_const_circulator neighbor_vertex_v_j)

Compute $w_{ij} = (i, j)$ coefficient of matrix A for j neighbor vertex of i. Implementation note: Subclasses must at least implement *compute_w_ij()*.

Parameterizer_traits_3<Adaptor>::Error_code

param.setup_inner_vertex_relations(Matrix& A,
Vector& Bu,
Vector& Bv,
Adaptor mesh,
Vertex_const_handle vertex)

Compute the line i of matrix A for i inner vertex:

- call *compute_w_ij()* to compute the A coefficient w_{ij} for each neighbor v_j .
- compute $w_{ii} = - \text{sum of } w_{ijs}$.

Preconditions:

- vertices must be indexed.
- vertex i musn't be already parameterized.
- line i of A must contain only zeros.

void *param.set_mesh_uv_from_system(Adaptor& mesh, Vector Xu, Vector Xv)*

Copy Xu and Xv coordinates into the (u,v) pair of each surface vertex.

Parameterizer_traits_3<Adaptor>::Error_code

param.check_parameterize_postconditions(Adaptor mesh,
Matrix A,
Vector Bu,

Vector Bv)

Check parameterize() postconditions:

- 3D -> 2D mapping is one-to-one.

bool *param.is_one_to_one_mapping(Adaptor mesh, Matrix A, Vector Bu, Vector Bv)*

Check if 3D -> 2D mapping is one-to-one. The default implementation checks each normal.

Border_param& *param.get_border_parameterizer()*

Get the object that maps the surface's border onto a 2D space.

Sparse_LA& *param.get_linear_algebra_traits()*

Get the sparse linear algebra (traits object to access the linear system).

See Also

CGAL::Parameterizer_traits_3<ParameterizationMesh_3>page [1983](#)
CGAL::Barycentric_mapping_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>page [1934](#)
CGAL::Discrete_authalic_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>page [1943](#)
CGAL::Discrete_conformal_map_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>page [1945](#)
CGAL::LSCM_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>page [1951](#)
CGAL::Mean_value_coordinates_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>page [1956](#)

CGAL::LSCM_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>

Definition

The class *LSCM_parameterizer_3* implements the Least Squares Conformal Maps (LSCM) parameterization [LPRM02].

This is a conformal parameterization, i.e. it attempts to preserve angles.

This is a free border parameterization. No need to map the surface's border onto a convex polygon (only two pinned vertices are needed to ensure a unique solution), but one-to-one mapping is NOT guaranteed.

```
#include <CGAL/LSCM_parameterizer_3.h>
```

Is Model for the Concepts

Model of the *ParameterizerTraits_3* concept.

Design Pattern

LSCM_parameterizer_3<*ParameterizationMesh_3*, ...> class is a Strategy [GHJV95]: it implements a strategy of surface parameterization for models of *ParameterizationMesh_3*.

Parameters

The full template declaration is:

```
template<
class ParameterizationMesh_3,
class BorderParameterizer_3 = Two_vertices_parameterizer_3<ParameterizationMesh_3>,
class SparseLinearAlgebraTraits_d = OpenNL::SymmetricLinearSolverTraits<typename
ParameterizationMesh_3::NT>>
class LSCM_parameterizer_3;
```

Types

LSCM_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*, *SparseLinearAlgebraTraits_d*>::
Border_param

Export *BorderParameterizer_3* template parameter.

LSCM_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*, *SparseLinearAlgebraTraits_d*>::
Sparse_LA

Export *SparseLinearAlgebraTraits_d* template parameter.

Creation

LSCM_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*, *SparseLinearAlgebraTraits_d*>
param(*Border_param* *border_param* = *Border_param*() ,

Sparse_

LA sparse_la = *Sparse_LA*()

Constructor.

Parameters: *border_param* Object that maps the surface's border to 2D space
sparse_la Traits object to access a sparse linear system

Operations

Parameterizer_traits_3<*Adaptor*>::*Error_code*

param.parameterize(*Adaptor*& *mesh*)

Compute a one-to-one mapping from a triangular 3D surface 'mesh' to a piece of the 2D space. The mapping is linear by pieces (linear in each triangle). The result is the (u,v) pair image of each vertex of the 3D surface. Preconditions:

- 'mesh' must be a surface with one connected component.
- 'mesh' must be a triangular mesh.

Parameterizer_traits_3<*Adaptor*>::*Error_code*

param.check_parameterize_preconditions(*Adaptor*& *mesh*)

Check parameterize() preconditions:

- 'mesh' must be a surface with one connected component.
- 'mesh' must be a triangular mesh.

void

param.initialize_system_from_mesh_border(*LeastSquaresSolver*& *solver*,
Adaptor *mesh*)

Initialize $A \cdot X = B$ linear system after (at least two) border vertices are parameterized. Preconditions:

- vertices must be indexed.
- X and B must be allocated and empty.
- (at least two) border vertices must be parameterized.

void

param.project_triangle(*Point_3* *p0*,
Point_3 *p1*,
Point_3 *p2*,
Point_2& *z0*,
Point_2& *z1*,

Point_2& z2)

Utility for *setup_triangle_relations()*: Computes the coordinates of the vertices of a triangle in a local 2D orthonormal basis of the triangle's plane.

Parameterizer_traits_3<Adaptor>::Error_code

param.setup_triangle_relations(LeastSquaresSolver& solver,
Adaptor mesh,
Facet_const_handle facet)

Create two lines in the linear system per triangle (one for u, one for v). Preconditions:

- vertices must be indexed.

void param.set_mesh_uv_from_system(Adaptor& mesh, LeastSquaresSolver solver)

Copy X coordinates into the (u,v) pair of each vertex.

Parameterizer_traits_3<Adaptor>::Error_code

param.check_parameterize_postconditions(Adaptor mesh,
LeastSquaresSolver solver)

Check *parameterize()* postconditions:

- 3D -> 2D mapping is one-to-one.

bool param.is_one_to_one_mapping(Adaptor mesh, LeastSquaresSolver solver)

Check if 3D -> 2D mapping is one-to-one.

Border_param& param.get_border_parameterizer()

Get the object that maps the surface's border onto a 2D space.

Sparse_LA& param.get_linear_algebra_traits()

Get the sparse linear algebra (traits object to access the linear system).

See Also

CGAL::Parameterizer_traits_3<ParameterizationMesh_3>page [1983](#)
CGAL::Fixed_border_parameterizer_3<ParameterizationMesh_3,
SparseLinearAlgebraTraits_d>page [1947](#)
CGAL::Barycentric_mapping_parameterizer_3<ParameterizationMesh_3,
SparseLinearAlgebraTraits_d>page [1934](#)
CGAL::Discrete_authalic_parameterizer_3<ParameterizationMesh_3,
SparseLinearAlgebraTraits_d>page [1943](#)
CGAL::Discrete_conformal_map_parameterizer_3<ParameterizationMesh_3,
SparseLinearAlgebraTraits_d>page [1945](#)
CGAL::Mean_value_coordinates_parameterizer_3<ParameterizationMesh_3,
SparseLinearAlgebraTraits_d>page [1956](#)

SparseLinearAlgebra_d::Matrix

Definition

SparseLinearAlgebraTraits_d::Matrix is a concept of a sparse matrix class.

Refines

This is a sub-concept of *LinearAlgebraTraits_d::Matrix*.

Types

Matrix::NT

Creation

Matrix M(int dimension);

Create a square matrix initialized with zeros.

Matrix M(int rows, int columns);

Create a rectangular matrix initialized with zeros.

Operations

int

M.row_dimension()

Return the matrix number of rows.

int

M.column_dimension()

Return the matrix number of columns.

NT

M.get_coef(int row, int column)

Read access to a matrix coefficient. Preconditions:

- $0 \leq \text{row} < \text{row_dimension}()$.
- $0 \leq \text{column} < \text{column_dimension}()$.

void

M.add_coef(int row, int column, NT value)

Write access to a matrix coefficient: $a_{ij} \leftarrow a_{ij} + \text{val}$. Preconditions:

- $0 \leq \text{row} < \text{row_dimension}()$.
- $0 \leq \text{column} < \text{column_dimension}()$.

void

M.set_coef(int row, int column, NT value)

Write access to a matrix coefficient. Preconditions:

- $0 \leq \text{row} < \text{row_dimension}()$.
- $0 \leq \text{column} < \text{column_dimension}()$.

Has Models

Taucs_matrix<*T*>

Taucs_symmetric_matrix<*T*>

OpenNL::SparseMatrix<*T*> in OpenNL package

See Also

SparseLinearAlgebraTraits_d page [1986](#)

SparseLinearAlgebraTraits_d::Vector page ??

CGAL::Mean_value_coordinates_parameterizer_3<ParameterizationMesh_3, BorderParameterizer_3, SparseLinearAlgebraTraits_d>

Definition

The class *Mean_value_coordinates_parameterizer_3* implements Floater Mean Value Coordinates parameterization [Flo03a]. This method is sometimes called simply *Floater parameterization*.

This is a conformal parameterization, i.e. it attempts to preserve angles.

One-to-one mapping is guaranteed if the surface's border is mapped to a convex polygon.

This class is a Strategy [GHJV95] called by the main parameterization algorithm *Fixed_border_parameterizer_3::parameterize()*. *Mean_value_coordinates_parameterizer_3*:

- provides default *BorderParameterizer_3* and *SparseLinearAlgebraTraits_d* template parameters that make sense.
- implements *compute_w_ij()* to compute $w_{ij} = (i, j)$ coefficient of matrix A for j neighbor vertex of i based on Floater Mean Value Coordinates parameterization.
- implements an optimized version of *is_one_to_one_mapping()*.

```
#include <CGAL/Mean_value_coordinates_parameterizer_3.h>
```

Is Model for the Concepts

Model of the *ParameterizerTraits_3* concept.

Design Pattern

Mean_value_coordinates_parameterizer_3<*ParameterizationMesh_3*, ...>class is a Strategy [GHJV95]: it implements a strategy of surface parameterization for models of *ParameterizationMesh_3*.

Parameters

The full template declaration is:

```
template<
class ParameterizationMesh_3,
class BorderParameterizer_3 = Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3>,
class SparseLinearAlgebraTraits_d = OpenNL::DefaultLinearSolverTraits<typename ParameterizationMesh_3::NT>>
class Mean_value_coordinates_parameterizer_3;
```

Types

Creation

Mean_value_coordinates_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d> *param*(*Border_param* *border_param* = *Border_param*() ,

LA sparse_la = *Sparse_LA*())

Constructor.

Parameters: *border_param* Object that maps the surface's border to 2D space.

sparse_la Traits object to access a sparse linear system.

Operations

virtual NT *param.compute_w_ij*(*Adaptor mesh*,
Vertex_const_handle main_vertex_v_i,
Vertex_around_vertex_const_circulator neighbor_vertex_v_j)

Compute $w_{ij} = (i, j)$ coefficient of matrix A for j neighbor vertex of i.

virtual bool *param.is_one_to_one_mapping*(*Adaptor mesh*, *Matrix A*, *Vector Bu*, *Vector Bv*)

Check if 3D \rightarrow 2D mapping is one-to-one. Theorem: one-to-one mapping is guaranteed if all w_{ij} coefficients are > 0 (for j vertex neighbor of i) and if the surface border is mapped onto a 2D convex polygon. Floater formula above implies that $w_{ij} > 0$ (for j vertex neighbor of i), thus mapping is guaranteed if the surface border is mapped onto a 2D convex polygon.

See Also

CGAL::Parameterizer_traits_3<*ParameterizationMesh_3*>page [1983](#)
CGAL::Fixed_border_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d>page [1947](#)
CGAL::Barycentric_mapping_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d>page [1934](#)
CGAL::Discrete_authalic_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d>page [1943](#)
CGAL::Discrete_conformal_map_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d>page [1945](#)
CGAL::LSCM_parameterizer_3<*ParameterizationMesh_3*, *BorderParameterizer_3*,
SparseLinearAlgebraTraits_d>page [1951](#)

Example

See *Simple_parameterization.C* example.

ParameterizationMesh_3

Definition

ParameterizationMesh_3 is a concept for a 3D surface mesh. Its main purpose is to allow the parameterization methods to access meshes in a uniform manner.

A *ParameterizationMesh_3* surface consists of vertices, facets and an incidence relation on them. No notion of edge is requested. Vertices represent points in 3d-space. Facets are planar polygons without holes defined by the circular sequence of vertices along their border. The surface itself can have holes. The vertices along the border of a hole are called *border vertices*. A surface is *closed* if it contains no border vertices.

The surface must be an oriented 2-manifold with border vertices, i.e. the neighborhood of each point on the surface is either homeomorphic to a disc or to a half disc, except for vertices where many holes and surfaces with border can join.

ParameterizationMesh_3 defines the types, data and methods that a mesh must implement to allow surface parameterization. Among other things, this concept defines accessors to fields specific to parameterizations methods: `index`, `u`, `v`, `is_parameterized`.

ParameterizationMesh_3 meshes can have any genus, arity or number of components. On the other hand, as parameterization methods deal only with topological disks, *ParameterizationMesh_3* defines an interface oriented towards topological disks.

Design Pattern

ParameterizationMesh_3 is an Adaptor [GHJV95]: it changes the interface of a 3D mesh to match the interface expected by the parameterization methods.

Types

The following mutable handles, iterators, and circulators have appropriate non-mutable counterparts, i.e. *const_handle*, *const_iterator*, and *const_circulator*. The mutable types are assignable to their non-mutable counterparts. Both circulators are assignable to the *Vertex_iterator*. The iterators are assignable to the respective handle types. Wherever the handles appear in function parameter lists, the corresponding iterators can be used as well.

<i>ParameterizationMesh_3:: NT</i>	Number type to represent coordinates.
<i>ParameterizationMesh_3:: Point_2</i>	2D point that represents (u,v) coordinates computed by parameterization methods. Must provide X() and Y() methods.
<i>ParameterizationMesh_3:: Point_3</i>	3D point that represents vertices coordinates. Must provide X() and Y() methods.
<i>ParameterizationMesh_3:: Vector_2</i>	2D vector. Must provide X() and Y() methods.
<i>ParameterizationMesh_3:: Vector_3</i>	3D vector. Must provide X() and Y() methods.
<i>ParameterizationMesh_3:: Facet</i>	Opaque type representing a facet of the 3D mesh. No methods are expected.
<i>ParameterizationMesh_3:: Facet_handle</i>	Handle to a facet. Model of the Handle concept.
<i>ParameterizationMesh_3:: Facet_const_handle</i>	
<i>ParameterizationMesh_3:: Facet_iterator</i>	Iterator over all mesh facets. Model of the ForwardIterator concept.

<i>ParameterizationMesh_3:: Facet_const_iterator</i>	
<i>ParameterizationMesh_3:: Vertex</i>	Opaque type representing a vertex of the 3D mesh. No methods are expected.
<i>ParameterizationMesh_3:: Vertex_handle</i>	Handle to a vertex. Model of the Handle concept.
<i>ParameterizationMesh_3:: Vertex_const_handle</i>	
<i>ParameterizationMesh_3:: Vertex_iterator</i>	Iterator over all vertices of a mesh. Model of the ForwardIterator concept.
<i>ParameterizationMesh_3:: Vertex_const_iterator</i>	
<i>ParameterizationMesh_3:: Border_vertex_iterator</i>	Iterator over vertices of the mesh <i>main border</i> . Model of the ForwardIterator concept.
<i>ParameterizationMesh_3:: Border_vertex_const_iterator</i>	
<i>ParameterizationMesh_3:: Vertex_around_facet_circulator</i>	Counter-clockwise circulator over a facet's vertices. Model of the BidirectionalCirculator concept.
<i>ParameterizationMesh_3:: Vertex_around_facet_const_circulator</i>	
<i>ParameterizationMesh_3:: Vertex_around_vertex_circulator</i>	Clockwise circulator over the vertices incident to a vertex. Model of the BidirectionalCirculator concept.
<i>ParameterizationMesh_3:: Vertex_around_vertex_const_circulator</i>	

Creation

Construction and destruction are undefined.

Operations

The following mutable methods returning a handle, iterator, or circulator have appropriate non-mutable counterpart methods, i.e. *const*, returning a *const_handle*, *const_iterator*, or *const_circulator*.

<i>Vertex_iterator</i>	<i>mesh.mesh_vertices_begin()</i>	Get iterator over first vertex of mesh.
<i>Vertex_const_iterator</i>		
	<i>mesh.mesh_vertices_begin()</i>	
<i>Vertex_iterator</i>	<i>mesh.mesh_vertices_end()</i>	Get iterator over past-the-end vertex of mesh.
<i>Vertex_const_iterator</i>		
	<i>mesh.mesh_vertices_end()</i>	
<i>int</i>	<i>mesh.count_mesh_vertices()</i>	Count the number of vertices of the mesh.
<i>void</i>	<i>mesh.index_mesh_vertices()</i>	Index vertices of the mesh from 0 to <i>count_mesh_vertices()</i> -1.

Border_vertex_iterator

mesh.mesh_main_border_vertices_begin()

Get iterator over first vertex of mesh's *main border*.

Border_vertex_const_iterator

mesh.mesh_main_border_vertices_begin()

Border_vertex_iterator

mesh.mesh_main_border_vertices_end()

Get iterator over past-the-end vertex of mesh's *main border*.

Border_vertex_const_iterator

mesh.mesh_main_border_vertices_end()

std::list<Vertex_handle>

mesh.get_border(Vertex_handle seed_vertex)

Return the border containing *seed_vertex*. Return an empty list if not found.

Facet_iterator

mesh.mesh_facets_begin()

Get iterator over first facet of mesh.

Facet_const_iterator

mesh.mesh_facets_begin()

Facet_iterator

mesh.mesh_facets_end()

Get iterator over past-the-end facet of mesh.

Facet_const_iterator

mesh.mesh_facets_end()

int

mesh.count_mesh_facets()

Count the number of facets of the mesh.

bool

mesh.is_mesh_triangular()

Return true if all mesh's facets are triangles.

int

mesh.count_mesh_halfedges()

Count the number of halfedges of the mesh.

Vertex_around_facet_circulator

mesh.facet_vertices_begin(Facet_handle facet)

Get circulator over facet's vertices.

Vertex_around_facet_const_circulator

mesh.facet_vertices_begin(Facet_const_handle facet)

int

mesh.count_facet_vertices(Facet_const_handle facet)

Count the number of vertices of a facet.

Point_3

mesh.get_vertex_position(Vertex_const_handle vertex)

Get the 3D position of a vertex.

<i>Point_2</i>	<i>mesh.get_vertex_uv(Vertex_const_handle vertex)</i>	Get/set the 2D position (u/v pair) of a vertex. Default value is undefined.
<i>void</i>	<i>mesh.set_vertex_uv(Vertex_handle vertex, Point_2 uv)</i>	
<i>bool</i>	<i>mesh.is_vertex_parameterized(Vertex_const_handle vertex)</i>	Get/set <i>is parameterized</i> field of vertex. Default value is undefined.
<i>void</i>	<i>mesh.set_vertex_parameterized(Vertex_handle vertex, bool parameterized)</i>	
<i>int</i>	<i>mesh.get_vertex_index(Vertex_const_handle vertex)</i>	Get/set vertex index. Default value is undefined.
<i>void</i>	<i>mesh.set_vertex_index(Vertex_handle vertex, int index)</i>	
<i>int</i>	<i>mesh.get_vertex_tag(Vertex_const_handle vertex)</i>	Get/set vertex' all purpose tag. Default value is undefined.
<i>void</i>	<i>mesh.set_vertex_tag(Vertex_handle vertex, int tag)</i>	
<i>bool</i>	<i>mesh.is_vertex_on_border(Vertex_const_handle vertex)</i>	Return true if a vertex belongs to ANY mesh's border.
<i>bool</i>	<i>mesh.is_vertex_on_main_border(Vertex_const_handle vertex)</i>	Return true if a vertex belongs to the UNIQUE mesh's main border.
<i>Vertex_around_vertex_circulator</i>		
	<i>mesh.vertices_around_vertex_begin(Vertex_handle vertex,</i> <i>Vertex_handle start_position = Vertex_handle())</i>	Get circulator over the vertices incident to 'vertex'. 'start_position' defines the optional initial position of the circulator.
<i>Vertex_around_vertex_const_circulator</i>		
	<i>mesh.vertices_around_vertex_begin(Vertex_const_handle vertex,</i> <i>Vertex_const_handle start_position = Vertex_</i> <i>const_handle())</i>	

Has Models

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>
Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>

See Also

ParameterizationPatchableMesh_3 page [1962](#)

ParameterizationPatchableMesh_3

Definition

ParameterizationPatchableMesh_3 inherits from concept *ParameterizationMesh_3*, thus is a concept of a 3D surface mesh.

ParameterizationPatchableMesh_3 adds the ability to support patches and virtual seams. *Patches* are a subset of a 3D mesh. *Virtual seams* are the ability to behave exactly as if the surface was cut following a certain path.

This mainly means that:

- vertices can be tagged as inside or outside the patch to parameterize.
- the fields specific to parameterizations (index, u, v, *is_parameterized*) can be set per *corner* (aka half-edge).

The main purpose of this feature is to allow the *Surface_mesh_parameterization* package to parameterize any 3D surface by decomposing it as a list of topological disks.

Design Pattern

ParameterizationPatchableMesh_3 is an Adaptor [GHJV95]: it changes the interface of a 3D mesh to match the interface expected by class *Parameterization_mesh_patch_3*.

Refines

ParameterizationPatchableMesh_3 inherits from concept *ParameterizationMesh_3*.

In addition to the requirements described in the concept *ParameterizationMesh_3*, *ParameterizationPatchableMesh_3* provides the following:

Types

Creation

Construction and destruction are undefined.

Operations

int *mesh.get_vertex_seaming(Vertex_const_handle vertex)*

Get/set vertex seaming flag. Default value is undefined.

void *mesh.set_vertex_seaming(Vertex_handle vertex, int seaming)*

<i>int</i>	<i>mesh.get_halfedge_seaming(Vertex_const_handle source, Vertex_const_handle target)</i>	
		Get/set oriented edge's seaming flag, ie position of the oriented edge wrt to the UNIQUE main border.
<i>void</i>	<i>mesh.set_halfedge_seaming(Vertex_handle source, Vertex_handle target, int seaming)</i>	
<i>Point_2</i>	<i>mesh.get_corners_uv(Vertex_const_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex)</i>	
		Get/set the 2D position (= (u,v) pair) of corners at the <i>right</i> of the <i>prev_vertex</i> -> <i>vertex</i> -> <i>next_vertex</i> line. Default value is undefined.
<i>void</i>	<i>mesh.set_corners_uv(Vertex_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex, Point_2 uv)</i>	
<i>bool</i>	<i>mesh.are_corners_parameterized(Vertex_const_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex)</i>	
		Get/set <i>is parameterized</i> field of corners at the <i>right</i> of the <i>prev_vertex</i> -> <i>vertex</i> -> <i>next_vertex</i> line. Default value is undefined.
<i>void</i>	<i>mesh.set_corners_parameterized(Vertex_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex, bool parameterized)</i>	
<i>int</i>	<i>mesh.get_corners_index(Vertex_const_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex)</i>	
		Get/set index of corners at the <i>right</i> of the <i>prev_vertex</i> -> <i>vertex</i> -> <i>next_vertex</i> line. Default value is undefined.
<i>void</i>	<i>mesh.set_corners_index(Vertex_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex, int index)</i>	
<i>int</i>	<i>mesh.get_corners_tag(Vertex_const_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex)</i>	
		Get/set all purpose tag of corners at the <i>right</i> of the <i>prev_vertex</i> -> <i>vertex</i> -> <i>next_vertex</i> line. Default value is undefined.
<i>void</i>	<i>mesh.set_corners_tag(Vertex_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex, int tag)</i>	

Has Models

Adaptator for *Polyhedron_3* is provided.

CGAL::Parameterization_polyhedron_adaptor_3<*Polyhedron_3*> page [1972](#)

See Also

ParameterizationMesh_3 page [1958](#)

CGAL::Parameterization_mesh_feature_extractor<ParameterizationMesh_3>

Definition

The class *Parameterization_mesh_feature_extractor* computes features (genus, borders, ...) of a 3D surface, model of the *ParameterizationMesh_3* concept.

```
#include <CGAL/Parameterization_mesh_feature_extractor.h>
```

Parameters

The full template declaration is:

```
template<
class ParameterizationMesh_3>
class Parameterization_mesh_feature_extractor;
```

Types

Parameterization_mesh_feature_extractor<ParameterizationMesh_3>::Adaptor

Export *ParameterizationMesh_3* template parameter.

Parameterization_mesh_feature_extractor<ParameterizationMesh_3>::Border

Type representing a border = STL container of vertex handles.

Parameterization_mesh_feature_extractor<ParameterizationMesh_3>::Skeleton

Type representing the list of all borders of the mesh = STL container of Border elements.

Creation

Parameterization_mesh_feature_extractor<ParameterizationMesh_3> extractor(Adaptor& mesh);

Constructor. CAUTION: This class caches the result of feature extractions => The caller must NOT modify 'mesh' during the *Parameterization_mesh_feature_extractor* life cycle.

Operations

int extractor.get_nb_borders()

Get number of borders.

Skeleton extractor.get_borders()

Get extracted borders. The longest border is the first one.

<i>Border</i>	<i>extractor.get_longest_border()</i>	Get longest border.
<i>int</i>	<i>extractor.get_nb_connex_components()</i>	Get # of connected components.
<i>int</i>	<i>extractor.get_genus()</i>	Get the genus.

See Also

ParameterizationMesh_3 page [1958](#)

Example

See *Mesh_cutting_parameterization.C* example.

CGAL::Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>

Definition

Parameterization_mesh_patch_3 is a Decorator class to *virtually* cut a patch in a *ParameterizationPatchableMesh_3* 3D surface. Only the patch is exported, making the 3D surface look like a topological disk.

The input mesh can be of any genus, but it has to come with a *seam* that describes the border of a topological disc. This border may be an actual border of the mesh or a virtual border.

```
#include <CGAL/Parameterization_mesh_patch_3.h>
```

Is Model for the Concepts

Model of *ParameterizationMesh_3* concept, whose purpose is to allow the *Surface_mesh_parameterization* package to access meshes in a uniform manner.

Design Pattern

Parameterization_mesh_patch_3 is a Decorator [GHJV95]: it changes the behavior of a *ParameterizationPatchableMesh_3* 3D surface while keeping its *ParameterizationMesh_3* interface.

Parameters

The full template declaration is:

```
template<
class ParameterizationPatchableMesh_3>
class Parameterization_mesh_patch_3;
```

Types

The following mutable handles, iterators, and circulators have appropriate non-mutable counterparts, i.e. *const_handle*, *const_iterator*, and *const_circulator*. The mutable types are assignable to their non-mutable counterparts. Both circulators are assignable to the *Vertex_iterator*. The iterators are assignable to the respective handle types. Wherever the handles appear in function parameter lists, the corresponding iterators can be used as well.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>::Adaptor

Export template parameter.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>::NT

Number type to represent coordinates.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>::Point_2

2D point that represents (u,v) coordinates computed by parameterization methods. Must provide X() and Y() methods.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Point_3

3D point that represents vertices coordinates. Must provide X() and Y() methods.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Vector_2

2D vector. Must provide X() and Y() methods.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Vector_3

3D vector. Must provide X() and Y() methods.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Facet

Opaque type representing a facet of the 3D mesh. No methods are expected.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Facet_handle

Handle to a facet. Model of the Handle concept.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Facet_const_handle

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Facet_iterator

Iterator over all mesh facets. Model of the ForwardIterator concept.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Facet_const_iterator

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Vertex

Opaque type representing a vertex of the 3D mesh. No methods are expected.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Vertex_handle

Handle to a vertex. Model of the Handle concept.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Vertex_const_handle

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Vertex_iterator

Iterator over all vertices of a mesh. Model of the ForwardIterator concept.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Vertex_const_iterator

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Border_vertex_iterator

Iterator over vertices of the mesh *main border*. Model of the ForwardIterator concept.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Border_vertex_const_iterator

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Vertex_around_facet_circulator

Counter-clockwise circulator over a facet's vertices. Model of the BidirectionalCirculator concept.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Vertex_around_facet_const_circulator

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Vertex_around_vertex_circulator

Clockwise circulator over the vertices incident to a vertex. Model of the BidirectionalCirculator concept.

Parameterization_mesh_patch_3<ParameterizationPatchableMesh_3>:: Vertex_around_vertex_const_circulator

Creation

Parameterization_mesh_patch_3<*ParameterizationPatchableMesh_3*> *mesh*(*Adaptor*& *mesh*,
InputIterator *first_seam_vertex*,
InputIterator *end_seam_vertex*)

Create a Decorator for an existing *ParameterizationPatchableMesh_3* mesh. The input mesh can be of any genus, but it has to come with a *seam* that describes the border of a topological disc. This border may be an actual border of the mesh or a virtual border. Preconditions:

- *first_seam_vertex* -> *end_seam_vertex* defines the outer seam, ie *Parameterization_mesh_patch_3* will export the *right* of the seam.
- the *seam* is given as a container of *Adaptor::Vertex_handle* elements.

Operations

The following methods returning a mutable handle, iterator, or circulator have appropriate non-mutable counterpart methods, i.e. *const*, returning a *const_handle*, *const_iterator*, or *const_circulator*.

Adaptor& *mesh.get_decorated_mesh()*

Get the decorated mesh.

Adaptor *mesh.get_decorated_mesh()*
Vertex_iterator *mesh.mesh_vertices_begin()*

Get iterator over first vertex of mesh.

Vertex_const_iterator

Vertex_iterator *mesh.mesh_vertices_begin()*
 mesh.mesh_vertices_end()

Get iterator over past-the-end vertex of mesh.

Vertex_const_iterator

int *mesh.mesh_vertices_end()*
 mesh.count_mesh_vertices()

Count the number of vertices of the mesh.

void *mesh.index_mesh_vertices()*

Index vertices of the mesh from 0 to *count_mesh_vertices()*-1.

Border_vertex_iterator

mesh.mesh_main_border_vertices_begin()

Get iterator over first vertex of mesh's main border (aka *seam*).

Border_vertex_const_iterator

mesh.mesh_main_border_vertices_begin()

Border_vertex_iterator

mesh.mesh_main_border_vertices_end()

Get iterator over past-the-end vertex of mesh's main border (aka *seam*).

Border_vertex_const_iterator

mesh.mesh_main_border_vertices_end()

std::list<Vertex_handle>

mesh.get_border(Vertex_handle seed_vertex)

Return the border containing *seed_vertex*. Return an empty list if not found.

Facet_iterator

mesh.mesh_facets_begin()

Get iterator over first facet of mesh.

Facet_const_iterator

mesh.mesh_facets_begin()

Facet_iterator

mesh.mesh_facets_end()

Get iterator over past-the-end facet of mesh.

Facet_const_iterator

mesh.mesh_facets_end()

int

mesh.count_mesh_facets()

Count the number of facets of the mesh.

bool

mesh.is_mesh_triangular()

Return true if all mesh's facets are triangles.

int

mesh.count_mesh_halfedges()

Count the number of halfedges of the mesh.

Vertex_around_facet_circulator

mesh.facet_vertices_begin(Facet_handle facet)

Get circulator over facet's vertices.

Vertex_around_facet_const_circulator

mesh.facet_vertices_begin(Facet_const_handle facet)

int

mesh.count_facet_vertices(Facet_const_handle facet)

Count the number of vertices of a facet.

Point_3

mesh.get_vertex_position(Vertex_const_handle vertex)

Get the 3D position of a vertex.

Point_2

mesh.get_vertex_uv(Vertex_const_handle vertex)

Get/set the 2D position (u/v pair) of a vertex. Default value is undefined.

<i>void</i>	<i>mesh.set_vertex_uv(Vertex_handle vertex, Point_2 uv)</i>	
<i>bool</i>	<i>mesh.is_vertex_parameterized(Vertex_const_handle vertex)</i>	Get/set <i>is parameterized</i> field of vertex. Default value is undefined.
<i>void</i>	<i>mesh.set_vertex_parameterized(Vertex_handle vertex, bool parameterized)</i>	
<i>int</i>	<i>mesh.get_vertex_index(Vertex_const_handle vertex)</i>	Get/set vertex index. Default value is undefined.
<i>void</i>	<i>mesh.set_vertex_index(Vertex_handle vertex, int index)</i>	
<i>int</i>	<i>mesh.get_vertex_tag(Vertex_const_handle vertex)</i>	Get/set vertex' all purpose tag. Default value is undefined.
<i>void</i>	<i>mesh.set_vertex_tag(Vertex_handle vertex, int tag)</i>	
<i>bool</i>	<i>mesh.is_vertex_on_border(Vertex_const_handle vertex)</i>	Return true if a vertex belongs to ANY mesh's border.
<i>bool</i>	<i>mesh.is_vertex_on_main_border(Vertex_const_handle vertex)</i>	Return true if a vertex belongs to the UNIQUE mesh's main border set by the constructor.
<i>Vertex_around_vertex_circulator</i>		
	<i>mesh.vertices_around_vertex_begin(Vertex_handle vertex,</i> <i>Vertex_handle start_position = Vertex_handle())</i>	
		Get circulator over the vertices incident to 'vertex'. 'start_position' defines the optional initial position of the circulator.
<i>Vertex_around_vertex_const_circulator</i>		
	<i>mesh.vertices_around_vertex_begin(Vertex_const_handle vertex,</i> <i>Vertex_const_handle start_position = Vertex_</i> <i>const_handle())</i>	

See Also

CGAL::Parameterization_polyhedron_adaptor_3<Polyhedron_3> page [1972](#)

Example

See *Mesh_cutting_parameterization.C* example.

CGAL::Parameterization_polyhedron_adaptor_3<Polyhedron_3_>

Definition

Parameterization_polyhedron_adaptor_3 is an adaptor class to access to a Polyhedron 3D mesh using the *ParameterizationPatchableMesh_3* interface. Among other things, this concept defines the accessor to the (u,v) values computed by parameterizations methods.

Note that these interfaces are decorators that add *on the fly* the necessary fields to unmodified CGAL data structures (using STL maps). For performance reasons, it is recommended to use CGAL data structures enriched with the proper fields.

A *ParameterizationMesh_3* surface consists of vertices, facets and an incidence relation on them. No notion of edge is requested.

ParameterizationMesh_3 meshes can have any genus, arity or number of components.

It can have any number of borders. Its *main border* will be the mesh's longest border (if there is at least one border).

It has also the ability to support patches and virtual seams. *Patches* are a subset of a 3D mesh. *Virtual seams* are the ability to behave exactly as if the surface was cut following a certain path.

```
#include <CGAL/Parameterization_polyhedron_adaptor_3.h>
```

Is Model for the Concepts

Model of *ParameterizationPatchableMesh_3* concept, whose purpose is to allow the *Surface_mesh_parameterization* package to access meshes in a uniform manner.

Design Pattern

Parameterization_polyhedron_adaptor_3 is an Adaptor [GHJV95]: it changes the Polyhedron interface to match the *ParameterizationPatchableMesh_3* concept.

Parameters

The full template declaration is:

```
template<
class Polyhedron_3_>
class Parameterization_polyhedron_adaptor_3;
```

Types

The following mutable handles, iterators, and circulators have appropriate non-mutable counterparts, i.e. *const_handle*, *const_iterator*, and *const_circulator*. The mutable types are assignable to their non-mutable counterparts. Both circulators are assignable to the *Vertex_iterator*. The iterators are assignable to the respective handle types. Wherever the handles appear in function parameter lists, the corresponding iterators can be used as well.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Polyhedron

Export template parameter.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: NT

Number type to represent coordinates.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Point_2

2D point that represents (u,v) coordinates computed by parameterization methods. Must provide X() and Y() methods.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Point_3

3D point that represents vertices coordinates. Must provide X() and Y() methods.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Vector_2

2D vector. Must provide X() and Y() methods.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Vector_3

3D vector. Must provide X() and Y() methods.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Facet

Opaque type representing a facet of the 3D mesh. No methods are expected.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Facet_handle

Handle to a facet. Model of the Handle concept.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Facet_const_handle

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Facet_iterator

Iterator over all mesh facets. Model of the ForwardIterator concept.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Facet_const_iterator

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Vertex

Opaque type representing a vertex of the 3D mesh. No methods are expected.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Vertex_handle

Handle to a vertex. Model of the Handle concept.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Vertex_const_handle

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Vertex_iterator

Iterator over all vertices of a mesh. Model of the ForwardIterator concept.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Vertex_const_iterator

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Border_vertex_iterator

Iterator over vertices of the mesh *main border*. Model of the ForwardIterator concept.

Parameterization_polyhedron_adaptor_3<Polyhedron_3_>:: Border_vertex_const_iterator

Parameterization_polyhedron_adaptor_3<Polyhedron_3>::Vertex_around_facet_circulator

Counter-clockwise circulator over a facet's vertices. Model of the BidirectionalCirculator concept.

Parameterization_polyhedron_adaptor_3<Polyhedron_3>::Vertex_around_facet_const_circulator

Parameterization_polyhedron_adaptor_3<Polyhedron_3>::Vertex_around_vertex_circulator

Clockwise circulator over the vertices incident to a vertex. Model of the BidirectionalCirculator concept.

Parameterization_polyhedron_adaptor_3<Polyhedron_3>::Vertex_around_vertex_const_circulator

Creation

Parameterization_polyhedron_adaptor_3<Polyhedron_3> mesh(Polyhedron& mesh);

Create an adaptor for an existing *Polyhedron_3* mesh. The input mesh can be of any genus. It can have any number of borders. Its *main border* will be the mesh's longest border (if there is at least one border).

Operations

The following methods returning a mutable handle, iterator, or circulator have appropriate non-mutable counterpart methods, i.e. *const*, returning a *const_handle*, *const_iterator*, or *const_circulator*.

Polyhedron& mesh.get_adapted_mesh()

Get the adapted mesh.

Polyhedron mesh.get_adapted_mesh()

Polyhedron::Halfedge_const_handle

mesh.get_halfedge(Vertex_const_handle source, Vertex_const_handle target)

Get halfedge from source and target vertices. Will assert if such an halfedge doesn't exist.

Polyhedron::Halfedge_handle

mesh.get_halfedge(Vertex_handle source, Vertex_handle target)

const Halfedge_info mesh.info(Halfedge_const_handle halfedge)*

Access to additional info attached to halfedges.

Halfedge_info mesh.info(Halfedge_const_handle halfedge)*

const Vertex_info mesh.info(Vertex_const_handle vertex)*

Access to additional info attached to vertices.

Vertex_info mesh.info(Vertex_const_handle vertex)*

Vertex_iterator mesh.mesh_vertices_begin()

Get iterator over first vertex of mesh.

Vertex_const_iterator

mesh.mesh_vertices_begin()

<i>Vertex_iterator</i>	<i>mesh.mesh_vertices_end()</i>	
		Get iterator over past-the-end vertex of mesh.
<i>Vertex_const_iterator</i>		
	<i>mesh.mesh_vertices_end()</i>	
<i>int</i>	<i>mesh.count_mesh_vertices()</i>	
		Count the number of vertices of the mesh.
<i>void</i>	<i>mesh.index_mesh_vertices()</i>	
		Index vertices of the mesh from 0 to <i>count_mesh_vertices()</i> -1.
<i>Border_vertex_iterator</i>		
	<i>mesh.mesh_main_border_vertices_begin()</i>	
		Get iterator over first vertex of mesh's <i>main border</i> .
<i>Border_vertex_const_iterator</i>		
	<i>mesh.mesh_main_border_vertices_begin()</i>	
<i>Border_vertex_iterator</i>		
	<i>mesh.mesh_main_border_vertices_end()</i>	
		Get iterator over past-the-end vertex of mesh's <i>main border</i> .
<i>Border_vertex_const_iterator</i>		
	<i>mesh.mesh_main_border_vertices_end()</i>	
<i>std::list<Vertex_handle></i>		
	<i>mesh.get_border(Vertex_handle seed_vertex)</i>	
		Return the border containing <i>seed_vertex</i> . Return an empty list if not found.
<i>Facet_iterator</i>	<i>mesh.mesh_facets_begin()</i>	
		Get iterator over first facet of mesh.
<i>Facet_const_iterator</i>	<i>mesh.mesh_facets_begin()</i>	
<i>Facet_iterator</i>	<i>mesh.mesh_facets_end()</i>	
		Get iterator over past-the-end facet of mesh.
<i>Facet_const_iterator</i>	<i>mesh.mesh_facets_end()</i>	
<i>int</i>	<i>mesh.count_mesh_facets()</i>	
		Count the number of facets of the mesh.
<i>bool</i>	<i>mesh.is_mesh_triangular()</i>	
		Return true if all mesh's facets are triangles.
<i>int</i>	<i>mesh.count_mesh_halfedges()</i>	
		Count the number of halfedges of the mesh.

Vertex_around_facet_circulator

mesh.facet_vertices_begin(Facet_handle facet)

Get circulator over facet's vertices.

Vertex_around_facet_const_circulator

int mesh.facet_vertices_begin(Facet_const_handle facet)
mesh.count_facet_vertices(Facet_const_handle facet)

Count the number of vertices of a facet.

Point_3 mesh.get_vertex_position(Vertex_const_handle vertex)

Get the 3D position of a vertex.

Point_2 mesh.get_vertex_uv(Vertex_const_handle vertex)

Get/set the 2D position (u/v pair) of a vertex. Default value is undefined. (stored in halfedges sharing the same vertex).

void mesh.set_vertex_uv(Vertex_handle vertex, Point_2 uv)
bool mesh.is_vertex_parameterized(Vertex_const_handle vertex)

Get/set *is parameterized* field of vertex. Default value is undefined. (stored in halfedges sharing the same vertex).

void mesh.set_vertex_parameterized(Vertex_handle vertex, bool parameterized)
int mesh.get_vertex_index(Vertex_const_handle vertex)

Get/set vertex index. Default value is undefined. (stored in Polyhedron vertex for debugging purpose).

void mesh.set_vertex_index(Vertex_handle vertex, int index)
int mesh.get_vertex_tag(Vertex_const_handle vertex)

Get/set vertex' all purpose tag. Default value is undefined. (stored in halfedges sharing the same vertex).

void mesh.set_vertex_tag(Vertex_handle vertex, int tag)
bool mesh.is_vertex_on_border(Vertex_const_handle vertex)

Return true if a vertex belongs to ANY mesh's border.

bool mesh.is_vertex_on_main_border(Vertex_const_handle vertex)

Return true if a vertex belongs to the UNIQUE mesh's main border, ie the mesh's LONGEST border.

Vertex_around_vertex_circulator

mesh.vertices_around_vertex_begin(Vertex_handle vertex,
Vertex_handle start_position = Vertex_handle())

Get circulator over the vertices incident to 'vertex'. 'start_position' defines the optional initial position of the circulator.

Vertex_around_vertex_const_circulator

mesh.vertices_around_vertex_begin(Vertex_const_handle vertex,
Vertex_const_handle start_position = Vertex_
const_handle())

<i>int</i>	<i>mesh.get_vertex_seaming(Vertex_const_handle vertex)</i>	
		Get/set vertex seaming flag. Default value is undefined.
<i>void</i>	<i>mesh.set_vertex_seaming(Vertex_handle vertex, int seaming)</i>	
<i>int</i>	<i>mesh.get_halfedge_seaming(Vertex_const_handle source, Vertex_const_handle target)</i>	
		Get/set oriented edge's seaming flag, ie position of the oriented edge wrt to the UNIQUE main border.
<i>void</i>	<i>mesh.set_halfedge_seaming(Vertex_handle source, Vertex_handle target, int seaming)</i>	
<i>Point_2</i>	<i>mesh.get_corners_uv(Vertex_const_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex)</i>	
		Get/set the 2D position (= (u,v) pair) of corners at the <i>right</i> of the <i>prev_vertex</i> -> <i>vertex</i> -> <i>next_vertex</i> line. Default value is undefined. (stored in incident halfedges).
<i>void</i>	<i>mesh.set_corners_uv(Vertex_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex, Point_2 uv)</i>	
<i>bool</i>	<i>mesh.are_corners_parameterized(Vertex_const_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex)</i>	
		Get/set <i>is parameterized</i> field of corners at the <i>right</i> of the <i>prev_vertex</i> -> <i>vertex</i> -> <i>next_vertex</i> line. Default value is undefined. (stored in incident halfedges).
<i>void</i>	<i>mesh.set_corners_parameterized(Vertex_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex, bool parameterized)</i>	
<i>int</i>	<i>mesh.get_corners_index(Vertex_const_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex)</i>	
		Get/set index of corners at the <i>right</i> of the <i>prev_vertex</i> -> <i>vertex</i> -> <i>next_vertex</i> line. Default value is undefined. (stored in incident halfedges).
<i>void</i>	<i>mesh.set_corners_index(Vertex_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex, int index)</i>	
<i>int</i>	<i>mesh.get_corners_tag(Vertex_const_handle vertex, Vertex_const_handle prev_vertex, Vertex_const_handle next_vertex)</i>	
		Get/set all purpose tag of corners at the <i>right</i> of the <i>prev_vertex</i> -> <i>vertex</i> -> <i>next_vertex</i> line. Default value is undefined. (stored in incident halfedges).

```
void                mesh.set_corners_tag( Vertex_handle vertex,
                                         Vertex_const_handle prev_vertex,
                                         Vertex_const_handle next_vertex,
                                         int tag)
```

See Also

CGAL::Parameterization_mesh_patch_3<*ParameterizationPatchableMesh_3*>page [1967](#)

Example

See *Simple_parameterization.C* example.

CGAL::parameterize

Definition

CGAL::parameterize() is the main entry-point of the *Surface_mesh_parameterization* package.

It computes a one-to-one mapping from a 3D triangle surface 'mesh' to a simple 2D domain. The mapping is piecewise linear on the triangle mesh. The result is a pair (u,v) of parameter coordinates for each vertex of the input mesh. One-to-one mapping may be guaranteed or not, depending on the chosen *ParameterizerTraits* algorithm.

The *CGAL::parameterize()* function exists in two flavors, to provide a default parameterization algorithm of Floater Mean Value Coordinates.

```
#include <CGAL/parameterize.h>
```

```
Parameterizer_traits_3<ParameterizationMesh_3>::Error_code
```

```
parameterize( ParameterizationMesh_3& mesh)
```

Compute a one-to-one mapping from a 3D triangle surface 'mesh' to a 2D circle, using Floater Mean Value Coordinates algorithm. A one-to-one mapping is guaranteed. The mapping is piecewise linear on the input mesh triangles. The result is a (u,v) pair of parameter coordinates for each vertex of the input mesh. Preconditions:

- 'mesh' must be a surface with one connected component.
- 'mesh' must be a triangular mesh.

Parameters: **mesh** 3D mesh, model of *ParameterizationMesh_3* concept

```
Parameterizer_traits_3<ParameterizationMesh_3>::Error_code
```

```
parameterize( ParameterizationMesh_3& mesh,
               ParameterizerTraits_3 parameterizer)
```

Compute a one-to-one mapping from a 3D triangle surface 'mesh' to a simple 2D domain. The mapping is piecewise linear on the triangle mesh. The result is a pair (u,v) of parameter coordinates for each vertex of the input mesh. One-to-one mapping may be guaranteed or not, depending on the chosen *ParameterizerTraits_3* algorithm. Preconditions:

- 'mesh' must be a surface with one connected component.
- 'mesh' must be a triangular mesh.
- the mesh border must be mapped onto a convex polygon (for fixed border parameterizations).

Parameters: **mesh** 3D mesh, model of *ParameterizationMesh_3*
parameterizer Parameterization method for 'mesh'

Parameters

The full template declarations are:

```
template<
class ParameterizationMesh_3>
Parameterizer_traits_3<ParameterizationMesh_3::Error_code
parameterize (ParameterizationMesh_3 &mesh);
```

```
template<
class ParameterizationMesh_3,
class ParameterizerTraits_3>
Parameterizer_traits_3<ParameterizationMesh_3::Error_code
parameterize (ParameterizationMesh_3 &mesh, ParameterizerTraits_3 parameterizer);
```

See Also

<i>CGAL::Barycentric_mapping_parameterizer_3</i> < <i>ParameterizationMesh_3</i> ,	<i>BorderParameterizer_3</i> ,	
<i>SparseLinearAlgebraTraits_d</i> >		page 1934
<i>CGAL::Discrete_authalic_parameterizer_3</i> < <i>ParameterizationMesh_3</i> ,	<i>BorderParameterizer_3</i> ,	
<i>SparseLinearAlgebraTraits_d</i> >		page 1943
<i>CGAL::Discrete_conformal_map_parameterizer_3</i> < <i>ParameterizationMesh_3</i> ,	<i>BorderParameterizer_3</i> ,	
<i>SparseLinearAlgebraTraits_d</i> >		page 1945
<i>CGAL::LSCM_parameterizer_3</i> < <i>ParameterizationMesh_3</i> ,	<i>BorderParameterizer_3</i> ,	
<i>SparseLinearAlgebraTraits_d</i> >		page 1951
<i>CGAL::Mean_value_coordinates_parameterizer_3</i> < <i>ParameterizationMesh_3</i> ,	<i>BorderParameterizer_3</i> ,	
<i>SparseLinearAlgebraTraits_d</i> >		page 1956

Example

See *Simple_parameterization.C* example.

Implementation

This function simply calls the `parameterize()` method of the parameterization algorithm chosen.

ParameterizerTraits_3

Definition

ParameterizerTraits_3 is a concept of parameterization object for a given type of mesh, 'Adaptor', which is a model of the *ParameterizationMesh_3* concept.

Design Pattern

ParameterizerTraits_3 models are Strategies [GHJV95]: they implement a strategy of surface parameterization for models of *ParameterizationMesh_3*.

Types

ParameterizerTraits_3::Adaptor Export the type of mesh to parameterize.

Constants

```
enum Error_code { OK,
                  ERROR_EMPTY_MESH,
                  ERROR_NON_TRIANGULAR_MESH,
                  ERROR_NO_SURFACE_MESH,
                  ERROR_INVALID_BORDER,
                  ERROR_CANNOT_SOLVE_LINEAR_SYSTEM,
                  ERROR_NO_1_TO_1_MAPPING,
                  ERROR_NOT_ENOUGH_MEMORY,
                  ERROR_WRONG_PARAMETER}
```

List of errors detected by this package.

Enumeration values: **OK** Success.

ERROR_EMPTY_MESH Error: input mesh is empty.

ERROR_NON_TRIANGULAR_MESH Error: input mesh is not triangular.

ERROR_NO_SURFACE_MESH Error: input mesh is not a surface.

ERROR_INVALID_BORDER Error: parameterization requires a convex border.

ERROR_CANNOT_SOLVE_LINEAR_SYSTEM Error: cannot solve linear system.

ERROR_NO_1_TO_1_MAPPING Error: parameterization does not ensure a one-to-one mapping.

ERROR_NOT_ENOUGH_MEMORY Error: not enough memory.

ERROR_WRONG_PARAMETER Error: a method received an unexpected parameter.

Creation

Construction and destruction are undefined.

Operations

Error_code *param.parameterize(Adaptor& mesh)*

Compute a one-to-one mapping from a triangular 3D surface 'mesh' to a piece of the 2D space. The mapping is linear by pieces (linear in each triangle). The result is the (u,v) pair image of each vertex of the 3D surface. Preconditions:

- 'mesh' must be a surface with one connected component and no hole.
- 'mesh' must be a triangular mesh.

Has Models

CGAL::Parameterizer_traits_3<ParameterizationMesh_3> page [1983](#)
CGAL::Fixed_border_parameterizer_3<ParameterizationMesh_3, *BorderParameterizer_3,*
SparseLinearAlgebraTraits_d> page [1947](#)
CGAL::Barycentric_mapping_parameterizer_3<ParameterizationMesh_3, *BorderParameterizer_3,*
SparseLinearAlgebraTraits_d> page [1934](#)
CGAL::Discrete_authalic_parameterizer_3<ParameterizationMesh_3, *BorderParameterizer_3,*
SparseLinearAlgebraTraits_d> page [1943](#)
CGAL::Discrete_conformal_map_parameterizer_3<ParameterizationMesh_3, *BorderParameterizer_3,*
SparseLinearAlgebraTraits_d> page [1945](#)
CGAL::LSCM_parameterizer_3<ParameterizationMesh_3, *BorderParameterizer_3,*
SparseLinearAlgebraTraits_d> page [1951](#)
CGAL::Mean_value_coordinates_parameterizer_3<ParameterizationMesh_3, *BorderParameterizer_3,*
SparseLinearAlgebraTraits_d> page [1956](#)

See Also

ParameterizationMesh_3 page [1958](#)

CGAL::Parameterizer_traits_3<ParameterizationMesh_3>

Definition

The class *Parameterizer_traits_3* is the base class of all parameterization methods. This class is a pure virtual class, thus cannot be instantiated.

This class doesn't do much. Its main goal is to ensure that subclasses will be proper models of the *ParameterizerTraits_3* concept:

- *Parameterizer_traits_3* defines the *Error_code* list of errors detected by this package
- *Parameterizer_traits_3* declares a pure virtual method *parameterize()*

```
#include <CGAL/Parameterizer_traits_3.h>
```

Is Model for the Concepts

Model of the *ParameterizerTraits_3* concept (although you cannot instantiate this class).

Design Pattern

ParameterizerTraits_3 models are Strategies [[GHJV95](#)]: they implement a strategy of surface parameterization for models of *ParameterizationMesh_3*.

Parameters

The full template declaration is:

```
template<
class ParameterizationMesh_3>
class Parameterizer_traits_3;
```

Types

Parameterizer_traits_3<ParameterizationMesh_3>::Adaptor

Export *ParameterizationMesh_3* template parameter.

Constants

```
enum Error_code { OK,
                  ERROR_EMPTY_MESH,
                  ERROR_NON_TRIANGULAR_MESH,
                  ERROR_NO_SURFACE_MESH,
                  ERROR_INVALID_BORDER,
                  ERROR_CANNOT_SOLVE_LINEAR_SYSTEM,
                  ERROR_NO_1_TO_1_MAPPING,
                  ERROR_NOT_ENOUGH_MEMORY,
```

ERROR_WRONG_PARAMETER}

List of errors detected by this package.

Enumeration values: **OK** Success.

ERROR_EMPTY_MESH input mesh is empty

ERROR_NON_TRIANGULAR_MESH input mesh is not triangular

ERROR_NO_SURFACE_MESH input mesh is not a surface

ERROR_INVALID_BORDER parameterization requires a convex border

ERROR_CANNOT_SOLVE_LINEAR_SYSTEM cannot solve linear system

ERROR_NO_1_TO_1_MAPPING parameterization does not ensure a one-to-one mapping

ERROR_NOT_ENOUGH_MEMORY not enough memory

ERROR_WRONG_PARAMETER a method received an unexpected parameter

Creation

Parameterizer_traits_3<ParameterizationMesh_3> *param*;

default constructor.

Operations

virtual Error_code *param.parameterize(Adaptor& mesh)*

Compute a one-to-one mapping from a 3D surface 'mesh' to a piece of the 2D space. The mapping is linear by pieces (linear in each triangle). The result is the (u,v) pair image of each vertex of the 3D surface. Preconditions:

- 'mesh' must be a surface with one connected component.
- 'mesh' must be a triangular mesh.

*static const char** *param.get_error_message(int error_code)*

Get message (in english) corresponding to an error code

Parameters: *error_code* The code returned by *parameterize()*

Returns: The string describing the error code

See Also

CGAL::Fixed_border_parameterizer_3<ParameterizationMesh_3, *BorderParameterizer_3,*
SparseLinearAlgebraTraits_d>page [1947](#)
CGAL::Barycentric_mapping_parameterizer_3<ParameterizationMesh_3, *BorderParameterizer_3,*
SparseLinearAlgebraTraits_d>page [1934](#)
CGAL::Discrete_authalic_parameterizer_3<ParameterizationMesh_3, *BorderParameterizer_3,*
SparseLinearAlgebraTraits_d>page [1943](#)

<i>CGAL::Discrete_conformal_map_parameterizer_3<ParameterizationMesh_3,</i>	<i>BorderParameterizer_3,</i>	
<i>SparseLinearAlgebraTraits_d></i>		page 1945
<i>CGAL::LSCM_parameterizer_3<ParameterizationMesh_3,</i>	<i>BorderParameterizer_3,</i>	
<i>SparseLinearAlgebraTraits_d></i>		page 1951
<i>CGAL::Mean_value_coordinates_parameterizer_3<ParameterizationMesh_3,</i>	<i>BorderParameterizer_3,</i>	
<i>SparseLinearAlgebraTraits_d></i>		page 1956

SparseLinearAlgebraTraits_d

Definition

The concept *SparseLinearAlgebraTraits_d* is used to solve sparse linear systems $A * X = B$.

Refines

This is a sub-concept of *LinearAlgebraTraits_d*.

Types

SparseLinearAlgebraTraits_d::Matrix

SparseLinearAlgebraTraits_d::Vector

SparseLinearAlgebraTraits_d::NT

Creation

SparseLinearAlgebraTraits_d *sparse_LA*; Default constructor.

Operations

bool *sparse_LA.linear_solver(Matrix A, Vector B, Vector& X, NT& D)*

Solve the sparse linear system $A * X = B$. Return true on success. The solution is then $(1/D) * X$. Preconditions:

- *A.row_dimension() == B.dimension()*.
- *A.column_dimension() == X.dimension()*.

Has Models

CGAL::Taucs_solver_traits<T> page [1995](#)

CGAL::Taucs_symmetric_solver_traits<T> page [1998](#)

OpenNL::DefaultLinearSolverTraits<COEFFTYPE, MATRIX, VECTOR, SOLVER> in OpenNL package

OpenNL::SymmetricLinearSolverTraits<COEFFTYPE, MATRIX, VECTOR, SOLVER> in OpenNL package

See Also

SparseLinearAlgebraTraits_d::Matrix page ??

SparseLinearAlgebraTraits_d::Vector page ??

CGAL::Square_border_arc_length_parameterizer_3<ParameterizationMesh_3>

Definition

This class parameterizes the border of a 3D surface onto a square, with an arc-length parameterization: (u,v) values are proportional to the length of border edges.

Square_border_parameterizer_3 implements most of the border parameterization algorithm. This class implements only *compute_edge_length()* to compute a segment's length.

```
#include <CGAL/Square_border_parameterizer_3.h>
```

Is Model for the Concepts

Model of the *BorderParameterizer_3* concept.

Design Pattern

BorderParameterizer_3 models are Strategies [GHJV95]: they implement a strategy of border parameterization for models of *ParameterizationMesh_3*

Parameters

The full template declaration is:

```
template<
class ParameterizationMesh_3>
class Square_border_arc_length_parameterizer_3;
```

Types

Creation

```
Square_border_arc_length_parameterizer_3<ParameterizationMesh_3> bp;
```

default constructor.

Operations

```
virtual double      bp.compute_edge_length( Adaptor mesh,
                                             Vertex_const_handle source,
                                             Vertex_const_handle target)
```

Compute the length of an edge. Arc-length border parameterization: (u,v) values are proportional to the length of border edges.

See Also

CGAL::Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3> page [1937](#)
CGAL::Circular_border_uniform_parameterizer_3<ParameterizationMesh_3> page [1941](#)
CGAL::Square_border_uniform_parameterizer_3<ParameterizationMesh_3> page [1991](#)
CGAL::Two_vertices_parameterizer_3<ParameterizationMesh_3> page [2001](#)

Example

See *Square_border_parameterization.C* example.

CGAL::Square_border_parameterizer_3<ParameterizationMesh_3>

Definition

This is the base class of strategies that parameterize the border of a 3D surface onto a square. *Square_border_parameterizer_3* is a pure virtual class, thus cannot be instantiated.

It implements most of the algorithm. Subclasses just have to implement *compute_edge_length()* to compute a segment's length.

Implementation note: To simplify the implementation, *BorderParameterizer_3* models know only the *ParameterizationMesh_3* class. They do not know the parameterization algorithm requirements or the kind of sparse linear system used.

```
#include <CGAL/Square_border_parameterizer_3.h>
```

Is Model for the Concepts

Model of the *BorderParameterizer_3* concept (although you cannot instantiate this class).

Design Pattern

BorderParameterizer_3 models are Strategies [GHJV95]: they implement a strategy of border parameterization for models of *ParameterizationMesh_3*.

Parameters

The full template declaration is:

```
template<
class ParameterizationMesh_3>
class Square_border_parameterizer_3;
```

Types

Square_border_parameterizer_3<ParameterizationMesh_3>::Adaptor

Export *ParameterizationMesh_3* template parameter.

Creation

Square_border_parameterizer_3<ParameterizationMesh_3> bp;

default constructor.

Operations

Parameterizer_traits_3<Adaptor>::Error_code

bp.parameterize_border(Adaptor& mesh)

Assign to mesh's border vertices a 2D position (ie a (u,v) pair) on border's shape. Mark them as *parameterized*.

bool

bp.is_border_convex()

Indicate if border's shape is convex.

virtual double

*bp.compute_edge_length(Adaptor mesh,
Vertex_const_handle source,
Vertex_const_handle target)*

Compute the length of an edge.

See Also

CGAL::Square_border_arc_length_parameterizer_3<ParameterizationMesh_3> page [1987](#)

CGAL::Square_border_uniform_parameterizer_3<ParameterizationMesh_3> page [1991](#)

CGAL::Square_border_uniform_parameterizer_3<ParameterizationMesh_3>

Definition

This class parameterizes the border of a 3D surface onto a square in a uniform manner: points are equally spaced.

Square_border_parameterizer_3 implements most of the border parameterization algorithm. This class implements only *compute_edge_length()* to compute a segment's length.

```
#include <CGAL/Square_border_parameterizer_3.h>
```

Is Model for the Concepts

Model of the *BorderParameterizer_3* concept.

Design Pattern

BorderParameterizer_3 models are Strategies [GHJV95]: they implement a strategy of border parameterization for models of *ParameterizationMesh_3*.

Parameters

The full template declaration is:

```
template<
class ParameterizationMesh_3>
class Square_border_uniform_parameterizer_3;
```

Types

Creation

```
Square_border_uniform_parameterizer_3<ParameterizationMesh_3> bp;
```

default constructor.

Operations

```
virtual double      bp.compute_edge_length( Adaptor mesh,
                                           Vertex_const_handle source,
                                           Vertex_const_handle target)
```

Compute the length of an edge. Uniform border parameterization: points are equally spaced.

See Also

<i>CGAL::Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3></i>	page 1937
<i>CGAL::Circular_border_uniform_parameterizer_3<ParameterizationMesh_3></i>	page 1941
<i>CGAL::Square_border_arc_length_parameterizer_3<ParameterizationMesh_3></i>	page 1987
<i>CGAL::Two_vertices_parameterizer_3<ParameterizationMesh_3></i>	page 2001

CGAL::Taucs_matrix<T>

Definition

The class *Taucs_matrix* is a C++ wrapper around TAUCS' matrix type *taucs_ccs_matrix*.

This kind of matrix can be either symmetric or not. Symmetric matrices store only the lower triangle.

```
#include <CGAL/Taucs_matrix.h>
```

Is Model for the Concepts

Model of the *SparseLinearAlgebraTraits_d::Matrix* concept.

Parameters

The full template declaration is:

```
template<
class T>
struct Taucs_matrix;
```

Types

Taucs_matrix<T>::NT

Creation

Taucs_matrix<T> M(int dim, bool is_symmetric = false);

Create a square matrix initialized with zeros.

Parameters: **dim** Matrix dimension.

is_symmetric Symmetric/hermitian?

Taucs_matrix<T> M(int rows, int columns, bool is_symmetric = false);

Create a rectangular matrix initialized with zeros.

Parameters: **rows** Matrix dimensions.

is_symmetric Symmetric/hermitian?

Operations

int M.row_dimension() Return the matrix number of rows.

int M.column_dimension()

Return the matrix number of columns.

<i>T</i>	<i>M.get_coef(int i, int j)</i>	<p>Read access to a matrix coefficient. Preconditions:</p> <ul style="list-style-type: none"> • $0 \leq i < \text{row_dimension}()$. • $0 \leq j < \text{column_dimension}()$.
<i>void</i>	<i>M.set_coef(int i, int j, T val)</i>	<p>Write access to a matrix coefficient: $a_{ij} \leftarrow \text{val}$. Optimization: For symmetric matrices, <i>Taucs_matrix</i> stores only the lower triangle <i>set_coef()</i> does nothing if (i, j) belongs to the upper triangle. Preconditions:</p> <ul style="list-style-type: none"> • $0 \leq i < \text{row_dimension}()$. • $0 \leq j < \text{column_dimension}()$.
<i>void</i>	<i>M.add_coef(int i, int j, T val)</i>	<p>Write access to a matrix coefficient: $a_{ij} \leftarrow a_{ij} + \text{val}$. Optimization: For symmetric matrices, <i>Taucs_matrix</i> stores only the lower triangle <i>add_coef()</i> does nothing if (i, j) belongs to the upper triangle. Preconditions:</p> <ul style="list-style-type: none"> • $0 \leq i < \text{row_dimension}()$. • $0 \leq j < \text{column_dimension}()$.
<i>const taucs_ccs_matrix*</i>	<i>M.get_taucs_matrix()</i>	<p>Construct and return the TAUCS matrix wrapped by this object. Note: the TAUCS matrix returned by this method is valid only until the next call to <i>set_coef()</i>, <i>add_coef()</i> or <i>get_taucs_matrix()</i>.</p>

See Also

<i>CGAL::Taucs_solver_traits<T></i>	page 1995
<i>CGAL::Taucs_symmetric_solver_traits<T></i>	page 1998
<i>CGAL::Taucs_symmetric_matrix<T></i>	page 1997
<i>CGAL::Taucs_vector<T></i>	page 2000

CGAL::Taucs_solver_traits<T>

Definition

The class *Taucs_solver_traits* is a traits class for solving GENERAL (aka unsymmetric) sparse linear systems using TAUCS out-of-core LU factorization.

```
#include <CGAL/Taucs_solver_traits.h>
```

Is Model for the Concepts

Model of the *SparseLinearAlgebraTraits_d* concept.

Parameters

The full template declaration is:

```
template<
class T>
class Taucs_solver_traits;
```

Types

```
Taucs_solver_traits<T>:: Matrix
Taucs_solver_traits<T>:: Vector
Taucs_solver_traits<T>:: NT
```

Creation

Taucs_solver_traits<T> solver; default constructor.

Taucs_solver_traits<T> solver; Create a TAUCS sparse linear solver for GENERAL (aka unsymmetric) matrices.

Operations

bool *solver.linear_solver(Matrix A, Vector B, Vector& X, NT& D)*

Solve the sparse linear system $A * X = B$. Return true on success. The solution is then $(1/D) * X$. Preconditions:

- *A.row_dimension()* == *B.dimension()*.
- *A.column_dimension()* == *X.dimension()*.

See Also

CGAL::Taucs_symmetric_solver_traits<T> page [1998](#)
CGAL::Taucs_matrix<T> page [1993](#)

`CGAL::Taucs_symmetric_matrix<T>` page [1997](#)
`CGAL::Taucs_vector<T>` page [2000](#)
`OpenNL::DefaultLinearSolverTraits<COEFFTYPE, MATRIX, VECTOR, SOLVER>` in OpenNL package
`OpenNL::SymmetricLinearSolverTraits<COEFFTYPE, MATRIX, VECTOR, SOLVER>` in OpenNL package

Example

See `Taucs_parameterization.C` example.

CGAL::Taucs_symmetric_matrix<T>

Definition

The class *Taucs_symmetric_matrix* is a C++ wrapper around a TAUCS **symmetric** matrix (type *taucs_ccs_matrix*).

Symmetric matrices store only the lower triangle.

```
#include <CGAL/Taucs_matrix.h>
```

Is Model for the Concepts

Model of the *SparseLinearAlgebraTraits_d::Matrix* concept.

Parameters

The full template declaration is:

```
template<
class T>
struct Taucs_symmetric_matrix;
```

Types

Taucs_symmetric_matrix<T>::NT

Creation

Taucs_symmetric_matrix<T> M(int dim); Create a square SYMMETRIC matrix initialized with zeros.
The max number of non 0 elements in the matrix is automatically computed.

Parameters: **dim** Matrix dimension.

Taucs_symmetric_matrix<T> M(int rows, int columns, int nb_max_elements = 0);

Create a square SYMMETRIC matrix initialized with zeros.

Parameters: **rows** Matrix dimensions.

nb_max_elements Max number of non 0 elements in the matrix (automatically computed if 0).

Operations

See Also

CGAL::Taucs_solver_traits<T> page [1995](#)
CGAL::Taucs_symmetric_solver_traits<T> page [1998](#)
CGAL::Taucs_matrix<T> page [1993](#)
CGAL::Taucs_vector<T> page [2000](#)
OpenNL::DefaultLinearSolverTraits<COEFFTYPE, MATRIX, VECTOR, SOLVER> in OpenNL package
OpenNL::SymmetricLinearSolverTraits<COEFFTYPE, MATRIX, VECTOR, SOLVER> in OpenNL package

CGAL::Taucs_symmetric_solver_traits<T>

Definition

The class *Taucs_symmetric_solver_traits* is a traits class for solving SYMMETRIC DEFINITE POSITIVE sparse linear systems using TAUCS solvers family. The default solver is the Multifrontal Supernodal Cholesky Factorization.

```
#include <CGAL/Taucs_solver_traits.h>
```

Is Model for the Concepts

Model of the *SparseLinearAlgebraTraits_d* concept.

Parameters

The full template declaration is:

```
template<
class T>
class Taucs_symmetric_solver_traits;
```

Types

```
Taucs_symmetric_solver_traits<T>:: Matrix
Taucs_symmetric_solver_traits<T>:: Vector
Taucs_symmetric_solver_traits<T>:: NT
```

Creation

```
Taucs_symmetric_solver_traits<T> solver( const char * options[] = NULL,
                                          const void * arguments[] = NULL)
```

Create a TAUCS sparse linear solver for SYMMETRIC DEFINITE POSITIVE matrices. The default solver is the Multifrontal Supernodal Cholesky Factorization. See *taucs_solve()* documentation for the meaning of the 'options' and 'arguments' parameters.

Parameters: **options** must be persistent
arguments must be persistent

Operations

```
bool solver.linear_solver( Matrix A, Vector B, Vector& X, NT& D)
```

Solve the sparse linear system $A \cdot X = B$. Return true on success. The solution is then $(1/D) \cdot X$. Preconditions:

- *A.row_dimension()* == *B.dimension()*.
- *A.column_dimension()* == *X.dimension()*.

See Also

CGAL::Taucs_solver_traits<T> page [1995](#)
CGAL::Taucs_matrix<T> page [1993](#)
CGAL::Taucs_symmetric_matrix<T> page [1997](#)
CGAL::Taucs_vector<T> page [2000](#)
OpenNL::DefaultLinearSolverTraits<COEFFTYPE, MATRIX, VECTOR, SOLVER> in OpenNL package
OpenNL::SymmetricLinearSolverTraits<COEFFTYPE, MATRIX, VECTOR, SOLVER> in OpenNL package

CGAL::Taucs_vector<T>

Definition

The class *Taucs_vector* is a C++ wrapper around TAUCS' vector type, which is a simple array.

```
#include <CGAL/Taucs_vector.h>
```

Is Model for the Concepts

Model of the *SparseLinearAlgebraTraits_d::Vector* concept.

Parameters

The full template declaration is:

```
template<
class T>
class Taucs_vector;
```

Types

Taucs_vector<T>::NT

Creation

<i>Taucs_vector<T></i> <i>v</i> (<i>int dimension</i>);	Create a vector initialized with zeros.
<i>Taucs_vector<T></i> <i>v</i> (<i>toCopy</i>);	Copy constructor.

Operations

<i>Taucs_vector&</i>	<i>v = toCopy</i>	operator =()
<i>int</i>	<i>v.dimension()</i>	Return the vector's number of coefficients.
<i>T</i>	<i>v[int i]</i>	Read/write access to a vector coefficient. Preconditions: 0 <= i < dimension().
<i>T&</i>	<i>v[int i]</i>	
<i>const T*</i>	<i>v.get_taucs_vector()</i>	Get TAUCS vector wrapped by this object.
<i>T*</i>	<i>v.get_taucs_vector()</i>	

See Also

CGAL::Taucs_solver_traits<T> page [1995](#)
CGAL::Taucs_symmetric_solver_traits<T> page [1998](#)
CGAL::Taucs_matrix<T> page [1993](#)
CGAL::Taucs_symmetric_matrix<T> page [1997](#)
OpenNL::DefaultLinearSolverTraits<COEFFTYPE, MATRIX, VECTOR, SOLVER> in OpenNL package
OpenNL::SymmetricLinearSolverTraits<COEFFTYPE, MATRIX, VECTOR, SOLVER> in OpenNL package

CGAL::Two_vertices_parameterizer_3<ParameterizationMesh_3>

Definition

Two_vertices_parameterizer_3 is the default border parameterizer for Least Squares Conformal Maps parameterization.

The class *Two_vertices_parameterizer_3* parameterizes two extreme vertices of a 3D surface. This kind of border parameterization is used by free border parameterizations.

Implementation note: To simplify the implementation, *BorderParameterizer_3* models know only the *ParameterizationMesh_3* class. They do not know the parameterization algorithm requirements or the kind of sparse linear system used.

```
#include <CGAL/Two_vertices_parameterizer_3.h>
```

Is Model for the Concepts

Model of the *BorderParameterizer_3* concept.

Design Pattern

BorderParameterizer_3 models are Strategies [GHJV95]: they implement a strategy of border parameterization for models of *ParameterizationMesh_3*.

Parameters

The full template declaration is:

```
template<
class ParameterizationMesh_3>
class Two_vertices_parameterizer_3;
```

Types

Two_vertices_parameterizer_3<ParameterizationMesh_3>::Adaptor

Export *ParameterizationMesh_3* template parameter.

Creation

Two_vertices_parameterizer_3<ParameterizationMesh_3> bp;

default constructor.

Operations

Parameterizer_traits_3<Adaptor>::Error_code

bp.parameterize_border(Adaptor& mesh)

Map two extreme vertices of the 3D mesh and mark them as *parameterized*. Map two extreme vertices of the 3D mesh and mark them as *parameterized*. Return false on error.

bool

bp.is_border_convex()

Indicate if border's shape is convex. Meaningless for free border parameterization algorithms.

See Also

CGAL::Circular_border_arc_length_parameterizer_3<ParameterizationMesh_3> page [1937](#)
CGAL::Circular_border_uniform_parameterizer_3<ParameterizationMesh_3> page [1941](#)
CGAL::Square_border_arc_length_parameterizer_3<ParameterizationMesh_3> page [1987](#)
CGAL::Square_border_uniform_parameterizer_3<ParameterizationMesh_3> page [1991](#)

SparseLinearAlgebra_d::Vector

Definition

SparseLinearAlgebraTraits_d::Vector is a concept of a vector that can be multiplied by a sparse matrix.

Refines

This is a sub-concept of *LinearAlgebraTraits_d::Vector*.

Types

Vector::NT

Creation

<i>Vector</i> <i>v</i> (<i>int</i> <i>rows</i>);	Create a vector initialized with zeros.
<i>Vector</i> <i>v</i> (<i>Vector toCopy</i>);	Copy constructor.

Operations

<i>Vector</i> &	<i>v</i> = <i>Vector toCopy</i>	operator =()
<i>int</i>	<i>v.dimension</i> ()	Return the vector's number of coefficients.
<i>NT</i>	<i>v</i> [<i>int</i> <i>row</i>]	Read/write access to a vector coefficient. Precondition: 0 <= <i>row</i> < <i>dimension</i> () .
<i>NT</i> &	<i>v</i> [<i>int</i> <i>row</i>]	

Has Models

CGAL::Taucs_vector<*T*>
OpenNL::FullVector<*T*> in OpenNL package

See Also

SparseLinearAlgebraTraits_dpage [1986](#)
SparseLinearAlgebraTraits_d::Matrix page ??

Part X

Search Structures

Chapter 33

2D Search Structures

Matthias Bäsken

Contents

33.1 Introduction	2007
33.2 Examples	2008
33.2.1 Range search operations	2008

33.1 Introduction

Geometric queries are fundamental to many applications in computational geometry. The task is to maintain a dynamic set of geometric objects in such a way that certain queries can be performed efficiently. Typical examples of queries are: find out whether a given object is contained in the set, find all objects of the set lying in a given area (e.g. rectangle), find the object closest to a given point or find the pair of objects in the set lying closest to each other. Furthermore, the set should be dynamic in the sense that deletions and insertions of objects can be performed efficiently.

In computational geometry literature one can find many different data structures for maintaining sets of geometric objects. Most of them are data structures that have been developed to support a single very special kind of query operation. Examples are Voronoi diagrams for answering nearest neighbor searches, range trees for orthogonal range queries, partition trees for more general range queries, hierarchical triangulations for point location and segment trees for intersection queries

In many applications, different types of queries have to be performed on the same set of objects. A naive approach to this problem would use a collection of the above mentioned data structures to represent the set of objects and delegate every query operation to the corresponding structure. However, this is completely impractical since it uses too much memory and requires the maintenance of all these data structures in the presence of update operations.

Data structures that are non-optimal in theory seem to perform quite well in practice for many of these queries. For example, the Delaunay diagram turns out to be a very powerful data structure for storing dynamic sets of points under range and nearest neighbor queries. A first implementation and computational study of using Delaunay diagrams for geometric queries is described by Mehlhorn and Näher in [\[MN00\]](#).

In this section we present a generic variant of a two dimensional point set data type supporting various geometric queries.

The `CGAL::Point_set_2` class in this section is inherited from the two-dimensional *CGAL Delaunay Triangulation* data type.

The `CGAL::Point_set_2` class depends on two template parameters *T1* and *T2*. They are used as template parameters for the `CGAL::Delaunay_triangulation_2` class `CGAL::Point_set_2` is inherited from. *T1* is a model for the geometric traits and *T2* is a model for the triangulation data structure that the Delaunay triangulation expects.

The `CGAL::Point_set_2` class supports the following kinds of queries:

- circular range search
- triangular range search
- isorectangular range search
- (k) nearest neighbor(s)

For details about the running times see [\[MN00\]](#).

33.2 Examples

33.2.1 Range search operations

The following example program demonstrates the various range search operations of the two dimensional point set. First we construct a two dimensional point set *PSet* and initialize it with a few points. Then we perform circular, triangular and isorectangular range search operations on the point set.

rs.example.C :

```
// file: examples/Point_set_2/rs_example.C

#include <CGAL/Cartesian.h>
#include <list>
#include <CGAL/Point_set_2.h>

typedef CGAL::Cartesian<double>      K;

typedef CGAL::Point_set_2<K>::Vertex_handle  Vertex_handle;
typedef CGAL::Point_2<K>                    Point;

int main()
{
    CGAL::Point_set_2<K> PSet;
    std::list<Point> Lr;

    Point p1(12,14);
    Point p2(-12,14);
    Point p3(2,11);
    Point p4(5,6);
    Point p5(6.7,3.8);
```



```

Point p6(11,20);
Point p7(-5,6);
Point p8(12,0);
Point p9(4,31);
Point p10(-10,-10);

Lr.push_back(p1); Lr.push_back(p2); Lr.push_back(p3);
Lr.push_back(p4); Lr.push_back(p5); Lr.push_back(p6);
Lr.push_back(p7); Lr.push_back(p8); Lr.push_back(p9);
Lr.push_back(p10);

PSet.insert(Lr.begin(),Lr.end());

std::cout << "circular range search !\n";
CGAL::Circle_2<K> rc(p5,p6);

std::list<Vertex_handle> LV;
PSet.range_search(rc, std::back_inserter(LV));

std::list<Vertex_handle>::const_iterator it;
for (it=LV.begin();it != LV.end(); it++)
    std::cout << (*it)->point() << "\n";

std::cout << "triangular range search !\n";

LV.clear();
PSet.range_search(p1,p2,p3, std::back_inserter(LV));
for (it=LV.begin();it != LV.end(); it++)
    std::cout << (*it)->point() << "\n";
LV.clear();

std::cout << "isorectangular range search !\n";
Point pt1=p10;
Point pt3=p3;
Point pt2 = Point(pt3.x(),pt1.y());
Point pt4 = Point(pt1.x(),pt3.y());

PSet.range_search(pt1,pt2,pt3,pt4, std::back_inserter(LV));
for (it=LV.begin();it != LV.end(); it++)
    std::cout << (*it)->point() << "\n";
return 0;
}

```


2D Search Structures

Reference Manual

Matthias Bäcken

The two dimensional point set is a class for geometric queries. It supports circular, triangular and iso rectangular range searches and nearest neighbor searches. The point set is inherited from the CGAL Delaunay triangulation data type. That means that it is a dynamic data structure supporting the insertion and deletion of points.

This package also provides function template versions of the range search and nearest neighbor query operations. They all have to be templated by the type of a CGAL Delaunay triangulation and provide functionality similar to the corresponding member functions of the point set class.

33.3 Classified Reference Pages

Concepts

PointSetTraits

Classes

CGAL::Point_set_2

Functions

CGAL::nearest_neighbor
CGAL::nearest_neighbors
CGAL::range_search

33.4 Alphabetical List of Reference Pages

nearest_neighbors page [2018](#)
nearest_neighbor page [2017](#)

<i>PointSetTraits</i>	page 2016
<i>Point_set_2<Gt,Tds></i>	page 2013
<i>range_search</i>	page 2020

CGAL::Point_set_2<Gt,Tds>

Definition

`#include <CGAL/Point_set_2.h>`

An instance *PS* of the data type *Point_set_2<Gt,Tds>* is a *Delaunay Triangulation* of its vertex set. The class *Point_set_2<Gt,Tds>* is inherited from the CGAL Delaunay triangulation, and provides additional nearest neighbor query operations and range searching operations.

The *Point_set_2<Gt,Tds>* class of CGAL depends on template parameters standing for the geometric traits classes used by the point set and by the Delaunay triangulation (Gt) and for the triangulation data structure (Tds).

Types

<code>typedef Gt::Point_2</code>	<code>Point;</code>	the point type
<code>typedef Gt::Segment_2</code>	<code>Segment;</code>	the segment type
<code>typedef Gt::Circle_2</code>	<code>Circle;</code>	the circle type
<code>typedef Gt::FT</code>	<code>Numb_type;</code>	the representation field number type.
<code>Point_set_2<Gt,Tds>:: Triangulation::Vertex</code>		the vertex type of the underlying triangulation.
<code>Point_set_2<Gt,Tds>:: Triangulation::Edge</code>		the edge type of the underlying triangulation.
<code>Point_set_2<Gt,Tds>:: Triangulation::Vertex_handle</code>		handles to vertices.

Creation

<code>Point_set_2<Gt,Tds> PS;</code>	creates an empty <i>Point_set_2<Gt,Tds></i> .
<code>template<class InputIterator></code> <code>Point_set_2<Gt,Tds> PS(InputIterator first, InputIterator last);</code>	creates a <i>Point_set_2<Gt,Tds></i> <i>PS</i> of the points in the range <i>[first,last)</i> .

Operations

Vertex_handle

PS.lookup(Point p)

if *PS* contains a Vertex v with $|pos(v)| = p$ the result is a handle to v otherwise the result is *NULL*.

Vertex_handle

PS.nearest_neighbor(Point p)

computes a handle to a vertex v of *PS* that is closest to p . If *PS* is empty, *NULL* is returned.

Vertex_handle

PS.nearest_neighbor(Vertex_handle v)

computes a handle to a vertex w of *PS* that is closest to v . If v is the only vertex in *PS*, *NULL* is returned.

template<class OutputIterator>

OutputIterator

PS.nearest_neighbors(Point p, int k, OutputIterator res)

computes the k nearest neighbors of p in *PS*, and places the handles to the corresponding vertices as a sequence of objects of type *Vertex_handle* in a container of value type of *res* which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence.

template<class OutputIterator>

OutputIterator

PS.nearest_neighbors(Vertex_handle v, int k, OutputIterator res)

computes the k nearest neighbors of v , and places them as a sequence of objects of type *Vertex_handle* in a container of value type of *res* which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence.

template<class OutputIterator>

OutputIterator

PS.range_search(Circle C, OutputIterator res)

computes handles to all vertices contained in the closure of disk C . The computed vertex handles will be placed as a sequence of objects in a container of value type of *res* which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence.

template<class OutputIterator>

OutputIterator

PS.range_search(Point a, Point b, Point c, OutputIterator res)

computes handles to all vertices contained in the closure of the triangle (a,b,c) .

Precondition: a , b , and c must not be collinear. The computed vertex handles will be placed as a sequence of objects in a container of value type of res which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence.

template<class OutputIterator>

OutputIterator

PS.range_search(Point a1, Point b1, Point c1, Point d1, OutputIterator res)

computes handles to all vertices contained in the closure of the iso-rectangle $(a1,b1,c1,d1)$.

Precondition: $a1$ is the upper left point, $b1$ the lower left, $c1$ the lower right and $d1$ the upper right point of the iso rectangle. The computed vertex handles will be placed as a sequence of objects in a container of value type of res which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence.

PointSetTraits

A point set traits class has to provide some primitives that are used by the point set class. The following catalog lists the involved primitives. For details about these types see the Kernel traits documentation.

Types

PointSetTraits:: Point_2

PointSetTraits:: Circle_2

PointSetTraits:: Segment_2

PointSetTraits:: FT

PointSetTraits:: Orientation_2

PointSetTraits:: Side_of_oriented_circle_2

PointSetTraits:: Construct_circle_2

PointSetTraits:: Compute_squared_distance_2

PointSetTraits:: Bounded_side_2

PointSetTraits:: Compare_distance_2

PointSetTraits:: Construct_center_2

CGAL::nearest_neighbor

Definition

The function *nearest_neighbor* is the function template version of the nearest neighbor search on Delaunay triangulations.

```
#include <CGAL/nearest_neighbor_delaunay_2.h>
```

```
template<class Dt>  
Dt::Vertex_handle nearest_neighbor( Dt delau, Dt::Vertex_handle v)
```

computes a handle to a vertex *w* of *delau* that is closest to *v*.
If *v* is the only vertex in *delau*, *NULL* is returned.

Requirements

Dt is a CGAL Delaunay triangulation and contains the following subset of types from the concept PointSetTraits and from the Delaunay triangulation data type:

- *Dt::Geom_traits*
- *Dt::Point*
- *Dt::Vertex_circulator*
- *Dt::Vertex_handle*
- *Dt::Geom_traits::Compare_distance_2*

CGAL::nearest_neighbors

Definition

The function *nearest_neighbors* is the function template version of the k nearest neighbors search on Delaunay triangulations. There are two versions of this function, one taking a point of the Delaunay triangulation and the other taking a vertex handle.

```
#include <CGAL/nearest_neighbor_delaunay_2.h>
```

```
template<class Dt, class OutputIterator>
OutputIterator nearest_neighbors( Dt& delau, Dt::Point p, int k, OutputIterator res)
```

computes the k nearest neighbors of p in *delau*, and places the handles to the corresponding vertices as a sequence of objects of type *Vertex_handle* in a container of value type of *res* which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence.

Requirements

Dt is a CGAL Delaunay triangulation and contains the following subset of types from the concept *PointSetTraits* and from the Delaunay triangulation data type:

- *Dt::Geom_traits*
- *Dt::Vertex_handle*
- *Dt::Vertex_iterator*
- *Dt::Vertex_circulator*
- *Dt::Vertex*
- *Dt::Face*
- *Dt::Face_handle*
- *Dt::Locate_type*
- *Dt::Point*
- *Dt::Geom_traits::FT*
- *Dt::Geom_traits::Compute_squared_distance_2*

```
template<class Dt, class OutputIterator>
```

OutputIterator *nearest_neighbors(Dt& delau, Dt::Vertex_handle v, int k, OutputIterator res)*

computes the k nearest neighbors of v (including v) in *delau*, and places them as a sequence of objects of type *Vertex_handle* in a container of value type of *res* which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence.

Requirements

Dt is a CGAL Delaunay triangulation and contains the following subset of types from the concept *PointSetTraits* and from the Delaunay triangulation data type:

- *Dt::Geom_traits*
- *Dt::Vertex_handle*
- *Dt::Vertex_iterator*
- *Dt::Vertex_circulator*
- *Dt::Vertex*
- *Dt::Point*
- *Dt::Geom_traits::FT*
- *Dt::Geom_traits::Compute_squared_distance_2*

CGAL::range_search

Definition

There are six versions of the function template *range_search* that perform range searches on Delaunay triangulations. The first performs circular range searches, the second triangular range searches and the third performs iso-rectangular range searches. The other three range search function templates perform enhanced variants of the three beforementioned operations.

They get a user-defined object that has to control the range search operation. This way one can for instance stop the search, when n points were found.

```
#include <CGAL/range_search_delaunay_2.h>
```

```
template<class Dt, class Circle, class OutputIterator>
OutputIterator      range_search( Dt& delau, Circle C, OutputIterator res)
```

computes handles to all vertices contained in the closure of disk C . The computed vertex handles will be placed as a sequence of objects in a container of value type of *res* which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence. *delau* is the CGAL Delaunay triangulation on which we perform the range search operation.

Requirements

- *Dt* is a CGAL Delaunay triangulation and contains the following subset of types from the concept PointSetTraits and from the Delaunay triangulation data type:
 - *Dt::Geom_traits*
 - *Dt::Vertex_handle*
 - *Dt::Vertex*
 - *Dt::Vertex_circulator*
 - *Dt::Vertex_iterator*
 - *Dt::Point*
 - *Dt::Geom_traits::Bounded_side_2*
 - *Dt::Geom_traits::Construct_center_2*
- the template parameter *Circle* corresponds to *Dt::Geom_traits::Circle_2*

```
template<class Dt, class OutputIterator>
```

OutputIterator *range_search(Dt& delau, Dt::Point a, Dt::Point b, Dt::Point c, OutputIterator res)*

computes handles to all vertices contained in the closure of the triangle (a, b, c) .

Precondition: a , b , and c must not be collinear. The computed vertex handles will be placed as a sequence of objects in a container of value type of *res* which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence. *delau* is the CGAL Delaunay triangulation on which we perform the range search operation.

Requirements

Dt is a CGAL Delaunay triangulation and contains the following subset of types from the concept *PointSetTraits* and from the Delaunay triangulation data type:

- *Dt::Geom_traits*
- *Dt::Vertex_handle*
- *Dt::Vertex*
- *Dt::Vertex_circulator*
- *Dt::Vertex_iterator*
- *Dt::Point*
- *Dt::Geom_traits::Circle_2*
- *Dt::Geom_traits::Bounded_side_2*
- *Dt::Geom_traits::Construct_center_2*
- *Dt::Geom_traits::Orientation_2*
- *Dt::Geom_traits::Construct_circle_2*

```
template<class Dt, class OutputIterator>
OutputIterator      range_search( Dt& delau,
                                Dt::Point a,
                                Dt::Point b,
                                Dt::Point c,
                                Dt::Point d,
```

OutputIterator res)

computes handles to all vertices contained in the closure of the iso-rectangle (a, b, c, d) .

Precondition: a is the upper left point, b the lower left, c the lower right and d the upper right point of the iso rectangle. The computed vertex handles will be placed as a sequence of objects in a container of value type of res which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence. $delau$ is the CGAL Delaunay triangulation on which we perform the range search operation.

Requirements

Dt is a CGAL Delaunay triangulation and contains the following subset of types from the concept PointSetTraits and from the Delaunay triangulation data type:

- $Dt::Geom_traits$
- $Dt::Vertex_handle$
- $Dt::Vertex$
- $Dt::Vertex_circulator$
- $Dt::Vertex_iterator$
- $Dt::Point$
- $Dt::Geom_traits::Circle_2$
- $Dt::Geom_traits::Bounded_side_2$
- $Dt::Geom_traits::Construct_center_2$
- $Dt::Geom_traits::Orientation_2$
- $Dt::Geom_traits::Construct_circle_2$

```
template<class Dt, class Circle, class OutputIterator, class Pred>
OutputIterator      range_search( Dt& delau,
                                Circle C,
                                OutputIterator res,
                                Pred& pred,
                                bool return_if_succeeded)
```

computes handles to all vertices contained in the closure of disk C . The computed vertex handles will be placed as a sequence of objects in a container of value type of res which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence. $delau$ is the CGAL Delaunay triangulation on that we perform the range search operation. $pred$ controls the search operation. If $return_if_succeeded$ is *true*, we will end the search after the first success of the predicate, otherwise we will continue till the search is finished.

Requirements

For the requirements of *Dt* see the description for the non-predicate version.

Requirements of *Pred*:

- *void set_result(bool);*
- *bool operator()(const Point&);*

The *operator()* is used for testing the current point in the search operation. If this operator returns *true* and *return_if_succeeded* is *true*, the range search will stop. Otherwise the range search operation will continue. Member function *set_result* can be used to store the result of the range search in the function object. The result will be *true* if the last call to the *operator()* of the predicate returned *true*, *false* otherwise.

```
template<class Dt, class OutputIterator, class Pred>
OutputIterator      range_search( Dt& delau,
                                Dt::Point a,
                                Dt::Point b,
                                Dt::Point c,
                                OutputIterator res,
                                Pred& pred,
                                bool return_if_succeeded)
```

computes handles to all vertices contained in the closure of the triangle (a, b, c) .

Precondition: *a*, *b*, and *c* must not be collinear. The computed vertex handles will be placed as a sequence of objects in a container of value type of *res* which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence. *delau* is the CGAL Delaunay triangulation on which we perform the range search operation. *pred* controls the search operation. If *return_if_succeeded* is *true*, we will end the search after the first success of the predicate, otherwise we will continue till the search is finished.

Requirements

For the requirements of *Dt* see the description for the non-predicate version.

For the requirements of *Pred* see the description above.

```
template<class Dt, class OutputIterator, class Pred>
OutputIterator      range_search( Dt& delau,
                                Dt::Point a,
                                Dt::Point b,
                                Dt::Point c,
                                Dt::Point d,
                                OutputIterator res,
                                Pred& pred,
```

bool return_if_succeeded)

computes handles to all vertices contained in the closure of the iso-rectangle (a, b, c, d) .

Precondition: a is the upper left point, b the lower left, c the lower right and d the upper right point of the iso rectangle. The computed vertex handles will be placed as a sequence of objects in a container of value type of *res* which points to the first object in the sequence. The function returns an output iterator pointing to the position beyond the end of the sequence. *delau* is the CGAL Delaunay triangulation on which we perform the range search operation. *pred* controls the search operation. If *return_if_succeeded* is *true*, we will end the search after the first success of the predicate, otherwise we will continue till the search is finished.

Requirements

For the requirements of *Dt* see the description for the non-predicate version.

For the requirements of *Pred* see the description above.

Chapter 34

Interval Skip List

Andreas Fabri

34.1 Definition

An interval skip list is a data structure for finding all intervals that contain a point, and for stabbing queries, that is for answering the question whether a given point is contained in an interval or not. The implementation we provide is dynamic, that is the user can freely mix calls to the methods *insert(..)*, *remove(..)*, *find_intervals(..)*, and *is_contained(..)*

The interval skip list class is parameterized with an interval class.

The data structure was introduced by Hanson [Han91], and it is called interval skip list, because it is an extension of the randomized list structure known as skip list [Pug90].

34.2 Example Programs

We give two examples. The first one uses a basic interval class. In the second example we create an interval skip list for the z -ranges of the faces of a terrain, allowing to answer level queries.

34.2.1 First Example with Simple Interval

The first example reads two numbers n and d from *std::cin*. It creates n intervals of length d with the left endpoint at n . It then reads out the intervals for the 1-dimensional points with coordinates $0 \dots n + d$.

The interval skip list class has as template argument an interval class. In this example we use the class *Interval_skip_list_interval*.

```

// file: examples/Interval_skip_list/intervals.C

#include <CGAL/Interval_skip_list.h>
#include <CGAL/Interval_skip_list_interval.h>
#include <vector>
#include <list>
#include <iostream>

typedef CGAL::Interval_skip_list_interval<double> Interval;
typedef CGAL::Interval_skip_list<Interval> Interval_skip_list;

int main()
{
    Interval_skip_list isl;
    int i, n, d;

    n = 10;
    d = 3;
    //std::cin >> n >> d;
    std::vector<Interval> intervals(n);
    for(i = 0; i < n; i++) {
        intervals[i] = Interval(i, i+d);
    }
    std::random_shuffle(intervals.begin(), intervals.end());

    isl.insert(intervals.begin(), intervals.end());

    for(i = 0; i < n+d; i++) {
        std::list<Interval> L;
        isl.find_intervals(i, std::back_inserter(L));
        for(std::list<Interval>::iterator it = L.begin(); it != L.end(); it++){
            std::cout << *it;
        }
        std::cout << std::endl;
    }

    for(i = 0; i < n; i++) {
        isl.remove(intervals[i]);
    }
    return 0;
}

```

34.2.2 Example with Faces of a Triangulated Terrain

The second example creates an interval skip list that allows to find all the faces of a terrain intersected by an horizontal plane at a given height. The data points for the terrain are read from a file.

As model for the interval concept, we use a class that is a wrapper around a face handle of a triangulated terrain. Lower and upper bound of the interval are smallest and largest z -coordinate of the face.

```

// file: examples/Interval_skip_list/terrain.C

#include <CGAL/Simple_cartesian.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/Triangulation_euclidean_traits_xy_3.h>
#include <CGAL/Interval_skip_list.h>
#include <CGAL/Level_interval.h>
#include <iostream>
#include <fstream>

typedef CGAL::Simple_cartesian<double>          SC;
typedef SC::Point_3                             Point_3;
typedef CGAL::Triangulation_euclidean_traits_xy_3<SC> K;
typedef CGAL::Delaunay_triangulation_2<K>      Delaunay;
typedef Delaunay::Face_handle                   Face_handle;
typedef Delaunay::Finite_faces_iterator        Finite_faces_iterator;
typedef CGAL::Level_interval<Face_handle>      Interval;
typedef CGAL::Interval_skip_list<Interval>     Interval_skip_list;

int main()
{
    std::ifstream fin("terrain.pts"); // elevation ranges from 0 to 100
    Delaunay dt;

    dt.insert(std::istream_iterator<Point_3>(fin),
              std::istream_iterator<Point_3>());

    Interval_skip_list isl;
    for(Finite_faces_iterator fh = dt.finite_faces_begin();
        fh != dt.finite_faces_end();
        ++fh){
        isl.insert(Interval(fh));
    }
    std::list<Interval> level;
    isl.find_intervals(50, std::back_inserter(level));
    for(std::list<Interval>::iterator it = level.begin();
        it != level.end();
        ++it){
        std::cout << dt.triangle(it->face_handle()) << std::endl;
    }
    return 0;
}

```


Interval Skip List Reference Manual

Andreas Fabri

This chapter presents the interval skip list introduced by Hanson [[Han91](#)], and derived from the skip list data structure [[Pug90](#)].

The data structure stores intervals and allows to perform stabbing queries, that is to test whether a point is covered by any of the intervals. It further allows to find all intervals that contain a point.

The interval skip list is, as far as its functionality is concerned, related to the *Segment_tree*. Both allow to do stabbing queries and both allow to find all intervals that contain a given point. The implementation of segment trees in CGAL works in higher dimensions, whereas the interval skip list is limited to the 1D case. However, this interval skip list implementation is fully dynamic, whereas the segment tree implementation in CGAL is static, that is all intervals must be known in advance.

This package has one concept, namely for the interval with which the interval skip list class is parameterized.

34.3 Classified Reference Pages

Concepts

Interval page [2033](#)

Classes

CGAL::Interval_skip_list<*Interval*> page [2031](#)
CGAL::Interval_skip_list_interval<*Value*> page [2034](#)
CGAL::Level_interval<*FaceHandle*> page [2035](#)

34.4 Alphabetical List of Reference Pages

Interval_skip_list<*Interval*> page [2031](#)
Interval_skip_list_interval<*Value*> page [2034](#)

<i>Interval</i>	page 2033
<i>Level_interval</i> < <i>FaceHandle</i> >	page 2035

CGAL::Interval_skip_list<Interval>

Definition

The class *Interval_skip_list<Interval>* is a dynamic datastructure that allows to find all members of a set of intervals that overlap a point.

#include <CGAL/Interval_skip_list.h>

Types

typedef Interval::Value Value; the type of inf and sup of the interval.

Interval_skip_list<Interval>:: const_iterator

An iterator over all intervals.

Creation

Interval_skip_list<Interval> isl; Default constructor.

template < class InputIterator >

Interval_skip_list<Interval> isl(InputIterator first, InputIterator last);

Constructor that inserts the iterator range *[first, last)* in the interval skip list.

Precondition: The *value_type* of *first* and *last* is *Interval*.

Operations

template < class InputIterator >

int isl.insert(InputIterator first, InputIterator last)

Inserts the iterator range *[first, last)* in the interval skip list, and returns the number of inserted intervals.

Precondition: The *value_type* of *first* and *last* is *Interval*.

void isl.insert(Interval i)

inserts interval *i* in the interval skip list.

bool isl.remove(Interval i)

removes interval *i* from the interval skip list. Returns *true* iff removal was succesful.

bool *isl.is_contained(Value v)*

Returns *true* iff there is an interval that contains *v*.

template < class OutputIterator >
OutputIterator *isl.find_intervals(Value v, OutputIterator out)*

Writes the intervals *i* with $i.inf() \leq v \leq i.sup$ to the output iterator *out*.
Precondition: The *value_type* of *out* is *Interval*.

void *isl.clear()*

Removes all intervals from the interval skip list.

const_iterator *isl.begin()*

Returns an iterator over all intervals.

const_iterator *isl.end()* Returns the past the end iterator.

I/O

ostream& *ostream& os << isl*

Inserts the interval skip list *isl* into the stream *os*.
Precondition: The output operator must be defined for *Interval*.

Implementation

The insertion and deletion of a segment in the interval skip list takes expected time $O(\log^2 n)$, if the segment endpoints are chosen from a continuous distribution. A stabbing query takes expected time $O(\log n)$, and finding all intervals that contain a point takes expected time $O(\log n + k)$, where *k* is the number of intervals.

The implementation is based on the code developed by Eric N. Hansen, which can be found at <http://www-pub.cise.ufl.edu/~hanson/IS-lists/>. Attention, this code has memory leaks.

CGAL::Interval_skip_list_interval<Value>

Definition

The class *Interval_skip_list_interval*<Value> represents intervals with lower and upper bound of type *Value*. These intervals can be opened or closed at each endpoint.

```
#include <CGAL/Interval_skip_list_interval.h>
```

Creation

```
Interval_skip_list_interval<Value> i;
```

Default constructor.

```
Interval_skip_list_interval<Value> i( Value i, Value s, bool ic = true, bool uc = true);
```

Constructs the interval with infimum *i* and supremum *s*. The arguments *ic* and *uc* have value *true*, iff the interval is closed at the lower and upper bound, respectively.

Operations

```
bool          i.inf_closed()    returns true, iff the interval is closed at the lower bound.
```

```
bool          i.sup_closed()    returns true, iff the interval is closed at the upper bound.
```

I/O

The output operator is defined for *std::ostream*.

```
ostream&      ostream& os << Interval_skip_list_interval<V> i
```

Inserts the interval *i* into the stream *os*.

Precondition: The output operator for *Value* is defined.

Is Model for the Concepts

Interval

CGAL::Level_interval<FaceHandle>

Definition

The class *Level_interval<FaceHandle>* represents intervals for the minimum and maximum value of the *z*-coordinate of a face of a triangulation.

```
#include <CGAL/Level_interval.h>
```

Requirements

The *value_type* of *FaceHandle* must be *Face*, which must have a nested type *Vertex*, which must have a nested type *Point*, whose *Kernel_traits<Point>Kernel* must have a nested type *FT*. These requirements are fulfilled, if one uses a CGAL triangulation and a CGAL kernel.

Types

```
typedef FT      Value;           The type of the z-coordinate of points stored in vertices of faces.
```

Creation

```
Level_interval<FaceHandle> i;      Default constructor.
```

```
Level_interval<FaceHandle> i( FaceHandle fh);
```

Constructs the interval with smallest and largest *z* coordinate of the points stored in the vertices of the face *fh* points to.

Operations

```
FaceHandle      i.face_handle()    returns the face handle.
```

I/O

```
ostream&        ostream& os << i    Inserts the interval i into the stream os.  
Precondition: The output operator for *Face_handle is defined.
```

Is Model for the Concepts

```
Interval
```


Chapter 35

dD Spatial Searching

Hans Tangelder and Andreas Fabri

35.1 Introduction

The spatial searching package implements exact and approximate distance browsing by providing implementations of algorithms supporting

- both nearest and furthest neighbor searching
- both exact and approximate searching
- (approximate) range searching
- (approximate) k -nearest and k -furthest neighbor searching
- (approximate) incremental nearest and incremental furthest neighbor searching
- query items representing points and spatial objects.

In these searching problems a set P of data points in d -dimensional space is given. The points can be represented by Cartesian coordinates or homogeneous coordinates. These points are preprocessed into a tree data structure, so that given any query item q the points of P can be browsed efficiently. The approximate spatial searching package is designed for data sets that are small enough to store the search structure in main memory (in contrast to approaches from databases that assume that the data reside in secondary storage).

35.1.1 Neighbor Searching

Spatial searching supports browsing through a collection of d -dimensional spatial objects stored in a spatial data structure on the basis of their distances to a query object. The query object may be a point or an arbitrary spatial object, e.g., a d -dimensional sphere. The objects in the spatial data structure are d -dimensional points.

Often the number of the neighbors to be computed is not known beforehand, e.g., because the number may depend on some properties of the neighbors (for example when querying for the nearest city to Paris with population greater than a million) or the distance to the query point. The conventional approach is *k-nearest neighbor searching* that makes use of a k -nearest neighbor algorithm, where k is known prior to the invocation

of the algorithm. Hence, the number of nearest neighbors has to be guessed. If the guess is too large redundant computations are performed. If the number is too small the computation has to be reinvoked for a larger number of neighbors, thereby performing redundant computations. Therefore, Hjalton and Samet [HS95] introduced *incremental nearest neighbor searching* in the sense that having obtained the k nearest neighbors, the $k + 1^{st}$ neighbor can be obtained without having to calculate the $k + 1$ nearest neighbor from scratch.

Spatial searching typically consists of a preprocessing phase and a searching phase. In the preprocessing phase one builds a search structure and in the searching phase one makes the queries. In the preprocessing phase the user builds a tree data structure storing the spatial data. In the searching phase the user invokes a searching method to browse the spatial data.

With relatively minor modifications, nearest neighbor searching algorithms can be used to find the furthest object from the query object. Therefore, *furthest neighbor searching* is also supported by the spatial searching package.

The execution time for exact neighbor searching can be reduced by relaxing the requirement that the neighbors should be computed exactly. If the distances of two objects to the query object are approximately the same, instead of computing the nearest/furthest neighbor exactly, one of these objects may be returned as the approximate nearest/furthest neighbor. I.e., given some non-negative constant ϵ the distance of an object returned as an approximate k -nearest neighbor must not be larger than $(1 + \epsilon)r$, where r denotes the distance to the real k^{th} nearest neighbor. Similar the distance of an approximate k -furthest neighbor must not be smaller than $r/(1 + \epsilon)$. Obviously, for $\epsilon = 0$ we get the exact result, and the larger ϵ is, the less exact the result.

Neighbor searching is implemented by the following four classes.

The class `CGAL::Orthogonal_k_neighbor_search<Traits, OrthogonalDistance, Splitter, SpatialTree>` implements the standard search strategy for orthogonal distances like the weighted Minkowski distance. It requires the use of extended nodes in the spatial tree and supports only k neighbor searching for point queries.

The class `CGAL::K_neighbor_search<Traits, GeneralDistance, Splitter, SpatialTree>` implements the standard search strategy for general distances like the Manhattan distance for iso-rectangles. It does not require the use of extended nodes in the spatial tree and supports only k neighbor searching for queries defined by points or spatial objects.

The class `Orthogonal_incremental_neighbor_search<Traits, GeneralDistance, Splitter, SpatialTree>` implements the incremental search strategy for general distances like the weighted Minkowski distance. It requires the use of extended nodes in the spatial tree and supports incremental neighbor searching and distance browsing for point queries.

The class `CGAL::Incremental_neighbor_search<Traits, GeneralDistance, Splitter, SpatialTree>` implements the incremental search strategy for general distances like the Manhattan distance for iso-rectangles. It does not require the use of extended nodes in the spatial tree and supports incremental neighbor searching and distance browsing for queries defined by points or spatial objects.

35.1.2 Range Searching

Exact range searching and *approximate range searching* is supported using exact or fuzzy d -dimensional objects enclosing a region. The fuzziness of the query object is specified by a parameter ϵ denoting a maximal allowed distance to the boundary of a query object. If the distance to the boundary is at least ϵ , points inside the object are always reported and points outside the object are never reported. Points within distance ϵ to the boundary may be or may be not reported. For exact range searching the fuzziness parameter ϵ is set to zero.

The class `Kd_tree` implements range searching in the method `search`, which is a template method with an output iterator and a model of the concept `FuzzyQueryItem` as `CGAL::Fuzzy_iso_box_d` or `CGAL::Fuzzy_sphere_d`.

For range searching of large data sets the user may set the parameter *bucket_size* used in building the *k-d* tree to a large value (e.g. 100), because in general the query time will be less then using the default value.

35.2 Splitting Rules

Instead of using the default splitting rule *Sliding_midpoint* described below, a user may, depending upon the data, select one from the following splitting rules, which determine how a separating hyperplane is computed:

Midpoint_of_rectangle This splitting rule cuts a rectangle through its midpoint orthogonal to the longest side.

Midpoint_of_max_spread This splitting rule cuts a rectangle through $(Mind + Maxd)/2$ orthogonal to the dimension with the maximum point spread $[Mind, Maxd]$.

Sliding_midpoint This is a modification of the midpoint of rectangle splitting rule. It first attempts to perform a midpoint of rectangle split as described above. If data points lie on both sides of the separating plane the sliding midpoint rule computes the same separator as the midpoint of rectangle rule. If the data points lie only on one side it avoids this by sliding the separator, computed by the midpoint of rectangle rule, to the nearest datapoint.

Median_of_rectangle The splitting dimension is the dimension of the longest side of the rectangle. The splitting value is defined by the median of the coordinates of the data points along this dimension.

Median_of_max_spread The splitting dimension is the dimension of the longest side of the rectangle. The splitting value is defined by the median of the coordinates of the data points along this dimension.

Fair This splitting rule is a compromise between the median of rectangle splitting rule and the midpoint of rectangle splitting rule. This splitting rule maintains an upper bound on the maximal allowed ratio of the longest and shortest side of a rectangle (the value of this upper bound is set in the constructor of the fair splitting rule). Among the splits that satisfy this bound, it selects the one in which the points have the largest spread. It then splits the points in the most even manner possible, subject to maintaining the bound on the ratio of the resulting rectangles.

Sliding_fair This splitting rule is a compromise between the fair splitting rule and the sliding midpoint rule. Sliding fair-split is based on the theory that there are two types of splits that are good: balanced splits that produce fat rectangles, and unbalanced splits provided the rectangle with fewer points is fat.

Also, this splitting rule maintains an upper bound on the maximal allowed ratio of the longest and shortest side of a rectangle (the value of this upper bound is set in the constructor of the fair splitting rule). Among the splits that satisfy this bound, it selects the one one in which the points have the largest spread. It then considers the most extreme cuts that would be allowed by the aspect ratio bound. This is done by dividing the longest side of the rectangle by the aspect ratio bound. If the median cut lies between these extreme cuts, then we use the median cut. If not, then consider the extreme cut that is closer to the median. If all the points lie to one side of this cut, then we slide the cut until it hits the first point. This may violate the aspect ratio bound, but will never generate empty cells.

35.3 Example Programs

We give six examples. The first example illustrates *k* nearest neighbor searching, and the second example incremental neighbor searching. The third is an example of approximate furthest neighbor searching using a *d*-dimensional iso-rectangle as an query object. Approximate range searching is illustrated by the fourth example. The fifth example illustrates *k* neighbour searching for a user defined point class. The last example shows how to choose another splitting rule in the *k-d* tree that is used as search tree.

35.3.1 Example of K Neighbor Searching

The first example illustrates k neighbor searching with an Euclidean distance and 2-dimensional points. The generated random data points are inserted in a search tree. We then initialize the k neighbor search object with the origin as query. Finally, we obtain the result of the computation in the form of an iterator range. The value of the iterator is a pair of a point and its square distance to the query point. We use square distances, or *transformed distances* for other distance classes, as they are computationally cheaper.

```
// file: examples/Spatial_searching/Nearest_neighbor_searching.C

#include <CGAL/Simple_cartesian.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/Orthogonal_k_neighbor_search.h>
#include <CGAL/Search_traits_2.h>
#include <list>
#include <cmath>

typedef CGAL::Simple_cartesian<double> K;
typedef K::Point_2 Point_d;
typedef CGAL::Search_traits_2<K> TreeTraits;
typedef CGAL::Orthogonal_k_neighbor_search<TreeTraits> Neighbor_search;
typedef Neighbor_search::Tree Tree;

int main() {
    const int N = 1;

    std::list<Point_d> points;
    points.push_back(Point_d(0,0));

    Tree tree(points.begin(), points.end());

    Point_d query(0,0);

    // Initialize the search structure, and search all N points

    Neighbor_search search(tree, query, N);

    // report the N nearest neighbors and their distance
    // This should sort all N points by increasing distance from origin
    for(Neighbor_search::iterator it = search.begin(); it != search.end(); ++it){
        std::cout << it->first << " " << std::sqrt(it->second) << std::endl;
    }

    return 0;
}
```


35.3.2 Example of Incremental Searching

This example program illustrates incremental searching for the closest point with a positive first coordinate. We can use the orthogonal incremental neighbor search class, as the query is also a point and as the distance is the Euclidean distance.

As for the k neighbor search, we first initialize the search tree with the data. We then create the search object, and finally obtain the iterator with the *begin()* method. Note that the iterator is of the input iterator category, that is one can make only one pass over the data.

```
// file: examples/Spatial_searching/Distance_browsing.C

#include <CGAL/Simple_cartesian.h>
#include <CGAL/Orthogonal_incremental_neighbor_search.h>
#include <CGAL/Search_traits_2.h>

typedef CGAL::Simple_cartesian<double> K;
typedef K::Point_2 Point_d;
typedef CGAL::Search_traits_2<K> TreeTraits;
typedef CGAL::Orthogonal_incremental_neighbor_search<TreeTraits> NN_incremental_search;
typedef NN_incremental_search::iterator NN_iterator;
typedef NN_incremental_search::Tree Tree;

// A functor that returns true, iff the x-coordinate of a dD point is not positive
struct X_not_positive {
    bool operator()(const NN_iterator& it) { return ((*it).first)[0]<0; }
};

// An iterator that only enumerates dD points with positive x-coordinate
typedef CGAL::Filter_iterator<NN_iterator, X_not_positive> NN_positive_x_iterator;

int main() {

    Tree tree;
    tree.insert(Point_d(0,0));
    tree.insert(Point_d(1,1));
    tree.insert(Point_d(0,1));
    tree.insert(Point_d(10,110));
    tree.insert(Point_d(45,0));
    tree.insert(Point_d(0,2340));
    tree.insert(Point_d(0,30));

    Point_d query(0,0);

    NN_incremental_search NN(tree, query);
    NN_positive_x_iterator it(NN.end(), X_not_positive(), NN.begin(), end(NN.end(), X_not_positive()));

    std::cout << "The first 5 nearest neighbours with positive x-coord are: " << std::endl;
    for (int j=0; (j < 5)&&(it!=end); ++j,++it)
        std::cout << (*it).first << " at squared distance = " << (*it).second << std::endl;

    return 0;
}
```


35.3.3 Example of General Neighbor Searching

This example program illustrates approximate nearest and furthest neighbor searching using 4-dimensional Cartesian coordinates. Five approximate nearest neighbors of the query rectangle $[0.1, 0.2]^4$ are computed. Because the query object is a rectangle we cannot use the Orthogonal neighbor search. As in the previous examples we first initialize a search tree, create the search object with the query, and obtain the result of the search as iterator range.

```
// file: examples/Spatial_searching/General_neighbor_searching.C

#include <CGAL/Cartesian_d.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/Manhattan_distance_iso_box_point.h>
#include <CGAL/K_neighbor_search.h>
#include <CGAL/Search_traits_2.h>

typedef CGAL::Cartesian_d<double> K;
typedef K::Point_d Point_d;
typedef CGAL::Random_points_in_square_2<Point_d> Random_points_iterator;
typedef K::Iso_box_d Iso_box_d;
typedef K TreeTraits;
typedef CGAL::Manhattan_distance_iso_box_point<TreeTraits> Distance;
typedef CGAL::K_neighbor_search<TreeTraits, Distance> Neighbor_search;
typedef Neighbor_search::Tree Tree;

int main() {
    const int N = 1000;
    const int K = 10;

    Tree tree;
    Random_points_iterator rpg;
    for(int i = 0; i < N; i++){
        tree.insert(*rpg++);
    }
    Point_d pp(0.1,0.1);
    Point_d qq(0.2,0.2);
    Iso_box_d query(pp,qq);

    Distance tr_dist;
    Neighbor_search N1(tree, query, K, 0.0, false); // eps=10.0, nearest=false

    std::cout << "For query rectangle = [0.1,0.2]^2 " << std::endl
        << "The " << K << " approximate furthest neighbors are: " << std::endl;
    for (Neighbor_search::iterator it = N1.begin(); it != N1.end(); it++) {
        std::cout << " Point " << it->first << " at distance = " << tr_dist.inverse_of_transformed_distance(it)
    }
    return 0;
}
```

35.3.4 Example of a Range Query

This example program illustrates approximate range querying for 4-dimensional fuzzy iso-rectangles and spheres using homogeneous coordinates. The range queries are member functions of the k - d tree class.

```
// file: examples/Spatial_searching/Fuzzy_range_query.C
#include <CGAL/Cartesian_d.h>
#include <CGAL/point_generators_d.h>
#include <CGAL/Kd_tree.h>
#include <CGAL/Fuzzy_sphere.h>
#include <CGAL/Fuzzy_iso_box.h>
#include <CGAL/Search_traits_d.h>

typedef CGAL::Cartesian_d<double> K;
typedef K::Point_d Point_d;
typedef CGAL::Search_traits_d<K> Traits;
typedef CGAL::Random_points_in_iso_box_d<Point_d> Random_points_iterator;
typedef CGAL::Counting_iterator<Random_points_iterator> N_Random_points_iterator;
typedef CGAL::Kd_tree<Traits> Tree;
typedef CGAL::Fuzzy_sphere<Traits> Fuzzy_sphere;
typedef CGAL::Fuzzy_iso_box<Traits> Fuzzy_iso_box;

int main() {
    const int D = 4;
    const int N = 1000;
    // generator for random data points in the square ( (-1000,-1000), (1000,1000) )
    Random_points_iterator rpit(4, 1000.0);

    // Insert N points in the tree
    Tree tree(N_Random_points_iterator(rpit,0),
             N_Random_points_iterator(N));

    // define range query objects
    double pcoord[D] = { 300, 300, 300, 300 };
    double qcoord[D] = { 900.0, 900.0, 900.0, 900.0 };
    Point_d p(D, pcoord, pcoord+D);
    Point_d q(D, qcoord, qcoord+D);
    Fuzzy_sphere fs(p, 700.0, 100.0);
    Fuzzy_iso_box fib(p, q, 100.0);

    std::cout << "points approximately in fuzzy range query" << std::endl;
    std::cout << "with center (300.0, 300.0, 300.0, 300.0)" << std::endl;
    std::cout << "and fuzzy radius <200.0,400.0> are:" << std::endl;
    tree.search(std::ostream_iterator<Point_d>(std::cout, "\n"), fs);

    std::cout << "points approximately in fuzzy range query ";
    std::cout << "[<200,4000>,<800,1000>]]^4 are:" << std::endl;

    tree.search(std::ostream_iterator<Point_d>(std::cout, "\n"), fib);
    return 0;
}
```

35.3.5 Example Illustrating Use of User Defined Point and Distance Class

The neighbor searching works with all CGAL kernels, as well as with user defined points and distance classes. In this example we assume that the user provides the following 3-dimensional points class.

```
struct Point {
    double vec[3];

    Point() { vec[0]= vec[1] = vec[2] = 0; }
    Point (double x, double y, double z) { vec[0]=x; vec[1]=y; vec[2]=z; }

    double x() const { return vec[ 0 ]; }
    double y() const { return vec[ 1 ]; }
    double z() const { return vec[ 2 ]; }

    double& x() { return vec[ 0 ]; }
    double& y() { return vec[ 1 ]; }
    double& z() { return vec[ 2 ]; }

    bool operator==(const Point& p) const
    {
        return (x() == p.x()) && (y() == p.y()) && (z() == p.z()) ;
    }

    bool operator!=(const Point& p) const { return ! (*this == p); }
}; //end of class

namespace CGAL {

    template <>
    struct Kernel_traits<Point> {
        struct Kernel {
            typedef double FT;
            typedef double RT;
        };
    };

}

struct Construct_coord_iterator {
    const double* operator()(const Point& p) const
    { return static_cast<const double*>(p.vec); }

    const double* operator()(const Point& p, int) const
    { return static_cast<const double*>(p.vec+3); }
};
```

We have put the glue layer in this file as well, that is a class that allows to iterate over the Cartesian coordinates of the point, and a class to construct such an iterator for a point. We next need a distance class

```

struct Distance {
    typedef Point Query_item;

    double transformed_distance(const Point& p1, const Point& p2) const {
        double distx= p1.x()-p2.x();
        double disty= p1.y()-p2.y();
        double distz= p1.z()-p2.z();
        return distx*distx+disty*disty+distz*distz;
    }

    template <class TreeTraits>
    double min_distance_to_rectangle(const Point& p,
        const CGAL::Kd_tree_rectangle<TreeTraits>& b) const {
        double distance(0.0), h = p.x();
        if (h < b.min_coord(0)) distance += (b.min_coord(0)-h) * (b.min_coord(0)-h);
        if (h > b.max_coord(0)) distance += (h-b.max_coord(0)) * (h-b.max_coord(0));
        h=p.y();
        if (h < b.min_coord(1)) distance += (b.min_coord(1)-h) * (b.min_coord(1)-h);
        if (h > b.max_coord(1)) distance += (h-b.max_coord(1)) * (h-b.min_coord(1));
        h=p.z();
        if (h < b.min_coord(2)) distance += (b.min_coord(2)-h) * (b.min_coord(2)-h);
        if (h > b.max_coord(2)) distance += (h-b.max_coord(2)) * (h-b.max_coord(2));
        return distance;
    }

    template <class TreeTraits>
    double max_distance_to_rectangle(const Point& p,
        const CGAL::Kd_tree_rectangle<TreeTraits>& b) const {
        double h = p.x();

        double d0 = (h >= (b.min_coord(0)+b.max_coord(0))/2.0) ?
            (h-b.min_coord(0)) * (h-b.min_coord(0)) : (b.max_coord(0)-h) * (b.max_coord(0)-h);

        h=p.y();
        double d1 = (h >= (b.min_coord(1)+b.max_coord(1))/2.0) ?
            (h-b.min_coord(1)) * (h-b.min_coord(1)) : (b.max_coord(1)-h) * (b.max_coord(1)-h);
        h=p.z();
        double d2 = (h >= (b.min_coord(2)+b.max_coord(2))/2.0) ?
            (h-b.min_coord(2)) * (h-b.min_coord(2)) : (b.max_coord(2)-h) * (b.max_coord(2)-h);
        return d0 + d1 + d2;
    }

    double new_distance(double& dist, double old_off, double new_off,
        int cutting_dimension) const {
        return dist + new_off*new_off - old_off*old_off;
    }

    double transformed_distance(double d) const { return d*d; }

    double inverse_of_transformed_distance(double d) { return std::sqrt(d); }
}; // end of struct Distance

```

We are ready to put the pices together. The class *Search_traits<..>* which you see in the next file is then a mere wrapper for all these types. The searching itself works exactly as for CGAL kernels.

```
//file: examples/Spatial_searching/User_defined_point_and_distance.C

#include <CGAL/basic.h>
#include <CGAL/Search_traits.h>
#include <CGAL/point_generators_3.h>
#include <CGAL/Orthogonal_k_neighbor_search.h>
#include "Point.h" // defines types Point, Construct_coord_iterator
#include "Distance.h"

typedef CGAL::Random_points_in_cube_3<Point> Random_points_iterator;
typedef CGAL::Counting_iterator<Random_points_iterator> N_Random_points_iterator;
typedef CGAL::Search_traits<double, Point, const double*, Construct_coord_iterator> Traits;
typedef CGAL::Orthogonal_k_neighbor_search<Traits, Distance> K_neighbor_search;
typedef K_neighbor_search::Tree Tree;

int main() {
    const int N = 1000;
    const int K = 5;
    // generator for random data points in the cube ( (-1,-1,-1), (1,1,1) )
    Random_points_iterator rpit( 1.0);

    // Insert number_of_data_points in the tree
    Tree tree(N_Random_points_iterator(rpit,0),
        N_Random_points_iterator(N));

    Point query(0.0, 0.0, 0.0);
    Distance tr_dist;

    // search K nearest neighbours
    K_neighbor_search search(tree, query, K);
    for(K_neighbor_search::iterator it = search.begin(); it != search.end(); it++){
        std::cout << " d(q, nearest neighbor)= "
            << tr_dist.inverse_of_transformed_distance(it->second) << std::endl;
    }
    // search K furthest neighbour searching, with eps=0, search_nearest=false
    K_neighbor_search search2(tree, query, K, 0.0, false);

    for(K_neighbor_search::iterator it = search2.begin(); it != search2.end(); it++){
        std::cout << " d(q, furthest neighbor)= "
            << tr_dist.inverse_of_transformed_distance(it->second) << std::endl;
    }
    return 0;
}
```

35.3.6 Example of Selecting a Splitting Rule and Setting the Bucket Size

This example program illustrates selecting a splitting rule and setting the maximal allowed bucket size. The only differences with the first example are the declaration of the *Fair* splitting rule, needed to set the maximal allowed bucket size.

```
// file: examples/Spatial_searching/Using_fair_splitting_rule.C

#include <CGAL/Simple_cartesian.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/Search_traits_2.h>
#include <CGAL/Orthogonal_k_neighbor_search.h>
#include <cmath>

typedef CGAL::Simple_cartesian<double> R;
typedef R::Point_2 Point_d;
typedef CGAL::Random_points_in_square_2<Point_d> Random_points_iterator;
typedef CGAL::Counting_iterator<Random_points_iterator> N_Random_points_iterator;
typedef CGAL::Search_traits_2<R> Traits;
typedef CGAL::Euclidean_distance<Traits> Distance;
typedef CGAL::Fair<Traits> Fair;
typedef CGAL::Orthogonal_k_neighbor_search<Traits,Distance,Fair> Neighbor_search;
typedef Neighbor_search::Tree Tree;

int main() {
    const int N = 1000;
    // generator for random data points in the square ( (-1,-1), (1,1) )
    Random_points_iterator rpit( 1.0);

    Fair fair(5); // bucket size=5
    // Insert number_of_data_points in the tree
    Tree tree(N_Random_points_iterator(rpit,0),
        N_Random_points_iterator(N),
        fair);

    Point_d query(0,0);

    // Initialize the search structure, and search all N points
    Neighbor_search search(tree, query, N);

    // report the N nearest neighbors and their distance
    // This should sort all N points by increasing distance from origin
    for(Neighbor_search::iterator it = search.begin(); it != search.end(); ++it){
        std::cout << it->first << " " << std::sqrt(it->second) << std::endl;
    }
    return 0;
}
```


35.4 Software Design

35.4.1 The k - d tree

Bentley [Ben75] introduced the k - d tree as a generalization of the binary search tree in higher dimensions. k - d trees hierarchically decompose space into a relatively small number of rectangles such that no rectangle contains too many input objects. For our purposes, a *rectangle* in real d dimensional space, \mathbb{R}^d , is the product of d closed intervals on the coordinate axes. k - d trees are obtained by partitioning point sets in \mathbb{R}^d using $(d-1)$ -dimensional hyperplanes. Each node in the tree is split into two children by one such separating hyperplane. Several splitting rules (see Section 35.2) can be used to compute a separating $(d-1)$ -dimensional hyperplane.

Each internal node of the k - d tree is associated with a rectangle and a hyperplane orthogonal to one of the coordinate axis, which splits the rectangle into two parts. Therefore, such a hyperplane, defined by a splitting dimension and a splitting value, is called a separator. These two parts are then associated with the two child nodes in the tree. The process of partitioning space continues until the number of data points in the rectangle falls below some given threshold. The rectangles associated with the leaf nodes are called *buckets*, and they define a subdivision of the space into rectangles. Data points are only stored in the leaf nodes of the tree, not in the internal nodes.

Friedmann, Bentley and Finkel [FBF77] described the standard search algorithm to find the k th nearest neighbor by searching a k - d tree recursively.

When encountering a node of the tree, the algorithm first visits the child that is closest to the query point. On return, if the rectangle containing the other child lies within $1/(1+\epsilon)$ times the distance to the k th nearest neighbors so far, then the other child is visited recursively. Priority search [AM93b] visits the nodes in increasing order of distance from the queue with help of a priority queue. The search stops when the distance of the query point to the nearest nodes exceeds the distance to the nearest point found with a factor $1/(1+\epsilon)$. Priority search supports next neighbor search, standard search does not.

In order to speed-up the internal distance computations in nearest neighbor searching in high dimensional space, the approximate searching package supports orthogonal distance computation. Orthogonal distance computation implements the efficient incremental distance computation technique introduced by Arya and Mount [AM93a]. This technique works only for neighbor queries with query items represented as points and with a quadratic form distance, defined by $d_A(x, y) = (x - y)A(x - y)^T$, where the matrix A is positive definite, i.e. $d_A(x, y) \geq 0$. An important class of quadratic form distances are weighted Minkowski distances. Given a parameter $p > 0$ and parameters $w_i \geq 0, 0 < i \leq d$, the weighted Minkowski distance is defined by $l_p(w)(r, q) = (\sum_{i=1}^d w_i(r_i - q_i)^p)^{1/p}$ for $0 < p < \infty$ and defined by $l_\infty(w)(r, q) = \max\{w_i|r_i - q_i| \mid 1 \leq i \leq d\}$. The Manhattan distance ($p = 1, w_i = 1$) and the Euclidean distance ($p = 2, w_i = 1$) are examples of a weighted Minkowski metric.

To speed up distance computations also transformed distances are used instead of the distance itself. For instance for the Euclidean distance, to avoid the expensive computation of square roots, squared distances are used instead of the Euclidean distance itself.

dD Spatial Searching Reference Manual

Hans Tangelder and Andreas Fabri

This package provides data structures and algorithms for exact and approximate distance browsing, supporting

- both nearest and furthest neighbor searching,
- both exact and approximate searching,
- (approximate) range searching,
- (approximate) k -nearest and k -furthest neighbor searching,
- (approximate) incremental nearest and incremental furthest neighbor searching,
- query items representing points and spatial objects.

The spatial searching package consists of the following concepts and classes that are described in the reference pages.

35.5 Classified Reference Pages

Search Classes

CGAL::K_neighbor_search<Traits, GeneralDistance, Splitter, SpatialTree> page [2067](#)
CGAL::Incremental_neighbor_search<Traits, GeneralDistance, Splitter, SpatialTree> page [2065](#)
CGAL::Orthogonal_incremental_neighbor_search<Traits, OrthogonalDistance, Splitter, SpatialTree>
page [2084](#)
CGAL::Orthogonal_k_neighbor_search<Traits, OrthogonalDistance, Splitter, SpatialTree> page [2086](#)
CGAL::Kd_tree<Traits, Splitter, UseExtendedNode> page [2069](#)

Range Query Item Classes

CGAL::Fuzzy_iso_box<Traits> page [2060](#)
CGAL::Fuzzy_sphere<Traits> page [2062](#)

Search Traits Classes

<i>CGAL::Search_traits_2<Kernel></i>	page 2091
<i>CGAL::Search_traits_3<Kernel></i>	page 2093
<i>CGAL::Search_traits_d<Kernel></i>	page 2095
<i>CGAL::Search_traits<NT,Point,CartesianIterator,ConstructCartesianIterator,ConstructMinVertex,ConstructMaxVertex></i>	
page 2097	

Distance Classes

<i>CGAL::Euclidean_distance<Traits></i>	page 2054
<i>CGAL::Euclidean_distance_sphere_point<Traits></i>	page 2056
<i>CGAL::Manhattan_distance_iso_box_point<Traits></i>	page 2076
<i>CGAL::Weighted_Minkowski_distance<Traits></i>	page 2106

Splitter Classes

<i>CGAL::Sliding_midpoint<Traits, SpatialSeparator></i>	page 2101
<i>CGAL::Sliding_fair<Traits, SpatialSeparator></i>	page 2099
<i>CGAL::Fair<Traits, SpatialSeparator></i>	page 2058
<i>CGAL::Median_of_max_spread<Traits, SpatialSeparator></i>	page 2078
<i>CGAL::Median_of_rectangle<Traits, SpatialSeparator></i>	page 2079
<i>CGAL::Midpoint_of_max_spread<Traits, SpatialSeparator></i>	page 2080
<i>CGAL::Midpoint_of_rectangle<Traits, SpatialSeparator></i>	page 2081

Advanced Classes

<i>CGAL::Kd_tree_node<Traits, Splitter, UseExtendedNode></i>	page 2072
<i>CGAL::Kd_tree_rectangle<Traits></i>	page 2074
<i>CGAL::Plane_separator<FT></i>	page 2088

Concepts

<i>FuzzyQueryItem</i>	page 2059
<i>GeneralDistance</i>	page 2064
<i>OrthogonalDistance</i>	page 2082
<i>SearchTraits</i>	page 2090
<i>SpatialSeparator</i>	page 2102
<i>SpatialTree</i>	page 2103

35.6 Alphabetical List of Reference Pages

<i>Euclidean_distance<Traits></i>	page 2054
<i>Euclidean_distance_sphere_point<Traits></i>	page 2056
<i>Fair<Traits, SpatialSeparator></i>	page 2058
<i>FuzzyQueryItem</i>	page 2059
<i>Fuzzy_iso_box<Traits></i>	page 2060
<i>Fuzzy_sphere<Traits></i>	page 2062

<i>GeneralDistance</i>	page 2064
<i>Incremental_neighbor_search</i> <Traits, GeneralDistance, Splitter, SpatialTree>	page 2065
<i>Kd_tree</i> <Traits, Splitter, UseExtendedNode>	page 2069
<i>Kd_tree_node</i> <Traits, Splitter, UseExtendedNode>	page 2072
<i>Kd_tree_rectangle</i> <Traits>	page 2074
<i>K_neighbor_search</i> <Traits, GeneralDistance, Splitter, SpatialTree>	page 2067
<i>Manhattan_distance_iso_box_point</i> <Traits>	page 2076
<i>Median_of_max_spread</i> <Traits, SpatialSeparator>	page 2078
<i>Median_of_rectangle</i> <Traits, SpatialSeparator>	page 2079
<i>Midpoint_of_max_spread</i> <Traits, SpatialSeparator>	page 2080
<i>Midpoint_of_rectangle</i> <Traits, SpatialSeparator>	page 2081
<i>OrthogonalDistance</i>	page 2082
<i>Orthogonal_incremental_neighbor_search</i> <Traits, OrthogonalDistance, Splitter, SpatialTree>	page 2084
<i>Orthogonal_k_neighbor_search</i> <Traits, OrthogonalDistance, Splitter, SpatialTree>	page 2086
<i>Plane_separator</i> <FT>	page 2088
<i>SearchTraits</i>	page 2090
<i>Search_traits</i> <NT,Point, CartesianIterator, ConstructCartesianIterator, ConstructMinVertex, ConstructMaxVertex> page 2097	
<i>Search_traits_2</i> <Kernel>	page 2091
<i>Search_traits_3</i> <Kernel>	page 2093
<i>Search_traits_d</i> <Kernel>	page 2095
<i>Sliding_fair</i> <Traits, SpatialSeparator>	page 2099
<i>Sliding_midpoint</i> <Traits, SpatialSeparator>	page 2101
<i>SpatialSeparator</i>	page 2102
<i>SpatialTree</i>	page 2103
<i>Splitter</i>	page 2105
<i>Weighted_Minkowski_distance</i> <Traits>	page 2106

CGAL::Euclidean_distance<Traits>

Definition

The class *Euclidean_distance<Traits>* provides an implementation of the concept *OrthogonalDistance*, with the Euclidean distance (l_2 metric). To optimize distance computations squared distances are used.

```
#include <CGAL/Euclidean_distance.h>
```

Parameters

Expects for the first template argument a model of the concept *SearchTraits*, for example *CGAL::Search_traits_2<CGAL::Cartesian<double>>*.

Is Model for the Concepts

OrthogonalDistance

Types

<i>Traits::FT</i>	<i>FT</i> ;	Number type.
<i>Traits::Point_d</i>	<i>Point_d</i> ;	Point type.
<i>Point_d</i>	<i>Query_item</i> ;	Query item type.

Creation

Euclidean_distance<Traits> ed;

Default constructor.

Operations

FT *ed.transformed_distance(Query_item q, Point_d p)*

Returns the squared Euclidean distance between *q* and *p*.

FT *ed.min_distance_to_rectangle(Query_item q, Kd_tree_rectangle<FT> r)*

Returns the squared Euclidean distance between *q* and the point on the boundary of *r* closest to *q*.

FT *ed.max_distance_to_rectangle(Query_item q, Kd_tree_rectangle<FT> r;)*

Returns the squared Euclidean distance, where *d* denotes the distance between *q* and the point on the boundary of *r* farthest to *q*.

FT *ed.new_distance(FT dist, FT old_off, FT new_off, int cutting_dimension)*

Updates the squared *dist* incrementally and returns the updated squared distance.

FT *ed.transformed_distance(FT d)*

Returns d^2 .

FT *ed.inverse_of_transformed_distance(FT d)*

Returns $d^{1/2}$.

See Also

OrthogonalDistance
CGAL::Weighted_Minkowski_distance<Traits>.

NT *ed.max_distance_to_rectangle(Query_item s, Kd_tree_rectangle<Traits> r)*

Returns the maximal distance between the sphere s and a point from r furthest to s .

NT *ed.transformed_distance(NT d)*

Returns d^2 .

NT *ed.inverse_of_transformed_distance(NT d)*

Returns $d^{1/2}$.

See Also

GeneralDistance

FuzzyQueryItem

Definition

The concept FuzzyQueryItem describes the requirements for fuzzy d -dimensional spatial objects.

Has Models

CGAL::Fuzzy_sphere<Traits>,
CGAL::Fuzzy_iso_box<Traits>

Types

FuzzyQueryItem:: Point_d represents a d -dimensional point.

FuzzyQueryItem:: FT Number type.

Operations

bool *q.contains(Point_d p)*
 test whether q contains p .

bool *q.inner_range_intersects(Kd_tree_rectangle<Traits> rectangle)*
 test whether the inner approximation of the spatial object intersects a rectangle associated with a node of a tree.

bool *q.outer_range_contains(Kd_tree_rectangle<Traits> rectangle)*
 test whether the outer approximation of the spatial object encloses the rectangle associated with a node of a tree.

CGAL::Fuzzy_iso_box<Traits>

Definition

The class *Fuzzy_iso_box<Traits>* implements fuzzy d -dimensional iso boxes. A fuzzy iso box with fuzziness value ϵ has as outer approximation a box dilated, and as inner approximation a box eroded by a d -dim square with side length ϵ .

```
#include <CGAL/Fuzzy_iso_box.h>
```

Parameters

Expects for the template argument a model of the concept *SearchTraits*, for example *CGAL::Search_traits_2<CGAL::Simple_cartesian<double>>*.

Is Model for the Concepts

FuzzyQueryItem

Types

<i>Traits::Point_d</i>	<i>Point_d</i> ;	Point type.
<i>Traits::FT</i>	<i>FT</i> ;	Number type.

Creation

```
Fuzzy_iso_box<Traits> b( Point_d p, Point_d q, FT epsilon=FT(0));
```

Constructs a fuzzy iso box specified by the minimal iso box containing p and q and fuzziness value *epsilon*.

Precondition: p must be lexicographically smaller than q .

Operations

<i>bool</i>	<i>b.contains(Point_d p)</i>	test whether b contains p .
-------------	-------------------------------	---------------------------------

<i>bool</i>	<i>b.inner_range_intersects(Kd_tree_rectangle<FT> rectangle)</i>	test whether the inner box intersects the rectangle associated with a node of a tree.
-------------	-------------------------------------------------------------------------	---------------------------------------------------------------------------------------

bool

b.outer_range_contains(Kd_tree_rectangle<FT> rectangle)

test whether the outer box encloses the rectangle associated with a node of a tree.

See Also

FuzzyQueryItem

CGAL::Fuzzy_sphere<Traits>

Definition

The class *Fuzzy_sphere<Traits>* implements fuzzy d -dimensional spheres. A fuzzy sphere with radius r and fuzziness value ϵ has as outer approximation a sphere with radius $r + \epsilon$ and as inner approximation a sphere with radius $r - \epsilon$.

#include <CGAL/Fuzzy_sphere.h>

Parameters

Expects for the template argument a model of the concept *SearchTraits*, for example *CGAL::Cartesian_d<double>*.

Is Model for the Concepts

FuzzyQueryItem

Types

<i>Traits::Point_d</i>	<i>Point_d</i> ;	Point type.
<i>Traits::FT</i>	<i>FT</i> ;	Number type.

Creation

Fuzzy_sphere<Traits> s(Point_d center, FT radius, FT epsilon=FT(0));

Constructs a fuzzy sphere centered at *center* with radius *radius* and fuzziness value *epsilon*.

Operations

bool s.contains(Point_d p) test whether *s* contains *p*.

bool s.inner_range_intersects(Kd_tree_rectangle<FT> rectangle)

test whether the inner sphere intersects the rectangle associated with a node of a tree.

bool s.outer_range_contains(Kd_tree_rectangle<FT> rectangle)

test whether the outer sphere encloses the rectangle associated with a node of a tree.

See Also

FuzzyQueryItem

GeneralDistance

Definition

Requirements of a distance class defining a distance between a query item denoting a spatial object and a point. To optimize distance computations transformed distances are used, e.g., for a Euclidean distance the transformed distance is the squared Euclidean distance.

Has Models

CGAL::Manhattan_distance_rectangle_point<Traits, IsoBox>
CGAL::Euclidean_distance_sphere_point<Traits, Sphere>.

Types

<i>GeneralDistance:: FT</i>	Number type.
<i>GeneralDistance:: Point_d</i>	Point type.
<i>GeneralDistance:: Query_item</i>	Query item type.

Operations

<i>FT</i>	<i>gd.transformed_distance(Query_item q, Point_d r)</i>	Returns the transformed distance between <i>q</i> and <i>r</i> .
<i>FT</i>	<i>gd.min_distance_to_rectangle(Query_item q, Kd_tree_rectangle<FT> r)</i>	Returns the transformed distance between <i>q</i> and the point on the boundary of <i>r</i> closest to <i>q</i> .
<i>FT</i>	<i>gd.max_distance_to_rectangle(Query_item q, Kd_tree_rectangle<FT> r)</i>	Returns the transformed distance between <i>q</i> and the point on the boundary of <i>r</i> furthest to <i>q</i> .
<i>FT</i>	<i>gd.transformed_distance(FT d)</i>	Returns the transformed distance.
<i>FT</i>	<i>gd.inverse_of_transformed_distance(FT d)</i>	Returns the inverse of the transformed distance.

CGAL::Incremental_neighbor_search<Traits, GeneralDistance, Splitter, SpatialTree>

Definition

The class *Incremental_neighbor_search*<Traits, GeneralDistance, Splitter, SpatialTree> implements incremental nearest and furthest neighbor searching on a tree. The tree may have extended or unextended nodes.

```
#include <CGAL/Incremental_neighbor_search.h>
```

Parameters

Expects for the first template argument a model of the concept *SearchTraits*, for example *CGAL::Search_traits_2*<*CGAL::Cartesian*<*double*>>.

Expects for the second template argument a model of the concept *GeneralDistance*. The default type is *CGAL::Euclidean_distance*<Traits>.

Expects for third template argument a model of the concept *Splitter*. The default type is *CGAL::Sliding_midpoint*<Traits>.

Expects for fourth template argument a model of the concept *SpatialTree*. The default type is *CGAL::Kd_tree*<Traits, Splitter, CGAL::Tag_false>. The template argument *CGAL::Tag_false* makes that the tree is built with unextended nodes.

Types

Traits::Point_d *Point_d*; Point type.

Traits::NT *NT*; Number type.

std::pair<*Point_d*, *NT*>

Point_with_transformed_distance;

Pair of point and transformed distance.

Incremental_neighbor_search<Traits, GeneralDistance, Splitter, SpatialTree>:: *iterator*

Input iterator with value type *Point_with_transformed_distance* for enumerating approximate neighbors.

GeneralDistance::Query_item

Query_item; Query item type.

SpatialTree *Tree*; The tree type.

Creation

```
Incremental_neighbor_search<Traits, GeneralDistance, Splitter, SpatialTree> s(
    Tree& tree,
    QueryItem q,
    NT eps=NT(0.0),
    bool search_nearest=true,
    GeneralDistance d=GeneralDistance())
```

Constructor for incremental neighbor searching of the query item q in the points stored $tree$ using a distance d and approximation factor eps .

Operations

<i>iterator</i>	<i>s.begin()</i>	Returns an iterator to the approximate nearest or furthest neighbor.
-----------------	------------------	----------------------------------------------------------------------

<i>iterator</i>	<i>s.end()</i>	Past-the-end iterator.
-----------------	----------------	------------------------

<i>advanced</i>		
<i>std::ostream&</i>	<i>s.statistics(std::ostream& s)</i>	Inserts statistics of the search process into the output stream s .

<i>advanced</i>

See Also

CGAL::Orthogonal_incremental_neighbor_search<Traits, OrthogonalDistance, Splitter, SpatialTree>.

CGAL::K_neighbor_search<Traits, GeneralDistance, Splitter, SpatialTree>

Definition

The class *K_neighbor_search*<Traits, GeneralDistance, Splitter, SpatialTree> implements approximate *k*-nearest and *k*-furthest neighbor searching using standard search on a tree using a general distance class. The tree may be built with extended or unextended nodes.

```
#include <CGAL/K_neighbor_search.h>
```

Parameters

Expects for the first template argument an implementation of the concept *SearchTraits*, for example *CGAL::Cartesian_d*<double>.

Expects for the second template argument a model of the concept *GeneralDistance*. The default type is *CGAL::Euclidean_distance*<Traits>.

Expects for fourth template argument an implementation of the concept *SpatialTree*. The default type is *CGAL::Kd_tree*<Traits, Splitter, CGAL::Tag_false>. The template argument *CGAL::Tag_false* makes that the tree is built with unextended nodes.

Types

Traits::Point_d *Point_d*; Point type.

Traits::FT *FT*; Number type.

std::pair<*Point_d*,*FT*>

Point_with_transformed_distance;

Pair of point and transformed distance.

K_neighbor_search<Traits, GeneralDistance, Splitter, SpatialTree>:: *iterator*

Bidirectional iterator with value type *Point_with_distance* for enumerating approximate neighbors.

GeneralDistance::Query_item

Query_item; Query item type.

SpatialTree *Tree*; The tree type.

Creation

```
K_neighbor_search<Traits, GeneralDistance, Splitter, SpatialTree> s(  
    Tree& tree,  
    Query_item q,  
    int k=1,  
    FT eps=FT(0.0),  
    bool search_nearest=true,  
    GeneralDistance d=GeneralDistance())
```

Constructor for searching approximately k neighbors of the query item q in the points stored in $tree$ using distance class d and approximation factor eps .

Operations

<i>iterator</i>	<i>s.begin()</i>	Returns an iterator to the approximate neighbors.
-----------------	------------------	---------------------------------------------------

<i>iterator</i>	<i>s.end()</i>	Past-the-end iterator.
-----------------	----------------	------------------------

_____ *advanced* _____
|

<i>std::ostream</i> &	<i>s.statistics(std::ostream& s)</i>
-----------------------	--------------------------------------------

Inserts statistics of the search process into the output stream s .

_____ *advanced* _____
|

See Also

CGAL::Orthogonal_k_neighbor_search<*Traits*, *OrthogonalDistance*, *Splitter*, *SpatialTree*>.

CGAL::Kd_tree<Traits, Splitter, UseExtendedNode>

Definition

The class *Kd_tree*<*Traits*, *Splitter*, *UseExtendedNode*> defines a *k-d* tree.

#include <CGAL/Kd_tree.h>

Parameters

Expects for the first template argument a model of the concept *SearchTraits*, for example *CGAL::Search_traits_2*<*CGAL::Cartesian*<*double*> >.

Expects for the second template argument a model for the concept *Splitter*. It defaults to *Sliding_midpoint*<*Traits*>.

Expects for the third template argument *CGAL::Tag_true*, if the tree shall be built with extended nodes, and *CGAL::Tag_false* otherwise.

Types

Traits::Point_d *Point_d*; Point class.

Traits::FT *FT*; Number type.

Kd_tree<*Traits*, *Splitter*, *UseExtendedNode*>::*Splitter*;
Splitter type.

Kd_tree<*Traits*, *Splitter*, *UseExtendedNode*>::*iterator*;
Bidirectional iterator with value type *Point_d* that allows to enumerate all points in the tree.

— advanced —

Kd_tree<*Traits*, *Splitter*, *UseExtendedNode*>::*Node_handle*;
A handle with value type *Kd_tree_node*<*Traits*,*Splitter*>.

Kd_tree<*Traits*, *Splitter*, *UseExtendedNode*>::*Point_d_iterator*;
Random access iterator with value type *Point_d**.

— advanced —

Creation

```
Kd_tree<Traits, Splitter, UseExtendedNode> tree( Splitter s=Splitter());
```

Constructs an empty k - d tree.

[illegible]

Constructs a k - d tree on the elements from the sequence $[first, beyond)$ using the splitting rule implemented by s . The value type of the *InputIterator* must be *Point_d*.

Operations

```
void tree.insert( Point_d p)
```

Inserts the point p in the k - d tree.

```
template <class InputIterator>
void          tree.insert( InputIterator first, InputIterator beyond)
```

Inserts the elements from the sequence $[first, beyond)$ in the k - d tree. The value type of the *InputIterator* must be *Point_d*.

```
template <class OutputIterator, class FuzzyQueryItem>
OutputIterator      tree.search( OutputIterator it, FuzzyQueryItem q)
```

Reports the points that are approximately contained by q . The types *FuzzyQueryItem::Point_d* and *Point_d* must be equivalent.

<i>iterator</i>	<i>tree.begin()</i>
-----------------	---------------------

Returns an iterator to the first point in the tree.

<i>iterator</i>	<i>tree.end()</i>
-----------------	-------------------

Returns the corresponding past-the-end iterator.

```
void tree.clear()
```

Removes all points from the k - d tree.

```
unsigned int      tree.size()
```

Returns the number of points that are stored in the tree.

advanced

Node_handle

tree.root()

Returns a handle to the root node of the tree.

Kd_tree_rectangle<Traits>

tree.bounding_box() returns a const reference to the bounding box of the root node of the tree.

std::ostream& *tree.statistics(std::ostream& s)*

Inserts statistics of the tree into the output stream *s*.

└────────── *advanced* ─────────┘

See Also

Tree. *CGAL::Kd_tree_node<Traits>*,
CGAL::Search_traits_2<Kernel>,
CGAL::Search_traits_3<Kernel>,
CGAL::Search_traits<FT_,Point,CartesianIterator,ConstructCartesianIterator>.

CGAL::Kd_tree_node<Traits, Splitter, UseExtendedNode>

advanced

Definition

The class *Kd_tree_node*<*Traits*, *Splitter*, *UseExtendedNode*> implements a node class for a *k-d* tree. A node is either a leaf node, an internal node or an extended internal node. A leaf node contains one or more points. An internal node contains a pointer to its lower child, a pointer to its upper child, and a pointer to its separator. An extended internal node is an internal node containing the lower and upper limit of an extended node's rectangle along the node's cutting dimension.

```
#include <CGAL/Kd_tree_node.h>
```

Parameters

Expects for the template argument a model of the concept *SearchTraits*, for example *CGAL::Search_traits_2*<*CGAL::Cartesian*<*double*> >, or *CGAL::Cartesian_d*<*double*>.

Types

```
enum Node_type { LEAF, INTERNAL, EXTENDED_INTERNAL};
```

Denotes type of node.

```
Traits::FT          FT;          Number type.
```

```
Traits::Point_d     Point_d;     Point type.
```

```
Splitter::Separator Separator;   Separator type.
```

```
Kd_tree<Traits,Splitter,UseExtendedNode>::Point_d_iterator
```

```
Point_d_iterator;    Iterator over points.
```

```
Kd_tree<Traits,Splitter,UseExtendedNode>::Node_handle
```

```
Node_handle;        Node handle.
```

Creation

Operations

```
template <class OutputIterator, class FuzzyQueryItem>
```


<i>OutputIterator</i>	<i>n.search(OutputIterator it, FuzzyQueryItem q)</i>	Reports the points from the subtree of the node, that are approximately contained by q.
<i>template <class OutputIterator></i> <i>OutputIterator</i>	<i>n.tree_points(OutputIterator it)</i>	Reports all the points contained by the subtree of the node.
<i>bool</i>	<i>n.is_leaf()</i>	Indicates whether a node is a leaf node.
<i>int</i>	<i>n.size()</i>	Returns the number of items stored in a leaf node.
<i>Point_d_iterator</i>	<i>n.begin()</i>	Returns the iterator to the first item in a leaf node.
<i>Point_d_iterator</i>	<i>n.end()</i>	Returns the past-the-end iterator in a leaf node.
<i>Node_handle</i>	<i>n.lower()</i>	Returns a handle to the lower child of an internal node.
<i>Node_handle</i>	<i>n.upper()</i>	Returns a handle to the upper child of an internal node.
<i>Separator&</i>	<i>n.separator()</i>	Returns a reference to the separator.
<i>FT</i>	<i>n.low_val()</i>	Returns the lower limit of an extended node's rectangle along the node's cutting dimension.
<i>FT</i>	<i>n.high_val()</i>	Returns the upper limit of an extended node's rectangle along the node's cutting dimension.

└────────── *advanced* ─────────┘

CGAL::Kd_tree_rectangle<Traits>

advanced

Definition

The class *Kd_tree_rectangle<Traits>* implements *d*-dimensional iso-rectangles and related operations, e.g., methods to compute bounding boxes of point sets.

```
#include <CGAL/Kd_tree_rectangle.h>
```

Types

Traits::FT *FT*; Number type.

Creation

Kd_tree_rectangle<Traits> *r*(*int d*); Constructs a *d*-dimensional rectangle *r* with lower bound and upper bound set to zero in each dimension.

```
template <class PointIter>
Kd_tree_rectangle<Traits> r( int d, PointIter begin, PointIter end);
```

Constructs the bounding box of the points in the range [*begin*,*end*), where the value type of *PointIter* must be *Traits::Point_d*.

Operations

FT *r.min_coord(int i)* Returns the lower bound of the rectangle in dimension *i*.

FT *r.max_coord(int i)* Returns the upper bound of the rectangle in dimension *i*.

void *r.set_upper_bound(int i, FT x)*
Sets upper bound in dimension *i* to *x*.

void *r.set_lower_bound(int i, FT x)*
Sets lower bound in dimension *i* to *x*.

FT *r.max_span()* Returns the maximal span of the rectangle.

FT *r.max_span_coord()* Returns the smallest coordinate for which the rectangle has its maximal span.

int *r.dimension()* Returns the dimension of the rectangle.

void *r.split(& r, int d, FT value)*

Splits rectangle in dimension *d* at coordinate-value *value* by modifying itself to lower half and by modifying *r* to upper half.

Output Routines

template<class FT>
std::ostream& *std::ostream& s << & r*

Inserts rectangle *r* in the output stream *s* and returns *s*.

└────────── *advanced* ─────────┘

CGAL::Manhattan_distance_iso_box_point<Traits>

Definition

The class *Manhattan_distance_iso_box_point*<Traits> provides an implementation of the *GeneralDistance* concept for the Manhattan distance (l_1 metric) between a d -dimensional iso-box and a d -dimensional point and the Manhattan distance between a d -dimensional iso-box and a d -dimensional iso-box defined as a k - d tree rectangle.

```
#include <CGAL/Manhattan_distance_iso_box_point.h>
```

Parameters

Expects for the template argument a model for the concept *SearchTraits*, for example *CGAL::Search_traits_3<CGAL::Cartesian<double>>*.

Is Model for the Concepts

GeneralDistance

Types

<i>Traits::FT</i>	<i>FT</i> ;	Number type.
<i>Traits::Point_d</i>	<i>Point_d</i> ;	Point type.
<i>Traits::Iso_box_d</i>	<i>Query_item</i> ;	Query item type.

Creation

```
Manhattan_distance_iso_box_point<Traits> md;
```

Default constructor.

Operations

<i>FT</i>	<i>md.transformed_distance(Query_item b, Point_d p)</i>	Returns the transformed distance between b and p .
-----------	----------------------------------------------------------	--------------------------------------------------------

<i>FT</i>	<i>md.transformed_distance(FT d)</i>	Returns the transformed value of d .
-----------	---------------------------------------	----------------------------------------

FT *md.inverse_of_transformed_distance(FT d)*

Returns the value of the inverse of the transform function applied to *d*.

FT *md.min_distance_to_rectangle(Query_item b, Kd_tree_rectangle<Traits> r)*

Returns the minimal distance between a point from *b* and a point from *r*.

FT *md.max_distance_to_rectangle(Query_item b, Kd_tree_rectangle<Traits> r)*

Returns the maximal distance between the iso-box *b* and a point from *r* furthest to *b*.

See Also

GeneralDistance.

CGAL::Median_of_max_spread<Traits, SpatialSeparator>

Definition

Implements the *median of max spread* splitting rule. The splitting dimension is the dimension of the longest side of the rectangle. The splitting value is defined by the median of the coordinates of the data points along this dimension.

```
#include <CGAL/Splitters.h>
```

Parameters

Expects for the first template argument a model of the concept *SearchTraits*, for example the type `CGAL::Search_traits_3<Cartesian<double>>`.

Expects for the second template argument a model of the concept *Separator*. It has as default value the type, `CGAL::Plane_separator<Traits::FT>`.

Is Model for the Concepts

Splitter

Creation

```
Median_of_max_spread<Traits, SpatialSeparator> s;
```

Default constructor.

```
Median_of_max_spread<Traits, SpatialSeparator> s( unsigned int bucket_size);
```

Constructor.

Operations

<i>unsigned int</i>	<i>s.bucket_size()</i>	Returns the bucket size of the leaf nodes.
---------------------	------------------------	--------------------------------------------

See Also

Splitter,
SpatialSeparator

CGAL::Median_of_rectangle<Traits, SpatialSeparator>

Definition

Implements the *median of rectangle* splitting rule. The splitting dimension is the dimension of the longest side of the rectangle. The splitting value is defined by the median of the coordinates of the data points along this dimension.

#include <CGAL/Splitters.h>

Parameters

Expects for the first template argument a model of the concept *SearchTraits*, for example the type *CGAL::Search_traits_3<Cartesian<double>>*.

Expects for the second template argument a model of the concept *Separator*. It has as default value the type, *CGAL::Plane_separator<Traits::FT>*.

Is Model for the Concepts

Splitter

Creation

Median_of_rectangle<Traits, SpatialSeparator> s;

Default constructor.

Median_of_rectangle<Traits, SpatialSeparator> s(unsigned int bucket_size);

Constructor.

Operations

<i>unsigned int</i>	<i>s.bucket_size()</i>	Returns the bucket size of the leaf nodes.
---------------------	------------------------	--------------------------------------------

See Also

Splitter,
SpatialSeparator

CGAL::Midpoint_of_max_spread<Traits, SpatialSeparator>

Definition

Implements the *midpoint of max spread* splitting rule. A rectangle is cut through $(Mind + Maxd)/2$ orthogonal to the dimension with the maximum point spread $[Mind, Maxd]$.

```
#include <CGAL/Splitters.h>
```

Parameters

Expects for the first template argument a model of the concept *SearchTraits*, for example the type `CGAL::Search_traits_3< Cartesian<double> >`.

Expects for the second template argument a model of the concept *Separator*. It has as default value the type, `CGAL::Plane_separator<Traits::FT>`

Is Model for the Concepts

Splitter

Creation

```
Midpoint_of_max_spread<Traits, SpatialSeparator> s;
```

Default constructor.

```
Midpoint_of_max_spread<Traits, SpatialSeparator> s( unsigned int bucket_size);
```

Constructor.

Operations

<i>unsigned int</i>	<i>s.bucket_size()</i>	Returns the bucket size of the leaf nodes.
---------------------	------------------------	--------------------------------------------

See Also

Splitter,
SpatialSeparator

CGAL::Midpoint_of_rectangle<Traits, SpatialSeparator>

Definition

Implements the *midpoint of rectangle* splitting rule. A rectangles is cut through its midpoint orthogonal to the longest side.

```
#include <CGAL/Splitters.h>
```

Parameters

Expects for the first template argument a model of the concept *SearchTraits*, for example the type `CGAL::Search_traits_3< Cartesian<double> >`.

Expects for the second template argument a model of the concept *Separator*. It has as default value the type, `CGAL::Plane_separator<Traits::FT>`

Is Model for the Concepts

Splitter

Creation

```
Midpoint_of_rectangle<Traits, SpatialSeparator> s;
```

Default constructor.

```
Midpoint_of_rectangle<Traits, SpatialSeparator> s( unsigned int bucket_size);
```

Constructor.

Operations

<i>unsigned int</i>	<i>s.bucket_size()</i>	Returns the bucket size of the leaf nodes.
---------------------	------------------------	--------------------------------------------

See Also

Splitter,
SpatialSeparator

OrthogonalDistance

Definition

Requirements of an orthogonal distance class supporting incremental distance updates. To optimize distance computations transformed distances are used. E.g., for an Euclidean distance the transformed distance is the squared Euclidean distance.

Refines

GeneralDistance

Has Models

CGAL::Euclidean_distance<Traits>,
CGAL::Weighted_Minkowski_distance<Traits>

Types

<i>OrthogonalDistance:: FT</i>	Number type.
<i>OrthogonalDistance:: Point_d</i>	Point type.
<i>OrthogonalDistance:: Query_item</i>	Query item type.

Creation

<i>OrthogonalDistance od(int d);</i>	Constructor implementing distance for d -dimensional points.
---------------------------------------	----------------------------------------------------------------

Operations

<i>FT</i>	<i>od.transformed_distance(Query_item q, Point_d r)</i>	Returns the transformed distance between q and r .
<i>FT</i>	<i>od.min_distance_to_rectangle(Query_item q, Kd_tree_rectangle<Traits> r)</i>	Returns the transformed distance between q and the point on the boundary of r closest to q .
<i>FT</i>	<i>od.max_distance_to_rectangle(Query_item q, Kd_tree_rectangle<Traits> r)</i>	Returns the transformed distance between q and the point on the boundary of r farthest to q .

<i>FT</i>	<i>od.transformed_distance(FT d)</i>	Returns the transformed distance.
<i>FT</i>	<i>od.inverse_of_transformed_distance(FT d)</i>	Returns the inverse of the transformed distance.
<i>FT</i>	<i>od.new_distance(FT dist, FT old_off, FT new_off, int cutting_dimension)</i>	Updates <i>dist</i> incrementally and returns the updated distance.

CGAL::Orthogonal_incremental_neighbor_search<Traits, OrthogonalDistance, Splitter, SpatialTree>

Definition

The class *Orthogonal_incremental_neighbor_search*<Traits, *OrthogonalDistance*, *Splitter*, *SpatialTree*> implements incremental nearest and furthest neighbor searching on a tree.

```
#include <CGAL/Orthogonal_incremental_neighbor_search.h>
```

Parameters

Expects for the first template argument a model of the concept *SearchTraits*, for example *CGAL::Search_traits_2*<*CGAL::Cartesian*<*double*>>.

Expects for the second template argument a model of the concept *GeneralDistance*. The default type is *CGAL::Euclidean_distance*<Traits>.

Expects for third template argument a model of the concept *Splitter*. The default type is *CGAL::Sliding_midpoint*<Traits>.

Expects for fourth template argument a model of the concept *SpatialTree*. The default type is *CGAL::Kd_tree*<Traits, *Splitter*, *CGAL::Tag_true*>. The template argument must be *CGAL::Tag_true* because orthogonal search needs extended kd tree nodes.

Types

Traits::Point_d *Point_d*; Point type.

Traits::FT *FT*; Number type.

OrthogonalDistance::Query_item
 Query_item; Query item.

std::pair<*Point_d*,*FT*>
 Point_with_transformed_distance;
 Pair of point and transformed distance.

Orthogonal_incremental_neighbor_search<Traits, *OrthogonalDistance*, *Splitter*, *SpatialTree*>:: *iterator*
 Input iterator with value type *Point_with_transformed_distance* for enumerating approximate neighbors.

SpatialTree *Tree*; The tree type.

Creation

```
Orthogonal_incremental_neighbor_search<Traits, OrthogonalDistance, Splitter, SpatialTree> s(  
    SpatialTree& tree,  
    Query_item query,  
    FT eps=FT(0.0),  
    bool search_nearest=true,  
    OrthogonalDistance d=OrthogonalDistance())
```

Constructor for incremental neighbor searching of the query item *query* in the points stored *tree* using a distance *d* and approximation factor *eps*.

Operations

<i>iterator</i>	<i>s.begin()</i>	Returns an iterator to the approximate neighbors.
<i>iterator</i>	<i>s.end()</i>	Returns the corresponding past-the-end iterator.

_____ *advanced* _____
|

```
std::ostream& s.statistics( std::ostream& s)
```

Inserts statistics of the search process into the output stream *s*.

_____ *advanced* _____
|

See Also

CGAL::Incremental_neighbor_search<Traits, GeneralDistance, SpatialTree>.

CGAL::Orthogonal_k_neighbor_search<Traits, OrthogonalDistance, Splitter, SpatialTree>

Definition

The class *Orthogonal_k_neighbor_search*<Traits, *OrthogonalDistance*, *Splitter*, *SpatialTree*> implements approximate *k*-nearest and *k*-furthest neighbor searching on a tree using an orthogonal distance class.

```
#include <CGAL/Orthogonal_k_neighbor_search.h>
```

Parameters

Expects for the first template argument an implementation of the concept *SearchTraits*, for example *CGAL::Search_traits_2*<*CGAL::Cartesian*<*double*>>.

Expects for the second template argument a model of the concept *GeneralDistance*. The default type is *CGAL::Euclidean_distance*<Traits>.

Expects for third template argument a model of the concept *Splitter*. The default type is *CGAL::Sliding_midpoint*<Traits>.

Expects for fourth template argument an implementation of the concept *SpatialTree*. The default type is *CGAL::Kd_tree*<Traits, *Splitter*, *CGAL::Tag_true*>. The template argument must be *CGAL::Tag_true* because orthogonal search needs extended kd tree nodes.

Types

Traits::Point_d *Point_d*; Point type.

Traits::FT *FT*; Number type.

GeneralDistance::Query_item
 Query_item; Query item.

std::pair<*Point_d*,*FT*>
 Point_with_transformed_distance;
 Pair of point and transformed distance.

Orthogonal_k_neighbor_search<Traits, *OrthogonalDistance*, *Splitter*, *SpatialTree*>:: *iterator*
 Bidirectional iterator with value type *Point_with_transformed_distance* for enumerating approximate neighbors.

SpatialTree *Tree*; The tree type.

Operations

```
Orthogonal_k_neighbor_search<Traits, OrthogonalDistance, Splitter, SpatialTree> s(
    SpatialTree tree,
    Query_item query,
    int k=1,
    FT eps=FT(0.0),
    bool search_nearest=true,
    OrthogonalDistance d=OrthogonalDistance())
```

Constructor for searching approximately k neighbors of the query item *query* in the points stored in *tree* using distance d and approximation factor eps .

iterator *s.begin()* Returns an iterator to the approximate neighbors.

iterator *s.end()* Past-the-end iterator.

_____ *advanced* _____
|

std::ostream& *s.statistics(std::ostream& s)*

Inserts statistics of the search process into the output stream *s*.

_____ *advanced* _____

See Also

CGAL::K_neighbor_search<Traits, GeneralDistance, Splitter, SpatialTree>.

CGAL::Plane_separator<FT>

— advanced —

Definition

The class *Plane_separator*<FT> implements a plane separator, i.e., a hyperplane that is used to separate two half spaces. This hyperplane is defined by a cutting dimension d and a cutting value v as $x_d = v$, where v denotes the d^{th} coordinate value.

```
#include <CGAL/Plane_separator.h>
```

Is Model for the Concepts

SpatialSeparator

Creation

```
Plane_separator<FT> s( int d, FT v);
```

Constructs a separator that separates two half spaces by a hyperplane defined by $x_d = v$, where v denotes the d^{th} coordinate value.

```
Plane_separator<FT> s( p);
```

Copy constructor.

Operations

```
void s.set_cutting_dimension( int d)
```

Sets the cutting dimension to d .

```
void s.set_cutting_value( FT v)
```

Sets the cutting value to v .

```
int s.cutting_dimension()
```

Returns the number of the cutting dimension.

```
FT s.cutting_value()
```

Returns the cutting value.

```
template <class SpatialPoint>
```

```
bool s.has_on_negative_side( SpatialPoint p)
```

Returns true if and only if the coordinate of p in the cutting dimension is smaller than the cutting value.

```
Plane_separator<FT>
```

```
s = s2
```

Assignment operator.

Output Operators

```
template<class FT>  
std::ostream&      std::ostream& os << s
```

Inserts the plane separator *s* in the output stream *os* and returns *os*.

└────────── *advanced* ─────────┘

SearchTraits

Definition

The concept *SearchTraits* defines the requirements for the template parameter of the search classes.

Types

<i>SearchTraits:: Point_d</i>	Point type. <i>CGAL::Kernel_traits</i> has to be specialized for this type.
<i>SearchTraits:: Iso_box_d</i>	Iso box type. It is only needed for range search queries.
<i>SearchTraits:: Sphere_d</i>	Sphere type. It is only needed for range search queries.
<i>SearchTraits:: FT</i>	The number type of the Cartesian coordinates of types <i>Point_d</i>
<i>SearchTraits:: Cartesian_const_iterator</i>	An random access iterator type to enumerate the Cartesian coordinates of a point.
<i>SearchTraits:: Construct_cartesian_const_iterator</i>	Functor with operators to construct iterators on the first and the past-the-end iterator for the Cartesian coordinates of a point.
<i>SearchTraits:: Construct_center_d;</i>	Functor with operator to construct the center of an object of type <i>Sphere_d</i> .
<i>SearchTraits:: Construct_squared_radius_d;</i>	Functor with operator to compute the squared radius of a an object of type <i>Sphere_d</i> .
<i>SearchTraits:: Construct_min_vertex_d;</i>	Functor with operator to construct the vertex with lexicographically smallest coordinates of an object of type <i>Iso_box_d</i> .
<i>SearchTraits:: Construct_max_vertex_d;</i>	Functor with operator to construct the vertex with lexicographically largest coordinates of an object of type <i>Iso_box_d</i> .

Has Models

```
CGAL::Cartesian_d<FT>
CGAL::Homogeneous_d<RT>
CGAL::Search_traits_2<Kernel>
CGAL::Search_traits_3<Kernel>,
CGAL::Search_traits<NT,Point, CartesianCoordinateIterator,ConstructCartesianCoordinateIterator,ConstructMinVertex,ConstructMaxVertex>
```

CGAL::Search_traits_2<Kernel>

Definition

The class *Search_traits_2<Kernel>* can be used as a template parameter of the kd tree and the search classes.

```
#include <CGAL/Search_traits_2.h>
```

Parameters

Expects for the template argument a model of the concept *Kernel*, for example *CGAL::Cartesian<double>* or *CGAL::Simple_cartesian<CGAL::Gmp_q>*.

Is Model for the Concepts

SearchTraits.

Types

Kernel::FT *FT*; Number type.

Kernel::Point_2 *Point_d*; Point type.

Kernel::Iso_rectangle_2
 Iso_box_d; Iso box type.

Kernel::Sphere_2 *Sphere_d*; Sphere type.

Kernel::Cartesian_const_iterator_2
 Cartesian_const_iterator_d;
 An iterator over the Cartesian coordinates.

Kernel::Construct_cartesian_const_iterator_2
 Construct_cartesian_const_iterator_d;
 A functor with two function operators, which return the begin and past the end iterator for the Cartesian coordinates. The functor for begin has as argument a *Point_d*. The functor for the past the end iterator, has as argument a *Point_d* and an *int*.

Kernel::Construct_iso_rectangle_2

Construct_iso_box_d; Functor with operator to construct the iso box from two points.

Kernel::Construct_center_2

Construct_center_d; Functor with operator to construct the center of an object of type *Sphere_d*.

Kernel::Compute_squared_radius_2

Construct_squared_radius_d;

Functor with operator to compute the squared radius of a an object of type *Sphere_d*.

Kernel::Construct_min_vertex_2

Construct_min_vertex_d;

Functor with operator to construct the vertex with lexicographically smallest coordinates of an object of type *Iso_box_d*.

Kernel::Construct_max_vertex_2

Construct_max_vertex_d;

Functor with operator to construct the vertex with lexicographically largest coordinates of an object of type *Iso_box_d*.

See Also

Search_traits_3<Kernel>

Search_traits<NT_,Point, CartesianConstIterator, ConstructCartesianConstIterator

CGAL::Search_traits_3<Kernel>

Definition

The class *Search_traits_3<Kernel>* can be used as a template parameter of the kd tree and the search classes. *Kernel* must be a CGAL kernel.

```
#include <CGAL/Search_traits_3.h>
```

Parameters

Expects for the template argument a model of the concept *Kernel*, for example *CGAL::Cartesian<double>* or *CGAL::Simple_cartesian<CGAL::Gmp_q>*.

Is Model for the Concepts

SearchTraits.

Types

Kernel::FT *FT*; Number type.

Kernel::Point_3 *Point_d*; Point type.

Kernel::Iso_cuboid_3
 Iso_box_d; Iso box type.

Kernel::Sphere_3 *Sphere_d*; Sphere type.

Kernel::Cartesian_const_iterator_3
 Cartesian_const_iterator_d;
 An iterator over the Cartesian coordinates.

Kernel::Construct_cartesian_const_iterator_3
 Construct_cartesian_const_iterator_d;
 A functor with two function operators, which return the begin and past the end iterator for the Cartesian coordinates. The functor for begin has as argument a *Point_d*. The functor for the past the end iterator, has as argument a *Point_d* and an *int*.

Kernel::Construct_center_3

Construct_center_d; Functor with operator to construct the center of an object of type *Sphere_d*.

Kernel::Compute_squared_radius_3

Construct_squared_radius_d;

Functor with operator to compute the squared radius of a an object of type *Sphere_d*.

Kernel::Construct_min_vertex_3

Construct_min_vertex_d;

Functor with operator to construct the vertex with lexicographically smallest coordinates of an object of type *Iso_box_d*.

Kernel::Construct_max_vertex_3

Construct_max_vertex_d;

Functor with operator to construct the vertex with lexicographically largest coordinates of an object of type *Iso_box_d*.

See Also

Search_traits_2<Kernel>

Search_traits<Point, CartesianConstIterator, ConstructCartesianConstIterator

CGAL::Search_traits_d<Kernel>

Definition

The class *Search_traits_d<Kernel>* can be used as a template parameter of the kd tree and the search classes. *Kernel* must be a CGAL kernel.

Kernel must be a d-dimensional CGAL kernel.

```
#include <CGAL/Search_traits_d.h>
```

Parameters

Expects for the template argument a model of the concept *Kernel_d*, for example *CGAL::Cartesian_d<double>* or *CGAL::Homogeneous_d<CGAL::Gmp_z>*.

Is Model for the Concepts

SearchTraits.

Types

Kernel::FT *NT*; Number type.

Kernel::Point_d *Point_d*; Point type.

Kernel::Iso_cuboid_d
 Iso_box_d; Iso box type.

Kernel::Sphere_d *Sphere_d*; Sphere type.

Kernel::Cartesian_const_iterator_d
 Cartesian_const_iterator;
 An iterator over the Cartesian coordinates.

Kernel::Construct_cartesian_const_iterator_d
 Construct_cartesian_const_iterator;
 A functor with two function operators, which return the begin and past the end iterator for the Cartesian coordinates. The functor for begin has as argument a *Point_d*. The functor for the past the end iterator, has as argument a *Point_d* and an *int*.

Kernel::Construct_min_vertex_d

Construct_min_vertex_d;

Functor with operator to construct the vertex with lexicographically smallest coordinates of an object of type *Iso_box_d*.

Kernel::Construct_max_vertex_d

Construct_max_vertex_d;

Functor with operator to construct the vertex with lexicographically largest coordinates of an object of type *Iso_box_d*.

See Also

Search_traits_2<Kernel>

Search_traits_3<Kernel>

Search_traits<Point, CartesianConstIterator, ConstructCartesianConstIterator>

CGAL::Search_traits<NT,Point,CartesianIterator,ConstructCartesianIterator,ConstructMinVertex,ConstructMaxVertex>

Definition

The class *Search_traits*<*NT*,*Point*,*CartesianIterator*,*ConstructCartesianIterator*,*ConstructMinVertex*,*ConstructMaxVertex*> can be used as a template parameter of the kd tree and the search classes. It is a mere wrapper for the geometric types needed by these classes.

```
#include <CGAL/Search_traits.h>
```

Is Model for the Concepts

SearchTraits.

Types

NT *FT*; The number type of the coordinates.

Point *Point_d*; Point type.

CartesianIterator *Cartesian_const_iterator_d*;
An iterator over the coordinates.

ConstructCartesianIterator

Construct_Cartesian_const_iterator_d;

A functor with two function operators, which return the begin and past the end iterator for the Cartesian coordinates. The functor for begin has as argument a *Point_d*. The functor for the past the end iterator, has as argument a *Point_d* and an *int*.

ConstructMinVertex *Construct_min_vertex_d*;

Functor with operator to construct the vertex with lexicographically smallest coordinates of an object of type *Iso_box_d*.

Kernel::ConstructMaxVertex

Construct_max_vertex_d;

Functor with operator to construct the vertex with lexicographically largest coordinates of an object of type *Iso_box_d*.

See Also

Search_traits_2<Kernel>

Search_traits_3<Kernel>

Search_traits_d<Kernel>

CGAL::Sliding_fair<Traits, SpatialSeparator>

Definition

Implements the *sliding fair* splitting rule. This splitting rule is a compromise between the *Fair* splitting rule and the *Sliding_midpoint* rule. Sliding fair-split is based on the theory that there are two types of splits that are good: balanced splits that produce fat rectangles, and unbalanced splits provided the rectangle with fewer points is fat.

Also, this splitting rule maintains an upper bound on the maximal allowed ratio of the longest and shortest side of a rectangle (the value of this upper bound is set in the constructor of the fair splitting rule). Among the splits that satisfy this bound, it selects the one one in which the points have the largest spread. It then considers the most extreme cuts that would be allowed by the aspect ratio bound. This is done by dividing the longest side of the rectangle by the aspect ratio bound. If the median cut lies between these extreme cuts, then we use the median cut. If not, then consider the extreme cut that is closer to the median. If all the points lie to one side of this cut, then we slide the cut until it hits the first point. This may violate the aspect ratio bound, but will never generate empty cells.

```
#include <CGAL/Splitters.h>
```

Parameters

Expects for the first template argument a model of the concept *SearchTraits*, for example *CGAL::Cartesian_d<double>*.

Expects for the second template argument a model of the concept *Separator*. It has as default value the type, *CGAL::Plane_separator<Traits::FT>*

Is Model for the Concepts

Splitter

Types

Traits::FT *FT*; Number type.

Creation

```
Sliding_fair<Traits, SpatialSeparator> s( unsigned int bucket_size, FT aspect_ratio=FT(3));
```

Constructor.

Operations

FT *s.aspect_ratio()* Returns the maximal ratio between the largest and smallest side of a cell allowed for fair splitting.

unsigned int *s.bucket_size()* Returns the bucket size of the leaf nodes.

See Also

Splitter,
SpatialSeparator

CGAL::Sliding_midpoint<Traits, SpatialSeparator>

Definition

Implements the *sliding midpoint* splitting rule. This is a modification of the *Midpoint_of_rectangle* splitting rule. It first attempts to perform a midpoint of rectangle split as described above. If data points lie on both sides of the separating plane the sliding midpoint rule computes the same separator as the midpoint of rectangle rule. If the data points lie only on one side it avoids this by sliding the separator, computed by the midpoint of rectangle rule, to the nearest datapoint.

```
#include <CGAL/Splitters.h>
```

Parameters

Expects for the first template argument a model of the concept *SearchTraits*, for example *CGAL::Cartesian_d<double>*.

Expects for the second template argument a model of the concept *Separator*. It has as default value the type, *CGAL::Plane_separator<Traits::FT>*.

Is Model for the Concepts

Splitter

Creation

```
Sliding_midpoint<Traits, SpatialSeparator> s( unsigned int bucket_size);
```

Constructor.

Operations

<i>unsigned int</i>	<i>s.bucket_size()</i>	Returns the bucket size of the leaf nodes.
---------------------	------------------------	--------------------------------------------

See Also

Splitter,
SpatialSeparator

SpatialSeparator

advanced

Definition

The concept `SpatialSeparator` defines the requirements for a separator. A separator is a $(d-1)$ -dimensional subspace that separates a d -dimensional space into two parts. One part of space is said to be on the negative side of the separator and the other part of space is said to be on the positive side of the separator.

Has Models

`CGAL::Plane_separator<FT>`.

Types

`SpatialSeparator::FT`; Number type.

Creation

`SpatialSeparator s`; Default constructor.

Operations

`void` `s.set_cutting_dimension(int d)` Sets the cutting dimension to d .

`void` `s.set_cutting_value(FT v)` Sets the cutting value to v .

`int` `s.cutting_dimension()` Returns the number of the cutting dimension.

`FT` `s.cutting_value()` Returns the cutting value.

`template <class Point_d>`
`bool` `s.has_on_negative_side(Point_d p)` Returns true if and only if the point p is on the negative side of the separator.

advanced

SpatialTree

Definition

The concept `SpatialTree` defines the requirements for a tree supporting both neighbor searching and approximate range searching.

Types

<i>SpatialTree:: SearchTraits</i>	Search traits.
<i>SpatialTree:: Point_d</i>	Point type.
<i>SpatialTree:: iterator;</i>	Bidirectional iterator with value type <i>Point_d</i> that allows to enumerate all points in the tree.
<i>SpatialTree:: Node_handle;</i>	Node handle.
<i>SpatialTree:: Point_d_iterator;</i>	Iterator with value type <i>Point_d*</i> .
<i>SpatialTree:: Splitter</i>	Splitter.
<i>SpatialTree:: Distance</i>	Distance.

Creation

```
template <class InputIterator>
SpatialTree tree( InputIterator first, InputIterator beyond, SearchTraits t);
```

Constructs a tree on the elements from the sequence *[first,beyond)*.

Operations

```
template <class OutputIterator, class FuzzyQueryItem>
OutputIterator tree.search( OutputIterator it, FuzzyQueryItem q)
```

Reports the points that are approximately contained by *q*. The value type of *OutputIterator* must be *Point_d*.

<i>iterator</i>	<i>tree.begin()</i>	Returns an iterator to the first point in the tree.
<i>iterator</i>	<i>tree.end()</i>	Returns the corresponding past-the-end iterator.

<i>Node_handle</i>	<i>tree.root()</i>	Returns a handle to the root node of the tree.
<i>Kd_tree_rectangle<SearchTraits></i>		
	<i>tree.bounding_box()</i>	returns a const reference to the bounding box of the root node of the tree.
<i>unsigned int</i>	<i>tree.size()</i>	Returns the number of items that are stored in the tree.

Has Models

CGAL::Kd_tree<Traits,Splitter,UseExtendedNode>.

CGAL::Weighted_Minkowski_distance<Traits>

Definition

The class *Weighted_Minkowski_distance*<Traits> provides an implementation of the concept *OrthogonalDistance*, with a weighted Minkowski metric on d -dimensional points defined by $l_p(w)(r, q) = (\sum_{i=1}^d w_i(r_i - q_i)^p)^{1/p}$ for $0 < p < \infty$ and defined by $l_\infty(w)(r, q) = \max\{w_i|r_i - q_i| \mid 1 \leq i \leq d\}$. For the purpose of the distance computations it is more efficient to compute the transformed distance $\sigma_{i=1}^d w_i(r_i - q_i)^p$ instead of the actual distance.

```
#include <CGAL/Weighted_Minkowski_distance.h>
```

Parameters

Expects for the template argument a model of the concept *SearchTraits*, for example *CGAL::Search_traits_2*<Kernel>.

Is Model for the Concepts

OrthogonalDistance

Types

Traits::FT *FT*; Number type.

Traits::Point_d *Point_d*; Point type.

Creation

```
Weighted_Minkowski_distance<Traits> wd( int d);
```

Constructor implementing l_2 metric for d -dimensional points.

```
template <class InputIterator>
```

```
Weighted_Minkowski_distance<Traits> wd( FT power, int dim, InputIterator wb, InputIterator we);
```

Constructor implementing the $l_{power}(weights)$ metric. $power \leq 0$ denotes the $l_\infty(weights)$ metric. The values in the iterator range $[wb, we)$ are the weight.

Operations

```
FT                                      wd.transformed_distance( Point_d q, Point_d r)
```

Returns d^{power} , where d denotes the distance between q and r .

<i>FT</i>	<i>wd.min_distance_to_rectangle(Point_d q, Kd_tree_rectangle<Traits> r;)</i>	Returns d^{power} , where d denotes the distance between the query item q and the point on the boundary of r closest to q .
<i>FT</i>	<i>wd.max_distance_to_rectangle(Point_d q, Kd_tree_rectangle<Traits> r;)</i>	Returns d^{power} , where d denotes the distance between the query item q and the point on the boundary of r farthest to q .
<i>FT</i>	<i>wd.new_distance(FT dist, FT old_off, FT new_off, int cutting_dimension)</i>	Updates <i>dist</i> incrementally and returns the updated distance.
<i>FT</i>	<i>wd.transformed_distance(FT d)</i>	Returns d^p for $0 < p < \infty$. Returns d for $p = \infty$.
<i>FT</i>	<i>wd.inverse_of_transformed_distance(FT d)</i>	Returns $d^{1/p}$ for $0 < p < \infty$. Returns d for $p = \infty$.

See Also

OrthogonalDistance
CGAL::Euclidean_distance<Traits>

Chapter 36

dD Range and Segment Trees

Gabriele Neyer

36.1 Introduction

This chapter presents the CGAL range tree and segment tree data structures.

36.2 Definitions

This section presents d -dimensional range and segment trees. A one-dimensional range tree is a binary search tree on *one-dimensional point data*. Here we call all one-dimensional data types having a strict ordering (like integer and double) *point data*. d -dimensional *point data* are d -tuples of one-dimensional point data.

A one-dimensional segment tree is a binary search tree as well, but with *one-dimensional interval data* as input data. One-dimensional interval data is a pair (i.e., 2-tuple) (a, b) , where a and b are one-dimensional point data of the same type and $a < b$. The pair (a, b) represents a half open interval $[a, b)$. Analogously, a d -dimensional interval is represented by a d -tuple of one-dimensional intervals.

The *input data type* for a d -dimensional tree is a container class consisting of a d -dimensional point data type, interval data type or a mixture of both, and optionally a *value type*, which can be used to store arbitrary data. E.g., the d -dimensional bounding box of a d -dimensional polygon may define the interval data of a d -dimensional segment tree and the polygon itself can be stored as its value. An *input data item* is an instance of an input data type.

The range and segment tree classes are fully generic in the sense that they can be used to define *multilayer trees*. A multilayer tree of dimension (number of layers) d is a simple tree in the d -th layer, whereas the k -th layer, $1 \leq k \leq d - 1$, of the tree defines a tree where each (inner) vertex contains a multilayer tree of dimension $d - k + 1$. The $k - 1$ -dimensional tree which is nested in the k -dimensional tree (T) is called the *sublayer tree* (of T). For example, a d -dim tree can be a range tree on the first layer, constructed with respect to the first dimension of d -dimensional data items. On all the data items in each subtree, a $(d - 1)$ -dimensional tree is built, either a range or a segment tree, with respect to the second dimension of the data items. And so on. Figures 36.2, 36.3 and 36.5 illustrate the meaning of a sublayer tree graphically.

After creation of the tree, further insertions or deletions of data items are disallowed. The tree class does neither depend on the type of data nor on the concrete physical representation of the data items. E.g., let a multilayer

tree be a segment tree for which each vertex defines a range tree. We can choose the data items to consist of intervals of type *double* and the point data of type *integer*. As value type we can choose *string*.

For this generality we have to define what the tree of each dimension looks like and how the input data is organized. For dimension k , $1 \leq k \leq 4$, CGAL provides ready-to-use range and segment trees that can store k -dimensional keys (intervals resp.). Examples illustrating the use of these classes are given in Sections 36.5.1 and 36.6.1. The description of the functionality of these classes as well as the definition of higher dimensional trees and mixed multilayer trees is given in the reference manual.

In the following two sections we give short definitions of the version of the range tree and segment tree implemented here together with some examples. The presentation closely follows [dBvKOS97].

36.3 Software Design

In order to be able to define a multilayer tree we first designed the range and segment tree to have a template argument defining the type of the sublayer tree. With this sublayer tree type information the sublayers could be created. This approach lead to nested template arguments, since the sublayer tree can again have a template argument defining the sublayer. Therefore, the internal class and function identifiers got longer than a compiler-dependent limit. This happend already for $d = 2$.

Therefore, we chose another, object oriented, design. We defined a pure virtual base class called *Tree_base* from which we derived the classes *Range_tree_d* and *Segment_tree_d*. The constructor of these classes expects an argument called *sublayer_prototype* of type *Tree_base*. Since class *Range_tree_d* and class *Segment_tree_d* are derived from class *Tree_base*, one can use an instantiation of class *Range_tree_d* or class *Segment_tree_d* as constructor argument. This argument defines the sublayer tree of the tree. E.g., you can construct a *Range_tree_d* with an instantiation of class *Segment_tree_d* as constructor argument. You then have defined a range tree with a segment tree as sublayer tree. Since both classes *Range_tree_d* and *Segment_tree_d* expect a sublayer tree in their constructor we had to derive a third class called *Tree_anchor* from class *Tree_base* which does not expect a constructor argument. An instantiation of this class is used as constructor argument of class *Range_tree_d* or *Segment_tree_d* in order to stop the recursion.

All classes provide a *clone()* function which returns an instance (a copy) of the same tree type. The *clone()* function of the *sublayer_prototype* is called in the construction of the tree. In case that the sublayer tree again has a sublayer, it also has a *sublayer_prototype* which is also cloned and so on. Thus, a call to the *clone()* function generates a sublayer tree which has the complete knowledge about its sublayer tree.

The trees allow to perform window queries, enclosing queries, and inverse range queries on the keys. Clearly, an inverse range query makes only sense in the segment tree. In order to perform an inverse range query, a range query of ϵ width has to be performed. We preferred not to offer an extra function for this sort of query, since the inverse range query is a special case of the range query. Furthermore, offering an inverse range query in the segment tree class implies offering this function also in the range tree class and having an extra item in the traits class that accesses the inverse range query point.

The trees are templatized with three arguments: *Data*, *Window* and *Traits*. Type *Data* defines the input data type and type *Window* defines the query window type. The tree uses a well defined set of functions in order to access data. These functions have to be provided by class *Traits*.

The design partly follows the *prototype design pattern* in [GHJV95]. In comparison to our first approach using templates we want to note the following: In this approach the sublayer type is defined in use of object oriented programming at run time, while in the approach using templates, the sublayer type is defined at compile time.

The runtime overhead caused in use of virtual member functions in this object oriented design is negligible since all virtual functions are non trivial. The design concept is illustrated in Figure 36.1.

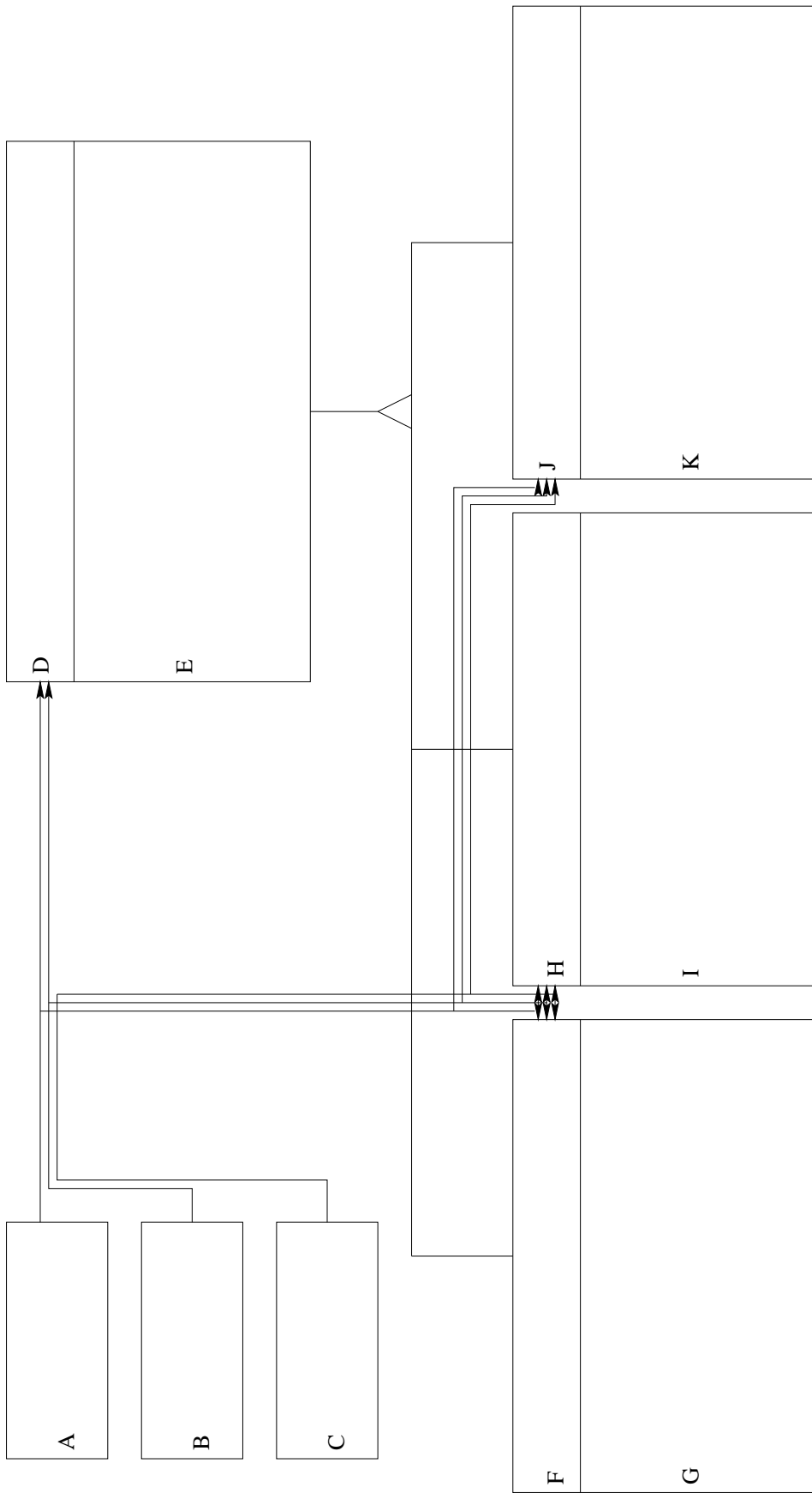


Figure 36.1: Design of the range and segment tree data structure. The symbol triangle means that the lower class is derived from the upper class.

E.g. in order to define a two dimensional multilayer tree, which consists of a range tree in the first dimension and a segment tree in the second dimension we proceed as follows: We construct an object of type *Tree_anchor* which stops the recursion. Then we construct an object of type *Segment_tree_d*, which gets as prototype argument our object of type *Tree_anchor*. After that, we define an object of type *Range_tree_d* which is constructed with the object of type *Segment_tree_d* as prototype argument. The following piece of code illustrates the construction of the two-dimensional multilayer tree.

```
int main(){
    Tree_Anchor *anchor=new Tree_Anchor;
    Segment_Tree_d *segment_tree = new Segment_Tree_d(*anchor);
    Range_Tree_d *range_segment_tree = new Range_Tree_d(*segment_tree);
    /* let data_items be a list of Data items */
    range_segment_tree->make_tree(data_items.begin(),data_items.end());
}
```

Here, class *Tree_Anchor*, *Segment_Tree_d*, and *Range_Tree_d* are defined by *typedefs*:

```
typedef Tree_anchor<Data,Window> Tree_Anchor;
typedef Segment_tree_d<Data,Window,Interval_traits> Segment_Tree_d;
typedef Range_tree_d<Data,Window,Point_traits> Range_Tree_d;
```

Class *Tree_base* and class *Tree_anchor* get two template arguments: a class *Data* which defines the type of data that is stored in the tree, and a class *Window* which defines the type of a query range. The derived classes *Range_tree_d* and *Segment_tree_d* additionally get an argument called *Tree_traits* which defines the interface between the *Data* and the tree. Let the *Data* type be a *d*-dimensional tuple, which is either a point data or an interval data in each dimension. Then, the class *Tree_traits* provides accessors to the point (resp. interval) data of that tree layer and a compare function. Remind our example of the two-dimensional tree which is a range tree in the first dimension and a segment tree in the second dimension. Then, the *Tree_traits* class template argument of class *Segment_tree_d* defines an accessor to the interval data of the *Data*, and the *Tree_traits* class template argument of class *Range_tree_d* defines an accessor to the point data of *Data*. An example implementation for these classes is listed below.

```
struct Data{
    int min,max; /* interval data */
    double point; /* point data */
};

struct Window{
    int min,max;
    double min_point, max_point;
};

class Point_traits{
public:
    typedef double Key;
    Key get_key(Data& d){return d.point;} /*key accessor */
    Key get_left(Window& w){return w.min_point;}
    Key get_right(Window& w){return w.max_point;}
    bool comp(Key& key1, Key& key2){return (key1 < key2);}
}
```



```

class Interval_traits{
public:
    typedef int Key;
    Key get_left(Data& d){return d.min;}
    Key get_right(Data& d){return d.max;}
    Key get_left_win(Window& w){return w.min;}
    Key get_right_win(Window& w){return w.max;}
    bool comp(Key& key1, Key& key2){return (key1 < key2);}
}

```

36.4 Creating an Arbitrary Multilayer Tree

Now let us have a closer look on how a multilayer tree is built. In case of creating a d -dimensional tree, we handle a sequence of arbitrary data items, where each item defines a d -dimensional interval, point or other object. The tree is constructed with an iterator over this structure. In the i -th layer, the tree is built with respect to the data slot that defines the i -th dimension. Therefore, we need to define which data slot corresponds to which dimension. In addition we want our tree to work with arbitrary data items. This requires an adaptor between the algorithm and the data item. This is resolved by the use of traits classes, implemented in form of a traits class using function objects. These classes provide access functions to a specified data slot of a data item. A d -dimensional tree is then defined separately for each layer by defining a traits class for each layer.

36.5 Range Trees

A one-dimensional range tree is a binary search tree on one-dimensional point data. The point data of the tree is stored in the leaves. Each inner vertex stores the highest entry of its left subtree. The version of a range tree implemented here is static, which means that after construction of the tree, no elements be inserted or deleted. A d -dimensional range tree is a binary leaf search tree according to the first dimension of the d -dimensional point data, where each vertex contains a $(d - 1)$ -dimensional search tree of the points in the subtree (sublayer tree) with respect to the second dimension. See [dBvKOS97] and [Sam90] for more detailed information.

A d -dimensional range tree can be used to determine all d -dimensional points that lie inside a given d -dimensional interval (*window_query*). Figure 36.2 shows a two-dimensional range tree, Figure 36.3 a d -dimensional one.

The tree can be built in $O(n \log^{d-1} n)$ time and needs $O(n \log^{d-1} n)$ space. The d -dimensional points that lie in the d -dimensional query interval can be reported in $O(\log^d n + k)$ time, where n is the total number of points and k is the number of reported points.

36.5.1 Example of Range Tree on Map-like Data

The following example program uses the predefined *Range_tree_2* data structure together with the predefined traits class *Range_tree_map_traits_2* which has two template arguments specifying the type of the point data in each dimension (*CGAL::Cartesian<double>*) and the value type of the 2-dimensional point data (*char*). Therefore the *Range_tree_2* is defined on 2-dimensional point data each of which is associated with a character. Then, a few data items are created and put into a list. After that the tree is constructed according to that list, a window query is performed, and the query elements are given out.

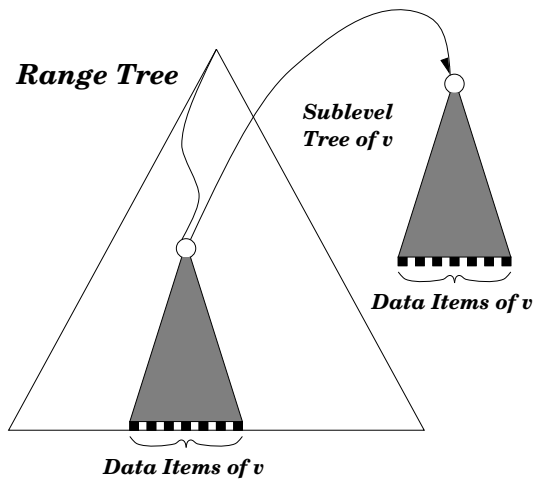


Figure 36.2: A two-dimensional range tree. The tree is a binary search tree on the first dimension. Each sublayer tree of a vertex v is a binary search tree on the second dimension. The data items in a sublayer tree of v are all data items of the subtree of v .

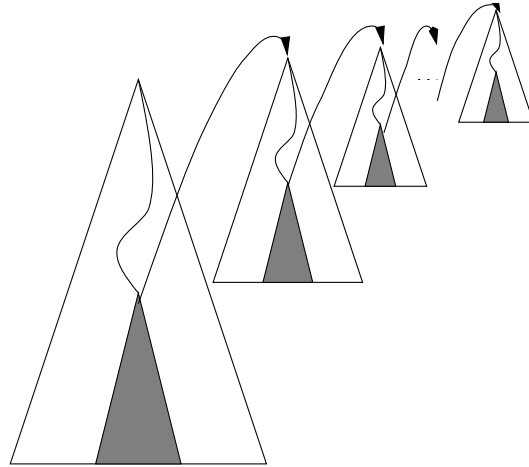


Figure 36.3: A d -dimensional range tree. For each layer of the tree, one sublayer tree is illustrated.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Range_segment_tree_traits.h>
#include <CGAL/Range_tree_k.h>

typedef CGAL::Cartesian<double> K;
typedef CGAL::Range_tree_map_traits_2<K, char> Traits;
typedef CGAL::Range_tree_2<Traits> Range_tree_2_type;

int main()
{
    typedef Traits::Key Key;
    typedef Traits::Interval Interval;

    std::vector<Key> InputList, OutputList;
    InputList.push_back(Key(K::Point_2(8,5.1), 'a'));
    InputList.push_back(Key(K::Point_2(1,1.1), 'b'));
    InputList.push_back(Key(K::Point_2(3,2.1), 'c'));

    Range_tree_2_type Range_tree_2(InputList.begin(),InputList.end());
    Interval win(Interval(K::Point_2(4,8.1),K::Point_2(5,8.2)));
    std::cout << "\n Window Query:\n ";
    Range_tree_2.window_query(win, std::back_inserter(OutputList));
    std::vector<Key>::iterator current=OutputList.begin();
    while(current!=OutputList.end()){
        std::cout << (*current).first.x() << "," << (*current).first.y()
            << ":" << (*current++).second << std::endl;
    }
}
```

36.5.2 Example of Range Tree on Set-like Data

This example illustrates the use of the range tree on 2-dimensional point data (no value is associated to a data item). After the definition of the tree, some input data items are created and the tree is constructed according to the input data items. After that, a window query is performed and the query elements are given to standard out.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Range_segment_tree_traits.h>
#include <CGAL/Range_tree_k.h>

typedef CGAL::Cartesian<double> K;
typedef CGAL::Range_segment_tree_set_traits_2<K> Traits;
typedef CGAL::Range_tree_2<Traits> Range_tree_2_type;

int main()
{
    typedef Traits::Key Key;
    typedef Traits::Interval Interval;
    std::vector<Key> InputList, OutputList;
    std::vector<Key>::iterator first, last, current;

    InputList.push_back(Key(8,5.1));
    InputList.push_back(Key(1,1.1));
    InputList.push_back(Key(3,2.1));

    Range_tree_2_type Range_tree_2(InputList.begin(),InputList.end());

    Interval win=Interval(Key(4,8.1),Key(5,8.2));
    std::cout << std::endl << "Window Query: lower left point: (4.0,5.0),";
    std::cout << "upper right point: (8.1,8.2)" << std::endl;
    Range_tree_2.window_query(win, std::back_inserter(OutputList));
    current=OutputList.begin();
    while(current!=OutputList.end()) {
        std::cout << (*current).x() << "-" << (*current).y() << std::endl;
        current++;
    }
}
```

36.6 Segment Trees

A segment tree is a static binary search tree for a given set of coordinates. The set of coordinates is defined by the endpoints of the input data intervals. Any two adjacent coordinates build an elementary interval. Every leaf corresponds to an elementary interval. Inner vertices correspond to the union of the subtree intervals of the vertex. Each vertex or leaf v contains a sublayer type (or a list, if it is one-dimensional) that will contain all intervals I , such that I contains the interval of vertex v but not the interval of the parent vertex of v .

A d -dimensional segment tree can be used to solve the following problems:

- Determine all d -dimensional intervals that contain a d -dimensional point. This query type is called “inverse range query”.

- Determine all d -dimensional intervals that enclose a given d -dimensional interval (*enclosing_query*).
- Determine all d -dimensional intervals that partially overlap or are contained in a given d -dimensional interval (*window_query*).

In Figure 36.4 an example of a one-dimensional segment tree is given. Figure 36.5 shows a two-dimensional segment tree.

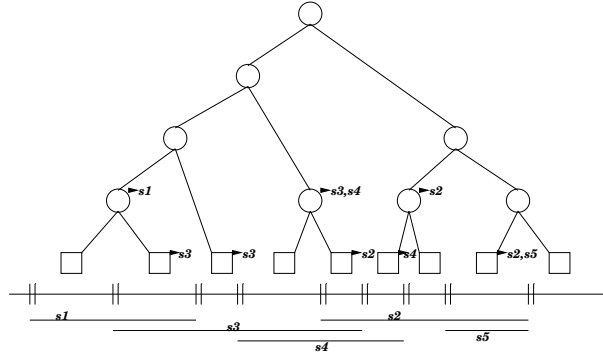


Figure 36.4: A one-dimensional segment tree. The segments and the corresponding elementary intervals are shown below the tree. The arcs from the nodes point to their subsets.

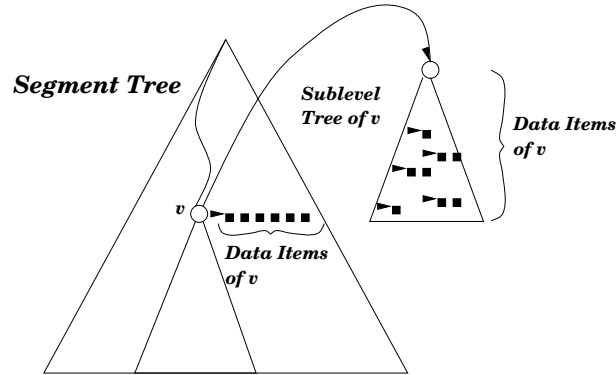


Figure 36.5: A two-dimensional segment tree. The first layer of the tree is built according to the elementary intervals of the first dimension. Each sublevel tree of a vertex v is a segment tree according to the second dimension of all data items of v .

The tree can be built in $O(n \log^d n)$ time and needs $O(n \log^d n)$ space. The processing time for inverse range queries in an d -dimensional segment tree is $O(\log^d n + k)$ time, where n is the total number of intervals and k is the number of reported intervals.

One possible application of a two-dimensional segment tree is the following. Given a set of convex polygons in two-dimensional space (CGAL::Polygon_2), we want to determine all polygons that intersect a given rectangular query window. Therefore, we define a two-dimensional segment tree, where the two-dimensional interval of a data item corresponds to the bounding box of a polygon and the value type corresponds to the polygon itself. The segment tree is created with a sequence of all data items, and a window query is performed. The polygons of the resulting data items are finally tested independently for intersections.

36.6.1 Example of Segment Tree on Map-like Data

The following example program uses the predefined *Segment_tree_2* data structure together with the predefined traits class *Segment_tree_map_traits_2* which has two template arguments specifying the type of the point data in each dimension (*CGAL::Cartesian<double>*) and the value type of the 2-dimensional point data (*char*). Therefore the *Segment_tree_2* is defined on 2-dimensional point data (*CGAL::Point_2<Cartesian<double>>*) each of which is associated with a character. Then, a few data items are created and put into a list. After that the tree is constructed according to that list, a window query is performed, and the query elements are given out.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Segment_tree_k.h>
#include <CGAL/Range_segment_tree_traits.h>

typedef CGAL::Cartesian<double> K;
typedef CGAL::Segment_tree_map_traits_2<K, char> Traits;
typedef CGAL::Segment_tree_2<Traits> Segment_tree_2_type;

int main()
{
    typedef Traits::Interval Interval;
    typedef Traits::Pure_interval Pure_interval;
    typedef Traits::Key Key;
    std::list<Interval> InputList, OutputList1, OutputList2;

    InputList.push_back(Interval(Pure_interval(Key(1,5), Key(2,7)), 'a'));
    InputList.push_back(Interval(Pure_interval(Key(2,7), Key(3,8)), 'b'));
    InputList.push_back(Interval(Pure_interval(Key(6,9), Key(9,13)), 'c'));
    InputList.push_back(Interval(Pure_interval(Key(1,3), Key(3,9)), 'd'));

    Segment_tree_2_type Segment_tree_2(InputList.begin(), InputList.end());

    Interval a=Interval(Pure_interval(Key(3,6), Key(7,12)), 'e');
    Segment_tree_2.window_query(a, std::back_inserter(OutputList1));

    std::list<Interval>::iterator j = OutputList1.begin();
    std::cout << "\n window_query (3,6), (7,12)\n";
    while(j!=OutputList1.end()){
        std::cout << (*j).first.first.x() << "-" << (*j).first.second.x() << " "
            << (*j).first.first.y() << "-" << (*j).first.second.y() << std::endl;
        j++;
    }

    Interval b=Interval(Pure_interval(Key(6,10), Key(7,11)), 'f');
    Segment_tree_2.enclosing_query(b, std::back_inserter(OutputList2));
    j = OutputList2.begin();
    std::cout << "\n enclosing_query (6,10), (7,11)\n";
    while(j!=OutputList2.end()){
        std::cout << (*j).first.first.x() << "-" << (*j).first.second.x() << " "
            << (*j).first.first.y() << "-" << (*j).first.second.y() << std::endl;
        j++;
    }
    return 0;
}
```

36.6.2 Example of Segment Tree on Set-like Data

This example illustrates the use of the predefined segment tree on 3-dimensional interval data (with no value associated). After the definition of the traits type and tree type, some intervals are constructed and the tree is build according to the intervals. Then, a window query is performed and the query elements are given out.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Segment_tree_k.h>
#include <CGAL/Range_segment_tree_traits.h>

typedef CGAL::Cartesian<int> K;
typedef CGAL::Range_segment_tree_set_traits_3<K> Traits;
typedef CGAL::Segment_tree_3<Traits > Segment_tree_3_type;

int main()
{
    typedef Traits::Interval Interval;
    typedef Traits::Key Key;
    std::list<Interval> InputList, OutputList;

    InputList.push_back(Interval(Key(1,5,7), Key(2,7,9)));
    InputList.push_back(Interval(Key(2,7,6), Key(3,8,9)));
    InputList.push_back(Interval(Key(6,9,5), Key(9,13,8)));
    InputList.push_back(Interval(Key(1,3,4), Key(3,9,8)));

    Segment_tree_3_type Segment_tree_3(InputList.begin(), InputList.end());

    Interval a(Key(3,6,5), Key(7,12,8));
    Segment_tree_3.window_query(a, std::back_inserter(OutputList));
    std::list<Interval>::iterator j = OutputList1.begin();
    std::cout << "\n window_query (3,6,5), (7,12,8) \n";
    while(j!=OutputList.end()){
        std::cout << (*j).first.x() << "," << (*j).first.y() << ",";
        std::cout << (*j).first.z() <<"," << (*j).second.x() << ",";
        std::cout << (*j).second.y() << "," << (*j).second.z() << std::endl;
        j++;
    }
}
```

dD Range and Segment Trees

Reference Manual

Gabriele Neyer

This chapter presents the CGAL range tree and segment tree data structures.

The range tree is theoretically superior to the *Kd*-tree, but the latter often seems to perform better. However, the range tree as implemented in CGAL is more flexible than the *Kd*-tree implementation, in that it enables to layer together range trees and segment trees in the same data structure.

36.7 Classified Reference Pages

Concepts

RangeSegmentTreeTraits_k	page 2121
Sublayer	page 2139

Traits Classes

CGAL::Range_segment_tree_traits_set_2<R>	page 2123
CGAL::Range_segment_tree_traits_set_3<R>	page 2124
CGAL::Range_tree_traits_map_2<R,T>	page 2130
CGAL::Range_tree_traits_map_3<R,T>	page 2131
CGAL::Segment_tree_traits_map_2<R,T>	page 2137
CGAL::Segment_tree_traits_map_3<R,T>	page 2138
CGAL::tree_interval_traits	page 2140
CGAL::tree_point_traits	page 2142

Search Structure Classes

CGAL::Range_tree_d<Data, Window, Traits>	page 2125
CGAL::Range_tree_k<Traits>	page 2128
CGAL::Segment_tree_d<Data, Window, Traits>	page 2132
CGAL::Segment_tree_k<Traits>	page 2134
CGAL::Tree_anchor<Data, Window>	page 2144

36.8 Alphabetical List of Reference Pages

<i>RangeSegmentTreeTraits_k</i>	page 2121
<i>Range_segment_tree_traits_set_2<R></i>	page 2123
<i>Range_segment_tree_traits_set_3<R></i>	page 2124
<i>Range_tree_d<Data, Window, Traits></i>	page 2125
<i>Range_tree_k<Traits></i>	page 2128
<i>Range_tree_traits_map_2<R,T></i>	page 2130
<i>Range_tree_traits_map_3<R,T></i>	page 2131
<i>Segment_tree_d<Data, Window, Traits></i>	page 2132
<i>Segment_tree_k<Traits></i>	page 2134
<i>Segment_tree_traits_map_2<R,T></i>	page 2137
<i>Segment_tree_traits_map_3<R,T></i>	page 2138
<i>Sublayer</i>	page 2139
<i>Tree_anchor<Data, Window></i>	page 2144
<i>tree_interval_traits</i>	page 2140
<i>tree_point_traits</i>	page 2142

RangeSegmentTreeTraits_k

Definition

A tree traits class gives the range tree and segment tree class the necessary type information of the keys and intervals. Further more, they define function objects that allow to access the keys and intervals, and provide comparison functions that are needed for window queries.

Types

<i>RangeSegmentTreeTraits_k:: Key</i>	The k-dimensional key type.
<i>RangeSegmentTreeTraits_k:: Interval</i>	The k-dimensional interval type.
<i>RangeSegmentTreeTraits_k:: Key_i</i>	The type in dimension i , with $1 \leq i \leq k$.
<i>RangeSegmentTreeTraits_k:: key_i</i>	function object providing an <i>operator()</i> that takes an argument of type <i>Key</i> and returns a component of type <i>Key_i</i> .
<i>RangeSegmentTreeTraits_k:: low_i</i>	function object providing an <i>operator()</i> that takes an argument of type <i>Interval</i> and returns a component of type <i>Key_i</i> .
<i>RangeSegmentTreeTraits_k:: high_i</i>	function object providing an <i>operator()</i> that takes an argument of type <i>Interval</i> and returns a component of type <i>Key_i</i> .
<i>RangeSegmentTreeTraits_k:: compare_i</i>	function object providing an <i>operator()</i> that takes two arguments argument a, b of type <i>Key_i</i> and returns true if $a < b$, false otherwise.

Example

The following piece of code gives an example of how a traits class might look like, if you have keys that are of the type *int* in the first and that are of the type *double* in the second dimension.

```
class Int_double_tree_traits_2{
public:
    typedef std::pair<int, double> Key;
    typedef int Key_1;
    typedef double Key_2;
    typedef std::pair<Key,Key> Interval;

    class C_Key_1{
    public:
        Key_1 operator()(const Key& k)
        { return k.first;}
    };
    class C_Key_2{
```

```

public:
    Key_2 operator() (const Key& k)
    { return k.second;}
};
class C_Low_1{
public:
    Key_1 operator() (const Interval& i)
    { return i.first.first;}
};
class C_High_1{
public:
    Key_1 operator() (const Interval& i)
    { return i.second.first;}
};
class C_Low_2{
public:
    Key_2 operator() (const Interval& i)
    { return i.first.second;}
};
class C_High_2{
public:
    Key_2 operator() (const Interval& i)
    { return i.second.second;}
};
class C_Compare_1{
public:
    bool operator() (Key_1 k1, Key_1 k2)
    { return less<int>() (k1,k2);}
};
class C_Compare_2{
public:
    bool operator() (Key_2 k1, Key_2 k2)
    { return less<double>() (k1,k2);}
};
typedef C_Compare_1 compare_1;
typedef C_Compare_2 compare_2;
typedef C_Low_1 low_1;
typedef C_High_1 high_1;
typedef C_Key_1 key_1;
typedef C_Low_2 low_2;
typedef C_High_2 high_2;
typedef C_Key_2 key_2;
};

```

CGAL::Range_segment_tree_traits_set_2<R>

Definition

The class is a range and segment tree traits class for the 2-dimensional point class from the CGAL kernel. The class is parameterized with a representation class R .

```
#include <CGAL/Range_segment_tree_traits.h>
```

Types

```
Point_2<R>                    Key;
```

```
std::pair<Key, Key>          Interval;
```

CGAL::Range_segment_tree_traits_set_3<R>

Definition

The class is a range and segment tree traits class for the 3-dimensional point class from the CGAL kernel. The class is parameterized with a representation class R .

```
#include <CGAL/Range_segment_tree_traits.h>
```

Types

```
Point_3<R>                    Key;
```

```
std::pair<Key, Key>          Interval;
```

CGAL::Range_tree_d<Data, Window, Traits>

Types

Range_tree_d<Data, Window, Traits>::Data container *Data*.

Range_tree_d<Data, Window, Traits>::Window
container *Window*.

Range_tree_d<Data, Window, Traits>::Traits container *Traits*.

Creation

```
#include <CGAL/Range_tree_d.h>
Range_tree_d<Data, Window, Traits> r( Tree_base<Data, Window> sublayer_tree);
```

A range tree is constructed, such that the subtree of each vertex is of the same type prototype *sublayer_tree* is.

We assume that the dimension of the tree is d . This means, that *sublayer_tree* is a prototype of a $d - 1$ -dimensional tree. All data items of the d -dimensional range tree have container type *Data*. The query window of the tree has container type *Window*. *Traits* provides access to the corresponding data slots of container *Data* and *Window* for the d -th dimension. The traits class *Traits* must at least provide all functions and type definitions as described in, for example, the reference page for *tree_point_traits*. The template class described there is fully generic and should fulfill the most requirements one can have. In order to generate a one-dimensional range tree instantiate *Tree_anchor<Data, Window> sublayer_tree* with the same template parameters (*Data* and *Window*) *Range_tree_d* is defined. In order to construct a two-dimensional range tree, create *Range_tree_d* with a one-dimensional *Range_tree_d* with the corresponding *Traits* class of the first dimension.

Precondition: *Traits::Data==Data* and *Traits::Window==Window*.

Operations

```
template<class ForwardIterator>
bool r.make_tree( ForwardIterator first, ForwardIterator last)
```

The tree is constructed according to the data items in the sequence between the element pointed by iterator *first* and iterator *last*. The data items of the iterator must have type *Data*.

Precondition: This function can only be called once. If it is the first call the tree is build and *true* is returned. Otherwise, nothing is done but a *CGAL warning* is given and *false* returned.

```
template<class OutputIterator>
OutputIterator r.window_query( Window win, OutputIterator result)
```

All elements that lay inside the d -dimensional interval defined through *win* are placed in the sequence container of *OutputIterator*; the output iterator that points to the last location the function wrote to is returned.

```
bool r.is_valid()
```

The tree structure is checked. For each vertex the subtree is checked on being valid and it is checked whether the value of the *Key_type* of a vertex corresponds to the highest *Key_type* value of the left subtree.

Protected Operations

bool *r.is_inside(Window win, Data object)*

returns true, if the data of *object* lies between the start and endpoint of interval *win*. False otherwise.

bool *r.is_anchor()* returns false.

Implementation

The construction of a d -dimensional range tree takes $O(n \log n^{d-1})$ time. The points in the query window are reported in time $O(k + \log^d n)$, where k is the number of reported points. The tree uses $O(n \log n^{d-1})$ storage.

CGAL::Range_tree_k<Traits>

Definition

An object of the class is a k -dimensional range tree that can store k -dimensional keys of type *Key*. The class allows to perform window queries on the keys. The class is parameterized with a range tree traits class *Traits* that defines, among other things, the type of the *Key*.

CGAL provides traits class implementations that allow to use the range tree with point classes from the CGAL kernel as keys. These classes are *CGAL::Range_segment_tree_traits_set_2<R>*, *CGAL::Range_segment_tree_traits_set_3<R>*, *CGAL::Range_tree_traits_map_2<R>* and *CGAL::Range_tree_traits_map_3<R>*. The concept *RangeSegmentTreeTraits_d* defines the requirements that range tree traits classes must fulfill. This allows the advanced user to develop further range tree traits classes.

```
#include <CGAL/Range_tree_k.h>
```

Types

Range_tree_k<Traits>::Traits the type of the range tree traits class.

```
typedef Traits::Key Key;
```

```
typedef Traits::Interval  
Interval;
```

Creation

Range_tree_k<Traits> R; Introduces an empty range tree *R*.

```
template < class ForwardIterator >  
Range_tree_k<Traits> R( ForwardIterator first, ForwardIterator last);
```

Introduces a range tree *R* and initializes it with the data in the range *[first, last)*.
Precondition: *value_type(first) == Traits::Key*.

Operations

```
template < class ForwardIterator >  
void R.make_tree( ForwardIterator first, ForwardIterator last)
```

Introduces a range tree *R* and initializes it with the data in the range *[first, last)*. This function can only be applied once on an empty range tree.
Precondition: *value_type(first) == Traits::Key*.


```
template < class OutputIterator >
OutputIterator      R.window_query( Interval window, OutputIterator out)
```

writes all data that are in the interval *window* to the container where *out* points to, and returns an output iterator that points to the last location the function wrote to.

Precondition: `value_type(out) == Traits::Key`.

Example

The following example program uses the predefined *Range_tree_2* data structure together with the predefined traits class *Range_tree_map_traits_2* which has two template arguments specifying the type of the point data in each dimension (*CGAL::Cartesian<double>*) and the value type of the 2-dimensional point data (*char*). Therefore the *Range_tree_2* is defined on 2-dimensional point data (*CGAL::Point_2<Cartesian<double> >*) each of which is associated with a character. Then, a few data items are created and put into a list. After that the tree is constructed according to that list, a window query is performed, and the query elements are given out.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Range_segment_tree_traits.h>
#include <CGAL/Range_tree_k.h>

typedef CGAL::Cartesian<double> K;
typedef CGAL::Range_tree_map_traits_2<K, char> Traits;
typedef CGAL::Range_tree_2<Traits> Range_tree_2_type;

int main()
{
    typedef Traits::Key Key;
    typedef Traits::Interval Interval;

    std::vector<Key> InputList, OutputList;
    InputList.push_back(Key(K::Point_2(8,5.1), 'a'));
    InputList.push_back(Key(K::Point_2(1,1.1), 'b'));
    InputList.push_back(Key(K::Point_2(3,2.1), 'c'));

    Range_tree_2_type Range_tree_2(InputList.begin(), InputList.end());
    Interval win(Interval(K::Point_2(4,8.1), K::Point_2(5,8.2)));
    std::cout << "\n Window Query:\n ";
    Range_tree_2.window_query(win, std::back_inserter(OutputList));
    std::vector<Key>::iterator current=OutputList.begin();
    while(current!=OutputList.end()) {
        std::cout << (*current).first.x() << "," << (*current).first.y()
            << ":" << (*current++).second << std::endl;
    }
}
```

CGAL::Range_tree_traits_map_2<R,T>

Definition

The class is a range tree traits class for the 2-dimensional point class from the CGAL kernel, where data of type T is associated to each key. The class is parameterized with a representation class R and the type of the associated data T .

```
#include <CGAL/Range_segment_tree_traits.h>
```

Types

```
std::pair<R::Point_2,T>
```

Key;

```
std::pair<R::Point_2, R::Point_2 >
```

Interval;

CGAL::Range_tree_traits_map_3<R,T>

Definition

The class is a range and segment tree traits class for the 3-dimensional point class from the CGAL kernel, where data of type T is associated to each key. The class is parameterized with a representation class R and the type of the associated data T .

```
#include <CGAL/Range_segment_tree_traits.h>
```

Types

```
std::pair<R::Point_3,T>
```

```
Key;
```

```
std::pair<R::Point_3, R::Point_3>
```

```
Interval;
```

CGAL::Segment_tree_d<Data, Window, Traits>

Types

Segment_tree_d<*Data*, *Window*, *Traits*>::*Data*

container *Data*.

Segment_tree_d<*Data*, *Window*, *Traits*>::*Window*

container *Window*.

Segment_tree_d<*Data*, *Window*, *Traits*>::*Traits*

class *Traits*.

Creation

```
#include <CGAL/Segment_tree_d.h>
```

```
Segment_tree_d<Data, Window, Traits> s( Tree_base<Data, Window> sublayer_tree);
```

A segment tree is defined, such that the subtree of each vertex is of the same type prototype *sublayer_tree* is.

We assume that the dimension of the tree is d . This means, that *sublayer_tree* is a prototype of a $d - 1$ -dimensional tree. All data items of the d -dimensional segment tree have container type *Data*. The query window of the tree has container type *Window*. *Traits* provides access to the corresponding data slots of container *Data* and *Window* for the d -th dimension. The traits class *Traits* must at least provide all functions and type definitions described, for example, in the reference page for *tree_point_traits*. The template class described there is fully generic and should fulfill the most requirements one can have. In order to generate a one-dimensional segment tree instantiate *Tree_anchor*<*Data*, *Window*> *sublayer_tree* with the same template parameters *Data* and *Window* *Segment_tree_d* is defined. In order to construct a two-dimensional segment tree, create *Segment_tree_d* with a one-dimensional *Segment_tree_d* with the corresponding *Traits* of the first dimension.

Precondition: *Traits*::*Data*==*Data* and *Traits*::*Window*==*Window*.

Operations

bool *s.make_tree(In_it first, In_it last)*

The tree is constructed according to the data items in the sequence between the element pointed by iterator *first* and iterator *last*.

Precondition: This function can only be called once. If it is the first call the tree is build and *true* is returned. Otherwise, nothing is done but a *CGAL warning* is given and *false* returned.

OutputIterator *s.window_query(Window win, OutputIterator result)*

$win = [a_1, b_1), \dots, [a_d, b_d), a_i, b_i \in T_i, 1 \leq i \leq d$. All elements that intersect the associated d -dimensional interval of *win* are placed in the associated sequence container of *OutputIterator* and returns an output iterator that points to the last location the function wrote to. In order to perform an inverse range query, a range query of ϵ width has to be performed.

OutputIterator *s.enclosing_query(Window win, OutputIterator result)*

All elements that enclose the associated d -dimensional interval of *win* are placed in the associated sequence container of *OutputIterator* and returns an output iterator that points to the last location the function wrote to.

bool *s.is_valid()*

The tree structure is checked. For each vertex either the sublayer tree is a tree anchor, or it stores a (possibly empty) list of data items. In the first case, the sublayer tree of the vertex is checked on being valid. In the second case, each data item is checked weather it contains the associated interval of the vertex and does not contain the associated interval of the parent vertex or not. True is returned if the tree structure is valid, false otherwise.

Protected Operations

bool *s.is_inside(Window win, Data object)*

returns true, if the interval of *object* is contained in the interval of *win*. False otherwise.

bool *s.is_anchor()* returns false.

Implementation

A d -dimensional segment tree is constructed in $O(n \log n^d)$ time. An inverse range query is performed in time $O(k + \log^d n)$, where k is the number of reported intervals. The tree uses $O(n \log n^d)$ storage.

CGAL::Segment_tree_k<Traits>

Definition

An object of the class is a k -dimensional segment tree that can store k -dimensional intervals of type *Interval*. The class allows to perform window queries, enclosing queries, and inverse range queries on the keys. The class is parameterized with a segment tree traits class *Traits* that defines, among other things, the type of the *Interval*. In order to perform an inverse range query, a range query of ε width has to be performed. We preferred not to offer an extra function for this sort of query, since the inverse range query is a special case of the range query. Furthermore, offering an inverse range query in the segment tree class implies offering this function also in the range tree class and having an extra item in the traits class that accesses the inverse range query point.

CGAL provides traits class implementations that allow to use the segment tree with point classes from the CGAL kernel as keys. These classes are *CGAL::Range_segment_tree_traits_set_2<R>*, *CGAL::Range_segment_tree_traits_set_3<R>*, *CGAL::Segment_tree_traits_map_2<R>* and *CGAL::Segment_tree_traits_map_3<R>*. The concept *RangeSegmentTreeTraits_d* defines the requirements that segment tree traits classes must fulfill. This allows the advanced user to develop further segment tree traits classes.

```
#include <CGAL/Segment_tree_k.h>
```

Types

Segment_tree_k<Traits>::Traits the type of the segment tree traits class.

```
typedef Traits::Key      Key;
```

```
typedef Traits::Interval  
                        Interval;
```

Creation

Segment_tree_k<Traits> S; Introduces an empty segment tree S .

```
template < class ForwardIterator >  
Segment_tree_k<Traits> S( ForwardIterator first, ForwardIterator last);
```

Introduces a segment tree S and initializes it with the data in the range $[first, last)$.
Precondition: $value_type(first) == Traits::Interval$.

Operations

```
template < class ForwardIterator >  
void S.make_tree( ForwardIterator first, ForwardIterator last)
```

Introduces a segment tree S and initializes it with the data in the range $[first, last)$. This function can only be applied once on an empty segment tree.
Precondition: $value_type(first) == Traits::Interval$.

```
template < class OutputIterator >
OutputIterator      S.window_query( Interval window, OutputIterator out)
```

writes all intervals that have non empty intersection with interval *window* to the container where *out* points to, and returns an output iterator that points to the last location the function wrote to.

Precondition: `value_type(out) == Traits::Interval`.

```
template < class OutputIterator >
OutputIterator      S.enclosing_query( Interval window, OutputIterator out)
```

writes all intervals that enclose in the interval *window* to the container where *out* points to, and returns an output iterator that points to the last location the function wrote to.

Precondition: `value_type(out) == Traits::Interval`.

Example

This example illustrates the use of the predefined segment tree on 3-dimensional interval data (with no value associated). After the definition of the traits type and tree type, some intervals are constructed and the tree is build according to the intervals. Then, a window query is performed and the query elements are given out.

```
#include <CGAL/Cartesian.h>
#include <CGAL/Segment_tree_k.h>
#include <CGAL/Range_segment_tree_traits.h>

typedef CGAL::Cartesian<int> K;
typedef CGAL::Range_segment_tree_set_traits_3<K> Traits;
typedef CGAL::Segment_tree_3<Traits> Segment_tree_3_type;

int main()
{
    typedef Traits::Interval Interval;
    typedef Traits::Key Key;
    std::list<Interval> InputList, OutputList;

    InputList.push_back(Interval(Key(1,5,7), Key(2,7,9)));
    InputList.push_back(Interval(Key(2,7,6), Key(3,8,9)));
    InputList.push_back(Interval(Key(6,9,5), Key(9,13,8)));
    InputList.push_back(Interval(Key(1,3,4), Key(3,9,8)));

    Segment_tree_3_type Segment_tree_3(InputList.begin(),InputList.end());

    Interval a(Key(3,6,5), Key(7,12,8));
    Segment_tree_3.window_query(a,std::back_inserter(OutputList));
    std::list<Interval>::iterator j = OutputList1.begin();
    std::cout << "\n window_query (3,6,5), (7,12,8) \n";
    while(j!=OutputList.end()){
        std::cout << (*j).first.x() << "," << (*j).first.y() << ",";
        std::cout << (*j).first.z() <<"," << (*j).second.x() << ",";
        std::cout << (*j).second.y() << "," << (*j).second.z() << std::endl;
    }
```

```
        j++;  
    }  
}
```


CGAL::Segment_tree_traits_map_2<R,T>

Definition

The class is a segment tree traits class for the 2-dimensional point class from the CGAL kernel, where data of type T is associated to each interval. The class is parameterized with a representation class R and the type of the associated data T .

```
#include <CGAL/Range_segment_tree_traits.h>
```

Types

```
R::Point_2          Key;
```

```
std::pair<std::pair<Key,Key>,T>
```

```
Interval;
```

CGAL::Segment_tree_traits_map_3<R,T>

Definition

The class is a segment tree traits class for the 3-dimensional point class from the CGAL kernel, where data of type T is associated to each interval. The class is parameterized with a representation class R and the type of the associated data T .

```
#include <CGAL/Range_segment_tree_traits.h>
```

Types

```
std::pair<R::Point_3 >
```

```
Key;
```

```
std::pair<std::pair<Key, Key>, T>
```

```
Interval;
```

Sublayer

Definition

Defines the requirements that a *Sublayer_type* of class *Range_tree_d* or *Segment_tree_d* has to fulfill.

First of all, the class has to be derived from the abstract base class *Tree_base* and therefore has to provide methods *make_tree*, *window_query*, *enclosing_query* and *is_inside* with the same parameter types as the instantiated class *Range_tree_d* or *Segment_tree_d*, respectively. Furthermore a method *bool is_anchor()* has to be provided. If the *Sublayer_type* class builds a recursion anchor for class *Segment_tree_d*, this function is expected to return *true*, *false* otherwise.

Such a recursion anchor class is provided by the class class. *Tree_anchor<Data, Window>*.

CGAL::tree_interval_traits

Definition

tree_interval_traits is a template class that provides an interface to data items. It is similar to *tree_point_traits*, except that it provides access to two data slots of the same type of each container class (*Data*, *Window*) instead of providing access to one data slot of container class *Data* and two data slots of class *Window*.

```
#include <CGAL/Tree_traits.h>
```

Types

```
typedef
tree_interval_traits<Data, Window, Key, Data_left_func, Data_right_func, Window_left_func, Window_right_
func, Compare> Interval_traits;
```

tree_interval_traits::Data the container *Data* — the data type. It may consist of several data slots. Two of these data slots have to be of type *Key*.

tree_interval_traits::Window the container *Window* — the query window type. It may consist of several data slots. Two of these data slots have to be of type *Key*.

tree_interval_traits::Key the type *Key* of the data slot this traits class provides access to.

tree_interval_traits::Data_left_func *Data_left_func* is a function object providing an *operator()* that takes an argument of type *Data* and returns a (the left) component of type *Key*.

tree_interval_traits::Data_right_func *Data_right_func* is a function object providing an *operator()* that takes an argument of type *Data* and returns a (the right) component of type *Key*.

tree_interval_traits::Window_left_func *Window_left_func* is a function objects that allow to access the left data slot of container *Window* which has type *Key*

tree_interval_traits::Window_right_func *Window_right_func* is a function objects that allow to access the right data slot of container *Window* which has type *Key*

tree_interval_traits::Compare defines a comparison relation which must define a strict ordering of the objects of type *Key*. If defined, *less<Key>* is sufficient.

Creation

```
tree_interval_traits<Data, Window, Key, Data_left_func, Data_right_func, Window_left_func, Window_right_
func, Compare> d();
```

Generation of a *tree_point_traits* instance. It is a template class that provides an interface to data items.

Operations

<i>Key</i>	<i>d.get_left(Data d)</i>	The data slot of the data item of <i>d</i> of type <i>Key</i> is accessed by function object <i>Data_left_func</i> .
<i>Key</i>	<i>d.get_right(Data d)</i>	The data slot of the data item of <i>d</i> of type <i>Key</i> is accessed by function object <i>Data_right_func</i> .
<i>Key</i>	<i>d.get_left_win(Window w)</i>	The data slot of the data item of <i>w</i> of type <i>Key</i> is accessed by function object <i>Window_left_func</i> .
<i>Key</i>	<i>d.get_right_win(Window w)</i>	The data slot of the data item of <i>w</i> of type <i>Key</i> is accessed by function object <i>Window_right_func</i> .
<i>bool</i>	<i>d.comp(Key& key1, Key& key2)</i>	returns Compare(key1, key2).

CGAL::tree_point_traits

Definition

tree_point_traits is a template class that provides an interface to data items.

```
#include <CGAL/Tree_traits.h>
```

Types

typedef

```
tree_point_traits<Data, Window, Key, Data_func, Window_left_func, Window_right_func, Compare> Point_traits;
```

tree_point_traits::Data

the container *Data* — defines the *Data* type. It may consist of several data slots. One of these data slots has to be of type *Key*.

tree_point_traits::Window

the container *Window* — defines the type of the query rectangle. It may consist of several data slots. Two of these data slots has to be of type *Key*

tree_point_traits::Key

the type *Key* of the data slot this traits class provides access to.

tree_point_traits::Data_func

Data_func is a function object providing an *operator()* that takes an argument of type *Data* and returns a component of type *Key*.

tree_point_traits::Window_left_func

Window_left_func is a function objects that allow to access the left data slot of container *Window* which has type *Key*

tree_point_traits::Window_right_func

Window_right_func is a function objects that allow to access the right data slot of container *Window* which has type *Key*

tree_point_traits::Compare

defines a comparison relation which must define a strict ordering of the objects of type *Key*. If defined, *less<Key>* is sufficient.

Creation

```
tree_point_traits<Data, Window, Key, Data_func, Window_left_func, Window_right_func, Compare> d();
```

Generation of a *tree_point_traits* instance. It is a template class that provides an interface to data items.

Operations

<i>Key</i>	<i>d.get_key(Data d)</i>	The data slot of the data item of <i>d</i> of type <i>Key</i> is accessed by function object <i>Data_func</i> .
<i>Key</i>	<i>d.get_left(Window w)</i>	The data slot of the data item of <i>w</i> of type <i>Key</i> is accessed by function object <i>Window_left_func</i> .
<i>Key</i>	<i>d.get_right(Window w)</i>	The data slot of the data item of <i>w</i> of type <i>Key</i> is accessed by function object <i>Window_right_func</i> .
<i>bool</i>	<i>d.comp(Key& key1, Key& key2)</i>	returns <i>Compare(key1, key2)</i> .
<i>bool</i>	<i>d.key_comp(Data& data1, Data& data2)</i>	returns <i>Compare(get_key(data1), get_key(data2))</i> .

CGAL::Tree_anchor<Data, Window>

Definition

Tree_anchor is also derived from *Tree_base*. Therefore, it provides the same methods as *Range_tree_d* and *Segment_tree_d*, but does nothing; it can be used as a recursion anchor for those classes. Therefore, instantiate *Sublayer_type* of *Range_tree_d* (*Segment_tree_d* respectively) with *Tree_anchor* and the container classes for the data items (*Data* and *Window*).

Definition

Tree_anchor<Data, Window>::Data container *Data*.

Tree_anchor<*Data*, *Window*>:: *Window* container *Window*.

Creation

```
#include <CGAL/Tree_base.h>
```

$$Tree_anchor<Data, Window> \ a;$$

Operations

```
template<class OutputIterator>
OutputIterator      a.window_query( Window win, OutputIterator result)
```

```
template<class OutputIterator>
OutputIterator      a.enclosing_query( Window win, OutputIterator result)
```

bool *a.is_valid()* returns true;

Protected Operations

bool *a.is_inside(Window win, Data object)*

returns true.

bool *a.is_anchor()* returns true.

Example

The following figures show a number of rectangles and a 2-dimensional segment tree built on them.

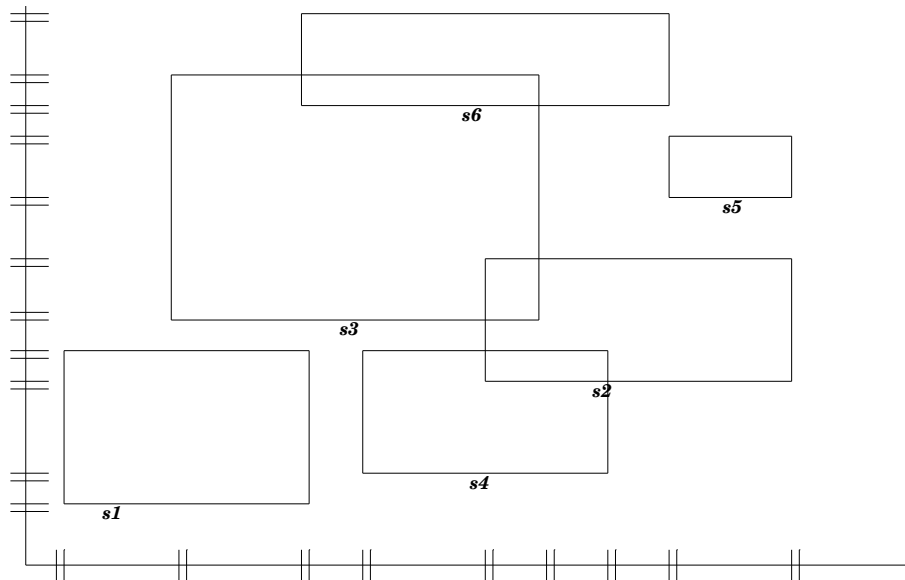


Figure 36.6: Two dimensional interval data.

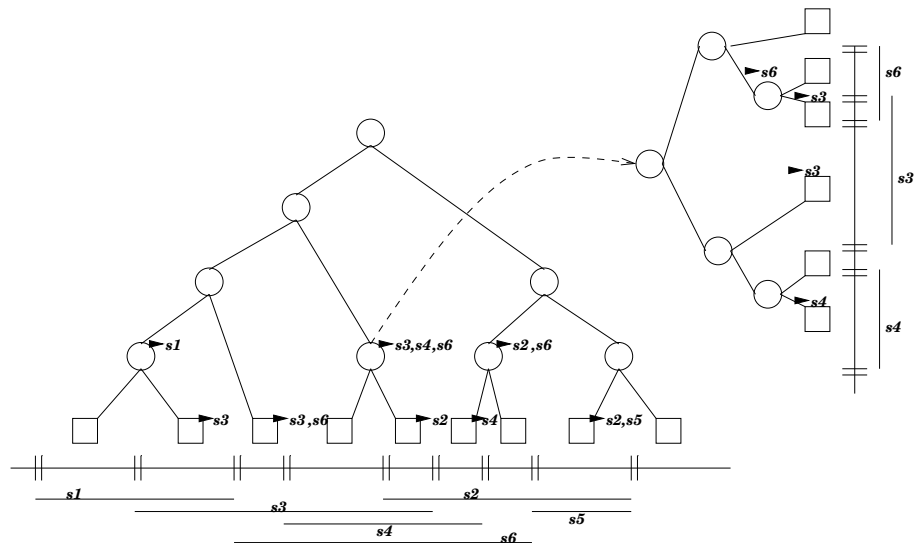
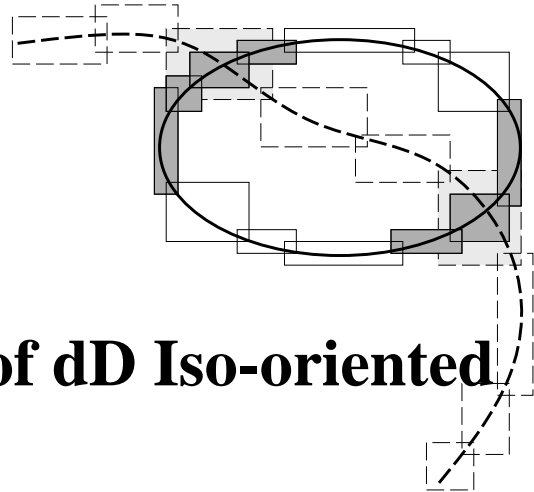


Figure 36.7: Two dimensional segment tree according to the interval data of Figure 36.6.

Chapter 37

Intersecting Sequences of dD Iso-oriented Boxes

Lutz Kettner, Andreas Meyer, and Afra Zomorodian



Contents

37.1 Introduction	2147
37.2 Definition	2148
37.3 Software Design	2148
37.4 Minimal Example for Intersecting Boxes	2149
37.5 Example for Finding Intersecting 3D Triangles	2150
37.6 Example for Using Pointers to Boxes	2152
37.7 Example Using the <i>topology</i> and the <i>cutoff</i> Parameters	2152
37.8 Runtime Performance	2154
37.9 Example Using a Custom Box Implementation	2155
37.10 Example for Point Proximity Search with a Custom Traits Class	2157
37.11 Design and Implementation History	2158

37.1 Introduction

Simple questions on geometric primitives, such as intersection and distance computations, can themselves become quite expensive if the primitives are not so simple anymore, for example, three-dimensional triangles and facets of polyhedral surfaces. Thus algorithms operating on these primitives tend to be slow in practice. A common (heuristic) optimization approximates the geometric primitives with their axis-aligned bounding boxes, runs a suitable modification of the algorithm on the boxes, and whenever a pair of boxes has an interesting interaction, only then the exact answer is computed on the complicated geometric primitives contained in the boxes.

We provide an efficient algorithm [ZE02] for finding all intersecting pairs for large numbers of iso-oriented boxes, i.e., typically these will be such bounding boxes of more complicated geometries. One immediate application of this algorithm is the detection of all intersections (and self-intersections) for polyhedral surfaces, i.e., applying the algorithm on a large set of triangles in space, we give an example program later in this chapter. Not so obvious applications are proximity queries and distance computations among such surfaces, see Section 37.10 for an example and [ZE02] for more details.

37.2 Definition

A d -dimensional iso-oriented box is defined as the Cartesian product of d intervals. We call the box *half-open* if the d intervals $\{[lo_i, hi_i) \mid 0 \leq i < d\}$ are half-open intervals, and we call the box *closed* if the d intervals $\{[lo_i, hi_i] \mid 0 \leq i < d\}$ are closed intervals. Note that closed boxes support zero-width boxes and they can intersect at their boundaries, while non-empty half-open boxes always have a positive volume and they only intersect iff their interiors overlap. The distinction between closed and half-open boxes does not require a different representation of boxes, just a different interpretation when comparing boxes, which is selected with the two possible values for the *topology* parameter:

- `CGAL::Box_intersection_d::HALF_OPEN` and
- `CGAL::Box_intersection_d::CLOSED`.

The number type of the interval boundaries must be one of the builtin types `int`, `unsigned int`, `double` or `float`.

In addition, a box has an unique *id*-number. It is used to order boxes consistently in each dimension even if boxes have identical coordinates. In consequence, the algorithm guarantees that a pair of intersecting boxes is reported only once. Note that boxes with equal *id*-number are not reported since they obviously intersect trivially.

The box intersection algorithm comes in two flavors: One algorithm works on a single sequence of boxes and computes all pairwise intersections, which is called the *complete* case, and used, for example, in the self-intersection test. The other algorithm works on two sequences of boxes and computes the pairwise intersections between boxes from the first sequence with boxes from the second sequence, which is called the *bipartite* case. For each pairwise intersection found a callback function is called with two arguments; the first argument is a box from the first sequence and the second argument a box from the second sequence. In the complete case, the second argument is a box from an internal copy of the first sequence.

37.3 Software Design

The box intersection algorithm is implemented as a family of generic functions; the functions for the complete case accept one iterator range, and the functions for the bipartite case accept two iterator ranges. The callback function for reporting the intersecting pairs is provided as a template parameter of the *BinaryFunction* concept. The two principle function calls utilizing all default arguments look as follows:

```
#include <CGAL/box_intersection_d.h>
```

```
template< class RandomAccessIterator, class Callback >
void    box_intersection_d( RandomAccessIterator begin, RandomAccessIterator end, Callback callback)
```

```
template< class RandomAccessIterator1, class RandomAccessIterator2, class Callback >
void    box_intersection_d( RandomAccessIterator1 begin1,
                          RandomAccessIterator1 end1,
                          RandomAccessIterator2 begin2,
                          RandomAccessIterator2 end2,
                          Callback callback)
```

Additional parameters to the functions calls are a *cutoff* value to adjust performance tradeoffs, and a *topology* parameter selecting between topologically closed boxes (the default) and topologically half-open boxes.

The algorithm reorders the boxes in the course of the algorithm. Now, depending on the size of a box it can be faster to copy the boxes, or to work with pointers to boxes and copy only pointers. We offer automatic support for both options. To simplify the description, let us call the *value_type* of the iterator ranges *box handle*. The *box handle* can either be our box type itself or a pointer (or const pointer) to the box type; these choices represent both options from above.

In general, the algorithms treat the box type as opaque type and just assume that they are models of the *Assignable* concept, so that the algorithms can modify the input sequences and reorder the boxes. The access to the box dimension and box coordinates is mediated with a traits class of the *BoxIntersectionTraits_d* concept. A default traits class is provided that assumes that the box type is a model of the *BoxIntersectionBox_d* concept and that the box handle, i.e., the iterators value type, is identical to the box type or a pointer to the box type (see the previous paragraph for the value versus pointer nature of the box handle).

Two implementations of iso-oriented boxes are provided; *CGAL::Box_intersection_d::Box_d* as a plain box, and *CGAL::Box_intersection_d::Box_with_handle_d* as a box plus a handle that can be used to point to the full geometry that is approximated by the box. Both implementations have template parameters for the number type used for the interval bounds, for the fixed dimension of the box, and for a policy class [Ale01] selecting among several solutions for providing the *id*-number.

The function signatures for the bipartite case look as follows. The signatures for the complete case with the *box_self_intersection_d* function look the same except for the single iterator range.

```
#include <CGAL/box_intersection_d.h>
```

```
template< class RandomAccessIterator1, class RandomAccessIterator2, class Callback >
void    box_intersection_d( RandomAccessIterator1 begin1,
                          RandomAccessIterator1 end1,
                          RandomAccessIterator2 begin2,
                          RandomAccessIterator2 end2,
                          Callback callback,
                          std::ptrdiff_t cutoff = 10,
                          Box_intersection_d::Topology topology = Box_intersection_d::CLOSED,
                          Box_intersection_d::Setting setting = Box_intersection_d::BIPARTITE)
```

```
template< class RandomAccessIterator1, class RandomAccessIterator2, class Callback, class BoxTraits >
void    box_intersection_d( RandomAccessIterator1 begin1,
                          RandomAccessIterator1 end1,
                          RandomAccessIterator2 begin2,
                          RandomAccessIterator2 end2,
                          Callback callback,
                          BoxTraits box_traits,
                          std::ptrdiff_t cutoff = 10,
                          Box_intersection_d::Topology topology = Box_intersection_d::CLOSED,
                          Box_intersection_d::Setting setting = Box_intersection_d::BIPARTITE)
```

37.4 Minimal Example for Intersecting Boxes

The box implementation provided with *CGAL::Box_intersection_d::Box_d<double,2>* has a dedicated constructor for the CGAL bounding box type *CGAL::Bbox_2* (similar for dimension 3). We use this in our minimal example to create easily nine two-dimensional *boxes* in a grid layout of 3×3 boxes. Additionally we pick the center box and the box in the upper-right corner as our second box sequence *query*.

The default policy of the box type implements the *id*-number with an explicit counter in the boxes, which is the default choice since it always works, but it costs space that could potentially be avoided, see the example in the next section. We use the *id*-number in our callback function to report the result of the intersection algorithm. The result will be that the first *query* box intersects all nine *boxes* and the second *query* box intersects the four boxes in the upper-right quadrant. See Section 37.7 for the change of the *topology* parameter and its effect.

```
// file: examples/Box_intersection_d/minimal.C
#include <CGAL/box_intersection_d.h>
#include <CGAL/Bbox_2.h>
#include <iostream>

typedef CGAL::Box_intersection_d::Box_d<double,2> Box;
typedef CGAL::Bbox_2                                Bbox;

// 9 boxes of a grid
Box boxes[9] = { Bbox( 0,0,1,1), Bbox( 1,0,2,1), Bbox( 2,0,3,1), // low
                Bbox( 0,1,1,2), Bbox( 1,1,2,2), Bbox( 2,1,3,2), // middle
                Bbox( 0,2,1,3), Bbox( 1,2,2,3), Bbox( 2,2,3,3)}; // upper
// 2 selected boxes as query; center and upper right
Box query[2] = { Bbox( 1,1,2,2), Bbox( 2,2,3,3)};

void callback( const Box& a, const Box& b ) {
    std::cout << "box " << a.id() << " intersects box " << b.id() << std::endl;
}

int main() {
    CGAL::box_intersection_d( boxes, boxes+9, query, query+2, callback);
    return 0;
}
```

37.5 Example for Finding Intersecting 3D Triangles

The conventional application of the axis-aligned box intersection algorithm will start from complex geometry, here 3D triangles, approximate them with their bounding box, compute the intersecting pairs of boxes, and check only for those if the original triangles intersect as well.

We start in the *main* function and create ten triangles with endpoints chosen randomly in a cube $[-1, +1]^3$. We store the triangles in a vector called *triangles*.

Next we create a vector for the bounding boxes of the triangles called *boxes*. For the boxes we choose the type *Box_with_handle_d<double,3,Iterator>* that works nicely together with the CGAL bounding boxes of type *CGAL::Bbox_3*. In addition, each box stores the iterator to the corresponding triangle.

The default policy of this box type uses for the *id*-number the address of the value of the iterator, i.e., the address of the triangle. This is a good choice that works correctly iff the boxes have unique iterators, i.e., there is a one-to-one mapping between boxes and approximated geometry, which is the case here. It saves us the extra space that was needed for the explicit *id*-number in the previous example.

We run the self intersection algorithm with the *report_inters* function as callback. This callback reports the intersecting boxes. It uses the *handle* and the global *triangles* vector to calculate the triangle numbers. Then it checks the triangles themselves for intersection and reports if not only the boxes but also the triangles intersect. We take some precautions before the intersection test in order to avoid problems, although unlikely, with degenerate triangles that we might have created with the random process.

This example can be easily extended to test polyhedral surfaces of the *Polyhedron_3* class for (self-) intersections. The main difference are the numerous cases of incidences between triangles in the polyhedral surface that should not be reported as intersections, see the `examples/Polyhedron/polyhedron_self_intersection.C` example program in the CGAL distribution.

```
// file: examples/Box_intersection_d/triangle_self_intersect.C
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/intersections.h>
#include <CGAL/point_generators_3.h>
#include <CGAL/Bbox_3.h>
#include <CGAL/box_intersection_d.h>
#include <CGAL/function_objects.h>
#include <CGAL/Join_input_iterator.h>
#include <CGAL/copy_n.h>
#include <vector>

typedef CGAL::Exact_predicates_inexact_constructions_kernel    Kernel;
typedef Kernel::Point_3                                         Point_3;
typedef Kernel::Triangle_3                                     Triangle_3;
typedef std::vector<Triangle_3>                                Triangles;
typedef Triangles::iterator                                    Iterator;
typedef CGAL::Box_intersection_d::Box_with_handle_d<double,3,Iterator> Box;

Triangles triangles; // global vector of all triangles

// callback function that reports all truly intersecting triangles
void report_inters( const Box& a, const Box& b) {
    std::cout << "Box " << (a.handle() - triangles.begin()) << " and "
                << (b.handle() - triangles.begin()) << " intersect";
    if ( ! a.handle()->is_degenerate() && ! b.handle()->is_degenerate()
        && CGAL::do_intersect( *(a.handle()), *(b.handle())) {
        std::cout << ", and the triangles intersect also";
    }
    std::cout << '.' << std::endl;
}

int main() {
    // Create 10 random triangles
    typedef CGAL::Random_points_in_cube_3<Point_3>              Pts;
    typedef CGAL::Creator_uniform_3< Point_3, Triangle_3>       Creator;
    typedef CGAL::Join_input_iterator_3<Pts,Pts,Pts,Creator> Triangle_gen;
    Pts    points( 1); // in centered cube [-1,1]^3
    Triangle_gen triangle_gen( points, points, points);
    CGAL::copy_n( triangle_gen, 10, std::back_inserter(triangles));

    // Create the corresponding vector of bounding boxes
    std::vector<Box> boxes;
    for ( Iterator i = triangles.begin(); i != triangles.end(); ++i)
        boxes.push_back( Box( i->bbox(), i));

    // Run the self intersection algorithm with all defaults
    CGAL::box_self_intersection_d( boxes.begin(), boxes.end(), report_inters);
    return 0;
}
```

37.6 Example for Using Pointers to Boxes

We modify the previous example, finding intersecting 3D triangles, and add an additional vector *ptr* that stores pointers to the bounding boxes, so that the intersection algorithm will work on a sequence of pointers and not on a sequence of boxes. The change just affects the preparation of the additional vector and the call of the box intersection function. The box intersection function (actually its default traits class) detects automatically that the value type of the iterators is a pointer type and not a class type.

```
// Create the corresponding vector of pointers to bounding boxes
std::vector<Box *> ptr;
for ( std::vector<Box>::iterator i = boxes.begin(); i != boxes.end(); ++i)
    ptr.push_back( &*i);

// Run the self \ccHtmlNoLinksFrom{intersection} algorithm with all defaults on the
// indirect pointers to bounding boxes. Avoids copying the boxes.
CGAL::box_self_intersection_d( ptr.begin(), ptr.end(), report_inters);
```

In addition, the callback function *report_inters* needs to be changed to work with pointers to boxes. See the following file for the full example program.

```
examples/Box_intersection_d/triangle_self_intersect_pointers.C
```

A note on performance: The algorithm sorts and partitions the input sequences. It is clearly costly to copy a large box compared to a simple pointer. However, the algorithm benefits from memory locality in the later stages when it copies the boxes, while the pointers would refer to boxes that become wildly scattered in memory. These two effects, copying costs and memory locality, counteract each other. For small box sizes, i.e., small dimension, memory locality wins and one should work with boxes, while for larger box sizes one should work with pointers. The exact threshold depends on the memory hierarchy (caching) of the hardware platform and the size of the boxes, most notably the type used to represent the box coordinates. A concrete example; on a laptop with an Intel Mobile Pentium4 running at 1.80GHz with 512KB cache and 254MB main memory under Linux this version with pointers was 20% faster than the version above that copies the boxes for 10000 boxes, but the picture reversed for 100000 boxes, where the version above that copies the boxes becomes 300% faster.

Note that switching to the builtin type *float* is supported by the box intersection algorithm, but the interfacing with the CGAL bounding box *CGAL::Bbox_3* would not be that easy. In particular, just converting from the *double* to the *float* representation incurs rounding that needs to be controlled properly, otherwise the box might shrink and one might miss intersections.

37.7 Example Using the *topology* and the *cutoff* Parameters

Boxes can be interpreted by the box intersection algorithm as closed or as half-open boxes, see also Section 37.2. Closed boxes support zero-width boxes and they can intersect at their boundaries, while half-open boxes always have a positive volume and they only intersect iff their interiors overlap. The choice between closed or half-open boxes is selected with the *topology* parameter and its two values:

- *CGAL::Box_intersection_d::HALF_OPEN* and
- *CGAL::Box_intersection_d::CLOSED*.

The example program uses a two-dimensional box with *int* coordinates and *id*-numbers that are by default explicitly stored. We create the same boxes as in the minimal example in Section 37.4. We create a 3×3 grid of *boxes*, and two boxes for the *query* sequence, namely the box at the center and the box from the upper-right corner of the grid.

We write a more involved callback function object *Report* that stores an output iterator and writes the *id*-number of the box in the first argument to the output iterator. We also provide a small helper function *report* that simplifies the use of the function object.

We call the box intersection algorithm twice; once for the default *topology*, which is the closed box topology, and once for the half-open box topology. We sort the resulting output for better readability and verify its correctness with the *check1* and *check2* data. For the closed box topology, the center box in *query* intersects all *boxes*, and the upper-right box in *query* intersects the four boxes of the upper-right quadrant in *boxes*. Almost all intersections are with the box boundaries, thus, for the half-open topology only one intersection remains per *query* box, namely its corresponding box in *boxes*. So, the output of the algorithm will be:

```
0 1 2 3 4 4 5 5 6 7 7 8 8
4 8
```

For the second box intersection function call we have to specify the *cutoff* parameter explicitly. See the Section 37.8 below for a detailed discussion.

```
// file: examples/Box_intersection_d/box_grid.C
#include <CGAL/box_intersection_d.h>
#include <vector>
#include <algorithm>
#include <iterator>
#include <assert.h>

typedef CGAL::Box_intersection_d::Box_d<int,2> Box;

// coordinates for 9 boxes of a grid
int p[9*4] = { 0,0,1,1, 1,0,2,1, 2,0,3,1, // lower
               0,1,1,2, 1,1,2,2, 2,1,3,2, // middle
               0,2,1,3, 1,2,2,3, 2,2,3,3}; // upper

// 9 boxes
Box boxes[9] = { Box( p,    p+ 2), Box( p+ 4, p+ 6), Box( p+ 8, p+10),
                  Box( p+12, p+14), Box( p+16, p+18), Box( p+20, p+22),
                  Box( p+24, p+26), Box( p+28, p+30), Box( p+32, p+34)};

// 2 selected boxes as query; center and upper right
Box query[2] = { Box( p+16, p+18), Box( p+32, p+34)};

// callback function object writing results to an output iterator
template <class OutputIterator>
struct Report {
    OutputIterator it;
    Report( OutputIterator i) : it(i) {} // store iterator in object
    // We write the id-number of box a to the output iterator assuming
    // that box b (the query box) is not interesting in the result.
    void operator()( const Box& a, const Box&) { *it++ = a.id(); }
};

template <class Iter> // helper function to create the function object
Report<Iter> report( Iter it) { return Report<Iter>(it); }
```

```

int main() {
    // run the intersection algorithm and store results in a vector
    std::vector<std::size_t> result;
    CGAL::box_intersection_d( boxes, boxes+9, query, query+2,
                             report( std::back_inserter( result)));
    // sort, check, and show result
    std::sort( result.begin(), result.end());
    std::size_t check1[13] = {0,1,2,3,4,4,5,5,6,7,7,8,8};
    assert(result.size() == 13 && std::equal(check1, check1+13, result.begin()));
    std::copy( result.begin(), result.end(),
               std::ostream_iterator<std::size_t>( std::cout, " "));
    std::cout << std::endl;

    // run it again but for different cutoff value and half-open boxes
    result.clear();
    CGAL::box_intersection_d( boxes, boxes+9, query, query+2,
                             report( std::back_inserter( result)),
                             std::ptrdiff_t(1),
                             CGAL::Box_intersection_d::HALF_OPEN);
    // sort, check, and show result
    std::sort( result.begin(), result.end());
    std::size_t check2[2] = {4,8};
    assert(result.size() == 2 && std::equal(check2, check2+2, result.begin()));
    std::copy( result.begin(), result.end(),
               std::ostream_iterator<std::size_t>( std::cout, " "));
    std::cout << std::endl;
    return 0;
}

```

37.8 Runtime Performance

The implemented algorithm is described in [ZE02] as version two. Its performance depends on a *cutoff* parameter. When the size of both iterator ranges drops below the *cutoff* parameter the function switches from the streamed segment-tree algorithm to the two-way-scan algorithm, see [ZE02] for the details.

The streamed segment-tree algorithm needs $O(n \log^d(n) + k)$ worst-case running time and $O(n)$ space, where n is the number of boxes in both input sequences, d the (constant) dimension of the boxes, and k the output complexity, i.e., the number of pairwise intersections of the boxes. The two-way-scan algorithm needs $O(n \log(n) + l)$ worst-case running time and $O(n)$ space, where l is the number of pairwise overlapping intervals in one dimensions (the dimension where the algorithm is used instead of the segment tree). Note that l is not necessarily related to k and using the two-way-scan algorithm is a heuristic.

Unfortunately, we have no general method to automatically determine an optimal cutoff parameter, since it depends on the used hardware, the runtime ratio between callback runtime and segment-tree runtime, and of course the number of boxes to be checked and their distribution. In cases where the callback runtime is dominant, it may be best to make the threshold parameter small. Otherwise a *cutoff* = \sqrt{n} can lead to acceptable results. For well distributed boxes the original paper [ZE02] gives optimal cutoffs in the thousands. Anyway, for optimal runtime some experiments to compare different cutoff parameters are recommended.

To demonstrate that box intersection can be done quite fast, different box sequences are intersected in the range between 4 and 800000 boxes total. We use three-dimensional default boxes of closed topology with *float*

coordinates and without additional data fields. The algorithm works directly on the boxes, not on pointer to boxes. Each box intersection is reported to an empty dummy callback.

For each box set, a near-optimal cutoff parameter is determined using an adaptive approximation. The runtime required for streaming is compared against usual scanning. Results on a Xeon 2.4GHz with 4GB main memory can be seen in Figure 37.1. For a small number of boxes, pure scanning is still faster than streaming with optimal cutoff, which would just delegate the box sets to the scanning algorithm. As there are more and more boxes, the overhead becomes less important.

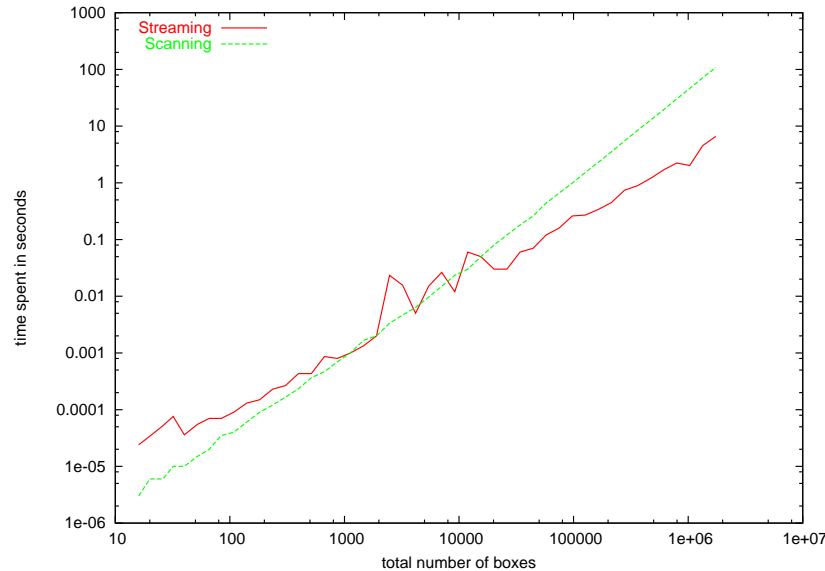


Figure 37.1: Runtime comparison between the scanning and the streaming algorithm.

37.9 Example Using a Custom Box Implementation

The example in the previous Section 37.7 uses an array to provide the coordinates and then creates another array for the boxes. In the following example we write our own box class *Box* that we can initialize directly with the four coordinates and create the array of boxes directly. We also omit the explicitly stored *id*-number and use the address of the box itself as *id*-number. This works only if the boxes do not change their position, i.e., we work with pointers to the boxes in the intersection algorithm.

We follow with our own box class *Box* the *BoxIntersectionBox_d* concept, which allows us to reuse the default traits implementation, i.e., we can use the same default function call to compute all intersections. See the example in the next section for a self-written traits class. So, in principle, the remainder of the example stays the same and we omit the part from the previous example for brevity that illustrates the half-open box topology.

The requirements for the box implementation are best studied on page 2171 in the Reference Manual. In a nutshell, we have to define the type *NT* for the box coordinates and the type *ID* for the *id*-number. Member functions give access to the coordinates and the *id*-number. A static member function returns the dimension.

```
// file: examples/Box_intersection_d/custom_box_grid.C
#include <CGAL/box_intersection_d.h>
#include <vector>
#include <algorithm>
```

```

#include <iterator>
#include <assert.h>

struct Box {
    typedef int          NT;
    typedef std::ptrdiff_t ID;
    int x[2], y[2];
    Box( int x0, int x1, int y0, int y1) { x[0]=x0; x[1]=x1; y[0]=y0; y[1]=y1;}
    static int dimension() { return 2; }
    int min_coord(int dim) const { return x[dim]; }
    int max_coord(int dim) const { return y[dim]; }
    // id-function using address of current box,
    // requires to work with pointers to boxes later
    std::ptrdiff_t id() const { return (std::ptrdiff_t)(this); }
};

// 9 boxes of a grid
Box boxes[9] = { Box( 0,0,1,1), Box( 1,0,2,1), Box( 2,0,3,1), // low
                 Box( 0,1,1,2), Box( 1,1,2,2), Box( 2,1,3,2), // middle
                 Box( 0,2,1,3), Box( 1,2,2,3), Box( 2,2,3,3)}; // upper
// 2 selected boxes as query; center and upper right
Box query[2] = { Box( 1,1,2,2), Box( 2,2,3,3)};

// With the special id-function we need to work on box pointers
Box* b_ptr[9] = { boxes, boxes+1, boxes+2, boxes+3, boxes+4, boxes+5,
                 boxes+6, boxes+7, boxes+8};
Box* q_ptr[2] = { query, query+1};

// callback function object writing results to an output iterator
template <class OutputIterator>
struct Report {
    OutputIterator it;
    Report( OutputIterator i) : it(i) {} // store iterator in object
    // We write the position with respect to 'boxes' to the output iterator
    // assuming that box b (the query box) is not interesting in the result.
    void operator()( const Box* a, const Box*) {
        *it++ = ( reinterpret_cast<Box*>(a->id()) - boxes);
    }
};

template <class Iter> // helper function to create the function object
Report<Iter> report( Iter it) { return Report<Iter>(it); }

int main() {
    // run the intersection algorithm and store results in a vector
    std::vector<std::size_t> result;
    CGAL::box_intersection_d( b_ptr, b_ptr+9, q_ptr, q_ptr+2,
                             report( std::back_inserter( result)),
                             std::ptrdiff_t(0));
    // sort and check result
    std::sort( result.begin(), result.end());
    std::size_t chk[13] = {0,1,2,3,4,4,5,5,6,7,7,8,8};
    assert( result.size()==13 && std::equal(chk,chk+13,result.begin()));
    return 0;
}

```

37.10 Example for Point Proximity Search with a Custom Traits Class

Given a set of 3D points, we want to find all pairs of points that are less than a certain distance apart. We use the box intersection algorithm to find good candidates, namely those that are less than this specified distance apart in the L_∞ norm, which is a good approximation of the Euclidean norm.

We use an unusual representation for the box, namely pointers to the 3D points themselves. We implement a special box traits class that interprets the point as a box of the dimensions $[-eps, +eps]^3$ centered at this point. The value for eps is half the specified distance from above, i.e., points are reported if their distance is smaller than $2*eps$.

The requirements for the box traits class are best studied on page 2172 in the Reference Manual. In a nutshell, we have to define the type *NT* for the box coordinates, the type *ID* for the *id*-number, and the type *Box_parameter* similar to the box handle, here *Point_3** since we work with the pointers. All member functions in the traits class are static. Two functions give access to the max and min coordinates that we compute from the point coordinates plus or minus the eps value, respectively. For the *id*-number function the address of the point itself is sufficient, since the points stay stable. Another function returns the dimension.

The *report* callback function computes than the Euclidean distance and prints a message for points that are close enough.

Note that we need to reserve sufficient space in the *points* vector to avoid reallocations while we create the *points* vector and the *boxes* vector in parallel, since otherwise the *points* vector might reallocate and invalidate all pointers stored in the *boxes* so far.

```
// file: examples/Box_intersection_d/proximity_custom_box_traits.C
#include <CGAL/Simple_cartesian.h>
#include <CGAL/box_intersection_d.h>
#include <CGAL/point_generators_3.h>
#include <CGAL/copy_n.h>
#include <vector>
#include <algorithm>
#include <iterator>
#include <cmath>

typedef CGAL::Simple_cartesian<float>          Kernel;
typedef Kernel::Point_3                        Point_3;
typedef CGAL::Random_points_on_sphere_3<Point_3> Points_on_sphere;

std::vector<Point_3> points;
std::vector<Point_3*> boxes;    // boxes are just pointers to points
const float          eps = 0.1; // finds point pairs of distance < 2*eps

// Boxes are just pointers to 3d points. The traits class adds the
// +- eps size to each interval around the point, effectively building
// on the fly a box of size 2*eps centered at the point.
struct Traits {
    typedef float          NT;
    typedef Point_3*       Box_parameter;
    typedef std::ptrdiff_t ID;

    static int  dimension() { return 3; }
    static float coord( Box_parameter b, int d) {
        return (d == 0) ? b->x() : ((d == 1) ? b->y() : b->z());
    }
};
```

```

    }
    static float min_coord( Box_parameter b, int d) { return coord(b,d)-eps;}
    static float max_coord( Box_parameter b, int d) { return coord(b,d)+eps;}
    // id-function using address of current box,
    // requires to work with pointers to boxes later
    static std::ptrdiff_t id(Box_parameter b) { return (std::ptrdiff_t)(b); }
};

// callback function reports pairs in close proximity
void report( const Point_3* a, const Point_3* b) {
    float dist = std::sqrt( CGAL::squared_distance( *a, *b));
    if ( dist < 2*eps) {
        std::cout << "Point " << (a - &(points.front())) << " and Point "
            << (b - &(points.front())) << " have distance " << dist
            << "." << std::endl;
    }
}

int main() {
    // create some random points on the sphere of radius 1.0
    Points_on_sphere generator( 1.0);
    points.reserve( 50);
    for ( int i = 0; i != 50; ++i) {
        points.push_back( *generator++);
        boxes.push_back( & points.back());
    }

    // run the intersection algorithm and report proximity pairs
    CGAL::box_self_intersection_d( boxes.begin(), boxes.end(),
        report, Traits());

    return 0;
}

```

37.11 Design and Implementation History

Lutz Kettner and Andreas Meyer implemented the algorithms starting from the publication [ZE02]. We had access to the original C implementation of Afra Zomorodian, which helped clarifying some questions, and we are grateful to the help of Afra Zomorodian in answering our questions during his visit. We thank Steve Robbins for an excellent review for this package. Steve Robbins provided an independent and earlier implementation of this algorithm, however, we learned too late about this implementation.

Intersecting Sequences of dD Iso-oriented Boxes

Reference Manual

Lutz Kettner, Andreas Meyer, and Afra Zomorodian

We provide an efficient algorithm [ZE02] for finding all intersecting pairs for large numbers of iso-oriented boxes, i.e., typically these will be bounding boxes of more complicated geometries. For comparison and as a base case for other methods, we also offer the simple all-pairs test as a generic function.

37.12 Classified Reference Pages

Concepts

BoxIntersectionBox_d page [2171](#)
BoxIntersectionTraits_d page [2172](#)

Classes

CGAL::Box_intersection_d::Box_d<NT,int D,IdPolicy> page [2174](#)
CGAL::Box_intersection_d::Box_with_handle_d<NT, int D, Handle, IdPolicy> page [2178](#)
CGAL::Box_intersection_d::Box_traits_d<BoxHandle> page [2177](#)

Functions

CGAL::box_intersection_d page [2161](#)
CGAL::box_self_intersection_d page [2166](#)

CGAL::box_intersection_all_pairs_d page [2164](#)
CGAL::box_self_intersection_all_pairs_d page [2169](#)

37.13 Alphabetical List of Reference Pages

<i>BoxIntersectionBox_d</i>	page 2171
<i>BoxIntersectionTraits_d</i>	page 2172
<i>Box_d</i> <NT,int D,IdPolicy>	page 2174
<i>box_intersection_all_pairs_d</i>	page 2164
<i>box_intersection_d</i>	page 2161
<i>box_self_intersection_all_pairs_d</i>	page 2169
<i>box_self_intersection_d</i>	page 2166
<i>Box_traits_d</i> <BoxHandle>	page 2177
<i>Box_with_handle_d</i> <NT, int D, Handle, IdPolicy>	page 2178

CGAL::box_intersection_d

Definition

The function *box_intersection_d* computes the pairwise intersecting boxes between two sequences of iso-oriented boxes in arbitrary dimension. The sequences of boxes are given with two random-access iterator ranges and will be reordered in the course of the algorithm. For each intersecting pair of boxes a *callback* function object is called with the two intersecting boxes as argument; the first argument is a box from the first sequence, the second argument a box from the second sequence. The performance of the algorithm can be tuned with a *cutoff* parameter, see the implementation section below for more details.

The algorithm reorders the boxes in the course of the algorithm. Now, depending on the size of a box it can be faster to copy the boxes, or to work with pointers to boxes and copy only pointers. We offer automatic support for both options. To simplify the description, let us call the *value_type* of the iterators *box handle*. The *box handle* can either be our box type itself or a pointer (or const pointer) to the box type.

A *d*-dimensional iso-oriented box is defined as the Cartesian product of *d* intervals. We call the box *half-open* if the *d* intervals $\{[lo_i, hi_i] \mid 0 \leq i < d\}$ are half-open intervals, and we call the box *closed* if the *d* intervals $\{[lo_i, hi_i] \mid 0 \leq i < d\}$ are closed intervals. Note that closed boxes support zero-width boxes and they can intersect at their boundaries, while non-empty half-open boxes always have a positive volume and they only intersect if their interiors overlap. The distinction between closed or half-open boxes does not require a different representation of boxes, just a different interpretation when comparing boxes, which is selected with the *topology* parameter and its two values, *CGAL::Box_intersection_d::HALF_OPEN* and *CGAL::Box_intersection_d::CLOSED*.

In addition, a box has an unique *id*-number. It is used to order boxes consistently in each dimension even if boxes have identical coordinates. In consequence, the algorithm guarantees that a pair of intersecting boxes is reported only once. Boxes with equal *id*-number are not reported since they obviously intersect trivially.

The algorithm uses a traits class of the *BoxIntersectionTraits_d* concept to access the boxes. A default traits class is provided that assumes that the box type is a model of the *BoxIntersectionBox_d* concept and that the box handle, i.e., the iterators value type, is identical to the box type or a pointer to the box type.

An important special application of this algorithm is the test for self-intersections where the second box sequence is an identical copy of the first sequence including the preserved *id*-number. Note that this implies that the address of the box is not sufficient for the *id*-number if boxes are copied by value. To ease the use of this special case we offer a simplified version of this function with one iterator range only, which then creates internally the second copy of the boxes, under the name *CGAL::box_self_intersection_d*.

In the general case, we distinguish between the bipartite case (the boxes are from different sequences) and the complete case (the boxes are from the same sequence, i.e., the self intersection case). The default is the bipartite case, since the complete case is typically handled with the simplified function call mentioned above. However, the general function call offers the *setting* parameter with the values *CGAL::Box_intersection_d::COMPLETE* and *CGAL::Box_intersection_d::BIPARTITE*.

```
#include <CGAL/box_intersection_d.h>
```

```
template< class RandomAccessIterator1, class RandomAccessIterator2, class Callback >
void box_intersection_d( RandomAccessIterator1 begin1,
```

```

RandomAccessIterator1 end1,
RandomAccessIterator2 begin2,
RandomAccessIterator2 end2,
Callback callback,
std::ptrdiff_t cutoff = 10,
Box_intersection_d::Topology topology = Box_intersection_d::CLOSED,
Box_intersection_d::Setting setting = Box_intersection_d::BIPARTITE)

```

Invocation of box intersection with default box traits *CGAL::Box_intersection_d::Box_traits_d*<*Box_handle*>, where *Box_handle* corresponds to the iterator value type of *RandomAccessIterator1*.

```

template< class RandomAccessIterator1, class RandomAccessIterator2, class Callback, class BoxTraits >
void box_intersection_d( RandomAccessIterator1 begin1,
                        RandomAccessIterator1 end1,
                        RandomAccessIterator2 begin2,
                        RandomAccessIterator2 end2,
                        Callback callback,
                        BoxTraits box_traits,
                        std::ptrdiff_t cutoff = 10,
                        Box_intersection_d::Topology topology = Box_intersection_d::CLOSED,
                        Box_intersection_d::Setting setting = Box_intersection_d::BIPARTITE)

```

Invocation with custom box traits.

Requirements

- *RandomAccessIterator1*, and ...2, must be mutable random-access iterators and both value types must be the same. We call this value type *Box_handle* in the following.
- *Callback* must be of the *BinaryFunction* concept. The *Box_handle* must be convertible to both argument types. The return type is not used and can be *void*.
- The *Box_handle* must be a model of the *Assignable* concept.
- In addition, if the default box traits is used the *Box_handle* must be a class type *T* or a pointer to a class type *T*, where *T* must be a model of the *BoxIntersectionBox_d* concept. In both cases, the default box traits specializes to a suitable implementation.
- *BoxTraits* must be of the *BoxIntersectionTraits_d* concept.

See Also

CGAL::box_self_intersection_d page [2166](#)
CGAL::box_intersection_all_pairs_d page [2164](#)

CGAL::Box_intersection_d::Box_traits_d<*BoxHandle*> page [2177](#)
BoxIntersectionBox_d page [2171](#)
BoxIntersectionTraits_d page [2172](#)

Implementation

The implemented algorithm is described in [ZE02] as version two. Its performance depends on a *cutoff* parameter. When the size of both iterator ranges drops below the *cutoff* parameter the function switches from the streamed segment-tree algorithm to the two-way-scan algorithm, see [ZE02] for the details.

The streamed segment-tree algorithm needs $O(n \log^d(n) + k)$ worst-case running time and $O(n)$ space, where n is the number of boxes in both input sequences, d the (constant) dimension of the boxes, and k the output complexity, i.e., the number of pairwise intersections of the boxes. The two-way-scan algorithm needs $O(n \log(n) + l)$ worst-case running time and $O(n)$ space, where l is the number of pairwise overlapping intervals in one dimensions (the dimension where the algorithm is used instead of the segment tree). Note that l is not necessarily related to k and using the two-way-scan algorithm is a heuristic.

Unfortunately, we have no general method to automatically determine an optimal cutoff parameter, since it depends on the used hardware, the runtime ratio between callback runtime and segment-tree runtime, and of course the number of boxes to be checked and their distribution. In cases where the callback runtime is dominant, it may be best to make the threshold parameter small. Otherwise a *cutoff* = \sqrt{n} can lead to acceptable results. For well distributed boxes the original paper [ZE02] gives optimal cutoffs in the thousands. Anyway, for optimal runtime some experiments to compare different cutoff parameters are recommended. See also Section 37.8.

Example

The box implementation provided with `CGAL::Box_intersection_d::Box_d<double,2>` has a special constructor for the CGAL bounding box type `CGAL::Bbox_2` (and similar for dimension 3). We use this in the example to create 3×3 boxes in a grid layout. Additionally we pick the center box and the box in the upper-right corner as our second box sequence *query*.

The default policy of the box type implements the *id*-number with an explicit counter in the boxes, which is the default choice since it always works. We use the *id*-number in our callback function to report the result of the intersection algorithm call. The result will be that the first *query* box intersects all nine boxes and the second *query* box intersects the four boxes in the upper-right quadrant.

```
// file: examples/Box_intersection_d/minimal.C
#include <CGAL/box_intersection_d.h>
#include <CGAL/Bbox_2.h>
#include <iostream>

typedef CGAL::Box_intersection_d::Box_d<double,2> Box;
typedef CGAL::Bbox_2 Bbox;

// 9 boxes of a grid
Box boxes[9] = { Bbox( 0,0,1,1), Bbox( 1,0,2,1), Bbox( 2,0,3,1), // low
                 Bbox( 0,1,1,2), Bbox( 1,1,2,2), Bbox( 2,1,3,2), // middle
                 Bbox( 0,2,1,3), Bbox( 1,2,2,3), Bbox( 2,2,3,3)}; // upper
// 2 selected boxes as query; center and upper right
Box query[2] = { Bbox( 1,1,2,2), Bbox( 2,2,3,3)};

void callback( const Box& a, const Box& b ) {
    std::cout << "box " << a.id() << " intersects box " << b.id() << std::endl;
}

int main() {
    CGAL::box_intersection_d( boxes, boxes+9, query, query+2, callback);
    return 0;
}
```

CGAL::box_intersection_all_pairs_d

Definition

The function `box_intersection_all_pairs_d` computes the pairwise intersecting boxes between two sequences of iso-oriented boxes in arbitrary dimension. It does so by comparing all possible pairs of boxes and is thus inferior to the fast `CGAL::box_intersection_d` algorithm on page 2161.

The sequences of boxes are given with two forward iterator ranges. The sequences are not modified. For each intersecting pair of boxes a *callback* function object is called with the two intersecting boxes as argument; the first argument is a box from the first sequence, the second argument a box from the second sequence.

The algorithm is interface compatible with the `CGAL::box_intersection_d` function. Similarly, we call the *value_type* of the iterators the *box handle*, which is either our box type or a pointer type to our box type.

A d -dimensional iso-oriented box is defined as the Cartesian product of d intervals. We call the box *half-open* if the d intervals $\{[lo_i, hi_i) \mid 0 \leq i < d\}$ are half-open intervals, and we call the box *closed* if the d intervals $\{[lo_i, hi_i] \mid 0 \leq i < d\}$ are closed intervals. Note that closed boxes support zero-width boxes and they can intersect at their boundaries, while non-empty half-open boxes always have a positive volume and they only intersect iff their interiors overlap. The distinction between closed or half-open boxes does not require a different representation of boxes, just a different interpretation when comparing boxes, which is selected with the *topology* parameter and its two values, `CGAL::Box_intersection_d::HALF_OPEN` and `CGAL::Box_intersection_d::CLOSED`.

In addition, a box has an unique *id*-number. Boxes with equal *id*-number are not reported since they obviously intersect trivially.

The algorithm uses a traits class of the `BoxIntersectionTraits_d` concept to access the boxes. A default traits class is provided that assumes that the box type is a model of the `BoxIntersectionBox_d` concept and that the box handle, i.e., the iterators value type, is identical to the box type or a pointer to the box type.

An important special application of this algorithm is the test for self-intersections where the second box sequence is an identical copy of the first sequence including the preserved *id*-number. We offer a specialized implementation `CGAL::box_self_intersection_all_pairs` for this application.

```
#include <CGAL/box_intersection_d.h>
```

```
template< class ForwardIterator1, class ForwardIterator2, class Callback >
void    box_intersection_all_pairs_d( ForwardIterator1 begin1,
                                     ForwardIterator1 end1,
                                     ForwardIterator2 begin2,
                                     ForwardIterator2 end2,
                                     Callback callback,
                                     Box_intersection_d::Topology topology = Box_intersection_d::CLOSED)
```

Invocation of box intersection with default box traits `CGAL::Box_intersection_d::Box_traits_d<Box_handle>`, where *Box_handle* corresponds to the iterator value type of `ForwardIterator1`.

```
template< class ForwardIterator1, class ForwardIterator2, class Callback, class BoxTraits >
void    box_intersection_all_pairs_d( ForwardIterator1 begin1,
```

```

ForwardIterator1 end1,
ForwardIterator2 begin2,
ForwardIterator2 end2,
Callback callback,
BoxTraits box_traits,
Box_intersection_d::Topology topology = Box_intersection_d::CLOSED)

```

Invocation with custom box traits.

Requirements

- *ForwardIterator1*, and ...2, must be forward iterators and both value types must be the same. We call this value type *Box_handle* in the following.
- *Callback* must be of the *BinaryFunction* concept. The *Box_handle* must be convertible to both argument types. The return type is not used and can be *void*.
- The *Box_handle* must be a model of the *Assignable* concept.
- In addition, if the default box traits is used the *Box_handle* must be a class type *T* or a pointer to a class type *T*, where *T* must be a model of the *BoxIntersectionBox_d* concept. In both cases, the default box traits specializes to a suitable implementation.
- *BoxTraits* must be of the *BoxIntersectionTraits_d* concept.

See Also

<i>CGAL::box_intersection_d</i>	page 2161
<i>CGAL::box_self_intersection_d</i>	page 2166
<i>CGAL::box_self_intersection_all_pairs_d</i>	page 2169
<i>CGAL::Box_intersection_d::Box_traits_d<BoxHandle></i>	page 2177
<i>BoxIntersectionBox_d</i>	page 2171
<i>BoxIntersectionTraits_d</i>	page 2172

Implementation

The algorithm is trivially testing all pairs and runs therefore in time $O(nm)$ where n is the size of the first sequence and m is the size of the second sequence.

CGAL::box_self_intersection_d

Definition

The function *box_self_intersection_d* computes the pairwise intersecting boxes in a sequence of iso-oriented boxes in arbitrary dimension. The sequence of boxes is given with as a random-access iterator range and will be reordered in the course of the algorithm. For each intersecting pair of boxes a *callback* function object is called with the two intersecting boxes as argument; the first argument is a box from the sequence, the second argument is a copy of a box from the sequence. The performance of the algorithm can be tuned with a *cutoff* parameter, see the implementation section of the *CGAL::box_intersection_d* function on page 2161.

The algorithm creates a second copy of the boxes and reorders the boxes in the course of the algorithm. Now, depending on the size of a box it can be faster to copy the boxes, or to work with pointers to boxes and copy only pointers. We offer automatic support for both options. To simplify the description, let us call the *value_type* of the iterators *box handle*. The *box handle* can either be our box type itself or a pointer (or const pointer) to the box type.

A d -dimensional iso-oriented box is defined as the Cartesian product of d intervals. We call the box *half-open* if the d intervals $\{[lo_i, hi_i) \mid 0 \leq i < d\}$ are half-open intervals, and we call the box *closed* if the d intervals $\{[lo_i, hi_i] \mid 0 \leq i < d\}$ are closed intervals. Note that closed boxes support zero-width boxes and they can intersect at their boundaries, while non-empty half-open boxes always have a positive volume and they only intersect iff their interiors overlap. The distinction between closed or half-open boxes does not require a different representation of boxes, just a different interpretation when comparing boxes, which is selected with the *topology* parameter and its two values, *CGAL::Box_intersection_d::HALF_OPEN* and *CGAL::Box_intersection_d::CLOSED*.

In addition, a box has an unique *id*-number. It is used to order boxes consistently in each dimension even if boxes have identical coordinates. In consequence, the algorithm guarantees that a pair of intersecting boxes is reported only once. This self-intersection function creates internally a second copy of the box sequence. The copying has to preserve the *id*-number of boxes. Note that this implies that the address of the box is not sufficient for the *id*-number if boxes are copied by value. Boxes of equal *id*-number are not reported as intersecting pairs since they are always intersecting trivially.

The algorithm uses a traits class of the *BoxIntersectionTraits_d* concept to access the boxes. A default traits class is provided that assumes that the box type is a model of the *BoxIntersectionBox_d* concept and that the box handle, i.e., the iterators value type, is identical to the box type or a pointer to the box type.

```
#include <CGAL/box_intersection_d.h>
```

```
template< class RandomAccessIterator, class Callback >
void    box_self_intersection_d( RandomAccessIterator begin,
                               RandomAccessIterator end,
                               Callback callback,
                               std::ptrdiff_t cutoff = 10,
                               Box_intersection_d::Topology topology = Box_intersection_d::CLOSED)
```

Invocation of box intersection with default box traits *CGAL::Box_intersection_d::Box_traits_d<Box_handle>*, where *Box_handle* corresponds to the iterator value type of *RandomAccessIterator*.

```
template< class RandomAccessIterator, class Callback, class BoxTraits >
void    box_self_intersection_d( RandomAccessIterator begin,
```

```

RandomAccessIterator end,
Callback callback,
BoxTraits box_traits,
std::ptrdiff_t cutoff = 10,
Box_intersection_d::Topology topology = Box_intersection_d::CLOSED)

```

Invocation with custom box traits.

Requirements

- *RandomAccessIterator* must be a mutable random-access iterator. We call its value type *Box_handle* in the following.
- *Callback* must be of the *BinaryFunction* concept. The *Box_handle* must be convertible to both argument types. The return type is not used and can be *void*.
- The *Box_handle* must be a model of the *Assignable* concept.
- In addition, if the default box traits is used the *Box_handle* must be a class type *T* or a pointer to a class type *T*, where *T* must be a model of the *BoxIntersectionBox_d* concept. In both cases, the default box traits specializes to a suitable implementation.
- *BoxTraits* must be of the *BoxIntersectionTraits_d* concept.

See Also

CGAL::box_intersection_d [page 2161](#)
CGAL::box_self_intersection_all_pairs_d [page 2169](#)
CGAL::Box_intersection_d::Box_traits_d<BoxHandle> [page 2177](#)
BoxIntersectionBox_d [page 2171](#)
BoxIntersectionTraits_d [page 2172](#)

Implementation

See the implementation section of the *CGAL::box_intersection_d* function on [page 2161](#).

Example

The box implementation provided with *CGAL::Box_intersection_d::Box_d<double,2>* has a special constructor for the CGAL bounding box type *CGAL::Bbox_2* (and similar for dimension 3). We use this in the example to create 3×3 boxes in a grid layout.

The default policy of the box type implements the *id*-number with an explicit counter in the boxes, which is the default choice since it always works. We use the *id*-number in our callback function to report the result of the intersection algorithm call. The result will be 20 pairwise intersections, but the order in which they are reported is non-intuitive.

```

// file: examples/Box_intersection_d/minimal_self.C
#include <CGAL/box_intersection_d.h>
#include <CGAL/Bbox_2.h>
#include <iostream>

typedef CGAL::Box_intersection_d::Box_d<double,2> Box;
typedef CGAL::Bbox_2                               Bbox;

// 9 boxes of a grid
Box boxes[9] = { Bbox( 0,0,1,1), Bbox( 1,0,2,1), Bbox( 2,0,3,1), // low
                 Bbox( 0,1,1,2), Bbox( 1,1,2,2), Bbox( 2,1,3,2), // middle
                 Bbox( 0,2,1,3), Bbox( 1,2,2,3), Bbox( 2,2,3,3)}; // upper

void callback( const Box& a, const Box& b ) {
    std::cout << "box " << a.id() << " intersects box " << b.id() << std::endl;
}

int main() {
    CGAL::box_self_intersection_d( boxes, boxes+9, callback);
    return 0;
}

```


CGAL::box_self_intersection_all_pairs_d

Definition

The function `box_self_intersection_all_pairs_d` computes the pairwise intersecting boxes in a sequence of iso-oriented boxes in arbitrary dimension. It does so by comparing all possible pairs of boxes and is thus inferior to the fast `CGAL::box_self_intersection_d` algorithm on page 2166.

The sequence of boxes is given with a forward iterator range. The sequences are not modified. For each intersecting pair of boxes a *callback* function object is called with the two intersecting boxes as argument.

The algorithm is interface compatible with the `CGAL::box_self_intersection_d` function. Similarly, we call the *value_type* of the iterators the *box handle*, which is either our box type or a pointer type to our box type.

A d -dimensional iso-oriented box is defined as the Cartesian product of d intervals. We call the box *half-open* if the d intervals $\{[lo_i, hi_i) \mid 0 \leq i < d\}$ are half-open intervals, and we call the box *closed* if the d intervals $\{[lo_i, hi_i] \mid 0 \leq i < d\}$ are closed intervals. Note that closed boxes support zero-width boxes and they can intersect at their boundaries, while non-empty half-open boxes always have a positive volume and they only intersect iff their interiors overlap. The distinction between closed or half-open boxes does not require a different representation of boxes, just a different interpretation when comparing boxes, which is selected with the *topology* parameter and its two values, `CGAL::Box_intersection_d::HALF_OPEN` and `CGAL::Box_intersection_d::CLOSED`.

The algorithm uses a traits class of the `BoxIntersectionTraits_d` concept to access the boxes. A default traits class is provided that assumes that the box type is a model of the `BoxIntersectionBox_d` concept and that the box handle, i.e., the iterators value type, is identical to the box type or a pointer to the box type.

```
#include <CGAL/box_intersection_d.h>
```

```
template< class ForwardIterator, class Callback >
void box_self_intersection_all_pairs_d(
    ForwardIterator begin,
    ForwardIterator end,
    Callback callback,
    Box_intersection_d::Topology topology = Box_intersection_d::CLOSED)
```

Invocation of box intersection with default box traits `CGAL::Box_intersection_d::Box_traits_d<Box_handle>`, where *Box_handle* corresponds to the iterator value type of *ForwardIterator*.

```
template< class ForwardIterator, class Callback, class BoxTraits >
void box_self_intersection_all_pairs_d(
    ForwardIterator begin,
    ForwardIterator end,
    Callback callback,
    BoxTraits box_traits,
    Box_intersection_d::Topology topology = Box_intersection_d::CLOSED)
```

Invocation with custom box traits.

Requirements

- *ForwardIterator* must be a forward iterator. We call its value type *Box_handle* in the following.
- *Callback* must be of the *BinaryFunction* concept. The *Box_handle* must be convertible to both argument types. The return type is not used and can be *void*.
- The *Box_handle* must be a model of the *Assignable* concept.
- In addition, if the default box traits is used the *Box_handle* must be a class type *T* or a pointer to a class type *T*, where *T* must be a model of the *BoxIntersectionBox_d* concept. In both cases, the default box traits specializes to a suitable implementation.
- *BoxTraits* must be of the *BoxIntersectionTraits_d* concept.

See Also

<i>CGAL::box_intersection_d</i>	page 2161
<i>CGAL::box_self_intersection_d</i>	page 2166
<i>CGAL::box_intersection_all_pairs_d</i>	page 2164
<i>CGAL::Box_intersection_d::Box_traits_d<BoxHandle></i>	page 2177
<i>BoxIntersectionBox_d</i>	page 2171
<i>BoxIntersectionTraits_d</i>	page 2172

Implementation

The algorithm is trivially testing all pairs and runs therefore in time $O(n^2)$ where n is the size of the input sequence. This algorithm does not use the id-number of the boxes.

BoxIntersectionBox_d

Definition

The `BoxIntersectionBox_d` concept is used in the context of the intersection algorithms for sequences of iso-oriented boxes. These algorithms come with a default traits class that assumes that the boxes are a model of this `BoxIntersectionBox_d` concept. This concept defines the access functions to the dimension, the *id*-number, and the boundaries of the box.

Refines

Assignable.

Has Models

`CGAL::Box_intersection_d::Box_d<NT,int D,IdPolicy>` page 2174
`CGAL::Box_intersection_d::Box_with_handle_d<NT, int D, Handle, IdPolicy>` page 2178

Types

`BoxIntersectionBox_d::NT` number type to represent the box boundaries. Allowed are the builtin types *int*, *unsigned int*, *float*, and *double*.

`BoxIntersectionBox_d::ID` type for the box *id*-number, must be a model of the *LessThanComparable* concept.

Access Functions

<i>int</i>	<code>BoxIntersectionBox_d::dimension()</code>	returns the dimension of the box.
<i>ID</i>	<code>box.id()</code>	returns the unique <i>id</i> -number for the <i>box</i> .
<i>NT</i>	<code>box.min_coord(int d)</code>	returns the lower boundary in dimension <i>d</i> , $0 \leq d < \text{dimension}()$.
<i>NT</i>	<code>box.max_coord(int d)</code>	returns the upper boundary in dimension <i>d</i> , $0 \leq d < \text{dimension}()$.

See Also

`CGAL::box_intersection_d` page 2161
`CGAL::box_self_intersection_d` page 2166
`CGAL::box_intersection_all_pairs_d` page 2164
`CGAL::box_self_intersection_all_pairs_d` page 2169

`CGAL::Box_intersection_d::Box_traits_d<BoxHandle>` page 2177
`BoxIntersectionTraits_d` page 2172

BoxIntersectionTraits_d

Definition

The `BoxIntersectionTraits_d` concept is used for the intersection algorithms for sequences of iso-oriented boxes. This concept defines the access functions to the dimension, the *id*-number, and the boundaries of the boxes manipulated in these algorithms.

Refines

Assignable.

Has Models

`CGAL::Box_intersection_d::Box_traits_d<BoxHandle>` page [2177](#)

Types

`BoxIntersectionTraits_d::Box_parameter`

type used for passing box parameters in the functions below. Since we support in our algorithms passing the boxes by value as well as passing them as pointers, this type can be either *const B&*, *B**, or *const B** respectively, where *B* is the actual box type. The difference to the box handle type lies in the first case where the box handle would be *B* where this type is *const B&*.

`BoxIntersectionTraits_d::NT` number type to represent the box boundaries. Allowed are the builtin types *int*, *unsigned int*, *float*, and *double*.

`BoxIntersectionTraits_d::ID` type for the *id*-number, model of the *LessThanComparable* concept.

Access Functions

int `BoxIntersectionTraits_d::dimension()` returns the dimension of the box.

ID `BoxIntersectionTraits_d::id(Box_parameter box)` returns the unique *id*-number for the *box*.

NT `BoxIntersectionTraits_d::min_coord(Box_parameter box, int d)`
returns the lower boundary of the *box* in dimension *d*, $0 \leq d < \text{dimension}()$.

NT `BoxIntersectionTraits_d::max_coord(Box_parameter box, int d)`
returns the upper boundary of the *box* in dimension *d*, $0 \leq d < \text{dimension}()$.

See Also

<i>CGAL::box_intersection_d</i>	page 2161
<i>CGAL::box_self_intersection_d</i>	page 2166
<i>CGAL::box_intersection_all_pairs_d</i>	page 2164
<i>CGAL::box_self_intersection_all_pairs_d</i>	page 2169

CGAL::Box_intersection_d::Box_d<NT,int D,IdPolicy>

Definition

Box_d<*NT*,*int D*,*IdPolicy*> is a generic iso-oriented bounding box in dimension *D*. It provides in each dimension an interval with lower and upper endpoints represented with the number type *NT*. This class is designed to work smoothly with the algorithms for intersecting sequences of iso-oriented boxes. For degeneracy handling, the boxes need to provide a unique *id*-number. The policy parameter *IdPolicy* offers several choices. The template parameters have to comply with the following requirements:

- *NT*: number type for the box boundaries, needs to be a model of the *Assignable* and the *LessThanComparable* concept.
- *int D*: the dimension of the box.
- *IdPolicy*: specifies how the *id*-number will be provided. Can be one of the following types, where *ID_EXPLICIT* is the default for this parameter:
 - *ID_NONE*: no *id*-number is provided. Can be useful if *Box_d* is used as a base class for a different implementation of *id*-numbers than the ones provided here.
 - *ID_EXPLICIT*: the *id*-number is stored explicitly in the box and automatically created and assigned at construction time of the box. Note that copying a box (copy-constructor and assignment) does not create a new *id*-number but keeps the old one, which is the behavior needed by the *CGAL::box_self_intersection* algorithm. This is therefore the safe default implementation.
 - *ID_FROM_BOX_ADDRESS*: casts the address of the box into a *std::ptrdiff_t* to create the *id*-number. Works fine if the intersection algorithms work effectively with pointers to boxes, but not in the case where the algorithms work with box values, because the algorithms modify the order of the boxes, and the *CGAL::box_self_intersection* algorithm creates copies of the boxes that would not have identical *id*-numbers.

```
#include <CGAL/Box_intersection_d/Box_d.h>
and also automatically with
#include <CGAL/box_intersection_d.h>
```

Is Model for the Concepts

BoxIntersectionBox_d.....page [2171](#)

Types

Box_d<*NT*,*int D*,*IdPolicy*>::*NT* number type to represent the box boundaries. Allowed are the builtin types *int*, *unsigned int*, *float*, and *double*.

typedef std::size_t *ID*; type for the box *id*-number.

Creation

<code>Box_d<NT,int D,IdPolicy> box;</code>	Default constructor. No particular initialization.
<code>Box_d<NT,int D,IdPolicy> box(bool complete);</code>	initializes to the complete or the empty space. If empty, all interval starting(end) points will be set to positive(negative) infinity.
<code>Box_d<NT,int D,IdPolicy> box(NT lo[D], NT hi[D]);</code>	initializes the box intervals to $[lo[i],hi[i]]$, $0 \leq i < D$. <i>Precondition:</i> $lo[i] < hi[i]$ for $0 \leq i < D$.
<code>Box_d<NT,int D,IdPolicy> box(Bbox_2 bbox);</code>	constructs from bbox, exists iff $D = 2$ and $NT \equiv double$.
<code>Box_d<NT,int D,IdPolicy> box(Bbox_3 bbox);</code>	constructs from bbox, exists iff $D = 3$ and $NT \equiv double$.

Modifiers

<code>void box.init(bool complete = false)</code>	initializes to the complete or the empty space. If empty, all interval starting(end) points will be set to positive(negative) infinity.
<code>void box.extend(NT point[D])</code>	extend <i>box</i> to contain the old <i>box</i> and <i>point</i> .

Access Functions

<code>int Box_d::dimension()</code>	returns D , the dimension of the box.
<code>std::size_t box.id()</code>	returns a unique box id, see the <i>IdPolicy</i> template parameter above for the different choices. Does not exist if <i>ID_NONE</i> has been chosen for the <i>IdPolicy</i> .
<code>NT box.min_coord(int d)</code>	returns the lower boundary in dimension d , $0 \leq d < D$.
<code>NT box.max_coord(int d)</code>	returns the upper boundary in dimension d , $0 \leq d < D$.
<code>Bbox_2 box.bbox()</code>	returns the bounding box iff $D = 2$ and $NT \equiv double$.
<code>Bbox_3 box.bbox()</code>	returns the bounding box iff $D = 3$ and $NT \equiv double$.
<code>void box.extend(NT p[N])</code>	extends <i>box</i> to the smallest box that additionally contains the point represented by coordinates in <i>p</i> .
<code>void box.extend(std::pair<NT,NT> p[N])</code>	extends <i>box</i> to the smallest box that additionally contains the point represented by coordinate intervals in <i>p</i> .

See Also

<i>CGAL::box_intersection_d</i>	page 2161
<i>CGAL::box_self_intersection_d</i>	page 2166
<i>CGAL::box_intersection_all_pairs_d</i>	page 2164
<i>CGAL::box_self_intersection_all_pairs_d</i>	page 2169
<i>CGAL::Box_intersection_d::Box_with_handle_d</i> <NT, int D, Handle, IdPolicy>	page 2178
<i>CGAL::Box_intersection_d::Box_traits_d</i> <BoxHandle>	page 2177
<i>BoxIntersectionTraits_d</i>	page 2172

CGAL::Box_intersection_d::Box_traits_d<BoxHandle>

Definition

This is the default traits class for the intersection algorithms for iso-oriented boxes. There are actually three versions depending on the type of *BoxHandle*; there is one if *BoxHandle* is a class type and there are two if *BoxHandle* is a pointer type, one for a mutable and one for a const pointer, respectively.

This class implements the mapping from its *BoxHandle* argument to the *Box_parameter* type required in the *BoxIntersectionTraits_d* concept. In particular in the case where *BoxHandle* is a class type *B*, it defines *Box_parameter* to be of type *const B&*, while for the other cases it just uses the pointer type.

- *BoxHandle*: either a class type *B*, a pointer *B**, or a const-pointer *const B**, where *B* is a model of the *BoxIntersectionBox_d* concept.

`#include <CGAL/Box_intersection_d/Box_traits_d.h>` and also automatically with
`#include <CGAL/box_intersection_d.h>`

Is Model for the Concepts

`BoxIntersectionTraits_d` page [2172](#)

Creation

`Box_traits_d<BoxHandle> traits;` default constructor.

See Also

`CGAL::box_intersection_d` page [2161](#)
`CGAL::box_self_intersection_d` page [2166](#)
`CGAL::box_intersection_all_pairs_d` page [2164](#)

`BoxIntersectionBox_d` page [2171](#)
`CGAL::Box_intersection_d::Box_d<NT,int D,IdPolicy>` page [2174](#)

CGAL::Box_intersection_d::Box_with_handle_d<NT, int D, Handle, IdPolicy>

Definition

Box_with_handle_d<*NT*, *int D*, *Handle*, *IdPolicy*> is a generic iso-oriented bounding box in dimension *D* that stores additionally a handle to some underlying geometric object. It provides in each dimension an interval with lower and upper endpoints represented with the number type *NT*. This class is designed to work smoothly with the algorithms for intersecting sequences of iso-oriented boxes. For degeneracy handling, the boxes need to provide a unique *id*-number. The policy parameter *IdPolicy* offers several choices. The template parameters have to comply with the following requirements:

- *NT*: number type for the box boundaries, needs to be a model of the *Assignable* and the *LessThanComparable* concept.
- *int D*: the dimension of the box.
- *Handle* concept, e.g., a pointer, an iterator, or a circulator.
- *IdPolicy*: specifies how the *id*-number will be provided. Can be one of the following types, where *ID_EXPLICIT* is the default for this parameter:
 - *ID_NONE*: no *id*-number is provided. Can be useful to have this class as a base class for different implementations of *id*-numbers than the ones provided here.
 - *ID_EXPLICIT*: the *id*-number is stored explicitly in the box and automatically created and assigned at construction time of the box. Note that copying a box (copy-constructor and assignment) does not create a new *id*-number but keeps the old one, which is the behavior needed by the *CGAL::box_self_intersection* algorithm. This is therefore the safe default implementation.
 - *ID_FROM_BOX_ADDRESS*: casts the address of the box into a *std::ptrdiff_t* to create the *id*-number. Works fine if the intersection algorithms work effectively with pointers to boxes, but not in the case where the algorithms work with box values, because the algorithms modify the order of the boxes, and the *CGAL::box_self_intersection* algorithm creates copies of the boxes that would not have identical *id*-numbers.
 - *ID_FROM_HANDLE*: casts the address of the value of the handle into a *std::ptrdiff_t* to create the *id*-number. Works in many conceivable settings, e.g., it works with boxes copied by value or by pointer, and the self intersection test. It will not work if there is no one-to-one mapping between boxes and the geometry that is referred to with the handles, i.e., this *id*-number scheme fails if a geometric object creates several boxes with the same handle value. Note that this option is not available for the *CGAL::Box_intersection_d::Box_d* type that does not store a handle.

```
#include <CGAL/Box_intersection_d/Box_with_handle_d.h>
and also automatically with
#include <CGAL/box_intersection_d.h>
```

Is Model for the Concepts

BoxIntersectionBox_d.....page 2171

Types

Box_with_handle_d<*NT*, *int D*, *Handle*, *IdPolicy*>::*NT*

number type to represent the box boundaries. Allowed are the builtin types *int*, *unsigned int*, *float*, and *double*.

typedef std::size_t ID;

type for the box *id*-number.

Creation

Box_with_handle_d<*NT*, *int D*, *Handle*, *IdPolicy*> *box*;

Default constructor. No particular initialization.

Box_with_handle_d<*NT*, *int D*, *Handle*, *IdPolicy*> *box*(*bool complete*, *Handle h*);

initializes to the complete or the empty space. If empty, all interval starting(end) points will be set to positive(negative) infinity, sets handle to *h*.

Box_with_handle_d<*NT*, *int D*, *Handle*, *IdPolicy*> *box*(*NT lo*[*D*], *NT hi*[*D*], *Handle h*);

initializes the box intervals to [*lo*[*i*],*hi*[*i*]], $0 \leq i < D$ and sets the handle to *h*.

Precondition: *lo*[*i*] < *hi*[*i*] for $0 \leq i < D$.

Box_with_handle_d<*NT*, *int D*, *Handle*, *IdPolicy*> *box*(*Bbox_2 bbox*, *Handle h*);

constructs from *bbox* and sets the handle to *h*, exists iff $D = 2$ and $NT \equiv double$.

Box_with_handle_d<*NT*, *int D*, *Handle*, *IdPolicy*> *box*(*Bbox_3 bbox*, *Handle h*);

constructs from *bbox* and sets the handle to *h*, exists iff $D = 3$ and $NT \equiv double$.

Modifiers

void box.init(*bool complete = false*)

initializes to the complete or the empty space. If empty, all interval starting(end) points will be set to positive(negative) infinity.

void box.extend(*NT point*[*D*])

extend *box* to contain the old *box* and *point*.

Access Functions

<i>Handle</i>	<i>box.handle()</i>	returns the handle stored in <i>box</i> .
<i>int</i>	<i>Box_with_handle_d::dimension()</i>	returns <i>D</i> , the dimension of the box.
<i>std::size_t</i>	<i>box.id()</i>	returns a unique box id, see the <i>IdPolicy</i> template parameter above for the different choices. Does not exist if <i>ID_NONE</i> has been chosen for the <i>IdPolicy</i> .
<i>NT</i>	<i>box.min_coord(int d)</i>	returns the lower boundary in dimension <i>d</i> , $0 \leq d < D$.
<i>NT</i>	<i>box.max_coord(int d)</i>	returns the upper boundary in dimension <i>d</i> , $0 \leq d < D$.
<i>Bbox_2</i>	<i>box.bbox()</i>	returns the bounding box iff $D = 2$ and $NT \equiv \text{double}$.
<i>Bbox_3</i>	<i>box.bbox()</i>	returns the bounding box iff $D = 3$ and $NT \equiv \text{double}$.

See Also

<i>CGAL::box_intersection_d</i>	page 2161
<i>CGAL::box_self_intersection_d</i>	page 2166
<i>CGAL::box_intersection_all_pairs_d</i>	page 2164
<i>CGAL::box_self_intersection_all_pairs_d</i>	page 2169
<i>CGAL::Box_intersection_d::Box_traits_d<BoxHandle></i>	page 2177
<i>BoxIntersectionTraits_d</i>	page 2172

Part XI

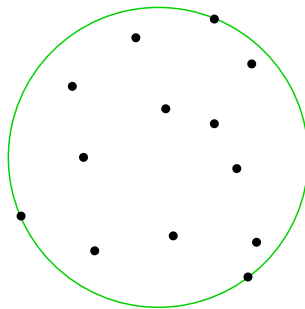
Geometric Optimization

Chapter 38

Geometric Optimization

Kaspar Fischer, Bernd Gärtner, Thomas Herrmann, Michael Hoffmann, Eli Packer, and Sven Schönherr

Geometric optimization deals with problems of computing geometric objects which are optimal subject to certain criteria and constraints. Our running example will be the problem of computing the sphere of smallest radius which contains a given point set in d -dimensional Euclidean space.



The geometric optimization algorithms in CGAL fall into three categories, “Bounding Volumes”, “Optimal Distances” and “Advanced Techniques”.

38.1 Bounding and Inscribed Volumes

This category contains algorithms which for a given point set compute the “best” circumscribing (or inscribed) object from a specific class. If the class consists of all spheres in d -dimensional Euclidean space and “best” is defined as having smallest radius, then we obtain the smallest enclosing sphere problem already mentioned above.

In the following example a smallest enclosing circle (`CGAL::Min_circle_2<Traits>`) is constructed from points on a line and written to standard output. The example shows that it is advisable to switch on random shuffling in order to deal with a ‘bad’ order of the input points.

```
#include <CGAL/Homogeneous.h>
#include <CGAL/Min_circle_2.h>
#include <CGAL/Min_circle_2_traits_2.h>
```

```

#include <CGAL/Gmpz.h>
#include <iostream>

// typedefs
typedef CGAL::Gmpz NT;
typedef CGAL::Homogeneous<NT> K;
typedef CGAL::Min_circle_2_traits_2<K> Traits;
typedef CGAL::Min_circle_2<Traits> Min_circle;

typedef K::Point_2 Point;

// main
int
main( int, char**)
{
    int n = 100;
    Point* P = new Point[ n];

    for ( int i = 0; i < n; ++i)
P[ i] = Point( (i%2 == 0 ? i : -i), 0);
    // (0,0), (-1,0), (2,0), (-3,0), ...

    Min_circle mc1( P, P+n, false); // very slow
    Min_circle mc2( P, P+n, true); // fast

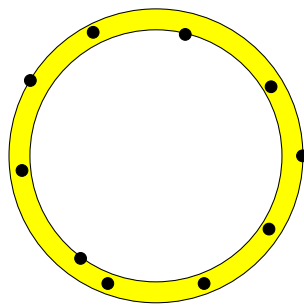
    CGAL::set_pretty_mode( std::cout);
    std::cout << mc2;

    delete[] P;

    return( 0);
}

```

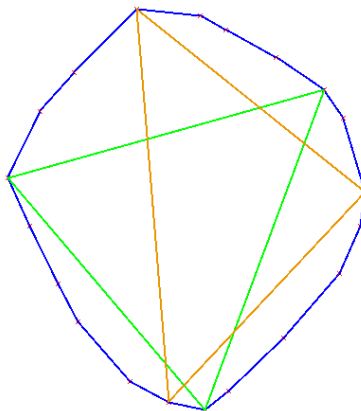
Other classes for which we provide solutions are ellipses (*CGAL::Min_ellipse_2<Traits>*), rectangles (*CGAL::min_rectangle_2*), parallelograms (*CGAL::min_parallelogram_2*) and strips (*CGAL::min_strip_2*) in the plane, with appropriate optimality criteria. For arbitrary dimensions we provide smallest enclosing spheres for points (*CGAL::Min_sphere_d<Traits>*) and spheres for spheres (*CGAL::Min_sphere_of_spheres_d<Traits>*), smallest enclosing annuli (*CGAL::Min_annulus_d<Traits>*), and approximate minimum-volume enclosing ellipsoid with user-specified approximation ratio (*CGAL::Approximate_min_ellipsoid_d<Traits>*).



Bounding volumes can be used to obtain simple approximations of complicated objects. For example, consider the problem of deciding whether two moving polygons currently intersect. An obvious solution is to discretize

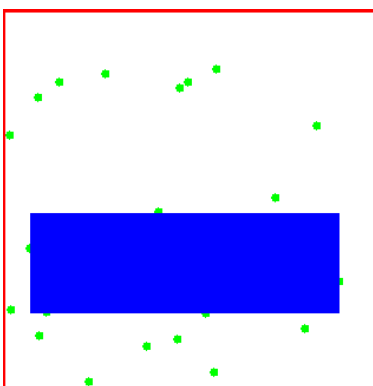
time and perform a full intersection test for any time step. If the polygons are far apart most of the time, this is unnecessary. Instead, simple bounding volumes (for examples, circles) are computed for both polygons at their initial positions. At subsequent time steps, an intersection test between the moving bounding circles replaces the actual intersection test; only if the circles do intersect, the expensive intersection test between the polygons is performed. In practice, bounding volume hierarchies are often used on top of simple bounding volumes to approximate complicated objects more accurately.

As far as inscribed volumes are concerned, we provide algorithms for computing maximal inscribed k -gons (triangles, quadrilaterals, ...) of a planar point set P . Maximal k -gons are convex, and it is known that their vertices can be chosen to be vertices of the convex hull of P . Hence, the functions `CGAL::maximum_area_inscribed_k_gon_2` and `CGAL::maximum_perimeter_inscribed_k_gon_2` operate on convex polygons only. The example below shows that the largest area triangle (green) and the largest perimeter triangle (orange, containing the top point) of a point set are different in general.



We further provide an algorithm for computing the maximal area inscribed axis parallel rectangle

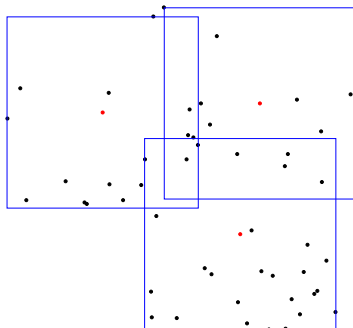
Given a set of points in the plane, the class `CGAL::Largest_empty_iso_rectangle_2<T>` is a data structure that maintains an iso-rectangle with the largest area among all iso-rectangles that are inside a given iso-rectangles, and that do not contain any point of the point set.



Bounding and inscribed volumes are also frequently applied to extract geometric properties of objects. For example, the smallest enclosing annulus of a point set can be used to test whether a set of points is approximately cospherical. Here, the width of the annulus (or its area, or still another criterion that we use) is a good measure for this property. The largest area triangle is for example used in heuristics for matching archaeological aerial photographs. Largest perimeter triangles are used in scoring cross country soaring flights, where the goal is

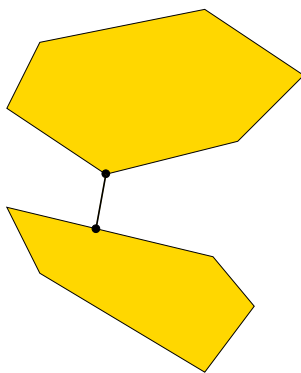
basically to fly as far as possible, but still return to the departure airfield. To score simply based on the total distance flown is not a good measure, since circling in thermals allows to increase it easily.

Bounding volumes also define geometric “center points” of objects. For example, if two objects are to be matched (approximately), one approach is to first apply the translation that maps the centers of their smallest enclosing spheres onto each other. Simpler centers are possible, of course (center of gravity, center of bounding box), but more advanced bounding volumes might give better results in some cases. It can also make sense to consider several center points instead of just one. For example, we provide algorithms to cover a planar point set with between two and four minimal boxes (*CGAL::rectangular_p_center_2*). Below is an example covering with three boxes; the center points are shown in red.



38.2 Optimal Distances

This section currently consists of two algorithms only. On one hand, one can compute the computation of the distance between the convex hulls of two given point sets in d -dimensional Euclidean space (*CGAL::Polytope_distance_d<Traits>*). Moreover, it is possible to compute the width of a point set in three dimensions (*CGAL::Width_3<Traits>*).



The obvious application is collision detection between convex bodies in space. In the spirit of the bounding volume application above, it also makes sense for nonconvex objects: a full intersection test between complicated objects could in a first stage be approximated with the test between the convex hulls of the objects. Only if the hulls intersect, a full intersection test is necessary.

To dampen fears concerning the performance of the distance computation, we want to mention that the convex hulls of the input point sets are not explicitly computed. This avoids a runtime which grows exponentially in d . In fact, the runtime is almost always linear in the size of the two point sets.

38.3 Monotone and Sorted Matrix Search

CGAL::monotone_matrix_search and *CGAL::sorted_matrix_search* are techniques that deal with the problem of efficiently finding largest entries in matrices with certain structural properties. Many concrete problems can be modelled as matrix search problems, and for some of them we provide explicit solutions that allow you to solve them without knowing about the matrix search technique. Examples are, the computation of all furthest neighbors for the vertices of a convex polygon, maximal k -gons inscribed into a planar point set, and computing rectangular p -centers.

Geometric Optimization Reference Manual

Kaspar Fischer, Bernd Gärtner, Thomas Herrmann, Michael Hoffmann, Eli Packer, and Sven Schönherr

This chapter describes concepts, classes, and functions for solving geometric optimization problems. They are divided into four categories.

Bounding Areas and Volumes. Smallest enclosing circle and ellipse (2D), smallest enclosing rectangle, parallelogram, and strip (2D), rectangular p -center (2D), smallest enclosing sphere and annulus (dD), approximate minimum-volume enclosing ellipsoid with user-specified approximation ratio (dD).

Inscribed Areas. Maximum area and perimeter inscribed k -gon (2D), extremal inscribed k -gon (2D), largest empty isorectangle (2D).

Optimal Distances. All furthest neighbors (2D), width of point set (3D), polytope distance (dD).

Advanced Techniques. Monotone and sorted matrix search.

Assertions

The optimization code uses infix *OPTIMISATION* in the assertions, e.g. defining the compiler flag *CGAL_OPTIMISATION_NO_PRECONDITIONS* switches precondition checking off, cf. Section ??.

38.4 Classified References Pages

Bounding Areas and Volumes

<i>CGAL::Min_circle_2<Traits></i>	page 2193
<i>CGAL::Min_circle_2_traits_2<K></i>	page 2199
<i>MinCircle2Traits</i>	page 2200

<i>CGAL::Min_ellipse_2<Traits></i>	page 2203
<i>CGAL::Min_ellipse_2_traits_2<K></i>	page 2210
<i>MinEllipse2Traits</i>	page 2212
<i>CGAL::Approximate_min_ellipsoid_d<Traits></i>	page 2265
<i>ApproximateMinEllipsoid_d_Traits_d</i>	page 2274
<i>CGAL::min_rectangle_2</i>	page 2215
<i>CGAL::min_parallelogram_2</i>	page 2217
<i>CGAL::min_strip_2</i>	page 2219
<i>CGAL::Min_quadrilateral_default_traits_2<R></i>	page ??
<i>MinQuadrilateralTraits_2</i>	page 2225
<i>CGAL::rectangular_p_center_2</i>	page 2229
<i>CGAL::Rectangular_p_center_default_traits_2<R></i>	page 2232
<i>RectangularPCenterTraits_2</i>	page 2235
<i>CGAL::Min_sphere_d<Traits></i>	page 2238
<i>CGAL::Min_annulus_d<Traits></i>	page 2244
<i>CGAL::Optimisation_d_traits_2<K,ET,NT></i>	page 2279
<i>CGAL::Optimisation_d_traits_3<K,ET,NT></i>	page 2281
<i>CGAL::Optimisation_d_traits_d<K,ET,NT></i>	page 2283
<i>OptimisationDTraits</i>	page 2285
<i>CGAL::Min_sphere_of_spheres_d<Traits></i>	page 2251
<i>MinSphereOfSpheresTraits</i>	page 2257

Inscribed Areas

<i>CGAL::maximum_area_inscribed_k_gon_2</i>	page 2287
<i>CGAL::maximum_perimeter_inscribed_k_gon_2</i>	page 2289
<i>CGAL::extremal_polygon_2</i>	page 2291
<i>CGAL::Largest_empty_iso_rectangle_2<T></i>	page 2298
<i>CGAL::Extremal_polygon_area_traits_2<K></i>	page 2292
<i>CGAL::Extremal_polygon_perimeter_traits_2<K></i>	page 2294
<i>ExtremalPolygonTraits_2</i>	page 2296
<i>LargestEmptyIsoRectangleTraits_2</i>	page 2301

Optimal Distances

<i>CGAL::all_furthest_neighbors_2</i>	page 2303
<i>AllFurthestNeighborsTraits_2</i>	page 2305
<i>CGAL::Width_3<Traits></i>	page 2306
<i>CGAL::Width_default_traits_3<K></i>	page 2310

WidthTraits_3	page 2312
CGAL::Polytope_distance_d<Traits>	page 2314
CGAL::Optimisation_d_traits_2<K,ET,NT>	page 2279
CGAL::Optimisation_d_traits_3<K,ET,NT>	page 2281
CGAL::Optimisation_d_traits_d<K,ET,NT>	page 2283
OptimisationDTraits	page 2285

Advanced Techniques

CGAL::monotone_matrix_search	page 2321
CGAL::Dynamic_matrix<M>	page 2323
MonotoneMatrixSearchTraits	page 2325
BasicMatrix	page 2327
CGAL::sorted_matrix_search	page 2328
CGAL::Sorted_matrix_search_traits_adaptor<F,M>	page 2331
SortedMatrixSearchTraits	page 2333

38.5 Alphabetical List of Reference Pages

AllFurthestNeighborsTraits_2	page 2305
all_furthest_neighbors_2	page 2303
ApproximateMinEllipsoid_d_Traits_d	page 2274
Approximate_min_ellipsoid_d<Traits>	page 2265
Approximate_min_ellipsoid_d_traits_2<K,ET>	page 2276
Approximate_min_ellipsoid_d_traits_3<K,ET>	page 2277
Approximate_min_ellipsoid_d_traits_d<K,ET>	page 2278
BasicMatrix	page 2327
Dynamic_matrix<M>	page 2323
ExtremalPolygonTraits_2	page 2296
extremal_polygon_2	page 2291
Extremal_polygon_area_traits_2<K>	page 2292
Extremal_polygon_perimeter_traits_2<K>	page 2294
LargestEmptyIsoRectangleTraits_2	page 2301
Largest_empty_iso_rectangle_2<T>	page 2298
maximum_area_inscribed_k_gon_2	page 2287
maximum_perimeter_inscribed_k_gon_2	page 2289
MinCircle2Traits	page 2200
MinEllipse2Traits	page 2212
MinQuadrilateralTraits_2	page 2225
MinSphereOfSpheresTraits	page 2257
Min_annulus_d<Traits>	page 2244
Min_circle_2<Traits>	page 2193
Min_circle_2_traits_2<K>	page 2199
Min_ellipse_2<Traits>	page 2203
Min_ellipse_2_traits_2<K>	page 2210

<i>min_parallelogram_2</i>	page 2217
<i>Min_quadrilateral_default_traits_2<Kernel></i>	page 2221
<i>min_rectangle_2</i>	page 2215
<i>Min_sphere_d<Traits></i>	page 2238
<i>Min_sphere_of_spheres_d<Traits></i>	page 2251
<i>Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm></i>	page 2259
<i>Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm></i>	page 2261
<i>Min_sphere_of_spheres_d_traits_d<K,FT,Dim,UseSqrt,Algorithm></i>	page 2263
<i>min_strip_2</i>	page 2219
<i>MonotoneMatrixSearchTraits</i>	page 2325
<i>monotone_matrix_search</i>	page 2321
<i>OptimisationDTraits</i>	page 2285
<i>Optimisation_d_traits_2<K,ET,NT></i>	page 2279
<i>Optimisation_d_traits_3<K,ET,NT></i>	page 2281
<i>Optimisation_d_traits_d<K,ET,NT></i>	page 2283
<i>Polytope_distance_d<Traits></i>	page 2314
<i>RectangularPCenterTraits_2</i>	page 2235
<i>rectangular_p_center_2</i>	page 2229
<i>Rectangular_p_center_default_traits_2<R></i>	page 2232
<i>SortedMatrixSearchTraits</i>	page 2333
<i>Sorted_matrix_search_traits_adaptor<F,M></i>	page 2331
<i>sorted_matrix_search</i>	page 2328
<i>WidthTraits_3</i>	page 2312
<i>Width_3<Traits></i>	page 2306
<i>Width_default_traits_3<K></i>	page 2310

CGAL::Min_circle_2<Traits>

Definition

An object of the class *Min_circle_2<Traits>* is the unique circle of smallest area enclosing a finite (multi)set of points in two-dimensional Euclidean space \mathbb{E}_2 . For a point set P we denote by $mc(P)$ the smallest circle that contains all points of P . Note that $mc(P)$ can be degenerate, i.e. $mc(P) = \emptyset$ if $P = \emptyset$ and $mc(P) = \{p\}$ if $P = \{p\}$.

An inclusion-minimal subset S of P with $mc(S) = mc(P)$ is called a *support set*, the points in S are the *support points*. A support set has size at most three, and all its points lie on the boundary of $mc(P)$. In general, neither the support set nor its size are necessarily unique.

The underlying algorithm can cope with all kinds of input, e.g. P may be empty or points may occur more than once. The algorithm computes a support set S which remains fixed until the next insert or clear operation.

Please note: This class is (almost) obsolete. The class *CGAL::Min_sphere_of_spheres_d<Traits>* solves a more general problem and is faster than *Min_circle_2<Traits>* even if used only for points in two dimensions as input. Most importantly, *CGAL::Min_sphere_of_spheres_d<Traits>* has a specialized implementation for floating-point arithmetic which ensures correct results in a large number of cases (including highly degenerate ones). In contrast, *Min_circle_2<Traits>* is not tuned for floating-point computations. The only advantage of *Min_circle_2<Traits>* over *CGAL::Min_sphere_of_spheres_d<Traits>* is that the former can deal with points in homogeneous coordinates, in which case the algorithm is division-free. Thus, *Min_circle_2<Traits>* might still be an option in case your input number type cannot (efficiently) divide.

```
#include <CGAL/Min_circle_2.h>
```

Requirements

The template parameter *Traits* is a model for *MinCircle2Traits*.

We provide the model *CGAL::Min_circle_2_traits_2* using the two-dimensional CGAL kernel.

Types

Min_circle_2<Traits>::Point typedef to *Traits::Point*.

Min_circle_2<Traits>::Circle typedef to *Traits::Circle*.

Min_circle_2<Traits>::Point_iterator

non-mutable model of the STL concept *BidirectionalIterator* with value type *Point*. Used to access the points of the smallest enclosing circle.

Min_circle_2<Traits>::Support_point_iterator

non-mutable model of the STL concept *RandomAccessIterator* with value type *Point*. Used to access the support points of the smallest enclosing circle.

Creation

A *Min_circle_2*<Traits> object can be created from an arbitrary point set P and by specialized construction methods expecting no, one, two or three points as arguments. The latter methods can be useful for reconstructing $mc(P)$ from a given support set S of P .

```
template < class InputIterator >
Min_circle_2<Traits> min_circle( InputIterator first,
                                InputIterator last,
                                bool randomize,
                                Random& random = CGAL::default_random,
                                Traits traits = Traits())
```

initializes *min_circle* to $mc(P)$ with P being the set of points in the range $[first, last)$. If *randomize* is *true*, a random permutation of P is computed in advance, using the random numbers generator *random*. Usually, this will not be necessary, however, the algorithm's efficiency depends on the order in which the points are processed, and a bad order might lead to extremely poor performance (see example below).

Requirement: The value type of *first* and *last* is *Point*.

```
Min_circle_2<Traits> min_circle( Traits traits = Traits());
```

initializes *min_circle* to $mc(\emptyset)$, the empty set.
Postcondition: *min_circle.is_empty()* = *true*.

```
Min_circle_2<Traits> min_circle( Point p, Traits traits = Traits());
```

initializes *min_circle* to $mc(\{p\})$, the set $\{p\}$.
Postcondition: *min_circle.is_degenerate()* = *true*.

```
Min_circle_2<Traits> min_circle( Point p1, Point p2, Traits traits = Traits());
```

initializes *min_circle* to $mc(\{p1, p2\})$, the circle with diameter equal to the segment connecting $p1$ and $p2$.

```
Min_circle_2<Traits> min_circle( Point p1, Point p2, Point p3, Traits traits = Traits());
```

initializes *min_circle* to $mc(\{p1, p2, p3\})$.

Access Functions

int *min_circle.number_of_points()* returns the number of points of *min_circle*, i.e. $|P|$.

int *min_circle.number_of_support_points()*

returns the number of support points of *min_circle*, i.e. $|S|$.

Point_iterator *min_circle.points_begin()* returns an iterator referring to the first point of *min_circle*.

Point_iterator *min_circle.points_end()* returns the corresponding past-the-end iterator.

Support_point_iterator *min_circle.support_points_begin()* returns an iterator referring to the first support point of *min_circle*.

Support_point_iterator *min_circle.support_points_end()* returns the corresponding past-the-end iterator.

Point *min_circle.support_point(int i)*

returns the *i*-th support point of *min_circle*. Between two modifying operations (see below) any call to *min_circle.support_point(i)* with the same *i* returns the same point.
Precondition: $0 \leq i < \text{min_circle.number_of_support_points}()$.

Circle *min_circle.circle()* returns the current circle of *min_circle*.

Predicates

By definition, an empty *Min_circle_2<Traits>* has no boundary and no bounded side, i.e. its unbounded side equals the whole space \mathbb{E}_2 .

CGAL::Bounded_side

min_circle.bounded_side(Point p)

returns *CGAL::ON_BOUNDED_SIDE*, *CGAL::ON_BOUNDARY*, or *CGAL::ON_UNBOUNDED_SIDE* iff *p* lies properly inside, on the boundary of, or properly outside of *min_circle*, resp.

bool *min_circle.has_on_bounded_side(Point p)*

returns *true*, iff *p* lies properly inside *min_circle*.

bool *min_circle.has_on_boundary(Point p)*

returns *true*, iff *p* lies on the boundary of *min_circle*.

bool *min_circle.has_on_unbounded_side(Point p)*

returns *true*, iff *p* lies properly outside of *min_circle*.

bool *min_circle.is_empty()*

returns *true*, iff *min_circle* is empty (this implies degeneracy).

bool min_circle.is_degenerate() returns *true*, iff *min_circle* is degenerate, i.e. if *min_circle* is empty or equal to a single point, equivalently if the number of support points is less than 2.

Modifiers

New points can be added to an existing *min_circle*, allowing to build $mc(P)$ incrementally, e.g. if P is not known in advance. Compared to the direct creation of $mc(P)$, this is not much slower, because the construction method is incremental itself.

void min_circle.insert(Point p) inserts p into *min_circle* and recomputes the smallest enclosing circle.

template < class InputIterator >

void min_circle.insert(InputIterator first, InputIterator last)

inserts the points in the range $[first, last)$ into *min_circle* and recomputes the smallest enclosing circle by calling *insert(p)* for each point p in $[first, last)$.

Requirement: The value type of *first* and *last* is *Point*.

void min_circle.clear() deletes all points in *min_circle* and sets *min_circle* to the empty set.
Postcondition: *min_circle.is_empty() = true*.

Validity Check

An object *min_circle* is valid, iff

- *min_circle* contains all points of its defining set P ,
- *min_circle* is the smallest circle spanned by its support set S , and
- S is minimal, i.e. no support point is redundant.

bool min_circle.is_valid(bool verbose = false, int level = 0)

returns *true*, iff *min_circle* is valid. If *verbose* is *true*, some messages concerning the performed checks are written to standard error stream. The second parameter *level* is not used, we provide it only for consistency with interfaces of other classes.

Miscellaneous

const Traits&

min_circle.traits() returns a const reference to the traits class object.

I/O

std::ostream&

std::ostream& os << min_circle

writes *min_circle* to output stream *os*.

Requirement: The output operator is defined for *Point* (and for *Circle*, if pretty printing is used).

std::istream&

std::istream& is >> min_circle&

reads *min_circle* from input stream *is*.

Requirement: The input operator is defined for *Point*.

#include <CGAL/IO/Window_stream.h>

CGAL::Window_stream&

CGAL::Window_stream& ws << min_circle

writes *min_circle* to window stream *ws*.

Requirement: The window stream output operator is defined for *Point* and *Circle*.

See Also

<i>CGAL::Min_ellipse_2<Traits></i>	page 2203
<i>CGAL::Min_sphere_d<Traits></i>	page 2238
<i>CGAL::Min_sphere_of_spheres_d<Traits></i>	page 2251
<i>CGAL::Min_circle_2_traits_2<K></i>	page 2199
<i>MinCircle2Traits</i>	page 2200

Implementation

We implement the incremental algorithm of Welzl, with move-to-front heuristic [[Wel91](#)]. The whole implementation is described in [[GS98a](#)].

If randomization is chosen, the creation time is almost always linear in the number of points. Access functions and predicates take constant time, inserting a point might take up to linear time, but substantially less than computing the new smallest enclosing circle from scratch. The clear operation and the check for validity each takes linear time.

Example

To illustrate the creation of *Min_circle_2<Traits>* and to show that randomization can be useful in certain cases, we give an example.

```

// file: examples/Min_circle_2/example_Min_circle_2.C

// includes
#include <CGAL/Homogeneous.h>
#include <CGAL/Min_circle_2.h>
#include <CGAL/Min_circle_2_traits_2.h>
#include <CGAL/Gmpz.h>
#include <iostream>

// typedefs
typedef CGAL::Gmpz NT;
typedef CGAL::Homogeneous<NT> K;
typedef CGAL::Min_circle_2_traits_2<K> Traits;
typedef CGAL::Min_circle_2<Traits> Min_circle;

typedef K::Point_2 Point;

// main
int
main( int, char**)
{
    int n = 100;
    Point* P = new Point[ n];

    for ( int i = 0; i < n; ++i)
        P[ i] = Point( (i%2 == 0 ? i : -i), 0);
    // (0,0), (-1,0), (2,0), (-3,0), ...

    Min_circle mc1( P, P+n, false); // very slow
    Min_circle mc2( P, P+n, true); // fast

    CGAL::set_pretty_mode( std::cout);
    std::cout << mc2;

    delete[] P;

    return( 0);
}

```


MinCircle2Traits

Definition

This concept defines the requirements for traits classes of *CGAL::Min_circle_2<Traits>*.

Types

<i>MinCircle2Traits:: Point</i>	The point type must provide default and copy constructor, assignment and equality test.
<i>MinCircle2Traits:: Circle</i>	The circle type must fulfill the requirements listed below in the next section.

Variables

<i>Circle</i>	<i>circle;</i>	The current circle. This variable is maintained by the algorithm, the user should neither access nor modify it directly.
---------------	----------------	--------------------------------------------------------------------------------------------------------------------------

Creation

Only default and copy constructor are required.

MinCircle2Traits traits;

MinCircle2Traits traits(MinCircle2Traits);

Operations

The following predicate is only needed, if the member function *is_valid* of *Min_circle_2* is used.

<i>CGAL::Orientation</i>	<i>traits.orientation(Point p, Point q, Point r)</i>	returns constants <i>CGAL::LEFT_TURN</i> , <i>CGAL::COLLINEAR</i> , or <i>CGAL::RIGHT_TURN</i> iff <i>r</i> lies properly to the left of, on, or properly to the right of the oriented line through <i>p</i> and <i>q</i> , resp.
--------------------------	-------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Has Models

CGAL::Min_circle_2_traits_2<K> page [2199](#)

See Also

CGAL::Min_circle_2<Traits> page [2193](#)

Circle Type (*Circle*)

Definition

An object of the class *Circle* is a circle in two-dimensional Euclidean plane \mathbb{E}_2 . Its boundary splits the plane into a bounded and an unbounded side. By definition, an empty *Circle* has no boundary and no bounded side, i.e. its unbounded side equals the whole plane \mathbb{E}_2 . A *Circle* containing exactly one point p has no bounded side, its boundary is $\{p\}$, and its unbounded side equals $\mathbb{E}_2 \setminus \{p\}$.

Types

Circle::Point Point type.

The following type is only needed, if the member function *is_valid* of *Min_circle_2* is used.

<i>Circle::Distance</i>	Distance type. The function <i>squared_radius</i> (see below) returns an object of this type.
-------------------------	-----------------------------------------------------------------------------------------------

Creation

<code>void</code>	<code>circle.set()</code>	sets <i>circle</i> to the empty circle.
-------------------	---------------------------	-----------------------------------------

<i>void</i>	<i>circle.set(Point p)</i>	sets <i>circle</i> to the circle containing exactly $\{p\}$.
-------------	-----------------------------	---------------------------------------------------------------

<i>void</i>	<i>circle.set(Point p, Point q)</i>	sets <i>circle</i> to the circle with diameter equal to the segment connecting <i>p</i> and <i>q</i> . The algorithm guarantees that <i>set</i> is never called with two equal points.
-------------	--------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

void circle.set(Point p, Point q, Point r)

sets *circle* to the circle through *p,q,r*. The algorithm guarantees that *set* is never called with three collinear points.

Predicates

bool *circle.has_on_unbounded_side(Point p)*

returns *true*, iff *p* lies properly outside of *circle*.

Each of the following predicates is only needed, if the corresponding predicate of *Min_circle_2* is used.

CGAL::Bounded_side

circle.bounded_side(Point p)

returns `CGAL::ON_BOUNDED_SIDE`, `CGAL::ON_BOUNDARY`, or `CGAL::ON_UNBOUNDED_SIDE` iff p lies properly inside, on the boundary, or properly outside of *circle*, resp.

bool *circle.has_on_bounded_side(Point p)*

returns *true*, iff *p* lies properly inside *circle*.

bool *circle.has_on_boundary(Point p)*

returns *true*, iff *p* lies on the boundary of *circle*.

bool *circle.is_empty()*

returns *true*, iff *circle* is empty (this implies degeneracy).

bool *circle.is_degenerate()*

returns *true*, iff *circle* is degenerate, i.e. if *circle* is empty or equal to a single point.

Additional Operations for Checking

The following operations are only needed, if the member function *is_valid* of *Min_circle_2* is used.

bool *circle == circle2*

returns *true*, iff *circle* and *circle2* are equal.

Point *circle.center()*

returns the center of *circle*.

Distance *circle.squared_radius()*

returns the squared radius of *circle*.

I/O

The following I/O operators are only needed, if the corresponding I/O operators of *Min_circle_2* are used.

std::ostream&

std::ostream& os << circle writes *circle* to output stream *os*.

CGAL::Window_stream&

CGAL::Window_stream& ws << circle

writes *circle* to window stream *ws*.

CGAL::Min_ellipse_2<Traits>

Definition

An object of the class *Min_ellipse_2<Traits>* is the unique ellipse of smallest area enclosing a finite (multi)set of points in two-dimensional euclidean space \mathbb{E}_2 . For a point set P we denote by $me(P)$ the smallest ellipse that contains all points of P . Note that $me(P)$ can be degenerate, i.e. $me(P) = \emptyset$ if $P = \emptyset$, $me(P) = \{p\}$ if $P = \{p\}$, and $me(P) = \{(1-\lambda)p + \lambda q \mid 0 \leq \lambda \leq 1\}$ if $P = \{p, q\}$.

An inclusion-minimal subset S of P with $me(S) = me(P)$ is called a *support set*, the points in S are the *support points*. A support set has size at most five, and all its points lie on the boundary of $me(P)$. In general, neither the support set nor its size are necessarily unique.

The underlying algorithm can cope with all kinds of input, e.g. P may be empty or points may occur more than once. The algorithm computes a support set S which remains fixed until the next insert or clear operation.

```
#include <CGAL/Min_ellipse_2.h>
```

Requirements

The template parameter *Traits* is a model for *MinEllipse2Traits*.

We provide the model *CGAL::Min_ellipse_2_traits_2<K>* using the two-dimensional CGAL kernel.

Types

Min_ellipse_2<Traits>::Point Typedef to *Traits::Point*.

Min_ellipse_2<Traits>::Ellipse Typedef to *Traits::Ellipse*. If you are using the predefined traits class *CGAL::Min_ellipse_2_traits_2<K>*, you can access the coefficients of the ellipse, see the documentation of *CGAL::Min_ellipse_2_traits_2<K>* and the example below.

Min_ellipse_2<Traits>::Point_iterator

Non-mutable model of the STL concept *BidirectionalIterator* with value type *Point*. Used to access the points of the smallest enclosing ellipse.

Min_ellipse_2<Traits>::Support_point_iterator

Non-mutable model of the STL concept *RandomAccessIterator* with value type *Point*. Used to access the support points of the smallest enclosing ellipse.

Creation

A *Min_ellipse_2*<*Traits*> object can be created from an arbitrary point set P and by specialized construction methods expecting no, one, two, three, four or five points as arguments. The latter methods can be useful for reconstructing $me(P)$ from a given support set S of P .

template < class InputIterator >

Min_ellipse_2<*Traits*> *min_ellipse*(*InputIterator* *first*,
InputIterator *last*,
bool *randomize*,
Random& *random* = *default_random*,
Traits *traits* = *Traits*())

initializes *min_ellipse* to $me(P)$ with P being the set of points in the range $[first, last)$. If *randomize* is *true*, a random permutation of P is computed in advance, using the random numbers generator *random*. Usually, this will not be necessary, however, the algorithm's efficiency depends on the order in which the points are processed, and a bad order might lead to extremely poor performance (see example below).

Requirement: The value type of *first* and *last* is *Point*.

Min_ellipse_2<*Traits*> *min_ellipse*(*Traits* *traits* = *Traits*());

creates a variable *min_ellipse* of type *Min_ellipse_2*<*Traits*>. It is initialized to $me(\emptyset)$, the empty set.

Postcondition: *min_ellipse.is_empty()* = *true*.

Min_ellipse_2<*Traits*> *min_ellipse*(*Point* *p*, *Traits* *traits* = *Traits*());

initializes *min_ellipse* to $me(\{p\})$, the set $\{p\}$.

Postcondition: *min_ellipse.is_degenerate()* = *true*.

Min_ellipse_2<*Traits*> *min_ellipse*(*Point* *p*, *Point* *q*, *Traits* *traits* = *Traits*());

initializes *min_ellipse* to $me(\{p, q\})$,
the set $\{(1 - \lambda)p + \lambda q \mid 0 \leq \lambda \leq 1\}$.

Postcondition: *min_ellipse.is_degenerate()* = *true*.

Min_ellipse_2<*Traits*> *min_ellipse*(*Point* *p1*, *Point* *p2*, *Point* *p3*, *Traits* *traits* = *Traits*());

initializes *min_ellipse* to $me(\{p1, p2, p3\})$.

Min_ellipse_2<*Traits*> *min_ellipse*(*Point* *p1*, *Point* *p2*, *Point* *p3*, *Point* *p4*, *Traits* *traits* = *Traits*());

initializes *min_ellipse* to $me(\{p1, p2, p3, p4\})$.

Min_ellipse_2<*Traits*> *min_ellipse*(*Point* *p1*, *Point* *p2*, *Point* *p3*, *Point* *p4*, *Point* *p5*, *Traits* *traits* = *Traits*());

initializes *min_ellipse* to $me(\{p1, p2, p3, p4, p5\})$.

Access Functions

int *min_ellipse.number_of_points()*

returns the number of points of *min_ellipse*, i.e. $|P|$.

int *min_ellipse.number_of_support_points()*

returns the number of support points of *min_ellipse*, i.e. $|S|$.

Point_iterator

min_ellipse.points_begin()

returns an iterator referring to the first point of *min_ellipse*.

Point_iterator

min_ellipse.points_end()

returns the corresponding past-the-end iterator.

Support_point_iterator

min_ellipse.support_points_begin()

returns an iterator referring to the first support point of *min_ellipse*.

Support_point_iterator

min_ellipse.support_points_end()

returns the corresponding past-the-end iterator.

Point *min_ellipse.support_point(int i)*

returns the *i*-th support point of *min_ellipse*. Between two modifying operations (see below) any call to *min_ellipse.support_point(i)* with the same *i* returns the same point.

Precondition: $0 \leq i < \text{min_ellipse.number_of_support_points}()$.

Ellipse *min_ellipse.ellipse()*

returns the current ellipse of *min_ellipse*.

Predicates

By definition, an empty *Min_ellipse_2<Traits>* has no boundary and no bounded side, i.e. its unbounded side equals the whole space \mathbb{E}_2 .

CGAL::Bounded_side

min_ellipse.bounded_side(Point p)

returns *CGAL::ON_BOUNDED_SIDE*, *CGAL::ON_BOUNDARY*, or *CGAL::ON_UNBOUNDED_SIDE* iff *p* lies properly inside, on the boundary of, or properly outside of *min_ellipse*, resp.

bool *min_ellipse.has_on_bounded_side(Point p)*

returns *true*, iff *p* lies properly inside *min_ellipse*.

bool *min_ellipse.has_on_boundary(Point p)*

returns *true*, iff *p* lies on the boundary of *min_ellipse*.

bool *min_ellipse.has_on_unbounded_side(Point p)*

returns *true*, iff *p* lies properly outside of *min_ellipse*.

bool *min_ellipse.is_empty()*

returns *true*, iff *min_ellipse* is empty (this implies degeneracy).

bool *min_ellipse.is_degenerate()*

returns *true*, iff *min_ellipse* is degenerate, i.e. if *min_ellipse* is empty, equal to a single point or equal to a segment, equivalently if the number of support points is less than 3.

Modifiers

New points can be added to an existing *min_ellipse*, allowing to build $me(P)$ incrementally, e.g. if P is not known in advance. Compared to the direct creation of $me(P)$, this is not much slower, because the construction method is incremental itself.

void *min_ellipse.insert(Point p)*

inserts *p* into *min_ellipse* and recomputes the smallest enclosing ellipse.

template < *class InputIterator* >

void *min_ellipse.insert(InputIterator first, InputIterator last)*

inserts the points in the range $[first, last)$ into *min_ellipse* and recomputes the smallest enclosing ellipse by calling *insert(p)* for each point *p* in $[first, last)$.

Requirement: The value type of *first* and *last* is *Point*.

void *min_ellipse.clear()*

deletes all points in *min_ellipse* and sets *min_ellipse* to the empty set.

Postcondition: *min_ellipse.is_empty() = true*.

Validity Check

An object *min_ellipse* is valid, iff

- *min_ellipse* contains all points of its defining set P ,
- *min_ellipse* is the smallest ellipse spanned by its support set S , and
- S is minimal, i.e. no support point is redundant.

Note: In this release only the first item is considered by the validity check.

bool min_ellipse.is_valid(bool verbose = false, int level = 0)

returns *true*, iff *min_ellipse* contains all points of its defining set *P*. If *verbose* is *true*, some messages concerning the performed checks are written to standard error stream. The second parameter *level* is not used, we provide it only for consistency with interfaces of other classes.

Miscellaneous

const Traits&

min_ellipse.traits() returns a const reference to the traits class object.

I/O

std::ostream&

std::ostream& os << min_ellipse

writes *min_ellipse* to output stream *os*.

Requirement: The output operator is defined for *Point* (and for *Ellipse*, if pretty printing is used).

std::istream&

std::istream& is >> min_ellipse&

reads *min_ellipse* from input stream *is*.

Requirement: The input operator is defined for *Point*.

#include <CGAL/IO/Window_stream.h>

CGAL::Window_stream&

CGAL::Window_stream& ws << min_ellipse

writes *min_ellipse* to window stream *ws*.

Requirement: The window stream output operator is defined for *Point* and *Ellipse*.

See Also

CGAL::Min_circle_2<Traits> page [2193](#)

CGAL::Min_ellipse_2_traits_2<K> page [2210](#)

MinEllipse2Traits page [2212](#)

Implementation

We implement the incremental algorithm of Welzl, with move-to-front heuristic [Wel91], using the primitives as described in [GS97a, GS97b]. The whole implementation is described in [GS98b].

If randomization is chosen, the creation time is almost always linear in the number of points. Access functions and predicates take constant time, inserting a point might take up to linear time, but substantially less than computing the new smallest enclosing ellipse from scratch. The clear operation and the check for validity each takes linear time.

Example

To illustrate the usage of *Min_ellipse_2<Traits>* and to show that randomization can be useful in certain cases, we give an example. The example also shows how the coefficients of the constructed ellipse can be accessed.

```
// file: examples/Min_ellipse_2/example_Min_ellipse_2.C

// includes
#include <cassert>
#include <CGAL/Cartesian.h>
#include <CGAL/Point_2.h>
#include <CGAL/Min_ellipse_2.h>
#include <CGAL/Min_ellipse_2_traits_2.h>
#include <CGAL/Gmpq.h>

// typedefs
typedef CGAL::Gmpq          NT;
typedef CGAL::Cartesian<NT> K;
typedef CGAL::Point_2<K>    Point;
typedef CGAL::Min_ellipse_2_traits_2<K> Traits;
typedef CGAL::Min_ellipse_2<Traits>    Min_ellipse;

// main
int
main( int, char**)
{
    int      n = 200;
    Point*   P = new Point[ n];

    for ( int i = 0; i < n; ++i)
P[ i] = Point( i % 2 ? i : -i , 0);
    // (0,0), (-1,0), (2,0), (-3,0)

    std::cout << "Computing ellipse (without randomization)...\n";
    std::cout.flush();
    Min_ellipse me1( P, P+n, false);    // very slow
    std::cout << "done." << std::endl;

    std::cout << "Computing ellipse (with randomization)...\n";
    std::cout.flush();
    Min_ellipse me2( P, P+n, true);     // fast
    std::cout << "done." << std::endl;
```



```

// because all input points are collinear, the ellipse is
// degenerate and equals a line segment; the ellipse has
// two support points
assert(me2.is_degenerate());
assert(me2.number_of_support_points()==2);

// prettyprinting
CGAL::set_pretty_mode( std::cout);
std::cout << me2;

// in general, the ellipse is not explicitly representable
// over the input number type NT; when you use the default
// traits class CGAL::Min_ellipse_2_traits_2<K>, you can
// get double approximations for the coefficients of the
// underlying conic curve. NOTE: this curve only exists
// in the nondegenerate case!

me2.insert(Point(0,1)); // resolves the degeneracy
assert(!me2.is_degenerate());

// get the coefficients
double r,s,t,u,v,w;
me2.ellipse().double_coefficients( r, s, t, u, v, w);
std::cout << "ellipse has the equation " <<
  r << " x^2 + " <<
  s << " y^2 + " <<
  t << " xy + " <<
  u << " x + " <<
  v << " y + " <<
  w << " = 0." << std::endl;

delete[] P;

return( 0);
}

// ===== EOF =====

```

CGAL::Min_ellipse_2_traits_2<K>

Definition

The class `Min_ellipse_2_traits_2<K>` is a traits class for `CGAL::Min_ellipse_2<Traits>` using the two-dimensional CGAL kernel.

```
#include <CGAL/Min_ellipse_2_traits_2.h>
```

Requirements

The template parameter K is a model for *Kernel*.

Is Model for the Concepts

MinEllipse2Traits page 2212

Types

$$Min_ellipse_2_traits_2<K>:: Point \quad \text{typedef to } K::Point_2.$$

Min_ellipse_2_traits_2 $\langle K \rangle$:: *Ellipse* internal type.

The `Ellipse` type provides the following access methods not required by the concept `MinEllipse2Traits`.

<i>bool</i>	<i>ellipse.is_circle()</i>	tests whether the ellipse is a circle.
-------------	----------------------------	----------------------------------------

```
void      ellipse.double_coefficients( double &r,
                                       double &s,
                                       double &t,
                                       double &u,
                                       double &v,
                                       double &w)
```

gives a double approximation of the ellipse's conic equation. If K is a Cartesian kernel, the ellipse is the set of all points (x, y) satisfying $rx^2 + sy^2 + txy + ux + vy + w = 0$. In the Homogeneous case, the ellipse is the set of points (hx, hy, hw) satisfying $r(hx)^2 + s(hy)^2 + t(hx)(hy) + u(hx)(hw) + v(hy)(hw) + w(hw)^2 = 0$.

Creation

Min_ellipse_2_traits_2<*K*> *traits*; default constructor.

Min_ellipse_2_traits_2<*K*> traits(*Min_ellipse_2_traits_2*<*K*>);

copy constructor.

See Also

CGAL::Min_ellipse_2<*Traits*> page [2203](#)

MinEllipse2Traits page [2212](#)

MinEllipse2Traits

Definition

This concept defines the requirements for traits classes of *CGAL::Min_ellipse_2<Traits>*.

Types

<i>MinEllipse2Traits:: Point</i>	The point type must provide default and copy constructor, assignment and equality test.
<i>MinEllipse2Traits:: Ellipse</i>	The ellipse type must fulfill the requirements listed below in the next section.

Variables

<i>Ellipse</i>	<i>ellipse;</i>	The current ellipse. This variable is maintained by the algorithm, the user should neither access nor modify it directly.
----------------	-----------------	---------------------------------------------------------------------------------------------------------------------------

Creation

Only default and copy constructor are required.

```
MinEllipse2Traits traits;

MinEllipse2Traits traits( Traits);
```

Has Models

CGAL::Min_ellipse_2_traits_2<K> page [2210](#)

See Also

CGAL::Min_ellipse_2<Traits> page [2203](#)

Ellipse Type (*Ellipse*)

Definition

An object of the class *Ellipse* is an ellipse in two-dimensional Euclidean plane \mathbb{E}_2 . Its boundary splits the plane into a bounded and an unbounded side. By definition, an empty *Ellipse* has no boundary and no bounded side, i.e. its unbounded side equals the whole plane \mathbb{E}_2 .

Types

Ellipse::Point Point type.

Creation

<i>void</i>	<i>ellipse.set()</i>	sets <i>ellipse</i> to the empty ellipse.
-------------	----------------------	-------------------------------------------

<i>void</i>	<i>ellipse.set(Point p)</i>	sets <i>ellipse</i> to the ellipse containing exactly $\{p\}$.
-------------	------------------------------	-----------------------------------------------------------------

void ellipse.set(Point p, Point q)

sets *ellipse* to the ellipse containing exactly the segment connecting *p* and *q*. The algorithm guarantees that *set* is never called with two equal points.

void ellipse.set(Point p, Point q, Point r)

sets *ellipse* to the smallest ellipse through *p,q,r*. The algorithm guarantees that *set* is never called with three collinear points.

void *ellipse.set*(Point *p*, Point *q*, Point *r*, Point *s*)

sets *ellipse* to the smallest ellipse through *p,q,r,s*. The algorithm guarantees that this ellipse exists.

void *ellipse.set*(Point *p*, Point *q*, Point *r*, Point *s*, Point *t*)

sets *ellipse* to the unique conic through *p,q,r,s,t*. The algorithm guarantees that this conic is an ellipse.

Predicates

bool ellipse.has_on_unbounded_side(Point p)

returns *true*, iff *p* lies properly outside of *ellipse*.

Each of the following predicates is only needed, if the corresponding predicate of *Min_ellipse_2* is used.

CGAL::Bounded_side

ellipse.bounded_side(Point p)

returns *CGAL::ON_BOUNDED_SIDE*, *CGAL::ON_BOUNDARY*, or *CGAL::ON_UNBOUNDED_SIDE* iff *p* lies properly inside, on the boundary, or properly outside of *ellipse*, resp.

bool ellipse.has_on_bounded_side(Point p)

returns *true*, iff *p* lies properly inside *ellipse*.

bool ellipse.has_on_boundary(Point p)

returns *true*, iff *p* lies on the boundary of *ellipse*.

bool ellipse.is_empty() returns *true*, iff *ellipse* is empty (this implies degeneracy).

bool ellipse.is_degenerate()

returns *true*, iff *ellipse* is degenerate, i.e. if *ellipse* is empty or equal to a single point.

I/O

The following I/O operators are only needed, if the corresponding I/O operators of *Min_ellipse_2* are used.

ostream& ostream& os << ellipse

writes *ellipse* to output stream *os*.

Window_stream& Window_stream& ws << ellipse

writes *ellipse* to window stream *ws*.

CGAL::min_rectangle_2

Definition

The function computes a minimum area enclosing rectangle $R(P)$ of a given convex point set P . Note that $R(P)$ is not necessarily axis-parallel, and it is in general not unique. The focus on convex sets is no restriction, since any rectangle enclosing P – as a convex set – contains the convex hull of P . For general point sets one has to compute the convex hull as a preprocessing step.

```
#include <CGAL/min_quadrilateral_2.h>
```

```
template < class ForwardIterator, class OutputIterator, class Traits >
OutputIterator      min_rectangle_2(
                    ForwardIterator points_begin,
                    ForwardIterator points_end,
                    OutputIterator o,
                    Traits& t = Default_traits)
```

computes a minimum area enclosing rectangle of the point set described by $[points_begin, points_end)$, writes its vertices (counterclockwise) to o , and returns the past-the-end iterator of this sequence.

If the input range is empty, o remains unchanged.

If the input range consists of one element only, this point is written to o four times.

Precondition: The points denoted by the range $[points_begin, points_end)$ form the boundary of a simple convex polygon P in counterclockwise orientation.

The geometric types and operations to be used for the computation are specified by the traits class parameter t . The parameter can be omitted, if *ForwardIterator* refers to a two-dimensional point type from one the CGAL Kernels. In this case, a default traits class (*Min_quadrilateral_default_traits_2*<*Kernel*>) is used.

Requirement:

1. If *Traits* is specified, it is a model for *MinQuadrilateralTraits_2* and the value type *VT* of *ForwardIterator* is *Traits::Point_2*. Otherwise *VT* is *CGAL::Point_2*<*Kernel*> for some kernel *Kernel*.
2. *OutputIterator* accepts *VT* as value type.

See Also

CGAL::min_parallelogram_2 page [2217](#)
CGAL::min_strip_2 page [2219](#)
MinQuadrilateralTraits_2 page [2225](#)
CGAL::Min_quadrilateral_default_traits_2<*Kernel*> page [2221](#)

Implementation

We use a rotating caliper algorithm [[Tou83](#)] with worst case running time linear in the number of input points.

Example

The following code generates a random convex polygon P with 20 vertices and computes the minimum enclosing rectangle of P .

```
#include <CGAL/Cartesian.h>
#include <CGAL/Polygon_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_convex_set_2.h>
#include <CGAL/min_quadrilateral_2.h>
#include <iostream>

struct Kernel : public CGAL::Cartesian<double> {};

typedef Kernel::Point_2          Point_2;
typedef Kernel::Line_2           Line_2;
typedef CGAL::Polygon_2<Kernel>  Polygon_2;
typedef CGAL::Random_points_in_square_2<Point_2> Generator;

int main()
{
    // build a random convex 20-gon p
    Polygon_2 p;
    CGAL::random_convex_set_2(20, std::back_inserter(p), Generator(1.0));
    std::cout << p << std::endl;

    // compute the minimal enclosing rectangle p_m of p
    Polygon_2 p_m;
    CGAL::min_rectangle_2(
        p.vertices_begin(), p.vertices_end(), std::back_inserter(p_m));
    std::cout << p_m << std::endl;

    return 0;
}
```


CGAL::min_parallelgram_2

Definition

The function computes a minimum area enclosing parallelogram $A(P)$ of a given convex point set P . Note that $R(P)$ is not necessarily axis-parallel, and it is in general not unique. The focus on convex sets is no restriction, since any parallelogram enclosing P – as a convex set – contains the convex hull of P . For general point sets one has to compute the convex hull as a preprocessing step.

```
#include <CGAL/min_quadrilateral_2.h>
```

```
template < class ForwardIterator, class OutputIterator, class Traits >
OutputIterator      min_parallelgram_2(
                    ForwardIterator points_begin,
                    ForwardIterator points_end,
                    OutputIterator o,
                    Traits& t = Default_traits)
```

computes a minimum area enclosing parallelogram of the point set described by $[points_begin, points_end)$, writes its vertices (counterclockwise) to o and returns the past-the-end iterator of this sequence. If the input range is empty, o remains unchanged.

If the input range consists of one element only, this point is written to o four times.

Precondition: The points denoted by the range $[points_begin, points_end)$ form the boundary of a simple convex polygon P in counterclockwise orientation.

The geometric types and operations to be used for the computation are specified by the traits class parameter t . The parameter can be omitted, if *ForwardIterator* refers to a two-dimensional point type from one the CGAL Kernels. In this case, a default traits class (*Min_quadrilateral_default_traits_2<Kernel>*) is used.

Requirement:

1. If *Traits* is specified, it is a model for *MinQuadrilateralTraits_2* and the value type *VT* of *ForwardIterator* is *Traits::Point_2*. Otherwise *VT* is *CGAL::Point_2<Kernel>* for some Kernel *Kernel*.
2. *OutputIterator* accepts *VT* as value type.

See Also

CGAL::min_rectangle_2 page [2215](#)
CGAL::min_strip_2 page [2219](#)
MinQuadrilateralTraits_2 page [2225](#)
CGAL::Min_quadrilateral_default_traits_2<Kernel> page [2221](#)

Implementation

We use a rotating caliper algorithm [[STV⁺95](#), [Vai90](#)] with worst case running time linear in the number of input points.

Example

The following code generates a random convex polygon P with 20 vertices and computes the minimum enclosing parallelogram of P .

```
#include <CGAL/Cartesian.h>
#include <CGAL/Polygon_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_convex_set_2.h>
#include <CGAL/min_quadrilateral_2.h>
#include <iostream>

struct Kernel : public CGAL::Cartesian<double> {};

typedef Kernel::Point_2          Point_2;
typedef Kernel::Line_2           Line_2;
typedef CGAL::Polygon_2<Kernel>  Polygon_2;
typedef CGAL::Random_points_in_square_2<Point_2> Generator;

int main()
{
    // build a random convex 20-gon p
    Polygon_2 p;
    CGAL::random_convex_set_2(20, std::back_inserter(p), Generator(1.0));
    std::cout << p << std::endl;

    // compute the minimal enclosing parallelogram p_m of p
    Polygon_2 p_m;
    CGAL::min_parallelogram_2(
        p.vertices_begin(), p.vertices_end(), std::back_inserter(p_m));
    std::cout << p_m << std::endl;

    return 0;
}
```

CGAL::min_strip_2

Definition

The function computes a minimum width enclosing strip $S(P)$ of a given convex point set P . A strip is the closed region bounded by two parallel lines in the plane. Note that $S(P)$ is not unique in general. The focus on convex sets is no restriction, since any parallelogram enclosing P – as a convex set – contains the convex hull of P . For general point sets one has to compute the convex hull as a preprocessing step.

```
#include <CGAL/min_quadrilateral_2.h>
```

```
template < class ForwardIterator, class OutputIterator, class Traits >
OutputIterator min_strip_2(
    ForwardIterator points_begin,
    ForwardIterator points_end,
    OutputIterator o,
    Traits& t = Default_traits)
```

computes a minimum enclosing strip of the point set described by $[points_begin, points_end)$, writes its two bounding lines to o and returns the past-the-end iterator of this sequence.

If the input range is empty or consists of one element only, o remains unchanged.

Precondition: The points denoted by the range $[points_begin, points_end)$ form the boundary of a simple convex polygon P in counterclockwise orientation.

The geometric types and operations to be used for the computation are specified by the traits class parameter t . The parameter can be omitted, if *ForwardIterator* refers to a two-dimensional point type from one of the CGAL Kernels. In this case, a default traits class (*Min_quadrilateral_default_traits_2*<*Kernel*>) is used.

Requirement:

1. If *Traits* is specified, it is a model for *MinQuadrilateralTraits_2* and the value type *VT* of *ForwardIterator* is *Traits::Point_2*. Otherwise *VT* is *CGAL::Point_2*<*Kernel*> for some kernel *Kernel*.
2. *OutputIterator* accepts *Traits::Line_2* as value type.

See Also

CGAL::min_rectangle_2 page [2215](#)
CGAL::min_parallelogram_2 page [2217](#)
MinQuadrilateralTraits_2 page [2225](#)
CGAL::Min_quadrilateral_default_traits_2<*Kernel*> page [2221](#)

Implementation

We use a rotating caliper algorithm [[Tou83](#)] with worst case running time linear in the number of input points.

Example

The following code generates a random convex polygon P with 20 vertices and computes the minimum enclosing strip of P .

```
#include <CGAL/Cartesian.h>
#include <CGAL/Polygon_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_convex_set_2.h>
#include <CGAL/min_quadrilateral_2.h>
#include <iostream>

struct Kernel : public CGAL::Cartesian<double> {};

typedef Kernel::Point_2          Point_2;
typedef Kernel::Line_2           Line_2;
typedef CGAL::Polygon_2<Kernel>  Polygon_2;
typedef CGAL::Random_points_in_square_2<Point_2>  Generator;

int main()
{
    // build a random convex 20-gon p
    Polygon_2 p;
    CGAL::random_convex_set_2(20, std::back_inserter(p), Generator(1.0));
    std::cout << p << std::endl;

    // compute the minimal enclosing strip p_m of p
    Line_2 p_m[2];
    CGAL::min_strip_2(p.vertices_begin(), p.vertices_end(), p_m);
    std::cout << p_m[0] << "\n" << p_m[1] << std::endl;

    return 0;
}
```

CGAL::Min_quadrilateral_default_traits_2<Kernel>

Definition

The class *Min_quadrilateral_default_traits_2<Kernel>* is a traits class for the functions *min_rectangle_2*, *min_parallelagram_2* and *min_strip_2* using a two-dimensional CGAL kernel.

Requirements

The template parameter *Kernel* is a model for *Kernel*.

```
#include <CGAL/Min_quadrilateral_traits_2.h>
```

Is Model for the Concepts

MinQuadrilateralTraits_2 page [2225](#)

Types

Min_quadrilateral_default_traits_2<Kernel>:: Point_2

Kernel::Point_2 page [408](#).

Min_quadrilateral_default_traits_2<Kernel>:: Vector_2

Kernel::Vector_2 page [424](#).

Min_quadrilateral_default_traits_2<Kernel>:: Direction_2

Kernel::Direction_2 page [351](#).

Min_quadrilateral_default_traits_2<Kernel>:: Line_2

Kernel::Line_2 page [399](#).

Min_quadrilateral_default_traits_2<Kernel>:: Rectangle_2

internal type.

Min_quadrilateral_default_traits_2<Kernel>:: Parallelogram_2

internal type.

Min_quadrilateral_default_traits_2<Kernel>:: Strip_2

internal type.

Predicates

Min_quadrilateral_default_traits_2<Kernel>:: Equal_2

Kernel::Equal_2 page [361](#).

Min_quadrilateral_default_traits_2<Kernel>:: Less_xy_2

Kernel::Less_xy_2 page ??.

Min_quadrilateral_default_traits_2<Kernel>:: Less_yx_2

Kernel::Less_yx_2 page ??.

Min_quadrilateral_default_traits_2<Kernel>:: Orientation_2

Kernel::Orientation_2 page [403](#).

Min_quadrilateral_default_traits_2<Kernel>:: Has_on_negative_side_2

Kernel::Has_on_negative_side_2 page ??.

Min_quadrilateral_default_traits_2<Kernel>:: Compare_angle_with_x_axis_2

Kernel::Compare_angle_with_x_axis_2 page ??.

Min_quadrilateral_default_traits_2<Kernel>:: Area_less_rectangle_2

AdaptableBinaryFunction class
op: $Rectangle_2 \times Rectangle_2 \rightarrow bool$.
op(*r1*,*r2*) returns true, iff the area of *r1* is strictly less
than the area of *r2*.

Min_quadrilateral_default_traits_2<Kernel>:: Area_less_parallelogram_2

AdaptableBinaryFunction class
op: $Parallelogram_2 \times Parallelogram_2 \rightarrow bool$.
op(*p1*,*p2*) returns true, iff the area of *p1* is strictly less
than the area of *p2*.

Min_quadilateral_default_traits_2<Kernel>:: Width_less_strip_2

AdaptableBinaryFunction class
op: $Strip_2 \times Strip_2 \rightarrow bool$.
op(*s1*,*s2*) returns true, iff the width of *s1* is strictly less than the width of *s2*.

Constructions

Min_quadilateral_default_traits_2<Kernel>:: Construct_vector_2

Kernel::Construct_vector_2 page ??.

Min_quadilateral_default_traits_2<Kernel>:: Construct_vector_from_direction_2

AdaptableFunctor
op: $Direction_2 \rightarrow Vector_2$.
op(*d*) returns a vector in direction *d*.

Min_quadilateral_default_traits_2<Kernel>:: Construct_perpendicular_vector_2

Kernel::Construct_perpendicular_vector_2 page ??.

Min_quadilateral_default_traits_2<Kernel>:: Construct_direction_2

Kernel::Construct_direction_2 page ??.

Min_quadilateral_default_traits_2<Kernel>:: Construct_opposite_direction_2

Kernel::Construct_opposite_direction_2 page ??.

Min_quadilateral_default_traits_2<Kernel>:: Construct_line_2

Kernel::Construct_line_2 page ??.

Min_quadilateral_default_traits_2<Kernel>:: Construct_rectangle_2

Function class
op: $Point_2 \times Direction_2 \times Point_2 \times Point_2 \times Point_2 \rightarrow Rectangle_2$.
If the points *p1*,*p2*,*p3*,*p4* form the boundary of a convex polygon (oriented counterclockwise), *op*(*p1*,*d*,*p2*,*p3*,*p4*) returns the rectangle with one of the points on each side and one sides parallel to *d*.

Min_quadrilateral_default_traits_2<Kernel>:: Construct_parallelagram_2

Function class

op: $Point_2 \times Direction_2 \times Point_2 \times Direction_2 \times Point_2 \times Point_2 \rightarrow Rectangle_2$.

If the points $p1, p2, p3, p4$ form the boundary of a convex polygon (oriented counterclockwise), *op*($p1, d1, p2, d2, p3, p4$) returns the parallelogram with one of the points on each side and one side parallel to each of $d1$ and $d2$.

Min_quadrilateral_default_traits_2<Kernel>:: Construct_strip_2

Function class

op: $Point_2 \times Direction_2 \times Point_2 \rightarrow Strip_2$.

op($p1, d, p2$) returns the strip bounded by the lines through $p1$ resp. $p2$ with direction d .

Operations

template < class OutputIterator >

OutputIterator t.copy_rectangle_vertices_2(const Rectangle_2& r, OutputIterator o) const

copies the four vertices of r in counterclockwise order to o .

template < class OutputIterator >

OutputIterator t.copy_parallelagram_vertices_2(const Parallelogram_2& p, OutputIterator o) const

copies the four vertices of p in counterclockwise order to o .

template < class OutputIterator >

OutputIterator t.copy_strip_lines_2(const Strip_2& s, OutputIterator o) const

copies the two lines bounding s to o .

Additionally, for each of the predicate and construction functor types listed above, there is a member function that requires no arguments and returns an instance of that functor type. The name of the member function is the uncapitalized name of the type returned with the suffix *_object* appended. For example, for the functor type *Construct_vector_2* the following member function exists:

Construct_vector_2

t.construct_vector_2_object() const

See Also

CGAL::min_rectangle_2 [page 2215](#)
CGAL::min_parallelagram_2 [page 2217](#)
CGAL::min_strip_2 [page 2219](#)

MinQuadrilateralTraits_2

Definition

The concept `MinQuadrilateralTraits_2` defines types and operations needed to compute minimum enclosing quadrilaterals of a planar point set using the functions `min_rectangle_2`, `min_parallelogram_2` and `min_strip_2`.

Types

<code>MinQuadrilateralTraits_2:: Point_2</code>	type for representing points.
<code>MinQuadrilateralTraits_2:: Vector_2</code>	type for representing vectors.
<code>MinQuadrilateralTraits_2:: Direction_2</code>	type for representing directions.
<code>MinQuadrilateralTraits_2:: Line_2</code>	type for representing lines.
<code>MinQuadrilateralTraits_2:: Rectangle_2</code>	type for representing (not necessarily axis-parallel) rectangles.
<code>MinQuadrilateralTraits_2:: Parallelogram_2</code>	type for representing parallelograms.
<code>MinQuadrilateralTraits_2:: Strip_2</code>	type for representing strips, that is the closed region bounded by two parallel lines.

Predicates

<code>MinQuadrilateralTraits_2:: Equal_2</code>	a model for <code>Kernel::Equal_2</code> page 361.
<code>MinQuadrilateralTraits_2:: Less_xy_2</code>	a model for <code>Kernel::Less_xy_2</code> page ??.
<code>MinQuadrilateralTraits_2:: Less_yx_2</code>	a model for <code>Kernel::Less_yx_2</code> page ??.
<code>MinQuadrilateralTraits_2:: Has_on_negative_side_2</code>	a model for <code>Kernel::Has_on_negative_side_2</code> page ??.
<code>MinQuadrilateralTraits_2:: Compare_angle_with_x_axis_2</code>	a model for <code>Kernel::Compare_angle_with_x_axis_2</code> page ??.
<code>MinQuadrilateralTraits_2:: Area_less_rectangle_2</code>	<p>AdaptableFunctor</p> <p>$op: Rectangle_2 \times Rectangle_2 \rightarrow bool.$</p> <p>$op(r1,r2)$ returns true, iff the area of $r1$ is strictly less than the area of $r2$.</p>

MinQuadrilateralTraits_2:: Area_less_parallelogram_2

AdaptableFunctor

op: Parallelogram_2 \times Parallelogram_2 \rightarrow bool.

op(p1,p2) returns true, iff the area of *p1* is strictly less than the area of *p2*.

MinQuadrilateralTraits_2:: Width_less_strip_2

AdaptableFunctor

op: Strip_2 \times Strip_2 \rightarrow bool.

op(s1,s2) returns true, iff the width of *s1* is strictly less than the width of *s2*.

The following type is used for expensive precondition checking only.

MinQuadrilateralTraits_2:: Orientation_2

a model for Kernel::Orientation_2 page 403.

Constructions

MinQuadrilateralTraits_2:: Construct_vector_2

a model for Kernel::Construct_vector_2 page ??.

MinQuadrilateralTraits_2:: Construct_vector_from_direction_2

AdaptableFunctor

op: Direction_2 \rightarrow Vector_2.

op(d) returns a vector in direction *d*.

MinQuadrilateralTraits_2:: Construct_perpendicular_vector_2

a model for Kernel::Construct_perpendicular_vector_2
page ??.

MinQuadrilateralTraits_2:: Construct_direction_2

a model for Kernel::Construct_direction_2 page ??.

MinQuadrilateralTraits_2:: Construct_opposite_direction_2

a model for Kernel::Construct_opposite_direction_2
page ??.

MinQuadrilateralTraits_2:: Construct_line_2

a model for Kernel::Construct_line_2 page ??.

MinQuadrilateralTraits_2:: Construct_rectangle_2

Function class

op: $Point_2 \times Direction_2 \times Point_2 \times Point_2 \times Point_2 \rightarrow Rectangle_2$.

If the points $p1, p2, p3, p4$ form the boundary of a convex polygon (oriented counterclockwise), *op*($p1, d, p2, p3, p4$) returns the rectangle with one of the points on each side and one sides parallel to d .

MinQuadrilateralTraits_2:: Construct_parallelogram_2

Function class

op: $Point_2 \times Direction_2 \times Point_2 \times Direction_2 \times Point_2 \times Point_2 \rightarrow Rectangle_2$.

If the points $p1, p2, p3, p4$ form the boundary of a convex polygon (oriented counterclockwise), *op*($p1, d1, p2, d2, p3, p4$) returns the parallelogram with one of the points on each side and one side parallel to each of $d1$ and $d2$.

MinQuadrilateralTraits_2:: Construct_strip_2

Function class

op: $Point_2 \times Direction_2 \times Point_2 \rightarrow Strip_2$.

op($p1, d, p2$) returns the strip bounded by the lines through $p1$ resp. $p2$ with direction d .

Operations

template < class OutputIterator >

OutputIterator t.copy_rectangle_vertices_2(const Rectangle_2& r, OutputIterator o) const

copies the four vertices of r in counterclockwise order to o .

template < class OutputIterator >

OutputIterator t.copy_parallelogram_vertices_2(const Parallelogram_2& p, OutputIterator o) const

copies the four vertices of p in counterclockwise order to o .

template < class OutputIterator >

OutputIterator t.copy_strip_lines_2(const Strip_2& s, OutputIterator o) const

copies the two lines bounding s to o .

Additionally, for each of the predicate and construction functor types listed above, there must exist a member function that requires no arguments and returns an instance of that functor type. The name of the member function is the uncapitalized name of the type returned with the suffix *_object* appended. For example, for the functor type *Construct_vector_2* the following member function must exist:

Construct_vector_2

t.construct_vector_2_object() const

Has Models

CGAL::Min_quadrilateral_default_traits_2<Kernel> page [2221](#)

See Also

CGAL::min_rectangle_2 page [2215](#)

CGAL::min_parallelogram_2 page [2217](#)

CGAL::min_strip_2 page [2219](#)

CGAL::rectangular_p_center_2

Definition

The function *rectangular_p_center_2* computes rectilinear p -centers of a planar point set, i.e. a set of p points such that the maximum minimal L_∞ -distance between both sets is minimized.

More formally the problem can be defined as follows.

Given a finite set \mathcal{P} of points, compute a point set \mathcal{C} with $|\mathcal{C}| \leq p$ such that the p -radius of \mathcal{P} ,

$$\text{rad}_p(\mathcal{P}) := \max_{P \in \mathcal{P}} \min_{Q \in \mathcal{C}} \|P - Q\|_\infty$$

is minimized. We can interpret \mathcal{C} as the best approximation (with respect to the given metric) for \mathcal{P} with at most p points.

```
#include <CGAL/rectangular_p_center_2.h>
```

```
template < class ForwardIterator, class OutputIterator, class FT, class Traits >
OutputIterator rectangular_p_center_2(
    ForwardIterator f,
    ForwardIterator l,
    OutputIterator o,
    FT& r,
    int p,
    Traits t = Default_traits)
```

computes rectilinear p -centers for the point set described by the range $[f, l)$, sets r to the corresponding p -radius, writes the at most p center points to o and returns the past-the-end iterator of this sequence.

Precondition:

1. The range $[f, l)$ is not empty.
2. $2 \leq p \leq 4$.

The geometric types and operations to be used for the computation are specified by the traits class parameter t . This parameter can be omitted if *ForwardIterator* refers to a point type from the 2D-Kernel. In this case, a default traits class (*Rectangular_p_center_default_traits_2*< R >) is used.

Requirement:

1. *Either:* (if no traits parameter is given) Value type of *ForwardIterator* is *CGAL::Point_2*< R > for some representation class R and FT is equivalent to $R::FT$,
2. *Or:* (if a traits parameter is specified) *Traits* is a model for *RectangularPCenterTraits_2*.
3. *OutputIterator* accepts the value type of *ForwardIterator* as value type.

See Also

RectangularPCenterTraits_2 page [2235](#)
CGAL::Rectangular_p_center_default_traits_2<R> page [2232](#)
CGAL::sorted_matrix_search page [2328](#)

Implementation

The runtime is linear for $p \in \{2, 3\}$ and $O(n \cdot \log n)$ for $p = 4$ where n is the number of input points. These runtimes are worst case optimal. The 3-center algorithm uses a prune-and-search technique described in [[Hof99](#)]. The 4-center implementation uses sorted matrix search [[FJ83](#), [FJ84](#)] and fast algorithms for piercing rectangles [[SW96](#)].

Example

The following code generates a random set of ten points and computes its two-centers.

```
#include <CGAL/Cartesian.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/rectangular_p_center_2.h>
#include <CGAL/IO/Ostream_iterator.h>
#include <CGAL/algorithm.h>
#include <iostream>
#include <algorithm>
#include <vector>

typedef double FT;

struct Kernel : public CGAL::Cartesian<FT> {};

typedef Kernel::Point_2 Point;
typedef std::vector<Point> Cont;
typedef CGAL::Random_points_in_square_2<Point> Generator;
typedef CGAL::Ostream_iterator<Point, std::ostream> OIterator;

int main()
{
    int n = 10;
    int p = 2;
    OIterator cout_ip(std::cout);
    CGAL::set_pretty_mode(std::cout);

    Cont points;
    CGAL::copy_n(Generator(1), n, std::back_inserter(points));
    std::cout << "Generated Point Set:\n";
    std::copy(points.begin(), points.end(), cout_ip);

    FT p_radius;
    std::cout << "\n\n" << p << "-centers:\n";
    CGAL::rectangular_p_center_2(
        points.begin(), points.end(), cout_ip, p_radius, 3);
    std::cout << "\n\n" << p << "-radius = " << p_radius << std::endl;
```

```
    return 0;  
}
```

CGAL::Rectangular_p_center_default_traits_2<R>

Definition

The class *Rectangular_p_center_default_traits_2<R>* defines types and operations needed to compute rectilinear *p*-centers of a planar point set using the function *rectangular_p_center_2*.

Requirements

The template parameter *R* is a model for *Kernel*.

Is Model for the Concepts

RectangularPCenterTraits_2 page [2235](#)

Types

Rectangular_p_center_default_traits_2<R>:: FT

typedef to *R::FT*.

Rectangular_p_center_default_traits_2<R>:: Point_2

typedef to *R::Point_2*.

Rectangular_p_center_default_traits_2<R>:: Iso_rectangle_2

typedef to *R::Iso_rectangle_2*.

Rectangular_p_center_default_traits_2<R>:: Less_x_2

typedef to *R::Less_x_2*.

Rectangular_p_center_default_traits_2<R>:: Less_y_2

typedef to *R::Less_y_2*.

Rectangular_p_center_default_traits_2<R>:: Construct_vertex_2

typedef to *R::Construct_vertex_2*.

Rectangular_p_center_default_traits_2<R>:: Construct_iso_rectangle_2

typedef to *R::Construct_iso_rectangle_2*.

Rectangular_p_center_default_traits_2<R>:: Signed_x_distance_2

adaptable binary function class: $Point_2 \times Point_2 \rightarrow FT$
returns the signed distance of two points' x-coordinates.

Rectangular_p_center_default_traits_2<R>:: Signed_y_distance_2

adaptable binary function class: $Point_2 \times Point_2 \rightarrow FT$
returns the signed distance of two points' y-coordinates.

Rectangular_p_center_default_traits_2<R>:: Infinity_distance_2

adaptable binary function class: $Point_2 \times Point_2 \rightarrow FT$
returns the $\|\cdot\|_\infty$ distance of two points.

Rectangular_p_center_default_traits_2<R>:: Signed_infinity_distance_2

adaptable binary function class: $Point_2 \times Point_2 \rightarrow FT$
returns the signed $\|\cdot\|_\infty$ distance of two points.

Rectangular_p_center_default_traits_2<R>:: Construct_point_2_below_left_implicit_point_2

3-argument function class: $Point_2 \times Point_2 \times FT \rightarrow Point_2$. For arguments (p, q, r) it returns the lower-left corner of the iso-oriented square with sidelength r and upper-right corner at the intersection of the vertical line through p and the horizontal line through q .

Rectangular_p_center_default_traits_2<R>:: Construct_point_2_below_right_implicit_point_2

3-argument function class: $Point_2 \times Point_2 \times FT \rightarrow Point_2$. For arguments (p, q, r) it returns the lower-right corner of the iso-oriented square with sidelength r and upper-left corner at the intersection of the vertical line through p and the horizontal line through q .

Rectangular_p_center_default_traits_2<R>:: Construct_point_2_above_right_implicit_point_2

3-argument function class: $Point_2 \times Point_2 \times FT \rightarrow Point_2$. For arguments (p, q, r) it returns the upper-right corner of the iso-oriented square with sidelength r and lower-left corner at the intersection of the vertical line through p and the horizontal line through q .

Rectangular_p_center_default_traits_2<R>:: Construct_point_2_above_left_implicit_point_2

3-argument function class: $Point_2 \times Point_2 \times FT \rightarrow Point_2$. For arguments (p, q, r) it returns the upper-left corner of the iso-oriented square with sidelength r and lower-right corner at the intersection of the vertical line through p and the horizontal line through q .

Operations

For every function class listed above there is a member function to fetch the corresponding function object.

Inf_distance_2 *t.inf_distance_2_object() const*

Signed_inf_distance_2

t.signed_inf_distance_2_object() const

Construct_vertex_2 *t.construct_vertex_2_object() const*

Construct_iso_rectangle_2

t.construct_iso_rectangle_2_object() const

Construct_iso_rectangle_2_below_left_point_2

t.construct_iso_rectangle_2_below_left_point_2_object() const

Construct_iso_rectangle_2_above_left_point_2

t.construct_iso_rectangle_2_above_left_point_2_object() const

Construct_iso_rectangle_2_below_right_point_2

t.construct_iso_rectangle_2_below_right_point_2_object() const

Construct_iso_rectangle_2_above_right_point_2

t.construct_iso_rectangle_2_above_right_point_2_object() const

See Also

CGAL::rectangular_p_center_2 page [2229](#)

RectangularPCenterTraits_2

Definition

The concept RectangularPCenterTraits_2 defines types and operations needed to compute rectilinear p -centers of a planar point set using the function *rectangular_p_center_2*.

Types

RectangularPCenterTraits_2:: FT model for FieldNumberType page 2530.

RectangularPCenterTraits_2:: Point_2 model for Kernel::Point_2 page 408.

RectangularPCenterTraits_2:: Iso_rectangle_2
model for Kernel::Iso_rectangle_2 page ??.

RectangularPCenterTraits_2:: Less_x_2 model for Kernel::Less_x_2 page ??.

RectangularPCenterTraits_2:: Less_y_2 model for Kernel::Less_y_2 page ??.

RectangularPCenterTraits_2:: Construct_vertex_2
model for Kernel::Construct_vertex_2 page ??.

RectangularPCenterTraits_2:: Construct_iso_rectangle_2
model for Kernel::Construct_iso_rectangle_2 page ??.

RectangularPCenterTraits_2:: Signed_x_distance_2
adaptable binary function class: $Point_2 \times Point_2 \rightarrow FT$
returns the signed distance of two points' x -coordinates.

RectangularPCenterTraits_2:: Signed_y_distance_2
adaptable binary function class: $Point_2 \times Point_2 \rightarrow FT$
returns the signed distance of two points' y -coordinates.

RectangularPCenterTraits_2:: Infinity_distance_2
adaptable binary function class: $Point_2 \times Point_2 \rightarrow FT$
returns the $|| \cdot ||_{\infty}$ distance of two points.

RectangularPCenterTraits_2:: Signed_infinity_distance_2

adaptable binary function class: $Point_2 \times Point_2 \rightarrow FT$
returns the signed $\| \cdot \|_\infty$ distance of two points.

RectangularPCenterTraits_2:: Construct_point_2_below_left_implicit_point_2

3-argument function class: $Point_2 \times Point_2 \times FT \rightarrow Point_2$. For arguments (p, q, r) it returns the lower-left corner of the iso-oriented square with sidelength r and upper-right corner at the intersection of the vertical line through p and the horizontal line through q .

RectangularPCenterTraits_2:: Construct_point_2_below_right_implicit_point_2

3-argument function class: $Point_2 \times Point_2 \times FT \rightarrow Point_2$. For arguments (p, q, r) it returns the lower-right corner of the iso-oriented square with sidelength r and upper-left corner at the intersection of the vertical line through p and the horizontal line through q .

RectangularPCenterTraits_2:: Construct_point_2_above_right_implicit_point_2

3-argument function class: $Point_2 \times Point_2 \times FT \rightarrow Point_2$. For arguments (p, q, r) it returns the upper-right corner of the iso-oriented square with sidelength r and lower-left corner at the intersection of the vertical line through p and the horizontal line through q .

RectangularPCenterTraits_2:: Construct_point_2_above_left_implicit_point_2

3-argument function class: $Point_2 \times Point_2 \times FT \rightarrow Point_2$. For arguments (p, q, r) it returns the upper-left corner of the iso-oriented square with sidelength r and lower-right corner at the intersection of the vertical line through p and the horizontal line through q .

Operations

For every function class listed above there is a member function to fetch the corresponding function object.

```
Inf_distance_2          t.inf_distance_2_object() const
Signed_inf_distance_2
Construct_vertex_2      t.signed_inf_distance_2_object() const
Construct_iso_rectangle_2 t.construct_vertex_2_object() const
Construct_iso_rectangle_2
                        t.construct_iso_rectangle_2_object() const
```

Construct_iso_rectangle_2_below_left_point_2

t.construct_iso_rectangle_2_below_left_point_2_object() const

Construct_iso_rectangle_2_above_left_point_2

t.construct_iso_rectangle_2_above_left_point_2_object() const

Construct_iso_rectangle_2_below_right_point_2

t.construct_iso_rectangle_2_below_right_point_2_object() const

Construct_iso_rectangle_2_above_right_point_2

t.construct_iso_rectangle_2_above_right_point_2_object() const

Has Models

CGAL::Rectangular_p_center_default_traits_2<R> page [2232](#)

See Also

CGAL::rectangular_p_center_2 page [2229](#)

CGAL::Min_sphere_d<Traits>

Definition

An object of the class *Min_sphere_d<Traits>* is the unique sphere of smallest volume enclosing a finite (multi)set of points in d -dimensional Euclidean space \mathbb{E}_d . For a set P we denote by $ms(P)$ the smallest sphere that contains all points of P . $ms(P)$ can be degenerate, i.e. $ms(P) = \emptyset$ if $P = \emptyset$ and $ms(P) = \{p\}$ if $P = \{p\}$.

An inclusion-minimal subset S of P with $ms(S) = ms(P)$ is called a *support set*, the points in S are the *support points*. A support set has size at most $d + 1$, and all its points lie on the boundary of $ms(P)$. In general, neither the support set nor its size are unique.

The algorithm computes a support set S which remains fixed until the next insert or clear operation.

Please note: This class is (almost) obsolete. The class *CGAL::Min_sphere_of_spheres_d<Traits>* solves a more general problem and is faster than *Min_sphere_d<Traits>* even if used only for points as input. Most importantly, *CGAL::Min_sphere_of_spheres_d<Traits>* has a specialized implementation for floating-point arithmetic which ensures correct results in a large number of cases (including highly degenerate ones). In contrast, *Min_sphere_d<Traits>* is not reliable under floating-point computations. The only advantage of *Min_sphere_d<Traits>* over *CGAL::Min_sphere_of_spheres_d<Traits>* is that the former can deal with points in homogeneous coordinates, in which case the algorithm is division-free. Thus, *Min_sphere_d<Traits>* might still be an option in case your input number type cannot (efficiently) divide.

```
#include <CGAL/Min_sphere_d.h>
```

Requirements

The class *Min_sphere_d<Traits>* expects a model of the concept *OptimisationDTraits* as its template argument. We provide the models *CGAL::Optimisation_d_traits_2*, *CGAL::Optimisation_d_traits_3* and *CGAL::Optimisation_d_traits_d* for two-, three-, and d -dimensional points respectively.

Types

Min_sphere_d<Traits>:: Traits

Min_sphere_d<Traits>:: FT typedef to *Traits::FT*.

Min_sphere_d<Traits>:: Point typedef to *Traits::Point*.

Min_sphere_d<Traits>:: Point_iterator non-mutable model of the STL concept *BidirectionalIterator* with value type *Point*. Used to access the points used to build the smallest enclosing sphere.

Min_sphere_d<Traits>:: Support_point_iterator non-mutable model of the STL concept *BidirectionalIterator* with value type *Point*. Used to access the support points defining the smallest enclosing sphere.

Creation

Min_sphere_d<Traits> *min_sphere*(Traits traits = Traits());

creates a variable of type *Min_sphere_d*<Traits> and initializes it to *ms*(\emptyset). If the traits parameter is not supplied, the class *Traits* must provide a default constructor.

template < class InputIterator >

Min_sphere_d<Traits> *min_sphere*(InputIterator first, InputIterator last, Traits traits = Traits());

creates a variable *min_sphere* of type *Min_sphere_d*<Traits>. It is initialized to *ms*(*P*) with *P* being the set of points in the range [*first*,*last*).

Requirement: The value type of *first* and *last* is *Point*. If the traits parameter is not supplied, the class *Traits* must provide a default constructor.

Precondition: All points have the same dimension.

int *min_sphere.number_of_points*()

returns the number of points of *min_sphere*, i.e. $|P|$.

int *min_sphere.number_of_support_points*()

returns the number of support points of *min_sphere*, i.e. $|S|$.

Point_iterator *min_sphere.points_begin*()

returns an iterator referring to the first point of *min_sphere*.

Point_iterator *min_sphere.points_end*()

returns the corresponding past-the-end iterator.

Support_point_iterator *min_sphere.support_points_begin*()

returns an iterator referring to the first support point of *min_sphere*.

Support_point_iterator *min_sphere.support_points_end*()

returns the corresponding past-the-end iterator.

int *min_sphere.ambient_dimension*()

returns the dimension of the points in *P*. If *min_sphere* is empty, the ambient dimension is -1 .

Point *min_sphere.center*()

returns the center of *min_sphere*.

Precondition: *min_sphere* is not empty.

FT *min_sphere.squared_radius*()

returns the squared radius of *min_sphere*.

Precondition: *min_sphere* is not empty.

Predicates

By definition, an empty *Min_sphere.d*<Traits> has no boundary and no bounded side, i.e. its unbounded side equals the whole space \mathbb{E}_d .

Bounded_side *min_sphere.bounded_side(Point p)*

returns *CGAL::ON_BOUNDED_SIDE*, *CGAL::ON_BOUNDARY*, or *CGAL::ON_UNBOUNDED_SIDE* iff *p* lies properly inside, on the boundary, or properly outside of *min_sphere*, resp.
Precondition: if *min_sphere* is not empty, the dimension of *p* equals *ambient_dimension()*.

bool *min_sphere.has_on_bounded_side(Point p)*

returns *true*, iff *p* lies properly inside *min_sphere*.
Precondition: if *min_sphere* is not empty, the dimension of *p* equals *ambient_dimension()*.

bool *min_sphere.has_on_boundary(Point p)*

returns *true*, iff *p* lies on the boundary of *min_sphere*.
Precondition: if *min_sphere* is not empty, the dimension of *p* equals *ambient_dimension()*.

bool *min_sphere.has_on_unbounded_side(Point p)*

returns *true*, iff *p* lies properly outside of *min_sphere*.
Precondition: if *min_sphere* is not empty, the dimension of *p* equals *ambient_dimension()*.

bool *min_sphere.is_empty()*

returns *true*, iff *min_sphere* is empty (this implies degeneracy).

bool *min_sphere.is_degenerate()*

returns *true*, iff *min_sphere* is degenerate, i.e. if *min_sphere* is empty or equal to a single point, equivalently if the number of support points is less than 2.

Modifiers

void *min_sphere.clear()*

resets *min_sphere* to *ms(0)*.

template < class InputIterator >

void *min_sphere.set(InputIterator first, InputIterator last)*

sets *min_sphere* to the $ms(P)$, where P is the set of points in the range $[first, last)$.
Requirement: The value type of *first* and *last* is *Point*.
Precondition: All points have the same dimension.

void *min_sphere.insert(Point p)*

inserts p into *min_sphere*. If p lies inside the current sphere, this is a constant-time operation, otherwise it might take longer, but usually substantially less than recomputing the smallest enclosing sphere from scratch.
Precondition: The dimension of p equals *ambient_dimension()* if *min_sphere* is not empty.

template < class InputIterator >
void *min_sphere.insert(InputIterator first, InputIterator last)*

inserts the points in the range $[first, last)$ into *min_sphere* and recomputes the smallest enclosing sphere, by calling *insert* for all points in the range.
Requirement: The value type of *first* and *last* is *Point*.
Precondition: All points have the same dimension. If *min_sphere* is not empty, this dimension must be equal to *ambient_dimension()*.

Validity Check

An object *min_sphere* is valid, iff

- *min_sphere* contains all points of its defining set P ,
- *min_sphere* is the smallest sphere containing its support set S , and
- S is minimal, i.e. no support point is redundant.

Note: Under inexact arithmetic, the result of the validation is not reliable, because the checker itself can suffer from numerical problems.

bool *min_sphere.is_valid(bool verbose = false, int level = 0)*

returns *true*, iff *min_sphere* is valid. If *verbose* is *true*, some messages concerning the performed checks are written to standard error stream. The second parameter *level* is not used, we provide it only for consistency with interfaces of other classes.

Miscellaneous

`const Traits& min_sphere.traits()`

returns a const reference to the traits class object.

I/O

`std::ostream& os << min_sphere`

writes *min_sphere* to output stream *os*.

Requirement: The output operator is defined for *Point*.

`std::istream& is >> min_sphere&`

reads *min_sphere* from input stream *is*.

Requirement: The input operator is defined for *Point*.

See Also

[CGAL::Optimisation_d_traits_2<K,ET,NT>](#) [page 2279](#)
[CGAL::Optimisation_d_traits_3<K,ET,NT>](#) [page 2281](#)
[CGAL::Optimisation_d_traits_d<K,ET,NT>](#) [page 2283](#)
[OptimisationDTraits](#) [page 2285](#)
[CGAL::Min_circle_2<Traits>](#) [page 2193](#)
[CGAL::Min_sphere_of_spheres_d<Traits>](#) [page 2251](#)
[CGAL::Min_annulus_d<Traits>](#) [page 2244](#)

Implementation

We implement the algorithm of Welzl with move-to-front heuristic [Wel91] for small point sets, combined with a new efficient method for large sets, which is particularly tuned for moderately large dimension ($d \leq 20$) [Gär99]. The creation time is almost always linear in the number of points. Access functions and predicates take constant time, inserting a point might take up to linear time, but substantially less than computing the new smallest enclosing sphere from scratch. The clear operation and the check for validity each take linear time.

Example

```
#include <CGAL/Cartesian_d.h>
#include <iostream>
#include <cstdlib>
#include <CGAL/Random.h>
#include <CGAL/Optimisation_d_traits_d.h>
#include <CGAL/Min_sphere_d.h>

typedef CGAL::Cartesian_d<double>          K;
typedef CGAL::Optimisation_d_traits_d<K>   Traits;
typedef CGAL::Min_sphere_d<Traits>        Min_sphere;
typedef K::Point_d                         Point;
```

```

const int n = 10;                // number of points
const int d = 5;                // dimension of points

int main ()
{
    Point      P[n];             // n points
    double     coord[d];         // d coordinates
    CGAL::Random r;              // random number generator

    for (int i=0; i<n; ++i) {
        for (int j=0; j<d; ++j)
            coord[j] = r.get_double();
        P[i] = Point(d, coord, coord+d); // random point
    }

    Min_sphere ms (P, P+n);      // smallest enclosing sphere

    CGAL::set_pretty_mode (std::cout);
    std::cout << ms;            // output the sphere

    return 0;
}

```

CGAL::Min_annulus_d<Traits>

Definition

An object of the class *Min_annulus_d<Traits>* is the unique annulus (region between two concentric spheres with radii r and R , $r \leq R$) enclosing a finite set of points in d -dimensional Euclidean space \mathbb{E}_d , where the difference $R^2 - r^2$ is minimal. For a point set P we denote by $ma(P)$ the smallest annulus that contains all points of P . Note that $ma(P)$ can be degenerate, i.e. $ma(P) = \emptyset$ if $P = \emptyset$ and $ma(P) = \{p\}$ if $P = \{p\}$.

An inclusion-minimal subset S of P with $ma(S) = ma(P)$ is called a *support set*, the points in S are the *support points*. A support set has size at most $d + 2$, and all its points lie on the boundary of $ma(P)$. In general, the support set is not necessarily unique.

The underlying algorithm can cope with all kinds of input, e.g. P may be empty or points may occur more than once. The algorithm computes a support set S which remains fixed until the next set, insert, or clear operation.

```
#include <CGAL/Min_annulus_d.h>
```

Requirements

The template parameter *Traits* is a model for *OptimisationDTraits*.

We provide the models *Optimisation_d_traits_2*, *Optimisation_d_traits_3*, and *Optimisation_d_traits_d* using the two-, three-, and d -dimensional CGAL kernel, respectively.

Types

Min_annulus_d<Traits>:: Point typedef to *Traits::Point_d*. Point type used to represent the input points.

Min_annulus_d<Traits>:: FT typedef to *Traits::FT*. Number type used to return the squared radii of the smallest enclosing annulus.

Min_annulus_d<Traits>:: ET typedef to *Traits::ET*. Number type used to do the exact computations in the underlying solver for quadratic programs (cf. **Implementation**).

Min_annulus_d<Traits>:: Point_iterator

non-mutable model of the STL concept *RandomAccessIterator* with value type *Point*. Used to access the points of the smallest enclosing annulus.

Min_annulus_d<Traits>:: Support_point_iterator

non-mutable model of the STL concept *RandomAccessIterator* with value type *Point*. Used to access the support points of the smallest enclosing annulus.

Min_annulus_d<Traits>:: Inner_support_point_iterator

non-mutable model of the STL concept *RandomAccessIterator* with value type *Point*. Used to access the inner support points of the smallest enclosing annulus.

Min_annulus_d<Traits>:: Outer_support_point_iterator

non-mutable model of the STL concept *RandomAccessIterator* with value type *Point*. Used to access the outer support points of the smallest enclosing annulus.

Min_annulus_d<Traits>:: Coordinate_iterator

non-mutable model of the STL concept *RandomAccessIterator* with value type *ET*. Used to access the coordinates of the center of the smallest enclosing annulus.

Creation

```
Min_annulus_d<Traits> min_annulus( Traits traits = Traits(),  
    int verbose = 0,  
    std::ostream& stream = std::cout)  
  
    initializes min_annulus to ma( $\emptyset$ ).
```

```
template < class InputIterator >  
Min_annulus_d<Traits> min_annulus( InputIterator first,  
    InputIterator last,  
    Traits traits = Traits(),  
    int verbose = 0,  
    std::ostream& stream = std::cout)  
  
    initializes min_annulus to ma( $P$ ) with  $P$  being the set of points in  
    the range  $[first, last)$ .  
Requirement: The value type of InputIterator is Point.  
Precondition: All points have the same dimension.
```

Access Functions

```
int min_annulus.ambient_dimension()  
  
    returns the dimension of the points in  $P$ . If min_annulus is empty,  
    the ambient dimension is  $-1$ .
```

```
int min_annulus.number_of_points()  
  
    returns the number of points of min_annulus, i.e.  $|P|$ .
```

```
int min_annulus.number_of_support_points()  
  
    returns the number of support points of min_annulus, i.e.  $|S|$ .
```

<i>int</i>	<i>min_annulus.number_of_inner_support_points()</i>	returns the number of support points of <i>min_annulus</i> which lie on the inner sphere.
<i>int</i>	<i>min_annulus.number_of_outer_support_points()</i>	returns the number of support points of <i>min_annulus</i> which lie on the outer sphere.
<i>Point_iterator</i>	<i>min_annulus.points_begin()</i>	returns an iterator referring to the first point of <i>min_annulus</i> .
<i>Point_iterator</i>	<i>min_annulus.points_end()</i>	returns the corresponding past-the-end iterator.
<i>Support_point_iterator</i>	<i>min_annulus.support_points_begin()</i>	returns an iterator referring to the first support point of <i>min_annulus</i> . <i>Precondition:</i> <i>ma(P)</i> is not degenerate, i.e., <i>number_of_support_points()</i> is at least one.
<i>Support_point_iterator</i>	<i>min_annulus.support_points_end()</i>	returns the corresponding past-the-end iterator. <i>Precondition:</i> <i>ma(P)</i> is not degenerate, i.e., <i>number_of_support_points()</i> is at least one.
<i>Inner_support_point_iterator</i>	<i>min_annulus.inner_support_points_begin()</i>	returns an iterator referring to the first inner support point of <i>min_annulus</i> .
<i>Inner_support_point_iterator</i>	<i>min_annulus.inner_support_points_end()</i>	returns the corresponding past-the-end iterator.
<i>Outer_support_point_iterator</i>	<i>min_annulus.outer_support_points_begin()</i>	returns an iterator referring to the first outer support point of <i>min_annulus</i> .
<i>Outer_support_point_iterator</i>	<i>min_annulus.outer_support_points_end()</i>	returns the corresponding past-the-end iterator.
<i>Point</i>	<i>min_annulus.center()</i>	returns the center of <i>min_annulus</i> . <i>Requirement:</i> An implicit conversion from <i>ET</i> to <i>RT</i> is available. <i>Precondition:</i> <i>min_annulus</i> is not empty.
<i>FT</i>	<i>min_annulus.squared_inner_radius()</i>	returns the squared inner radius of <i>min_annulus</i> . <i>Requirement:</i> An implicit conversion from <i>ET</i> to <i>RT</i> is available. <i>Precondition:</i> <i>min_annulus</i> is not empty.

FT *min_annulus.squared_outer_radius()*

returns the squared outer radius of *min_annulus*.
Requirement: An implicit conversion from *ET* to *RT* is available.
Precondition: *min_annulus* is not empty.

Coordinate_iterator
min_annulus.center_coordinates_begin()

returns an iterator referring to the first coordinate of the center of *min_annulus*.
Note: The coordinates have a rational representation, i.e. the first d elements of the iterator range are the numerators and the $(d+1)$ -st element is the common denominator.

Coordinate_iterator
min_annulus.center_coordinates_end()

returns the corresponding past-the-end iterator.

ET *min_annulus.squared_inner_radius_numerator()*

returns the numerator of the squared inner radius of *min_annulus*.

ET *min_annulus.squared_outer_radius_numerator()*

returns the numerator of the squared outer radius of *min_annulus*.

ET *min_annulus.squared_radii_denominator()*

returns the denominator of the squared radii of *min_annulus*.

Predicates

The bounded area of the smallest enclosing annulus lies between the inner and the outer sphere. The boundary is the union of both spheres. By definition, an empty annulus has no boundary and no bounded side, i.e. its unbounded side equals the whole space \mathbb{E}_d .

CGAL::Bounded_side

min_annulus.bounded_side(Point p)

returns *CGAL::ON_BOUNDED_SIDE*, *CGAL::ON_BOUNDARY*, or *CGAL::ON_UNBOUNDED_SIDE* iff p lies properly inside, on the boundary, or properly outside of *min_annulus*, resp.
Precondition: The dimension of p equals *min_annulus.ambient_dimension()* if *min_annulus* is not empty.

bool *min_annulus.has_on_bounded_side(Point p)*

returns *true*, iff p lies properly inside *min_annulus*.
Precondition: The dimension of p equals *min_annulus.ambient_dimension()* if *min_annulus* is not empty.

bool *min_annulus.has_on_boundary(Point p)*

returns *true*, iff *p* lies on the boundary of *min_annulus*.
Precondition: The dimension of *p* equals *min_annulus.ambient_dimension()* if *min_annulus* is not empty.

bool *min_annulus.has_on_unbounded_side(Point p)*

returns *true*, iff *p* lies properly outside of *min_annulus*.
Precondition: The dimension of *p* equals *min_annulus.ambient_dimension()* if *min_annulus* is not empty.

bool *min_annulus.is_empty()* returns *true*, iff *min_annulus* is empty (this implies degeneracy).

bool *min_annulus.is_degenerate()*

returns *true*, iff *min_annulus* is degenerate, i.e. if *min_annulus* is empty or equal to a single point.

Modifiers

void *min_annulus.clear()* resets *min_annulus* to *ma(0)*.

template < class InputIterator >
void *min_annulus.set(InputIterator first, InputIterator last)*

sets *min_annulus* to *ma(P)*, where *P* is the set of points in the range *[first,last)*.
Requirement: The value type of *InputIterator* is *Point*.
Precondition: All points have the same dimension.

void *min_annulus.insert(Point p)*

inserts *p* into *min_annulus*.
Precondition: The dimension of *p* equals *min_annulus.ambient_dimension()* if *min_annulus* is not empty.

template < class InputIterator >
void *min_annulus.insert(InputIterator first, InputIterator last)*

inserts the points in the range *[first,last)* into *min_annulus* and recomputes the smallest enclosing annulus.
Requirement: The value type of *InputIterator* is *Point*.
Precondition: All points have the same dimension. If *min_annulus* is not empty, this dimension must be equal to *min_annulus.ambient_dimension()*.

Validity Check

An object *min_annulus* is valid, iff

- *min_annulus* contains all points of its defining set *P*,
- *min_annulus* is the smallest annulus containing its support set *S*, and
- *S* is minimal, i.e. no support point is redundant.

Note: In this release only the first item is considered by the validity check.

bool min_annulus.is_valid(bool verbose = false, int level = 0)

returns *true*, iff *min_annulus* is valid. If *verbose* is *true*, some messages concerning the performed checks are written to standard error stream. The second parameter *level* is not used, we provide it only for consistency with interfaces of other classes.

Miscellaneous

const Traits&

min_annulus.traits() returns a const reference to the traits class object.

I/O

std::ostream&

std::ostream& os << min_annulus

writes *min_annulus* to output stream *os*.

Requirement: The output operator is defined for *Point*.

std::istream&

std::istream& is >> min_annulus&

reads *min_annulus* from input stream *is*.

Requirement: The input operator is defined for *Point*.

See Also

CGAL::Min_sphere_d<Traits> page [2238](#)
CGAL::Optimisation_d_traits_2<K,ET,NT> page [2279](#)
CGAL::Optimisation_d_traits_3<K,ET,NT> page [2281](#)
CGAL::Optimisation_d_traits_d<K,ET,NT> page [2283](#)
OptimisationDTraits page [2285](#)

Implementation

The problem of finding the smallest enclosing annulus of a finite point set can be formulated as an optimization problem with linear constraints and a linear objective function. The solution is obtained using our exact solver for linear and quadratic programs [GS00].

The creation time is almost always linear in the number of points. Access functions and predicates take constant time, inserting a point takes almost always linear time. The clear operation and the check for validity each take linear time.

CGAL::Min_sphere_of_spheres_d<Traits>

Definition

An object of the class *Min_sphere_of_spheres_d<Traits>* is a data structure that represents the unique sphere of smallest volume enclosing a finite set of spheres in d -dimensional Euclidean space \mathbb{E}_d . For a set S of spheres we denote by $ms(S)$ the smallest sphere that contains all spheres of S ; we call $ms(S)$ the *minsphere* of S . $ms(S)$ can be degenerate, i.e., $ms(S) = \emptyset$, if $S = \emptyset$ and $ms(S) = \{s\}$, if $S = \{s\}$. Any sphere in S may be degenerate, too, i.e., any sphere from S may be a point. Also, S may contain several copies of the same sphere.

An inclusion-minimal subset R of S with $ms(R) = ms(S)$ is called a *support set* for $ms(S)$; the spheres in R are the *support spheres*. A support set has size at most $d + 1$, and all its spheres lie on the boundary of $ms(S)$. (A sphere s' is said to *lie on the boundary* of a sphere s , if s' is contained in s and if their boundaries intersect.) In general, the support set is not unique.

The algorithm computes the center and the radius of $ms(S)$, and finds a support set R (which remains fixed until the next *insert()*, *clear()* or *set()* operation). We also provide a specialization of the algorithm for the case when the center coordinates and radii of the input spheres are floating-point numbers. This specialized algorithm uses floating-point arithmetic only, is very fast and especially tuned for stability and robustness. Still, its output may be incorrect in some (rare) cases; termination is guaranteed.

When default constructed, an instance of type *Min_sphere_of_spheres_d<Traits>* represents the set $S = \emptyset$, together with its minsphere $ms(S) = \emptyset$. You can add spheres to the set S by calling *insert()*. Querying the minsphere is done by calling the routines *is_empty()*, *radius()* and *center_cartesian_begin()*, among others.

In general, the radius and the Euclidean center coordinates of $ms(S)$ need not be rational. Consequently, the algorithm computing the exact minsphere will have to deal with algebraic numbers. Fortunately, both the radius and the coordinates of the minsphere are numbers of the form $a_i + b_i\sqrt{t}$, where $a_i, b_i, t \in \mathbb{Q}$ and where $t \geq 0$ is the same for all coordinates and the radius. Thus, the exact minsphere can be described by the number t , which is called the sphere's *discriminant*, and by $d + 1$ pairs $(a_i, b_i) \in \mathbb{Q}^2$ (one for the radius and d for the center coordinates).

```
#include <CGAL/Min_sphere_of_spheres_d.h>
```

Note: This class (almost) replaces *CGAL::Min_sphere_d<Traits>*, which solves the less general problem of finding the smallest enclosing ball of a set of *points*. *Min_sphere_of_spheres_d<Traits>* is faster than *CGAL::Min_sphere_d<Traits>*, and in contrast to the latter provides a specialized implementation for floating-point arithmetic which ensures correct results in a large number of cases (including highly degenerate ones). The only advantage of *CGAL::Min_sphere_d<Traits>* over *Min_sphere_of_spheres_d<Traits>* is that the former can deal with points in homogeneous coordinates, in which case the algorithm is division-free. Thus, *CGAL::Min_sphere_d<Traits>* might still be an option in case your input number type cannot (efficiently) divide.

Requirements

The class *Min_sphere_of_spheres_d<Traits>* expects a model of the concept *MinSphereOfSpheresTraits* as its template argument.

Types

Min_sphere_of_spheres_d<Traits>::Sphere is a typedef to *Traits::Sphere*.

<i>Min_sphere_of_spheres_d<Traits>:: FT</i>	is a typedef to <i>Traits::FT</i> .
<i>Min_sphere_of_spheres_d<Traits>:: Result</i>	is the type of the radius and of the center coordinates of the computed minsphere: When <i>FT</i> is an inexact number type (<i>double</i> , for instance), then <i>Result</i> is simply <i>FT</i> . However, when <i>FT</i> is an exact number type, then <i>Result</i> is a typedef to a derived class of <i>std::pair<FT,FT></i> ; an instance of this type represents the number $a + b\sqrt{t}$, where <i>a</i> is the first and <i>b</i> the second element of the pair and where the number <i>t</i> is accessed using the member function <i>discriminant()</i> of class <i>Min_sphere_of_spheres_d<Traits></i> .
<i>Min_sphere_of_spheres_d<Traits>:: Algorithm</i>	is either <i>CGAL::LP_algorithm</i> or <i>CGAL::Farthest_first_heuristic</i> . As is described in the documentation of concept <i>MinSphereOfSpheresTraits</i> , the type <i>Algorithm</i> reflects the method which is used to compute the minsphere. (Normally, <i>Algorithm</i> coincides with <i>Traits::Algorithm</i> . However, if the method <i>Traits::Algorithm</i> should not be supported anymore in a future release, then <i>Algorithm</i> will have another type.)
<i>Min_sphere_of_spheres_d<Traits>:: Support_iterator</i>	non-mutable model of the STL concept <i>BidirectionalIterator</i> with value type <i>Sphere</i> . Used to access the support spheres defining the smallest enclosing sphere.
<i>Min_sphere_of_spheres_d<Traits>:: Cartesian_const_iterator</i>	non-mutable model of the STL concept <i>BidirectionalIterator</i> to access the center coordinates of the minsphere.

Creation

<i>Min_sphere_of_spheres_d<Traits> minsphere(Traits traits = Traits());</i>	creates a variable of type <i>Min_sphere_of_spheres_d<Traits></i> and initializes it to <i>ms(0)</i> . If the traits parameter is not supplied, the class <i>Traits</i> must provide a default constructor.
<pre>template < typename InputIterator > Min_sphere_of_spheres_d<Traits> minsphere(InputIterator first, InputIterator last, Traits traits = Traits());</pre>	<p>creates a variable <i>minsphere</i> of type <i>Min_sphere_of_spheres_d<Traits></i> and inserts (cf. <i>insert()</i>) the spheres from the range [<i>first</i>,<i>last</i>).</p> <p><i>Requirement:</i> The value type of <i>first</i> and <i>last</i> is <i>Sphere</i>. If the traits parameter is not supplied, the class <i>Traits</i> must provide a default constructor.</p>

Access Functions

<i>Support_iterator</i>	<i>minsphere.support_begin()</i>	returns an iterator referring to the first support sphere of <i>minsphere</i> .
<i>Support_iterator</i>	<i>minsphere.support_end()</i>	returns the corresponding past-the-end iterator.
<i>FT</i>	<i>minsphere.discriminant()</i>	<p>returns the discriminant of <i>minsphere</i>. This number is undefined when <i>FT</i> is an inexact number type. When <i>FT</i> is exact, the center coordinates and the radius of the minsphere are numbers of the form $a + b\sqrt{t}$, where t is the discriminant of the minsphere as returned by this function.</p> <p><i>Precondition:</i> <i>minsphere</i> is not empty, and <i>FT</i> is an exact number type.</p>
<i>Result</i>	<i>minsphere.radius()</i>	<p>returns the radius of <i>minsphere</i>. If <i>FT</i> is an exact number type then the radius of the minsphere is the real number $a + b\sqrt{t}$, where t is the minsphere's discriminant, a is the first and b the second component of the pair returned by <i>radius()</i>.</p> <p><i>Precondition:</i> <i>minsphere</i> is not empty.</p>
<i>Cartesian_const_iterator</i>	<i>minsphere.center_cartesian_begin()</i>	<p>returns a const-iterator to the first of the <i>Traits::D</i> center coordinates of <i>minsphere</i>. The iterator returns objects of type <i>Result</i>. If <i>FT</i> is an exact number type, then a center coordinate is represented by a pair (a, b) describing the real number $a + b\sqrt{t}$, where t is the minsphere's discriminant (cf. <i>discriminant()</i>).</p> <p><i>Precondition:</i> <i>minsphere</i> is not empty.</p>
<i>Cartesian_const_iterator</i>	<i>minsphere.center_cartesian_end()</i>	<p>returns the corresponding past-the-end iterator, i.e. <i>center_cartesian_begin()+Traits::D</i>.</p> <p><i>Precondition:</i> <i>minsphere</i> is not empty.</p>

Predicates

<i>bool</i>	<i>minsphere.is_empty()</i>	returns <i>true</i> , iff <i>minsphere</i> is empty, i.e. iff $ms(S) = \emptyset$.
-------------	-----------------------------	-------------------------------------------------------------------------------------

Modifiers

void *minsphere.clear()* resets *minsphere* to $ms(\emptyset)$, with $S := \emptyset$.

template < class *InputIterator* >

void *minsphere.set(InputIterator first, InputIterator last)*

sets *minsphere* to the $ms(S)$, where S is the set of spheres in the range $[first, last)$.

Requirement: The value type of *first* and *last* is *Sphere*.

void *minsphere.insert(Sphere s)*

inserts the sphere s into the set S of instance *minsphere*.

template < class *InputIterator* >

void *minsphere.insert(InputIterator first, InputIterator last)*

inserts the spheres in the range $[first, last)$ into the set S of instance *minsphere*.

Requirement: The value type of *first* and *last* is *Sphere*.

Validity Check

An object *minsphere* is valid, iff

- *minsphere* contains all spheres of its defining set S ,
- *minsphere* is the smallest sphere containing its support set R , and
- R is minimal, i.e., no support sphere is redundant.

bool *minsphere.is_valid()* returns *true*, iff *minsphere* is valid. When *FT* is inexact, this routine always returns *true*.

Miscellaneous

const Traits& *minsphere.traits()* returns a const reference to the traits class object.

See Also

CGAL::Min_sphere_d<Traits> page [2238](#)

CGAL::Min_circle_2<Traits> page [2193](#)

Implementation

We implement two algorithms, the LP-algorithm and a heuristic [MSW92]. As described in the documentation of concept *MinSphereOfSpheresTraits*, each has its advantages and disadvantages: Our implementation of the LP-algorithm has maximal expected running time $O(2^d n)$, while the heuristic comes without any complexity guarantee. In particular, the LP-algorithm runs in linear time for fixed dimension d . (These running times hold for the arithmetic model, so they count the number of operations on the number type FT .)

On the other hand, the LP-algorithm is, for inexact number types FT , much worse at handling degeneracies and should therefore not be used in such a case. (For exact number types FT , both methods handle all kinds of degeneracies.)

Currently, we require *Traits::FT* to be either an exact number type or *double* or *float*; other inexact number types are not supported at this time. Also, the current implementation only handles spheres with Cartesian coordinates; homogenous representation is not supported yet.

Example

```
// file: examples/Min_sphere_of_spheres_d/min_sphere_of_spheres_d_example_d.C

// Computes the minsphere of some random spheres.
// This example illustrates how to use CGAL::Point_d and CGAL::
// Weighted_point with the Min_sphere_of_spheres_d package.

#include <CGAL/Cartesian_d.h>
#include <CGAL/Random.h>
#include <CGAL/Gmpq.h>
#include <CGAL/Min_sphere_of_spheres_d.h>
#include <vector>

const int N = 1000;           // number of spheres
const int D = 3;             // dimension of points
const int LOW = 0, HIGH = 10000; // range of coordinates and radii

typedef CGAL::Gmpq            FT;
//typedef double              FT;
typedef CGAL::Cartesian_d<FT> K;
typedef CGAL::Min_sphere_of_spheres_d_traits_d<K,FT,D> Traits;
typedef CGAL::Min_sphere_of_spheres_d<Traits> Min_sphere;
typedef K::Point_d            Point;
typedef Traits::Sphere        Sphere;

int main () {
    std::vector<Sphere> S;      // n spheres
    FT coord[D];              // d coordinates
    CGAL::Random r;           // random number generator

    for (int i=0; i<N; ++i) {
        for (int j=0; j<D; ++j)
            coord[j] = r.get_int(LOW,HIGH);
        Point p(D,coord,coord+D); // random center...
        S.push_back(Sphere(p,r.get_int(LOW,HIGH))); // ...and random radius
    }
}
```

```
Min_sphere ms(S.begin(),S.end());      // check in the spheres
CGAL_assertion(ms.is_valid());
}
```


MinSphereOfSpheresTraits

Definition

A model of concept *MinSphereOfSpheresTraits* must provide the following constants, types, predicates and operations.

Has Models

```
CGAL::Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>
CGAL::Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>
CGAL::Min_sphere_of_spheres_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>
```

Constants

MinSphereOfSpheresTraits:: D specifies the dimension of the spheres you want to compute the minsphere of.

Types

MinSphereOfSpheresTraits:: Sphere is a typedef to to some class representing a sphere. (The package will compute the minsphere of spheres of type *Sphere*.) The type *Sphere* must provide a copy constructor.

MinSphereOfSpheresTraits:: FT is a (exact or inexact) field number type.
Requirement: Currently, *FT* must either be *double* or *float*, or an exact field number type. (An *exact* number type is one which evaluates arithmetic expressions involving the four basic operations and comparisons with infinite precision, that is, like in \mathbb{R} .)

MinSphereOfSpheresTraits:: Cartesian_const_iterator

non-mutable model of the STL concept *ForwardIterator* with value type *FT*. Used to access the center coordinates of a sphere.

MinSphereOfSpheresTraits:: Use_square_roots

must typedef to either *CGAL::Tag_true* or *CGAL::Tag_false*. The algorithm uses (depending on the type *MinSphereOfSpheresTraits::Algorithm*) floating-point arithmetic internally for some intermediate computations. The type *Use_square_roots* affects how these calculations are done: When *Use_square_roots* is *Tag_true*, the algorithm computing the minsphere will perform square-root operations on *doubles* and *floats* where appropriate. On the other hand, if *Use_square_roots* is *CGAL::Tag_false*, the algorithm will work without doing square-roots.

Note: On some platforms the algorithm is much faster when square-roots are disabled (due to lacking hardware support).

MinSphereOfSpheresTraits:: Algorithm

selects the method to compute the minsphere with. It must typedef to either *CGAL::Default_algorithm*, *CGAL::LP_algorithm* or *CGAL::Farthest_first_heuristic*. The recommended choice is the first, which is a synonym to the one of the other two methods which we consider “the best in practice.” In case of *CGAL::LP_algorithm*, the minsphere will be computed using the LP-algorithm [MSW92], which in our implementation has maximal expected running time $O(2^d n)$ (in the number of operations on the number type *FT*). In case of *CGAL::Farthest_first_heuristic*, a simple heuristic will be used instead which seems to work fine in practice, but comes without a guarantee on the running time. For an inexact number type *FT* we strongly recommend *CGAL::Default_algorithm*, or, if you want, *CGAL::Farthest_first_heuristic*, since these handle most degeneracies in a satisfying manner. Notice that this compile-time flag is taken as a hint only. Should one of the methods not be available anymore in a future release, then the default algorithm will be chosen.

Access Functions

FT *traits.radius(Sphere s)*

returns the radius of sphere *s*.

Postcondition: The returned number is greater or equal to 0.

Cartesian_const_iterator

traits.center_cartesian_begin(Sphere s)

returns an iterator referring to the first of the *D* Cartesian coordinates of the center of *s*.

CGAL::Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>

Definition

The class *Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>* is a model for concept *MinSphereOfSpheresTraits*. It uses the CGAL type *Point_2* to represent circles.

Is Model for the Concepts

MinSphereOfSpheresTraits

Parameters

The last two template parameters, *UseSqrt* and *Algorithm*, have default arguments, namely *CGAL::Tag_false* and *CGAL::Default_algorithm*, respectively.

The template parameters of class *Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>* must fulfill the following requirements:

Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>:: K

is a model for *Kernel*.

Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>:: FT

is a number type, which fulfills the requirements of type *FT* of concept *MinSphereOfSpheresTraits*: It must be either *double* or *float*, or an exact number type.

Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>:: UseSqrt

fulfills the requirements of type *Use_square_roots* of concept *MinSphereOfSpheresTraits*: It must be either *Tag_true* or *Tag_false*.

Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>:: Algorithm

fulfills the requirements of type *Algorithm* of concept *MinSphereOfSpheresTraits*: It must be either *Default_algorithm*, *LP_algorithm* or *Farthest_first_heuristic*.

Constants

Min_sphere_of_spheres_d_traits_2<K,FT,UseSqrt,Algorithm>:: D

is the constant 2, i.e. the dimension of \mathbb{R}^2 .

Types

In addition to the types required by the concept *MinSphereOfSpheresTraits*, this model also defines the types *Radius* and *Point*. Here's the complete list of defined types:

Min_sphere_of_spheres_d_traits_2<*K*,*FT*,*UseSqrt*,*Algorithm*>::*FT*

Min_sphere_of_spheres_d_traits_2<*K*,*FT*,*UseSqrt*,*Algorithm*>::*Use_square_roots*

Min_sphere_of_spheres_d_traits_2<*K*,*FT*,*UseSqrt*,*Algorithm*>::*Algorithm*

Min_sphere_of_spheres_d_traits_2<*K*,*FT*,*UseSqrt*,*Algorithm*>::*Radius*

is a typedef to the template parameter *FT*

Min_sphere_of_spheres_d_traits_2<*K*,*FT*,*UseSqrt*,*Algorithm*>::*Point*

is a typedef to *K::Point_2*.

Min_sphere_of_spheres_d_traits_2<*K*,*FT*,*UseSqrt*,*Algorithm*>::*Sphere*

is a typedef to *std::pair<Point,Radius>*.

Min_sphere_of_spheres_d_traits_2<*K*,*FT*,*UseSqrt*,*Algorithm*>::*Cartesian_const_iterator*

is a typedef to *K::Cartesian_const_iterator_2*.

Access Functions

The class provides the access functions required by the concept *MinSphereOfSpheresTraits*; they simply map to the corresponding routines of class *K::Point_2*:

FT *traits.radius(Sphere s)*

maps to *s.second*.

Cartesian_const_iterator

traits.center_cartesian_begin(Sphere s)

maps to *s.first.cartesian_begin()*.

CGAL::Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>

Definition

The class *Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>* is a model for concept *MinSphereOfSpheresTraits*. It uses the CGAL type *Point_3* to represent circles.

Is Model for the Concepts

MinSphereOfSpheresTraits

Parameters

The last two template parameters, *UseSqrt* and *Algorithm*, have default arguments, namely *CGAL::Tag_false* and *CGAL::Default_algorithm*, respectively.

The template parameters of class *Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>* must fulfill the following requirements:

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: K

is a model for *Kernel*.

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: FT

is a number type, which fulfills the requirements of type *FT* of concept *MinSphereOfSpheresTraits*: It must be either *double* or *float*, or an exact number type.

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: UseSqrt

fulfills the requirements of type *Use_square_roots* of concept *MinSphereOfSpheresTraits*: It must be either *Tag_true* or *Tag_false*.

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: Algorithm

fulfills the requirements of type *Algorithm* of concept *MinSphereOfSpheresTraits*: It must be either *Default_algorithm*, *LP_algorithm* or *Farthest_first_heuristic*.

Constants

Min_sphere_of_spheres_d_traits_3<K,FT,UseSqrt,Algorithm>:: D

is the constant 3, i.e. the dimension of \mathbb{R}^3 .

Types

In addition to the types required by the concept *MinSphereOfSpheresTraits*, this model also defines the types *Radius* and *Point*. Here's the complete list of defined types:

Min_sphere_of_spheres_d_traits_3<*K*,*FT*,*UseSqrt*,*Algorithm*>::*FT*

Min_sphere_of_spheres_d_traits_3<*K*,*FT*,*UseSqrt*,*Algorithm*>::*Use_square_roots*

Min_sphere_of_spheres_d_traits_3<*K*,*FT*,*UseSqrt*,*Algorithm*>::*Algorithm*

Min_sphere_of_spheres_d_traits_3<*K*,*FT*,*UseSqrt*,*Algorithm*>::*Radius*

is a typedef to the template parameter *FT*

Min_sphere_of_spheres_d_traits_3<*K*,*FT*,*UseSqrt*,*Algorithm*>::*Point*

is a typedef to *K::Point_3*.

Min_sphere_of_spheres_d_traits_3<*K*,*FT*,*UseSqrt*,*Algorithm*>::*Sphere*

is a typedef to *std::pair<Point,Radius>*.

Min_sphere_of_spheres_d_traits_3<*K*,*FT*,*UseSqrt*,*Algorithm*>::*Cartesian_const_iterator*

is a typedef to *K::Cartesian_const_iterator_2*.

Access Functions

The class provides the access functions required by the concept *MinSphereOfSpheresTraits*; they simply map to the corresponding routines of class *K::Point_3*:

FT *traits.radius(Sphere s)*

maps to *s.second*.

Cartesian_const_iterator

traits.center_cartesian_begin(Sphere s)

maps to *s.first.cartesian_begin()*.

CGAL::Min_sphere_of_spheres_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>

Definition

The class *Min_sphere_of_spheres_d_traits_d*<K,FT,Dim,UseSqrt,Algorithm> is a model for concept *MinSphereOfSpheresTraits*. It uses the CGAL type *Point_d* to represent circles.

Is Model for the Concepts

MinSphereOfSpheresTraits

Parameters

The last two template parameters, *UseSqrt* and *Algorithm*, have default arguments, namely *CGAL::Tag_false* and *CGAL::Default_algorithm*, respectively.

The template parameters of class *Min_sphere_of_spheres_d_traits_d*<K,FT,UseSqrt,Algorithm> must fulfill the following requirements:

Min_sphere_of_spheres_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: *K*

is a model for *Kernel*.

Min_sphere_of_spheres_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: *FT*

is a number type, which fulfills the requirements of type *FT* of concept *MinSphereOfSpheresTraits*: It must be either *double* or *float*, or an exact number type.

Min_sphere_of_spheres_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: *UseSqrt*

fulfills the requirements of type *Use_square_roots* of concept *MinSphereOfSpheresTraits*: It must be either *Tag_true* or *Tag_false*.

Min_sphere_of_spheres_d_traits_d<K,FT,Dim,UseSqrt,Algorithm>:: *Algorithm*

fulfills the requirements of type *Algorithm* of concept *MinSphereOfSpheresTraits*: It must be either *Default_algorithm*, *LP_algorithm* or *Farthest_first_heuristic*.

Constants

Min_sphere_of_spheres_d_traits_d<*K*,*FT*,*Dim*,*UseSqrt*,*Algorithm*>::*D*

is the constant *Dim*.

Types

In addition to the types required by the concept *MinSphereOfSpheresTraits*, this model also defines the types *Radius* and *Point*. Here's the complete list of defined types:

Min_sphere_of_spheres_d_traits_d<*K*,*FT*,*Dim*,*UseSqrt*,*Algorithm*>::*FT*

Min_sphere_of_spheres_d_traits_d<*K*,*FT*,*Dim*,*UseSqrt*,*Algorithm*>::*Use_square_roots*

Min_sphere_of_spheres_d_traits_d<*K*,*FT*,*Dim*,*UseSqrt*,*Algorithm*>::*Algorithm*

Min_sphere_of_spheres_d_traits_d<*K*,*FT*,*Dim*,*UseSqrt*,*Algorithm*>::*Radius*

is a typedef to the template parameter *FT*

Min_sphere_of_spheres_d_traits_d<*K*,*FT*,*Dim*,*UseSqrt*,*Algorithm*>::*Point*

is a typedef to *K::Point_d*.

Min_sphere_of_spheres_d_traits_d<*K*,*FT*,*Dim*,*UseSqrt*,*Algorithm*>::*Sphere*

is a typedef to *std::pair<Point,Radius>*.

Min_sphere_of_spheres_d_traits_d<*K*,*FT*,*Dim*,*UseSqrt*,*Algorithm*>::*Cartesian_const_iterator*

is a typedef to *K::Cartesian_const_iterator_d*.

Access Functions

The class provides the access functions required by the concept *MinSphereOfSpheresTraits*; they simply map to the corresponding routines of class *K::Point_d*:

FT *traits.radius(Sphere s)*

maps to *s.second*.

Cartesian_const_iterator

traits.center_cartesian_begin(Sphere s)

maps to *s.first.cartesian_begin()*.

CGAL::Approximate_min_ellipsoid_d<Traits>

Definition

An object of class *Approximate_min_ellipsoid_d<Traits>* is an approximation to the ellipsoid of smallest volume enclosing a finite multiset of points in d -dimensional Euclidean space \mathbb{E}^d , $d \geq 2$.

An *ellipsoid* in \mathbb{E}^d is a Cartesian pointset of the form $\{x \in \mathbb{E}^d \mid x^T E x + x^T e + \eta \leq 0\}$, where E is some positive definite matrix from the set $\mathbb{R}^{d \times d}$, e is some real d -vector, and $\eta \in \mathbb{R}$. A pointset $P \subseteq \mathbb{E}^d$ is called *full-dimensional* if its affine hull has dimension d . For a finite, full-dimensional pointset P we denote by $\text{MEL}(P)$ the smallest ellipsoid that contains all points of P ; this ellipsoid exists and is unique.

For a given finite and full-dimensional pointset $P \subset \mathbb{E}^d$ and a real number $\varepsilon \geq 0$, we say that an ellipsoid $\mathcal{E} \subset \mathbb{E}^d$ is an $(1 + \varepsilon)$ -approximation to $\text{MEL}(P)$ if $P \subset \mathcal{E}$ and $\text{VOL}(\mathcal{E}) \leq (1 + \varepsilon) \text{VOL}(\text{MEL}(P))$. In other words, an $(1 + \varepsilon)$ -approximation to $\text{MEL}(P)$ is an enclosing ellipsoid whose volume is by at most a factor of $1 + \varepsilon$ larger than the volume of the smallest enclosing ellipsoid of P .

Given this notation, an object of class *Approximate_min_ellipsoid_d<Traits>* represents an $(1 + \varepsilon)$ -approximation to $\text{MEL}(P)$ for a given finite and full-dimensional multiset of points $P \subset \mathbb{E}^d$ and a real constant $\varepsilon > 0$.¹ When an *Approximate_min_ellipsoid_d<Traits>* object is constructed, an iterator over the points P and the number ε have to be specified; the number ε defines the *desired approximation ratio* $1 + \varepsilon$. The underlying algorithm will then try to compute an $(1 + \varepsilon)$ -approximation to $\text{MEL}(P)$, and one of the following two cases takes place.

- The algorithm determines that P is not full-dimensional (see *is_full_dimensional()* below).

Important note: due to rounding errors, the algorithm cannot in all cases decide correctly whether P is full-dimensional or not. If *is_full_dimensional()* returns *false*, the points lie in such a “thin” subspace of \mathbb{E}^d that the algorithm is incapable of computing an approximation to $\text{MEL}(P)$. More precisely, if *is_full_dimensional()* returns *false*, there exist two parallel hyperplanes in \mathbb{E}^d with the points P in between so that the distance δ between the hyperplanes is very small, possibly zero. (If $\delta = 0$ then P is not full-dimensional.)

If P is not full-dimensional, linear algebra techniques should be used to determine an affine subspace S of \mathbb{E}^d that contains the points P as a (w.r.t. S) full-dimensional pointset; once S is determined, the algorithm can be invoked again to compute an approximation to (the lower-dimensional) $\text{MEL}(P)$ in S . Since *is_full_dimensional()* might (due to rounding errors, see above) return *false* even though P is full-dimensional, the lower-dimensional subspace S containing P need not exist. Therefore, it might be more advisable to fit a hyperplane H through the pointset P , project P onto this affine subspace H , and compute an approximation to the minimum-volume enclosing ellipsoid of the projected points within H ; the fitting can be done for instance using the *linear_least_squares_fitting()* function from the CGAL package *Principal_component_analysis*.

- The algorithm determines that P is full-dimensional. In this case, it provides an approximation \mathcal{E} to $\text{MEL}(P)$, but depending on the input problem (i.e., on the pair (P, ε)), it may not have achieved the desired approximation ratio but merely some *worse* approximation ratio $1 + \varepsilon' > 1 + \varepsilon$. The achieved approximation ratio $1 + \varepsilon'$ can be queried using *achieved_epsilon()*, which returns ε' . The ellipsoid \mathcal{E} itself can be queried via the methods *defining_matrix()*, *defining_vector()*, and *defining_scalar()*.

The ellipsoid \mathcal{E} computed by the algorithm satisfies the inclusions

$$\frac{1}{(1 + \varepsilon')^d} \mathcal{E} \subseteq \text{conv}(P) \subseteq \mathcal{E} \quad (38.1)$$

¹A *multiset* is a set where elements may have multiplicity greater than 1.

where $f\mathcal{E}$ denotes the ellipsoid \mathcal{E} scaled by the factor $f \in \mathbb{R}^+$ with respect to its center, and where $\text{conv}(A)$ denotes the *convex hull* of a pointset $A \subset \mathbb{E}^d$.

The underlying algorithm can cope with all kinds of inputs (multisets P , $\varepsilon \in [0, \infty)$) and terminates in all cases. There is, however, no guarantee that any desired approximation ratio is actually achieved; the performance of the algorithm in this respect highly depends on the input pointset. Values of at least 0.01 for ε are usually handled without problems.

Internally, the algorithm represents the input points' Cartesian coordinates as *double*'s. For this conversion to work, the input point coordinates must be convertible to *double*. Also, in order to compute the achieved epsilon ε' mentioned above, the algorithm requires a number type *ET* that provides *exact* arithmetic. (Both these aspects are discussed in the documentation of the concept *ApproximateMinEllipsoid_d_Traits_d*.)

```
#include <CGAL/Approximate_min_ellipsoid_d.h>
```

Requirements

The template parameter *Traits* is a model for *ApproximateMinEllipsoid_d_Traits_d*.

We provide the model *CGAL::Approximate_min_ellipsoid_d_traits_d<K>* using the d -dimensional CGAL kernel; the models *CGAL::Approximate_min_ellipsoid_d_traits_2<K>* and *CGAL::Approximate_min_ellipsoid_d_traits_3<K>* are for use with the 2- and 3-dimensional CGAL kernel, respectively.

Types

Approximate_min_ellipsoid_d<Traits>:: FT

typedef Traits::FT FT (which is always a typedef to *double*).

Approximate_min_ellipsoid_d<Traits>:: ET

typedef Traits::ET ET (which is an exact number type used for exact computation like for example in *achieved_epsilon()*).

Approximate_min_ellipsoid_d<Traits>:: Point

typedef Traits::Point Point

Approximate_min_ellipsoid_d<Traits>:: Cartesian_const_iterator

typedef Traits::Cartesian_const_iterator Cartesian_const_iterator

Approximate_min_ellipsoid_d<Traits>:: Center_coordinate_iterator

A model of STL concept *RandomAccessIterator* with value type *double* that is used to iterate over the Cartesian center coordinates of the computed ellipsoid, see *center_cartesian_begin()*.

Approximate_min_ellipsoid_d<Traits>::Axes_lengths_iterator

A model of STL concept *RandomAccessIterator* with value type *double* that is used to iterate over the lengths of the semiaxes of the computed ellipsoid, see *axes_lengths_begin()*.

Approximate_min_ellipsoid_d<Traits>::Axis_direction_iterator

A model of STL concept *RandomAccessIterator* with value type *double* that is used to iterate over the Cartesian coordinates of the direction of a fixed axis of the computed ellipsoid, see *axis_direction_cartesian_begin()*.

Creation

An object of type *Approximate_min_ellipsoid_d<Traits>* can be created from an arbitrary point set P and some nonnegative *double* value *eps*.

template < class Iterator >

Approximate_min_ellipsoid_d<Traits> ame(double eps, Iterator first, Iterator last, Traits traits = Traits());

initializes *ame* to an $(1 + \epsilon)$ -approximation of $\text{MEL}(P)$ with P being the set of points in the range $[first, last)$. The number ϵ in this will be at most *eps*, if possible. However, due to the limited precision in the algorithm's underlying arithmetic, it can happen that the computed approximation ellipsoid has a worse approximation ratio (and ϵ can thus be larger than *eps* in general). In any case, the number ϵ (and with this, the achieved approximation $1 + \epsilon$) can be queried by calling the routine *achieved_epsilon()* discussed below.

Requirement: *Iterator* must be a model for concept *InputIterator* with value type *Point*.

Precondition: The dimension d of the input points must be at least 2, and $\epsilon > 0$.

Access Functions

The following methods can be used to query the achieved approximation ratio $1 + \epsilon'$ and the computed ellipsoid $\mathcal{E} = \{x \in \mathbb{E}^d \mid x^T E x + x^T e + \eta \leq 0\}$. The methods *defining_matrix()*, *defining_vector()*, and *defining_scalar()* do not return E , e , and η directly but yield multiples of these quantities that are exactly representable using the *double* type. (This is necessary because the parameters E , e , and η of the computed approximation ellipsoid \mathcal{E} might not be exactly representable as *double* numbers.)

unsigned int ame.number_of_points()

returns the number of points of *ame*, i.e., $|P|$.

<i>double</i>	<i>ame.achieved_epsilon()</i>	<p>returns a number ϵ' such that the computed approximation is (under exact arithmetic) guaranteed to be an $(1 + \epsilon')$-approximation to $\text{MEL}(P)$.</p> <p><i>Precondition:</i> <i>ame.is_full_dimensional()</i> == <i>true</i>.</p> <p><i>Postcondition:</i> $\epsilon' > 0$.</p>
<i>double</i>	<i>ame.defining_matrix(int i, int j)</i>	<p>gives access to the (i, j)th entry of the matrix E in the representation $\{x \in \mathbb{E}^d \mid x^T E x + x^T e + \eta \leq 0\}$ of the computed approximation ellipsoid \mathcal{E}. The number returned by this routine is $(1 + \epsilon')(d + 1) E_{ij}$, where ϵ' is the number returned by <i>achieved_epsilon()</i>.</p> <p><i>Precondition:</i> $0 \leq i, j \leq d$, where d is the dimension of the points P, and <i>ame.is_full_dimensional()</i> == <i>true</i>.</p>
<i>double</i>	<i>ame.defining_vector(int i)</i>	<p>gives access to the ith entry of the vector e in the representation $\{x \in \mathbb{E}^d \mid x^T E x + x^T e + \eta \leq 0\}$ of the computed approximation ellipsoid \mathcal{E}. The number returned by this routine is $(1 + \epsilon')(d + 1) e_i$, where ϵ' is the number returned by <i>achieved_epsilon()</i>.</p> <p><i>Precondition:</i> $0 \leq i \leq d$, where d is the dimension of the points P, and <i>ame.is_full_dimensional()</i> == <i>true</i>.</p>
<i>double</i>	<i>ame.defining_scalar()</i>	<p>gives access to the scalar η from the representation $\{x \in \mathbb{E}^d \mid x^T E x + x^T e + \eta \leq 0\}$ of the computed approximation ellipsoid \mathcal{E}. The number returned by this routine is $(1 + \epsilon')(d + 1)(\eta + 1)$, where ϵ' is the number returned by <i>achieved_epsilon()</i>.</p> <p><i>Precondition:</i> <i>ame.is_full_dimensional()</i> == <i>true</i>.</p>
<i>Traits</i>	<i>ame.traits()</i>	returns a const reference to the traits class object.
<i>int</i>	<i>ame.dimension()</i>	returns the dimension of the ambient space, i.e., the dimension of the points P .

In order to access the center and semiaxes of the computed approximation ellipsoid, the functions *center_cartesian_begin()*, *axes_lengths_begin()*, and *axis_direction_cartesian_begin()* can be used. In contrast to the above access functions *achieved_epsilon()*, *defining_matrix()*, *defining_vector()*, and *defining_scalar()*, which return the described quantities exactly, the routines below return *numerical approximations* to the real center and real semiaxes of the computed ellipsoid; the comprised relative error may be larger than zero, and there are no guarantees for the returned quantities.

Center_coordinate_iterator

ame.center_cartesian_begin()

returns an iterator pointing to the first of the d Cartesian coordinates of the computed ellipsoid's center.

The returned point is a floating-point approximation to the ellipsoid's exact center; no guarantee is given w.r.t. the involved relative error.

Precondition: *ame.is_full_dimensional()* == *true*.

Center_coordinate_iterator

ame.center_cartesian_end()

returns the past-the-end iterator corresponding to *center_cartesian_begin()*.

Precondition: *ame.is_full_dimensional()* == *true*.

Axes_lengths_iterator

ame.axes_lengths_begin()

returns an iterator pointing to the first of the d descendantly sorted lengths of the computed ellipsoid's axes. The d returned numbers are floating-point approximations to the exact axes-lengths of the computed ellipsoid; no guarantee is given w.r.t. the involved relative error. (See also method *axes_direction_cartesian_begin()*.)

Precondition: *ame.is_full_dimensional()* == *true*, and $d \in \{2, 3\}$.

Axes_lengths_iterator

ame.axes_lengths_end()

returns the past-the-end iterator corresponding to *axes_lengths_begin()*.

Precondition: *ame.is_full_dimensional()* == *true*, and $d \in \{2, 3\}$.

Axes_direction_coordinate_iterator

ame.axis_direction_cartesian_begin(int i)

returns an iterator pointing to the first of the d Cartesian coordinates of the computed ellipsoid's i th axis direction (i.e., unit vector in direction of the ellipsoid's i th axis). The direction described by this iterator is a floating-point approximation to the exact axis direction of the computed ellipsoid; no guarantee is given w.r.t. the involved relative error. An approximation to the length of axis i is given by the i th entry of *axes_lengths_begin()*.

Precondition: *ame.is_full_dimensional()* == *true*, and $d \in \{2, 3\}$, and $0 \leq i < d$.

Axes_direction_coordinate_iterator

ame.axis_direction_cartesian_end(int i)

returns the past-the-end iterator corresponding to *axis_direction_cartesian_begin()*.

Precondition: *ame.is_full_dimensional()* == *true*, and $d \in \{2, 3\}$, and $0 \leq i < d$.

Predicates

bool

ame.is_full_dimensional()

returns whether P is full-dimensional or not, i.e., returns *true* if and only if P is full-dimensional.

Note: due to the limited precision in the algorithm's underlying arithmetic, the result of this method is not always correct. Rather, a return value of *false* means that the points P are contained in a “very thin” linear subspace of \mathbb{E}^d , and as a consequence, the algorithm cannot compute an approximation. More precisely, a return value of *false* means that the points P are contained between two parallel hyperplanes in \mathbb{E}^d that are very close to each other (possibly at distance zero) — so close, that the algorithm could not compute an approximation ellipsoid. Similarly, a return value of *true* does not guarantee P to be full-dimensional; but there exists an input pointset P' such that the points P' and P have almost identical coordinates and P' is full-dimensional.

Validity Check

An object *ame* is valid iff

- *ame* contains all points of its defining set P ,
- *ame* is an $(1 + \varepsilon')$ -approximation to the smallest ellipsoid $\text{MEL}(P)$ of P ,
- The ellipsoid represented by *ame* fulfills the inclusion (38.1).

bool *ame.is_valid(bool verbose = false)*

returns *true* iff *ame* is valid according to the above definition.
If *verbose* is *true*, some messages concerning the performed checks are written to the standard error stream.

Miscellaneous

void *ame.write_eps(const std::string& name)*

Writes the points P and the computed approximation to $\text{MEL}(P)$ as an EPS-file under pathname *name*.
Precondition: The dimension of points P must be 2.
Note: this routine is provided as a debugging routine; future version of CGAL might not provide it anymore.
Precondition: *ame.is_full_dimensional() == true*.

See Also

CGAL::Min_ellipse_2<Traits> page [2203](#)

Implementation

We implement Khachyan's algorithm for rounding polytopes [Kha96]. Internally, we use *double*-arithmetic and (initially a single) Cholesky-decomposition. The algorithm's running time is $O(nd^2(\varepsilon^{-1} + \ln d + \ln \ln(n)))$, where $n = |P|$ and $1 + \varepsilon$ is the desired approximation ratio.

Example

To illustrate the usage of *Approximate_min_ellipsoid_d<Traits>* we give two examples in 2D. The first program generates a random set $P \subset \mathbb{E}^2$ and outputs the points and a 1.01-approximation of $\text{MEL}(P)$ as an EPS-file, which you can view using *gv*, for instance. (In both examples you can change the variables n and d to experiment with the code.)

```
#include <vector>
#include <iostream>
```

```

#include <CGAL/basic.h>
#include <CGAL/Cartesian_d.h>
#include <CGAL/MP_Float.h>
#include <CGAL/point_generators_d.h>
#include <CGAL/Approximate_min_ellipsoid_d.h>
#include <CGAL/Approximate_min_ellipsoid_d_traits_d.h>

typedef CGAL::Cartesian_d<double> Kernel;
typedef CGAL::MP_Float ET;
typedef CGAL::Approximate_min_ellipsoid_d_traits_d<Kernel, ET> Traits;
typedef Traits::Point Point;
typedef std::vector<Point> Point_list;
typedef CGAL::Approximate_min_ellipsoid_d<Traits> AME;

int main()
{
    const int      n = 1000;           // number of points
    const int      d = 2;              // dimension
    const double eps = 0.01;           // approximation ratio is (1+eps)

    // create a set of random points:
    Point_list P;
    CGAL::Random_points_in_iso_box_d<Point> rpg(d,100.0);
    for (int i = 0; i < n; ++i) {
        P.push_back(*rpg);
        ++rpg;
    }

    // compute approximation:
    Traits traits;
    AME ame(eps, P.begin(), P.end(), traits);

    // write EPS file:
    if (ame.is_full_dimensional() && d == 2)
        ame.write_eps("example.eps");

    // output center coordinates:
    std::cout << "Cartesian center coordinates: ";
    for (AME::Center_coordinate_iterator c_it = ame.center_cartesian_begin();
        c_it != ame.center_cartesian_end();
        ++c_it)
        std::cout << *c_it << ' ';
    std::cout << ".\n";

    if (d == 2 || d == 3) {
        // output axes:
        AME::Axes_lengths_iterator axes = ame.axes_lengths_begin();
        for (int i = 0; i < d; ++i) {
            std::cout << "Semiaxis " << i << " has length " << *axes++ << "\n"
                << "and Cartesian coordinates ";
            for (AME::Axes_direction_coordinate_iterator
                d_it = ame.axis_direction_cartesian_begin(i);
                d_it != ame.axis_direction_cartesian_end(i); ++d_it)
                std::cout << *d_it << ' ';

```

```

        std::cout << ".\n";
    }
}
}

```

The second program outputs the approximation in a format suitable for display in Maplesoft's Maple.

```

// Usage: ./maple_example > maple.text
// Then enter in Maple 'read "maple.text";'.

#include <vector>
#include <iostream>
#include <iomanip>

#include <CGAL/basic.h>
#include <CGAL/Cartesian_d.h>
#include <CGAL/MP_Float.h>
#include <CGAL/point_generators_d.h>
#include <CGAL/Approximate_min_ellipsoid_d.h>
#include <CGAL/Approximate_min_ellipsoid_d_traits_d.h>

typedef CGAL::Cartesian_d<double>           Kernel;
typedef CGAL::MP_Float                      ET;
typedef CGAL::Approximate_min_ellipsoid_d<Kernel, ET> Traits;
typedef Traits::Point                      Point;
typedef std::vector<Point>                  Point_list;
typedef CGAL::Approximate_min_ellipsoid_d<Traits> AME;

int main()
{
    const int      n = 100;           // number of points
    const int      d = 2;             // dimension
    const double eps = 0.01;          // approximation ratio is (1+eps)

    // create a set of random points:
    Point_list P;
    CGAL::Random_points_in_iso_box_d<Point> rpg(d,1.0);
    for (int i = 0; i < n; ++i) {
        P.push_back(*rpg);
        ++rpg;
    }

    // compute approximation:
    Traits traits;
    AME mel(eps, P.begin(), P.end(), traits);

    // output for Maple:
    if (mel.is_full_dimensional() && d == 2) {

        const double alpha = (1+mel.achieved_epsilon())*(d+1);

        // output points:
        using std::cout;

```



```

cout << "restart;\n"
    << "with(LinearAlgebra):\n"
    << "with(plottools):\n"
    << "n:= " << n << ":\n"
    << "P:= Matrix(" << d << ", " << n << "):\n";

for (int i=0; i<n; ++i)
    for (int j=0; j<d; ++j)
        cout << "P[" << j+1 << ", " << i+1 << "] := "
            << std::setiosflags(std::ios::scientific)
            << std::setprecision(20) << P[i][j] << ":\n";
cout << "\n";

// output defining equation:
cout << "Mp:= Matrix([\n";
for (int i=0; i<d; ++i) {
    cout << " [";
    for (int j=0; j<d; ++j) {
        cout << mel.defining_matrix(i,j)/alpha;
        if (j<d-1)
            cout << ", ";
    }
    cout << "]\n";
    if (i<d-1)
        cout << ", ";
    cout << "\n";
}
cout << "]);\n" << "mp:= Vector([";
for (int i=0; i<d; ++i) {
    cout << mel.defining_vector(i)/alpha;
    if (i<d-1)
        cout << ", ";
}
cout << "]);\n"
    << "eta:= " << (mel.defining_scalar()/alpha-1.0) << ";\n"
    << "v:= Vector([x,y]):\n"
    << "e:= Transpose(v).Mp.v+Transpose(v).mp+eta;\n"
    << "plots[display]({seq(point([P[1,i],P[2,i]]),i=1..n),\n"
    << " plots[implicitplot](e,x=-5..5,y=-5..5,numpoints=10000)},\n"
    << " scaling=CONSTRAINED);\n";
}
}

```

ApproximateMinEllipsoid_d_Traits_d

Definition

This concept defines the requirements for traits classes of *CGAL::Approximate_min_ellipsoid_d<Traits>*.

Refines

DefaultConstructible

CopyConstructible

Assignable

Types

ApproximateMinEllipsoid_d_Traits_d:: FT *typedef double FT*

ApproximateMinEllipsoid_d_Traits_d:: ET Some model of concept *RingNumberType* that provides exact arithmetic, meaning that *CGAL::Number_type_traits<ET>::Has_exact_ring_operations* must be *CGAL::Tag_true*. In addition, *ET* must be able to exactly represent any finite *double* value. (An example for such a type is *CGAL::MP_Float*.)
The type *ET* is to be used by the *Approximate_min_ellipsoid_d<Traits>* class for internal, exact computations.

ApproximateMinEllipsoid_d_Traits_d:: Point Type of the input points. *Point* must provide the default and copy constructor, and must be a model of *DefaultConstructible*, *CopyConstructible*, and *Assignable*.

ApproximateMinEllipsoid_d_Traits_d:: Cartesian_const_iterator Model for the STL concept *RandomAccessIterator* whose value type must be convertible to *double*. This type is used to iterate over the Cartesian coordinates of an instance of type *Point*, see *cartesian_begin()* below.

Access Functions

int *traits.dimension(Point p)*
returns the dimension of a point *p*.

Cartesian_const_iterator *traits.cartesian_begin(Point p)*
returns an input iterator over the Euclidean coordinates of the point *p*. The range of the iterator must have size *dimension(p)*.

Has Models

CGAL::Approximate_min_ellipsoid_d_traits_2<*K*, *ET*> page ??
CGAL::Approximate_min_ellipsoid_d_traits_3<*K*, *ET*> page ??
CGAL::Approximate_min_ellipsoid_d_traits_d<*K*, *ET*> page ??

See Also

CGAL::Min_ellipse_2<*Traits*> page [2203](#)

CGAL::Approximate_min_ellipsoid_d_traits_2<K,ET>

Definition

The class *Approximate_min_ellipsoid_d_traits_2<K,ET>* is a traits class for *CGAL::Approximate_min_ellipsoid_d<Traits>* using the 2-dimensional CGAL kernel. In order to use this class, an exact number-type *ET* has to be provided which *Approximate_min_ellipsoid_d<Traits>* will use for its internal exact computations.

```
#include <CGAL/Approximate_min_ellipsoid_d_traits_2.h>
```

Requirements

The template parameter *K* must be a model for concept *Kernel*. The template parameter *ET* must be a model for concept *RingNumberType* with exact arithmetic operations, i.e., the type *CGAL::Number_type_traits<ET>::Has_exact_ring_operations* must be *CGAL::Tag_true*. In addition, *ET* must be able to exactly represent any finite *double* value. (Examples of such a number-type are *CGAL::MP_Float*, *CORE::Expr*, and *CGAL::Gmpq*.)

Is Model for the Concepts

ApproximateMinEllipsoid_d_Traits_d

Types

Approximate_min_ellipsoid_d_traits_2<K,ET>::FT *typedef double FT*. The kernel's number type *K::RT* must be convertible to *double*.

Approximate_min_ellipsoid_d_traits_2<K,ET>::ET *typedef* to the second template argument, *ET*.

Approximate_min_ellipsoid_d_traits_2<K,ET>::Point
typedef K::Point_2 Point

Approximate_min_ellipsoid_d_traits_2<K,ET>::Cartesian_const_iterator
typedef K::Cartesian_const_iterator_2 Cartesian_const_iterator

See Also

CGAL::Approximate_min_ellipsoid_d_traits_3<K,ET> page [2277](#)
CGAL::Approximate_min_ellipsoid_d_traits_d<K,ET> page [2278](#)
ApproximateMinEllipsoid_d_Traits_d

CGAL::Approximate_min_ellipsoid_d_traits_3<K,ET>

Definition

The class *Approximate_min_ellipsoid_d_traits_3<K,ET>* is a traits class for *CGAL::Approximate_min_ellipsoid_d<Traits>* using the 3-dimensional CGAL kernel. In order to use this class, an exact number-type *ET* has to be provided which *Approximate_min_ellipsoid_d<Traits>* will use for its internal exact computations.

```
#include <CGAL/Approximate_min_ellipsoid_d_traits_3.h>
```

Requirements

The template parameter *K* must be a model for concept *Kernel*. The template parameter *ET* must be a model for concept *RingNumberType* with exact arithmetic operations, i.e., the type *CGAL::Number_type_traits<ET>::Has_exact_ring_operations* must be *CGAL::Tag_true*. In addition, *ET* must be able to exactly represent any finite *double* value. (Examples of such a number-type are *CGAL::MP_Float*, *CORE::Expr*, and *CGAL::Gmpq*.)

Is Model for the Concepts

ApproximateMinEllipsoid_d_Traits_d

Types

Approximate_min_ellipsoid_d_traits_3<K,ET>::FT

typedef double FT. The kernel's number type *K::RT* must be convertible to *double*.

Approximate_min_ellipsoid_d_traits_3<K,ET>::ET

typedef to the second template argument, *ET*.

Approximate_min_ellipsoid_d_traits_3<K,ET>::Point

typedef K::Point_3 Point

Approximate_min_ellipsoid_d_traits_3<K,ET>::Cartesian_const_iterator

typedef K::Cartesian_const_iterator_3 Cartesian_const_iterator

See Also

CGAL::Approximate_min_ellipsoid_d_traits_2<K,ET> page [2276](#)
CGAL::Approximate_min_ellipsoid_d_traits_d<K,ET> page [2278](#)
ApproximateMinEllipsoid_d_Traits_d

CGAL::Approximate_min_ellipsoid_d_traits_d<K,ET>

Definition

The class *Approximate_min_ellipsoid_d_traits_d<K,ET>* is a traits class for *CGAL::Approximate_min_ellipsoid_d<Traits>* using the *d*-dimensional CGAL kernel. In order to use this class, an exact number-type *ET* has to be provided which *Approximate_min_ellipsoid_d<Traits>* will use for its internal exact computations.

```
#include <CGAL/Approximate_min_ellipsoid_d_traits_d.h>
```

Requirements

The template parameter *K* must be a model for concept *Kernel*. The template parameter *ET* must be a model for concept *RingNumberType* with exact arithmetic operations, i.e., the type *CGAL::Number_type_traits<ET>::Has_exact_ring_operations* must be *CGAL::Tag_true*. In addition, *ET* must be able to exactly represent any finite *double* value. (Examples of such a number-type are *CGAL::MP_Float*, *CORE::Expr*, and *CGAL::Gmpq*.)

Is Model for the Concepts

ApproximateMinEllipsoid_d_Traits_d

Types

Approximate_min_ellipsoid_d_traits_d<K,ET>:: FT

typedef double FT. The kernel's number type *K::RT* must be convertible to *double*.

Approximate_min_ellipsoid_d_traits_d<K,ET>:: ET

typedef to the second template argument, *ET*.

Approximate_min_ellipsoid_d_traits_d<K,ET>:: Point

typedef K::Point_d Point

Approximate_min_ellipsoid_d_traits_d<K,ET>:: Cartesian_const_iterator

typedef K::Cartesian_const_iterator Cartesian_const_iterator

See Also

CGAL::Approximate_min_ellipsoid_d_traits_2<K,ET> page [2276](#)
CGAL::Approximate_min_ellipsoid_d_traits_3<K,ET> page [2277](#)
ApproximateMinEllipsoid_d_Traits_d

Creation

Optimisation_d_traits_2<*K,ET,NT*> traits; default constructor.

Optimisation_d_traits_2<*K,ET,NT*> traits(*Optimisation_d_traits_2*<*K,ET,NT*>);
copy constructor.

Operations

The following functions just return the corresponding function class object.

Access_dimension_d *traits.access_dimension_d_object()*

Access_coordinates_begin_d *traits.access_coordinates_begin_d_object()*

Construct_point_d *traits.construct_point_d_object()*

See Also

CGAL::Min_sphere_d<*Traits*> [page 2238](#)
CGAL::Min_annulus_d<*Traits*> [page 2244](#)
CGAL::Polytope_distance_d<*Traits*> [page 2314](#)
CGAL::Optimisation_d_traits_3<*K,ET,NT*> [page 2281](#)
CGAL::Optimisation_d_traits_d<*K,ET,NT*> [page 2283](#)
OptimisationDTraits [page 2285](#)

CGAL::Optimisation_d_traits_3<K,ET,NT>

Definition

The class *Optimisation_d_traits_3<K,ET,NT>* is a traits class for the d -dimensional optimisation algorithms using the three-dimensional CGAL kernel.

```
#include <CGAL/Optimisation_d_traits_3.h>
```

Requirements

The template parameter K is a model for *Kernel*. Template parameters ET and NT are models for *RingNumberType*.

The second and third template parameter have default type `K::RT`.

Is Model for the Concepts

OptimisationDTraits page 2285

Types

$$Optimisation_d_traits_3\langle K, ET, NT \rangle :: Point_d \quad \text{typedef to } K :: Point_3.$$
$$Optimisation_d_traits_3\langle K, ET, NT \rangle :: Rep_tag \quad \text{typedef to } K :: Rep_tag.$$
$$Optimisation_d_traits_3<K,ET,NT>::RT \quad \text{typedef to } K::RT.$$
$$Optimisation_d_traits_3<K,ET,NT>::FT \quad \text{typedef to } K::FT.$$

```
Optimisation_d_traits_3<K,ET,NT>>:: Access_dimension_d
```

typedef to *K::Access_dimension_3*.

```
Optimisation_d_traits_3<K,ET,NT>>::Access_coordinates_begin_d
```

$$\text{Optimisation_d_traits_3}\langle K, ET, NT \rangle :: \text{Construct_point_d}$$

$$\text{typedef to } K :: \text{Construct_point_3.}$$

Optimisation_d_traits_3<K,ET,NT>::ET second template parameter (default is *K::RT*).

Optimisation_d_traits_3<*K,ET,NT*>::*NT* third template parameter (default is *K::RT*).

Creation

Optimisation_d_traits_3<*K,ET,NT*> traits; default constructor.

Optimisation_d_traits_3<*K,ET,NT*> traits(*Optimisation_d_traits_3*<*K,ET,NT*>);
copy constructor.

Operations

The following functions just return the corresponding function class object.

Access_dimension_d traits.access_dimension_d_object()

Access_coordinates_begin_d traits.access_coordinates_begin_d_object()

Construct_point_d traits.construct_point_d_object()

See Also

CGAL::Min_sphere_d<*Traits*> page [2238](#)
CGAL::Min_annulus_d<*Traits*> page [2244](#)
CGAL::Polytope_distance_d<*Traits*> page [2314](#)
CGAL::Optimisation_d_traits_2<*K,ET,NT*> page [2279](#)
CGAL::Optimisation_d_traits_d<*K,ET,NT*> page [2283](#)
OptimisationDTraits page [2285](#)

CGAL::Optimisation_d_traits_d<K,ET,NT>

Definition

The class *Optimisation_d_traits_d* $\langle K, ET, NT \rangle$ is a traits class for the d -dimensional optimisation algorithms using the d -dimensional CGAL kernel.

```
#include <CGAL/Optimisation_d_traits_d.h>
```

Requirements

The template parameter K is a model for *Kernel*. Template parameters ET and NT are models for *RingNumberType*.

The second and third template parameter have default type `K::RT`.

Is Model for the Concepts

OptimisationDTraits page 2285

Types

$$Optimisation_d_traits_d\langle K, ET, NT \rangle :: Point_d \quad \text{typedef to } K :: Point_d.$$
$$Optimisation_d_traits_d\langle K, ET, NT \rangle :: Rep_tag \quad \text{typedef to } K :: Rep_tag.$$
$$Optimisation_d_traits_d\langle K, ET, NT \rangle :: RT \quad \text{typedef to } K :: RT.$$
$$Optimisation_d_traits_d\langle K, ET, NT \rangle :: FT \quad \text{typedef to } K :: FT.$$
$$\text{Optimisation_d_traits_d} \langle K, ET, NT \rangle :: \text{Access_dimension_d}$$

$$\text{typedef to } K :: \text{Access_dimension_d.}$$

```
Optimisation_d_traits_d<K,ET,NT>::Access_coordinates_begin_d
```

$$\text{Optimisation_d_traits_d} \langle K, ET, NT \rangle :: \text{Construct_point_d}$$

$$\text{typedef to } K :: \text{Construct_point_d}.$$

Optimisation_d_traits_d $\langle K, ET, NT \rangle :: ET$ second template parameter (default is $K :: RT$).

Optimisation_d_traits_d<*K*,*ET*,*NT*>::*NT* third template parameter (default is *K*::*RT*).

Creation

Optimisation_d_traits_d<*K*,*ET*,*NT*> traits; default constructor.

Optimisation_d_traits_d<*K*,*ET*,*NT*> traits(*Optimisation_d_traits_d*<*K*,*ET*,*NT*>);
copy constructor.

Operations

The following functions just return the corresponding function class object.

Access_dimension_d traits.access_dimension_d_object()

Access_coordinates_begin_d traits.access_coordinates_begin_d_object()

Construct_point_d traits.construct_point_d_object()

See Also

CGAL::Min_sphere_d<*Traits*> page [2238](#)
CGAL::Min_annulus_d<*Traits*> page [2244](#)
CGAL::Polytope_distance_d<*Traits*> page [2314](#)
CGAL::Optimisation_d_traits_2<*K*,*ET*,*NT*> page [2279](#)
CGAL::Optimisation_d_traits_3<*K*,*ET*,*NT*> page [2281](#)
OptimisationDTraits page [2285](#)

OptimisationDTraits

Definition

This concept defines the requirements for traits classes of d -dimensional optimisation algorithms.

Types

<i>OptimisationDTraits:: Point_d</i>	point type used to represent the input points.
<i>OptimisationDTraits:: Rep_tag</i>	compile time tag to distinguish between Cartesian and homogeneous representation of the input points. <i>Rep_tag</i> has to be either <i>CGAL::Cartesian_tag</i> or <i>CGAL::Homogeneous_tag</i> .
<i>OptimisationDTraits:: RT</i>	number type used to represent the coordinates of the input points. It has to be a model for <i>RingNumberType</i> .
<i>OptimisationDTraits:: FT</i>	number type used to return either the squared radius of the smallest enclosing sphere or annulus, or the squared distance of the polytopes. <i>FT</i> has to be either <i>RT</i> or <i>CGAL::Quotient<RT></i> if the input points have Cartesian or homogeneous representation, respectively (cf. <i>Rep_tag</i>).
<i>OptimisationDTraits:: Access_dimension_d</i>	data accessor object used to access the dimension of the input points.
<i>OptimisationDTraits:: Access_coordinates_begin_d</i>	data accessor object used to access the coordinates of the input points.
<i>OptimisationDTraits:: Construct_point_d</i>	constructor object used to construct either the center of the smallest enclosing sphere or annulus, or the points realizing the distance between the two polytopes.

The following two number types are only needed for *CGAL::Min_annulus_d<Traits>* and *CGAL::Polytope_distance_d<Traits>*.

<i>OptimisationDTraits:: ET</i>	exact number type used to do the exact computations in the underlying solver for linear programs. It has to be a model for <i>RingNumberType</i> . There must be an implicit conversion from <i>RT</i> to <i>ET</i> available.
<i>OptimisationDTraits:: NT</i>	fast (possibly inexact) number type used to speed up the pricing step in the underlying solver for linear programs. It has to be a model for <i>RingNumberType</i> . There must be implicit conversions from <i>RT</i> to <i>NT</i> and from <i>NT</i> to <i>ET</i> available.

Creation

Only default and copy constructor are required.

```
OptimisationDTraits traits;
```

```
OptimisationDTraits traits( OptimisationDTraits);
```

Operations

The following functions just return the corresponding function class object.

```
Access_dimension_d traits.access_dimension_d_object()
```

```
Access_coordinates_begin_d traits.access_coordinates_begin_d_object()
```

```
Construct_point_d traits.construct_point_d_object()
```

Has Models

CGAL::Optimisation_d_traits_2<K,ET,NT> page [2279](#)

CGAL::Optimisation_d_traits_3<K,ET,NT> page [2281](#)

CGAL::Optimisation_d_traits_d<K,ET,NT> page [2283](#)

See Also

CGAL::Min_sphere_d<Traits> page [2238](#)

CGAL::Min_annulus_d<Traits> page [2244](#)

CGAL::Polytope_distance_d<Traits> page [2314](#)

CGAL::maximum_area_inscribed_k_gon_2

Definition

The function *maximum_area_inscribed_k_gon_2* computes a maximum area k -gon P_k that can be inscribed into a given convex polygon P . Note that

- P_k is not unique in general, but it can be chosen in such a way that its vertices form a subset of the vertex set of P and
- the vertices of a maximum area k -gon, where the k vertices are to be drawn from a planar point set S , lie on the convex hull of S i.e. a convex polygon.

```
#include <CGAL/extremal_polygon_2.h>
```

```
template < class RandomAccessIterator, class OutputIterator >
OutputIterator      maximum_area_inscribed_k_gon_2(
                    RandomAccessIterator points_begin,
                    RandomAccessIterator points_end,
                    int k,
                    OutputIterator o)
```

computes a maximum area inscribed k -gon of the convex polygon described by $[points_begin, points_end)$, writes its vertices to o and returns the past-the-end iterator of this sequence.

Precondition:

1. the – at least three – points denoted by the range $[points_begin, points_end)$ form the boundary of a convex polygon (oriented clock- or counterclockwise).
2. $k \geq 3$.

Requirement:

1. Value type of *RandomAccessIterator* is $K::Point_2$ where K is a model for *Kernel*.
2. *OutputIterator* accepts the value type of *RandomAccessIterator* as value type.

See Also

CGAL::maximum_perimeter_inscribed_k_gon_2 page 2289
 ExtremalPolygonTraits_2 page 2296
 CGAL::Extremal_polygon_area_traits_2<K> page 2292
 CGAL::Extremal_polygon_perimeter_traits_2<K> page 2294
 CGAL::extremal_polygon_2 page 2291
 CGAL::monotone_matrix_search page 2321

Implementation

The implementation uses monotone matrix search [AKM⁺87] and has a worst case running time of $O(k \cdot n + n \cdot \log n)$, where n is the number of vertices in P .

Example

The following code generates a random convex polygon p with ten vertices and computes the maximum area inscribed five-gon of p .

```
#include <CGAL/Cartesian.h>
#include <CGAL/Polygon_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_convex_set_2.h>
#include <CGAL/extremal_polygon_2.h>
#include <iostream>
#include <vector>

typedef double FT;

struct Kernel : public CGAL::Cartesian<FT> {};

typedef Kernel::Point_2 Point;
typedef std::vector<int> Index_cont;
typedef CGAL::Polygon_2<Kernel> Polygon;
typedef CGAL::Random_points_in_square_2<Point> Generator;

int main() {

    int n = 10;
    int k = 5;

    // generate random convex polygon:
    Polygon p;
    CGAL::random_convex_set_2(n, std::back_inserter(p), Generator(1));
    std::cout << "Generated Polygon:\n" << p << std::endl;

    // compute maximum area inscribed k-gon of p:
    Polygon k_gon;
    CGAL::maximum_area_inscribed_k_gon_2(
        p.vertices_begin(), p.vertices_end(), k, std::back_inserter(k_gon));
    std::cout << "Maximum area " << k << "-gon:\n"
        << k_gon << std::endl;

    return 0;
}
```


CGAL::maximum_perimeter_inscribed_k_gon_2

Definition

The function *maximum_perimeter_inscribed_k_gon_2* computes a maximum perimeter k -gon P_k that can be inscribed into a given convex polygon P . Note that

- P_k is not unique in general, but it can be chosen in such a way that its vertices form a subset of the vertex set of P and
- the vertices of a maximum perimeter k -gon, where the k vertices are to be drawn from a planar point set S , lie on the convex hull of S i.e. a convex polygon.

```
#include <CGAL/extremal_polygon_2.h>
```

```
template < class RandomAccessIterator, class OutputIterator >
OutputIterator      maximum_perimeter_inscribed_k_gon_2(
                    RandomAccessIterator points_begin,
                    RandomAccessIterator points_end,
                    int k,
                    OutputIterator o)
```

computes a maximum perimeter inscribed k -gon of the convex polygon described by $[points_begin, points_end)$, writes its vertices to o and returns the past-the-end iterator of this sequence.

Precondition:

1. the – at least three – points denoted by the range $[points_begin, points_end)$ form the boundary of a convex polygon (oriented clock- or counterclockwise).
2. $k \geq 2$.

Requirement:

1. Value type of *RandomAccessIterator* is $K::Point_2$ where K is a model for *Kernel*.
2. There is a global function $K::FT$ $CGAL::sqrt(K::FT)$ defined that computes the squareroot of a number.
3. *OutputIterator* accepts the value type of *RandomAccessIterator* as value type.

See Also

CGAL::maximum_area_inscribed_k_gon_2 page 2287
ExtremalPolygonTraits_2 page 2296
CGAL::Extremal_polygon_area_traits_2<K> page 2292
CGAL::Extremal_polygon_perimeter_traits_2<K> page 2294
CGAL::extremal_polygon_2 page 2291
CGAL::monotone_matrix_search page 2321

Implementation

The implementation uses monotone matrix search [AKM⁺87] and has a worst case running time of $O(k \cdot n + n \cdot \log n)$, where n is the number of vertices in P .

Example

The following code generates a random convex polygon p with ten vertices and computes the maximum perimeter inscribed five-gon of p .

```
#include <CGAL/Cartesian.h>
#include <CGAL/Polygon_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_convex_set_2.h>
#include <CGAL/extremal_polygon_2.h>
#include <iostream>
#include <vector>

typedef double FT;

struct Kernel : public CGAL::Cartesian<FT> {};

typedef Kernel::Point_2 Point;
typedef std::vector<int> Index_cont;
typedef CGAL::Polygon_2<Kernel> Polygon;
typedef CGAL::Random_points_in_square_2<Point> Generator;

int main() {

    int n = 10;
    int k = 5;

    // generate random convex polygon:
    Polygon p;
    CGAL::random_convex_set_2(n, std::back_inserter(p), Generator(1));
    std::cout << "Generated Polygon:\n" << p << std::endl;

    // compute maximum perimeter inccribed k-gon of p:
    Polygon k_gon;
    CGAL::maximum_perimeter_inscribed_k_gon_2(
        p.vertices_begin(), p.vertices_end(), k, std::back_inserter(k_gon));
    std::cout << "Maximum perimeter " << k << "-gon:\n"
        << k_gon << std::endl;

    return 0;
}
```

CGAL::extremal_polygon_2

— advanced —

Definition

The function *extremal_polygon_2* computes a maximal k -gon that can be inscribed into a given convex polygon. The criterion for maximality and some basic operations have to be specified in an appropriate traits class parameter.

```
#include <CGAL/extremal_polygon_2.h>
```

```
template < class RandomAccessIterator, class OutputIterator, class Traits >
OutputIterator      extremal_polygon_2(
                    RandomAccessIterator points_begin,
                    RandomAccessIterator points_end,
                    int k,
                    OutputIterator o,
                    Traits t)
```

computes a maximal (as specified by t) inscribed k -gon of the convex polygon described by $[points_begin, points_end)$, writes its vertices to o and returns the past-the-end iterator of this sequence.

Precondition:

1. the – at least three – points denoted by the range $[points_begin, points_end)$ form the boundary of a convex polygon (oriented clock- or counterclockwise).
2. $k \geq t.min_k()$.

Requirement:

1. *Traits* is a model for *ExtremalPolygonTraits_2*.
2. Value type of *RandomAccessIterator* is *Traits::Point_2*.
3. *OutputIterator* accepts *Traits::Point_2* as value type.

See Also

CGAL::maximum_area_inscribed_k_gon_2 page [2287](#)
 CGAL::maximum_perimeter_inscribed_k_gon_2 page [2289](#)
 ExtremalPolygonTraits_2 page [2296](#)
 CGAL::monotone_matrix_search page [2321](#)

Implementation

The implementation uses monotone matrix search [[AKM⁺87](#)] and has a worst case running time of $O(k \cdot n + n \cdot \log n)$, where n is the number of vertices in P .

— advanced —

CGAL::Extremal_polygon_area_traits_2<K>

- *advanced* -

Definition

The class `ExtremalPolygon` provides the types and operations needed to compute a maximum area k -gon P_k that can be inscribed into a given convex polygon P using the function `extremal_polygon_2`.

Requirements

The template parameter K is a model for *Kernel*.

Is Model for the Concepts

ExtremalPolygonTraits_2.....page 2296

Types

$$Extremal_polygon_area_traits_2<K>::FT \quad \text{typedef to } K::FT.$$

```
Extremal_polygon_area_traits_2<K>::Point_2 typedef to K::Point_2.
```

```
Extremal_polygon_area_traits_2<K>::Less_xy_2
```

```
typedef to K::Less_xy_2.
```

```
Extremal_polygon_area_traits_2<K>:: Orientation_2
```

```
typedef to K::Orientation_2.
```

Extremal_polygon_area_traits_2<K>:: Operation

AdaptableBinaryFunction class *op*: $Point_2 \times Point_2 \rightarrow FT$. For a fixed $Point_2$ *root*, *op*(*p*, *q*) returns twice the area of the triangle (*root*, *q*, *p*).

Operations

int *t.min_k()* *const* returns 3.

```

FT      t.init( const Point_2& p, const Point_2& q) const
                                returns FT(0).

```

Operation *t.operation(const Point_2& p) const*

returns *Operation* where *p* is the fixed *root* point.

```
template < class RandomAccessIterator, class OutputIterator >
OutputIterator      t.compute_min_k_gon( RandomAccessIterator points_begin,
                                         RandomAccessIterator points_end,
                                         FT& max_area,
                                         OutputIterator o) const
```

writes the vertices of [*points_begin*, *points_end*) forming a maximum area triangle rooted at *points_begin*[0] to *o* and returns the past-the-end iterator for that sequence ($= o + 3$).

Less_xy_2 *t.less_xy_2_object()*
Orientation_2 *t.orientation_2_object()*

See Also

CGAL::maximum_area_inscribed_k_gon_2 page [2287](#)
CGAL::maximum_perimeter_inscribed_k_gon_2 page [2289](#)
CGAL::extremal_polygon_2 page [2291](#)
CGAL::Extremal_polygon_perimeter_traits_2<K> page [2294](#)
ExtremalPolygonTraits_2 page [2296](#)

└────────── advanced ─────────┘

CGAL::Extremal_polygon_perimeter_traits_2<K>

— *advanced* —

Definition

The class provides the types and operations needed to compute a maximum perimeter k -gon P_k that can be inscribed into a given convex polygon P using the function *extremal_polygon_2*.

Requirements

The template parameter K is a model for *Kernel*.

Is Model for the Concepts

ExtremalPolygonTraits_2.....page [2296](#)

Types

Extremal_polygon_perimeter_traits_2<K>:: FT

typedef to $K::FT$.

Extremal_polygon_perimeter_traits_2<K>:: Point_2

typedef to $K::Point_2$.

Extremal_polygon_perimeter_traits_2<K>:: Less_xy_2

typedef to $K::Less_xy_2$.

Extremal_polygon_perimeter_traits_2<K>:: Orientation_2

typedef to $K::Orientation_2$.

Extremal_polygon_perimeter_traits_2<K>:: Operation

AdaptableBinaryFunction class *op*: $Point_2 \times Point_2 \rightarrow FT$. For a fixed *Point_2* *root*, *op*(*p*, *q*) returns $d(r, p) + d(p, q) - d(r, q)$ where d denotes the Euclidean distance.

Operations

int *t.min_k()* *const* returns 2.

FT $t.\text{init}(\text{const Point_2}\& p, \text{const Point_2}\& q) \text{ const}$

returns twice the Euclidean distance between p and q .

Operation *t.operation(const Point_2& p) const*

returns *Operation* where *p* is the fixed *root* point.

```
template < class RandomAccessIterator, class OutputIterator >
OutputIterator      t.compute_min_k_gon( RandomAccessIterator points_begin,
                                         RandomAccessIterator points_end,
                                         FT& max_area,
                                         OutputIterator o) const
```

writes the pair $(points_begin[0], p)$ where p is drawn from $[points_begin, points_end)$ such that the Euclidean distance between both points is maximized (maximum perimeter 2-gon rooted at $points_begin[0]$) to o and returns the past-the-end iterator for that sequence ($== o + 2$).

<i>Less_xy_2</i>	<i>t.less_xy_2_object()</i>
<i>Orientation_2</i>	<i>t.orientation_2_object()</i>

See Also

<i>CGAL::maximum_area_inscribed_k_gon_2</i>	page 2287
<i>CGAL::maximum_perimeter_inscribed_k_gon_2</i>	page 2289
<i>CGAL::extremal_polygon_2</i>	page 2291
<i>CGAL::Extremal_polygon_area_traits_2<K></i>	page 2292
<i>ExtremalPolygonTraits_2</i>	page 2296

_____ *advanced* _____

ExtremalPolygonTraits_2

— advanced —

Definition

The concept `ExtremalPolygonTraits_2` provides the types and operations needed to compute a maximal k -gon that can be inscribed into a given convex polygon.

Types

<code>ExtremalPolygonTraits_2:: FT</code>	model for <code>FieldNumberType</code> page 2530 .
<code>ExtremalPolygonTraits_2:: Point_2</code>	model for <code>Kernel::Point_2</code> page 408 .
<code>ExtremalPolygonTraits_2:: Less_xy_2</code>	model for <code>Kernel::Less_xy_2</code> page ?? .
<code>ExtremalPolygonTraits_2:: Orientation_2</code>	model for <code>Kernel::Orientation_2</code> page 403 .
<code>ExtremalPolygonTraits_2:: Operation</code>	AdaptableBinaryFunction class <code>op</code> : $Point_2 \times Point_2 \rightarrow FT$. Together with <code>init</code> this operation recursively defines the objective function to maximize. Let p and q be two vertices of a polygon P such that q precedes p in the oriented vertex chain of P starting with vertex <code>root</code> . Then <code>op(p,q)</code> returns the value by which an arbitrary sub-polygon of P with vertices from $[root, q]$ increases when p is added to it. E.g. in the maximum area case this is the area of the triangle $(root, q, p)$.

Operations

<code>int</code>	<code>t.min_k() const</code>	returns the minimal k for which a maximal k -gon can be computed. (e.g. in the maximum area case this is three.)
<code>FT</code>	<code>t.init(const Point_2& p, const Point_2& q) const</code>	returns the value of the objective function for a polygon consisting of the two points p and q . (e.g. in the maximum area case this is <code>FT(0)</code> .)
<code>Operation</code>	<code>t.operation(const Point_2& p) const</code>	return <code>Operation</code> where p is the fixed <code>root</code> point.

`template < class RandomAccessIterator, class OutputIterator >`

OutputIterator *t.compute_min_k_gon(RandomAccessIterator points_begin,*
 RandomAccessIterator points_end,
 FT& max_area,
 OutputIterator o) const

writes the points of [*points_begin*, *points_end*) forming a *min_k*()-gon rooted at *points_begin*[0] of maximal value to *o* and returns the past-the-end iterator for that sequence (*== o + min_k()*).

Less_xy_2 *t.less_xy_2_object()*
Orientation_2 *t.orientation_2_object()*

Notes

- *::Less_xy_2* and *::Orientation_2* are used for (expensive) precondition checking only. Therefore, they need not to be specified, in case that precondition checking is disabled.

Has Models

CGAL::Extremal_polygon_area_traits_2<K> page [2292](#)
CGAL::Extremal_polygon_perimeter_traits_2<K> page [2294](#)

See Also

CGAL::maximum_area_inscribed_k_gon_2 page [2287](#)
CGAL::maximum_perimeter_inscribed_k_gon_2 page [2289](#)
CGAL::extremal_polygon_2 page [2291](#)

└────────── *advanced* ─────────┘

CGAL::Largest_empty_iso_rectangle_2<T>

Definition

Given a set of points in the plane, the class *Largest_empty_iso_rectangle_2<T>* is a data structure that maintains an iso-rectangle with the largest area among all iso-rectangles that are inside a given bounding box(iso-rectangle), and that do not contain any point of the point set.

The class *Largest_empty_iso_rectangle_2<T>* expects a model of the concept *LargestEmptyIsoRectangleTraits_2* as its template argument.

```
#include <CGAL/Largest_empty_iso_rectangle_2.h>
```

Types

The class *Largest_empty_iso_rectangle_2<T>* defines the following types:

```
typedef T Traits;
```

```
typedef Traits::Point_2
```

```
Point_2;
```

```
typedef Traits::Iso_rectangle_2
```

```
Iso_rectangle_2;
```

The following iterator allows to enumerate the points. It is non mutable, bidirectional and its value type is *Point_2*. It is invalidated by any insertion or removal of a point.

```
Largest_empty_iso_rectangle_2<T>::const_iterator
```

Iterator over the points.

Creation

```
Largest_empty_iso_rectangle_2<T> l( Iso_rectangle_2 b);
```

Constructor. The iso-rectangle *b* is the bounding rectangle.

```
Largest_empty_iso_rectangle_2<T> l( const Point_2 p, const Point_2 q);
```

Constructor. The iso-rectangle whose lower left and upper right points are *p* and *q* respectively is the bounding rectangle.

```
Largest_empty_iso_rectangle_2<T> l;
```

Constructor. The iso-rectangle whose lower left point and upper right points are (0,0) and (1,1) respectively is the bounding rectangle.

Largest_empty_iso_rectangle_2<*T*> *l*(*const Largest_empty_iso_rectangle_2*<*Traits*> *tr*);

Copy constructor.

Operations

Assignment

Largest_empty_iso_rectangle_2<*T*>

l = *tr*

Access Functions

<i>Traits</i>	<i>l.traits()</i>	Returns a const reference to the traits object.
<i>const_iterator</i>	<i>l.begin()</i>	Returns an iterator to the beginning of the point set.
<i>const_iterator</i>	<i>l.end()</i>	Returns a past-the-end iterator for the point set.

Queries

Quadruple<*Point_2*, *Point_2*, *Point_2*, *Point_2*>

l.get_left_bottom_right_top()

Returns the four points that define the largest empty iso-rectangle. (Note that these points are not necessarily on a corner of an iso-rectangle.)

Iso_rectangle_2

l.get_largest_empty_iso_rectangle()

Returns the largest empty iso-rectangle. (Note that the two points defining the iso-rectangle are not necessarily part of the point set.)

Iso_rectangle_2

l.get_bounding_box() Returns the iso-rectangle passed in the constructor.

Insertion

void *l.insert(Point_2 p)* Inserts point *p* in the point set, if it is not already in the set.

void *l.push_back(Point_2 p)* Inserts point *p* in the point set, if it is not already in the set.

template < class InputIterator >

int *l.insert(InputIterator first, InputIterator last)*

Inserts the points in the range $[first, last)$. Returns the number of inserted points.

Requirements

The *value_type* of *first* and *last* is *Point*.

Removal

bool *l.remove(Point_2 p)* Removes point *p*. Returns false iff *p* is not in the point set.

void *l.clear()* Removes all points of *l*.

Implementation

The algorithm is an implementation of [Ori90]. The runtime of an insertion or a removal is $O(\log n)$. A query takes $O(n^2)$ worst case time and $O(n \log n)$ expected time. The working storage is $O(n)$.

LargestEmptyIsoRectangleTraits_2

Definition

The concept `LargestEmptyIsoRectangleTraits_2` describes the set of requirements to be fulfilled by any class used to instantiate the template parameter of the class `Largest_empty_iso_rectangle_2<T>`. This concept provides the types of the geometric primitives used in this class and some function object types for the required predicates on those primitives.

Types

`LargestEmptyIsoRectangleTraits_2::Point_2` The point type.

`LargestEmptyIsoRectangleTraits_2::Iso_rectangle_2`

The iso rectangle type.

`LargestEmptyIsoRectangleTraits_2::Compare_x_2`

Predicate object. Must provide the operator `Comparison_result operator()(Point_2 p, Point_2 q)` which returns *SMALLER*, *EQUAL* or *LARGER* according to the *x*-ordering of points *p* and *q*.

`LargestEmptyIsoRectangleTraits_2::Compare_y_2`

Predicate object. Must provide the operator `Comparison_result operator()(Point_2 p, Point_2 q)` which returns *SMALLER*, *EQUAL* or *LARGER* according to the *y*-ordering of points *p* and *q*.

`LargestEmptyIsoRectangleTraits_2::Less_x_2`

Predicate object. Must provide the operator `bool operator()(Point_2 p, Point_2 q)` which returns whether *p* is less than *q* according to their *x*-ordering.

`LargestEmptyIsoRectangleTraits_2::Less_y_2`

Predicate object. Must provide the operator `bool operator()(Point_2 p, Point_2 q)` which returns whether *p* is less than *q* according to their *y*-ordering.

Creation

Only a default constructor, copy constructor and an assignment operator are required. Note that further constructors can be provided.

`LargestEmptyIsoRectangleTraits_2 traits;` Default constructor.

`LargestEmptyIsoRectangleTraits_2 traits(LargestEmptyIsoRectangleTraits_2);`

Copy constructor

`LargestEmptyIsoRectangleTraits_2`

`traits = gtr`

Assignment operator.

Predicate functions

The following functions give access to the predicate and constructor objects.

<i>Compare_x_2</i>	<i>traits.compare_x_2_object()</i>
<i>Compare_y_2</i>	<i>traits.compare_y_2_object()</i>
<i>Less_x_2</i>	<i>traits.less_x_2_object()</i>
<i>Less_y_2</i>	<i>traits.less_y_2_object()</i>

Has Models

CGAL::Cartesian<R>
CGAL::Homogeneous<R>

See Also

CGAL::Largest_empty_iso_rectangle_2<Traits>

CGAL::all_furthest_neighbors_2

Definition

The function *all_furthest_neighbors_2* computes all furthest neighbors for the vertices of a convex polygon P , i.e. for each vertex v of P a vertex f_v of P such that the distance between v and f_v is maximized.

```
#include <CGAL/all_furthest_neighbors_2.h>
```

```
template < class RandomAccessIterator, class OutputIterator, class Traits >
OutputIterator      all_furthest_neighbors_2(
                    RandomAccessIterator points_begin,
                    RandomAccessIterator points_end,
                    OutputIterator o,
                    Traits t = Default_traits)
```

computes all furthest neighbors for the vertices of the convex polygon described by the range $[points_begin, points_end)$, writes their indices (relative to $points_begin$) to o ² and returns the past-the-end iterator of this sequence.

Precondition: The points denoted by the non-empty range $[points_begin, points_end)$ form the boundary of a convex polygon P (oriented clock- or counterclockwise).

The geometric types and operations to be used for the computation are specified by the traits class parameter t . This parameter can be omitted if *RandomAccessIterator* refers to a point type from a *Kernel*. In this case, the kernel is used as default traits class.

Requirement:

1. If t is specified explicitly, *Traits* is a model for *AllFurthestNeighborsTraits_2*.
2. Value type of *RandomAccessIterator* is *Traits::Point_2* or – if t is not specified explicitly – *K::Point_2* where K is a model for *Kernel*.
3. *OutputIterator* accepts *int* as value type.

See Also

AllFurthestNeighborsTraits_2 page [2305](#)
CGAL::monotone_matrix_search page [2321](#)

Implementation

The implementation uses monotone matrix search[AKM⁺87]. Its runtime complexity is linear in the number of vertices of P .

²i.e. the furthest neighbor of $points_begin[i]$ is $points_begin[i]$ -th number written to o

Example

The following code generates a random convex polygon p with ten vertices, computes all furthest neighbors and writes the sequence of their indices (relative to *points_begin*) to *cout* (e.g. a sequence of 4788911224 means the furthest neighbor of *points_begin*[0] is *points_begin*[4], the furthest neighbor of *points_begin*[1] is *points_begin*[7] etc.).

```
#include <CGAL/Cartesian.h>
#include <CGAL/Polygon_2.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_convex_set_2.h>
#include <CGAL/all_furthest_neighbors_2.h>
#include <CGAL/IO/Ostream_iterator.h>
#include <iostream>
#include <vector>

typedef double FT;

struct Kernel : public CGAL::Cartesian<FT> {};

typedef Kernel::Point_2 Point;
typedef std::vector<int> Index_cont;
typedef CGAL::Polygon_2<Kernel> Polygon;
typedef CGAL::Random_points_in_square_2<Point> Generator;
typedef CGAL::Ostream_iterator<int, std::ostream> Oiterator;

int main()
{
    // generate random convex polygon:
    Polygon p;
    CGAL::random_convex_set_2(10, std::back_inserter(p), Generator(1));

    // compute all furthest neighbors:
    CGAL::all_furthest_neighbors_2(p.vertices_begin(), p.vertices_end(),
                                   Oiterator(std::cout));

    std::cout << std::endl;

    return 0;
}
```


AllFurthestNeighborsTraits_2

Definition

The concept AllFurthestNeighborsTraits_2 defines types and operations needed to compute all furthest neighbors for the vertices of a convex polygon using the function *all_furthest_neighbors_2*.

Types

<i>AllFurthestNeighborsTraits_2:: FT</i>	model for FieldNumberType	page 2530.
<i>AllFurthestNeighborsTraits_2:: Point_2</i>	model for Kernel::Point_2	page 408.
<i>AllFurthestNeighborsTraits_2:: Compute_squared_distance_2</i>	model for Kernel::Compute_squared_distance_2 . . .	page ??.
<i>AllFurthestNeighborsTraits_2:: Less_xy_2</i>	model for Kernel::Less_xy_2	page ??.
<i>AllFurthestNeighborsTraits_2:: Orientation_2</i>	model for Kernel::Orientation_2	page 403.

Operations

The following member functions return function objects of the types listed above.
Compute_squared_distance_2

	<i>t.compute_squared_distance_2_object()</i>
<i>Less_xy_2</i>	<i>t.less_xy_2_object()</i>
<i>Orientation_2</i>	<i>t.orientation_2_object()</i>

Has Models

Cartesian<FieldNumberType> page ??, *Homogeneous<RingNumberType>* page ??,
Simple_cartesian<FieldNumberType> page ??, *Simple_homogeneous<RingNumberType>* page ??.

See Also

CGAL::all_furthest_neighbors_2 page 2303

Notes

- *::Less_xy_2* and *::Orientation_2* are used for (expensive) precondition checking only. Therefore, they need not to be specified, in case that precondition checking is disabled.

CGAL::Width_3<Traits>

Definition

Given a set of points $\mathcal{S} = \{p_1, \dots, p_n\}$ in \mathbb{R}^3 . The width of \mathcal{S} , denoted as $\mathcal{W}(\mathcal{S})$, is defined as the minimum distance between two parallel planes of support of $\text{conv}(\mathcal{S})$; where $\text{conv}(\mathcal{S})$ denotes the convex hull of \mathcal{S} . The width in direction \mathbf{d} , denoted as $\mathcal{W}_{\mathbf{d}}(\mathcal{S})$, is the distance between two parallel planes of support of $\text{conv}(\mathcal{S})$, which are orthogonal to \mathbf{d} .

Subject to the applications of the width algorithm, several objects might be interesting:

1. The two parallel planes of support such that the distance between them is as small as possible. These planes are called width-planes in further considerations.
2. The width $\mathcal{W}(\mathcal{S})$, i.e., the distance between the width-planes.
3. The direction \mathbf{d}_{opt} such that $\mathcal{W}(\mathcal{S}) = \mathcal{W}_{\mathbf{d}_{opt}}(\mathcal{S})$

Note: There might be several optimal build directions. Hence neither the width-planes nor the direction \mathbf{d}_{opt} are unique – only the width is.

```
#include <CGAL/Width_3.h>
```

Requirements

The template parameter *Traits* is a model for *WidthTraits_3*. We provide the model *Width_default_traits_3<Kernel>* based on a three-dimensional CGAL kernel.

Types

<i>Width_3<Traits>::Traits</i>		traits class.
<i>typedef typename Traits::Point_3</i>	<i>Point_3;</i>	point type.
<i>typedef typename Traits::Plane_3</i>	<i>Plane_3;</i>	plane type.
<i>typedef typename Traits::Vector_3</i>	<i>Vector_3;</i>	vector type.
<i>typedef typename Traits::RT</i>	<i>RT;</i>	algebraic ring type.
<i>typedef typename Traits::ChullTraits</i>	<i>ChullTraits;</i>	traits class for the 3D convex hull algorithm.

Creation

```
template < class InputIterator >
Width_3<Traits> width( InputIterator first, InputIterator beyond);
```

creates a variable *width* initialized to the width of \mathcal{S} – with \mathcal{S} being the set of points in the range $[first, beyond)$.

Requirement: The value type of *InputIterator* is *Point_3*.

```
template < class Polyhedron >
Width_3<Traits> width( Polyhedron& P);
```

creates a variable *width* initialized to the width of the polyhedron *P*. Note that the vertex point coordinates are altered!

Precondition: *P* is a convex polyhedron.

Requirement: *Polyhedron* is a *CGAL::Polyhedron_3* with facets supporting plane equations where *Polyhedron::Point_3* \equiv *Point_3* and *Polyhedron::Plane_3* \equiv *Plane_3*.

Access Functions

```
void          width.get_squared_width( RT& width_num, RT& width_denom)
```

returns the squared width. For the reason of exact computation not the width itself is stored, but the *squared* width as a fraction: The numerator in *width_num* and the denominator in *width_denom*. The width of the point set \mathcal{S} is $\sqrt{\frac{width_num}{width_denom}}$.

```
void          width.get_width_planes( Plane_3& e1, Plane_3& e2)
```

The planes *e1* and *e2* are the two parallel supporting planes, which distance is minimal (among all such planes).

```
void          width.get_width_coefficients( RT& A, RT& B, RT& C, RT& D, RT& K)
```

The returned coefficients *A,B,C,D,K* have the property that width-plane *e1* is given by the equation $Ax + By + Cz + D = 0$ and width-plane *e2* by $Ax + By + Cz + K = 0$.

```
Vector_3      width.get_build_direction()
```

returns a direction \mathbf{d}_{opt} such that the width-planes *e1* and *e2* are perpendicular to \mathbf{d}_{opt} . The width of the point set is minimal in this direction.

```
void          width.get_all_build_directions( std::vector<Vector_3>& dir)
```

All the build directions are stored in the vector *dir*. It might happen that a certain body has several different build directions, but it is also possible to have only one build direction.

```
int           width.get_number_of_optimal_solutions()
```

returns the number of optimal solutions, i.e., the number of optimal build directions.

See Also

CGAL::Width_default_traits_3<K> page [2310](#)
WidthTraits_3 page [2312](#)

Implementation

Since the width of the point set S and the width of the convex hull of S ($\text{conv}(S)$) is the same, the algorithm uses the 3D convex hull algorithm CGAL provides.

The width-algorithm is not incremental and therefore inserting and erasing points cause not an ‘automatic’ update of the width. Instead you have to run the width-algorithm again even if the point set is extended by only one new point.

————— *advanced* —————

Large Numbers. Because there is no need for dividing values during the algorithm, the numbers can get really huge (all the computations are made using a lot of multiplications). Therefore it is strongly recommended to use a number type that can handle numbers of arbitrary length (e.g., *leda_integer* in combination with the homogeneous representation of the points). But these large numbers have a disadvantage: Operations on them are slower as greater the number gets. Therefore it is possible to shorten the numbers by using the compiler flag `-DSIMPLIFY`. For using this option it is required that the underlying number type provides the ‘modulo’ operation.

Information Output during the Computations. If during the algorithm the program should output some information (e.g., during the debugging phase) you can turn on the output information by giving the compiler flag `DEBUG`. In the file `width_assertions.h` you can turn on/off the output of some functions and additional informations by changing the defined values from 0 (no output) to 1 (output available). But then it is required that the `<<-operator` has to been overloaded for *Point_3*, *Plane_3*, *Vector_3* and *RT*.

————— *advanced* —————

Example

```
#include <CGAL/Homogeneous.h>
#include <CGAL/Width_default_traits_3.h>
#include <CGAL/Width_3.h>
#include <iostream>
#include <vector>
#include <CGAL/leda_integer.h>

typedef leda_integer          RT;
typedef CGAL::Homogeneous<RT> Kernel;
typedef Kernel::Point_3      Point_3;
typedef Kernel::Plane_3      Plane_3;
typedef CGAL::Width_default_traits_3<Kernel> Width_traits;
typedef CGAL::Width_3<Width_traits> Width;

int main() {
    // Create a simplex using homogeneous integer coordinates
```

```

std::vector<Point_3> points;
points.push_back( Point_3(2,0,0,1));
points.push_back( Point_3(0,1,0,1));
points.push_back( Point_3(0,0,1,1));
points.push_back( Point_3(0,0,0,1));

// Compute width of simplex
Width simplex( points.begin(), points.end());

// Output of squared width, width-planes, and optimal direction
RT wnum, wdenom;
simplex.get_squared_width( wnum, wdenom);
std::cout << "Squared Width: " << wnum << "/" << wdenom << std::endl;

std::cout << "Direction: " << simplex.get_build_direction() << std::endl;

Plane_3 e1, e2;
std::cout << "Planes: E1: " << e1 << ". E2: " << e2 <<std::endl;

std::cout << "Number of optimal solutions: "
          << simplex.get_number_of_optimal_solutions() << std::endl;
return(0);
}

```

CGAL::Width_default_traits_3<K>

Definition

The class *Width_default_traits_3<K>* is a traits class for *Width_3<Traits>* using the three-dimensional CGAL kernel.

```
#include <CGAL/Width_default_traits_3.h>
```

Requirements

The template parameter *K* is a model for *Kernel*

Is Model for the Concepts

WidthTraits_3 page [2312](#)

Types

```
typedef typename K::Point_3      Point_3;
typedef typename K::Plane_3      Plane_3;
typedef typename K::Vector_3     Vector_3;
typedef typename K::RT           RT;
typedef Convex_hull_traits_3<K>  ChullTraits;
```

Creation

Width_default_traits_3<K> traits; default constructor.

Operations

<i>RT</i>	<i>traits.get_hx(Point_3 p)</i>	returns the homogeneous <i>x</i> -coordinate of point <i>p</i> .
<i>RT</i>	<i>traits.get_hy(Point_3 p)</i>	returns the homogeneous <i>y</i> -coordinate of point <i>p</i> .
<i>RT</i>	<i>traits.get_hz(Point_3 p)</i>	returns the homogeneous <i>z</i> -coordinate of point <i>p</i> .
<i>RT</i>	<i>traits.get_hw(Point_3 p)</i>	returns the homogenizing coordinate of point <i>p</i> .
<i>void</i>	<i>traits.get_point_coordinates(Point_3 p, RT& px, RT& py, RT& pz, RT& ph)</i>	returns all homogeneous coordinates of point <i>p</i> at once.
<i>RT</i>	<i>traits.get_a(Plane_3 f)</i>	returns the first coefficient of plane <i>f</i> .
<i>RT</i>	<i>traits.get_b(Plane_3 f)</i>	returns the second coefficient of plane <i>f</i> .
<i>RT</i>	<i>traits.get_c(Plane_3 f)</i>	returns the third coefficient of plane <i>f</i> .
<i>RT</i>	<i>traits.get_d(Plane_3 f)</i>	returns the fourth coefficient of plane <i>f</i> .
<i>void</i>	<i>traits.get_plane_coefficients(Plane_3 f, RT& a, RT& b, RT& c, RT& d)</i>	returns all four plane coefficients of <i>f</i> at once.

See Also

2311

WidthTraits_3

Definition

This concept defines the requirements for traits classes of *Width_3<Traits>*.

Types

<i>WidthTraits_3:: Point_3</i>	The point type. The (in)equality tests must be available. Access to the point coordinates is done via the <i>get_..()</i> functions. Constructing a point is done with the <i>make_point()</i> operation.
<i>WidthTraits_3:: Plane_3</i>	The plane type. Access to the coefficients of the plane is made via the <i>get_..()</i> functions. Constructing a plane is done with the <i>make_plane()</i> operation.
<i>WidthTraits_3:: Vector_3</i>	The vector type. There is no need to access the coefficients of a vector; only constructing is required and is done with the <i>make_vector</i> operation.
<i>WidthTraits_3:: ChullTraits</i>	The traits class for using the convex hull algorithm. It must be a model of the concept <i>ConvexHullTraits_3</i> . This class is used only if the width is computed from a set of points.
<i>WidthTraits_3:: RT</i>	Ring type numbers. Internally all numbers are treated as ring type numbers, i.e., neither $/$ -operator nor $\sqrt{}$ nor other inexact operations are used. But because the algorithm does not use any divisions, but multiplication instead, the numbers can get really big. Therefore it is recommended to use a ring type number, that provides values of arbitrary length. Furthermore it is assumed that the underlying number type of <i>Point_3</i> , <i>Plane_3</i> and <i>Vector_3</i> equals <i>RT</i> .

Notes: If you want to compute the width of a *polyhedron* then you have to make sure that the point type in the traits class and the point type in the polyhedron class are the same! The same holds for *Traits::Plane_3* and *Polyhedron::Plane_3*.

Creation

Only a default constructor is required.

```
WidthTraits_3 traits;
```

Operations

Whatever the coordinates of the points are, it is required for the width-algorithm to have access to the homogeneous representation of points.

<i>RT</i>	<i>get_hx(Point_3 p)</i>	returns the homogeneous <i>x</i> -coordinate of point <i>p</i> .
<i>RT</i>	<i>get_hy(Point_3 p)</i>	returns the homogeneous <i>y</i> -coordinate of point <i>p</i> .
<i>RT</i>	<i>get_hz(Point_3 p)</i>	returns the homogeneous <i>z</i> -coordinate of point <i>p</i> .
<i>RT</i>	<i>get_hw(Point_3 p)</i>	returns the homogenizing coordinate of point <i>p</i> .
<i>void</i>	<i>get_point_coordinates(Point_3 p, RT& px, RT& py, RT& pz, RT& ph)</i>	returns all homogeneous coordinates of point <i>p</i> at once.
<i>RT</i>	<i>get_a(Plane_3 f)</i>	returns the first coefficient of plane <i>f</i> .
<i>RT</i>	<i>get_b(Plane_3 f)</i>	returns the second coefficient of plane <i>f</i> .

<i>RT</i>	<i>get_c(Plane_3 f)</i>	returns the third coefficient of plane <i>f</i> .
<i>RT</i>	<i>get_d(Plane_3 f)</i>	returns the fourth coefficient of plane <i>f</i> .
<i>void</i>	<i>get_plane_coefficients(Plane_3 f, RT& a, RT& b, RT& c, RT& d)</i>	returns all four plane coefficients of <i>f</i> at once.
<i>Point_3</i>	<i>make_point(RT hx, RT hy, RT hz, RT hw)</i>	returns a point of type <i>Point_3</i> with homogeneous coordinates <i>hx</i> , <i>hy</i> , <i>hz</i> and <i>hw</i> .
<i>Plane_3</i>	<i>make_plane(RT a, RT b, RT c, RT d)</i>	returns a plane of type <i>Plane_3</i> with coefficients <i>a</i> , <i>b</i> , <i>c</i> and <i>d</i> .
<i>Vector_3</i>	<i>make_vector(RT a, RT b, RT c)</i>	returns a vector of type <i>Vector_3</i> with the four homogeneous coefficients <i>a</i> , <i>b</i> , <i>c</i> and 1.

Has Models

CGAL::Width_default_traits_3<K> page [2310](#)

See Also

CGAL::Width_3<Traits> page [2306](#)

CGAL::Polytope_distance_d<Traits>

Definition

An object of the class *Polytope_distance_d<Traits>* represents the (squared) distance between two convex polytopes, given as the convex hulls of two finite point sets in d -dimensional Euclidean space \mathbb{E}_d . For point sets P and Q we denote by $pd(P, Q)$ the distance between the convex hulls of P and Q . Note that $pd(P, Q)$ can degenerate, i.e. $pd(P, Q) = \infty$ if P or Q is empty.

Two inclusion-minimal subsets S_P of P and S_Q of Q with $pd(S_P, S_Q) = pd(P, Q)$ are called *pair of support sets*, the points in S_P and S_Q are the *support points*. A pair of support sets has size at most $d + 2$ (by size we mean $|S_P| + |S_Q|$). The distance between the two polytopes is *realized* by a pair of points p and q lying in the convex hull of S_P and S_Q , respectively, i.e. $\sqrt{\|p - q\|} = pd(P, Q)$. In general, neither the support sets nor the realizing points are necessarily unique.

The underlying algorithm can cope with all kinds of input, e.g. P and Q may be in non-convex position or points may occur more than once. The algorithm computes a pair of support sets S_P and S_Q with realizing points p and q which remain fixed until the next set, insert, or clear operation.

```
#include <CGAL/Polytope_distance_d.h>
```

Requirements

The template parameter *Traits* is a model for *OptimisationDTraits*.

We provide the models *Optimisation_d_traits_2*, *Optimisation_d_traits_3*, and *Optimisation_d_traits_d* using the two-, three-, and d -dimensional CGAL kernel, respectively.

Types

Polytope_distance_d<Traits>::Point typedef to *Traits::Point_d*. Point type used to represent the input points.

Polytope_distance_d<Traits>::FT typedef to *Traits::FT*. Number type used to return the squared distance between the two polytopes.

Polytope_distance_d<Traits>::ET typedef to *Traits::ET*. Number type used to do the exact computations in the underlying solver for quadratic programs (cf. **Implementation**).

Polytope_distance_d<Traits>::Point_iterator

non-mutable model of the STL concept *RandomAccessIterator* with value type *Point*. Used to access the points of the two polytopes.

Polytope_distance_d<Traits>::Support_point_iterator

non-mutable model of the STL concept *RandomAccessIterator* with value type *Point*. Used to access the support points.

Polytope_distance_d<Traits>:: *Coordinate_iterator*

non-mutable model of the STL concept *RandomAccessIterator* with value type *ET*. Used to access the coordinates of the realizing points.

Creation

```
Polytope_distance_d<Traits> poly_dist( Traits traits = Traits(),
                                         int verbose = 0,
                                         std::ostream& stream = std::cout)
```

initializes *poly_dist* to *pd*(\emptyset, \emptyset).

```
template < class InputIterator1, class InputIterator2 >
Polytope_distance_d<Traits> poly_dist( InputIterator1 p_first,
                                         InputIterator1 p_last,
                                         InputIterator2 q_first,
                                         InputIterator2 q_last,
                                         Traits traits = Traits(),
                                         int verbose = 0,
                                         std::ostream& stream = std::cout)
```

initializes *poly_dist* to *pd*(*P*, *Q*) with *P* and *Q* being the sets of points in the range [*p_first*, *p_last*) and [*q_first*, *q_last*), respectively.

Requirement: The value type of *InputIterator1* and *InputIterator2* is *Point*.

Precondition: All points have the same dimension.

— advanced —

If *verbose* is set to 1, 2, or 3 then some, more, or full verbose output of the underlying solver for quadratic programs is written to *stream*, resp.

— advanced —

Access Functions

int *poly_dist.ambient_dimension()*

returns the dimension of the points in *P* and *Q*. If *poly_dist* is *pd*(\emptyset, \emptyset), the ambient dimension is -1 .

int *poly_dist.number_of_points()*

returns the number of all points of *poly_dist*, i.e. $|P| + |Q|$.

int *poly_dist.number_of_points_p()*

returns the number of points in *P*.

int *poly_dist.number_of_points_q()*

returns the number of points in *Q*.

<i>int</i>	<i>poly_dist.number_of_support_points()</i> returns the number of support points of <i>poly_dist</i> , i.e. $ S_P + S_Q $.
<i>int</i>	<i>poly_dist.number_of_support_points_p()</i> returns the number of support points in S_P .
<i>int</i>	<i>poly_dist.number_of_support_points_q()</i> returns the number of support points in S_Q .
<i>Point_iterator</i>	<i>poly_dist.points_p_begin()</i> returns an iterator referring to the first point in P .
<i>Point_iterator</i>	<i>poly_dist.points_p_end()</i> returns the corresponding past-the-end iterator.
<i>Point_iterator</i>	<i>poly_dist.points_q_begin()</i> returns an iterator referring to the first point in Q .
<i>Point_iterator</i>	<i>poly_dist.points_q_end()</i> returns the corresponding past-the-end iterator.
<i>Support_point_iterator</i>	<i>poly_dist.support_points_p_begin()</i> returns an iterator referring to the first support point in S_P .
<i>Support_point_iterator</i>	<i>poly_dist.support_points_p_end()</i> returns the corresponding past-the-end iterator.
<i>Support_point_iterator</i>	<i>poly_dist.support_points_q_begin()</i> returns an iterator referring to the first support point in S_Q .
<i>Support_point_iterator</i>	<i>poly_dist.support_points_q_end()</i> returns the corresponding past-the-end iterator.
<i>Point</i>	<i>poly_dist.realizing_point_p()</i> returns the realizing point of P . <i>Requirement:</i> An implicit conversion from ET to RT is available. <i>Precondition:</i> $pd(P, Q)$ is finite.
<i>Point</i>	<i>poly_dist.realizing_point_q()</i> returns the realizing point of Q . <i>Requirement:</i> An implicit conversion from ET to RT is available. <i>Precondition:</i> $pd(P, Q)$ is finite.
<i>FT</i>	<i>poly_dist.squared_distance()</i> returns the squared distance of <i>poly_dist</i> , i.e. $(pd(P, Q))^2$. <i>Requirement:</i> An implicit conversion from ET to RT is available. <i>Precondition:</i> $pd(P, Q)$ is finite.

<i>Coordinate_iterator</i>	<i>poly_dist.realizing_point_p_coordinates_begin()</i>	returns an iterator referring to the first coordinate of the realizing point of P . <i>Note:</i> The coordinates have a rational representation, i.e. the first d elements of the iterator range are the numerators and the $(d+1)$ -st element is the common denominator.
<i>Coordinate_iterator</i>	<i>poly_dist.realizing_point_p_coordinates_end()</i>	returns the corresponding past-the-end iterator.
<i>Coordinate_iterator</i>	<i>poly_dist.realizing_point_q_coordinates_begin()</i>	returns an iterator referring to the first coordinate of the realizing point of Q . <i>Note:</i> The coordinates have a rational representation, i.e. the first d elements of the iterator range are the numerators and the $(d+1)$ -st element is the common denominator.
<i>Coordinate_iterator</i>	<i>poly_dist.realizing_point_q_coordinates_end()</i>	returns the corresponding past-the-end iterator.
<i>ET</i>	<i>poly_dist.squared_distance_numerator()</i>	returns the numerator of the squared distance of <i>poly_dist</i> .
<i>ET</i>	<i>poly_dist.squared_distance_denominator()</i>	returns the denominator of the squared distance of <i>poly_dist</i> .

Predicates

<i>bool</i>	<i>poly_dist.is_finite()</i>	returns <i>true</i> , if $pd(P, Q)$ is finite, i.e. none of the two polytopes is empty.
<i>bool</i>	<i>poly_dist.is_zero()</i>	returns <i>true</i> , if $pd(P, Q)$ is zero, i.e. the two polytopes intersect (this implies degeneracy).
<i>bool</i>	<i>poly_dist.is_degenerate()</i>	returns <i>true</i> , iff $pd(P, Q)$ is degenerate, i.e. $pd(P, Q)$ is not finite.

Modifiers

<i>void</i>	<i>poly_dist.clear()</i>	resets <i>poly_dist</i> to $pd(\emptyset, \emptyset)$.
<i>template < class InputIterator1, class InputIterator2 ></i> <i>void</i>	<i>poly_dist.set(InputIterator1 p_first,</i> <i> InputIterator1 p_last,</i> <i> InputIterator2 q_first,</i>	

InputIterator2 q_last)

sets *poly_dist* to $pd(P, Q)$ with P and Q being the sets of points in the ranges $[p_first, p_last)$ and $[q_first, q_last)$, respectively.

Requirement: The value type of *InputIterator1* and *InputIterator2* is *Point*.

Precondition: All points have the same dimension.

template < class InputIterator >

void poly_dist.set_p(InputIterator p_first, InputIterator p_last)

sets *poly_dist* to $pd(P, Q)$ with P being the set of points in the range $[p_first, p_last)$ (Q remains unchanged).

Requirement: The value type of *InputIterator* is *Point*.

Precondition: All points in P have dimension *poly_dist.ambient_dimension()* if Q is not empty.

template < class InputIterator >

void poly_dist.set_q(InputIterator q_first, InputIterator q_last)

sets *poly_dist* to $pd(P, Q)$ with Q being the set of points in the range $[q_first, q_last)$ (P remains unchanged).

Requirement: The value type of *InputIterator* is *Point*.

Precondition: All points in Q have dimension *poly_dist.ambient_dimension()* if P is not empty.

void poly_dist.insert_p(Point p)

inserts p into P .

Precondition: The dimension of p equals *poly_dist.ambient_dimension()* if *poly_dist* is not $pd(0, 0)$.

void poly_dist.insert_q(Point q)

inserts q into Q .

Precondition: The dimension of q equals *poly_dist.ambient_dimension()* if *poly_dist* is not $pd(0, 0)$.

template < class InputIterator1, class InputIterator2 >

*void poly_dist.insert(InputIterator1 p_first,
InputIterator1 p_last,
InputIterator2 q_first,*

InputIterator2 q_last)

inserts the points in the range $[p_first, p_last)$ and $[q_first, q_last)$ into P and Q , respectively, and recomputes the (squared) distance.

Requirement: The value type of *InputIterator1* and *InputIterator2* is *Point*.

Precondition: All points have the same dimension. If *poly_dist* is not $pd(0,0)$, this dimension must be equal to *poly_dist.ambient_dimension()*.

template < class InputIterator >

void poly_dist.insert_p(InputIterator p_first, InputIterator p_last)

inserts the points in the range $[p_first, p_last)$ into P and recomputes the (squared) distance (Q remains unchanged).

Requirement: The value type of *InputIterator* is *Point*.

Precondition: All points have the same dimension. If *poly_dist* is not empty, this dimension must be equal to *poly_dist.ambient_dimension()*.

template < class InputIterator >

void poly_dist.insert_q(InputIterator q_first, InputIterator q_last)

inserts the points in the range $[q_first, q_last)$ into Q and recomputes the (squared) distance (P remains unchanged).

Requirement: The value type of *InputIterator* is *Point*.

Precondition: All points have the same dimension. If *poly_dist* is not empty, this dimension must be equal to *poly_dist.ambient_dimension()*.

Validity Check

An object *poly_dist* is valid, iff ...

- *poly_dist* contains all points of its defining set P ,
- *poly_dist* is the smallest sphere containing its support set S , and
- S is minimal, i.e. no support point is redundant.

bool

poly_dist.is_valid(bool verbose = false, int level = 0)

returns *true*, iff *poly_dist* is valid. If *verbose* is *true*, some messages concerning the performed checks are written to standard error stream. The second parameter *level* is not used, we provide it only for consistency with interfaces of other classes.

Miscellaneous

`const Traits&` `poly_dist.traits()` returns a const reference to the traits class object.

I/O

`std::ostream&` `std::ostream& os << poly_dist`

writes `poly_dist` to output stream `os`.
Requirement: The output operator is defined for `Point_d`.

`std::istream&` `std::istream& is >> poly_dist&`

reads `poly_dist` from input stream `is`.
Requirement: The input operator is defined for `Point_d`.

See Also

`CGAL::Optimisation_d_traits_2<K,ET,NT>` page [2279](#)
`CGAL::Optimisation_d_traits_3<K,ET,NT>` page [2281](#)
`CGAL::Optimisation_d_traits_d<K,ET,NT>` page [2283](#)
`OptimisationDTraits` page [2285](#)

Implementation

The problem of finding the distance between two convex polytopes given as the convex hulls of two finite point sets can be formulated as an optimization problem with linear constraints and a convex quadratic objective function. The solution is obtained using our exact solver for quadratic programs [\[GS00\]](#).

The creation time is almost always linear in the number of points. Access functions and predicates take constant time, inserting a point might take up to linear time. The clear operation and the check for validity each take linear time.

CGAL::monotone_matrix_search

advanced

Definition

The function *monotone_matrix_search* computes the maxima for all rows of a totally monotone matrix.

More precisely, monotony for matrices is defined as follows.

Let K be a totally ordered set, $M \in K^{(n,m)}$ a matrix over K and for $0 \leq i < n$:

$$rmax_M(i) := \left\{ \min_{0 \leq j < m} j \mid M[i, j] = \max_{0 \leq k < m} M[i, k] \right\}$$

the (leftmost) column containing the maximum entry in row i . M is called monotone, iff

$$\forall 0 \leq i_1 < i_2 < n : rmax_M(i_1) \leq rmax_M(i_2) .$$

M is totally monotone, iff all of its submatrices are monotone (or equivalently: iff all 2×2 submatrices are monotone).

```
#include <CGAL/monotone_matrix_search.h>
```

```
template < class Matrix, class RandomAccessIC, class Compare_strictly >
void    monotone_matrix_search(
        Matrix m,
        RandomAccessIC t,
        Compare_strictly compare_strictly = less< Matrix::Value >())
```

computes the maximum (as specified by *compare_strictly*) entry for each row of m and writes the corresponding column to t , i.e. $t[i]$ is set to the index of the column containing the maximum element in row i . The maximum m_r of a row r is the leftmost element for which *compare_strictly*(m_r, x) is false for all elements x in r .

Precondition: t points to a structure of size at least $m.number_of_rows()$

Requirement:

1. *Matrix* is a model for *MonotoneMatrixSearchTraits*.
2. Value type of *RandomAccessIC* is *int*.
3. If *compare_strictly* is defined, it is an adaptable binary function: $Matrix::Value \times Matrix::Value \rightarrow bool$ describing a strict (non-reflexive) total ordering on *Matrix::Value*.

See Also

MonotoneMatrixSearchTraits page [2325](#)
 CGAL::all_furthest_neighbors_2 page [2303](#)
 CGAL::maximum_area_inscribed_k_gon_2 page [2287](#)
 CGAL::maximum_perimeter_inscribed_k_gon_2 page [2289](#)
 CGAL::extremal_polygon_2 page [2291](#)

Implementation

The implementation uses an algorithm by Aggarwal et al.[AKM⁺87]. The runtime is linear in the number of rows and columns of the matrix.

————— *advanced* —————

CGAL::Dynamic_matrix<M>

advanced

Definition

The class *Dynamic_matrix*<*M*> is an adaptor for an arbitrary matrix class *M* to provide the dynamic operations needed for monotone matrix search.

Requirements

M is a model for *BasicMatrix*.

`#include <CGAL/Dynamic_matrix.h>`

Is Model for the Concepts

MonotoneMatrixSearchTraits page [2325](#)
BasicMatrix page [2327](#)

Creation

Dynamic_matrix<*M*> *d*(*M* *m*); initializes *d* to *m*. *m* is *not* copied, we only store a reference.

Operations

<i>int</i>	<i>d.number_of_columns()</i>	returns the number of columns.
<i>int</i>	<i>d.number_of_rows()</i>	returns the number of rows.
<i>Entry</i>	<i>d(int row, int column)</i>	<p>returns the entry at position (<i>row</i>, <i>column</i>).</p> <p><i>Precondition:</i></p> <p>$0 \leq \text{row} < \text{number_of_rows}()$ and</p> <p>$0 \leq \text{column} < \text{number_of_columns}()$.</p>
<i>void</i>	<i>d.replace_column(int old, int new)</i>	<p>replace column <i>old</i> with column number <i>new</i>.</p> <p><i>Precondition:</i></p> <p>$0 \leq \text{old}, \text{new} < \text{number_of_columns}()$.</p>

*Matrix** *d.extract_all_even_rows()*

returns a new **Matrix** consisting of all rows of *d* with even index, (i.e. first row is row 0 of *d*, second row is row 2 of *d* etc.).
Precondition: *number_of_rows()* > 0.

void *d.shrink_to_quadratic_size()*

deletes the rightmost columns, such that *d* becomes quadratic.
Precondition:
number_of_columns() ≥ *number_of_rows()*.
Postcondition:
number_of_rows() == *number_of_columns()*.

See Also

CGAL::monotone_matrix_search page [2321](#)
MonotoneMatrixSearchTraits page [2325](#)
BasicMatrix page [2327](#)

Implementation

All operations take constant time except for *extract_all_even_rows* which needs time linear in the number of rows.

_____ *advanced* _____

MonotoneMatrixSearchTraits

advanced

Definition

The concept `MonotoneMatrixSearchTraits` is a refinement of *BasicMatrix* and defines types and operations needed to compute the maxima for all rows of a totally monotone matrix using the function *monotone_matrix_search*.

Types

MonotoneMatrixSearchTraits::Value The type of a matrix entry.

Operations

int *m.number_of_columns() const*
returns the number of columns.

int *m.number_of_rows() const*
returns the number of rows.

Entry *m.operator()(int row, int column) const*
returns the entry at position (*row*, *column*).
Precondition:
 $0 \leq \text{row} < \text{number_of_rows}()$ and
 $0 \leq \text{column} < \text{number_of_columns}()$.

void *m.replace_column(int old, int new)*
replace column *old* with column number *new*.
Precondition:
 $0 \leq \text{old}, \text{new} < \text{number_of_columns}()$.

*Matrix** *m.extract_all_even_rows() const*
returns a new `Matrix` consisting of all rows of *m* with even index, (i.e. first row is row 0 of *m*, second row is row 2 of *m* etc.).
Precondition: *number_of_rows()* > 0.

void *m.shrink_to_quadratic_size()*

deletes the rightmost columns, such that *m* becomes quadratic.

Precondition:

number_of_columns() ≥ number_of_rows().

Postcondition:

number_of_rows() == number_of_columns().

Notes

- For the sake of efficiency (and in order to achieve the time bounds claimed for *monotone_matrix_search*), all these operations have to be realized in constant time – except for *extract_all_even_rows* which may take linear time.
- There is an adaptor *Dynamic_matrix* that can be used to add most of the functionality described above to arbitrary matrix classes.

Has Models

CGAL::Dynamic_matrix<M> page [2323](#)

See Also

CGAL::monotone_matrix_search page [2321](#)

└────────── *advanced* ─────────┘

BasicMatrix

advanced

Definition

A class has to provide the following types and operations in order to be a model for *BasicMatrix*.

Types

BasicMatrix::Value The type of a matrix entry. It has to define a copy constructor.

Operations

int *m.number_of_columns() const*
returns the number of columns.

int *m.number_of_rows() const*
returns the number of rows.

Entry *m.operator()(int row, int column) const*
returns the entry at position (*row*, *column*).
Precondition:
 $0 \leq \textit{row} < \textit{number_of_rows}()$ and
 $0 \leq \textit{column} < \textit{number_of_columns}()$.

Has Models

CGAL::Dynamic_matrix<M> page [2323](#)

See Also

MonotoneMatrixSearchTraits page [2325](#)
SortedMatrixSearchTraits page [2333](#)

advanced

CGAL::sorted_matrix_search

— advanced —

Definition

The function *sorted_matrix_search* selects the smallest entry in a set of sorted matrices that fulfills a certain feasibility criterion.

More exactly, a matrix $M = (m_{ij}) \in S^{r \times l}$ (over a totally ordered set S) is sorted, iff

$$\begin{aligned} \forall 1 \leq i \leq r, 1 \leq j < l : m_{ij} \leq m_{i(j+1)} \text{ and} \\ \forall 1 \leq i < r, 1 \leq j \leq l : m_{ij} \leq m_{(i+1)j} . \end{aligned}$$

Now let \mathcal{M} be a set of n sorted matrices over S and f be a monotone predicate on S , i.e.

$$f : S \longrightarrow \text{bool} \quad \text{with} \quad f(r) \implies \forall t \in S, t > r : f(t) .$$

If we assume there is any feasible element in one of the matrices in \mathcal{M} , there certainly is a smallest such element. This is the one we are searching for.

The feasibility test as well as some other parameters can (and have to) be customized through a traits class.

```
#include <CGAL/sorted_matrix_search.h>
```

```
template < class RandomAccessIterator, class Traits >
Traits::Value sorted_matrix_search( RandomAccessIterator f, RandomAccessIterator l, Traits t)
```

returns the element x in one of the sorted matrices from the range $[f, l)$, for which *t.is_feasible(x)* is true and *t.compare(x, y)* is true for all other y values from any matrix for which *t.is_feasible(y)* is true.

Precondition:

1. All matrices in $[f, l)$ are sorted according to *Traits::compare_non_strictly*.
2. There is at least one entry x in a matrix $M \in [f, l)$ for which *Traits::is_feasible(x)* is true.

Requirement:

1. *Traits* is a model for *SortedMatrixSearchTraits*.
2. Value type of *RandomAccessIterator* is *Traits::Matrix*.

See Also

SortedMatrixSearchTraits page [2333](#)

Implementation

The implementation uses an algorithm by Frederickson and Johnson[FJ83, FJ84] and runs in $O(n \cdot k + f \cdot \log(n \cdot k))$, where n is the number of input matrices, k denotes the maximal dimension of any input matrix and f the time needed for one feasibility test.

Example

In the following program we build a random vector $a = (a_i)_{i=1, \dots, 5}$ (elements drawn uniformly from $\{0, \dots, 99\}$) and construct a Cartesian matrix M containing as elements all sums $a_i + a_j$, $i, j \in \{1, \dots, 5\}$. If a is sorted, M is sorted as well. So we can apply *sorted_matrix_search* to compute the upper bound for the maximal entry of a in M .

```
#include <CGAL/Random.h>
#include <CGAL/Cartesian_matrix.h>
#include <CGAL/sorted_matrix_search.h>
#include <CGAL/functional.h>
#include <vector>
#include <algorithm>
#include <iterator>

typedef int Value;
typedef std::vector<Value> Vector;
typedef Vector::iterator Value_iterator;
typedef std::vector<Vector> Vector_cont;
typedef CGAL::Cartesian_matrix<std::plus<int>,
                               Value_iterator,
                               Value_iterator> Matrix;

int main()
{
    // set of vectors the matrices are build from:
    Vector_cont vectors;

    // generate a random vector and sort it:
    Vector a;
    const int n = 5;
    for (int i = 0; i < n; ++i)
        a.push_back(CGAL::default_random(100));
    std::sort(a.begin(), a.end());
    std::cout << "a = ( ";
    std::copy(a.begin(), a.end(), std::ostream_iterator<int>(std::cout, " "));
    std::cout << ")\n";

    // build a Cartesian matrix from a:
    Matrix M(a.begin(), a.end(), a.begin(), a.end());

    // search for an upper bound for max(a):
    Value bound = a[n-1];
    Value upper_bound =
    CGAL::sorted_matrix_search(
        &M, &M + 1,
        CGAL::sorted_matrix_search_traits_adaptor(
```

```
    CGAL::bind_2(std::greater_equal<Value>(), bound), M));  
    std::cout << "Upper bound for " << bound << " is "  
        << upper_bound << "." << std::endl;
```

```
    return 0;  
}
```

|_____ *advanced* _____|

CGAL::Sorted_matrix_search_traits_adaptor<F,M>

— advanced —

#include <CGAL/Sorted_matrix_search_traits_adaptor.h>

Definition

The class *Sorted_matrix_search_traits_adaptor*<*F*,*M*> can be used as an adaptor to create sorted matrix search traits classes for arbitrary feasibility test and matrix classes *F* resp. *M*.

Is Model for the Concepts

SortedMatrixSearchTraits page [2333](#)

Requirements

1. *M* is a model for *BasicMatrix* and
2. *F* defines a copy constructor and a monotone *bool operator()*(*const Value*&).

Creation

Sorted_matrix_search_traits_adaptor<*F*,*M*> *t*(*const F* & *m*);

initializes *t* to use *m* for feasibility testing.

Types

Sorted_matrix_search_traits_adaptor<*F*,*M*>::*Matrix*

typedef to *M*.

Sorted_matrix_search_traits_adaptor<*F*,*M*>::*Value*

typedef to *Matrix*::*Value*.

Sorted_matrix_search_traits_adaptor<*F*,*M*>::*Compare_strictly*

typedef to *std::less*<*Value*>.

Sorted_matrix_search_traits_adaptor<*F*,*M*>::*Compare_non_strictly*

typedef to *std::less_equal*<*Value*>.

Operations

Compare_strictly *t.compare_strictly()* *const*

returns the *Compare_strictly* object to be used for the search.

Compare_non_strictly

t.compare_non_strictly() *const*

returns the *Compare_non_strictly* object to be used for the search.

bool *t.is_feasible(const Value& a)*

uses the feasibility test given during creation.

└────────── *advanced* ─────────┘

SortedMatrixSearchTraits

advanced

Definition

The concept `SortedMatrixSearchTraits` defines types and operations needed to compute the smallest entry in a set of sorted matrices that fulfills a certain feasibility criterion using the function `sorted_matrix_search`.

Types

`SortedMatrixSearchTraits::Matrix` The class used for representing matrices. It has to be a model for `BasicMatrix`.

`typedef Matrix::Value`

`Value;` The class used for representing the matrix elements.

`SortedMatrixSearchTraits::Compare_strictly` An adaptable binary function class: $Value \times Value \rightarrow bool$ defining a non-reflexive total order on `Value`. This determines the direction of the search.

`SortedMatrixSearchTraits::Compare_non_strictly`

An adaptable binary function class: $Value \times Value \rightarrow bool$ defining the reflexive total order on `Value` corresponding to `Compare_strictly`.

Operations

`Compare_strictly` `t.compare_strictly() const`

returns the `Compare_strictly` object to be used for the search.

`Compare_non_strictly`

`t.compare_non_strictly() const`

returns the `Compare_non_strictly` object to be used for the search.

`bool` `t.is_feasible(const Value& a)`

The predicate to determine whether an element `a` is feasible. It has to be monotone in the sense that `compare(a, b)` and `is_feasible(a)` imply `is_feasible(b)`.

Has Models

CGAL::Sorted_matrix_search_traits_adaptor<*F,M*>.....page [2331](#)

See Also

CGAL::sorted_matrix_search.....page [2328](#)

BasicMatrix.....page [2327](#)

└————— *advanced* —————┘

Chapter 39

Principal Component Analysis

Pierre Alliez and Sylvain Pion

Contents

39.0.1 Definitions	2335
39.1 Examples	2336
39.1.1 Bounding Box of a Point Set	2336
39.1.2 Centroid of a Point Set	2337
39.1.3 Barycenter of a Set of Weighted Points	2338
39.1.4 Best Fitting Line of a 2D Point Set	2339

This CGAL package provides functions to compute global informations on the shape of a set of 2D or 3D objects such as points. It provides the computation of axis-aligned bounding boxes, centroids of point sets, barycenters of weighted point sets, as well as linear least squares fitting for point sets in 2D, and point sets as well as triangle sets in 3D. The sets are specified by iterator ranges of containers.

39.0.1 Definitions

A *bounding box* for a set of objects is a cuboid that completely contains the set. An *axis-aligned bounding box* is a bounding box aligned with the axes of the coordinate system.

A *centroid* is defined as average of position. A *barycenter* of weighted point sets is defined as weighted average of position. When all weights are equal the barycenter coincides with the centroid.

Given a point set, *linear least squares fitting* amounts to find the linear sub-space which minimizes the sum of squared distances from the points to their projection onto this linear sub-space. This problem is equivalent to search for the linear sub-space which maximizes the variance of projected points, the latter being obtained by eigen decomposition of the covariance matrix of the point set. Eigenvectors corresponding to large eigenvalues are the directions in which the data has strong component, or equivalently large variance. If eigenvalues are the same there is no preferable sub-space.

Given a triangle set, *linear least squares fitting* amounts to find the linear sub-space which minimizes the sum of squared distances from all points in the set to their projection onto this linear sub-space. This problem is

equivalent to the one of fitting a linear sub-space to a point set, except that the covariance matrix is now derived from a continuous integral over the triangles instead of a discrete sum over the points.

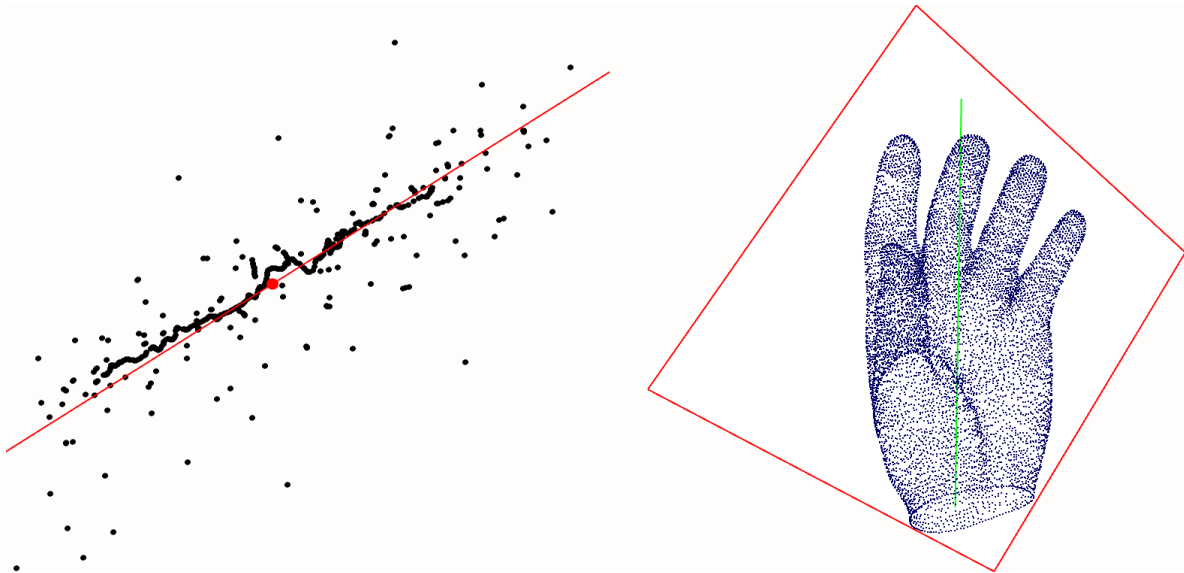


Figure 39.1: Left: fitting a line to a 2D point set. Right: fitting a line and a plane to a 3D point set.

39.1 Examples

39.1.1 Bounding Box of a Point Set

In the following example we use STL containers of 2D and 3D points, and compute their axis-aligned bounding box. The kernel from which the input points come is automatically deduced by the function.

```
// Example program for the bounding_box() function for 2D and 3D points.
```

```
#include <CGAL/Cartesian.h>
#include <CGAL/bounding_box.h>
```

```
#include <list>
#include <iostream>
```

```
typedef double          FT;
typedef CGAL::Cartesian<FT> K;
typedef K::Point_2      Point_2;
typedef K::Point_3      Point_3;
```

```
int main()
{
    // axis-aligned bounding box of 2D points
```



```

std::list<Point_2> points_2;
points_2.push_back(Point_2(1.0, 0.0));
points_2.push_back(Point_2(2.0, 2.0));
points_2.push_back(Point_2(3.0, 5.0));

K::Iso_rectangle_2 c2 = CGAL::bounding_box(points_2.begin(), points_2.end());
std::cout << c2 << std::endl;

// axis-aligned bounding box of 3D points
std::list<Point_3> points_3;
points_3.push_back(Point_3(1.0, 0.0, 0.5));
points_3.push_back(Point_3(2.0, 2.0, 1.2));
points_3.push_back(Point_3(3.0, 5.0, 4.5));

K::Iso_cuboid_3 c3 = CGAL::bounding_box(points_3.begin(), points_3.end());
std::cout << c3 << std::endl;

return 0;
}

```

39.1.2 Centroid of a Point Set

In the following example we use STL containers of 2D and 3D points, and compute their centroid. The kernel from which the input points come is automatically deduced by the function.

```

// Example program for the centroid() function for 2D and 3D points.

#include <CGAL/Cartesian.h>
#include <CGAL/centroid.h>

#include <list>
#include <iostream>

typedef double FT;
typedef CGAL::Cartesian<FT> K;
typedef K::Point_2 Point_2;
typedef K::Point_3 Point_3;

int main()
{
    // centroid of 2D points
    std::list<Point_2> points_2;
    points_2.push_back(Point_2(1.0, 0.0));
    points_2.push_back(Point_2(2.0, 2.0));
    points_2.push_back(Point_2(3.0, 5.0));

    Point_2 c2 = CGAL::centroid(points_2.begin(), points_2.end());
    std::cout << c2 << std::endl;

    // centroid of 3D points
    std::list<Point_3> points_3;
    points_3.push_back(Point_3(1.0, 0.0, 0.5));

```

```

points_3.push_back(Point_3(2.0, 2.0, 1.2));
points_3.push_back(Point_3(3.0, 5.0, 4.5));

Point_3 c3 = CGAL::centroid(points_3.begin(), points_3.end());
std::cout << c3 << std::endl;

return 0;
}

```

39.1.3 Barycenter of a Set of Weighted Points

In the following example we use STL containers of 2D and 3D weighted points, and compute their barycenter. The kernel from which the input points come is automatically deduced by the function.

```

// Example program for the barycenter() function for 2D and 3D points.

#include <CGAL/Cartesian.h>
#include <CGAL/barycenter.h>

#include <list>
#include <iostream>
#include <utility>

typedef double FT;
typedef CGAL::Cartesian<FT> K;
typedef K::Point_2 Point_2;
typedef K::Point_3 Point_3;

int main()
{
    // barycenter of 2D points
    std::list<std::pair<Point_2, FT> > points_2;
    points_2.push_back(std::make_pair(Point_2(1.0, 0.0), 1));
    points_2.push_back(std::make_pair(Point_2(2.0, 2.0), 2));
    points_2.push_back(std::make_pair(Point_2(3.0, 5.0), -2));

    Point_2 c2 = CGAL::barycenter(points_2.begin(), points_2.end());
    std::cout << c2 << std::endl;

    // barycenter of 3D points
    std::list<std::pair<Point_3, FT> > points_3;
    points_3.push_back(std::make_pair(Point_3(1.0, 0.0, 0.5), 1));
    points_3.push_back(std::make_pair(Point_3(2.0, 2.0, 1.2), 2));
    points_3.push_back(std::make_pair(Point_3(3.0, 5.0, 4.5), -5));

    Point_3 c3 = CGAL::barycenter(points_3.begin(), points_3.end());
    std::cout << c3 << std::endl;

    return 0;
}

```

39.1.4 Best Fitting Line of a 2D Point Set

In the following example we use an STL container of 2D points, and compute the best fitting line. The kernel from which the input points come is automatically deduced by the function.

```
// Example program for the linear_least_square_fitting function

#include <CGAL/Cartesian.h>
#include <CGAL/linear_least_squares_fitting_2.h>

typedef double          FT;
typedef CGAL::Cartesian<FT> K;
typedef K::Line_2       Line_2;
typedef K::Point_2      Point_2;

int main()
{
    std::list<Point_2> points;
    points.push_back(Point_2(1.0,0.0));
    points.push_back(Point_2(2.0,0.0));
    points.push_back(Point_2(3.0,0.0));

    Line_2 line;
    linear_least_squares_fitting_2(points.begin(),points.end(),line);

    return 0;
}
```


Principal Component Analysis Reference Manual

Pierre Alliez and Sylvain Pion

This CGAL package provides functions to compute global informations on the shape of a set of 2D or 3D objects such as points. It provides the computation of axis-aligned bounding boxes for point sets, centroids of point sets and triangle sets in 2D and 3D, barycenters of weighted point sets, as well as linear least squares fitting for point sets in 2D, and point sets and triangle sets in 3D. It assumes the set of kernel primitive elements to be stored into an iterator range of a container.

39.2 Classified Reference Pages

Functions

<i>CGAL::barycenter</i>	page 2342
<i>CGAL::bounding_box</i>	page 2344
<i>CGAL::centroid</i>	page 2345
<i>CGAL::linear_least_squares_fitting_2</i>	page 2346
<i>CGAL::linear_least_squares_fitting_3</i>	page 2347

39.3 Alphabetical List of Reference Pages

<i>barycenter</i>	page 2342
<i>bounding_box</i>	page 2344
<i>centroid</i>	page 2345
<i>linear_least_squares_fitting_2</i>	page 2346
<i>linear_least_squares_fitting_3</i>	page 2347

CGAL::barycenter

Definition

The function *barycenter* computes the barycenter (weighted center of mass) of a set of weighted 2D or 3D objects. The weight associated to each object is specified using a *std::pair* storing the object and its weight.

```
#include <CGAL/barycenter.h>
```

There is a set of overloaded *barycenter* functions for 2D and 3D weighted objects. The user can also optionally pass an explicit kernel, in case the default, based on *Kernel_traits* is not sufficient. The dimension is also deduced automatically.

```
template < typename InputIterator >
```

```
K::Point_2          barycenter( InputIterator first, InputIterator beyond)
```

computes the barycenter of a non-empty set of 2D weighted points. *K* is *Kernel_traits<std::iterator_traits<InputIterator>::value_type::first_type>::Kernel*. The value type must be *std::pair<K::Point_2, K::FT>*.

Precondition: first != beyond, and the sum of the weights is non-zero.

```
template < typename InputIterator, typename K >
```

```
K::Point_2          barycenter( InputIterator first, InputIterator beyond, K k)
```

computes the barycenter of a non-empty set of 2D weighted points. The value type must be *std::pair<K::Point_2, K::FT>*.

Precondition: first != beyond, and the sum of the weights is non-zero.

```
template < typename InputIterator >
```

```
K::Point_3          barycenter( InputIterator first, InputIterator beyond)
```

computes the barycenter of a non-empty set of 3D weighted points. *K* is *Kernel_traits<std::iterator_traits<InputIterator>::value_type::first_type>::Kernel*. The value type must be *std::pair<K::Point_3, K::FT>*.

Precondition: first != beyond, and the sum of the weights is non-zero.

```
template < typename InputIterator, typename K >
```

```
K::Point_3          barycenter( InputIterator first, InputIterator beyond, K k)
```

computes the barycenter of a non-empty set of 3D weighted points. The value type must be *std::pair<K::Point_3, K::FT>*.

Precondition: first != beyond, and the sum of the weights is non-zero.

See Also

CGAL::centroid page [2345](#)

CGAL::bounding_box

Definition

The function *bounding_box* computes the bounding box of a set of 2D or 3D objects.

```
#include <CGAL/bounding_box.h>
```

There is a set of overloaded *bounding_box* functions for 2D and 3D objects. The user can also optionally pass an explicit kernel, in case the default, based on *Kernel_traits* is not sufficient. The dimension is also deduced automatically.

```
template < typename InputIterator >
K::Iso_rectangle_2    bounding_box( InputIterator first, InputIterator beyond)
```

computes the bounding box of a non-empty set of 2D points. *K* is *Kernel_traits<std::iterator_traits<InputIterator>::value_type>::Kernel*. The value type must be *K::Point_2*.
Precondition: first != beyond.

```
template < typename InputIterator, typename K >
K::Iso_rectangle_2    bounding_box( InputIterator first, InputIterator beyond, K k)
```

computes the bounding box of a non-empty set of 2D points. The value type must be *K::Point_2*.
Precondition: first != beyond.

```
template < typename InputIterator >
K::Iso_cuboid_3       bounding_box( InputIterator first, InputIterator beyond)
```

computes the bounding box of a non-empty set of 3D points. *K* is *Kernel_traits<std::iterator_traits<InputIterator>::value_type>::Kernel*. The value type must be *K::Point_3*.
Precondition: first != beyond.

```
template < typename InputIterator, typename K >
K::Iso_cuboid_3       bounding_box( InputIterator first, InputIterator beyond, K k)
```

computes the bounding box of a non-empty set of 3D points. The value type must be *K::Point_3*.
Precondition: first != beyond.

CGAL::centroid

Definition

The function *centroid* computes the centroid (center of mass) of a set of 2D or 3D objects.

```
#include <CGAL/centroid.h>
```

There is a set of overloaded *centroid* functions for 2D and 3D objects. The user can also optionally pass an explicit kernel, in case the default, based on *Kernel_traits* is not sufficient. The dimension is also deduced automatically.

```
template < typename InputIterator >
K::Point_2          centroid( InputIterator first, InputIterator beyond)
```

computes the centroid of a non-empty set of 2D points. *K* is *Kernel_traits<std::iterator_traits<InputIterator>::value_type>::Kernel*. The value type must be *K::Point_2*.
Precondition: first != beyond.

```
template < typename InputIterator, typename K >
K::Point_2          centroid( InputIterator first, InputIterator beyond, K k)
```

computes the centroid of a non-empty set of 2D points. The value type must be *K::Point_2*.
Precondition: first != beyond.

```
template < typename InputIterator >
K::Point_3          centroid( InputIterator first, InputIterator beyond)
```

computes the centroid of a non-empty set of 3D points. *K* is *Kernel_traits<std::iterator_traits<InputIterator>::value_type>::Kernel*. The value type must be *K::Point_3*.
Precondition: first != beyond.

```
template < typename InputIterator, typename K >
K::Point_3          centroid( InputIterator first, InputIterator beyond, K k)
```

computes the centroid of a non-empty set of 3D points. The value type must be *K::Point_3*.
Precondition: first != beyond.

See Also

CGAL::barycenter [page 2342](#)

CGAL::linear_least_squares_fitting_2

Definition

The function *linear_least_squares_fitting_2* computes the best fitting 2D line of a 2D point set. The best fit line minimizes the sum of squared distances from the points to their projections onto the line.

```
#include <CGAL/linear_least_squares_fitting_2.h>
```

```
template < typename InputIterator, typename K>
typename K::FT      linear_least_squares_fitting_2( InputIterator first,
                                                    InputIterator beyond,
                                                    typename K::Line_2 & line,
                                                    typename K::Point_2 & centroid,
                                                    K k)
```

computes the best fitting 2D line of a 2D point set in the range *[first,beyond)*. The value returned is a fitting quality between 0 and 1, where 0 means that the variance is the same along any line (a horizontal line going through the centroid is output by default), and 1 means that the variance is null orthogonally to the best fitting line.

The class *K* is the kernel in which the type *InputIterator::value_type* is defined. It can be omitted and deduced automatically from the value type.

Requirements

1. *InputIterator::value_type* is equivalent to *K::Point_2*.
2. *line* is the best fitting line computed.
3. *centroid* is the centroid computed. This parameter can be omitted.

CGAL::linear_least_squares_fitting_3

Definition

The function *linear_least_squares_fitting_3* computes the best fitting 3D line or plane of 3D point sets or triangle sets. The best fit linear sub-space minimizes the sum of squared distances from the points to their projections onto the sub-space.

```
#include <CGAL/linear_least_squares_fitting_3.h>
```

```
template < typename InputIterator, typename K>
typename K::FT      linear_least_squares_fitting_3( InputIterator first,
                                                    InputIterator beyond,
                                                    typename K::Line_3 & line,
                                                    typename K::Point_3 & centroid,
                                                    K k)
```

computes the best fitting 3D line of a 3D point set or triangle set in the range *[first,beyond)*. The value returned is a fitting quality between 0 and 1, where 0 means that the variance is the same along any line (a horizontal line going through the centroid is output by default), and 1 means that the variance is null orthogonally to the best fitting line.

The class *K* is the kernel in which the type *InputIterator::value_type* is defined. It can be omitted and deduced automatically from the value type.

Requirements

1. *InputIterator::value_type* is equivalent to *K::Point_3* or *K::Triangle_3*.
2. *line* is the best fitting line computed.
3. *centroid* is the centroid computed. This parameter can be omitted.

```
template < typename InputIterator, typename K>
typename K::FT      linear_least_squares_fitting_3( InputIterator first,
                                                    InputIterator beyond,
                                                    typename K::Plane_3 & plane,
                                                    typename K::Point_3 & centroid,
                                                    K k)
```

computes the best fitting 3D plane of a 3D point set or triangle set in the range *[first,beyond)*. The value returned is a fitting quality between 0 and 1, where 0 means that the variance is the same along any plane (a horizontal plane going through the centroid is output by default), and 1 means that the variance is null orthogonally to the best fitting plane.

The class K is the kernel in which the type $InputIterator::value_type$ is defined. It can be omitted and deduced automatically from the value type.

Requirements

1. $InputIterator::value_type$ is equivalent to $K::Point_3$ or $K::Triangle_3$.
2. $plane$ is the best fitting plane computed.
3. $centroid$ is the centroid computed. This parameter can be omitted.

Part XII

Interpolation

Chapter 40

Interpolation

Julia Flötotto

Contents

40.1 Natural Neighbor Coordinates	2352
40.1.1 Introduction	2352
40.1.2 Implementation	2353
40.1.3 Example	2353
40.2 Surface Natural Neighbor Coordinates and Surface Neighbors	2355
40.2.1 Introduction	2355
40.2.2 Implementation	2355
40.2.3 Examples	2356
40.3 Interpolation Methods	2357
40.3.1 Introduction	2357
40.3.2 Gradient Fitting	2358
40.3.3 Examples	2359

This chapter describes CGAL’s interpolation package which implements natural neighbor coordinate functions as well as different methods for scattered data interpolation most of which are based on natural neighbor coordinates. The functions for computing natural neighbor coordinates in Euclidean space are described in Section 40.1, the functions concerning the coordinate and neighbor computation on surfaces are discussed in Section 40.2. In Section 40.3, we describe the different interpolation functions.

Scattered data interpolation solves the following problem: given measures of a function on a set of discrete data points, the task is to interpolate this function on an arbitrary query point. More formally, let $\mathcal{P} = \{\mathbf{p}_1, \dots, \mathbf{p}_n\}$ be a set of n points in \mathbb{R}^2 or \mathbb{R}^3 and Φ be a scalar function defined inside the convex hull of \mathcal{P} . We assume that the function values are known at the points of \mathcal{P} , i.e. to each $\mathbf{p}_i \in \mathcal{P}$, we associate $z_i = \Phi(\mathbf{p}_i)$. Sometimes, the gradient of Φ is also known at \mathbf{p}_i . It is denoted $\mathbf{g}_i = \nabla\Phi(\mathbf{p}_i)$. The interpolation is carried out for an arbitrary query point \mathbf{x} . Except for interpolation on surfaces, \mathbf{x} must lie inside the convex hull of \mathcal{P} .

40.1 Natural Neighbor Coordinates

40.1.1 Introduction

Natural neighbor interpolation has been introduced by Sibson [Sib81] to interpolate multivariate scattered data. Given a set of data points \mathcal{P} , the natural neighbor coordinates associated to \mathcal{P} are defined from the Voronoi diagram of \mathcal{P} . When simulating the insertion of a query point \mathbf{x} into the Voronoi diagram of \mathcal{P} , the potential Voronoi cell of \mathbf{x} “steals” some parts from the existing cells.

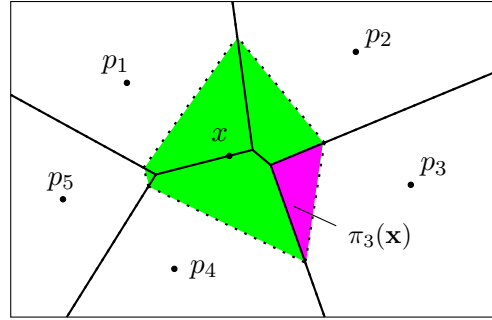


Figure 40.1: 2D example: \mathbf{x} has five natural neighbors $\mathbf{p}_1, \dots, \mathbf{p}_5$. The natural neighbor coordinate $\lambda_3(\mathbf{x})$ is the ratio of the area of the pink polygon, $\pi_3(\mathbf{x})$, over the area of the total highlighted zone.

Let $\pi(\mathbf{x})$ denote the volume of the potential Voronoi cell of \mathbf{x} and $\pi_i(\mathbf{x})$ denote the volume of the sub-cell that would be stolen from the cell of \mathbf{p}_i by the cell of \mathbf{x} . The natural neighbor coordinate of \mathbf{x} with respect to the data point $\mathbf{p}_i \in \mathcal{P}$ is defined by

$$\lambda_i(\mathbf{x}) = \frac{\pi_i(\mathbf{x})}{\pi(\mathbf{x})}.$$

A two-dimensional example is depicted in Figure 40.1.

Various papers ([Sib80], [Far90], [Pip93], [Bro97], [HS00]) show that the natural neighbor coordinates have the following properties:

- (i) $\mathbf{x} = \sum_{i=1}^n \lambda_i(\mathbf{x}) \mathbf{p}_i$ (barycentric coordinate property).
- (ii) For any $i, j \leq n$, $\lambda_i(\mathbf{p}_j) = \delta_{ij}$, where δ_{ij} is the Kronecker symbol.
- (iii) $\sum_{i=1}^n \lambda_i(\mathbf{x}) = 1$ (partition of unity property).

Furthermore, Piper [Pip93] shows that the coordinate functions are continuous in the convex hull of \mathcal{P} and continuously differentiable except on the data points \mathcal{P} .

The interpolation package of CGAL provides functions to compute natural neighbor coordinates for 2D and 3D points with respect to Voronoi diagrams as well as with respect to power diagrams (only 2D), i.e. for weighted points. Refer to the reference pages *natural_neighbor_coordinates_2*, *natural_neighbor_coordinates_3* and *regular_neighbor_coordinates_2*.

In addition, the package provides functions to compute natural neighbor coordinates on well sampled point set surfaces. See Section 40.2 and the reference page *surface_neighbor_coordinates_3* for further information.

40.1.2 Implementation

Given a Delaunay triangulation or a Regular triangulation, the vertices in conflict with the query point are determined. The areas $\pi_i(\mathbf{x})$ are computed by triangulating the Voronoi sub-cells. The normalization factor $\pi(\mathbf{x})$ is also returned. If the query point is already located and/or the boundary edges of the conflict zone are already determined, alternative functions allow to avoid the re-computation.

40.1.3 Example

The signature of all coordinate computation functions is about the same.

```
//file: examples/Interpolation/nn_coordinates_2.C

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/natural_neighbor_coordinates_2.h>

struct K : CGAL::Exact_predicates_inexact_constructions_kernel {};
typedef CGAL::Delaunay_triangulation_2<K> Delaunay_triangulation;
typedef std::vector< std::pair< K::Point_2, K::FT > >
                                     Point_coordinate_vector;

int main()
{
    Delaunay_triangulation dt;

    for (int y=0 ; y<3 ; y++)
        for (int x=0 ; x<3 ; x++)
            dt.insert(K::Point_2(x,y));

    //coordinate computation
    K::Point_2 p(1.2, 0.7);
    Point_coordinate_vector coords;
    CGAL::Triple<
        std::back_inserter_iterator<Point_coordinate_vector>,
        K::FT, bool> result =
        CGAL::natural_neighbor_coordinates_2(dt, p,
std::back_inserter(coords));
    if(!result.third){
        std::cout << "The coordinate computation was not successful."
        << std::endl;
        std::cout << "The point (" <<p << ") lies outside the convex hull."
        << std::endl;
    }
    K::FT norm = result.second;
    std::cout << "Coordinate computation successful." << std::endl;
    std::cout << "Normalization factor: " <<norm << std::endl;

    return 0;
}
```

Regular neighbor coordinate computation

For regular neighbor coordinates, it is sufficient to replace the name of the function and the type of triangulation passed as parameter. A special traits class is needed.

```
//
//file: examples/Interpolation/rn_coordinates_2.C
//
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Regular_triangulation_2.h>
#include <CGAL/Regular_triangulation_euclidean_traits_2.h>
#include <CGAL/regular_neighbor_coordinates_2.h>

struct K : CGAL::Exact_predicates_inexact_constructions_kernel {};

typedef CGAL::Regular_triangulation_euclidean_traits_2<K> Gt;
typedef CGAL::Regular_triangulation_2<Gt> Regular_triangulation;
typedef Regular_triangulation::Weighted_point Weighted_point;
typedef std::vector< std::pair< Weighted_point, K::FT > >
Point_coordinate_vector;

int main()
{
    Regular_triangulation rt;

    for (int y=0 ; y<3 ; y++)
        for (int x=0 ; x<3 ; x++)
            rt.insert(Weighted_point(K::Point_2(x,y), 0));

    //coordinate computation
    Weighted_point wp(K::Point_2(1.2, 0.7),2);
    Point_coordinate_vector coords;
    CGAL::Triple<
        std::back_insert_iterator<Point_coordinate_vector>,
        K::FT, bool> result =
        CGAL::regular_neighbor_coordinates_2(rt, wp,
std::back_inserter(coords));
    if(!result.third){
        std::cout << "The coordinate computation was not successful."
        << std::endl;
        std::cout << "The point (" <<wp.point() << ") lies outside the convex hull."
        << std::endl;
    }
    K::FT norm = result.second;
    std::cout << "Coordinate computation successful." << std::endl;
    std::cout << "Normalization factor: " <<norm << std::endl;

    return 0;
}
```

For surface neighbor coordinates, the surface normal at the query point must be provided, see Section [40.2](#).

40.2 Surface Natural Neighbor Coordinates and Surface Neighbors

This section introduces the functions to compute natural neighbor coordinates and surface neighbors associated to a set of sample points issued from a surface \mathcal{S} and given a query point \mathbf{x} on \mathcal{S} . We suppose that \mathcal{S} is a closed and compact surface of \mathbb{R}^3 , and let $\mathcal{P} = \{\mathbf{p}_1, \dots, \mathbf{p}_n\}$ be an ε -sample of \mathcal{S} (refer to Amenta and Bern [AB99]). The concepts are based on the definition of Boissonnat and Flötotto [BF02], [Flö03b]. Both references contain a thorough description of the requirements and the mathematical properties.

40.2.1 Introduction

Two observations lead to the definition of surface neighbors and surface neighbor coordinates: First, it is clear that the tangent plane $\mathcal{T}_{\mathbf{x}}$ of the surface \mathcal{S} at the point $\mathbf{x} \in \mathcal{S}$ approximates \mathcal{S} in the neighborhood of \mathbf{x} . It has been shown in [BF02] that, if the surface \mathcal{S} is well sampled with respect to the curvature and the local thickness of \mathcal{S} , i.e. it is an ε -sample, the intersection of the tangent plane $\mathcal{T}_{\mathbf{x}}$ with the Voronoi cell of \mathbf{x} in the Voronoi diagram of $\mathcal{P} \cup \{\mathbf{x}\}$ has a small diameter. Consequently, inside this Voronoi cell, the tangent plane $\mathcal{T}_{\mathbf{x}}$ is a reasonable approximation of \mathcal{S} . Furthermore, the second observation allows to compute this intersection diagram easily: one can show using Pythagoras' Theorem that the intersection of a three-dimensional Voronoi diagram with a plane \mathcal{H} is a two-dimensional power diagram. The points defining the power diagram are the projections of the points in \mathcal{P} onto \mathcal{H} , each point weighted with its negative square distance to \mathcal{H} . Algorithms for the computation of power diagrams via the dual regular triangulation are well known and for example provided by CGAL in the class *Regular_triangulation_2<Gt, Tds>*.

40.2.2 Implementation

Voronoi Intersection Diagrams

In CGAL, the regular triangulation dual to the intersection of a 3D Voronoi diagram with a plane \mathcal{H} can be computed by instantiating the *Regular_triangulation_2<Gt, Tds>* class with the traits class *Voronoi_intersection_2_traits_3<K>*. This traits class contains a point and a vector as class member which define the plane \mathcal{H} . All predicates and constructions used by *Regular_triangulation_2<Gt, Tds>* are replaced by the corresponding operators on three-dimensional points. For example, the power test predicate (which takes three weighted 2D points p' , q' , r' of the regular triangulation and tests the power distance of a fourth point t' with respect to the power circle orthogonal to p, q, r) is replaced by a *Side_of_plane_centered_sphere_2_3* predicate that tests the position of a 3D point t with respect to the sphere centered on the plane \mathcal{H} passing through the 3D points p, q, r . This approach allows to avoid the explicit constructions of the projected points and the weights which are very prone to rounding errors.

Natural Neighbor Coordinates on Surfaces

The computation of natural neighbor coordinates on surfaces is based upon the computation of regular neighbor coordinates with respect to the regular triangulation that is dual to $\text{Vor}(\mathcal{P}) \cap \mathcal{T}_{\mathbf{x}}$, the intersection of $\mathcal{T}_{\mathbf{x}}$ and the Voronoi diagram of \mathcal{P} , via the function *regular_neighbor_coordinates_2*.

Of course, we might introduce all data points \mathcal{P} into this regular triangulation. However, this is not necessary because we are only interested in the cell of \mathbf{x} . It is sufficient to guarantee that all surface neighbors of the query point \mathbf{x} are among the input points that are passed as argument to the function. The sample points \mathcal{P} can be filtered for example by distance, e.g. using range search or k -nearest neighbor queries, or with the help of the 3D Delaunay triangulation since the surface neighbors are necessarily a subset of the natural neighbors of

the query point in this triangulation. CGAL provides a function that encapsulates the filtering based on the 3D Delaunay triangulation. For input points filtered by distance, functions are provided that indicate whether or not points that lie outside the input range (i.e. points that are further from \mathbf{x} than the furthest input point) can still influence the result. This allows to iteratively enlarge the set of input points until the range is sufficient to certify the result.

Surface Neighbors

The surface neighbors of the query point are its neighbors in the regular triangulation that is dual to $\text{Vor}(\mathcal{P}) \cap \mathcal{T}_x$, the intersection of \mathcal{T}_x and the Voronoi diagram of \mathcal{P} . As for surface neighbor coordinates, this regular triangulation is computed and the same kind of filtering of the data points as well as the certification described above is provided.

40.2.3 Examples

```
//file: examples/Interpolation/surface_neighbor_coordinates_3.C
// example with random points on a sphere

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>

#include <CGAL/point_generators_3.h>
#include <CGAL/copy_n.h>
#include <CGAL/Origin.h>

#include <CGAL/surface_neighbor_coordinates_3.h>

struct K : CGAL::Exact_predicates_inexact_constructions_kernel {};
typedef K::FT Coord_type;
typedef K::Point_3 Point_3;
typedef K::Vector_3 Vector_3;
typedef std::vector< std::pair< Point_3, K::FT > >
    Point_coordinate_vector;

int main()
{
    int n=100;
    std::vector< Point_3> points;
    points.reserve(n);

    std::cout << "Generate " << n << " random points on a sphere."
        << std::endl;
    CGAL::Random_points_on_sphere_3<Point_3> g(1);
    CGAL::copy_n( g, n, std::back_inserter(points));

    Point_3 p(1, 0, 0);
    Vector_3 normal(p-CGAL::ORIGIN);
    std::cout << "Compute surface neighbor coordinates for "
        << p << std::endl;
    Point_coordinate_vector coords;
    CGAL::Triple< std::back_insert_iterator<Point_coordinate_vector>,
```

```

    K::FT, bool> result =
    CGAL::surface_neighbor_coordinates_3(points.begin(), points.end(),
    p, normal,
    std::back_inserter(coords),
    K());
    if(!result.third){
        //Undersampling:
        std::cout << "The coordinate computation was not successful."
        << std::endl;
        return 0;
    }
    K::FT norm = result.second;

    std::cout << "Testing the barycentric property " << std::endl;
    Point_3 b(0, 0,0);
    for(std::vector< std::pair< Point_3, Coord_type > >::const_iterator
    it = coords.begin(); it!=coords.end(); ++it)
        b = b + (it->second/norm)* (it->first - CGAL::ORIGIN);

    std::cout <<"    weighted barycenter: " << b <<std::endl;
    std::cout << "    squared distance: " <<
    CGAL::squared_distance(p,b) <<std::endl;
    return 0;
}

```

40.3 Interpolation Methods

40.3.1 Introduction

Linear Precision Interpolation

Sibson [Sib81] defines a very simple interpolant that re-produces linear functions exactly. The interpolation of $\Phi(\mathbf{x})$ is given as the linear combination of the neighbors' function values weighted by the coordinates:

$$Z^0(\mathbf{x}) = \sum_i \lambda_i(\mathbf{x}) z_i.$$

Indeed, if $z_i = a + \mathbf{b}'\mathbf{p}_i$ for all natural neighbors of \mathbf{x} , we have

$$Z^0(\mathbf{x}) = \sum_i \lambda_i(\mathbf{x})(a + \mathbf{b}'\mathbf{p}_i) = a + \mathbf{b}'\mathbf{x}$$

by the barycentric coordinate property. The first example in Subsection 40.3.3 shows how the function is called.

Sibson's C^1 Continuous Interpolant

In [Sib81], Sibson describes a second interpolation method that relies also on the function gradient \mathbf{g}_i for all $\mathbf{p}_i \in \mathcal{P}$. It is C^1 continuous with gradient \mathbf{g}_i at \mathbf{p}_i . Spherical quadrics of the form $\Phi(\mathbf{x}) = a + \mathbf{b}'\mathbf{x} + \gamma \mathbf{x}'\mathbf{x}$ are reproduced exactly. The proof relies on the barycentric coordinate property of the natural neighbor coordinates and assumes that the gradient of Φ at the data points is known or approximated from the function values as described in [Sib81] (see Section 40.3.2).

Sibson's Z^1 interpolant is a combination of the linear interpolant Z^0 and an interpolant ξ which is the weighted sum of the first degree functions

$$\xi_i(\mathbf{x}) = z_i + \mathbf{g}_i^t(\mathbf{x} - \mathbf{p}_i), \quad \xi(\mathbf{x}) = \frac{\sum_i \frac{\lambda_i(\mathbf{x})}{\|\mathbf{x} - \mathbf{p}_i\|} \xi_i(\mathbf{x})}{\sum_i \frac{\lambda_i(\mathbf{x})}{\|\mathbf{x} - \mathbf{p}_i\|}}.$$

Sibson observed that the combination of Z^0 and ξ reconstructs exactly a spherical quadric if they are mixed as follows:

$$Z^1(\mathbf{x}) = \frac{\alpha(\mathbf{x})Z^0(\mathbf{x}) + \beta(\mathbf{x})\xi(\mathbf{x})}{\alpha(\mathbf{x}) + \beta(\mathbf{x})} \text{ where } \alpha(\mathbf{x}) = \frac{\sum_i \lambda_i(\mathbf{x}) \frac{\|\mathbf{x} - \mathbf{p}_i\|^2}{f(\|\mathbf{x} - \mathbf{p}_i\|)}}{\sum_i \frac{\lambda_i(\mathbf{x})}{f(\|\mathbf{x} - \mathbf{p}_i\|)}} \text{ and } \beta(\mathbf{x}) = \sum_i \lambda_i(\mathbf{x}) \|\mathbf{x} - \mathbf{p}_i\|^2,$$

where in Sibson's original work, $f(\|\mathbf{x} - \mathbf{p}_i\|) = \|\mathbf{x} - \mathbf{p}_i\|$.

CGAL contains a second implementation with $f(\|\mathbf{x} - \mathbf{p}_i\|) = \|\mathbf{x} - \mathbf{p}_i\|^2$ which is less demanding on the number type because it avoids the square-root computation needed to compute the distance $\|\mathbf{x} - \mathbf{p}_i\|$. The theoretical guarantees are the same (see [Flö03b]). Simply, the smaller the slope of f around $f(0)$, the faster the interpolant approaches ξ_i as $\mathbf{x} \rightarrow \mathbf{p}_i$.

Farin's C^1 Continuous Interpolant

Farin [Far90] extended Sibson's work and realizes a C^1 continuous interpolant by embedding natural neighbor coordinates in the Bernstein-Bézier representation of a cubic simplex. If the gradient of Φ at the data points is known, this interpolant reproduces quadratic functions exactly. The function gradient can be approximated from the function values by Sibson's method [Sib81] (see Section 40.3.2) which is exact only for spherical quadrics.

Quadratic Precision Interpolants

Knowing the gradient \mathbf{g}_i for all $\mathbf{p}_i \in \mathcal{P}$, we formulate a very simple interpolant that reproduces exactly quadratic functions. This interpolant is not C^1 continuous in general. It is defined as follows:

$$I^1(\mathbf{x}) = \sum_i \lambda_i(\mathbf{x}) \left(z_i + \frac{1}{2} \mathbf{g}_i^t(\mathbf{x} - \mathbf{p}_i) \right)$$

40.3.2 Gradient Fitting

Sibson describes a method to approximate the gradient of the function f from the function values on the data sites. For the data point \mathbf{p}_i , we determine

$$\mathbf{g}_i = \min_{\mathbf{g}} \sum_j \frac{\lambda_j(\mathbf{p}_i)}{\|\mathbf{p}_i - \mathbf{p}_j\|^2} (z_j - (z_i + \mathbf{g}^t(\mathbf{p}_j - \mathbf{p}_i))),$$

where $\lambda_j(\mathbf{p}_i)$ is the natural neighbor coordinate of \mathbf{p}_i with respect to \mathbf{p}_i associated to $\mathcal{P} \setminus \{\mathbf{p}_i\}$. This method works only for points inside the convex hull of the data points because, for a point \mathbf{p}_i on the convex hull, $\lambda_j(\mathbf{p}_i)$ is not defined. For spherical quadrics, the result is exact.

CGAL provides functions to approximate the gradients of all data points that are inside the convex hull. There is one function for each type of natural neighbor coordinate (i.e. *natural_neighbor_coordinates_2*, *regular_neighbor_coordinates_2*).

40.3.3 Examples

Linear Interpolation Method

```
//file: examples/Interpolation/linear_interpolation_2.C
//
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>

#include <CGAL/Interpolation_traits_2.h>
#include <CGAL/natural_neighbor_coordinates_2.h>
#include <CGAL/interpolation_functions.h>

struct K : CGAL::Exact_predicates_inexact_constructions_kernel {};
typedef CGAL::Delaunay_triangulation_2<K>          Delaunay_triangulation;
typedef CGAL::Interpolation_traits_2<K>            Traits;
typedef K::FT                                       Coord_type;
typedef K::Point_2                                  Point;

int main()
{
    Delaunay_triangulation T;
    std::map<Point, Coord_type, K::Less_xy_2> function_values;
    typedef CGAL::Data_access< std::map<Point, Coord_type, K::Less_xy_2 > >
                                   Value_access;

    Coord_type a(0.25), bx(1.3), by(-0.7);

    for (int y=0 ; y<3 ; y++)
        for (int x=0 ; x<3 ; x++){
            K::Point_2 p(x,y);
            T.insert(p);
            function_values.insert(std::make_pair(p,a + bx* x+ by*y));
        }
    //coordinate computation
    K::Point_2 p(1.3,0.34);
    std::vector< std::pair< Point, Coord_type > > coords;
    Coord_type norm =
        CGAL::natural_neighbor_coordinates_2
        (T, p, std::back_inserter(coords)).second;

    Coord_type res = CGAL::linear_interpolation(coords.begin(), coords.end(),
                                                norm,
                                                Value_access(function_values));

    std::cout << "    Tested interpolation on " << p << " interpolation: "
        << res << " exact: " << a + bx* p.x()+ by* p.y()<< std::endl;
    return 0;
}
```

Sibson's C^1 interpolation scheme with gradient estimation

```
//file: examples/Interpolation/sibson_interpolation_2.C

#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>

#include <CGAL/natural_neighbor_coordinates_2.h>
#include <CGAL/Interpolation_gradient_fitting_traits_2.h>
#include <CGAL/sibson_gradient_fitting.h>
#include <CGAL/interpolation_functions.h>

struct K : CGAL::Exact_predicates_inexact_constructions_kernel {};
typedef CGAL::Delaunay_triangulation_2<K> Delaunay_triangulation;
typedef CGAL::Interpolation_gradient_fitting_traits_2<K> Traits;

typedef K::FT Coord_type;
typedef K::Point_2 Point;
typedef std::map<Point, Coord_type, K::Less_xy_2> Point_value_map ;
typedef std::map<Point, K::Vector_2 , K::Less_xy_2 > Point_vector_map;

int main()
{
    Delaunay_triangulation T;

    Point_value_map function_values;
    Point_vector_map function_gradients;

    //parameters for spherical function:
    Coord_type a(0.25), bx(1.3), by(-0.7), c(0.2);
    for (int y=0 ; y<4 ; y++)
        for (int x=0 ; x<4 ; x++){
            K::Point_2 p(x,y);
            T.insert(p);
            function_values.insert(std::make_pair(p,a + bx* x+ by*y + c*(x*x+y*y)));
        }
    sibson_gradient_fitting_nn_2(T,std::inserter(function_gradients,
        function_gradients.begin()),
        CGAL::Data_access<Point_value_map>
        (function_values),
        Traits());

    //cooridante computation
    K::Point_2 p(1.6,1.4);
    std::vector< std::pair< Point, Coord_type > > coords;
    Coord_type norm =
        CGAL::natural_neighbor_coordinates_2(T, p,std::back_inserter
        (coords)).second;

    //Sibson interpolant: version without sqrt:
    std::pair<Coord_type, bool> res =
        CGAL::sibson_c1_interpolation_square
```



```

(coords.begin(),
 coords.end(), norm, p,
 CGAL::Data_access<Point_value_map>(function_values),
 CGAL::Data_access<Point_vector_map>(function_gradients),
 Traits());
if(res.second)
    std::cout << "    Tested interpolation on " << p
        << " interpolation: " << res.first << " exact: "
        << a + bx * p.x() + by * p.y() + c*(p.x()*p.x()+p.y()*p.y())
        << std::endl;
else
    std::cout << "C^1 Interpolation not successful." << std::endl
        << " not all function_gradients are provided." << std::endl
        << " You may resort to linear interpolation." << std::endl;

return 0;
};

```

An additional example compares numerically the errors of the different interpolation functions with respect to a known function. It is distributed in the examples directory.

Interpolation

Reference Manual

Julia Flötotto

Scattered data interpolation solves the following problem: given measures of a function on a set of discrete data points, the task is to interpolate this function on an arbitrary query point.

If the function is a linear function and given barycentric coordinates that allow to express the query point as the convex combination of some data points, the function can be exactly interpolated. If the function gradients are known, we can exactly interpolate quadratic functions given barycentric coordinates. Any further properties of these interpolation functions depend on the properties of the barycentric coordinates. They are provided in this package under the name *linear_interpolation* and *quadratic_interpolation*.

Natural neighbor interpolation

Natural neighbor coordinates are defined by Sibson in 1980 and are based on the Voronoi diagram of the data points. Interpolation methods based on natural neighbor coordinates are particularly interesting because they adapt easily to non-uniform and highly anisotropic data. This package contains Sibson's C^1 continuous interpolation method which interpolates exactly spherical quadrics (of the form $\Phi(\mathbf{x}) = a + \mathbf{b}'\mathbf{x} + \gamma \mathbf{x}'\mathbf{x}$) and Farin's C^1 continuous interpolation method based on Bernstein-Bézier techniques and interpolating exactly quadratic functions – assuming that the function gradient is known. In addition, Sibson defines a method to approximate the function gradients for data points that are in the interior of the convex hull. This method is exact for spherical quadrics.

This CGAL package implements Sibson's and Farin's interpolation functions as well as Sibson's function gradient fitting method. Furthermore, it provides functions to compute the natural neighbor coordinates with respect to a two-dimensional Voronoi diagram (i. e., from the Delaunay triangulation of the data points) and to a two-dimensional power diagram for weighted points (i. e., from their regular triangulation). Natural neighbor coordinates on closed and well-sampled surfaces can also be computed if the normal to the surface at the query point is known. The latter coordinates are only approximately barycentric, see [\[BF02\]](#).

For a more thorough introduction see the user manual.

40.4 Classified Reference Pages

Concepts

InterpolationTraits	page 2371
GradientFittingTraits	page 2380

Interpolation Functions

CGAL::linear_interpolation	page 2365
CGAL::sibson_c1_interpolation	page 2366
CGAL::farin_c1_interpolation	page 2368
CGAL::quadratic_interpolation	page 2369
CGAL::sibson_gradient_fitting	page 2378

CGAL::Interpolation_traits_2<K>	page 2373
CGAL::Interpolation_gradient_fitting_traits_2<K>	page 2382

Natural neighbor coordinate computation

CGAL::natural_neighbor_coordinates_2	page 2374
CGAL::regular_neighbor_coordinates_2	page 2376

Surface neighbor and surface neighbor coordinate computation

CGAL::Voronoi_intersection_2_traits_3<K>	page 2384
CGAL::surface_neighbor_coordinates_3	page 2386
CGAL::surface_neighbors_3	page 2390

40.5 Alphabetical List of Reference Pages

Data_access<Map>	page 2370
farin_c1_interpolation	page 2368
GradientFittingTraits	page 2380
InterpolationTraits	page 2371
Interpolation_gradient_fitting_traits_2<K>	page 2382
Interpolation_traits_2<K>	page 2373
linear_interpolation	page 2365
natural_neighbor_coordinates_2	page 2374
quadratic_interpolation	page 2369
regular_neighbor_coordinates_2	page 2376
sibson_c1_interpolation	page 2366
sibson_gradient_fitting	page 2378
surface_neighbors_3	page 2390
surface_neighbor_coordinates_3	page 2386
Voronoi_intersection_2_traits_3<K>	page 2384

CGAL::linear_interpolation

Definition

The function *linear_interpolation* computes the weighted sum of the function values which must be provided via a functor.

```
#include <CGAL/interpolation_functions.h>
```

```
template < class ForwardIterator, class Functor>
typename Functor::result_type
```

```
    linear_interpolation( ForwardIterator first,
                          ForwardIterator beyond,
                          typename          std::iterator_traits<ForwardIterator>::value_
type::second_type norm,
                          Functor function_values)
```

ForwardIterator::value_type is a pair associating a point to a (non-normalized) barycentric coordinate. *norm* is the normalization factor. Given a point, the functor *function_values* allows to access a pair of a function value and a boolean. The boolean indicates whether the function value could be retrieved correctly. This function generates the interpolated function value as the weighted sum of the values corresponding to each point of the point/coordinate pairs in the range *[first, beyond)*.

Precondition: $norm \neq 0$. *function_value(p).second == true* for all points *p* of the point/coordinate pairs in the range *[first, beyond)*.

Requirements

1. *ForwardIterator::value_type* is a pair of point/coordinate value, thus *ForwardIterator::value_type::first_type* is equivalent to a point and *ForwardIterator::value_type::second_type* is a field number type.
2. *Functor::argument_type* must be equivalent to *ForwardIterator::value_type::first_type* and *Functor::result_type* is a pair of the function value type and a boolean value. The function value type must provide a multiplication and addition operation with the field number type *ForwardIterator::value_type::second_type* and a constructor with argument 0. A model of the functor is provided by the struct *Data_access*. It must be instantiated accordingly with an associative container (e.g. STL *std::map*) having the point type as *key_type* and the function value type as *mapped_type*.

See Also

CGAL::Data_access<Map> page [2370](#)
CGAL::natural_neighbor_coordinates_2 page [2374](#)
CGAL::regular_neighbor_coordinates_2 page [2376](#)
CGAL::surface_neighbor_coordinates_3 page [2386](#)

CGAL::sibson_c1_interpolation

Definition

The function *sibson_c1_interpolation* interpolates the function values and the gradients that are provided by functors following the method described in [Sib81].

Parameters

The template parameter *Traits* is to be instantiated with a model of *InterpolationTraits*. *ForwardIterator::value_type* is a pair associating a point to a (non-normalized) barycentric coordinate. *norm* is the normalization factor. The range $[first, beyond)$ contains the barycentric coordinates for the query point *p*. The functor *function_value* allows to access the value of the interpolated function given a point. *function_gradient* allows to access the function gradient given a point.

```
#include <CGAL/interpolation_functions.h>
```

```
template < class ForwardIterator, class Functor, class GradFunctor, class Traits>
std::pair< typename Functor::result_type, bool>
```

```
    sibson_c1_interpolation( ForwardIterator first,
                             ForwardIterator beyond,
                             typename      std::iterator_traits<ForwardIterator>::value_
type::second_type norm,
                             typename      std::iterator_traits<ForwardIterator>::value_
type::first_type p,
                             Functor function_value,
                             GradFunctor function_gradient,
                             Traits traits)
```

This function generates the interpolated function value at the point *p* using Sibson's Z^1 interpolant [Sib81].

If the functor *function_gradient* cannot supply the gradient of a point, the function returns a pair where the boolean is set to *false*. If the interpolation was successful, the pair contains the interpolated function value as first and *true* as second value.

Precondition: *norm* $\neq 0$. *function_value(p).second == true* for all points *p* of the point/coordinate pairs in the range $[first, beyond)$.

Requirements

1. *Traits* is a model of the concept *InterpolationTraits*.
2. *ForwardIterator::value_type* is a point/coordinate pair. Precisely *ForwardIterator::value_type::first_type* is equivalent to *Traits::Point_d* and *ForwardIterator::value_type::second_type* is equivalent to *Traits::FT*.

3. *Functor::argument_type* must be equivalent to *Traits::Point_d* and *Functor::result_type* is a pair of the function value type and a boolean. The function value type must provide a multiplication and addition operation with the type *Traits::FT* as well as a constructor with argument 0.
4. *GradFunctor::argument_type* must be equivalent to *Traits::Point_d* and *Functor::result_type* is a pair of the function's gradient type and a boolean. The function gradient type must provide a multiplication operation with *Traits::Vector_d*.
5. A model of the functor types *Functor* (resp. *GradFunctor*) is provided by the struct *Data_access*. It must be instantiated accordingly with an associative container (e.g. STL *std::map*) having the point type as *key_type* and the function value type (resp. function gradient type) as *mapped_type*.
6. The number type *FT* provided by *Traits* must support the square root operation *sqrt()*.

```
template < class ForwardIterator, class Functor, class GradFunctor, class Traits>
typename Functor::result_type
```

```

        sibson_c1_interpolation_square( ForwardIterator first,
                                         ForwardIterator beyond,
                                         typename      std::iterator_traits<ForwardIterator>
::value_type::second_type norm,
                                         Functor function_value,
                                         GradFunctor function_gradient,
                                         Traits traits)

```

The same as above except that no square root operation is needed for *FT*.

See Also

InterpolationTraits	page 2371
GradientFittingTraits	page 2380
CGAL::Data_access<Map>	page 2370
CGAL::sibson_gradient_fitting	page 2378
CGAL::linear_interpolation	page 2365
CGAL::Interpolation_traits_2<K>	page 2373
CGAL::Interpolation_gradient_fitting_traits_2<K>	page 2382
CGAL::natural_neighbor_coordinates_2	page 2374
CGAL::regular_neighbor_coordinates_2	page 2376
CGAL::surface_neighbor_coordinates_3	page 2386

CGAL::farin_c1_interpolation

Definition

The function *farin_c1_interpolation* interpolates the function values and the gradients that are provided by functors using the method described in [Far90].

```
#include <CGAL/interpolation_functions.h>
```

Parameters

RandomAccessIterator::value_type is a pair associating a point to a (non-normalized) barycentric coordinate. See *sibson_c1_interpolation* for the other parameters.

```
template < class RandomAccessIterator, class Functor, class GradFunctor, class Traits>
typename Functor::result_type
```

```
    farin_c1_interpolation( RandomAccessIterator first,
                           RandomAccessIterator beyond,
                           typename std::iterator_traits<RandomAccessIterator>::value_
type::second_type norm,
                           Functor function_value,
                           GradFunctor function_gradient,
                           Traits traits)
```

generates the interpolated function value computed by Farin's interpolant [Far90]. See also *sibson_c1_interpolation*.

Precondition: *norm* $\neq 0$. *function_value(p).second* == *true* for all points *p* of the point/coordinate pairs in the range [*first*, *beyond*).

Precondition: The range [*first*, *beyond*) contains either one or more than three elements.

Requirements

Same requirements as for *sibson_c1_interpolation* only the iterator must provide random access and *Traits::FT* does not need to provide the square root operation.

See Also

CGAL::Data_access<Map> page 2370
 CGAL::linear_interpolation page 2365
 CGAL::sibson_c1_interpolation page 2366
 CGAL::sibson_gradient_fitting page 2378
 CGAL::Interpolation_traits_2<K> page 2373
 CGAL::natural_neighbor_coordinates_2 page 2374
 CGAL::regular_neighbor_coordinates_2 page 2376
 CGAL::surface_neighbor_coordinates_3 page 2386

CGAL::quadratic_interpolation

Definition

The function *quadratic_interpolation* interpolates the function values and first degree functions defined from the function gradients. Both, function values and gradients, must be provided by functors.

```
#include <CGAL/interpolation_functions.h>
```

Parameters

See *sibson_c1_interpolation*.

```
template < class ForwardIterator, class Functor, class GradFunctor, class Traits>
typename Functor::result_type
```

```
    quadratic_interpolation( ForwardIterator first,
                           ForwardIterator beyond,
                           typename std::iterator_traits<ForwardIterator>:: value_
type::second_type norm,
                           Functor function_value,
                           GradFunctor function_gradient,
                           Traits traits)
```

This function generates the interpolated function value as the weighted sum of the values plus a linear term in the gradient for each point of the point/coordinate pairs in the range $[first, beyond)$. See also *sibson_c1_interpolation*.

Precondition: $norm \neq 0$ $function_value(p).second == true$ for all points p of the point/coordinate pairs in the range $[first, beyond)$.

Requirements

Same requirements as for *sibson_c1_interpolation* only that *Traits::FT* does not need to provide the square root operation.

See Also

InterpolationTraits page 2371
 GradientFittingTraits page 2380
 CGAL::Data_access<Map> page 2370
 CGAL::sibson_gradient_fitting page 2378
 CGAL::linear_interpolation page 2365
 CGAL::Interpolation_traits_2<K> page 2373
 CGAL::Interpolation_gradient_fitting_traits_2<K> page 2382
 CGAL::natural_neighbor_coordinates_2 page 2374
 CGAL::regular_neighbor_coordinates_2 page 2376
 CGAL::surface_neighbor_coordinates_3 page 2386

CGAL::Data_access<Map>

Definition

The struct *Data_access<Map>* implements a functor that allows to retrieve data from an associative container. The functor keeps a reference to the container. Given an instance of the container's key type, it returns a pair of the container's value type and a boolean indicating whether the retrieval was successful.

This class can be used to provide the values and gradients of the interpolation functions.

```
#include <CGAL/interpolation_functions.h>
```

Parameters

The class *Data_access<Map>* has the container type *Map* as template parameter.

Types

```
typedef Map::mapped_type
```

```
                                Data_type;
typedef Map::key_type
```

```
                                Key_type;
```

Creation

```
Data_access<Map> data_access( Map map);
```

Introduces a *Data_access* to the container *map*.

```
std::pair< Data_type, bool>
```

```
    data_access( Key_type p)
```

If there is an entry for *p* in the container *map*, then the pair of *map.find(p)* and *true* is returned. Otherwise, the boolean value of the pair is *false*.

InterpolationTraits

Definition

Most interpolation functions are parameterized by a traits class that defines the primitives used in the interpolation algorithms. The concept `InterpolationTraits` defines this common set of requirements.

Types

<i>InterpolationTraits:: FT</i>	The number type must follow the model <i>FieldNumberType</i> .
<i>InterpolationTraits:: Point_d</i>	The point type on which the function is defined and interpolated.
<i>InterpolationTraits:: Vector_d</i>	The corresponding vector type.
<i>InterpolationTraits:: Construct_vector_d</i>	A constructor object for <i>Vector_d</i> . Provides : <i>Vector_d operator() (Point_d a, Point_d b)</i> which produces the vector $b - a$ and <i>Vector_d operator() (Null_vector NULL_VECTOR)</i> which introduces the null vector.
<i>InterpolationTraits:: Construct_scaled_vector_d</i>	Constructor object for <i>Vector_d</i> . Provides : <i>Vector_d operator() (Vector_d v, FT scale)</i> which produces the vector v scaled by a factor $scale$.
<i>InterpolationTraits:: Compute_squared_distance_d</i>	Constructor object for <i>FT</i> . Provides the operator: <i>FT operator() (Point_d a, Point_d b)</i> returning the squared distance between a and b .
<i>InterpolationTraits traits;</i>	default constructor.

Construction objects

The following functions that create instances of the above constructor object types must exist.

Construct_vector_d *traits.construct_vector_d_object()*

Construct_scaled_vector_d

traits.construct_scaled_vector_d_object()

Compute_squared_distance_d

traits.compute_squared_distance_d_object()

Has Models

<i>CGAL::Interpolation_traits_2<K></i>	page 2373
<i>CGAL::Interpolation_gradient_fitting_traits_2<K></i>	page 2382

See Also

<i>GradientFittingTraits</i>	page 2380
<i>CGAL::sibson_c1_interpolation</i>	page 2366
<i>CGAL::sibson_gradient_fitting</i>	page 2378
<i>CGAL::farin_c1_interpolation</i>	page 2368
<i>CGAL::quadratic_interpolation</i>	page 2369

CGAL::Interpolation_traits_2<K>

Definition

Interpolation_traits_2<K> is a model for the concept *InterpolationTraits* and can be used to instantiate the geometric traits class of interpolation methods applied on a bivariate function over a two-dimensional domain. The traits class is templated by a kernel class *K*.

```
#include <CGAL/Interpolation_traits_2.h>
```

Is Model for the Concepts

InterpolationTraits page [2371](#)

Types

```
typedef K::FT          FT;
typedef K::Point_2     Point_d;
typedef K::Vector_2    Vector_d;
typedef K::Construct_vector_2
                        Construct_vector_d;
typedef K::Construct_scaled_vector_2
                        Construct_scaled_vector_d;
typedef K::Compute_squared_distance_2
                        Compute_squared_distance_d;
```

Operations

```
Construct_scaled_vector_d
                        traits.construct_scaled_vector_d_object()

Construct_vector_d     traits.construct_vector_d_object()

Compute_squared_distance_d
                        traits.compute_squared_distance_d_object()
```

See Also

InterpolationTraits page [2371](#)
 GradientFittingTraits page [2380](#)
 CGAL::Interpolation_gradient_fitting_traits_2<K> page [2382](#)

CGAL::natural_neighbor_coordinates_2

Definition

The function *natural_neighbor_coordinates_2* computes natural neighbor coordinates, also called Sibson's coordinates, for 2D points provided a two-dimensional triangulation and a query point inside the convex hull of the vertices of the triangulation.

```
#include <CGAL/natural_neighbor_coordinates_2.h>
```

```
template < class Dt, class OutputIterator >
```

```
CGAL::Triple< OutputIterator, typename Dt::Geom_traits::FT, bool >
```

```
    natural_neighbor_coordinates_2( Dt dt,
                                   typename Dt::Geom_traits::Point_2 p,
                                   OutputIterator out,
                                   typename Dt::Face_handle start = typename
Dt::Face_handle())
```

computes the natural neighbor coordinates for p with respect to the points in the two-dimensional Delaunay triangulation dt . The template class Dt should be of type *Delaunay_triangulation_2*<*Traits*, *Tds*>. The value type of the *OutputIterator* is a pair of $Dt::Point_2$ and the coordinate value of type $Dt::Geom_traits::FT$. The sequence of point/coordinate pairs that is computed by the function is placed starting at *out*. The function returns a triple with an iterator that is placed past-the-end of the resulting sequence of point/coordinate pairs, the normalization factor of the coordinates and a boolean value which is set to true iff the coordinate computation was successful, i.e. if p lies inside the convex hull of the points in dt .

```
template <class Dt, class OutputIterator, class EdgeIterator >
```

```
CGAL::Triple< OutputIterator, typename Dt::Geom_traits::FT, bool >
```

```
    natural_neighbor_coordinates_2( Dt dt,
                                   typename Dt::Geom_traits::Point_2 p,
                                   OutputIterator out,
                                   EdgeIterator hole_begin,
                                   EdgeIterator hole_end)
```

The same as above. *hole_begin* and *hole_end* determines the iterator range over the boundary edges of the conflict zone of p in the triangulation. It is the result of the function *T.get_boundary_of_conflicts*(p , *std::back_inserter*(*hole*), *start*), see *Delaunay_triangulation_2*<*Traits*, *Tds*>.

```
template <class Dt, class OutputIterator>
```

CGAL::Triple< OutputIterator, typename Dt::Geom_traits::FT, bool >

*natural_neighbor_coordinates_2(Dt dt,
 typename Dt::Vertex_handle vh,
 OutputIterator out)*

This function computes the natural neighbor coordinates of the point *vh->point()* with respect to the vertices of *dt* excluding *vh->point()*. The same as above for the remaining parameters.

Requirements

1. *Dt* are equivalent to the class *Delaunay_triangulation_2<Traits, Tds>*.
2. The traits class *Traits* of *Dt* is a model of the concept *DelaunayTriangulationTraits_2*. Only the following members of this traits class are used:
 - *Construct_circumcenter_2*
 - *FT*
 - *Point_2*
 - *construct_circumcenter_2_object*

Additionally, *Traits* must meet the requirements for the traits class of the *polygon_area_2* function.

3. *OutputIterator::value_type* is equivalent to *std::pair<Dt::Point_2, Dt::Geom_traits::FT>*, i.e. a pair associating a point and its natural neighbor coordinate.

See Also

CGAL::linear_interpolation [page 2365](#)
CGAL::sibson_c1_interpolation [page 2366](#)
CGAL::surface_neighbor_coordinates_3 [page 2386](#)
CGAL::regular_neighbor_coordinates_2 [page 2376](#)

Implementation

This function computes the area of the sub-cells stolen from the Voronoi cells of the points in *dt* when inserting *p*. The total area of the Voronoi cell of *p* is also computed and returned by the function. If *p* lies outside the convex hull, the coordinate values cannot be computed and the third value of the result triple is set to *false*.

CGAL::regular_neighbor_coordinates_2

Definition

The function *regular_neighbor_coordinates_2* computes natural neighbor coordinates, also called Sibson's coordinates, for weighted 2D points provided a two-dimensional regular triangulation and a (weighted) query point inside the convex hull of the vertices of the triangulation. We call these coordinates regular neighbor coordinates.

```
#include <CGAL/regular_neighbor_coordinates_2.h>
```

```
template < class Rt, class OutputIterator >
```

```
CGAL::Triple< OutputIterator, typename Rt::Geom_traits::FT, bool >
```

```
    regular_neighbor_coordinates_2( Rt rt,
                                   typename Rt::Weighted_point p,
                                   OutputIterator out,
                                   typename Rt::Face_handle start = typename
Rt::Face_handle())
```

computes the regular neighbor coordinates for p with respect to the weighted points in the two-dimensional regular triangulation rt . The template class Rt should be of type *Regular_triangulation_2*<*Traits*, *Tds*>. The value type of the *OutputIterator* is a pair of $Rt::Weighted_point$ and the coordinate value of type $Rt::Geom_traits::FT$. The sequence of point/coordinate pairs that is computed by the function is placed starting at *out*. The function returns a triple with an iterator that is placed past-the-end of the resulting sequence of point/coordinate pairs, the normalization factor of the coordinates and a boolean value which is set to true iff the coordinate computation was successful, i.e. if p lies inside the convex hull of the points in rt .

```
template <class Rt, class OutputIterator, class EdgeIterator, class VertexIterator >
```

```
CGAL::Triple< OutputIterator, typename Traits::FT, bool >
```

```
    regular_neighbor_coordinates_2( Rt rt,
                                   typename Traits::Weighted_point p,
                                   OutputIterator out,
                                   EdgeIterator hole_begin,
                                   EdgeIterator hole_end,
                                   VertexIterator hidden_vertices_begin,
```


VertexIterator hidden_vertices_end)

The same as above. *hole_begin* and *hole_end* determines the iterator range over the boundary edges of the conflict zone of *p* in the triangulation *rt*. *hidden_vertices_begin* and *hidden_vertices_end* determines the iterator range over the hidden vertices of the conflict zone of *p* in *rt*. It is the result of the function *T.get_boundary_of_conflicts(p, std::back_inserter(hole), std::back_inserter(hidden_vertices), start)*, see *Regular_triangulation_2<Traits, Tds>*.

```
template <class Rt, class OutputIterator>
CGAL::Triple< OutputIterator, typename Rt::Geom_traits::FT, bool >
```

```
regular_neighbor_coordinates_2( Rt rt,
                               typename Rt::Vertex_handle vh,
                               OutputIterator out)
```

This function computes the regular neighbor coordinates of the point *vh->point()* with respect to the vertices of *rt* excluding *vh->point()*. The same as above for the remaining parameters.

Requirements

1. *Rt* are equivalent to the class *Regular_triangulation_2<Traits, Tds>*.
2. The traits class *Traits* of *Rt* is a model of the concept *RegularTriangulationTraits_2*. It provides the number type *FT* which is a model for *FieldNumberType* and it must meet the requirements for the traits class of the *polygon_area_2* function. A model of this traits class is *Regular_triangulation_euclidean_traits_2<K, Weight>*.
3. *OutputIterator::value_type* is equivalent to *std::pair<Rt::Weighted_point, Rt::Geom_traits::FT>*, i.e. a pair associating a point and its regular neighbor coordinate.

Implementation

This function computes the areas stolen from the Voronoi cells of points in *rt* by the insertion of *p*. The total area of the Voronoi cell of *p* is also computed and returned by the function. If *p* lies outside the convex hull, the coordinate values cannot be computed and the third value of the result triple is set to *false*.

See Also

CGAL::natural_neighbor_coordinates_2 page [2374](#)

CGAL::sibson_gradient_fitting

Definition

The function *sibson_gradient_fitting* approximates the gradient of a function at a point p given natural neighbor coordinates for p and its neighbors' function values. The approximation method is described in [Sib81]. Further functions are provided to fit the gradient for all data points that lie inside the convex hull of the data points. One function exists for each type of natural neighbor coordinates.

```
#include <CGAL/sibson_gradient_fitting.h>
```

```
template < class ForwardIterator, class Functor, class Traits>
typename Traits::Vector_d
```

```
        sibson_gradient_fitting( ForwardIterator first,
                                ForwardIterator beyond,
                                typename      std::iterator_traits<ForwardIterator>::value_
type::second_type norm,
                                typename      std::iterator_traits<ForwardIterator>::value_
type::first_type p,
                                Functor f,
                                Traits traits)
```

This function estimates the gradient of a function at the point p given natural neighbor coordinates of p in the range $[first, beyond)$ and the function values of the neighbors provided by the functor f . *norm* is the normalization factor of the barycentric coordinates.

Requirements

1. *ForwardIterator::value_type* is a pair of point/coordinate value, thus *ForwardIterator::value_type::first_type* is equivalent to a point and *ForwardIterator::value_type::second_type* is a number type.
2. *Functor::argument_type* must be equivalent to *ForwardIterator::value_type::first_type* and *Functor::result_type* is the function value type. It must provide a multiplication and addition operation with the type *ForwardIterator::value_type::second_type*.
3. *Traits* is a model of the concept *GradientFittingTraits*.

```
template < class Dt, class OutputIterator, class Functor, class Traits>
```

OutputIterator *sibson_gradient_fitting_nn_2(Dt dt, OutputIterator out, Functor f, Traits traits)*

estimates the function gradients at all vertices of *dt* that lie inside the convex hull using the coordinates computed by the function *natural_neighbor_coordinates_2*. *OutputIterator::value_type* is a pair associating a point to a vector. The sequence of point/gradient pairs computed by this function is placed starting at *out*. The function returns an iterator that is placed past-the-end of the resulting sequence. The requirements are the same as above. The template class *Dt* must be equivalent to *Delaunay_triangulation_2<Gt, Tds>*.

template < class Rt, class OutputIterator, class Functor, class Traits>
OutputIterator *sibson_gradient_fitting_rn_2(Rt rt, OutputIterator out, Functor f, Traits traits)*

estimates the function gradients at all vertices of *rt* that lie inside the convex hull using the coordinates computed by the function *regular_neighbor_coordinates_2*. *OutputIterator::value_type* is a pair associating a point to a vector. The sequence of point/gradient pairs computed by this function is placed starting at *out*. The function returns an iterator that is placed past-the-end of the resulting sequence. The requirements are the same as above. The template class *Rt* must be equivalent to *Regular_triangulation_2<Gt, Tds>*.

See Also

<i>CGAL::linear_interpolation</i>	page 2365
<i>CGAL::sibson_c1_interpolation</i>	page 2366
<i>CGAL::farin_c1_interpolation</i>	page 2368
<i>CGAL::quadratic_interpolation</i>	page 2369
<i>CGAL::Interpolation_gradient_fitting_traits_2<K></i>	page 2382
<i>CGAL::natural_neighbor_coordinates_2</i>	page 2374
<i>CGAL::regular_neighbor_coordinates_2</i>	page 2376
<i>CGAL::surface_neighbor_coordinates_3</i>	page 2386

Implementation

This function implements Sibson's gradient estimation method based on natural neighbor coordinates [[Sib81](#)].

GradientFittingTraits

Definition

The function *sibson_gradient_fitting* is parameterized by a traits class that defines the primitives used by the algorithm. The concept *GradientFittingTraits* defines this common set of requirements.

Types

<i>GradientFittingTraits:: FT</i>	The number type must follow the model <i>FieldNumberType</i> .
<i>GradientFittingTraits:: Point_d</i>	The point type on which the function is defined and interpolated.
<i>GradientFittingTraits:: Vector_d</i>	The corresponding vector type.
<i>GradientFittingTraits:: Aff_transformation_d</i>	defines a matrix type. Must provide the following member functions : <i>Aff_transformation tr.inverse ()</i> which gives the inverse transformation, and <i>Aff_transformation tr.transform(Vector v)</i> which returns the multiplication of <i>tr</i> with <i>v</i> .
<i>GradientFittingTraits:: Construct_vector_d</i>	A constructor object for <i>Vector_d</i> . Provides : <i>Vector_d operator() (Point_d a, Point_d b)</i> which produces the vector <i>b - a</i> and <i>Vector_d operator() (Null_vector NULL_VECTOR)</i> which introduces the null vector.
<i>GradientFittingTraits:: Construct_scaled_vector_d</i>	Constructor object for <i>Vector_d</i> . Provides : <i>Vector_d operator() (Vector_d v,FT scale)</i> which produces the vector <i>v</i> scaled by a factor <i>scale</i> .
<i>GradientFittingTraits:: Construct_null_matrix_d</i>	Constructor object for <i>Aff_transformation_d</i> . Provides : <i>Aff_transformation_d operator()()</i> which introduces an affine transformation whose matrix has only zero entries.
<i>GradientFittingTraits:: Construct_scaling_matrix_d</i>	Constructor object for <i>Aff_transformation_d</i> . Provides : <i>Aff_transformation_d operator()(FT scale)</i> which introduces a scaling by a scale factor <i>scale</i> .

GradientFittingTraits:: Construct_sum_matrix_d

Constructor object for *Aff_transformation_d*. Provides :
Aff_transformation_d operator()(Aff_transformation_d tr1,
Aff_transformation_d tr2) which returns the sum of the two
matrices representing *tr1* and *tr2*.

GradientFittingTraits:: Construct_outer_product_d

Constructor object for *Aff_transformation_d*. Provides :
Aff_transformation_d operator()(Vector v) which returns the
outerproduct, i.e. the quadratic matrix $v^t v$.

Creation

GradientFittingTraits traits; default constructor.

Operations

The following functions that create instances of the above constructor object types must exist.

Construct_vector_d traits.construct_vector_d_object()

Construct_scaled_vector_d

traits.construct_scaled_vector_d_object()

Construct_null_matrix_d

traits.construct_null_matrix_d_object()

Construct_sum_matrix_d

traits.construct_sum_matrix_d_object()

Construct_outer_product_d

traits.construct_outer_product_d_object()

Has Models

CGAL::Interpolation_gradient_fitting_traits_2<K>

See Also

InterpolationTraits page [2371](#)
CGAL::Interpolation_traits_2<K> page [2373](#)
CGAL::sibson_gradient_fitting page [2378](#)
CGAL::sibson_c1_interpolation page [2366](#)
CGAL::farin_c1_interpolation page [2368](#)
CGAL::quadratic_interpolation page [2369](#)

CGAL::Interpolation_gradient_fitting_traits_2<K>

Definition

Interpolation_gradient_fitting_traits_2<K> is a model for the concepts *InterpolationTraits* and *GradientFittingTraits*. It can be used to instantiate the geometric traits class of interpolation functions and of Sibson's gradient fitting function when applied on a function defined over a two-dimensional domain. The traits class is templated by a kernel class *K*.

```
#include <CGAL/Interpolation_gradient_fitting_traits_2.h>
```

Is Model for the Concepts

GradientFittingTraits *InterpolationTraits*

Types

```
typedef K::FT          FT;
typedef K::Point_2     Point_d;
typedef K::Vector_2    Vector_d;
typedef K::Aff_transformation_2 Aff_transformation_d;
typedef K::Construct_vector_2 Construct_vector_d;
typedef K::Construct_scaled_vector_2 Construct_scaled_vector_d;
typedef K::Compute_squared_distance_2 Compute_squared_distance_d;
typedef Construct_null_matrix_2<Aff_transformation_d>
                                Construct_null_matrix_d;
typedef Construct_scaling_matrix_2<Aff_transformation_d>
                                Construct_scaling_matrix_d;
typedef Construct_sum_matrix_2<Aff_transformation_d>
                                Construct_sum_matrix_d;
typedef Construct_outer_product_2<K> Construct_outer_product_d;
```

Operations

<i>Construct_scaled_vector_d</i>	<i>traits.construct_scaled_vector_d_object()</i>
<i>Construct_vector_d</i>	<i>traits.construct_vector_d_object()</i>
<i>Compute_squared_distance_d</i>	<i>traits.compute_squared_distance_d_object()</i>
<i>Construct_null_matrix_d</i>	<i>traits.construct_null_matrix_d_object()</i>
<i>Construct_scaling_matrix_d</i>	<i>traits.construct_scaling_matrix_d_object()</i>

<i>Construct_sum_matrix_d</i>	<i>traits.construct_sum_matrix_d_object()</i>
<i>Construct_outer_product_d</i>	<i>traits.construct_outer_product_d_object()</i>

See Also

InterpolationTraits	page 2371
GradientFittingTraits	page 2380
CGAL::Interpolation_traits_2<K>	page 2373

CGAL::Voronoi_intersection_2_traits_3<K>

Definition

Voronoi_intersection_2_traits_3<K> is a model for the concept *RegularTriangulationTraits_2* and *InterpolationTraits*. It can be used to instantiate the geometric traits class of a two-dimensional regular triangulation. A three-dimensional plane is defined by a point and a vector that are members of the traits class. The triangulation is defined on 3D points. It is the regular triangulation of the input points projected onto the plane and each weighted with the negative squared distance of the input point to the plane. It can be shown that it is dual to the power diagram obtained by intersecting the three-dimensional Voronoi diagram of the input points with the plane. All predicates and constructions used in the computation of the regular triangulation are formulated on the three dimensional points without explicitly constructing the projected points and the weights. This reduces the arithmetic demands. The traits class is templated by a kernel class *K*.

```
#include <CGAL/Voronoi_intersection_2_traits_3.h>
```

Is Model for the Concepts

RegularTriangulationTraits_2 page [1408](#)

Types

```
typedef K::RT          Weight;
```

```
typedef K::FT          FT;
```

```
typedef K::Point_3     Point_2;
```

```
typedef K::Segment_3
```

```
Segment_2;
```

```
typedef K::Triangle_3
```

```
Triangle_2;
```

```
typedef K::Line_3      Line_2;
```

```
typedef K::Ray_3       Ray_2;
```

```
typedef K::Vector_3    Vector_2;
```

```
typedef K::Construct_triangle_3
```

```
Construct_triangle_2;
```

```
typedef K::Construct_ray_3
```

```
Construct_ray_2;
```

```
typedef K::Compare_distance_3
```

```
Compare_distance_2;
```

```
Compute_area_3<Rep>
```

```
Compute_area_2;
```

An instance of this function object class computes the square root of the result of *K::Compute_squared_area_3*. If the number type *FT* does not support the square root operation, the result is cast to *double* before computing the square root.

typedef Orientation_with_normal_plane_2_3<Rep>

Orientation_2;

typedef Side_of_plane_centered_sphere_2_3<Point_2>

Power_test_2;

typedef Construct_plane_centered_circumcenter_3<Point_2>

Construct_weighted_circumcenter_2;

typedef Compare_first_projection_3<Point_2>

Compare_x_2;

typedef Compare_second_projection_3<Point_2>

Compare_y_2;

Operations

See Also

RegularTriangulationTraits_2 page [1408](#)
CGAL::Regular_triangulation_2<Gt, Tds> page [??](#)
CGAL::regular_neighbor_coordinates_2 page [2376](#)
CGAL::surface_neighbor_coordinates_3 page [2386](#)

CGAL::surface_neighbor_coordinates_3

Definition

The function *surface_neighbor_coordinates_3* computes natural neighbor coordinates for surface points associated to a finite set of sample points issued from the surface. The coordinates are computed from the intersection of the Voronoi cell of the query point p with the tangent plane to the surface at p . If the sampling is sufficiently dense, the coordinate system meets the properties described in the manual pages and in [BF02],[Fl603b]. The query point p needs to lie inside the convex hull of the projection of the sample points onto the tangent plane at p .

```
#include <CGAL/surface_neighbor_coordinates_3.h>
```

```
template <class OutputIterator, class InputIterator, class Kernel>
CGAL::Triple< OutputIterator, typename Kernel::FT, bool >
```

```
    surface_neighbor_coordinates_3( InputIterator first,
                                   InputIterator beyond,
                                   typename Kernel::Point_3 p,
                                   typename Kernel::Vector_3 normal,
                                   OutputIterator out,
                                   Kernel K)
```

The sample points \mathcal{P} are provided in the range $[first, beyond)$. *InputIterator::value_type* is the point type *Kernel::Point_3*. The tangent plane is defined by the point p and the vector *normal*. The parameter K determines the kernel type that will instantiate the template parameter of *Voronoi_intersection_2_traits_3<K>*.

The natural neighbor coordinates for p are computed in the power diagram that results from the intersection of the 3D Voronoi diagram of \mathcal{P} with the tangent plane. The sequence of point/coordinate pairs that is computed by the function is placed starting at *out*. The function returns a triple with an iterator that is placed past-the-end of the resulting sequence of point/coordinate pairs, the normalization factor of the coordinates and a boolean value which is set to true iff the coordinate computation was successful, i.e. if p lies inside the convex hull of the projection of the points \mathcal{P} onto the tangent plane.

```
template <class OutputIterator, class InputIterator, class ITraits>
CGAL::Triple< OutputIterator, typename ITraits::FT, bool >
```

```
    surface_neighbor_coordinates_3( InputIterator first,
                                   InputIterator beyond,
                                   typename ITraits::Point_2 p,
                                   OutputIterator out,
```

ITraits traits)

the same as above only that the traits class must be instantiated by the user. *ITraits* must be equivalent to *Voronoi_intersection_2_traits_3<K>*.

The next functions return, in addition, a second boolean value (the fourth value of the quadrupel) that certifies whether or not, the Voronoi cell of p can be affected by points that lie outside the input range, i.e. outside the ball centered on p passing through the furthest sample point from p in the range $[first, beyond)$. If the sample points are collected by a k -nearest neighbor or a range search query, this permits to check whether the neighborhood which has been considered is large enough.

```
template <class OutputIterator, class InputIterator, class Kernel>
CGAL::Quadruple< OutputIterator, typename Kernel::FT, bool, bool >
```

```
    surface_neighbor_coordinates_certified_3( InputIterator first,
                                              InputIterator beyond,
                                              typename Kernel::Point_3 p,
                                              typename Kernel::Vector_3 normal,
                                              OutputIterator out,
                                              Kernel K)
```

Similar to the first function. The additional fourth return value is *true* if the furthest point in the range $[first, beyond)$ is further away from p than twice the distance from p to the furthest vertex of the intersection of the Voronoi cell of p with the tangent plane defined by $(p, normal)$. It is *false* otherwise.

```
template <class OutputIterator, class InputIterator, class Kernel>
CGAL::Quadruple< OutputIterator, typename Kernel::FT, bool, bool >
```

```
    surface_neighbor_coordinates_certified_3( InputIterator first,
                                              InputIterator beyond,
                                              typename Kernel::Point_2 p,
                                              typename Kernel::FT max_distance,
                                              OutputIterator out,
                                              Kernel kernel)
```

The same as above except that this function takes the maximal distance from p to the points in the range $[first, beyond)$ as additional parameter.

```
template <class OutputIterator, class InputIterator, class ITraits>
CGAL::Quadruple< OutputIterator, typename ITraits::FT, bool, bool >
```

```
    surface_neighbor_coordinates_certified_3( InputIterator first,
                                              InputIterator beyond,
                                              typename ITraits::Point_2 p,
                                              OutputIterator out,
                                              ITraits traits)
```

The same as above only that the traits class must be instantiated by the user and without the parameter *max_distance*. *ITraits* must be equivalent to *Voronoi_intersection_2_traits_3<K>*.

```
template <class OutputIterator, class InputIterator, class ITraits>
CGAL::Quadruple< OutputIterator, typename ITraits::FT, bool, bool >
```

```
surface_neighbor_coordinates_certified_3( InputIterator first,
                                         InputIterator beyond,
                                         typename ITraits::Point_2 p,
                                         typename ITraits::FT max_distance,
                                         OutputIterator out,
                                         ITraits traits)
```

The same as above with the parameter *max_distance*.

The next function allows to filter some potential neighbors of the query point p from \mathcal{P} via its three-dimensional Delaunay triangulation. All surface neighbors of p are necessarily neighbors in the Delaunay triangulation of $\mathcal{P} \cup \{p\}$.

```
template < class Dt, class OutputIterator >
CGAL::Triple< OutputIterator, typename Dt::Geom_traits::FT, bool >
```

```
surface_neighbor_coordinates_3( Dt dt,
                               typename Dt::Geom_traits::Point_2 p,
                               typename Dt::Geom_traits::Vector_3 normal,
                               OutputIterator out,
                               typename Dt::Face_handle start = typename
```

```
Dt::Face_handle())
```

computes the surface neighbor coordinates with respect to the points that are vertices of the Delaunay triangulation dt . The type Dt must be equivalent to *Delaunay_triangulation_3*< Gt , Tds >. The optional parameter *start* is used as a starting place for the search of the conflict zone. It may be the result of the call $dt.locate(p)$. This function instantiates the template parameter *ITraits* to be *Voronoi_intersection_2_traits_3*< $Dt::Geom_traits$ >.

```
template < class Dt, class OutputIterator, class ITraits>
CGAL::Triple< OutputIterator, typename Dt::Geom_traits::FT, bool >
```

```
surface_neighbor_coordinates_3( Dt dt,
                               typename Dt::Geom_traits::Point_2 p,
                               OutputIterator out,
                               ITraits traits,
                               typename Dt::Face_handle start = typename
```

```
Dt::Face_handle())
```

The same as above only that the parameter *traits* instantiates the geometric traits class. Its type *ITraits* must be equivalent to *Voronoi_intersection_2_traits_3*< K >.

Requirements

1. Dt is equivalent to the class *Delaunay_triangulation_3*.

2. *OutputIterator::value_type* is equivalent to *std::pair<Dt::Point_3, Dt::Geom_traits::FT>*, i.e. a pair associating a point and its natural neighbor coordinate.
3. *ITraits* is equivalent to the class *Voronoi_intersection_2_traits_3<K>*.

See Also

<i>CGAL::linear_interpolation</i>	page 2365
<i>CGAL::sibson_c1_interpolation</i>	page 2366
<i>CGAL::farin_c1_interpolation</i>	page 2368
<i>CGAL::Voronoi_intersection_2_traits_3<K></i>	page 2384
<i>CGAL::surface_neighbors_3</i>	page 2390

Implementation

This functions construct the regular triangulation of the input points instantiated with *Voronoi_intersection_2_traits_3<Kernel>* or *ITraits* if provided. They return the result of the function call *regular_neighbor_coordinates_2* with the regular triangulation and *p* as arguments.

CGAL::surface_neighbors_3

Definition

Given a set of sample points issued from a surface and a query point p , the function *surface_neighbors_3* computes the neighbors of p on the surface within the sample points. If the sampling is sufficiently dense, the neighbors are provably close to the point p on the surface (cf. the manual pages and [BF02],[Fl03b]). They are defined to be the neighbors of p in the regular triangulation dual to the power diagram which is equivalent to the intersection of the Voronoi cell of the query point p with the tangent plane to the surface at p .

```
#include <CGAL/surface_neighbors_3.h>
```

```
template <class OutputIterator, class InputIterator, class Kernel>
OutputIterator      surface_neighbors_3( InputIterator first,
                                         InputIterator beyond,
                                         typename Kernel::Point_3 p,
                                         typename Kernel::Vector_3 normal,
                                         OutputIterator out,
                                         Kernel K)
```

The sample points \mathcal{P} are provided in the range $[first, beyond)$. *InputIterator::value_type* is the point type *Kernel::Point_3*. The tangent plane is defined by the point p and the vector *normal*. The parameter K determines the kernel type that will instantiate the template parameter of *Voronoi_intersection_2_traits_3<K>*.

The surface neighbors of p are computed which are the neighbors of p in the regular triangulation that is dual to the intersection of the 3D Voronoi diagram of \mathcal{P} with the tangent plane. The point sequence that is computed by the function is placed starting at *out*. The function returns an iterator that is placed past-the-end of the resulting point sequence.

```
template <class OutputIterator, class InputIterator, class ITraits>
OutputIterator      surface_neighbors_3( InputIterator first,
                                         InputIterator beyond,
                                         typename ITraits::Point_2 p,
                                         OutputIterator out,
                                         ITraits traits)
```

the same as above only that the traits class must be instantiated by the user. *ITraits* must be equivalent to *Voronoi_intersection_2_traits_3<K>*.

The next functions return, in addition, a boolean value that certifies whether or not, the Voronoi cell of p can be affected by points that lie outside the input range, i.e. outside the ball centered on p passing through the furthest sample point from p in the range $[first, beyond)$. If the sample points are collected by a k -nearest neighbor or a range search query, this permits to verify that a large enough neighborhood has been considered.

```
template <class OutputIterator, class InputIterator, class Kernel>
```

std::pair< OutputIterator, bool >

```
surface_neighbors_certified_3( InputIterator first,
                               InputIterator beyond,
                               typename Kernel::Point_3 p,
                               typename Kernel::Vector_3 normal,
                               OutputIterator out,
                               Kernel K)
```

Similar to the first function. The additional third return value is *true* if the furthest point in the range $[first, beyond)$ is further away from p than twice the distance from p to the furthest vertex of the intersection of the Voronoi cell of p with the tangent plane defined by $(p, normal)$. It is *false* otherwise.

```
template <class OutputIterator, class InputIterator, class Kernel>
std::pair< OutputIterator, bool >
```

```
surface_neighbors_certified_3( InputIterator first,
                               InputIterator beyond,
                               typename Kernel::Point_2 p,
                               typename Kernel::FT max_distance,
                               OutputIterator out,
                               Kernel kernel)
```

The same as above except that this function takes the maximal distance from p to the points in the range $[first, beyond)$ as additional parameter.

```
template <class OutputIterator, class InputIterator, class ITraits>
std::pair< OutputIterator, bool >
```

```
surface_neighbors_certified_3( InputIterator first,
                               InputIterator beyond,
                               typename ITraits::Point_2 p,
                               OutputIterator out,
                               ITraits traits)
```

The same as above only that the traits class must be instantiated by the user. *ITraits* must be equivalent to *Voronoi_intersection_2_traits_3<K>*. There is no parameter *max_distance*.

```
template <class OutputIterator, class InputIterator, class ITraits>
std::pair< OutputIterator, bool >
```

```
surface_neighbors_certified_3( InputIterator first,
                               InputIterator beyond,
                               typename ITraits::Point_2 p,
                               typename ITraits::FT max_distance,
                               OutputIterator out,
                               ITraits traits)
```

The same as above with the parameter *max_distance*.

The next function allows to filter some potential neighbors of the query point p from \mathcal{P} via its three-dimensional Delaunay triangulation. All surface neighbors of p are necessarily neighbors in the Delaunay triangulation of $\mathcal{P} \cup \{p\}$.

```
template < class Dt, class OutputIterator >
OutputIterator      surface_neighbors_3( Dt dt,
                                     typename Dt::Geom_traits::Point_2 p,
                                     typename Dt::Geom_traits::Vector_3 normal,
                                     OutputIterator out,
                                     typename Dt::Face_handle start = typename Dt::Face_
handle())
```

computes the surface neighbor coordinates with respect to the points that are vertices of the Delaunay triangulation dt . The type Dt must be equivalent to *Delaunay_triangulation_3*< Gt , Tds >. The optional parameter *start* is used for the used as a starting place for the search of the conflict zone. It may be the result of the call $dt.locate(p)$. This function instantiates the template parameter *ITraits* to be *Voronoi_intersection_2_traits_3*< $Dt::Geom_traits$ >.

```
template < class Dt, class OutputIterator, class ITraits>
OutputIterator      surface_neighbors_3( Dt dt,
                                     typename Dt::Geom_traits::Point_2 p,
                                     OutputIterator out,
                                     ITraits traits,
                                     typename Dt::Face_handle start = typename Dt::Face_
handle())
```

The same as above only that the parameter *traits* instantiates the geometric traits class. Its type *ITraits* must be equivalent to *Voronoi_intersection_2_traits_3*< K >.

Requirements

1. Dt is equivalent to the class *Delaunay_triangulation_3*.
2. *OutputIterator::value_type* is equivalent to $Dt::Point_3$, i.e. a point type.
3. *ITraits* is equivalent to the class *Voronoi_intersection_2_traits_3*< K >.

See Also

CGAL::Voronoi_intersection_2_traits_3< K > page [2384](#)
CGAL::surface_neighbor_coordinates_3 page [2386](#)

Implementation

These functions compute the regular triangulation of the sample points and the point p using a traits class equivalent to *Voronoi_intersection_2_traits_3*< K >. They determine the neighbors of p in this triangulation. The functions which certify the result need to compute, in addition, the Voronoi vertices of the cell of p in this diagram.

Chapter 41

2D Placement of Streamlines

Abdelkrim Mebarki

Contents

41.1 Definitions	2394
41.2 Fundamental Notions	2394
41.2.1 Euler integrator	2394
41.2.2 Second Order Runge Kutta Integrator	2395
41.3 Farthest Point Seeding Strategy	2395
41.4 Implementation	2396
41.5 Examples	2396

This chapter describes the CGAL's 2D streamline placement package. Basic definitions and notions are given in Section 41.1. Section 41.2 gives a description of the integration process. Section 41.3 provides a brief description of the algorithm. Section 41.4 presents the implementation of the package, and Section 41.5 details two example placements.

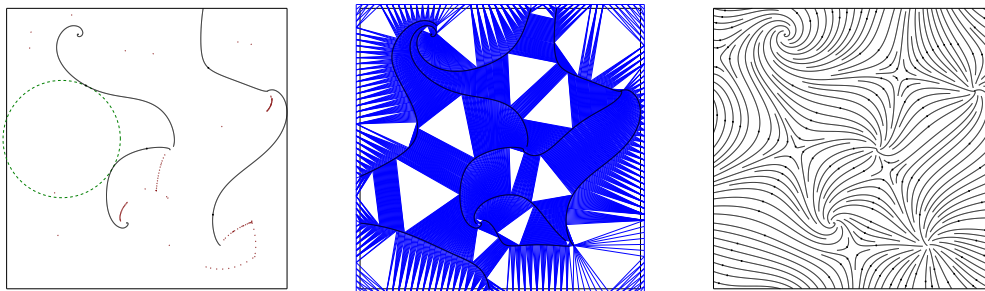


Figure 41.1: The core idea of the algorithm is to integrate the streamlines from the center of the biggest empty cavities in the domain (left). A Delaunay triangulation of all the sample points is used to model the streamlines and the spaces within the domain (middle). A final result is shown (right).

41.1 Definitions

In physics, a *field* is an assignment of a quantity to every point in space. For example, a gravitational field assigns a gravitational potential to each point in space.

Vector and direction fields are commonly used for modeling physical phenomena, where a direction and magnitude, namely a vector is assigned to each point inside a domain (such as the magnitude and direction of the force at each point in a magnetic field).

Streamlines are important tools for visualizing flow fields. A *streamline* is a curve everywhere tangent to the field. In practice, a streamline is often represented as a polyline (series of points) iteratively elongated by bidirectional numerical integration started from a *seed point*, until it comes close to another streamline (according to a specified distance called *the separating distance*), hits the domain boundary, reaches a critical point or generates a closed path.

A *valid* placement of streamlines consists of saturating the domain with a set of tangential streamlines in accordance with a specified *density*, determined by the *separating distance* between the streamlines.

41.2 Fundamental Notions

A streamline can be considered as the path traced by an imaginary massless particle dropped into a steady fluid flow described by the field. The construction of this path consists in the solving an ordinary differential equation for successive time intervals. In this way, we obtain a series of points $p_k, 0 < k < n$ which allow visualizing the streamline. The differential equation is defined as follows :

$$\frac{dp}{dt} = v(p(t)), \quad p(0) = p_0$$

where $p(t)$ is the position of the particle at time t , v is a function which assigns a vector value at each point in the domain (possibly by interpolation), and p_0 is the initial position. The position after a given interval Δt is given by :

$$p(t + \Delta t) = p(t) + \int_t^{t+\Delta t} v(p(t))dt$$

. Several numeric methods have been proposed to solve this equation. In this package, the Euler, and the Second Order Runge Kutta algorithm are implemented.

41.2.1 Euler integrator

This algorithm approximates the point computation by this formula

$$p_{k+1} = p_k + hv(p_k)$$

where h specifies the *integration step* (see Figure 41.2). The integration can be done forward (resp. backward) by specifying a positive (resp. negative) integration step h . The streamline is then constructed by successive integration from a seed point both forward and backward.

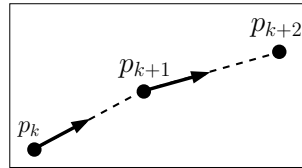


Figure 41.2: Euler integrator.

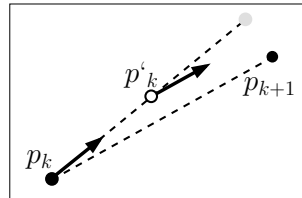


Figure 41.3: Runge-kutta second order integrator (The empty circle represents the intermediate point, and the gray disk represents the Euler integrated point).

41.2.2 Second Order Runge Kutta Integrator

This method introduces an intermediate point p'_k between p_k and p_{k+1} to increase the precision of the computation (see Figure 41.3), where:

$$\begin{aligned} p'_k &= p_k + \frac{1}{2}hv(p_k) \\ p_{k+1} &= p_k + hv(p'_k) \end{aligned}$$

See [PTVF02] for further details about numerical integration.

41.3 Farthest Point Seeding Strategy

The algorithm implemented in this package [MAD05] consists of placing one streamline at a time by numerical integration starting farthest away from all previously placed streamlines.

The input of our algorithm is given by (i) a flow field, (ii) a *density* specified either globally, by the inverse of the ideal spacing distance, or locally by a density field, and (iii) a *saturation* ratio over the desired spacing required to trigger the seeding of a new streamline.

The input flow field is given by a discrete set of vectors or directions sampled within a domain, associated with an interpolation scheme (*e.g.* bilinear interpolation over a regular grid, or natural neighbor interpolation over an irregular point set to allow for an evaluation at each point coordinate within the domain).

The *output* is a streamline placement, represented as a list of streamlines. The core idea of our algorithm consists of placing one streamline at a time by numerical integration seeded at the farthest point from all previously placed streamlines.

The streamlines are approximated by polylines, whose points are inserted to a 2D Delaunay triangulation (see figure 41). The empty circumscribed circles of the Delaunay triangles provide us with a good approximation of the cavities in the domain.

After each streamline integration, all incident triangles whose circumcircle diameter is larger (within the saturation ratio) than the desired spacing distance are pushed to a priority queue sorted by the triangle circumcircle diameter. To start each new streamline integration, the triangle with largest circumcircle diameter (and hence the biggest cavity) is popped out of the queue. We first test if it is still a valid triangle of the triangulation, since it could have been destroyed by a streamline previously added to the triangulation. If it is not, we pop another triangle out of the queue. If it is, we use the center of its circumcircle as seed point to integrate a new streamline.

Our algorithm terminates when the priority queue is empty. The size of the biggest cavity being monotonically decreasing, our algorithm guarantees the domain saturation.

41.4 Implementation

Streamlines are represented as polylines, and are obtained by iterative integration from the seed point. A polyline is represented as a range of points. The computation is processed via a list of Delaunay triangulation vertices.

To implement the triangular grid, the CGAL *Delaunay_triangulation_2* class is used. The priority queue used to store candidate seed points is taken from the Standard Template Library [Sil97].

41.5 Examples

The first example illustrates the generation of a 2D streamline placement from a vector field defined on a regular grid.

```
#include <iostream>
#include <fstream>

#include <CGAL/Cartesian.h>
#include <CGAL/Filtered_kernel.h>

#include <CGAL/Stream_lines_2.h>
#include <CGAL/Runge_kutta_integrator_2.h>
#include <CGAL/Regular_grid_2.h>
#include <CGAL/Triangular_field_2.h>

typedef double coord_type;
typedef CGAL::Cartesian<coord_type> K1;
```

```

typedef CGAL::Filtered_kernel<K1> K;
typedef CGAL::Regular_grid_2<K> Field;
typedef CGAL::Runge_kutta_integrator_2<Field> Runge_kutta_integrator;
typedef CGAL::Stream_lines_2<Field, Runge_kutta_integrator> Stl;
typedef Stl::Point_iterator_2 Point_iterator;
typedef Stl::Stream_line_iterator_2 Stl_iterator;
typedef Stl::Point_2 Point_2;
typedef Stl::Vector_2 Vector_2;

int main()
{
    Runge_kutta_integrator runge_kutta_integrator;

    /*data.vec.cin is an ASCII file containing the vector field values*/
    std::ifstream infile("data/vnoise.vec.cin", std::ios::in);
    double iXSize, iYSize;
    unsigned int x_samples, y_samples;
    iXSize = iYSize = 512;
    infile >> x_samples;
    infile >> y_samples;
    Field regular_grid_2(x_samples, y_samples, iXSize, iYSize);
    /*fill the grid with the appropriate values*/
    for (unsigned int i=0; i<x_samples; i++)
        for (unsigned int j=0; j<y_samples; j++)
        {
            double xval, yval;
            infile >> xval;
            infile >> yval;
            regular_grid_2.set_field(i, j, Vector_2(xval, yval));
        }
    infile.close();

    /* the placement of streamlines */
    std::cout << "processing...\n";
    double dSep = 3.5;
    double dRat = 1.6;
    Stl Stream_lines(regular_grid_2, runge_kutta_integrator, dSep, dRat);
    std::cout << "placement generated\n";

    /*writing streamlines to streamlines_on_regular_grid_1.stl */
    std::ofstream fw("streamlines_on_regular_grid_1.stl", std::ios::out);
    fw << Stream_lines.number_of_lines() << "\n";
    for(Stl_iterator sit = Stream_lines.begin(); sit != Stream_lines.end(); sit++)
    {
        fw << "\n";
        for(Point_iterator pit = sit->first; pit != sit->second; pit++){
            Point_2 p = *pit;
            fw << p.x() << " " << p.y() << "\n";
        }
    }

    fw.close();
}

```

The second example depicts the generation of a streamline placement from a vector field defined on a triangular grid.

```
#include <iostream>
#include <fstream>

#include <CGAL/Cartesian.h>
#include <CGAL/Filtered_kernel.h>

#include <CGAL/Stream_lines_2.h>
#include <CGAL/Runge_kutta_integrator_2.h>
#include <CGAL/Triangular_field_2.h>

typedef double coord_type;
typedef CGAL::Cartesian<coord_type> K1;
typedef CGAL::Filtered_kernel<K1> K;
typedef K::Point_2 Point;
typedef K::Vector_2 Vector;
typedef CGAL::Triangular_field_2<K> Field;
typedef CGAL::Runge_kutta_integrator_2<Field> Runge_kutta_integrator;
typedef CGAL::Stream_lines_2<Field, Runge_kutta_integrator> Stl;
typedef Stl::Stream_line_iterator_2 stl_iterator;

int main()
{
    Runge_kutta_integrator runge_kutta_integrator(1);

    /*datap.tri.cin and datav.tri.cin are ascii files where are stored the vector values*/
    std::ifstream inp("data/datap.tri.cin");
    std::ifstream inv("data/datav.tri.cin");
    std::istream_iterator<Point> beginp(inp);
    std::istream_iterator<Vector> beginv(inv);
    std::istream_iterator<Point> endp;

    Field triangular_field(beginp, endp, beginv);

    /* the placement of streamlines */
    std::cout << "processing...\n";
    double dSep = 30.0;
    double dRat = 1.6;
    Stl Stream_lines(triangular_field, runge_kutta_integrator, dSep, dRat);
    std::cout << "placement generated\n";

    /*writing streamlines to streamlines.stl */
    std::cout << "streamlines.stl\n";
    std::ofstream fw("streamlines.stl", std::ios::out);
    Stream_lines.print_stream_lines(fw);
}
```

2D Placement of Streamlines

Reference Manual

Abdelkrim Mebarki

Vector and direction fields are commonly used for modeling physical phenomena, where a direction and magnitude, namely a vector is assigned to each point inside a domain.

A streamline is a curve everywhere tangent to the field. It can be considered as the path traced by an imaginary massless particle dropped into a steady fluid flow described by the field.

A streamline is represented as a polyline iteratively elongated by bidirectional numerical integration started from a seed point, until it comes close to another streamline, hits the domain boundary, or reaches a critical point.

The *Stream_lines_2* class consists of saturating the domain with a set of tangential streamlines in accordance with a specified density.

Streamlines are represented as containers of points, manipulated by an iterator range of points, and the whole placement is accessible via an iterator range of streamlines.

The main class in this package, the *Stream_lines_2* class of CGAL depends on two template parameters. The first template parameter stands for a class which represents both the vector field and the visualisation domain with operations on them, and should be instantiated by a model of the concept *VectorField_2*. The second template parameter stands for a function object that ensures the numerical integration used to construct the streamlines, and should be instantiated by a model of the concept *Integrator_2*.

41.6 Classified Reference Pages

Concepts

StreamLinesTraits_2	page 2406
Integrator_2	page 2402
VectorField_2	page 2410

Classes

<i>CGAL::Stream_lines_2<VectorField_2,Integrator_2></i>	page 2407
<i>CGAL::Euler_integrator_2<VectorField_2></i>	page 2401
<i>CGAL::Runge_kutta_integrator_2<VectorField_2></i>	page 2405
<i>CGAL::Regular_grid_2<StreamLinesTraits_2></i>	page 2403
<i>CGAL::Triangular_field_2<StreamLinesTraits_2></i>	page 2409

41.7 Alphabetical List of Reference Pages

<i>Euler_integrator_2<VectorField_2></i>	page 2401
<i>Integrator_2</i>	page 2402
<i>Regular_grid_2<StreamLinesTraits_2></i>	page 2403
<i>Runge_kutta_integrator_2<VectorField_2></i>	page 2405
<i>StreamLinesTraits_2</i>	page 2406
<i>Stream_lines_2<VectorField_2,Integrator_2></i>	page 2407
<i>Triangular_field_2<StreamLinesTraits_2></i>	page 2409
<i>VectorField_2</i>	page 2410

CGAL::Euler_integrator_2<VectorField_2>

Definition

This class implements the first order Euler integrator. The template parameter *VectorField_2* has to be instantiated by a model of the concept *VectorField_2*.

Creation

```
Euler_integrator_2<VectorField_2> einteg( FT integration_step);
```

Creates an Euler integrator class *einteg* with *integration_step* as integration step.

Is Model for the Concepts

Integrator_2

See Also

Runge_kutta_integrator_2<*VectorField_2*>

Integrator_2

Definition

The concept `Integrator_2` describes the set of requirements to be fulfilled by any function object used to instantiate the second template parameter of the class `Stream_lines_2<VectorField_2,Integrator_2>`. This concept provides the operation that integrates a new point from a given point with a predefined step, and according to a specified vector.

Types

<code>Integrator_2:: FT</code>	The scalar type.
<code>Integrator_2:: Point_2</code>	The point type.
<code>Integrator_2:: Vector_2</code>	The vector type.
<code>Integrator_2:: Vector_field_2</code>	The vector field type.

Creation

`Integrator_2 integ;` only a default constructor is needed.

Operations

The following operations return the newly integrated point.

`Point_2` `integ(Point_2 p, Vector_field_2 vector_field_2)`

returns the new position from the actual position defined by `p`, according to the vector given by `vector_field_2` at `p`.
Precondition: `vector_field_2.is_in_domain(p)` must be true.

`Point_2` `integ(Point_2 p, Vector_field_2 vector_field_2, FT integration_step)`

As above. The integration step is defined by `integration_step`.
Precondition: `vector_field_2.is_in_domain(p)` must be true.

`Point_2` `integ(Point_2 p, Vector_field_2 vector_field_2, FT integration_step, bool direction)`

As above. In addition, this function integrates forward if *direction* is true, and backward if it is false.
Precondition: `vector_field_2.is_in_domain(p)` must be true.

Has Models

`CGAL::Euler_integrator_2<VectorField_2>`
`CGAL::Runge_kutta_integrator_2<VectorField_2>`

CGAL::Regular_grid_2<StreamLinesTraits_2>

Definition

The template parameter *StreamLinesTraits_2* has to be instantiated by a model of the concept *StreamLinesTraits_2*.

This class provides a 2D vector field specified by a set of sample points defined on a regular grid, with a bilinear interpolation scheme over its cells (i.e. for each point p in a cell c , the vector value is interpolated from the vertices of c).

Types

<code>typedef StreamLinesTraits_2::FT</code>	<code>FT;</code>	the scalar type.
<code>typedef StreamLinesTraits_2::Point_2</code>	<code>Point_2;</code>	the point type.
<code>typedef StreamLinesTraits_2::Vector_2</code>	<code>Vector_2;</code>	the vector type.

Creation

`Regular_grid_2<StreamLinesTraits_2> rgrid(int x_samples, int y_samples, FT x_size, FT y_size);`

Generate a regular grid *rgrid* whose size is *x_size* by *y_size*, while *x_samples* and *y_samples* specify the number of samples on *x* and *y*.

Modifiers

In addition to the minimum interface required by the concept definition, the class *Regular_grid_2<StreamLinesTraits_2>* provides the following function to fill the vector field with the user data.

`void rgrid.set_xy(int i, int j, Vector_2 v)`

Attribute the vector *v* to the position (*i,j*) on the regular grid.

Access Functions

<code>std::pair<int, int></code>	<code>rgrid.get_dimension()</code>	returns the dimension of the grid.
----------------------------------------	------------------------------------	------------------------------------

<code>std::pair<FT, FT></code>	<code>rgrid.get_size()</code>	returns the size of the grid.
--------------------------------------	-------------------------------	-------------------------------

Is Model for the Concepts

VectorField_2

See Also

Triangular_field_2<StreamLinesTraits_2>

CGAL::Runge_kutta_integrator_2<VectorField_2>

Definition

The template parameter *VectorField_2* has to be instantiated by a model of the concept *VectorField_2*. This class implements the second order Runge-Kutta integrator.

Creation

Runge_kutta_integrator_2<VectorField_2> *rkinteg*(*FT* *integration_step*);

Creates a Runge-kutta second order integrator class *rkinteg* with *integration_step* as integration step.

Is Model for the Concepts

Integrator_2

See Also

Euler_integrator_2<VectorField_2>

StreamLinesTraits_2

Definition

The concept `StreamLinesTraits_2` describes the set of requirements to be fulfilled by any class used to instantiate the template parameter of the class `Regular_grid_2<StreamLinesTraits_2>`. This concept provides the types handled by the `Stream_lines_2<VectorField_2, Integrator_2>` class.

Types

<code>StreamLinesTraits_2:: FT</code>	The scalar type.
<code>StreamLinesTraits_2:: Point_2</code>	The point type.
<code>StreamLinesTraits_2:: Vector_2</code>	The vector type.

Has Models

The kernels of CGAL are models for this traits class.

CGAL::Stream_lines_2<VectorField_2,Integrator_2>

Definition

The class *Stream_lines_2<VectorField_2,Integrator_2>* is designed to handle a placement of streamlines in a 2D domain according to a bidimensional vector field.

The class *Stream_lines_2<VectorField_2,Integrator_2>* creates a placement of streamlines according to a specified density and gives access to those streamlines via two iterators over a container of iterators that provide access to the streamline points.

Parameters

The class *Stream_lines_2<VectorField_2,Integrator_2>* has two template parameters. The first parameter *VectorField_2* has to be instantiated by a model of the concept *VectorField_2*. The second parameter is the function object *Integrator_2*, and has to be instantiated by a model of the concept *Integrator_2*.

Types

<i>typedef VectorField_2::Geom_traits</i>	<i>Geom_traits;</i>	the traits class.
<i>typedef VectorField_2::FT</i>	<i>FT;</i>	the scalar type.
<i>typedef VectorField_2::Point_2</i>	<i>Point_2;</i>	the point type.
<i>typedef VectorField_2::Vector_2</i>	<i>Vector_2;</i>	the vector type.

The class *Stream_lines_2<VectorField_2,Integrator_2>* provides also two types for handling streamlines:

Stream_lines_2<VectorField_2,Integrator_2>:: Point_iterator_2;

iterator of points with value type *Point_2*.

Stream_lines_2<VectorField_2,Integrator_2>:: Stream_line_iterator_2;

an iterator to visit the streamlines with value type *std::pair< Point_iterator_2, Point_iterator_2>*.

Creation

```
Stream_lines_2<VectorField_2,Integrator_2> stl( VectorField_2 vector_field_2,
Integrator_2 integrator_2,
FT separating_distance,
FT saturation_ratio)
```

Generates a streamline placement *stl*.

Modifiers

```
void stl.set_separating_distance( FT new_value)
```

Modify the separating distance.

void *stl.set_saturation_ratio(FT new_value)*

Modify the saturation ratio.

void *stl.update()*

Update the placement after changing the separating distance or the saturation ratio.

Access Functions

void *stl.get_separating_distance()*

returns the separating distance.

void *stl.get_saturation_ratio()*

returns the saturation ratio.

void *stl.print_stream_lines(std::ofstream & fw)*

prints the streamlines to an ASCII file : line by line, and point by point.

Streamline iterators

The following iterators allow to visit all the streamlines generated by the constructor or the update function.

Stream_line_iterator *stl.begin()*

Starts at the first streamline

Stream_line_iterator *stl.end()*

Past-the-end iterator

CGAL::Triangular_field_2<StreamLinesTraits_2>

Definition

The template parameter *StreamLinesTraits_2* has to be instantiated by a model of the concept *StreamLinesTraits_2*.

This class provides a vector field specified by a set of sample points defined on a triangulated domain. All sample points are inserted to a *Delaunay triangulation*, and for each point p in the domain located in a face f , its vector value is interpolated from the vertices of the face f .

Types

<code>typedef StreamLinesTraits_2::FT</code>	<code>FT;</code>	the scalar type.
<code>typedef StreamLinesTraits_2::Point_2</code>	<code>Point_2;</code>	the point type.
<code>typedef StreamLinesTraits_2::Vector_2</code>	<code>Vector_2;</code>	the vector type.

Creation

```
Triangular_field_2<StreamLinesTraits_2> tfield_2( InputIterator1 first_point,
                                                    InputIterator1 last_point,
                                                    InputIterator2 first_vector)
```

Defines the points in the range $[first_point, last_point)$ as the sample points of the grid, with the corresponding number of vectors started at *first_vector*.
Precondition: The *value_type* of *InputIterator1* is *Point*.
Precondition: The *value_type* of *InputIterator2* is *Vector*.

Is Model for the Concepts

VectorField_2

See Also

Regular_grid_2<StreamLinesTraits_2>

VectorField_2

Definition

The concept `VectorField_2` describes the set of requirements to be fulfilled by any class used to instantiate the first template parameter of the class `Stream_lines_2<VectorField_2,Integrator_2>`. This concept provides the types of the geometric primitives used in the placement of streamlines and some functions for answering different queries.

Types

<code>VectorField_2::Geom_traits</code>	The traits class.
<code>VectorField_2::FT</code>	The scalar type.
<code>VectorField_2::Point_2</code>	The point type.
<code>VectorField_2::Vector_2</code>	The vector type.

Creation

<code>VectorField_2 vfield;</code>	Any constructor has to allow the user to fill the vector values (i.e. assign a vector to each position within the domain).
------------------------------------	----------------------------------------------------------------------------------------------------------------------------

Query Functions

<code>Geom_traits::Iso_rectangle_2</code>	
<code>std::pair<Vector_2,FT> vfield.bbox()</code>	returns the bounding box of the whole domain.
<code>bool vfield.get_field(Point_2 p)</code>	returns the vector field value and the local density. <i>Precondition:</i> <code>is_in_domain(p)</code> must be true.
<code>bool vfield.is_in_domain(Point_2 p)</code>	returns true if the point <code>p</code> is inside the domain boundaries, false otherwise.
<code>FT vfield.get_integration_step(Point_2 p)</code>	returns the integration step at the point <code>p</code> (i.e. the distance between <code>p</code> and the next point in the polyline.). <i>Precondition:</i> <code>is_in_domain(p)</code> must be true.

Has Models

`CGAL::Regular_grid_2<StreamLinesTraits_2>`
`CGAL::Triangular_field_2<StreamLinesTraits_2>`

Part XIII

Kinetic Data Structures

Chapter 42

Kinetic Data Structures

Daniel Russel

Contents

42.1 An Overview of Kinetic Data Structures and Sweep Algorithms	2413
42.1.1 Terms Used	2414
42.2 An Overview of the Kinetic Framework	2415
42.3 Using Kinetic Data Structures	2416
42.3.1 A Simple Example	2416
42.3.2 Creating Kinetic Primitives	2417
42.3.3 Visualization of Kinetic Data Structures	2418
42.3.4 Extending Kinetic Data Structures	2419

Lets say you want to maintain a sorted list of items (each item is associate with a real number key). You can imagine placing each of the items on the point on the real line corresponding to its key. Now, let the key for each item change continuously (i.e. no jumps are allowed). As long as no two (consecutive) items cross, the sorted order is intact. When two cross, they need to be exchanged in the list and then the sorted order is once again correct. This is a trivial example of a kinetic data structure. The key observation is that the combinatorial structure which is maintained changes at discrete times (events) even though the basic building blocks are changing continuously.

This chapter describes a number of such kinetic data structures implemented using the Kinetic framework described in Chapter 43. We first, in Section 42.1 introduce kinetic data structures and sweep algorithms. This section can be skipped if the reader is already familiar with the area. The next sections, Section 42.1.1 and Section 42.2 introduces the terms and gives an overview of the framework. They are recommended reading for all readers, even if you are just using provided kinetic data structures. We then present kinetic data structures for Delaunay triangulations in two and three dimensions in Section 42.3.

42.1 An Overview of Kinetic Data Structures and Sweep Algorithms

Kinetic data structures were first introduced in by Basch et. al. in 1997 [BGH97]. The idea stems from the observation that most, if not all, computational geometry structures are built using *predicates* — functions on quantities defining the geometric input (e.g. point coordinates), which return a discrete set of values. Many predicates reduce to determining the sign of a polynomial on the defining parameters of the primitive objects. For example, to test whether a point lies above or below a plane we compute the dot product of the point

with the normal of the plane and subtract the plane's offset along the normal. If the result is positive, the point is above the plane, zero on the plane, negative below. The validity of many combinatorial structures built on top of geometric primitives can be verified by checking a finite number of predicates of the geometric primitives. These predicates, which collectively certify the correctness of the structure, are called *certificates*. For a Delaunay triangulation in three dimensions, for example, the certificates are one *InCircle* test per facet of the triangulation, plus a point plane orientation test for each facet or edge of the convex hull.

The kinetic data structures approach is built on top of this view of computational geometry. Let the geometric primitives move by replacing each of their defining quantities with a function of time (generally a polynomial). As time advances, the primitives trace out paths in space called *trajectories*. The values of the polynomial functions of the defining quantities used to evaluate the predicates now also become functions of time. We call these functions *certificate functions*. Typically, a geometric structure is valid when all predicates have a specific non-zero sign. In the kinetic setting, as long as the certificate functions maintain the correct sign as time varies, the corresponding predicates do not change values, and the original data structure remains correct. However, if one of the certificate functions changes sign, the original structure must be updated, as well as the set of certificate functions that verify it. We call such occurrences *events*.

Maintaining a kinetic data structure is then a matter of determining which certificate function changes sign next, i.e. determining which certificate function has the first real root that is greater than the current time, and then updating the structure and the set of certificate functions. In addition, the trajectories of primitives are allowed to change at any time, although C^0 -continuity of the trajectories must be maintained. When a trajectory update occurs for a geometric primitive, all certificates involving that primitive must be updated. We call the collection of kinetic data structures, primitives, event queue and other support structures a *simulation*.

Sweepline algorithms for computing arrangements in d dimensions easily map on to kinetic data structures by taking one of the coordinates of the ambient space as the time variable. The kinetic data structure then maintains the arrangement of a set of objects defined by the intersection of a hyperplane of dimension $d - 1$ with the objects whose arrangement is being computed.

Time is one of the central concepts in a kinetic simulation. Just as static geometric data structures divide the continuous space of all possible inputs (as defined by sets of coordinates) into a discrete set of combinatorial structures, kinetic data structures divide the continuous time domain into a set of disjoint intervals. In each interval the combinatorial structure does not change, so, in terms of the combinatorial structure, all times in the interval are equivalent. We capitalize on this equivalence in the framework in order to simplify computations. If the primitives move on polynomial trajectories and the certificates are polynomials in the coordinates, then events occur at real roots of polynomials of time. Real numbers, which define the endpoints of the interval, are more expensive to compute with than rational numbers, so performing computations at a rational number inside the interval is preferable whenever possible. See Section 43.1.4 for an example of where this equivalence is exploited.

42.1.1 Terms Used

primitive The basic geometric types—i.e. the points of a triangulation. A primitive has a set of *coordinates*.

combinatorial structure A structure built on top of the primitives. The structure does not depend directly on the coordinates of the primitives, only on relationships between them.

trajectory The path traced out by a primitive as time passes. In other words how the coordinates of a primitive change with time.

snapshot The position of all the primitives at a particular moment in time.

static Having to do with geometric data structures on non-moving primitives.

predicate A function which takes the coordinates of several primitives from a snapshot as input and produces one of a discrete set of outputs.

certificate One of a set of predicates which, when all having the correct values, ensure that the combinatorial structure is correct.

certificate function A function of time which is positive when the corresponding certificate has the correct value. When the certificate function changes sign, the combinatorial structure needs to be updated.

event When a certificate function changes sign and the combinatorial structure needs to be updated.

42.2 An Overview of the Kinetic Framework

The provided kinetic data structures are implemented on top of the Kinetic framework presented in Chapter 43. It is not necessary to know the details of the framework, but some familiarity is useful. Here we presented a quick overview of the framework.

The framework is structured around five main concepts. See Figure 42.1 for a schematic of how a kinetic data structure interacts with the various parts. The main concepts are

- the *Kinetic::Simulator*. Models of this concept process events in the correct order and audit kinetic data structures. There should be one instance of a model of this concept per simulation.
- the *Kinetic::Kernel*. The structure of a *Kinetic::Kernel* is analogous to the static CGAL (i.e., non-kinetic) kernels in that it defines a set of primitives and functors which generate certificates from the primitives.
- the *Kinetic::ActiveObjectsTable*. Models of this concept hold a collection of kinetic primitives in a centralized manner. This structure centralizes management of the primitives in order to properly disseminate notifications when trajectories change, new primitives are added or primitives are deleted. There is generally one instance of a model of this concept per simulation.
- the *Kinetic::InstantaneousKernel*. Models of this concept allow existing non-kinetic CGAL data structures to be used on a snapshot of kinetic data. As a result, pre-existing static structures can be used to initialize and audit kinetic data structures.
- the *Kinetic::FunctionKernel*. This concept is the computational kernel of our framework. Models of this concept are responsible for representing, generating and manipulating the motional and certificate functions and their roots. It is this concept that provides the kinetic data structures framework with the necessary algebraic operations for manipulating event times. The *Kinetic::FunctionKernel* is discussed in detail in Section 43.2.

For simplicity, we added an additional concept, that of *Kinetic::SimulationTraits*, which wraps together a particular set of choices for the above concepts and is responsible for creating instances of each of the models. As a user of existing kinetic data structures, this is the only framework object you will have to create. The addition of this concept reduces the choices the user has to make to picking the dimension of the ambient space and choosing between exact and inexact computations. The model of *Kinetic::SimulationTraits* creates an instance each of the *Kinetic::Simulator* and *Kinetic::ActiveObjectsTable*. Handles for these instances as well as instances of the *Kinetic::Kernel* and *Kinetic::InstantaneousKernel* can be requested from the simulation traits class. Both the *Kinetic::Kernel* and the *Kinetic::Simulator* use the *Kinetic::FunctionKernel*, the former to find certificate failure times and the later to operate on them. For technical reasons, each supplied model of *Kinetic::SimulationTraits* also picks out a particular type of kinetic primitive which will be used by the kinetic data structures.

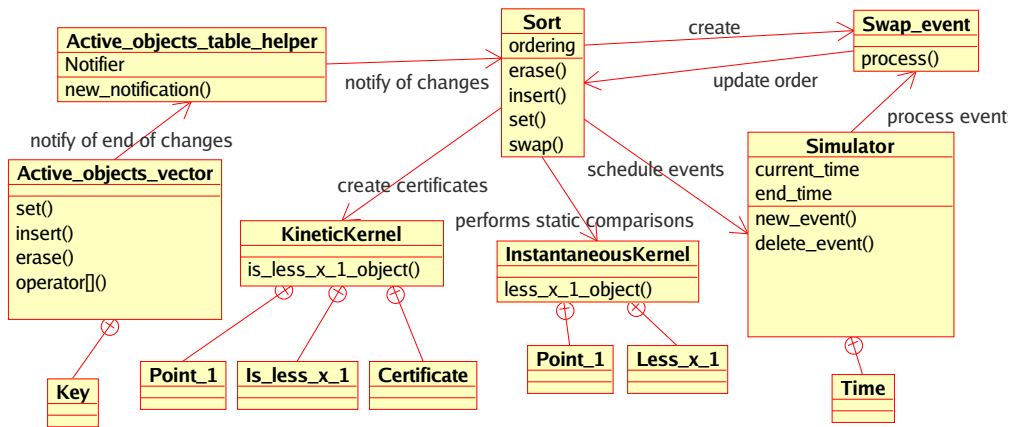


Figure 42.1: The figure shows the interaction between the *Kinetic::Sort<Traits, Visitor>* kinetic data structure and the various pieces of our package. Other, more complicated, kinetic data structures will also use the *Kinetic::InstantaneousKernel* in order to insert/remove geometric primitives and audit themselves. *Kinetic::Sort<Traits, Visitor>* uses the sorting functionality in STL instead.

42.3 Using Kinetic Data Structures

There are five provided kinetic data structures. They are

Kinetic::Sort<Traits, Visitor> maintain a list of points sorted by x-coordinate.

Kinetic::Delaunay_triangulation_2<Traits, Visitor, Triangulation> maintain the Delaunay triangulation of a set of two dimensional points

Kinetic::Delaunay_triangulation_3<Traits, Visitor, Triangulation> maintain the Delaunay triangulation of a set of three dimensional points.

Kinetic::Regular_triangulation_3<Traits, Visitor, Triangulation> maintain the regular triangulation of a set of waiting three dimensional points.

Kinetic::Enclosing_box_2<Traits>, *Kinetic::Enclosing_box_3<Traits>* restrict points to stay within a box by bouncing them off the walls.

42.3.1 A Simple Example

Using a kinetic data structure can be as simple as the following:

```

#include <CGAL/Kinetic/basic.h>

#include <CGAL/Kinetic/Exact_simulation_traits_1.h>
#include <CGAL/Kinetic/Insert_event.h>
#include <CGAL/Kinetic/Sort.h>

int main(int, char *[])

```



```

{

    typedef CGAL::Kinetic::Exact_simulation_traits_1 Traits;

    typedef CGAL::Kinetic::Insert_event<Traits::Active_points_1_table> Insert_event;
    typedef Traits::Active_points_1_table::Data Moving_point;
    typedef CGAL::Kinetic::Sort<Traits> Sort;
    typedef Traits::Simulator::Time Time;

    Traits tr;
    Sort sort(tr);
    Traits::Simulator::Handle sp= tr.simulator_handle();

    std::ifstream in("data/points_1");
    in >> *tr.active_points_1_table_handle();

    while (sp->next_event_time() != sp->end_time()) {
        sp->set_current_event_number(sp->current_event_number()+1);
    }

    return EXIT_SUCCESS;
};

```

Using the other kinetic data structures is substantially identical. Please see the appropriate files in the `demo/Kinetic_data_structures` directory.

In the example, first the *Kinetic::SimulationTraits* object is chosen (in this case one that supports exact computations). Then the kinetic data structure is defined using the chosen traits object and a visitor class which logs changes to the sorted list. Next, instances of the two are created and a set of points is read from a file. Then, the simulator is instructed to process all the events until the end of the simulation. Finally, a record of what happened is printed to the terminal.

Several important things happen behind the scenes in this example. First, the *Kinetic::ActiveObjectsTable* which holds the moving points notifies the kinetic data structure that new points have been added to the simulation. Second, the *Kinetic::Sort<Traits,Visitor>* kinetic data structure registers its events with the *Kinetic::Simulator* by providing a time and a proxy object for each event. When a particular event occurs, the *Kinetic::Simulator* calls a function on the proxy object which in turn updates the kinetic data structure.

The example illustrates how to monitor the supplied data structures as they evolve by using a *Kinetic::SortVisitor* object—a small class whose methods are called whenever the kinetic data structure changes. Hooks for such visitor concepts are provided for all of the shipped kinetic data structures. In the case of kinetic sorting, the visitor’s methods are called every time a new point is inserted in the sorted list, when one is removed, or when two points are swapped in the sorted order.

The visitor concept is quite powerful, allowing us, for example, to implement a data structure for computing and storing two-dimensional arrangements of x -monotone curves on top of the *Kinetic::Sort<Traits, Visitor>* data structure using about 60 lines of code. This sweepline code is presented in Section [42.3.4](#).

42.3.2 Creating Kinetic Primitives

One key part of the framework not shown is how to create kinetic primitives (rather than just reading them in from a file). There are two ways to construction the necessary motion functions (which are models of

Kinetic::FunctionKernel::Function). The first is to create an array of polynomial coefficients and simply call the constructor as in:

```
typedef Traits::Kinetic_kernel::Motion_function F;
std::vector<F::NT> coefs;
coefs.push_back(F::NT(1.0));
coefs.push_back(F::NT(2.0));
F x(coefs.begin(), coefs.end());
```

A slightly more flexible way is to use a *Kinetic::FunctionKernel::ConstructFunction* object. To do this do the following:

```
typedef Traits::Kinetic_kernel::Function_kernel::Construct_function
CF; typedef Traits::Kinetic_kernel::Motion_function F; CF cf; F
x=cf(F::NT(1.0), F::NT(2.0));
```

The *Kinetic::FunctionKernel::ConstructFunction* can be passed (almost) an number of arguments and will construct a polynomial with those arguments are coefficients.

Once the motion functions are constructed, constructing the primitive is just like constructing the corresponding static object.

```
typedef Traits::Kinetic_kernel::Point_1 Point_1;
Point_1 p(x);
```

42.3.3 Visualization of Kinetic Data Structures

The framework includes Qt widgets for displaying kinetic data structures in two and three dimensions. The following example shows using the two dimensional widget with a Delaunay triangulation:

```
#include <CGAL/Kinetic/Exact_simulation_traits_2.h>
#include <CGAL/Kinetic/Delaunay_triangulation_2.h>
#include <CGAL/Kinetic/Enclosing_box_2.h>
#include <CGAL/Kinetic/IO/Qt_moving_points_2.h>
#include <CGAL/Kinetic/IO/Qt_triangulation_2.h>
#include <CGAL/Kinetic/IO/Qt_widget_2.h>

int main(int argc, char *argv[]) {
    using namespace CGAL::Kinetic;
    typedef Exact_simulation_traits_2 Traits;
    typedef Delaunay_triangulation_2<Traits> Del_2;
    typedef Enclosing_box_2<Traits> Box_2;
    typedef Qt_widget_2<Traits::Simulator> Qt_widget;
    typedef Qt_moving_points_2<Traits, Qt_gui> Qt_mps;
    typedef Qt_triangulation_2<Del_2, Qt_widget, Qt_mps> Qt_dt2;

    // create a simulation traits and add two KDSs:
    // a kinetic Delaunay triangulation and an enclosing box;
    // the moving points bounce against the walls of the enclosing box
```

```

Traits tr;
Box_2::Handle box = new Box_2(tr);
Del_2::Handle kdel = new Del_2(tr);

// register the simulator, set of moving points and
// Delaunay triangulation with the kinetic Qt widget
Qt_widget::Handle qt_w = new Qt_widget(argc, argv, tr.simulator_handle());
Qt_mps::Handle qt_mps = new Qt_mps(qt_w, tr);
Qt_dt2::Handle qt_dt2 = new Qt_dt2(kdel, qt_w, qt_mps);

// read the trajectories of the moving points
// the simulation traits automatically inserts them in the two KDSs
// and schedules the appropriate kinetic events; as in the kinetic
// sorting example this is done with appropriate notifications
std::ifstream in("data/points_2");
in >> *tr.active_points_2_table_handle();

// run the interactive kinetic simulation
return qt_w->begin_event_loop();
};

```

The example shows how to use a number of additional features of the framework. First, it shows that two kinetic data structures (*Kinetic::Delaunay_triangulation_2<Traits, Triangulation>* and *Kinetic::Enclosing_box_2<Traits>*) can coexist on the same set of points without any extra effort. Both interact with the moving points through the active objects table, and never need to directly interact with one another. Second, objects (like *qt_w*, *qt_mps* and *qt_dt2*) are all stored by using reference counted handles (*Object::Handle*). This allows them to share references to one another without the user having to worry about memory management and order of deletion. For example, the *Kinetic::Qt_triangulation_2<KineticDelaunay_2, QtWidget_2, Qt_moving_points_2>* object needs a handle to the kinetic triangulation, in order to get the structure to display, and a handle to the *Active_points_1_table* to get the coordinates of the points.

Finally, the example shows how to use the graphical interface elements provided, see Figure 42.3. Our package includes Qt widgets for displaying kinetic geometry in two and three dimensions. In addition to being able to play and pause the simulation, the user can step through events one at a time and reverse the simulation to retrace what had happened. The three-dimensional visualization support is based on the Coin library <http://www.coin3d.org>.

42.3.4 Extending Kinetic Data Structures

Here we present a simple example that uses the *Kinetic::Sort<Traits, Visitor>* kinetic data structure to compute an arrangement of algebraic functions. It wraps the sorting data structure and uses a visitor to monitor changes and map them to corresponding features in the arrangement. To see an example using this kinetic data structure read the example at `examples/Kinetic_data_structures/sweepline.C`.

First we define the visitor class. An object of this type is passed to the *Kinetic::Sort<Traits, Visitor>* data structure and turns events into calls on the arrangement structure. This class has to be defined externally since the arrangement will inherit from the sorting structure.

```

template <class Arrangement>
struct Arrangement_visitor: public Kinetic::Sort_visitor_base
{
    Arrangement_visitor(Arrangement *a):p_(a){}

```

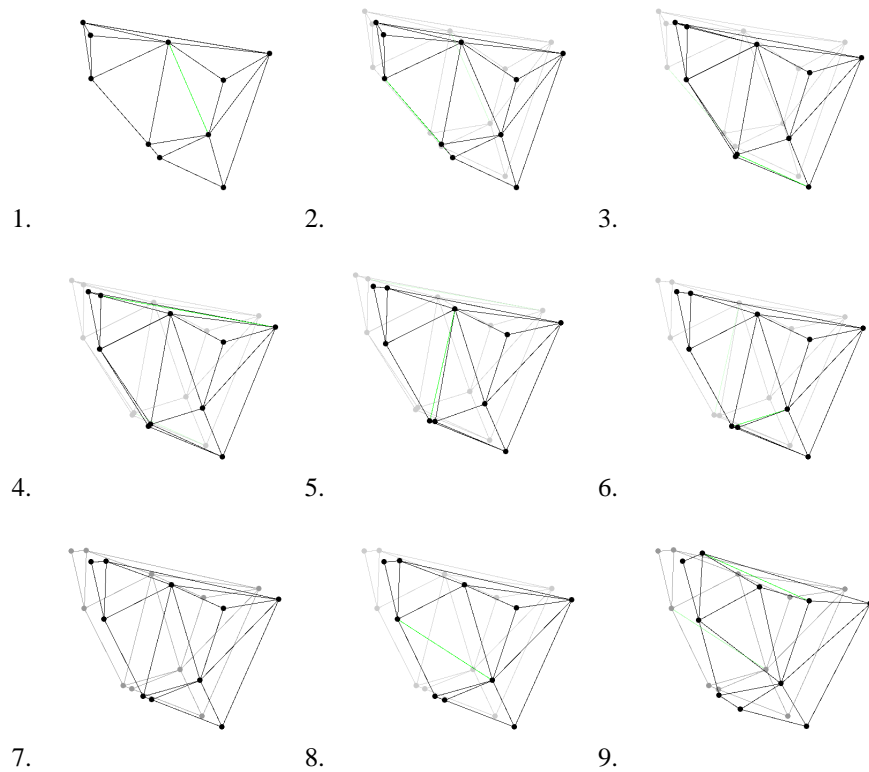


Figure 42.2: *Some events from a Delaunay triangulation kinetic data structure:* The state of the two dimensional Delaunay triangulation immediately following the first events is shown. Green edges are ones which were just created. The pictures are screen shots from `demo/Kinetic_data_structures/Delaunay_triangulation.2.C`.

```
template <class Vertex_handle>
void remove_vertex(Vertex_handle a) {
    p_>erase(a);
}
template <class Vertex_handle>
void create_vertex(Vertex_handle a) {
    p_>insert(a);
}
template <class Vertex_handle>
void after_swap(Vertex_handle a, Vertex_handle b) {
    p_>swap(a, b);
}
Arrangement *p_;
};
```

Now we define the actual arrangement data structure.

```
template <class TraitsT>
class Planar_arrangement:
    public Kinetic::Sort<TraitsT,
```

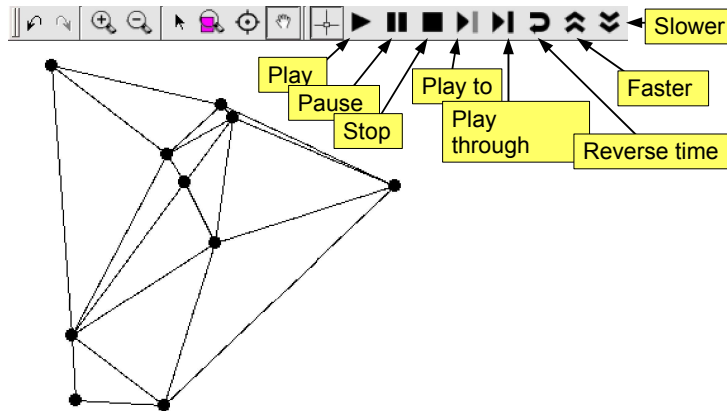


Figure 42.3: The figure shows the graphical user interface for controlling two-dimensional kinetic data structures. It is built on top of the *Qt_widget* and adds buttons to play, pause, step through and run the simulation backwards.

```

    Arrangement_visitor<Planar_arrangement<TraitsT> > > {
    typedef TraitsT Traits;
    typedef Planar_arrangement<TraitsT> This;
    typedef typename Kinetic::Sort<TraitsT,
Arrangement_visitor<This> > Sort;
    typedef Arrangement_visitor<This> Visitor;
    typedef typename Traits::Active_objects_table::Key Key;

public:
    typedef CGAL::Exact_predicates_inexact_constructions_kernel::Point_2 Approximate_point;
    typedef std::pair<int,int> Edge;
    typedef typename Sort::Vertex_handle Vertex_handle;

    // Register this KDS with the MovingObjectTable and the Simulator
    Planar_arrangement(Traits tr): Sort(tr, Visitor(this)) {}

    Approximate_point vertex(int i) const
    {
        return approx_coords_[i];
    }

    size_t vertices_size() const
    {
        return approx_coords_.size();
    }

    typedef std::vector<Edge >::const_iterator Edges_iterator;
    Edges_iterator edges_begin() const
    {
        return edges_.begin();
    }
    Edges_iterator edges_end() const
    {
        return edges_.end();
    }

```

```

void insert(Vertex_handle k) {
    last_points_[*k]=new_point(*k);
}

void swap(Vertex_handle a, Vertex_handle b) {
    int swap_point= new_point(*a);
    edges_.push_back(Edge(swap_point, last_points_[*a]));
    edges_.push_back(Edge(swap_point, last_points_[*b]));
    last_points_[*a]= swap_point;
    last_points_[*b]= swap_point;
}

void erase(Vertex_handle a) {
    edges_.push_back(Edge(last_points_[*a], new_point(*a)));
}

int new_point(typename Traits::Active_objects_table::Key k) {
    double tv= CGAL::to_double(Sort::traits().simulator_handle()->current_time());
    double dv= CGAL::to_double(Sort::traits().active_objects_table_handle()->at(k).x()(tv));
    approx_coords_.push_back(Approximate_point(tv, dv));
    return approx_coords_.size()-1;
}

std::vector<Approximate_point > approx_coords_;
std::map<Key, int> last_points_;
std::vector<Edge> edges_;

};

```

Finally, we have to set everything up. To do this we use some special event classes: *Kinetic::Insert_event<ActiveObjectsTable>* and *Kinetic::Erase_event<ActiveObjectsTable>*. These are events which can be put in the event queue which either insert a primitive into the set of active objects or remove it. Using these, we can allow curves in the arrangement to begin or end in arbitrary places.

```

typedef CGAL::Kinetic::Insert_event<Traits::Active_points_1_table> Insert_event;
typedef CGAL::Kinetic::Erase_event<Traits::Active_points_1_table> Erase_event;
do {
    NT begin, end;
    Point function;
    // initialize the function and the beginning and end somewhere
    tr.simulator_handle()->new_event(Time(begin),
        Insert_event(function, tr.active_points_1_table_handle()));
    tr.simulator_handle()->new_event(Time(end),
        Erase_event(Traits::Active_points_1_table::Key(num),
            tr.active_points_1_table_handle()));
    ++num;
} while (true);

```

Kinetic Data Structures

Reference Manual

Daniel Russel

42.4 Classified Reference Pages

Kinetic data structures are a way of adding motion to classical geometric data structures. CGAL provides several prepackaged kinetic data structures. Here we present those kinetic data structures and the helper classes that allow their activity to be monitored.

Sorting

<i>Kinetic::Sort<Traits, Visitor></i>	page ??
<i>Kinetic::SortVisitor</i>	page 2459
<i>Kinetic::Sort_visitor_base</i>	page ??
<i>Kinetic::Sort_event_log_visitor</i>	page ??

Delaunay Triangulation in 2D

<i>Kinetic::Delaunay_triangulation_2<Traits, Visitor, Triangulation></i>	page ??
<i>Kinetic::DelaunayTriangulationVisitor2</i>	page 2435
<i>Kinetic::Delaunay_triangulation_event_log_visitor_2</i>	page ??
<i>Kinetic::Delaunay_triangulation_recent_edges_visitor_2<Triangulation></i>	page ??
<i>Kinetic::Delaunay_triangulation_visitor_base_2</i>	page ??
<i>Kinetic::Delaunay_triangulation_face_base_2<Traits, Base></i>	page ??

Delaunay and Regular Triangulations in 3D

<i>Kinetic::Delaunay_triangulation_3<Traits, Visitor, Triangulation></i>	page ??
<i>Kinetic::DelaunayTriangulationVisitor3</i>	page 2436

<i>Kinetic::Delaunay_triangulation_event_log_visitor_3</i>	page ??
<i>Kinetic::Delaunay_triangulation_visitor_base_3</i>	page ??
<i>Kinetic::Delaunay_triangulation_cell_base_3<Traits, Base></i>	page ??
<i>Kinetic::Regular_triangulation_3<Traits, Visitor, Triangulation></i>	page ??
<i>Kinetic::RegularTriangulationVisitor3</i>	page 2453
<i>Kinetic::Regular_triangulation_visitor_base_3</i>	page ??
<i>Kinetic::Regular_triangulation_event_log_visitor_3</i>	page ??
<i>Kinetic::Regular_triangulation_cell_base_3<Traits, Base></i>	page ??
<i>Kinetic::Regular_triangulation_vertex_base_3<Traits, Base></i>	page ??

Support Classes

<i>Kinetic::Enclosing_box_2<Traits></i>	page ??
<i>Kinetic::Enclosing_box_3<Traits></i>	page ??
<i>Kinetic::Insert_event<ActiveObjectsTable></i>	page ??
<i>Kinetic::Erase_event<ActiveObjectsTable></i>	page ??
<i>Kinetic::Qt_moving_points_2<Traits, QWidget_2></i>	page ??
<i>Kinetic::Qt_triangulation_2<KineticTriangulation_2, QWidget_2, QtMovingPoints_2></i>	page ??
<i>Kinetic::Qt_widget_2<Simulator></i>	page ??

42.5 Alphabetical List of Reference Pages

<i>Kinetic::DelaunayTriangulationVisitor2</i>	page 2435
<i>Kinetic::DelaunayTriangulationVisitor3</i>	page 2436
<i>Kinetic::Delaunay_triangulation_2<Traits, Visitor, Triangulation></i>	page 2426
<i>Kinetic::Delaunay_triangulation_3<Traits, Visitor, Triangulation></i>	page 2428
<i>Kinetic::Delaunay_triangulation_cell_base_3<Traits, Base></i>	page 2430
<i>Kinetic::Delaunay_triangulation_event_log_visitor_2</i>	page 2431
<i>Kinetic::Delaunay_triangulation_event_log_visitor_3</i>	page 2432
<i>Kinetic::Delaunay_triangulation_face_base_2<Traits, Base></i>	page 2433
<i>Kinetic::Delaunay_triangulation_recent_edges_visitor_2<Triangulation></i>	page 2434
<i>Kinetic::Delaunay_triangulation_visitor_base_2</i>	page 2438
<i>Kinetic::Delaunay_triangulation_visitor_base_3</i>	page 2439
<i>Kinetic::Enclosing_box_2<Traits></i>	page 2440
<i>Kinetic::Enclosing_box_3<Traits></i>	page 2441
<i>Kinetic::Erase_event<ActiveObjectsTable></i>	page 2442
<i>Kinetic::EventLogVisitor</i>	page 2443
<i>Kinetic::Insert_event<ActiveObjectsTable></i>	page 2444
<i>Kinetic::Qt_moving_points_2<Traits, QWidget_2></i>	page 2445

<i>Kinetic::Qt_triangulation_2<KineticTriangulation_2, QtWidget_2, QtMovingPoints_2></i>	page 2446
<i>Kinetic::Qt_widget_2<Simulator></i>	page 2447
<i>Kinetic::RegularTriangulationVisitor3</i>	page 2453
<i>Kinetic::Regular_triangulation_3<Traits, Visitor, Triangulation></i>	page 2448
<i>Kinetic::Regular_triangulation_cell_base_3<Traits, Base></i>	page 2449
<i>Kinetic::Regular_triangulation_event_log_visitor_3</i>	page 2450
<i>Kinetic::Regular_triangulation_instantaneous_traits_3<ActiveObjectsTable, StaticKernel></i>	page 2451
<i>Kinetic::Regular_triangulation_vertex_base_3<Traits, Base></i>	page 2452
<i>Kinetic::Regular_triangulation_visitor_base_3</i>	page 2455
<i>Kinetic::Sort<Traits, Visitor></i>	page 2457
<i>Kinetic::SortVisitor</i>	page 2459
<i>Kinetic::Sort_event_log_visitor</i>	page 2456
<i>Kinetic::Sort_visitor_base</i>	page 2458

CGAL::Kinetic::Delaunay_triangulation_2<Traits, Visitor, Triangulation>

Definition

The class *Kinetic::Delaunay_triangulation_2<Traits, Visitor, Triangulation>* maintains a Delaunay triangulation on top of the points contained in a *Kinetic::ActiveObjectsTable*. It has one main method of interest, *triangulation()*, which returns the triangulation it is maintaining.

The class *Kinetic::Qt_triangulation_2<KineticTriangulation_2, QtWidget_2, QtMovingPoints_2>* displays a kinetic Delaunay triangulation using the Qt widget.

This class is a good example of a simple, but non-trivial, kinetic data structure.

The *Triangulation* template parameter must be a model of *CGAL::Delaunay_triangulation_2<Traits, Tds>* which uses *Traits::Instantaneous_kernel* as its geometric traits and a *Tds* whose face inherits from *Kinetic::Delaunay_triangulation_face_base_2<Traits, Base>*.

The optional *Visitor* parameter takes a model of *Kinetic::DelaunayTriangulationVisitor2*. Methods on this object will be called whenever the triangulation changes.

```
#include <CGAL/Kinetic/Delaunay_triangulation_2.h>
```

Is Model for the Concepts

```
Ref_counted<T>
```

Types

```
Kinetic::Delaunay_triangulation_2<Traits, Visitor, Triangulation>:: Triangulation
```

The template argument triangulation.

```
Kinetic::Delaunay_triangulation_2<Traits, Visitor, Triangulation>:: Visitor
```

The template argument for the visitor.

Creation

```
Kinetic::Delaunay_triangulation_2<Traits, Visitor, Triangulation> dt( Traits tr);
```

Maintain the Delaunay triangulation of the points in *tr.active_points_2_handle()*.

Operations

<i>Triangulation</i>	<i>dt.triangulation()</i>	Access the triangulation that is maintained.
<i>Visitor</i> &	<i>dt.visitor()</i>	Access the visitor.

See Also

Kinetic::DelaunayTriangulationVisitor2, *Kinetic::Delaunay_triangulation_default_visitor_2*,
Kinetic::Delaunay_triangulation_recent_edges_visitor_2<*Triangulation*>, *Kinetic::Delaunay_triangulation_*
event_log_visitor_2, *Kinetic::Qt_Delaunay_triangulation_2*.

CGAL::Kinetic::Delaunay_triangulation_3<Traits, Visitor, Triangulation>

Definition

The class *Kinetic::Delaunay_triangulation_3<Traits, Visitor, Triangulation>* maintains a Delaunay triangulation on top of the points contained in a *Kinetic::ActiveObjectsTable*. It has one main method of interest. *triangulation()* which returns the triangulation it is maintaining. In addition, as an optimisation, you can turn on and off whether it is currently maintaining its certificates. This allows a large number of changes to the underlying points to be made at one time without recomputing the certificates each time a single point changes. This flag is false upon construction.

The class *Kinetic::Qt_triangulation_3<Traits>*, included as part of the demo code, displays a kinetic Delaunay triangulation in three dimensions using the Coin library.

The optional *Visitor* template argument is a model of *Kinetic::DelaunayTriangulationVisitor3* and can be used to monitor changes in the kinetic data structure.

The optional *Triangulation* template argument must be a model of a *CGAL::DelaunayTriangulation_3* which uses *Traits::Instantaneous_kernel* as its geometric traits and has *Kinetic::Delaunay_triangulation_cell_base_3<Traits, Base>* as the cell type.

```
#include <CGAL/Kinetic/Delaunay_triangulation_3.h>
```

Types

Kinetic::Delaunay_triangulation_3<Traits, Visitor, Triangulation>:: Triangulation

The template argument.

Kinetic::Delaunay_triangulation_3<Traits, Visitor, Triangulation>:: Visitor

The template argument.

Creation

```
Kinetic::Delaunay_triangulation_3<Traits, Visitor, Triangulation> dt( Traits tr);
```

Maintain the Delaunay triangulation of the points in *tr.active_points_3_handle()*.

Operations

```
const Triangulation* dt.triangulation() Access the triangulation that is maintained.
```

<i>bool</i>	<i>dt.has_certificates()</i>	This method returns true if the <i>Kinetic::Delaunay_triangulation_3<Traits, Visitor, Triangulation></i> is currently maintaining certificates for a Delaunay triangulation.
<i>void</i>	<i>dt.set_has_certificates(bool tf)</i>	This method allows you to control whether the triangulation is maintaining certificates.
<i>Visitor&</i>	<i>dt.visitor()</i>	Access the visitor.

See Also

Kinetic::Regular_triangulation_3<Traits, Triangulation, Visitor>, *Kinetic::Delaunay_triangulation_2<Traits, Triangulation, Visitor, Kinetic::Delaunay_triangulation_visitor_base_3, Kinetic::Delaunay_triangulation_event_log_visitor_3*.

CGAL::Kinetic::Delaunay_triangulation_cell_base_3<Traits, Base>

Definition

This is the base class for faces used by *Kinetic::Delaunay_triangulation_3<Traits, Triangulation, Visitor>::Triangulation*.

```
#include <CGAL/Kinetic/Delaunay_triangulation_cell_base_3.h>
```

CGAL::Kinetic::Delaunay_triangulation_event_log_visitor_2

Definition

The concept *Kinetic::Delaunay_triangulation_event_log_visitor_2* provides a model of *Kinetic::DelaunayTriangulationVisitor2* and *Kinetic::EventLogVisitor* which logs edge flip events.

Is Model for the Concepts

Kinetic::DelaunayTriangulationVisitor2, *Kinetic::EventLogVisitor*

See Also

Kinetic::Delaunay_triangulation_2<*Traits*, *Triangulation*, *Visitor*>

CGAL::Kinetic::Delaunay_triangulation_event_log_visitor_3

Definition

The concept *Kinetic::Delaunay_triangulation_event_log_visitor_3* provides a model of *Kinetic::DelaunayTriangulationVisitor3* and *Kinetic::EventLogVisitor* which logs edge and facet flip events.

Is Model for the Concepts

Kinetic::DelaunayTriangulationVisitor3, *Kinetic::EventLogVisitor*

See Also

Kinetic::Delaunay_triangulation_3<Traits, Triangulation, Visitor>

CGAL::Kinetic::Delaunay_triangulation_face_base_2<Traits, Base>

Definition

This is the base class for faces used by the triangulation used in *Kinetic::Delaunay_triangulation_2<Traits, Triangulation, Visitor>*.

```
#include <CGAL/Kinetic/Delaunay_triangulation_face_base_2.h>
```

See Also

Kinetic::Delaunay_triangulation_2<Traits, Triangulation, Visitor>

CGAL::Kinetic::Delaunay_triangulation_recent_edges_visitor_2<Triangulation>

Definition

The concept *Kinetic::Delaunay_triangulation_recent_edges_visitor_2<Triangulation>* provides a model of *Kinetic::DelaunayTriangulationVisitor2* which tracks which edges were created in the most recent change.

Is Model for the Concepts

Kinetic::DelaunayTriangulationVisitor2

Creation

Kinetic::Delaunay_triangulation_recent_edges_visitor_2<Triangulation> *a*;
 default constructor.

Kinetic::Delaunay_triangulation_recent_edges_visitor_2<Triangulation>::iterator
 The iterator through the recently created edges.

Operations

<i>iterator</i>	<i>a.begin()</i>	Begin iteration through the recent edges.
<i>iterator</i>	<i>a.end()</i>	End iteration through the recent edges.
<i>bool</i>	<i>a.contains(Triangulation::Edge)</i>	Returns true if this edge exists in the set.

See Also

Kinetic::Delaunay_triangulation_2<Traits, Triangulation, Visitor>

Kinetic::DelaunayTriangulationVisitor2

Definition

This concept is for proxy objects which get notified when a kinetic Delaunay triangulation changes.

Operations

void *v.remove_vertex(Vertex_handle)*

The vertex is about to be deleted.

void *v.create_vertex(Vertex_handle)*

The vertex was just created.

void *v.modify_vertex(Vertex_handle)*

The trajectory of the point at the vertex changed.

template <class It>

void *v.create_faces(It begin, It end)*

New faces have just been made. The *value_type* of the iterator is a *TriangulationDataStructure_2::Face_handle*.

template <class It>

void *v.remove_faces(It, It)*

The faces in the range are about to be deleted. The *value_type* of the iterator is a *TriangulationDataStructure_2::Face_handle*.

void *v.before_flip(Edge)*

The edge is about to be flipped.

void *v.after_flip(Edge)*

The edge was just created with a flip.

Has Models

Kinetic::Delaunay_triangulation_visitor_base_2, *Kinetic::Delaunay_triangulation_recent_edges_visitor_2<Triangulation>*, *Kinetic::Delaunay_triangulation_event_log_visitor_2*

Kinetic::DelaunayTriangulationVisitor3

Definition

This concept is for proxy objects which get notified when a kinetic Delaunay triangulation changes.

Operations

void *v.remove_vertex(Vertex_handle)*

The vertex is about to be deleted.

void *v.create_vertex(Vertex_handle)*

The vertex was just created.

void *v.modify_vertex(Vertex_handle)*

The trajectory of the point at the vertex changed.

template <class It>

void *v.create_cells(It begin, It end)*

New faces have just been made. The iterator *value_type* is a *TriangulationDataStructure_3::Cell_handle*.

template <class It>

void *v.remove_cells(It, It)* The faces in the range are about to be deleted. The *value_type* of the iterator is a *TriangulationDataStructure_3::Cell_handle*.

void *v.before_edge_flip(Edge)*

The edge is about to be flipped.

void *v.after_edge_flip(Facet)*

The facet was just created with a flip.

void *v.before_facet_flip(Facet)*

The facet is about to be flipped.

void *v.after_facet_flip(Edge)*

The edge was just created with a flip.

Has Models

Kinetic::Delaunay_triangulation_visitor_base_3, Kinetic::Delaunay_triangulation_recent_edges_visitor_3< Triangulation>, Kinetic::Delaunay_triangulation_event_log_visitor_3

CGAL::Kinetic::Delaunay_triangulation_visitor_base_2

Definition

The concept *Kinetic::Delaunay_triangulation_visitor_base_2* provides a model of *Kinetic::DelaunayTriangulationVisitor2*. You can extend this class if you only want to implement a few methods from *Kinetic::DelaunayTriangulationVisitor2*.

Is Model for the Concepts

Kinetic::DelaunayTriangulationVisitor2

Creation

Kinetic::Delaunay_triangulation_visitor_base_2 *a*;

default constructor.

See Also

Kinetic::Delaunay_triangulation_2<*Traits*, *Triangulation*, *Visitor*>

CGAL::Kinetic::Delaunay_triangulation_visitor_base_3

Definition

The concept *Kinetic::Delaunay_triangulation_visitor_base_3* provides a model of *Kinetic::DelaunayTriangulationVisitor3*. You can extend this class if you only want to implement a few methods from *Kinetic::DelaunayTriangulationVisitor3*.

Is Model for the Concepts

Kinetic::DelaunayTriangulationVisitor3

Creation

Kinetic::Delaunay_triangulation_visitor_base_3 *a*;

default constructor.

See Also

Kinetic::Delaunay_triangulation_3<*Traits*, *Triangulation*, *Visitor*>

CGAL::Kinetic::Enclosing_box_2<Traits>

Definition

The class *Kinetic::Enclosing_box_2<Traits>* keeps the points in the simulation inside of a box. Whenever the points come close to the wall of the box they bounce off of the wall.

Note that, in general, points hit the wall of the box at times which are not easily represented by standard (rational) number types. The resulting trajectories would also have non-rational coefficients, complicating and slowing the simulation. In order to handle this, the *Kinetic::Enclosing_box_2<Traits>* bounces the points at the nearest easily representable time before the point would leave the box.

```
#include <CGAL/Kinetic/Enclosing_box_2.h>
```

Types

Kinetic::Enclosing_box_2<Traits>::NT The number type used to represent the walls of the box and perform calculations. Generally this is *Traits::NT*.

Creation

```
Kinetic::Enclosing_box_2<Traits> eb( Traits, NT xmin, NT xmax, NT ymin, NT ymax);
```

This constructs a bounding box with the dimensions specified by the last 4 arguments. They are optional and will take the values ± 10 if omitted.

CGAL::Kinetic::Enclosing_box_3<Traits>

Definition

The class *Kinetic::Enclosing_box_3<Traits>* keeps the points in the simulation inside of a box. Whenever the points come close to the wall of the box they bounce off of the wall.

Note that, in general, points hit the wall of the box at times which are not easily represented by standard (rational) number types. The resulting trajectories would also have non-rational coefficients, complicating and slowing the simulation. In order to handle this, the *Kinetic::Enclosing_box_3<Traits>* bounces the points at the nearest easily representable time before the point would leave the box.

```
#include <CGAL/Kinetic/Enclosing_box_3.h>
```

Types

Kinetic::Enclosing_box_3<Traits>::NT The number type used to represent the walls of the box and perform calculations. Generally this is *Traits::NT*.

Creation

```
Kinetic::Enclosing_box_3<Traits> eb( Traits, NT xmin, NT xmax, NT ymin, NT ymax, NT zmin, NT zmax);
```

This constructs a bounding box with the dimensions specified by the last 6 arguments. They are optional and will take the values ± 10 if omitted.

CGAL::Kinetic::Erase_event<ActiveObjectsTable>

Definition

This event erases a point from the *ActiveObjectsTable* when it is processed.

```
#include <CGAL/Kinetic/Erase_event.h>
```

Is Model for the Concepts

Kinetic::Simulator::Event

Creation

```
Kinetic::Erase_event<ActiveObjectsTable> i( ActiveObjectsTable::Key k, ActiveObjectsTable::Handle t);
```

Erase the object *k* from the table *t* when processed.

See Also

Kinetic::ActiveObjectsTable, *Kinetic::Active_objects_vector<MovingObject>*.

Example

```
typedef CGAL::Kinetic::Exact_simulation_traits_2 Simulation_traits;
typedef Simulation_traits::Kinetic_kernel::Point_2 Moving_point_2;
typedef CGAL::Kinetic::Insert_event<Simulation_traits::Active_points_2_table> Insert_event;
typedef CGAL::Kinetic::Delaunay_triangulation_2<Simulation_traits> KDel;

Simulation_traits tr;

KDel kdel(tr);

Moving_point_2 mp(Moving_point_2::NT(0),
                  Moving_point_2::NT(0));
tr.simulator_handle()->new_event(Simulation_traits::Simulator::Time(3),
                                Erase_event(*tr.active_objects_table_handle()->keys_begin(),
                                             tr.active_points_2_table_handle()));
```

Kinetic::EventLogVisitor

Definition

This concept is for visitors which maintain a text log of events.

Types

Kinetic::EventLogVisitor:: Event_iterator An iterator through strings defining the events that occurred.
Each event is represented by a *std::string*.

Operations

Event_iterator *v.events_begin()* Begin iterating through the events.

Event_iterator *v.events_end()*

Has Models

Kinetic::Delaunay_triangulation_event_log_visitor_3, *Kinetic::Delaunay_triangulation_event_log_visitor_2*,
Kinetic::Regular_triangulation_event_log_visitor_3, *Kinetic::Sort_event_log_visitor*

CGAL::Kinetic::Insert_event<ActiveObjectsTable>

Definition

This event inserts a point into the *ActiveObjectsTable* when it is processed.

```
#include <CGAL/Kinetic/Insert_event.h>
```

Is Model for the Concepts

Kinetic::Simulator::Event

Creation

```
Kinetic::Insert_event<ActiveObjectsTable> i( ActiveObjectsTable::Data o, ActiveObjectsTable::Handle t);
```

Insert the object o, into the table t when processed.

See Also

Kinetic::ActiveObjectsTable, *Kinetic::Active_objects_vector<MovingObject>*.

Example

```
typedef CGAL::Kinetic::Exact_simulation_traits_2 Simulation_traits;
typedef Simulation_traits::Kinetic_kernel::Point_2 Moving_point_2;
typedef CGAL::Kinetic::Insert_event<Simulation_traits::Active_points_2_table> Insert_event;
typedef CGAL::Kinetic::Delaunay_triangulation_2<Simulation_traits> KDel;

Simulation_traits tr;

KDel kdel(tr);

Moving_point_2 mp(Moving_point_2::Coordinate(0),
                  Moving_point_2::Coordinate(0));
tr.simulator_handle()->new_event(Simulation_traits::Simulator::Time(3),
                                Insert_event(mp,
                                              tr.active_points_2_table_handle()));
```

CGAL::Kinetic::Qt_moving_points_2<Traits, QWidget_2>

Definition

The class *Kinetic::Qt_moving_points_2<Traits, QWidget_2>* displays a set of moving points in 2D.

See Section [42.3.3](#) for an example using this class.

```
#include <CGAL/Kinetic/IO/Qt_moving_points_2.h>
```

Creation

```
Kinetic::Qt_moving_points_2<Traits, QWidget_2> a( QtGui::Handle,  
                                                Traits::Active_points_2_table::Handle)
```

default constructor.

See Also

Kinetic::Qt_widget_2<Simulator>.

CGAL::Kinetic::Qt_triangulation_2<KineticTriangulation_2, QtWidget_2, QtMovingPoints_2>

Definition

The class draws a triangulation into a *CGAL::Qt_widget_2*. This class is very simple and a good one to look at if you want to see how to draw your own two dimensional kinetic data structure.

See Section [42.3.3](#) for an example using this class.

```
#include <CGAL/Kinetic/IO/Qt_triangulation_2.h>
```

Creation

```
Kinetic::Qt_triangulation_2<KineticTriangulation_2,      QtWidget_2,      QtMovingPoints_2>      a(
KineticTriangulation_2::Handle,
                                                                    QtWidget_
2::Handle,
                                                                    QtMovingPoints_
2::Handle)
```

Construct the object and make all the connections with the appropriate other objects.

See Also

Kinetic::Qt_widget_2<Simulator>

CGAL::Kinetic::Qt_widget_2<Simulator>

Definition

The class *Kinetic::Qt_widget_2<Simulator>* implements a graphical interface for 2D kinetic data structures.

```
#include <CGAL/Kinetic/IO/Qt_widget_2.h>
```

Types

<i>Kinetic::Qt_widget_2<Simulator>::Listener</i>	The listener base to listen for when to update the picture. This class includes an extra method <i>Qt_widget widget()</i> which returns the <i>Qt_widget</i> object which can be used for drawing.
--------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Creation

```
Kinetic::Qt_widget_2<Simulator> a( int argc, char *argv[], Simulator::Handle);
```

default constructor.

CGAL::Kinetic::Regular_triangulation_3<Traits, Visitor, Triangulation>

Definition

The class *Kinetic::Regular_triangulation_3<Traits, Visitor, Triangulation>* maintains a triangulation of set of moving weighted points. Its interface is the same as *Kinetic::Delaunay_triangulation_3<Traits, Visitor, Triangulation>*.

The optional *Triangulation* template argument must be a model of *CGAL::RegularTriangulation_3* which has *Kinetic::Regular_triangulation_cell_base_3<Traits, Base>* as a cell base and *Kinetic::Regular_triangulation_vertex_base_3<Traits, Base>* as a vertex base.

```
#include <CGAL/Kinetic/Regular_triangulation_3.h>
```

See Also

Kinetic::Delaunay_triangulation_3<Traits, Visitor, Triangulation>. *Kinetic::RegularTriangulationVisitor3*.

Example

```
#include <CGAL/Kinetic/Regular_triangulation_exact_simulation_traits_3.h>
#include <CGAL/Kinetic/Regular_triangulation_3.h>

int main(int, char *[]) {
    typedef CGAL::Kinetic::Regular_triangulation_exact_simulation_traits_3 Traits;
    typedef CGAL::Kinetic::Regular_triangulation_3<Traits> KDel;

    Traits tr;
    KDel kdel(tr);

    Traits::Simulator::Handle sp= tr.simulator_handle();

    std::ifstream in("data/weighted_points_3");
    in >> *tr.active_points_3_table_handle();

    std::cout << *tr.active_points_3_table_handle() << std::endl;

    kdel.set_has_certificates(true);

    sp->set_current_event_number(10000);
    return EXIT_SUCCESS;
};
```


CGAL::Kinetic::Regular_triangulation_cell_base_3<Traits, Base>

Definition

This is the base class for faces used by *Kinetic::Regular_triangulation_3<Traits, Triangulation, Visitor>*.

```
#include <CGAL/Kinetic/Regular_triangulation_cell_base_3.h>
```

CGAL::Kinetic::Regular_triangulation_event_log_visitor_3

Definition

The concept *Kinetic::Regular_triangulation_event_log_visitor_3* provides a model of *Kinetic::RegularTriangulationVisitor3* and *EventLogVisitor* which logs edge flip events.

Is Model for the Concepts

Kinetic::RegularTriangulationVisitor3, *Kinetic::EventLogVisitor*

See Also

Kinetic::Regular_triangulation_3<*Traits*, *Triangulation*, *Visitor*>

CGAL::Kinetic::Regular_triangulation_instantaneous_traits_3<ActiveObjectsTable, StaticKernel>

Definition

The class *Kinetic::Regular_triangulation_instantaneous_traits_3<ActiveObjectsTable, StaticKernel>* is an instantaneous kernel for use with a regular triangulation data structure. There is not currently a reason for the user to call this directly unless the user wants created their own simulation traits as it is included as part of the *Kinetic::Regular_triangulation_exact_simulation_traits_3*.

```
#include <CGAL/Kinetic/Regular_triangulation_instantaneous_traits_3.h>
```

Is Model for the Concepts

CGAL::RegularTriangulationTraits_3, Kinetic::InstantaneousKernel

See Also

Kinetic::Regular_triangulation_3<Traits, Visitor, Triangulation>.

CGAL::Kinetic::Regular_triangulation_vertex_base_3<Traits, Base>

Definition

This is the base class for the vertices of the triangulation class used by *Kinetic::Regular_triangulation_3<Traits, Triangulation, Visitor>*.

```
#include <CGAL/Kinetic/Regular_triangulation_vertex_base_3.h>
```

Kinetic::RegularTriangulationVisitor3

Definition

This concept is for proxy objects which get notified when a kinetic regular triangulation changes.

Operations

void *v.remove_vertex(Vertex_handle)*

The vertex is about to be deleted.

void *v.create_vertex(Vertex_handle)*

The vertex was just created.

void *v.modify_vertex(Vertex_handle)*

The trajectory of the vertex just changed.

template <class It>

void *v.create_cells(It begin, It end)*

New faces have just been made. The iterator *value_type* is a *TriangulationDataStructure_3::Cell_handle*.

template <class It>

void *v.remove_cells(It, It)* The faces in the range are about to be deleted. The *value_type* of the iterator is a *TriangulationDataStructure_3::Cell_handle*.

void *v.before_edge_flip(Edge)*

The edge is about to be flipped.

void *v.after_edge_flip(Facet)*

The facet was just created with a flip.

void *v.before_facet_flip(Facet)*

The facet is about to be flipped.

void *v.after_facet_flip(Edge)*

The edge was just created with a flip.

void *v.pre_move(Key, Cell)*

The point defined by *Key* is about to move from the cell.

void *v.post_move(Key, Cell)*

The point defined by *Key* just moved to the cell.

void *v.pre_push(Key, Cell)*

The point defined by the key is about to be inserted into the cell.

void *v.post_push(Vertex_handle)*

The point referred to by the vertex handle was just added to the triangulation, it previously was redundant.

void *v.pre_pop(Vertex_handle)*

The vertex is about to be removed from the triangulation since its weight is too small.

void *v.post_pop(Key, Cell)*

The point was just removed from the triangulation.

Has Models

Kinetic::Regular_triangulation_visitor_base_3, Kinetic::Regular_triangulation_event_log_visitor_3

CGAL::Kinetic::Regular_triangulation_visitor_base_3

Definition

The concept *Kinetic::Regular_triangulation_visitor_base_3* provides a model of *Kinetic::RegularTriangulationVisitor3*. You can extend this class if you only want to implement a few methods from *Kinetic::RegularTriangulationVisitor3*.

Is Model for the Concepts

Kinetic::RegularTriangulationVisitor3

Creation

Kinetic::Regular_triangulation_visitor_base_3 a;

default constructor.

See Also

Kinetic::Regular_triangulation_3<Traits, Triangulation, Visitor>

CGAL::Kinetic::Sort_event_log_visitor

Definition

The concept *Kinetic::Sort_event_log_visitor* provides a model of *SortVisitor* and *EventLogVisitor* which logs changes to the structure.

Is Model for the Concepts

Kinetic::SortVisitor, *Kinetic::EventLogVisitor*

See Also

Kinetic::Sort<*Traits*, *Visitor*>

CGAL::Kinetic::Sort<Traits, Visitor>

Definition

The class *Kinetic::Sort<Traits, Visitor>* maintains a sorted list of objects. It is the simplest kinetic data structure provided and is a good place to start when looking at the basics of implementing a kinetic data structure.

The *Kinetic::SortVisitor* can be used to monitor what is happening.

```
#include <CGAL/Kinetic/Sort.h>
```

Creation

Kinetic::Sort<Traits, Visitor> *s*(*Traits tr*); The basic constructor.

Types

Kinetic::Sort<Traits, Visitor>:: Visitor The type of the visitor.

Kinetic::Sort<Traits, Visitor>:: Traits The traits type.

Kinetic::Sort<Traits, Visitor>:: Vertex_handle The handle used to refer to vertex in the sorted list. Dereferencing this returns a *Key* into the *ActiveObjectsTable*.

Kinetic::Sort<Traits, Visitor>:: Handle A reference counted pointer to be used for storing references to the object.

Kinetic::Sort<Traits, Visitor>:: Const_handle A reference counted pointer to be used for storing references to the object.

Operations

Visitor& *s.visitor()* Access the visitor.

Traits& *s.traits()* Access the traits.

See Also

Kinetic::Ref_counted<T>

CGAL::Kinetic::Sort_visitor_base

Definition

The concept *Kinetic::Sort_visitor_base* provides a model of *Kinetic::SortVisitor*. You can extend this class if you only want to implement a few methods from *Kinetic::SortVisitor*.

Is Model for the Concepts

Kinetic::SortVisitor.

Creation

Kinetic::Sort_visitor_base *a*; default constructor.

See Also

Kinetic::Sort<*Traits*, *Visitor*>

Kinetic::SortVisitor

Definition

This concept is for proxy objects which have functions called on them when a *Kinetic::Sort<Traits, Visitor>*.

Operations

void *v.remove_vertex(Vertex_handle)*
The vertex is about to be deleted.

void *v.create_vertex(Vertex_handle)*
The vertex was just created.

void *v.modify_vertex(Vertex_handle)*
Something changed at the vertex.

void *v.before_swap(Vertex_handle, Vertex_handle)*
The pair of vertices is about to be exchanged.

void *v.after_swap(Vertex_handle, Vertex_handle)*
The pair of vertices was just swapped.

Has Models

Kinetic::Sort_visitor_base, Kinetic::Sort_event_log_visitor

Chapter 43

Kinetic Framework

Daniel Russel

Contents

43.1 Architecture	2461
43.1.1 The <code>Kinetic::Simulator</code>	2462
43.1.2 The <code>Kinetic::Kernel</code>	2463
43.1.3 The <code>Kinetic::ActiveObjectsTable</code>	2464
43.1.4 The <code>Kinetic::InstantaneousKernel</code>	2464
43.1.5 Miscellaneous: notification and reference management	2465
43.2 Algebraic Kernel	2465
43.2.1 <code>Kinetic::FunctionKernel</code> customized for kinetic data structures	2466
43.3 Examples	2466
43.3.1 Using the Pieces of the Package	2467
43.3.2 The trivial kinetic data structure	2468
43.3.3 Adding a new certificate type	2471

This chapter describes a framework for implementing kinetic data structures and sweepline algorithms. If you just would like to use existing kinetic data structures, please read Chapter 42 instead. Readers wishing to brush up on their familiarity with kinetic data structures or better understand the terminology we use should read Section 42.1 of that chapter. A brief overview of the framework can be found in Section 42.2 (also of that chapter) and it too is recommended reading. Here we dive right in to discussing the architecture of the framework in Section 43.1 and finally we give several examples of using the framework to implement a kinetic data structure in Section 43.3. The framework makes heavy use of our *Polynomial_kernel* package to provide models of the *Kinetic::FunctionKernel* concept.

The framework was first presented at ALENEX [GKR04].

43.1 Architecture

This package provides a framework to allow exact implementation of kinetic data structures and sweepline algorithms. Below we discuss in detail each one of the first four major concepts which help in implementing kinetic data structures: the *Kinetic::Simulator*, the *Kinetic::Kernel*, the *Kinetic::ActiveObjectsTable* and the *Kinetic::InstantaneousKernel*. The *Kinetic::FunctionKernel* concept is discussed separately in Section 43.2.

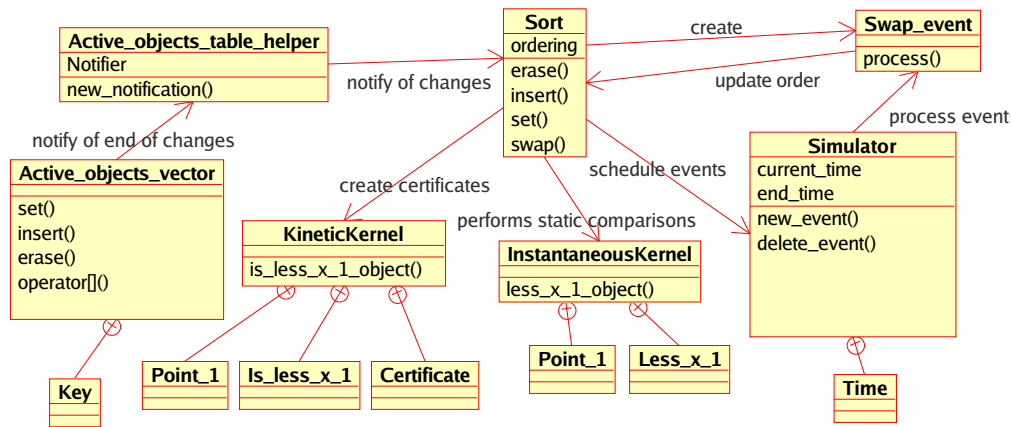


Figure 43.1: The figure, identical to the one in the overview of the previous chapter, shows the interaction between the *Kinetic::Sort*<Traits, Visitor> kinetic data structure and the various pieces of our framework. Other, more complicated, kinetic data structures will also use the *Kinetic::InstantaneousKernel* in order to insert/remove geometric primitives and audit themselves. *Kinetic::Sort*<Traits, Visitor> uses the sorting functionality in STL instead.

43.1.1 The Kinetic::Simulator

The *Kinetic::Simulator* is the central repository of all active events. It maintains the event queue and can use its knowledge of the events in the queue to find times for the kinetic data structures to easily check their own correctness (this will be discussed in more detail later in this section). Kinetic data structures call methods of the *Kinetic::Simulator* to schedule new events, deschedule old ones and access and change data contained in already scheduled events (the operations on existing events are performed using a key which was returned when the event was scheduled). For controlling the simulation, methods in the *Kinetic::Simulator* allow stepping through events, advancing time and even running the simulation backwards (that is we run the simulation with the time running in the opposite direction).

The kinetic sorting example in Figure 42.3.1 shows the basic usage of the *Kinetic::Simulator*. First, the *Simulator* is created by the *Kinetic::SimulationTraits*. The kinetic data structure gets a handle to the simulator from the traits class and uses the handle to add its events to the simulation. The *Kinetic::Simulator* is then told to advance time up until the end of the simulation, processing all events along the way.

Each event is represented by a *Kinetic::Simulator::Time* and an instance of a model of the *Kinetic::Simulator::Event* concept. Models of the *Kinetic::Simulator::Event* concept are responsible for taking the appropriate action in order to handle the kinetic event they represent. Specifically, the *Kinetic::Simulator::Event* concept specifies one method, *Kinetic::Simulator::Event::process()*, that is called when the event occurs. The body of the *Kinetic::Simulator::Event::process()* method typically simply calls a method of the kinetic data structure that created the event; for example in our kinetic sorting example, processing an event means calling the *Kinetic::Sort*<Traits, Visitor>::*swap*(Iterator) method of the kinetic sorting data structure.

In the model of the *Kinetic::Simulator* concept that we provide, *Kinetic::Default_simulator*<FunctionKernel, EventQueue>, any model of the *Kinetic::Simulator::Event* concept can be inserted as an event. This ability implies that events can be mixed at run time, which is essential when we want to support multiple kinetic data structures operating on the same set of moving geometric primitives.

The *Kinetic::Simulator::Time* concept is defined by the simulator, typically to be some representation of a root

of a polynomial, taken from the *Kinetic::FunctionKernel* (details of the algebraic side of the package will be discussed in Section 43.2). For most kinetic data structures *Kinetic::Simulator::Time* only needs to support comparisons (we need to compare events, in order to process them in the correct order) and a few other non-arithmetic operations.

When the failure times of certificates are sorted exactly (as opposed to when we numerically approximate the roots of the certificate polynomials) the correctness of kinetic data structures can be easily verified. Let I be an open interval between the last event processed and the next event to be processed. As was mentioned in the introduction kinetic data structures do not change combinatorially in I . In addition, although the static data structures can be degenerate at the roots defining the two ends of the interval, they are not, in general, degenerate in the interior. An independent check of the integrity of kinetic data structures can be provided by, for example, using an *Kinetic::InstantaneousKernel* (cf. Subsection 43.1.4) to rebuild the static version of the structure from scratch at some time interior to I and compare it to the kinetic version. This auditing can typically catch algorithmic or programming errors much closer to the time they arise in the simulation than, for example, using visual inspection. Such easy auditing is one of the powerful advantages of having an exact computational framework since, as with static data structures, when using inexact computations differentiating between errors of implementation and numeric errors is quite tricky.

Kinetic data structures receive alerts of appropriate times to audit themselves using a notification framework. The same framework is also used by the *Kinetic::ActiveObjectsTable* to alert kinetic data structures when the set of primitives changes (see Subsection 43.1.3). To use the notification framework, the kinetic data structure creates a proxy object which implements a standard *Listener* interface. It then registers this proxy with the *Kinetic::Simulator*. When the *Kinetic::Simulator* finds an appropriate time for the kinetic data structures to audit themselves it calls the function *Listener::new_notification(Type)* on each of the registered proxy objects. A helper for creating such proxy objects, called *Kinetic::Simulator_kds_listener<Listener, KDS>*, is provided by the framework. It translates the notification into a function call (*audit()*) on the kinetic data structure. Pointers in the notification framework are reference counted appropriately to avoid issues caused by the creation and destruction order of kinetic data structures and the simulator. See Section 43.1.5 for a more complete discussion of this part of the framework.

Internally the *Kinetic::Simulator* maintains a priority queue containing the scheduled events. The type of the priority queue is a template argument to our *Kinetic::Simulator* model and, as such, it can be replaced by the user. In our package, we provide two different types of priority queues, a heap and a two-list priority queue. A two-list queue is a queue in which there is a sorted front list, containing all events before some time and an unsorted back list. The queue tries to maintain a small number of elements in the front list, leaving most of them in the unsorted main pool. The two-list queue, although an unconventional choice, is our default queue when using exact computation because it minimizes comparisons involving events that are far in the future. These events are likely to be deleted before they are processed, so extra work done structuring them is wasted. Our experiments have shown that, for example, the two-list queue causes a 20% reduction in running time relative to a binary heap for Delaunay triangulations with degree 3 polynomial motions and 20 points.

43.1.2 The *Kinetic::Kernel*

The *Kinetic::Kernel* is structured very much like static CGAL kernels. It defines a number of primitives, which in the model provided are *Kinetic::Kernel::Point_1*, *Kinetic::Kernel::Point_2*, *Kinetic::Kernel::Point_3* and *Kinetic::Kernel::Weighted_point_3*. The primitives are defined by a set of Cartesian coordinates each of which is a function of time, a *Kernel::MotionFunction*. In addition it defines constructions and certificate generators which act on the primitives. The certificate generators are the direct analog of the non-kinetic predicates. Each certificate generator take a number of primitives as arguments, but instead of producing an element from a discrete set they produce a set of discrete failure times for the certificate. These failure times are wrapped in a model of *Kinetic::Certificate*.

A *Kinetic::Certificate* is a simple object whose primary function is to produce a *Kinetic::Simulator::Time* object

representing the failure time of the certificate. Since, the handling of most certificate failures involves creating a new certificate whose certificate function is the negation of the old certificate function, a *Kinetic::Certificate* object caches any work that could be useful to isolate future roots of the certificate function (such as the Sturm sequence of the certificate function). To illustrate this further, if you have two one-dimensional points with coordinate functions $p_0(t)$ and $p_1(t)$, the certificate that the first moving point is after the second corresponds to the inequality $p_0(t) - p_1(t) > 0$. When the certificate fails and the two points cross, the new certificate is $p_1(t) - p_0(t) > 0$, which is the negated version of the certificate just processed and which has the same roots.

The model of *Kinetic::Kernel* provided includes the certificate generators necessary for Delaunay triangulations (in one, two and three dimensions) and regular triangulations (in 3D). New certificates can be fairly easily added. An example is included in the distributed code.

43.1.3 The *Kinetic::ActiveObjectsTable*

The *Kinetic::ActiveObjectsTable* stores a set of kinetic primitives. Its purpose is to notify kinetic data structures when new primitives are added, when primitives are removed or when a trajectories change. Each primitive is uniquely identified by a Key, assigned by the table when the primitive is added, that can be used to change or remove it. We provide one model of the *Kinetic::ActiveObjectsTable* concept, called *Kinetic::Active_objects_vector<MovingObject>* which stores all the moving primitives in an *std::vector<D>*.

Notifications of changes to the set of active objects are handled using a setup similar to the *Kinetic::Simulator* audit time notification. We provide a helper class, *Kinetic::Active_objects_listener_helper<ActiveObjectsTable, KDS>*, which translates the notifications into *insert(Key)*, *erase(Key)* or *set(Key)* function calls on the kinetic data structure.

43.1.4 The *Kinetic::InstantaneousKernel*

The *Kinetic::InstantaneousKernel* allows existing CGAL data structures to be used on moving data as it appears at some instant of time. Models of this concept are, by definition, models of a CGAL Kernel or a traits class, and, therefore, can then be used as the traits class of CGAL's algorithms and data structures.

Consider for example the kinetic Delaunay data structure in either two or three dimensions. Internally, it uses a *Delaunay_triangulation_2<Traits, Tds>* or *Delaunay_triangulation_3<Traits, Tds>* to represent the triangulation, instantiated with a model of the *Kinetic::InstantaneousKernel* concept as its traits class. At initialization, as well as at times during the simulation when we want to insert a point to the kinetic Delaunay triangulation, a static version of the Delaunay triangulation is conceptually instantiated. More precisely, the time for the copy of the model of the *Kinetic::InstantaneousKernel* stored in the CGAL triangulation is set to be the current time (or rather, as discussed in the introduction, a more convenient time determined by the *Kinetic::Simulator* combinatorially equivalent to the current time). The kinetic data structure then calls the *Delaunay_triangulation_3<Traits, Tds>::insert(Point)* insert method to insert the point. The static insert method called uses various predicate functors on the moving points which evaluate to the values that the predicates have at that instant in time. Removal is handled in an analogous manner. Auditing of the geometric structure is easily handled in a similar manner (in the case of Delaunay triangulations by simply calling the *verify()* method after setting the time).

Internally, in order to evaluate a static predicate, our model of the *Kinetic::InstantaneousKernel* computes the current instantaneous coordinates of the involved primitives and then passes them to a static predicate. The static primitive coordinates are cached since they are likely to be involved in other static predicates evaluated at the same time.

Note that the time used in the *Kinetic::InstantaneousKernel* must be an arithmetic type that fully supports ring and possibly field operations. These are much more stringent requirements than for *Kinetic::Simulator::Time*

objects in the simulator, where we only require comparisons. These requirements currently rule out our algebraic kernels' *Root* objects (see Section 43.2), since they only support comparisons and some limited arithmetic operations. In practice this is not a strong limitation since we can almost always find a time that is representable by a rational number type (such as *Gmpq*) which is close to the current time and such that the combinatorial structure has not changed. Alternatively, if a real number type such as *CORE::Expr* is used for time, then computation can, of course, be performed at any time, although computations will be significantly slower. It is the job of the *Kinetic::Simulator* to determine an appropriate time to perform operations.

43.1.5 Miscellaneous: notification and reference management

We describe some coding conventions used, graphical display, notification and reference management support in the framework in the following sections.

Reference management

A number of objects need to maintain pointers to other independent objects. For example, each kinetic data structure must have access to the *Kinetic::Simulator* so that it can schedule and deschedule events. These pointers are all reference counted in order to guarantee that they are always valid. We provide a standard reference counting pointer and object base to facilitate this, namely *Ref_counted<Object>*.

Each shared object in the framework defines a type *Handle* which is the type for a reference counter pointer pointing to it. These should be used for storing pointers to the objects in order to avoid dangling pointers. In addition, many of the objects expect such pointers as arguments.

Runtime event passing

Runtime events must be passed from *notifiers*, namely the *Kinetic::ActiveObjectsTable* and the *Kinetic::Simulator* to *listeners*, typically the kinetic data structures. For example, kinetic data structures are notified when new primitives are added to the *Kinetic::ActiveObjectsTable*. On receiving the notification, it will add the new primitive to the combinatorial structure it is maintaining. The events are passed using a simple, standardized notification interface. To receive notifications, the listener first defines a small proxy class which inherits from a *Listener* base type provided by the notifier. On creation, the *Listener* base class registers itself with the notifier on construction (and unregisters itself on destruction).

When the some state of the notifier changes, it calls the *new_notification* method on the listener proxy object provided and passes it a label corresponding to the name of the field that changed. The proxy object can then call an appropriate method on the kinetic data structure or whatever the listening class is.

In order to unregister on destruction, the *Listener* must store a (reference counted) pointer to the object providing notifications. This pointer can be accessed through the *notifier()* field. The *Listener* object stores a reference counted pointer to the notifying object, while the notifying object stores a plain pointer to the *Listener*. It can do this since the *Listener* is guaranteed to unregister itself when it is destroyed. This avoids circular reference counted pointers as well as dangling pointers.

43.2 Algebraic Kernel

The interface between the algebraic kernel and the kinetic data structures package was kept quite minimal in order to ease the implementation of various underlying computation models. The interface is detailed in the

reference page (*Kinetic::FunctionKernel*).

We provide models of the algebraic kernel that handle polynomial *Kinetic::Function* objects. The provided models perform

- exact computations using Sturm sequences to isolate roots
- exact computations using Descartes rule of sign in order to isolate roots (Sturm sequences are also used in order to properly handle even multiplicity roots)
- filtered exact computations using Descartes rule of sign
- numeric (inexact) root approximations
- numeric root approximations which take advantage of certain assumptions that can be made about the types of polynomials solved in the process of evaluating kinetic data structures
- a wrapper for *CORE::Expr* which implements the required concepts.

The exact models, which we implement the numerics for, handle non-square-free polynomials and polynomials with arbitrary field number type coefficients and are quite robust.

43.2.1 *Kinetic::FunctionKernel* customized for kinetic data structures

There are several modifications we can make to how the roots are handled to optimize for the case of kinetic data structures. The first are motivated by the question of how to handle degeneracies (certificate functions which have roots at the same time). Naively, there is no way to differentiate between a certificate which fails immediately when it is created and one whose function is momentarily 0, but will be valid immediately in the future. In order to handle such degeneracies we ensure that all the certificate function generators produce certificate functions which are positive when the corresponding certificates are valid. Then, if we have a degeneracy we can differentiate between a certificate which fails immediately and one which is simply degenerate by looking at the sign of the certificate function immediately following the root (equivalently, by looking at the derivative). In addition, this allows us, under the assumption that computations are performed exactly, to check that all certificates are not invalid upon creation.

The assumption that certificates are positive when valid is particular useful when using numeric solvers. Without it there is no reliable way to tell whether a root near the current time is the certificate having become valid just before the current time, or failing shortly in the future. Testing the sign of the function immediately after the root reliably disambiguates the two cases.

In addition, we have to specially handle even roots of functions. For the most part these can just be discarded as dropping an even root is equivalent to perturbing the simulation to remove the degeneracy. However, when we are using the *Kinetic::Simulator* to audit the kinetic data structures, they must be broken up in to two, equal, roots to avoid auditing at the degeneracy.

43.3 Examples

We provide a number of examples of different levels of usage of the kinetic data structures framework, both for kinetic data structures as well as sweepline algorithms.

To see how to use existing kinetic data structures, look at the examples in the previous chapter such as Section [42.3.1](#).

Here we cover implementing kinetic data structures. The examples explained are

- A trivial kinetic data structure which has all the parts of a full kinetic data structure but doesn't do much in Section 43.3.2.
- Adding a new type of certificate to a kernel in Section 43.3.3.

In order to see more detail about how to implement a kinetic data structure, the best place to start is the source code for the kinetic sorting data structure, *Kinetic::Sort<Traits, Visitor>*. Once you are familiar with that, *Kinetic::Delaunay_2<Traits, Triangulation, Visitor>* is the next step in complexity.

We will first explain in detail how a typical kinetic data structure uses the various pieces of the framework, then move on to showing the actual code for a simpler data structure.

43.3.1 Using the Pieces of the Package

Here we will explain how the kinetic sorting data structure uses the various pieces of the package. A schematic of its relationship to the various components is shown in the UML diagram in Figure 43.1. In this subsection we abuse, for reasons of simplicity of presentation, the concept/model semantics: when we refer to concepts we actually refer to an instance of a model of them.

As with most kinetic data structures, *Kinetic::Sort<Traits, Visitor>* maintains some sort of combinatorial structure (in this case a sorted doubly linked list), each element of which has a corresponding certificate in the event queue maintained by the simulator. In the case of sorting, there is one certificate maintained for each “edge” between two consecutive elements in the list.

On creation, the data structure is passed a copy of the *Kinetic::SimulationTraits* for this simulation, which it saves for future use. It gets a handle to the *Kinetic::ActiveObjectsTable* by calling the *Kinetic::SimulationTraits::active_points_1_table_handle()* method and registers a proxy with the table in order to receive notifications of changes to the point set. The *Kinetic::SimulationTraits* method returns a handle to, rather than a copy of, the *Kinetic::ActiveObjectsTable*, since the table must be shared between all the kinetic data structures using these points. The handles are reference counted pointers, thus saving the user from worrying about cleaning things up properly.

When new points are added to the model of the *Kinetic::ActiveObjectsTable*, the table calls the *new_notification()* method on the proxy of the kinetic data structure, which in turn calls the *insert(Point_key)* method of the kinetic data structure. The *Point_key* here is the key which uniquely identifies the newly inserted point in the table. The data structure then requests an instance of a model of the *Kinetic::InstantaneousKernel* from the *Kinetic::SimulationTraits*. It sets the time on the instantaneous kernel to the time value gotten from the *Kinetic::Simulator::current_time_nt()* method. This method returns a field number type that is between the previous and next event, as discussed in the introduction. An instance of the *Kinetic::InstantaneousKernel::Less_x_1* predicate and the STL function *std::upper_bound()* are then used to insert the new point in the sorted list. For each inserted object, the kinetic data structure removes the no longer relevant certificate from the event queue by calling the *Kinetic::Simulator::delete_event(Key)* function and creates two new certificates using a *Kinetic::Kernel::Is_less_x_1* certificate functor. The new certificates are inserted in the event queue by calling the *Kinetic::Simulator::new_event(Time, Event)* method where *Kinetic::Simulator::Event* is a proxy object which instructs the sort kinetic data structure to swap two points when its *process()* method is called.

Now that the kinetic data structure has been initialized, the simulator is instructed to process all events. Each time an event occurs, the simulator calls the *process()* method on the corresponding proxy object. The proxy, in turn, tells the sort kinetic data structure to swap the two points whose order has changed.

The *Kinetic::Simulator* can periodically instruct the kinetic data structures to audit themselves. As is explained in Section 43.1.1, a proxy object maps the notification on to an *audit()* function call in the kinetic data structure. To audit itself the kinetic data structure builds a list of all the current points and uses *std::sort* to sort this list using a comparison function gotten from the *Kinetic::InstantaneousKernel*. This sorted list is compared to the maintained one to verify correctness. This auditing could also have been done by evaluating the *Kinetic::InstantaneousKernel* predicate for each sorted pair. Since auditing a kinetic data structure typically requires at least linear time in the size of the combinatorial structure, the auditing procedure in between events is deactivated by default. The user can however easily switch it on by defining the *CGAL_CHECK_EXACTNESS* and *CGAL_CHECK_EXPENSIVE* CGAL macros.

This general structure of the interaction between the kinetic data structure and the framework is shared by all of the provided kinetic data structures and has proved itself to go quite far.

43.3.2 The trivial kinetic data structure

To show how to implement such things, instead of presenting a full kinetic data structure, we present a trivial one which maintains one event in the queue which maintains one event in the queue, scheduled to occur one time unit after the last change was made to the set of active primitives. Two classes are defined, the *Trivial_event*, and the *Trivial_kds*. The event classes must be declared outside of the kinetic data structure so that the *operator<<* can be defined for them.

The kinetic data structure maintains the invariant that it was one event in the queue at all times. This event occurs one time unit after the last event or change in the set of objects occurs. As a result, the kinetic data structure has the main parts of a real one—it responds to changes in trajectories of the objects and certificate failures (when the event expires).

The public methods can be grouped into three sets which are shared with almost all other kinetic data structures:

- *has_certificates* and *set_has_certificates* which checks/sets whether the kinetic data structure is currently maintaining certificates.
- *insert*, *set*, *erase* which are called by the *Kinetic::Active_objects_listener_helper* in response to the addition, modification, or deletion of an object to, in or from the simulation.
- *audit* which is called periodically by the *Kinetic::Simulator_kds_listener* when kinetic data structures can easily audit themselves.

In addition, it has one method which is called when a certificate fails. The name/existence of such methods depend on the nature of the kinetic data structure in question.

Like many kinetic data structures, it takes a *Kinetic::SimulationTraits* as a template argument. This traits class defines the types needed for the simulation and is responsible for instantiating them.

```
#include <CGAL/Kinetic/Ref_counted.h>
#include <CGAL/Kinetic/Exact_simulation_traits_1.h>
#include <CGAL/Kinetic/Active_objects_listener_helper.h>
#include <CGAL/Kinetic/Simulator_kds_listener.h>
...

// This must be external since operator<< has to be defined
template <class KDS>
struct Trivial_event
```

```

{
    Trivial_event(){}
    Trivial_event(KDS* kds): kds_(kds) {
    }
    void process() const
    {
        kds_->process();
    }
    KDS* kds_;
};

template <class KDS>
std::ostream &operator<<(std::ostream &out,
    const Trivial_event<KDS> &) {
    out << "\"An event\"";
    return out;
}

template <class Traits>
struct Trivial_kds: CGAL::Kinetic::Ref_counted<Trivial_kds<Traits> >
{
    typedef Trivial_kds<Traits> This;
    typedef typename Traits::Active_points_1_table::Data Point;
    typedef typename Traits::Simulator::Time Time;
    typedef typename Traits::Active_objects_table::Key Point_key;
    typedef typename Traits::Simulator::Event_key Event_key;
    typedef CGAL::Kinetic::Active_objects_listener_helper<
        typename Traits::Active_points_1_table::Listener, This> Active_objects_helper;
    typedef CGAL::Kinetic::Simulator_kds_listener<
        typename Traits::Simulator::Listener, This> Simulator_helper;

    typedef Trivial_event<This> Event;

    Trivial_kds(Traits tr): has_certificates_(true),
        tr_(tr),
        nth_(tr.active_points_1_table_handle(), this),
        sh_(tr.simulator_handle(), this){}

    // this method is called with the value true when the event is processed
    void process(bool tf) {
        event_ = Event_key();
        set_has_certificates(false);
        set_has_certificates(true);
    }

    void audit() const
    {
        ...
    }

    void set_has_certificates(bool tf) {
        typename Traits::Simulator::Handle sp = tr_.simulator_handle();
        if (has_certificates_ != tf) {

```

```

        has_certificates_=tf;
        if (has_certificates_) {
bool ev= event_;
CGAL_assertion(!ev);
Time t= CGAL::to_interval(sp->current_time()).second+1;
event_= sp->new_event(t, Event(this));
        } else if (event_) {
sp->delete_event(event_);
event_=Event_key();
        }
    }
}

bool has_certificates() const {
    return has_certificates_;
}

void insert(Point_key k) {
    if (has_certificates_) {
        set_has_certificates(false);
        set_has_certificates(true);
    }
}

void set(Point_key k) {
    if (has_certificates_) {
        set_has_certificates(false);
        set_has_certificates(true);
    }
}

void erase(Point_key k) {
    if (has_certificates_) {
        set_has_certificates(false);
        set_has_certificates(true);
    }
}

~Trivial_kds(){
    set_has_certificates(false);
}

protected:
    bool has_certificates_;
    Event_key event_;
    Traits tr_;
    Active_objects_helper nth_;
    Simulator_helper sh_;
};

```

43.3.3 Adding a new certificate type

The following example shows how to add a new type of certificate to a simulation.

First we code the actual certificate function generator. It must take some sort (or sorts) of kinetic primitives, compute some function from their coordinates.

```
template <class KineticKernel>
struct Positive_x_f_2 {
    typedef typename KineticKernel::Certificate_function result_type;
    typedef typename KineticKernel::Point_2 argument_type;
    result_type operator()(const argument_type &p){
        return result_type(p.x()- result_type(0));
    }
};
```

Then we define a kinetic kernel which includes this predicate. To do this we wrap the function generator in a *Kinetic::Certificate_generator<Kernel, Generator>*. This wrapper uses the generator to create the certificate function and then the *Kinetic::FunctionKernel* to solve the certificate function. The result is wrapped in a *Kinetic::Certificate* object.

```
template <class FunctionKernel>
class My_kinetic_kernel:
    public CGAL::Kinetic::Cartesian_kinetic_kernel<FunctionKernel> {
    typedef CGAL::Kinetic::Cartesian_kinetic_kernel<FunctionKernel> P;
    typedef My_kinetic_kernel<FunctionKernel> This;
public:
    typedef CGAL::Kinetic::internal::Certificate_generator<This, Positive_x_f_2<This> > Positive_x_2;
    Positive_x_2 positive_x_2_object() const
    {
        return Positive_x_2(P::function_kernel_object());
    }
};
```

Now we have the unfortunately rather messy part of assembling a new *Kinetic::SimulationTraits* model. This is done in two steps for convenience.

```
struct My_st_types: public CGAL::Kinetic::Suggested_exact_simulation_traits_types {
    typedef CGAL::Kinetic::Suggested_exact_simulation_traits_types P;
    typedef My_kinetic_kernel<P::Function_kernel>::Point_2 Active_object;
    typedef CGAL::Kinetic::Active_objects_vector<Active_object> Active_objects_table;
    typedef CGAL::Kinetic::Cartesian_instantaneous_kernel< Active_objects_table,
        Static_kernel> Instantaneous_kernel;
};

struct My_simulation_traits:
    public CGAL::Kinetic::Simulation_traits<My_st_types::Static_kernel,
        My_st_types::Kinetic_kernel,
        My_st_types::Simulator>
{
    typedef CGAL::Kinetic::Simulation_traits<My_st_types::Static_kernel,
```

```

    My_st_types::Kinetic_kernel,
    My_st_types::Simulator> P;
My_simulation_traits(const P::Time &lb= P::Time(0),
    const P::Time &ub=std::numeric_limits<P::Time>::infinity()):
    P(lb,ub),
    ap_(new Active_points_2_table()) {}

typedef My_st_types::Active_objects_table Active_points_2_table;
Active_points_2_table* active_points_2_table_handle() {
    return ap_.get();
}
const Active_points_2_table* active_points_2_table_handle() const {
    return ap_.get();
}

typedef My_st_types::Instantaneous_kernel Instantaneous_kernel;
Instantaneous_kernel instantaneous_kernel_object() const
{
    return Instantaneous_kernel(ap_, static_kernel_object());
}
protected:
    Active_points_2_table::Handle ap_;
};

```

Now the simulation traits can be used by a kinetic data structure.

Kinetic Framework Reference Manual

Daniel Russel

43.4 Classified Reference Pages

Definition

Kinetic data structures are a way of adding motion to classical geometric data structures. CGAL provides a number of classes to aid implementation of kinetic data structures.

There are three levels at which the user can interact with the package. The user can use an existing kinetic data structure, write a new kinetic data structure, or replace parts of the framework. The first level is covered in the Chapter [43](#).

Main Support Classes and Concepts

Here we list the main classes and concepts provided by the framework to support implementing kinetic data structures

<code>Kinetic::ActiveObjectsTable</code>	page 2477
<code>Kinetic::Active_objects_vector<MovingObject></code>	page ??
<code>Kinetic::Cartesian_instantaneous_kernel<ActiveObjectsTable, StaticKernel></code>	page ??
<code>Kinetic::Cartesian_kinetic_kernel<FunctionKernel></code>	page ??
<code>Kinetic::FunctionKernel</code>	page 2488
<code>Kinetic::InstantaneousKernel</code>	page 2492
<code>Kinetic::Kernel</code>	page 2494
<code>Kinetic::SimulationTraits</code>	page 2501
<code>Kinetic::Simulator</code>	page 2506
<code>Kinetic::Default_simulator<FunctionKernel, EventQueue></code>	page ??

Other Concepts

Key	page 2493
Kinetic::Certificate	page 2483
Kinetic::EventQueue	page 2485
Kinetic::FunctionKernel::ConstructFunction	page 2484
Key	page 2493
Kinetic::FunctionKernel::Function	page 2491
Kinetic::RootStack	page 2500
Kinetic::Simulator::Event	page 2487
Kinetic::Simulator::Time	page 2510

Other Classes

<i>CGAL::Listener<Interface></i>	page 2495
<i>CGAL::Multi_listener<Interface></i>	page 2498
<i>CGAL::Ref_counted<T></i>	page 2499
<i>Kinetic::Active_objects_listener_helper<ActiveObjectsTable, KDS></i>	page ??
<i>Kinetic::Erase_event<ActiveObjectsTable></i>	page ??
<i>Kinetic::Insert_event<ActiveObjectsTable></i>	page ??
<i>Kinetic::Qt_moving_points_2<Traits, QtWidget_2></i>	page ??
<i>Kinetic::Qt_triangulation_2<KineticTriangulation_2, QtWidget_2, QtMovingPoints_2></i>	page ??
<i>Kinetic::Qt_widget_2<Simulator></i>	page ??
<i>Kinetic::Regular_triangulation_instantaneous_traits_3<ActiveObjectsTable, StaticKernel></i>	page ??
<i>Kinetic::Simulator_kds_listener<Listener, KDS></i>	page ??
<i>Kinetic::Simulator_objects_listener<Simulator_listener, KDS></i>	page ??

43.5 Alphabetical List of Reference Pages

<i>ConstructFunction</i>	page 2484
<i>Event</i>	page 2487
<i>Function</i>	page 2491
<i>Key</i>	page 2493
<i>Kinetic::ActiveObjectsTable</i>	page 2477

<i>Kinetic::Active_objects_listener_helper<ActiveObjectsTable, KDS></i>	page 2476
<i>Kinetic::Active_objects_vector<MovingObject></i>	page 2479
<i>Kinetic::Cartesian_instantaneous_kernel<ActiveObjectsTable, StaticKernel></i>	page 2480
<i>Kinetic::Cartesian_kinetic_kernel<FunctionKernel></i>	page 2481
<i>Kinetic::Certificate</i>	page 2483
<i>Kinetic::Default_simulator<FunctionKernel, EventQueue></i>	page 2503
<i>Kinetic::EventQueue</i>	page 2485
<i>Kinetic::FunctionKernel</i>	page 2488
<i>Kinetic::InstantaneousKernel</i>	page 2492
<i>Kinetic::Kernel</i>	page 2494
<i>Kinetic::RootStack</i>	page 2500
<i>Kinetic::SimulationTraits</i>	page 2501
<i>Kinetic::Simulator_kds_listener<Listener, KDS></i>	page 2504
<i>Kinetic::Simulator_objects_listener<Simulator_listener, KDS></i>	page 2505
<i>Kinetic::Simulator</i>	page 2506
<i>Listener<Interface></i>	page 2495
<i>Multi_listener<Interface></i>	page 2498
<i>Ref_counted<T></i>	page 2499
<i>Time</i>	page 2510

CGAL::Kinetic::Active_objects_listener_helper<ActiveObjectsTable, KDS>

Definition

The class *Kinetic::Active_objects_listener_helper*<*ActiveObjectsTable*, *KDS*> acts as an intermediate between a moving object table and a KDS. It translates the *ActiveObjectsTable::Listener::IS_EDITING* notification events into appropriate calls to *Kinetic::insert*, *Kinetic::set*, *Kinetic::erase*.

```
#include <CGAL/Kinetic/Active_objects_listener_helper.h>
```

See Also

Kinetic::Active_objects_vector<*MovingObject*>, *Kinetic::ActiveObjectsTable*.

Kinetic::ActiveObjectsTable

Definition

This container holds a set of objects of a particular type. It creates notifications using the standard *Multi_listener<Interface>* interface when a primitive changes or is added or deleted. Objects which are listening for events can then ask which primitives changed.

For speed, modifications to the *Kinetic::ActiveObjectsTable* can be grouped into editing sessions. A session is begun by calling *set_is_editing(true)* and ended by calling *set_is_editing(false)*. There is one type of notification, namely, *Listener::IS_EDITING* which occurs when the editing mode is set to false, signaling that a batch of changes is completed.

As an convenience, the change methods can be called without setting the editing state to true, this acts as if it were set to true for that one function call.

Types

<i>Kinetic::ActiveObjectsTable:: Key</i>	A key identifying an object in the table.
<i>Kinetic::ActiveObjectsTable:: Data</i>	The type being stored in the table.
<i>Kinetic::ActiveObjectsTable:: Listener</i>	The base class to derive from for listening for runtime events.

The following types are iterators. Each type, *Foo_iterator* has two corresponding methods *foo_begin* and *foo_end* which allow you to iterate through the objects in the set *Foo*.

<i>Kinetic::ActiveObjectsTable:: Key_iterator</i>	An iterator through all the valid keys in the table.
<i>Kinetic::ActiveObjectsTable:: Changed_iterator</i>	An iterator through all the objects which have been changed in the last editing session.
<i>Kinetic::ActiveObjectsTable:: Inserted_iterator</i>	An iterator through all the objects which were added in the last editing session.
<i>Kinetic::ActiveObjectsTable:: Erased_iterator</i>	An iterator through all the objects which were deleted in the last editing session.

Creation

Operations

<i>Data</i>	<i>mot[Key key]</i>	Access the object referenced by the key.
<i>Data</i>	<i>mot.at(Key key)</i>	Access the object referenced by the key.
<i>void</i>	<i>mot.set_is_editing(bool is_editing)</i>	Set the editing state of the object. A notification is sent when the editing state is set to false after it has been true, i.e. the editing session is finished. This allows changes to be batched together.
<i>bool</i>	<i>mot.is_editing()</i>	Access the editing state.
<i>void</i>	<i>mot.set(Key key, Data object)</i>	This method changes the motion of one moving object. The position at the current time should not be different from the previous current position. However, at the moment I do not check this as there is no reference to time in the <code>Kinetic::ActiveObjectsTable</code> . If <i>is_editing()</i> is not true, then it is as if the calls <i>set_is_editing(true)</i> , <i>set(key, value)</i> and finally <i>set_is_editing(false)</i> were made. If it is true, then no notifications are created.
<i>Key</i>	<i>mot.insert_object(Data ob)</i>	Insert a new object into the table and return a <i>Key</i> which can be used to refer to it. See <i>set(Key, Data)</i> for a description of editing modes.
<i>void</i>	<i>mot.erase(Key key)</i>	Delete an object from the table. The object with <i>Key key</i> must already be in the table. This does not necessarily decrease the amount of storage used at all. In fact, it is unlikely to do so. See <i>set(Key,Data)</i> for an explaining of how the editing modes are used.
<i>void</i>	<i>mot.clear()</i>	Remove all objects from the table and free all storage.

Has Models

Kinetic::Active_objects_vector<MovingObject>

See Also

Multi_listener<Interface>, *Kinetic::Active_objects_listener_helper<ActiveObjectsTable, KDS>*

CGAL::Kinetic::Active_objects_vector<MovingObject>

Definition

MovingObjects are stored in a vector. This means that access is constant time, but storage is not generally freed. The only way to be sure is to remove all reference counts for the table or to call *clear()*.

#include <CGAL/Kinetic/Active_objects_vector.h>

Is Model for the Concepts

Kinetic::ActiveObjectsTable

CGAL::Kinetic::Cartesian_instantaneous_kernel<ActiveObjectsTable, StaticKernel>

Definition

This class provides a model of the *Kinetic::InstantaneousKernel* for use with general Cartesian Geometry. It provides all the predicates needed for Delaunay triangulations and regular triangulations.

For technical reasons, the user must pick out one particular type of primitive to use when instantiating a model. For example, if the user passes a model of *Kinetic::Active_objects_vector<Data, Object>* with a *Kinetic::Kernel::Point_2* as the kinetic primitive, then the type *Point_2* will be properly defined and the predicates on it will work properly, but the other predicates will not work.

```
#include <CGAL/Kinetic/Cartesian_instantaneous_kernel.h>
```

Is Model for the Concepts

Kinetic::InstantaneousKernel

CGAL::Kinetic::Cartesian_kinetic_kernel<FunctionKernel>

Definition

This class provides a model of *Kinetic::Kernel* for use with general Cartesian geometry.

```
#include <CGAL/Kinetic/Cartesian_kinetic_kernel.h>
```

Types

Kinetic::Cartesian_kinetic_kernel<FunctionKernel>:: Certificate

This is a model of *Kinetic::Certificate*.

Kinetic::Cartesian_kinetic_kernel<FunctionKernel>:: Point_1

Kinetic::Cartesian_kinetic_kernel<FunctionKernel>:: Point_2

Kinetic::Cartesian_kinetic_kernel<FunctionKernel>:: Point_3

Kinetic::Cartesian_kinetic_kernel<FunctionKernel>:: Weighted_point_3

The following are functors which generate *Certificate* objects. Each has a corresponding *_object* method which creates the functor.

Kinetic::Cartesian_kinetic_kernel<FunctionKernel>:: Positive_orientation_2

Kinetic::Cartesian_kinetic_kernel<FunctionKernel>:: Positive_orientation_3

Kinetic::Cartesian_kinetic_kernel<FunctionKernel>:: Positive_side_of_oriented_circle_2

Kinetic::Cartesian_kinetic_kernel<FunctionKernel>:: Positive_side_of_oriented_sphere_3

Kinetic::Cartesian_kinetic_kernel<FunctionKernel>:: Power_test_3

Kinetic::Cartesian_kinetic_kernel<FunctionKernel>:: Weighted_positive_orientation_3

Kinetic::Cartesian_kinetic_kernel<FunctionKernel>:: Is_less_x_1

Kinetic::Cartesian_kinetic_kernel<FunctionKernel>:: Is_less_x_2

Kinetic::Cartesian_kinetic_kernel<FunctionKernel>:: Is_less_y_2

Kinetic::Cartesian_kinetic_kernel<FunctionKernel>:: Is_less_x_3

Kinetic::Cartesian_kinetic_kernel<FunctionKernel>:: Is_less_y_3

Kinetic::Cartesian_kinetic_kernel<FunctionKernel>:: Is_less_z_3

Is Model for the Concepts

Kinetic::Kernel.

Kinetic::Certificate

Definition

The concept represents certificate. Its main purpose is to provide a way of creating *Time* objects corresponding to when the certificate fails and to cache any useful work done in find the *Time* for later.

Operations

<i>Time</i>	<i>c.failure_time()</i>	Returns the next failure time.
<i>void</i>	<i>c.pop_failure_time()</i>	Advances to the next failure time (the next root of the certificate functions).

See Also

Kinetic::Kernel.

Kinetic::FunctionKernel::ConstructFunction

Definition

The concept `is` used to construct functions.

Operations

<i>Function</i>	<i>a(NT a, ...)</i>	This family of methods takes a list of coefficients and returns a function. There can be any number of coefficients passed as arguments (up to about 25 in the current implementations).
-----------------	----------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

See Also

FunctionKernel

Example

```
Function_kernel fk;
Function_kernel::Construct_function cf= fk.construct_function_object();
Function_kernel::Function f= cf(0,1,2,3,4,5);
```

Kinetic::EventQueue

Definition

The for priority queues used by the *Simulator*. The concept basically defines a priority queue which supports deletions and changes of items in the queue (but not their priorities). Items in the queue must implement the *Event* concept.

Types

Kinetic::EventQueue::Key The type used to access items in the queue in order to change or delete them.

Kinetic::EventQueue::Priority The priority type for items in the queue. This is typically the same as *Kinetic::Simulator::Time*

.

Creation

Kinetic::EventQueue *q*(*Priority start*, *Priority end*, *int size_hint*);

Construct a queue which will start at time *start* and run until time *end*.

Operations

template <class Event>

Key *q.insert(Priority, Event)*

Insert an event into the event queue. A *Key* which can be used to manipulated the event is returned.

void *q.erase(Key)*

Erase an event from the queue.

template <class Event>

void *q.set(Key, Event)*

Change the data in the event referred to by the key.

template <class Event>

Event& *q.get(Key)*

Access the event referred to by the passed key.

Priority *q.priority(Key)*

Return the priority of the event.

bool *q.empty()*

Return true if the queue is empty.

Priority *q.next_priority()*

Return the priority of the next event in the queue.

<i>void</i>	<i>q.process_next()</i>	Process the next <i>Event</i> by calling its process method with its <i>Priority</i> .
<i>void</i>	<i>q.set_end_priority()</i>	Set the priority beyond which to ignore events.

Has Models

<i>Kinetic::Two_list_pointer_event_queue<FunctionKernel>, FunctionKernel>.</i>	<i>Kinetic::Heap_pointer_event_queue<</i>
-----------------------------------------------------------------------------------------	----------------------------------------------

Kinetic::Simulator::Event

Definition

The concept represents a single event. Models of should be passed to the *Kinetic::Simulator* when scheduling events which will in turn pass them to the *EventQueue*.

Operations

<i>void</i>	<i>a.process()</i>	This method is called when the event occurs. This method will only be called once per time this event is scheduled and the event will be removed from the queue immediately afterwards.
-------------	--------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<i>std::ostream&</i>	<i>std::ostream& << Event</i>
--------------------------	-----------------------------------------

Write a text description of the event to a standard stream.

Has Models

All over the place.

See Also

Kinetic::EventQueue

Example

All of the kinetic data structures provided have models of Event. Here is the code implementing a swap event from the sorting kinetic data structure.

```
template <class Certificate, class Id, class Root_enumerator>
class Swap_event {
public:
    Swap_event(Id o, typename Sort::Handle sorter,
               const Certificate &s): left_object_(o),
                                   sorter_(sorter),
                                   s_(s) {}

    void process() {
        sorter_>swap(left_object_, s_);
    }
    Id left_object_;
    typename Sort::Handle sorter_;
    Certificate s_;
};
```

Kinetic::FunctionKernel

Definition

The concept `Kinetic::FunctionKernel` encapsulates all the methods for representing and handling functions. The set is kept deliberately small to easy use of new `Kinetic::FunctionKernels`, but together these operations are sufficient to allow the correct processing of events, handling of degeneracies, usage of static data structures, run-time error checking as well as run-time verification of the correctness of kinetic data structures. The computation of a polynomial with the variable negated is used for reversing time in kinetic data structures and can be omitted if that capability is not needed.

Types

<i>Kinetic::FunctionKernel:: Function</i>	The type of function being handled.
<i>Kinetic::FunctionKernel:: NT</i>	The basic representational number type.
<i>Kinetic::FunctionKernel:: Root</i>	A type representing the roots of a <i>Function</i> .
<i>Kinetic::FunctionKernel:: Root_stack</i>	A model of <i>RootStack</i> . These objects can be created by calling the <i>root_stack_object</i> method with a <i>Function</i> and two (optional) <i>Root</i> objects. The enumerator then enumerates all roots of the function in the open interval defined by the two root arguments. They optional arguments default to positive and negative infinity.
<i>Kinetic::FunctionKernel:: Root_enumerator_traits</i>	The traits for the <i>Root_enumerator</i> class.
Each of the following types has a corresponding <i>type_object</i> method (not explicitly documented) which takes a <i>Function</i> as an argument.	
<i>Kinetic::FunctionKernel:: Sign_at</i>	A functor which returns the sign of a <i>Function</i> at a <i>NT</i> or <i>Root</i> .
<i>Kinetic::FunctionKernel:: Multiplicity</i>	A functor which returns the multiplicity of roots.
<i>Kinetic::FunctionKernel:: Sign_above</i>	A functor which returns sign of a function immediately above a root.
The following type is used to construct functions from a list of coefficients. To get an instance use the <i>construct_function_object()</i> method.	
<i>Kinetic::FunctionKernel:: Construct_function</i>	This functor can be used create instances of <i>Function</i> . See its reference page <i>FunctionKernel::ConstructFunction</i> for more details.

The following functor likewise have a *type_object* method, but these take arguments other than a *Function*. The arguments are given below.

Kinetic::FunctionKernel:: Sign_between_roots This functor, creation of which requires two *Roots*, returns the sign of a passed function between the pair of roots.

Kinetic::FunctionKernel:: Differentiate This functor computes the derivative of a *Function*. Construction takes no arguments.

The following methods do not require any arguments to get the functor and take one *Function* as a functor argument.

Kinetic::FunctionKernel:: Negate_variable Map $f(x)$ to $f(-x)$.

Has Models

POLYNOMIAL::Kernel<RootStack>, *POLYNOMIAL::Filtered_kernel<RootStack>*.

See Also

Kinetic::RootEnumerator.

Example

We provide several models of the concept, which are not documented separately. The models of *Kinetic::SimulationTraits* all choose appropriate models. However, if more control is desired, we here provide examples of how to create the various supported *Kinetic::FunctionKernel*.

A Sturm sequence based kernel which supports exact comparisons of roots of polynomials (certificate failure times):

```
typedef CGAL::POLYNOMIAL::Polynomial<CGAL::Gmpq> Function;
typedef CGAL::POLYNOMIAL::Sturm_root_stack_traits<Function> Root_stack_traits;
typedef CGAL::POLYNOMIAL::Sturm_root_stack<Root_stack_traits> Root_stack;
typedef CGAL::POLYNOMIAL::Kernel<Function, Root_stack> Function_kernel;
```

A wrapper for *CORE::Expr* which implements the necessary operations:

```
typedef CGAL::POLYNOMIAL::CORE_kernel Function_kernel;
```

A function kernel which computes approximations to the roots of the polynomials:

```
typedef CGAL::POLYNOMIAL::Polynomial<double> Function;
typedef CGAL::POLYNOMIAL::Root_stack_default_traits<Function> Root_stack_traits;
typedef CGAL::POLYNOMIAL::Numeric_root_stack<Root_stack_traits> Root_stack;
typedef CGAL::POLYNOMIAL::Kernel<Function, Root_stack> Function_kernel;
```

When using the function kernel in kinetic data structures, especially one that is in exact, it is useful to wrap the root stack. The wrapper checks the sign of the certificate function being solved and uses that to handle degeneracies. This is done by, for the inexact solvers

```
typedef \ccc{Kinetic::Derivative}_filter_function_kernel<Function_kernel> KDS_function_kernel;
```

and for exact solvers

```
typedef \ccc{Kinetic::Handle}_degeneracy_function_kernel<Function_kernel> KDS_function_kernel;
```

For exact computations, the primary representation for roots is the now standard choice of a polynomial with an associated isolating interval (and interval containing exactly one distinct root of a polynomial) along with whether the root has odd or even multiplicity and, if needed, the Sturm sequence of the polynomial. Two intervals can be compared by first seeing if the isolating intervals are disjoint. If they are, then we know the ordering of the respective roots. If not we can subdivide each of the intervals (using the endpoints of the other interval) and repeat. In order to avoid subdividing endlessly when comparing equal roots, once we subdivide a constant number of times, we use the Sturm sequence of p and $p'q$ (where p and q are the two polynomials and p' is the derivative of p) to evaluate the sign of the second at the root of the first one directly (note that this Sturm sequence is applied to a common isolating interval of the roots of interest of both polynomials).

Kinetic::FunctionKernel::Function

Definition

The concept represents a function.

Types

Function::NT The number type used in describing the function;

Function a(NT); Construct a constant function from a number.

Operations

NT *a(NT)* Evaluate the function at an *NT*.

See Also

FunctionKernel, *FunctionKernel::ConstructFunction*

Example

Several ways to create functions:

Using *Kinetic::ConstructFunction*:

```
Traits::Function_kernel::Construct_function cf= traits.function_kernel_object().construct_function_object();
Traits::Kinetic_kernel::Motion_function x= cf(0.0,1.0,2.0);
Traits::Kinetic_kernel::Motion_function y= cf(0.0,1.0,2.0);
Traits::Kinetic_kernel::Point_2 pt(x,y);
```

Using the constructor:

```
double coefs[]={1.0, 2.0, 3.0};
Traits::Kinetic_kernel::Motion_function z(coefs, coefs+3);
```

Using ring operations:

```
Traits::Kinetic_kernel::Motion_function z= x*z+y;
```

Kinetic::InstantaneousKernel

Definition

The concept `Kinetic::InstantaneousKernel` covers models that act as adaptors allowing CGAL static data structures to act on snapshots of kinetic data. It typically only supports one type of moving object.

Currently, each model is created for one particular type of kinetic primitive. The primitives are identified by `Kinetic::ActiveObjectsTable::Key` objects.

Types

<code>Kinetic::InstantaneousKernel::Time</code>	The type used to represent the current time. This must be a ring or field type.
-------------------------------------------------	---------------------------------------------------------------------------------

Operations

<code>Time</code>	<code>a.time()</code>	Return the current time.
<code>void</code>	<code>a.set_time(Time)</code>	Set the current time to have a certain value. All existing predicates are updated automatically.
<code>Static_object</code>	<code>a.static_object(Key)</code>	Return a static object corresponding to the kinetic object at this instant in time.

Has Models

`Kinetic::Cartesian_instantaneous_kernel`

Key

Definition

The concept Key is a unique identifier for something in some sort of table. In general, they can be only created by the table and are returned when a appropriate *new_foo()* method is called on the table. There are two classes of values for a Key, valid and invalid. The latter cannot refer to something in a table. Use the method *is_valid()* to differentiate.

Key a;

The default constructor is guaranteed to construct an invalid key (i.e. one which is false when cast to a bool).

Operations

bool

a.is_valid()

This method returns false if the key was created using the default constructor or was otherwise created to be invalid.

std::ostream&

std::ostream& << Event

Write a text description of the key to a standard stream.

Has Models

Kinetic::Simulator::Event_key, Kinetic::Active_objects_vector<Object>::Key.

Kinetic::Kernel

Definition

The concept `Kinetic::Kernel` acts as the kinetic analog of a CGAL kernel. It provides some set of primitives and predicates acting on them.

Types

Kinetic::Kernel::Motion_function

The type which is used to represent coordinates of moving primitives. It is a model of the concept *FunctionKernel::Function*. This is the analog of the CGAL kernel *RT*.

Kinetic::Kernel::Certificate

The type representing the results of predicates. See *Kinetic::Certificate*.

Kinetic::Kernel::Function_kernel

The type of the function kernel used. See *Kinetic::FunctionKernel*.

Operations

Function_kernel *kk.function_kernel_object()*

Gets a copy of the function kernel.

Has Models

Kinetic::Cartesian_kinetic_kernel<*FunctionKernel*>.

CGAL::Listener<Interface>

Definition

The *Listener<Interface>* class provides the core of the run time notification system used by the kinetic data structures package. In short, notifications are handled through proxy objects called listeners. In order to listen for notifications from an object, called the notifier, you make define a small class called a listener proxy, which inherits from the Listener interface defined by the notifier. When constructing your listner poxy, you pass a reference counted pointer to the notifier, which is used to register the proxy for notifications. When a notification occurs, the notifier calls the *new_notification* method on the proxy, passing the type of the notification. The proxy stores a reference counted pointer to the notifier, ensuring that there are never any dangling pointers in the system.

The class *Listener<Interface>* provides base class for listener proxy objects. A notifier should provide a class which inherits from this base. To use this base class, implement a class, here called *Interface*, which defines a type *Interface::Notification_type* and a type *Interface::Notifier_handle*.

The *Notification_type* is generally an enum with one value for each type of notification which can be used.

The *Notifier_handle* is the type of a (ref counted) pointer to the object providing the notifications. The ref counter pointer must provide a nested type *Pointer* which is the type of a raw pointer.

The *Listener<Interface>* maintains a ref counted pointer to the object performing notifications. It is registered for notifications on construction and unregistered on destruction using the function *set_listener(Listener<Interface>*)* on the object providing the notifications. The use of ref counted pointers means that as long as the notification object exists, the object providing the notifications must exist, ensuring that the object providing the notifications is not prematurely destroyed.

These objects cannot be copied since the notifier only support one listener. If copying and more than one listener are desired, the *Multi_listener<Interface>* base class should be used instead.

As a side note, Boost provides a similar functionality in the Boost.Signal package. However, it is quite a bit more complex (and flexible). This complexity add significantly to compile time and (although I did not test this directly), I suspect it is much slower at runtime due to the overhead of worrying about signal orders and not supporting single signals. In addition, it does not get on well with Qt due to collisions with the Qt moc keywords.

There is also the TinyTL library which implements signals. As of writing it did not have any easy support for making sure all pointers are valid, so it did not seem to offer significant code saving over writing my own.

```
#include <CGAL/Kinetic/Listener.h>
```

Types

Listener<Interface>::Notifier_handle

This type is inherited from the *Interface* template argument. It is a reference counted pointer type for the object providing notifications.

Listener<Interface>::Notification_type

The type (usually an enum) used to distinguish different types of notifications. This is inherited from the *Interface* template argument.

Creation

Listener<Interface> l(Notifier_handle np); The *Listener<Interface>* subscribes to events coming from the notifier and stores a pointer to the notifier.

Operations

Notifier_handle *l.notifier()* Return a pointer to the notifier.

virtual void *l.new_notification(Notification_type)*

This method is pure virtual. A class which wishes to receive events must inherit from this class and implement this method. The method will then be called whenever there is a notification.

See Also

Multi_listener<Interface>.

Example

Here is a simpler class that provides notifications:

```
struct Notifier: public CGAL::Kinetic::Ref_counted<Notifier>
{
public:
    Notifier(): data_(0), listener_(NULL){}

    struct Listener_interface
    {
    public:
        typedef enum Notification_type {DATA_CHANGED} Notification_type;
        typedef Notifier::Handle Notifier_handle;
    };

    typedef CGAL::Kinetic::Listener<Listener_interface> Listener;
    friend class CGAL::Kinetic::Listener<Listener_interface>;

    void set_data(int d) {
        data_=d;
        if (listener_ != NULL) listener_>new_notification(Listener_interface::DATA_CHANGED);
    }

protected:
    void set_listener(Listener *l) {
        listener_= l;
    }
    Listener* listener() const {return listener_;}
```



```

    int data_;
    Listener *listener_;
};

```

Now the listener:

```

struct My_listener: public Notifier::Listener, public CGAL::Kinetic::Ref_counted<My_listener>
{
    typedef Notifier::Listener::Notifier_handle PP;
    My_listener(PP p): P(p){}

    void new_notification(P::Notification_type nt) {
        ...
    }
};

```

CGAL::Multi_listener<Interface>

Definition

The class *Multi_listener<Interface>* implements a base class for listeners where more than one listener is allowed to subscribe to a notifier. See *Listener* for full documentation. This uses the function calls *new_listener()* and *delete_listener()* to register and unregister the listener (instead of *set_listener()*).

```
#include <CGAL/Kinetic/Multi_listener.h>
```

See Also

Listener<Interface>.

CGAL::Ref_counted<T>

Definition

The class *Ref_counted*<T> implements a base class for objects which are reference counted. To use it simply inherit from *Ref_counted*<T> (passing the type to be reference counted as the template argument) and then access the object through *Handle* objects rather than bare C++ pointers.

```
#include <CGAL/Kinetic/Ref_counted.h>
```

Types

Ref_counted<T>:: *Handle* A reference counted pointer to an Object.

Ref_counted<T>:: *Const_handle* A const reference counted pointer to an Object.

Creation

Ref_counted<T> *rc*; default constructor.

Operations

There are no methods which should be called by users of this class.

Kinetic::RootStack

Definition

The concept `Kinetic::RootStack` enumerates through roots of a function contained in a certain interval. They may return one extra root which has value `std::numeric_limits<Root>::infinity()`, as this allows the root stacks to delay isolating roots until the isolation is needed to properly evaluate comparisons.

Types

`Kinetic::RootStack::Root` The root of a function.

`Kinetic::RootStack::Traits` The traits class for this concept.

Creation

`Kinetic::RootStack re;` default constructor.

`Kinetic::RootStack re(Function f, Root lb, Root ub, Traits tr);`

Construct a `Kinetic::RootStack` over the roots of f in the open interval lb to ub .

Operations

`void` `re.pop()` Advance to the next root. As a precondition, `empty()` must be false.

`Root` `re.top()` Return the current root. As a precondition, `empty()` must be false. Note that the `Root` returned might not actually be in the interval (since the solver has not yet proved that there are no more roots).

`bool` `re.empty()` Return true if there are known to be no more roots left. There might not actually be any roots of the polynomial left in the interval, but the work necessary to prove this has been delayed.

See Also

`Kinetic::FunctionKernel`, `Kinetic::Certificate`.

Kinetic::SimulationTraits

Definition

This concept ties together the parts needed in order to run a kinetic data structure. We provide several models of this concept:

- *Kinetic::Exact_simulation_traits_1*
- *Kinetic::Exact_simulation_traits_2*
- *Kinetic::Exact_simulation_traits_3*
- *Kinetic::Inexact_simulation_traits_1*
- *Kinetic::Inexact_simulation_traits_2*
- *Kinetic::Inexact_simulation_traits_3*
- *Kinetic::Exact_linear_simulation_traits_2*
- *Kinetic::Exact_linear_simulation_traits_3*
- *Kinetic::Inexact_linear_simulation_traits_2*
- *Kinetic::Inexact_linear_simulation_traits_3*
- *Kinetic::Regular_triangulation_exact_simulation_traits_3*
- *Kinetic::Regular_triangulation_inexact_simulation_traits_3*

All support trajectories defined by polynomial coordinates. The *Exact* vs *Inexact* picks whether the roots of the certificate functions are compared exactly or approximated numerically. The regular triangulation models have weighted points of the appropriate dimension as the primitive used in the *Kinetic::InstantaneousKernel* and the *Kinetic::ActiveObjectsTable*.

Types

Kinetic::SimulationTraits:: NT The number type used for representation.

Kinetic::SimulationTraits:: Instantaneous_kernel

A model of *Kinetic::InstantaneousKernel* which can be used to apply static CGAL data structures to snapshots of moving data.

Kinetic::SimulationTraits:: Kinetic_kernel A model of *Kinetic::Kernel*.

Kinetic::SimulationTraits:: Function_kernel A model of *Kinetic::FunctionKernel*.

Kinetic::SimulationTraits:: Active_points_[123]_table

A model of *Kinetic::ActiveObjectsTable* which holds the relevant kinetic primitives.

Kinetic::SimulationTraits:: Simulator

A model of *Kinetic::Simulator* which will be used by all the kinetic data structures.

Operations

Instantaneous_kernel *st.instantaneous_kernel_object()*

Get a new instantaneous kernel.

Kinetic_kernel *st.kinetic_kernel_object()*

Get a new kinetic kernel.

Function_kernel *st.function_kernel_object()*

Get a new function kernel.

Simulator::Handle *st.simulator_handle()*

Return a pointer to the *Kinetic::Simulator* which is to be used in the simulation.

Active_points_[123]_table::Handle

st.active_points_[123]_table_handle()

Return a pointer to the *Kinetic::ActiveObjectsTable* which is to be used in the simulation.

Has Models

Kinetic::Exact_simulation_traits_1, *Kinetic::Exact_simulation_traits_2*, *Kinetic::Exact_simulation_traits_3*,
Kinetic::Inexact_simulation_traits_1, *Kinetic::Inexact_simulation_traits_2*, *Kinetic::Inexact_simulation_traits_3*,
Kinetic::Exact_linear_simulation_traits_2, *Kinetic::Exact_linear_simulation_traits_3*, *Kinetic::Inexact_linear_simulation_traits_2*,
Kinetic::Inexact_linear_simulation_traits_3, *Kinetic::Regular_triangulation_exact_simulation_traits_3*, *Kinetic::Regular_triangulation_inexact_simulation_traits_3*

CGAL::Kinetic::Default_simulator<FunctionKernel, EventQueue>

Definition

The class *Kinetic::Default_simulator<FunctionKernel, EventQueue>* controls kinetic data structures by maintaining a concept of time and ensuring that events are processed when necessary.

```
#include <CGAL/Kinetic/Default_simulator.h>
```

Is Model for the Concepts

Kinetic::Simulator.

Creation

```
Kinetic::Default_simulator<FunctionKernel, EventQueue> sim( const Time start=Time(0),  
                                                         const Time end= Time::infinity())
```

Construct a *Kinetic::Default_simulator<FunctionKernel, EventQueue>* which will process events between times start and end (events outside this window will be discarded).

CGAL::Kinetic::Simulator_kds_listener<Listener, KDS>

Definition

The class *Kinetic::Simulator_kds_listener<Listener, KDS>* acts as a helper class for kinetic data structures which want to respond to *Simulator::Listener::HAS_AUDIT_TIME* notifications. When kinetic data structures can audit themselves, the *Kinetic::Simulator_kds_listener<Listener, KDS>* calls the *audit()* method on the kinetic data structure.

```
#include <CGAL/Kinetic/Simulator_kds_listener.h>
```

Creation

```
Kinetic::Simulator_kds_listener<Listener, KDS> a( Simulator::Handle, KDS *kds);
```

default constructor.

See Also

Kinetic::Simulator, *Listener<Interface>*.

CGAL::Kinetic::Simulator_objects_listener<Simulator_listener, KDS>

Definition

The class *Kinetic::Simulator_objects_listener<Simulator_listener, KDS>* is a helper for classes which wish to react to *Simulator::Listener::DIRECTION_OF_TIME* notifications. The helper object translates such notifications *reverse_time* function calls on the responder. See *Kinetic::Qt_moving_points_2* for a simple example of using this helper function.

```
#include <CGAL/Kinetic/Simulator_objects_listener.h>
```

Creation

```
Kinetic::Simulator_objects_listener<Simulator_listener, KDS> a( Simulator::Handle, KDS*);
```

default constructor.

See Also

Kinetic::Listener.

Kinetic::Simulator

Definition

The class `Kinetic::Simulator` controls kinetic data structures by maintaining a the current time and ensuring that events are processed when necessary.

In addition, the `Kinetic::Simulator` can call on the kinetic data structures to audit themselves at appropriate times. When the last event processed and the next to be processed have different times, then there is a rational value of time at which all kinetic data structures should be non-degenerate (since there are no events at that time). At such a time, kinetic data structures can easily verify their correctness by checking that all the certificate predicates have the correct value. When exactness checks are enabled, whenever the last event processed and the next event to be processed have different times, a `Kinetic::Simulator::Listener::HAS_AUDIT_TIME` notification is made. Kinetic data structures can listen for that event, and when it is made, they can call `Kinetic::Simulator::audit_time()` to get the time value and then verify that their structure is correct.

Typically, the simulator is created by the `Kinetic::SimulationTraits` class and kinetic data structures request a handle to it from there.

Types

`Kinetic::Simulator::Function_kernel`

The type of the function kernel used to instantiate this `Kinetic::Simulator`.

`Kinetic::Simulator::Listener`

Extend this base class to listen to notifications from this `Kinetic::Simulator`. There are two types of notifications: `HAS_AUDIT_TIME` and `DIRECTION_OF_TIME`. The first is made when kinetic data structures can perform an audit. The second is made when the direction of time is changed.

`Kinetic::Simulator::Time`

The representation type for times in the simulator. It is `Function_kernel::Root_enumerator::Root`.

`Kinetic::Simulator::Event_key`

The key to access scheduled `Event` in order to inspect or delete them.

`Kinetic::Simulator::NT`

The basic number type used in computations.

`Kinetic::Simulator::Handle`

A reference counted pointer to be used for storing references to the object.

`Kinetic::Simulator::Const_handle`

A reference counted pointer to be used for storing references to the object.

Creation

`Kinetic::Simulator sim(const Time start=Time(0), const Time end= Time::infinity());`

Construct a `Kinetic::Simulator` which will process events between times `start` and `end` (events outside this window will be discarded).

Operations

<i>Function_kernel</i>	<i>sim.function_kernel_object()</i>	Access the <i>Function_kernel</i> object used by the <i>Kinetic::Simulator</i> .
<i>Time</i>	<i>sim.current_time()</i>	Return the current time.
<i>void</i>	<i>sim.set_current_time(Time t)</i>	Set the current time to <i>t</i> , which cannot be less than <i>current_time</i> . Any events in the queue before time <i>t</i> are processed.
<i>NT</i>	<i>sim.current_time_nt()</i>	This returns a ration number representing the current time. This only returns a valid time if <i>has_current_time_nt()</i> is true.
<i>bool</i>	<i>sim.has_rational_current_time()</i>	Return true if there is a rational number which is equivalent to the current time. Equivalent means that it has the same ordering relation to all previous and scheduled events.
<i>bool</i>	<i>sim.has_audit_time()</i>	Returns true if the current time is a rational number and there are no events at the current time. This means that the simulation can be audited at this time.
<i>Time</i>	<i>sim.next_event_time()</i>	Return the time of the next event in the queue.
<i>Time</i>	<i>sim.end_time()</i>	Return the time the simulation will end. If time is running backwards, then this returns <i>Time::infinity()</i> .
<i>void</i>	<i>sim.set_end_time(Time t)</i>	Advance the end time to <i>t</i> . This is not generally a safe operation as all the kinetic data structures must rebuild their certificates at this time.
<i>template <class Event></i> <i>Event_key</i>	<i>sim.new_event(Time t, const Event event)</i>	Schedule a new event at time <i>t</i> . The object <i>event</i> must implement the concept <i>Event</i> . The <i>Event_key</i> returned can be used to access or deschedule the event.
<i>Event_key</i>	<i>sim.null_event()</i>	This method returns an <i>Event_key</i> which is guaranteed never to be assigned to any real event.

void *sim.delete_event(const Event_key k)*

Remove the event referenced by *k* from the event queue.

template <class Ev>
typename Queue::Event_handle<Ev>::Handle

sim.event(const Event_key k, const Ev e)

This method returns a pointer to an event, which can be used for recovering data, such as cached solvers, from that event. The second argument really shouldn't be there, but gcc seems to sometimes have issues if you try to specify the template value directly.

Time *sim.event_time(Event_key k)*

Return the time at which the event referenced by *k* occurs.

template <class Ev>
Event_key *sim.set_event(Event_key k, const Ev ev)*

Set the event referenced by key *k* to *ev*, for example if you want to change what happens when that event occurs. A new event key is returned.

Sign *sim.direction_of_time()*

Return *POSITIVE* if time is running forwards or *NEGATIVE* if it is running backwards.

void *sim.set_direction_of_time(Sign dir)*

Set which direction time is running.

unsigned int *sim.current_event_number()*

Return the number of events which have been processed.

void *sim.set_current_event_number(unsigned int i)*

Process all events up to the *i*th event. *i* cannot be less than *current_event_number*.

See Also

Kinetic::Simulator_objects_listener<Simulator_listener, KDS>, *Kinetic::Simulator_kds_listener<Simulator_listener, KDS>*.

Has Models

Kinetic::Default_simulator<FunctionKernel, EventQueue>

Kinetic::Simulator::Time

Definition

The concept `time` represents time in the simulator.

Time $a(NT)$;

Construct an instance of time from a number type, where NT is the number type used in the simulation.

Operations

std::ostream& *std::ostream&* << *Time*

Write it to a stream.

double *to_double(Time)*

Return a double approximation of the time value.

std::pair<double, double>

to_interval(Time)

Return an interval containing the time value.

Comparisons with other *Kinetic::Simulator::Time* objects are supported.

Has Models

double, *Kinetic::FunctionKernel::Root*

See Also

Kinetic::Simulator

Part XIV

Support Library

Chapter 44

Number Type Support

Olivier Devillers, Susan Hert, Lutz Kettner, Sylvain Pion, and Stefan Schirra

CGAL kernel classes are parameterized by number types. Depending on the problem and the input data that have to be handled, one has to make a trade-off between efficiency and accuracy in order to select an appropriate number type and kernel class.

In homogeneous representation, two number types are involved, although only one of them appears as a template parameter in the homogeneous kernel classes. This type, for the sake of simplicity and readability called ring type, is used for the representation of homogeneous coordinates and all internal computations. If it is assured that the second operand divides the first one, these internal computations are basically division-free. The ring type is a placeholder for an integer type (or an integral domain type) rather than for elements of arbitrary rings. The name should remind you that the division operation is not needed for this number type. Of course, also more general number types can be used as a ring type in a homogeneous kernel class. In some computations, e.g. accessing Cartesian coordinates, divisions cannot be avoided. In these computations a second number type, the field type, is used. CGAL automatically generates this number type as a *Quotient*, cf. Subsection 44.9. For the Cartesian kernels there is only one number type that is used for all calculations.

The kernel classes provide access to the number types involved in the representation, although it is not expected that such access is needed at this level, since low-level geometric operations are wrapped in geometric primitives provided by CGAL. This access can be useful if appropriate primitives are missing. In a homogeneous kernel class K , ring type and field type can be accessed as $K::RT$ and $K::FT$, respectively. The number type used in Cartesian kernels is considered as ring type or as field type depending on the context. It can be accessed as $K::RT$ and $K::FT$, according to the use of number types used in the homogeneous counterpart.

44.1 Required Functionality of Number Types

Number types must fulfill certain requirements, such that they can be successfully used in CGAL code. The syntactical requirements of number types are described in the concepts `RingNumberType` and `FieldNumberType` included in the kernel reference manual. Of course, number types also have evident semantic constraints. They should be meaningful in the sense that they approximate the integers or the rationals or some other subfield of the real numbers.

44.2 Utility Routines

The number type concepts mentioned in the previous section list all the required functionality. For the user of a number type it is handy to have a larger set of operations available. CGAL defines a number of such operations, to compute, for example, the minimum or maximum of two numbers, and the absolute value, square, sign or square root of a number. These are available both as global functions and as functors. See the reference manual for more details.

Those routines are implemented using the required operations from the number type concepts. They are defined by means of templates, so you do not have to supply all those operations when you write a new number type. But if you have a better implementation for any of them, you can provide a corresponding overloading function with the same name for your number types, which will get preference over the template functions listed above.

For the number types *int* and *double* there is also a random numbers generator *CGAL::Random*.

44.3 Built-in Number Types

The built-in number types *float*, *double* and *long double* have the required arithmetic and comparison operators. They lack some required routines though which are automatically included by CGAL. ¹

All built-in number types of C++ can represent a discrete (bounded) subset of the rational numbers only. We assume that the floating-point arithmetic of your machine follows IEEE floating-point standard. Since the floating-point culture has much more infrastructural support (hardware, language definition and compiler) than exact computation, it is very efficient. Like with all number types with finite precision representation which are used as approximations to the infinite ranges of integers or real numbers, the built-in number types are inherently potentially inexact. Be aware of this if you decide to use the efficient built-in number types: you have to cope with numerical problems. For example, you can compute the intersection point of two lines and then check whether this point lies on the two lines. With floating point arithmetic, roundoff errors may cause the answer of the check to be *false*. With the built-in integer types overflow might occur.

44.4 Number Types Provided by CGAL

CGAL provides several number types that are that can be used for exact computation. These include the *Quotient* class that can be used to create, for example, a number type that behaves like a rational number. When used in conjunction with the number type *MP_Float* that is able to represent multi-precision floating point values, you achieve an exact rational number representation.

The templated number type *Lazy_exact_nt<NT>* is able to represent any number that *NT* is able to represent, but because it first tries to use an approximate value to perform computations it can be faster than the provided number type *NT*. CGAL also provides a fixed-precision number type,

Fixed_precision_nt is a number type that provides 24-bit numbers in fixed point representation. This number type provides some specialized predicates that are exact and efficient for numbers known to be representable using 24 bits.

A number type for doing interval arithmetic, *Interval_nt*, is provided. This number type helps in doing filtering of predicates.

¹ The functions can be found in the header files `<CGAL/float.h>`, `<CGAL/double.h>` and `<CGAL/long_double.h>`.

CGAL::Root_of_2 is a number type that allows to represent algebraic numbers of degree up to 2 over a *RingNumberType*. A generic function *CGAL::make_root_of_2* allows to build this type generically.

A debugging helper *Number_type_checker<NT1,NT2,Comparator>* is also provided which allows to compare the behavior of operations over two number types.

44.5 Number Type Provided by CORE

CGAL defines the functions needed to use the number type *CORE::Expr* provided by CORE [KLPY99]. To use CORE with CGAL, just install CGAL with CORE support, and include the file *CGAL/CORE_Expr.h*. CORE version 1.7 or later is required.

CORE::Exprs are a subset of real algebraic numbers. Any integer is a *CORE::Expr* and *CORE::Exprs* are closed under the operations $+$, $-$, \times , $/$ and $\sqrt{}$. *CORE::Exprs* guarantee that all comparisons between expressions involving *CORE::Exprs* produce the exact result.

This number type provides an equivalent functionality to *leda_real*.

44.6 Number Types Provided by GMP

CGAL provides wrapper classes for number types defined in the GNU Multiple Precision arithmetic library [Gra]. The file *CGAL/Gmpz.h* provides the class *Gmpz*, a wrapper class for the integer type *mpz_t*, that is compliant with the CGAL number type requirements. The file *CGAL/Gmpq.h* provides the class *Gmpq*, a wrapper class for the rational type *mpq_t*, that is compliant with the CGAL number type requirements.

In addition, it is possible to directly use the C++ number types provided by GMP : *mpz_class*, *mpq_class* (note that support for *mpf_class* is incomplete). The file *CGAL/gmpxx.h* provides the necessary functions to make these classes compliant to the CGAL number type requirements.

To use this, GMP must be installed.

44.7 Number Types Provided by LEDA

LEDA provides number types that can be used for exact computation with both Cartesian and homogeneous representations. If you are using homogeneous representation with the built-in integer types *short*, *int*, and *long* as ring type, exactness of computations can be guaranteed only if your input data come from a sufficiently small integral range and the depth of the computations is sufficiently small. LEDA provides the number type *leda_integer* for integers of arbitrary length. (Of course the length is somehow bounded by the resources of your computer.) It can be used as ring type in homogeneous kernels and leads to exact computation as long as all intermediate results are rational. For the same kind of problems, Cartesian representation with number type *leda_rational* leads to exact computation as well. The number type *leda_bigfloat* in LEDA is a variable precision floating-point type. Rounding mode and precision (i.e. mantissa length) of *leda_bigfloat* can be set.

The most sophisticated number type in LEDA is the number type called *leda_real*. Like in Pascal, where the name *real* is used for floating-point numbers, the name *leda_real* does not describe the number type precisely, but intentionally. *leda_reals* are a subset of real algebraic numbers. Any integer is *leda_real* and *leda_reals* are closed under the operations $+$, $-$, \times , $/$ and k -th root computation. *leda_reals* guarantee that all comparisons between expressions involving *leda_reals* produce the exact result.

44.8 User-supplied Number Types

You can also use your own number type with the CGAL kernel classes. Depending on the arithmetic operations carried out by the algorithms that you are going to use, the number types must fulfill the corresponding requirements from Section [44.1](#).

Number Type Support Reference Manual

Olivier Devillers, Susan Hert, Lutz Kettner, Sylvain Pion, and Stefan Schirra

44.9 Classified Reference Pages

Global Functions

<i>CGAL::abs</i>	page 2521
<i>CGAL::compare</i>	page 2523
<i>CGAL::div</i>	page 2526
<i>CGAL::is_negative</i>	page 2549
<i>CGAL::is_one</i>	page 2550
<i>CGAL::is_positive</i>	page 2551
<i>CGAL::is_zero</i>	page 2552
<i>CGAL::make_root_of_2</i>	page 2563
<i>CGAL::max</i>	page 2564
<i>CGAL::min</i>	page 2565
<i>CGAL::sign</i>	page 2583
<i>CGAL::sqrt</i>	page 2585
<i>CGAL::square</i>	page 2588

Function Object Classes

<i>CGAL::Abs<NT></i>	page 2522
<i>CGAL::Compare<NT></i>	page 2524
<i>CGAL::Gcd<NT></i>	page 2538
<i>CGAL::Div<NT></i>	page 2527
<i>CGAL::Is_negative<NT></i>	page 2553
<i>CGAL::Is_one<NT></i>	page 2554
<i>CGAL::Is_positive<NT></i>	page 2555
<i>CGAL::Is_zero<NT></i>	page 2556
<i>CGAL::Max<NT, Compare></i>	page 2566
<i>CGAL::Min<NT, Compare></i>	page 2567
<i>CGAL::Sgn<NT></i>	page 2589
<i>CGAL::Sqrt<NT></i>	page 2590
<i>CGAL::Square<NT></i>	page 2591
<i>CGAL::To_double<NT></i>	page 2592

CGAL::To_interval<NT> page [2593](#)

Tags

CGAL::Ring_tag page [2579](#)

CGAL::Euclidean_ring_tag page [2529](#)

CGAL::Field_tag page [2531](#)

CGAL::Sqrt_field_tag page [2587](#)

Traits Classes

CGAL::Number_type_traits<NT> page [2572](#)

CGAL::Rational_traits<NT> page [2575](#)

CGAL::Root_of_traits_2<RT> page [2581](#)

Number Type Concepts

EuclideanRingNumberType page [2528](#)

FieldNumberType page [2530](#)

RingNumberType page [2576](#)

RootOf_2 page [2580](#)

SqrtFieldNumberType page [2586](#)

Number Type Classes

CGAL::Filtered_exact<CT, ET> page [2532](#)

CGAL::Fixed_precision_nt page [2534](#)

CGAL::Gmpq page [2541](#)

CGAL::Gmpz page [2543](#)

CGAL::Interval_nt<Protected> page [2544](#)

CGAL::Lazy_exact_nt<NT> page [2557](#)

CGAL::MP_Float page [2568](#)

CGAL::Number_type_checker<NT1,NT2,Comparator> page [2570](#)

CGAL::Quotient<NT> page [2573](#)

CGAL::Root_of_2<RT> page [2582](#)

CORE::Expr page [2525](#)

leda_bigfloat page [2559](#)

leda_integer page [2560](#)

leda_rational page [2561](#)

leda_real page [2562](#)

mpq_class page [2539](#)

mpz_class page [2540](#)

44.10 Alphabetical List of Reference Pages

<i>Abs</i> <NT>	page 2522
<i>abs</i>	page 2521
<i>Compare</i> <NT>	page 2524
<i>compare</i>	page 2523
<i>CORE::Expr</i>	page 2525
<i>Div</i> <NT>	page 2527
<i>div</i>	page 2526
<i>EuclideanRingNumberType</i>	page 2528
<i>Euclidean_ring_tag</i>	page 2529
<i>FieldNumberType</i>	page 2530
<i>Field_tag</i>	page 2531
<i>Filtered_exact</i> <CT, ET>	page 2532
<i>Fixed_precision_nt</i>	page 2534
<i>Gcd</i> <NT>	page 2538
<i>gcd</i>	page 2537
<i>Gmpq</i>	page 2541
<i>Gmpz</i>	page 2543
<i>Interval_nt</i> <Protected>	page 2544
<i>Is_negative</i> <NT>	page 2553
<i>is_negative</i>	page 2549
<i>Is_one</i> <NT>	page 2554
<i>is_one</i>	page 2550
<i>Is_positive</i> <NT>	page 2555
<i>is_positive</i>	page 2551
<i>Is_zero</i> <NT>	page 2556
<i>is_zero</i>	page 2552
<i>Lazy_exact_nt</i> <NT>	page 2557
<i>leda_bigfloat</i>	page 2559
<i>leda_integer</i>	page 2560
<i>leda_rational</i>	page 2561
<i>leda_real</i>	page 2562
<i>make_root_of_2</i>	page 2563
<i>Max</i> <NT, Compare>	page 2566
<i>max</i>	page 2564
<i>Min</i> <NT, Compare>	page 2567
<i>min</i>	page 2565
<i>mpq_class</i>	page 2539
<i>mpz_class</i>	page 2540
<i>MP_Float</i>	page 2568
<i>Number_type_checker</i> <NT1, NT2, Comparator>	page 2570
<i>Number_type_traits</i> <NT>	page 2572
<i>Quotient</i> <NT>	page 2573
<i>Rational_traits</i> <NT>	page 2575
<i>RingNumberType</i>	page 2576
<i>Ring_tag</i>	page 2579
<i>RootOf_2</i>	page 2580
<i>Root_of_2</i> <RT>	page 2582
<i>Root_of_traits_2</i> <RT>	page 2581
<i>Sgn</i> <NT>	page 2589
<i>sign</i>	page 2583
<i>simplest_rational_in_interval</i>	page 2584
<i>Sqrt</i> <NT>	page 2590

<i>SqrtFieldNumberType</i>	page 2586
<i>Sqrt_field_tag</i>	page 2587
<i>sqrt</i>	page 2585
<i>Square<NT></i>	page 2591
<i>square</i>	page 2588
<i>To_double<NT></i>	page 2592
<i>To_interval<NT></i>	page 2593
<i>to_rational</i>	page 2594

CGAL::abs

Definition

The function *abs* returns the absolute value of a number.

```
#include <CGAL/number_utils.h>
```

```
template <class NT>  
NT abs( NT ntval)
```

CGAL::Abs<NT>

Definition

The function object class *Abs*<*NT*> computes the absolute value of a number.

#include <*CGAL/number_utils_classes.h*>

Is Model for the Concepts

AdaptableFunctor.....page [2656](#)

NT *f(NT nval)* returns the absolute value of *nval*.

CGAL::compare

Definition

The function *compare* compares two numbers to see which is larger or smaller or if they are equal.

```
#include <CGAL/number_utils.h>
```

```
template <class NT1, class NT2>  
Comparison_result    compare( NT1 n1, NT2 n2)
```

returns *LARGER* iff $n1 > n2$, *SMALLER* iff $n1 < n2$, and *EQUAL* otherwise.

CGAL::Compare<NT>

Definition

The function object class *Compare*<*NT*> compares two numbers to see which is larger or smaller or if they are equal.

#include <*CGAL/number_utils_classes.h*>

Is Model for the Concepts

AdaptableFunctor.....page [2656](#)

Comparison_result *f*(*NT* *n1*, *NT* *n2*) returns *LARGER* iff *n1* > *n2*, *EQUAL* iff *n1* = *n2*, and
SMALLER i ff *n1* < *n2*.

CORE::Expr

Definition

The class *CORE::Expr* provides exact computation over the subset of real numbers that contains integers, and which is closed by the operations $+$, $-$, \times , $/$ and $\sqrt{}$. Comparisons between objects of this type are guaranteed to be exact. This number type is provided by the CORE library [KLPY99].

CGAL defines the necessary functions so that this class complies to the requirements on number types.

```
#include <CGAL/CORE_Expr.h>
```

Is Model for the Concepts

FieldNumberType

CGAL::div

Definition

The function *div* computes the integral division of two numbers.

```
#include <CGAL/number_utils.h>
```

```
NT div( NT ntval1, NT ntval2)
```

CGAL::Div<NT>

Definition

The function object class *Div<NT>* computes the integral division of two numbers.

```
#include <CGAL/number_utils_classes.h>
```

Is Model for the Concepts

AdaptableFunctor.....page 2656

$$NT \quad f(NT \text{ ntv}1, NT \text{ ntv}2)$$

computes the integral divisor of two numbers.

EuclideanRingNumberType

The number type supports the operations $+$, $-$ and $*$ as well as a function *div*, which performs an integer division, the modulus operator $\%$, that returns the remainder of integer division and the function *gcd*. This implies that *CGAL::Number_type_traits<EuclideanRingNumberType>::Has_gcd* is *CGAL::Tag_true*.

Refines

RingNumberType

Operations

<i>EuclideanRingNumberType</i>	<i>div</i> (<i>n1</i> , <i>n2</i>)	returns the result of the integer division <i>n1/n2</i> .
<i>EuclideanRingNumberType</i>	<i>gcd</i> (<i>n1</i> , <i>n2</i>)	computes the greatest common divisor between <i>n1</i> and <i>n2</i> .
<i>EuclideanRingNumberType</i>	<i>n1</i> % <i>n2</i>	returns the remainder achieved when dividing <i>n1</i> by <i>n2</i> .
<i>EuclideanRingNumberType</i>	<i>n1</i> % = <i>n2</i>	computes the remainder achieved when dividing <i>n1</i> by <i>n2</i> and assigns it to <i>n1</i> .

Has Models

int
long
long long
CGAL::Gmpz
leda_integer

See Also

FieldNumberType	page 2530
Kernel	page 35
<i>CGAL::Euclidean_ring_tag</i>	page 2529
Support Library Manual	

CGAL::Euclidean_ring_tag

Definition

The class *Euclidean_ring_tag* is used as a tag in some algorithms. It indicates that a number type is to be considered as a model for `EuclideanRingNumberType`. Only operations defined for `EuclideanRingNumberType` are used. For example, no divisions are computed, even if the number type as such supports divisions.

```
#include <CGAL/Number_type_traits.h>
```

See Also

`EuclideanRingNumberType` page [2528](#)
Ring_tag page ??
Field_tag page ??
Sqrt_field_tag page ??

FieldNumberType

The concept `FieldNumberType` defines the syntactic requirements of a number type to be used as a template parameter for the Cartesian kernels. This number type supports the operations `+`, `-`, `*` and `/`. This implies that `CGAL::Number_type_traits<FieldNumberType>::Has_division` is `CGAL::Tag_true`.

Refines

`RingNumberType`

Operations

<code>FieldNumberType</code>	<code>n1 / n2</code>
<code>FieldNumberType</code>	<code>int n1 / n2</code>
<code>FieldNumberType</code>	<code>n1 / int n2</code>
<code>FieldNumberType</code>	<code>n1 / = n2</code>
<code>FieldNumberType</code>	<code>n1 / = int n2</code>

Has Models

`float`
`double`
`CGAL::Filtered_exact<FieldNumberType, ET>`
`CGAL::Fixed_precision_nt`
`CGAL::Gmpq`
`CGAL::Interval_nt`
`CGAL::Interval_nt_advanced`
`CGAL::Lazy_exact_nt<FieldNumberType>`
`CGAL::MP_Float`
`CGAL::Quotient<RingNumberType>`
`leda_rational`
`leda_bigfloat`
`leda_real`

See Also

`EuclideanRingNumberType` page [2528](#)
`Kernel` page [35](#)
`CGAL::Field_tag` page [2531](#)
 Support Library Manual

CGAL::Field_tag

Definition

The class *Field_tag* is used as a tag in some algorithms. It indicates that a number type is to be considered as a model for `FieldNumberType`. Only operations defined for `FieldNumberType` are used. For example, no squareroot operations are computed, even if the number type as such supports squareroots.

```
#include <CGAL/Number_type_traits.h>
```

See Also

`FieldNumberType` page [2530](#)
`Ring_tag` page ??
`Euclidean_ring_tag` page ??
`Sqrt_field_tag` page ??

CGAL::Filtered_exact<CT, ET>

Definition

NOTE : The functionality provided by *Filtered_exact<CT,ET>* is superseded by *Filtered_kernel*, hence the use of *Filtered_exact<CT,ET>* is deprecated. We recommend that users update their code as soon as possible, as we may not guarantee proper functioning of *Filtered_exact<CT,ET>* in the future.

The class *Filtered_exact<CT,ET>* is a wrapper type for the number type *CT*, with the difference that all predicates are specialized such that they are guaranteed to be exact. Speed is achieved via a filtering scheme using interval arithmetic (see Section 44.9). Here are the necessary requirements:

- *CT* is the construction and storage type. The only data member of the class *Filtered_exact<CT,ET>* is the *value* of type *CT*. All arithmetic operations performed *outside* of the predicates will be executed with this number type. You can disallow these operations compiling with the flag *CGAL_DENY_INEXACT_OPERATIONS_ON_FILTER* (it allows you to spot the inexact operations that should be incorporated in the predicates). The arithmetic operations called from inside the predicates are always computed exactly.
- The *ET* type must be able to compute exactly the operations involved in the predicates called.
- A *to_interval(CT)* function must be provided, that returns an interval containing the value of the argument of type *CT*, see Section 44.9.
- A *convert_to<ET>(CT)* function must also be provided, that returns a number of type *ET* representing exactly the argument of type *CT*. It's a conversion function that is used for the exact computation, when the filter fails. This conversion has to be done exactly to ensure robustness.

```
#include <CGAL/Filtered_exact.h>
```

Is Model for the Concepts

FieldNumberType

Operations

The following member functions are used to access the numerical value for the different number types:

<i>CT</i>	<i>ntvar.value()</i>	returns the wrapped value.
-----------	----------------------	----------------------------

<i>ET</i>	<i>ntvar.exact()</i>	returns the converted value to <i>ET</i> .
-----------	----------------------	--------------------------------------------

Interval_nt_advanced

<i>ntvar.interval()</i>	returns the converted value to <i>Interval_nt_advanced</i> .
-------------------------	--------------------------------------------------------------

This type actually has additional parameters for experimental features. They will be documented when they will be considered stable, in a next release.

Example

You might use at the beginning of your program a *typedef* as follows:

```
#include<CGAL/Filtered_exact.h>
#include<CGAL/leda_real.h>
#include<CGAL/double.h>
typedef Filtered_exact<double, leda_real> NT;
```

Or if you are sure that the predicates involved do not use divisions nor square roots:

```
#include<CGAL/Filtered_exact.h>
#include<CGAL/Gmpz.h>
#include<CGAL/int.h>
typedef Filtered_exact<int, Gmpz> NT;
```

And if you know that the double variables contain integer values, you can use:

```
#include<CGAL/Filtered_exact.h>
#include<CGAL/Gmpz.h>
#include<CGAL/double.h>
typedef Filtered_exact<double, Gmpz> NT;
```

As a general rule, we advise the use of *Filtered_exact<double, leda_real>*.

Implementation

The template definition of the low level predicates of CGAL are overloaded for the type *Filtered_exact<CT,ET>*.

For each predicate file, the overloaded code is generated automatically by the *PERL* script (*scripts/cgal_filtered_predicates_generator.pl*) that you can use for your own predicates. This script parses the template functions and generates the overloaded code the following way:

- convert the entries to intervals using *to_interval(CT)*, via the *interval()* member function,
- call the original template function with the type *Interval_nt_advanced*,
- if no exception is thrown, return the value,
- if an exception is thrown (the filter failed), convert the original entries using *convert_to<ET>(CT)*, using the *exact()* member function,
- and call the original template function with the type *ET*.

Example

The low level template predicates of CGAL are in files named *CGAL/predicates/kernel_ftC2.h* (resp. *ftC3*), the script is used to produce the files *CGAL/Arithmetic_filter/predicates/kernel_ftC2.h* (resp. *ftC3*).

At the moment, only the predicates of the Cartesian and Simple_cartesian kernels are supported, as well as the power tests used by the regular triangulations.

CGAL::Fixed_precision_nt

Definition

The class *Fixed_precision_nt* provides 24-bit numbers in fixed point representation. Basically these numbers are integers in the range $[-2^{24}, 2^{24}]$ with a multiplying factor 2^b . The multiplying factor 2^b has to be initialized by the user before the construction of the first *Fixed_precision_nt* and is common to all variables.

The interest of such a number type is that geometric predicates can be overloaded to get exact and very efficient predicates. The drawback is that any *Fixed_precision_nt* is rounded to the nearest multiple of 2^b , which yields to a very poor arithmetic. The idea is to not use the arithmetic on *Fixed_precision_nt* but only the specialized predicates.

Note: you must call *CGAL::force_ieee_double_precision()* in order for the *Fixed_precision_nt* to work properly on Intel platforms. This initializes the FPU to an IEEE compliant rounding mode which is not the default.

```
#include <CGAL/Fixed_precision_nt.h>
```

Is Model for the Concepts

RingNumberType

Creation

```
Fixed_precision_nt fvar;
```

Declaration.

```
Fixed_precision_nt fvar( double d);
```

Initialization of a variable. The variable is rounded to the nearest legal fixed number (i.e. a multiple of $2^b = \text{Fixed_precision_nt}::\text{unit_value}()$)

```
Fixed_precision_nt fvar( fval);
```

Declaration and initialization.

```
Fixed_precision_nt fvar( int i);
```

Declaration and initialization with an integer.

Operations

```
Fixed_precision_nt &
```

```
    fvar = fval
```

Assignment.

```
bool    is_valid( fval)
```

In case of overflow or division by 0, numbers becomes invalid. If the precision is changed by usage of *Fixed_precision_nt::init()*, already existing numbers may become invalid if they are no longer multiple of 2^b .

```
bool    is_finite( fval)
```

do not implement infinite numbers. *is_finite* is identical to *is_valid*.

The comparison operations $==$, $!=$, $<$, $>$, $<=$, and $>=$ are all available.

The arithmetic operators $+$, $-$, $*$, $/$, $+=$, $-=$, $*=$ and $/=$ are all available. The result of the computation is rounded to the nearest legal . Overflow is possible, and even probable in case of multiplication or division. are designed to use specialized predicates, not to use arithmetic.

double *to_double(fval)* casts to *double*.

Precision initialization

As mentioned before, the *numbers* takes their values in an interval $[-2^{24+b}, 2^{24+b}]$ of multiples of 2^b , this number b as to be defined before any use of *Fixed_precision_nt*.

static bool *init(float B)* B is an upper bound on the data, b is the smallest integer such that $|B| \leq 2^b$. The result of the function is false if initialization was already done, in that case already existing may become invalid.

static float *unit_value()* returns 2^b .

static float *upper_bound()* returns 2^{24+b} .

Perturbation scheme

implements perturbation scheme as described by Alliez, Devillers and Snoeyink [ADS98]. The perturbation mode can be activated or deactivated for different kinds of perturbations. The default mode is no perturbation.

static void *perturb_incircle()* Activate. *side_of_oriented_circle* predicate of 4 cocircular points answers degenerate only if the 4 points are collinear.

static void *unperturb_incircle()* Deactivate

static bool *is_perturbed_incircle()* returns current mode

static void *perturb_insphere()* Activate. *side_of_oriented_sphere* predicate of 5 cospherical points answers degenerate only if the 5 points are coplanar.

static void *unperturb_insphere()* Deactivate

static bool *is_perturbed_insphere()* returns current mode

Geometric predicates

Through overloading mechanisms, functions such that *orientation* for *Point_2<Cartesian< Fixed_precision_nt>* > will correctly call the function below.

Orientation *orientationC2(x0, y0, x1, y1, x2, y2)*
Oriented_side *side_of_oriented_circleC2(x0, y0, x1, y1, x2, y2, x3, y3)*

Perturbation mode can be activated.

Orientation *orientationC3(x0, y0, z0, x1, y1, z1, x2, y2, z2, x3, y3, z3)*
Oriented_side *side_of_oriented_sphereC3(x0, y0, z0, x1, y1, z1, x2, y2, z2, x3, y3, z3, x4, y4, z4)*

Perturbation mode can be activated.

CGAL::gcd

Definition

The function *gcd* computes the greatest common divisor of two values.

#include <CGAL/number_utils.h>

NT *gcd*(*NT nval1*, *NT nval2*)

computes the greatest common divisor of two numbers. If *nval1* is 0, the function returns *nval2*.
Precondition: : *nval2* is not zero.

Implementation

Uses Euclid's algorithm.

CGAL::Gcd<NT>

Definition

The function *Gcd<NT>* computes the greatest common divisor of two values.

#include <CGAL/number_utils_classes.h>

Is Model for the Concepts

AdaptableFunctor.....page [2656](#)

NT *f(NT nval1, NT nval2)*

computes the greatest common divisor of two numbers. If *nval1* is 0, the function returns *nval2*.
Precondition: : *nval2* is not zero.

Implementation

Uses Euclid's algorithm.

mpq_class

Definition

The class *mpq_class* is an exact multiprecision rational number type, provided by GMP. CGAL provides the necessary functions to make it compliant to the number type concept.

```
#include <CGAL/gmpxx.h>
```

Is Model for the Concepts

FieldNumberType

See the GMP documentation for additionnal details.

mpz_class

Definition

The class *mpz_class* is an exact multiprecision integer number type, provided by GMP. CGAL provides the necessary functions to make it compliant to the number type concept.

```
#include <CGAL/gmpxx.h>
```

Is Model for the Concepts

RingNumberType

See the GMP documentation for additionnal details.

CGAL::Gmpq

Definition

An object of the class *Gmpq* is an arbitrary precision rational number based on the GNU Multiple Precision Arithmetic Library.

```
#include <CGAL/Gmpq.h>
```

Is Model for the Concepts

FieldNumberType

Types

Gmpq::NT the field type, which is *Gmpz*.

Creation

Gmpq *q*; creates an uninitialized multiple precision rational number *q*.

Gmpq *q*(*int* *i*); creates a multiple-precision rational number initialized with *i*.

Gmpq *q*(*Gmpz* *n*); creates a multiple-precision rational number initialized with *n*.

Gmpq *q*(*int* *n*, *int* *d*); creates a multiple-precision rational number initialized with *n/d*.

Gmpq *q*(*signed long* *n*, *unsigned long* *d*); creates a multiple-precision rational number initialized with *n/d*.

Gmpq *q*(*unsigned long* *n*, *unsigned long* *d*); creates a multiple-precision rational number initialized with *n/d*.

Gmpq *q*(*Gmpz* *n*, *Gmpz* *d*); creates a multiple-precision rational number initialized with *n/d*.

Gmpq *q*(*double* *d*); creates a multiple-precision rational number initialized with *d*.

Gmpq *q*(*std::string* *str*); creates a multiple-precision rational number initialized with *str*.

Gmpq *q*(*std::string* *str*, *int* *base*); creates a multiple-precision rational number initialized with *str*.

Operations

There are two access functions, namely to the numerator and the denominator of a rational. Note that these values are not uniquely defined. It is guaranteed that *q.numerator()* and *q.denominator()* return values *nt_num* and *nt_den* such that $q = nt_num/nt_den$, only if *q.numerator()* and *q.denominator()* are called consecutively wrt *q*, i.e. *q* is not involved in any other operation between these calls.

<i>Gmpz</i>	<i>q.numerator()</i>	returns the numerator of <i>q</i> .
<i>Gmpz</i>	<i>q.denominator()</i>	returns the denominator of <i>q</i> .

Implementation

Gmpqs are reference counted.

CGAL::Gmpz

Definition

An object of the class *Gmpz* is an arbitrary precision integer based on the GNU Multiple Precision Arithmetic Library.

```
#include <CGAL/Gmpz.h>
```

Is Model for the Concepts

EuclideanRingNumberType

Creation

<i>Gmpz</i> <i>q</i> ;	creates an uninitialized multiple precision integer <i>q</i> .
<i>Gmpz</i> <i>q</i> (<i>int</i> <i>i</i>);	creates a multiple-precision integer initialized with <i>i</i> .
<i>Gmpz</i> <i>q</i> (<i>double</i> <i>d</i>);	creates a multiple-precision integer initialized with the integral part of <i>d</i> .

Implementation

Gmpzs are reference counted.

CGAL::Interval_nt<Protected>

Definition

This section describes briefly what interval arithmetic is, its implementation in CGAL, and its possible use by geometric programs. The main reason for having interval arithmetic in CGAL is its integration into the filtered robust and fast predicates scheme, but we also provide a number type so that you can use it separately if you find any use for it, such as interval analysis, or to represent data with tolerance...

The purpose of interval arithmetic is to provide an efficient way to bound the roundoff errors made by floating point computations. You can choose the behaviour of your program depending on these errors; that is what is done for the filtered robust predicates (see Section 44.9). You can find more theoretical information on this topic in [BBP01].

Interval arithmetic is a large concept and we will only consider here a simple arithmetic based on intervals whose bounds are *doubles*. So each variable is an interval representing any value inside the interval. All arithmetic operations (+, -, *, /, $\sqrt{}$, *square()*, *min()*, *max()* and *abs()*) on intervals preserve the inclusion. This property can be expressed by the following formula (*x* and *y* are reals, *X* and *Y* are intervals, *OP* is an arithmetic operation):

$$\forall x \in X, \forall y \in Y, (x \text{ OP } y) \in (X \text{ OP } Y)$$

For example, if the final result of a sequence of arithmetic operations is an interval that does not contain zero, then you can safely determine its sign.

```
#include <CGAL/Interval_nt.h>
```

Parameters

The template parameter *Protected* is a boolean parameter, which defaults to *true*. It provides a way to select faster computations by avoiding rounding mode switches, at the expense of more care to be taken by the user (see below). The default value, *true*, is the safe way, and takes care of proper rounding mode changes. When specifying *false*, the user has to take care about positioning the rounding mode towards plus infinity before doing any computations with the interval class.

Is Model for the Concepts

SqrtFieldNumberType

Types

The class *Interval_nt* defines the following types:

<i>typedef double</i>	<i>value_type;</i>	The type of the bounds of the interval.
<i>typedef std::exception</i>	<i>unsafe_comparison;</i>	

The type of the exceptions raised when uncertain comparisons are performed.

Interval_nt<Protected>:: *Protector*

A type whose default constructor and destructor allow to protect a block of code from FPU rounding modes necessary for the computations with *Interval_nt*<false>. It does nothing for *Interval_nt*<true>.

Creation

<i>Interval_nt</i> <Protected> <i>I</i> (int <i>i</i>);	introduces the interval [<i>i</i> ; <i>i</i>].
<i>Interval_nt</i> <Protected> <i>I</i> (double <i>d</i>);	introduces the interval [<i>d</i> ; <i>d</i>].
<i>Interval_nt</i> <Protected> <i>I</i> (double <i>i</i> , double <i>s</i>);	introduces the interval [<i>i</i> ; <i>s</i>].
<i>Interval_nt</i> <Protected> <i>I</i> (std::pair<double, double> <i>p</i>);	introduces the interval [<i>p.first</i> ; <i>p.second</i>].

Operations

All functions required by a class to be considered as a CGAL number type (see 44) are present, as well as the utility functions, sometimes with a particular semantic which is described below. There are also a few additional functions.

<i>Interval_nt</i> <i>I</i> / <i>Interval_nt</i> <i>J</i>	returns $[-\infty; +\infty]$ when the denominator contains 0.
<i>Interval_nt</i> <i>sqrt</i> (<i>Interval_nt</i> <i>I</i>)	returns $[0; \sqrt{\text{upper_bound}(I)}]$ when only the lower bound is negative (expectable case with roundoff errors), and is unspecified when the upper bound also is negative (unexpected case).
double <i>to_double</i> (<i>Interval_nt</i> <i>I</i>)	returns the middle of the interval, as a double approximation of the interval.
double <i>I.inf</i> ()	returns the lower bound of the interval.
double <i>I.sup</i> ()	returns the upper bound of the interval.
bool <i>I.is_point</i> ()	returns whether both bounds are equal.
bool <i>I.is_same</i> (<i>Interval_nt</i> <i>J</i>)	returns whether both intervals have the same bounds.
bool <i>I.do_overlap</i> (<i>Interval_nt</i> <i>J</i>)	returns whether both intervals have a non empty intersection.

The comparison operators (<, >, <=, >=, ==, !=, *sign()* and *compare()*) have the following semantic: it is the intuitive one when for all couples of values in both intervals, the comparison is identical (case of non-overlapping intervals). This can be expressed by the following formula (*x* and *y* are reals, *X* and *Y* are intervals, *OP* is a comparison operator):

$$(\forall x \in X, \forall y \in Y, (x \text{ OP } y) = \text{true}) \Rightarrow (X \text{ OP } Y) = \text{true}$$

and

$$(\forall x \in X, \forall y \in Y, (x \text{ OP } y) = \text{false}) \Rightarrow (X \text{ OP } Y) = \text{false}$$

Otherwise, the comparison is not safe, and we specify this by returning a type encoding this uncertainty, namely using *Uncertain*, which can be probed for uncertainty by hand, and which has a conversion to the normal type (e.g. *bool*) which throws an exception when the conversion is not certain. Note that each failed conversion increments the counter *number_of_failures()*, and then throw the exception of type *unsafe_comparison*.

<i>Uncertain</i> < <i>bool</i> >	<i>Interval_nt i < Interval_nt j</i>
<i>Uncertain</i> < <i>bool</i> >	<i>Interval_nt i > Interval_nt j</i>
<i>Uncertain</i> < <i>bool</i> >	<i>Interval_nt i <= Interval_nt j</i>
<i>Uncertain</i> < <i>bool</i> >	<i>Interval_nt i >= Interval_nt j</i>
<i>Uncertain</i> < <i>bool</i> >	<i>Interval_nt i == Interval_nt j</i>
<i>Uncertain</i> < <i>bool</i> >	<i>Interval_nt i != Interval_nt j</i>
<i>Uncertain</i> < <i>Comparison_result</i> >	<i>compare(Interval_nt i, Interval_nt j)</i>
<i>Uncertain</i> < <i>Sign</i> >	<i>sign(Interval_nt i)</i>

<i>static unsigned</i>	<i>number_of_failures()</i>
------------------------	-----------------------------

Returns a global counter incremented at each conversion which thrown an exception.

<i>typedef Interval_nt<false></i>	<i>Interval_nt_advanced;</i>
-----------------------------------------	------------------------------

This typedef (at namespace CGAL scope) exists for backward compatibility, as well as removing the need to remember the boolean value for the template parameter.

— *advanced* —

Implementation

The operations on *Interval_nt* with the default parameter *true*, are automatically protected against rounding modes, and are thus slower than those on *Interval_nt_advanced*, but easier to use. Users that need performance are encouraged to use *Interval_nt_advanced* instead.

Changing the rounding mode affects all floating point computations, and might cause problems with parts of your code, or external libraries (even CGAL), that expect the rounding mode to be the default (round to the nearest).

We provide two interfaces to change the rounding mode. The first one is to use a protector object whose default constructor and destructor will take care of changing the rounding mode.

The second one is the following detailed set of functions :

<i>typedef int</i>	<i>FPU_CW_t;</i>
--------------------	------------------

The type used by the following functions to deal with rounding modes. This is usually an *int*.

<i>void</i>	<i>FPU_set_cw(FPU_CW_t R)</i>
-------------	--------------------------------

sets the rounding mode to *R*.

FPU_CW_t

FPU_get_cw(void)

returns the current rounding mode.

FPU_CW_t

FPU_get_and_set_cw(FPU_CW_t R)

sets the rounding mode to *R* and returns the old one.

The macros *CGAL_FE_TONEAREST*, *CGAL_FE_TOWARDZERO*, *CGAL_FE_UPWARD* and *CGAL_FE_DOWNWARD* are the values corresponding to the rounding modes.

Example

Protecting an area of code that uses operations on the class *Interval_nt_advanced* can be done in the following way:

```
{
    Interval_nt_advanced::Protector P;
    ... // The code to be protected.
}
```

The basic idea is to use the directed rounding modes specified by the *IEEE 754* standard, which are implemented by almost all processors nowadays. It states that you have the possibility, concerning the basic floating point operations (+, −, *, /, √) to specify the rounding mode of each operation instead of using the default, which is set to 'round to the nearest'. This feature allows us to compute easily on intervals. For example, to add the two intervals [a.i;a.s] and [b.i;b.s], compute $c.i = a.i + b.i$ rounded towards minus infinity, and $c.s = a.s + b.s$ rounded towards plus infinity, and the result is the interval [c.i;c.s]. This method can be extended easily to the other operations.

The problem is that we have to change the rounding mode very often, and the functions of the C library doing this operation are slow and not portable. That's why assembly versions are used as often as possible. Another trick is to store the opposite of the lower bound, instead of the lower bound itself, which allows us to never change the rounding mode inside simple operations. Therefore, all basic operations, which are in the class *Interval_nt_advanced* assume that the rounding mode is set to 'round to infinity', and everything works with this correctly set.

So, if the user needs the speed of *Interval_nt_advanced*, he must take care of setting the rounding mode to 'round to infinity' before each block of operations on this number type. And if other operations might be affected by this, he must take care to reset it to 'round to the nearest' before they are executed.

Notes:

- On Intel platforms (with any operating system and compiler), due to a misfeature of the floating point unit, which does not handle exactly IEEE compliant operations on doubles, we are forced to use a workaround which slows down the code, but is only useful when the intervals can overflow or underflow. If you know that the intervals will never overflow nor underflow for your code, then you can disable this workaround with the flag *CGAL_IA_NO_X86_OVER_UNDER_FLOW_PROTECT*. Other platforms are not affected by this flag.

- When optimizing, compilers usually propagate the value of variables when they know it's a constant. This can break the interval routines because the compiler then does some floating point operations on these constants with the default rounding mode, which is wrong. This kind of problem is avoided by stopping constant propagation in the interval routines. However, this solution slows down the code and is rarely useful, so you can disable it by setting the flag `CGAL_IA_DONT_STOP_CONSTANT_PROPAGATION`.

└──────── *advanced* ─────────┘

CGAL::is_negative

Definition

The function *is_negative* determines if a value is negative or not.

```
#include <CGAL/number_utils.h>
```

```
bool is_negative( NT nval)
```

CGAL::is_one

Definition

The function *is_one* determines if a value is equal to 1 or not.

```
#include <CGAL/number_utils.h>
```

```
bool is_one( NT ntval)
```

CGAL::is_positive

Definition

The function *is_positive* determines if a value is positive or not.

```
#include <CGAL/number_utils.h>
```

```
bool is_positive( NT nval)
```

CGAL::is_zero

Definition

The function *is_zero* determines if a value is equal to 0 or not.

```
#include <CGAL/number_utils.h>
```

```
bool is_zero( NT ntval)
```


CGAL::Is_negative<NT>

Definition

The function *Is_negative*<*NT*> determines if a value is negative or not.

#include <*CGAL/number_utils_classes.h*>

Is Model for the Concepts

AdaptableFunctor.....page [2656](#)

bool *f(NT nval)* returns *true* if *nval* is negative and *false* otherwise.

CGAL::Is_one<NT>

Definition

The function *Is_one*<*NT*> determines if a value is equal to 1 or not.

#include <*CGAL/number_utils_classes.h*>

Is Model for the Concepts

AdaptableFunctor.....page [2656](#)

bool *f(NT nval)* returns *true* if *nval* is 1 and *false* otherwise.

CGAL::Is_positive<NT>

Definition

The function *Is_positive*<*NT*> determines if a value is positive or not.

#include <*CGAL/number_utils_classes.h*>

Is Model for the Concepts

AdaptableFunctor.....page [2656](#)

bool *f*(*NT nval*) returns *true* if *nval* is positive and *false* otherwise.

CGAL::Is_zero<NT>

Definition

The function *Is_zero*<NT> determines if a value is equal to 0 or not.

#include <CGAL/number_utils_classes.h>

Is Model for the Concepts

AdaptableFunctor.....page [2656](#)

bool *f(NT nval)* returns *true* if *nval* is 0 and *false* otherwise.

CGAL::Lazy_exact_nt<NT>

Definition

An object of the class *Lazy_exact_nt<NT>* is able to represent any number which *NT* is able to represent. The arithmetic operations it can do are those *NT* can do, limited to the 4 basic operations, the square root and the comparisons. The idea is that *Lazy_exact_nt<NT>* works exactly like *NT*, except that it is faster because it tries to only compute an approximation of the value, and only refers to *NT* when needed. The goal is to speed up exact computations done by any exact but slow number type *NT*.

In addition to the filtering at each arithmetic operation, the predicates are overloaded in the same way as for *Filtered_exact*, so you get the additional speed up without requiring to encapsulate *Lazy_exact_nt<NT>* into *Filtered_exact*.

```
#include <CGAL/Lazy_exact_nt.h>
```

Is Model for the Concepts

FieldNumberType

Creation

<i>Lazy_exact_nt<NT></i> <i>m</i> ;	introduces an uninitialized variable <i>m</i> .
<i>Lazy_exact_nt<NT></i> <i>m</i> (<i>int i</i>);	introduces the integral value <i>i</i> .
<i>Lazy_exact_nt<NT></i> <i>m</i> (<i>double d</i>);	introduces the floating point value <i>d</i> (works only if <i>NT</i> has a constructor from a double too).
<i>Lazy_exact_nt<NT></i> <i>m</i> (<i>NT n</i>);	introduces the value <i>n</i> .
<i>template <class NT1></i> <i>Lazy_exact_nt<NT></i> <i>m</i> (<i>Lazy_exact_nt<NT1></i> <i>n</i>);	introduces the value <i>n</i> . <i>NT1</i> needs to be convertible to <i>NT</i> (and this conversion will only be done if necessary).

Operations

<i>NT</i>	<i>m.exact()</i>	returns the corresponding <i>NT</i> value.
<i>Interval_nt<true></i>	<i>m.approx()</i>	returns an interval containing the exact value.
<i>Interval_nt<false></i>	<i>m.interval()</i>	returns an interval containing the exact value.
<i>static void</i>	<i>m.set_relative_precision_of_to_double(double d)</i>	specifies the relative precision that <i>to_double()</i> has to fulfill. The default value is 10^{-5} . <i>Precondition:</i> $d \geq 0$ and $d \leq 1$.

static double *m.get_relative_precision_of_to_double()*

returns the relative precision that *to_double()* currently fulfills.

std::ostream& *std::ostream& out << m*

writes *m* to ostream *out* in an interval format.

std::istream& *std::istream& in >> & m*

reads a *NT* from *in*, then converts it to a *Lazy_exact_nt<NT>*.

Example

```
#include <CGAL/Cartesian.h>
#include <CGAL/MP_Float.h>
#include <CGAL/Lazy_exact_nt.h>
#include <CGAL/Quotient.h>

typedef CGAL::Lazy_exact_nt<CGAL::Quotient<CGAL::MP_Float> > NT;
typedef CGAL::Cartesian<NT> K;
```

leda_bigfloat

Definition

The class *leda_bigfloat* is a wrapper class that provides the functions needed to use the number type *bigfloat*, representing multiprecision floating point numbers provided by LEDA.

```
#include <CGAL/leda_bigfloat.h>
```

Is Model for the Concepts

FieldNumberType

For more details on the number types of LEDA we refer to the LEDA manual [[MNSU](#)].

leda_integer

Definition

The class *leda_integer* is a wrapper class that provides the functions needed to use the number type *integer*, representing exact multiprecision integers provided by LEDA.

```
#include <CGAL/leda_integer.h>
```

Is Model for the Concepts

RingNumberType

For more details on the number types of LEDA we refer to the LEDA manual [[MNSU](#)].

leda_rational

Definition

The class *leda_rational* is a wrapper class that provides the functions needed to use the number type *rational*, representing exact multiprecision rational numbers provided by LEDA.

```
#include <CGAL/leda_rational.h>
```

Is Model for the Concepts

FieldNumberType

For more details on the number types of LEDA we refer to the LEDA manual [[MNSU](#)].

leda_real

Definition

The class *leda_real* is a wrapper class that provides the functions needed to use the number type *real*, representing exact real numbers numbers provided by LEDA.

```
#include <CGAL/leda_real.h>
```

Is Model for the Concepts

FieldNumberType

For more details on the number types of LEDA we refer to the LEDA manual [[MNSU](#)].

CGAL::make_root_of_2

Definition

The function *make_root_of_2* returns the root of a polynomial of degree 2 given its 3 coefficients and a boolean value selecting the smaller or larger root. The type of the arguments must be a model of the *RingNumberType* concept.

```
#include <CGAL/make_root_of_2.h>
```

```
template <typename RT>
Root_of_traits<RT>::RootOf_2
```

```
make_root_of_2( RT a, RT b, RT c, bool s)
```

Returns the root of the polynomial $aX^2 + bX + c$ which is the smallest if s is true, or the largest if s is false. If RT supports a *sqrt* operation, then the usual formula is used. Otherwise the *Root_of_2* class is used.

Precondition: a must be non-zero, and the polynomial must have a root.

See Also

RootOf_2 page [2580](#)
CGAL::Root_of_2<RT> page [2582](#)

CGAL::max

Definition

The function *max* returns the larger of two values.

#include <CGAL/number_utils.h>

NT *max*(*NT ntval1*, *NT ntval2*)

CGAL::min

Definition

The function *min* returns the smaller of two values.

```
#include <CGAL/number_utils.h>
```

```
NT min( NT nval1, NT nval2)
```

CGAL::Max<NT,Compare>

Definition

The function object class *Max*<*NT*,*Compare*> returns the larger of two values. The default value for *Compare* is *std::less*.

#include <CGAL/number_utils_classes.h>

Is Model for the Concepts

AdaptableFunctor.....page [2656](#)

NT *f*(*NT* *ntval1*, *NT* *ntval2*)

returns the larger of *ntval1* and *ntval2*.

CGAL::Min<NT,Compare>

Definition

The function object class *Min*<*NT*,*Compare*> returns the smaller of two values. The default value for *Compare* is *std::less*.

#include <*CGAL/number_utils_classes.h*>

Is Model for the Concepts

AdaptableFunctor.....page [2656](#)

NT *f*(*NT* *ntval1*, *NT* *ntval2*)

returns the smaller of *ntval1* and *ntval2*.

CGAL::MP_Float

Definition

An object of the class *MP_Float* is able to represent a floating point value with arbitrary precision. This number type has the property that additions, subtractions and multiplications are computed exactly, as well as the construction from *float*, *double* and *long double*.

Division and square root are not enabled by default since CGAL release 3.2, since they are computed approximately. We suggest that you use rationals like *Quotient<MP_Float>* when you need exact divisions. To enable division and square root, you have to define the preprocessor macro *CGAL_MP_FLOAT_ALLOW_INEXACT*.

Note on the implementation : although the mantissa length is basically only limited by the available memory, the exponent is currently represented by a (integral valued) *double*, which can overflow in some circumstances. We plan to also have a multiprecision exponent to fix this issue.

```
#include <CGAL/MP_Float.h>
```

Is Model for the Concepts

RingNumberType or *SqrtFieldNumberType* if *CGAL_MP_FLOAT_ALLOW_INEXACT* is set.

Creation

<i>MP_Float</i> <i>m</i> ;	introduces an uninitialized variable <i>m</i> .
<i>MP_Float</i> <i>m</i> (<i>MP_Float</i>);	copy constructor.
<i>MP_Float</i> <i>m</i> (int <i>i</i>);	introduces the integral value <i>i</i> .
<i>MP_Float</i> <i>m</i> (float <i>d</i>);	introduces the floating point value <i>d</i> (exact conversion).
<i>MP_Float</i> <i>m</i> (double <i>d</i>);	introduces the floating point value <i>d</i> (exact conversion).
<i>MP_Float</i> <i>m</i> (long double <i>d</i>);	introduces the floating point value <i>d</i> (exact conversion).

Operations

<i>std::ostream&</i>	<i>std::ostream& out</i> << <i>m</i>	writes a double approximation of <i>m</i> to the ostream <i>out</i> .
--------------------------	------------------------------------------	-----------------------------------------------------------------------

<i>std::istream&</i>	<i>std::istream& in</i> >> & <i>m</i>	reads a <i>double</i> from <i>in</i> , then converts it to an <i>MP_Float</i> .
--------------------------	-------------------------------------------	---------------------------------------------------------------------------------

<i>MP_Float</i>	<i>approximate_division</i> (<i>a</i> , <i>b</i>)	computes an approximation of the division by converting the operands to <i>double</i> , performing the division on <i>double</i> , and converting back to <i>MP_Float</i> .
-----------------	-----------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------

MP_Float *approximate_sqrt(a)* computes an approximation of the square root by converting the operand to *double*, performing the square root on *double*, and converting back to *MP_Float*.

Implementation

The implementation of *MP_Float* is simple but provides a quadratic complexity for multiplications. This can be a problem for large operands. For faster implementations of the same functionality with large integral values, you may want to consider using *GMP* or *LEDA* instead.

CGAL::Number_type_checker<NT1,NT2,Comparator>

Definition

Number_type_checker is a number type whose instances store two numbers of types *NT1* and *NT2*. It forwards all arithmetic operations to them, and calls the binary predicate *Comparator* to check the equality of the instances after each modification, as well as for each comparison.

This is a debugging tool which is useful when dealing with number types.

Parameters

NT1 and *NT2* must be models of some number type concept. *Comparator* has to be a model of a binary predicate taking *NT1* as first argument, and *NT2* as second. The *Comparator* parameter has a default value which is a functor calling *operator==* between the two arguments.

```
#include <CGAL/Number_type_checker.h>
```

Is Model for the Concepts

SqrtFieldNumberType

Creation

```
Number_type_checker<NT1,NT2,Comparator> c;
```

introduces an uninitialized variable *c*.

```
Number_type_checker<NT1,NT2,Comparator> c( int i);
```

introduces the integral value *i*.

```
Number_type_checker<NT1,NT2,Comparator> c( double d);
```

introduces the floating point value *d*.

```
Number_type_checker<NT1,NT2,Comparator> c( NT1 n1, NT2 n2);
```

introduces a variable storing the pair *n1*, *n2*.

Operations

Some operations have a particular behavior documented here.

<i>NT1</i>	<i>c.n1()</i>	returns a const reference to the object of type <i>NT1</i> .
<i>NT2</i>	<i>c.n2()</i>	returns a const reference to the object of type <i>NT2</i> .
<i>NT1</i> &	<i>c.n1()</i>	returns a reference to the object of type <i>NT1</i> .
<i>NT2</i> &	<i>c.n2()</i>	returns a reference to the object of type <i>NT2</i> .
<i>bool</i>	<i>c.is_valid()</i>	calls the <i>Comparator</i> binary predicate on the two stored objects and returns its result.

double *to_double(Number_type_checker c)*

returns *to_double(c.nI())*. No particular check is made here.

std::pair<double, double>

to_interval(Number_type_checker c)

returns *to_interval(c.nI())*. No particular check is made here.

std::ostream& *std::ostream& out << Number_type_checker c*

writes *c.nI()* to the ostream *out*.

std::istream& *std::istream& in >> Number_type_checker& c*

reads an *NT1* from *in*, then converts it to an *NT2*, so a conversion from *NT1* to *NT2* is required here.

CGAL::Number_type_traits<NT>

Definition

The class *Number_type_traits<NT>* can be used to determine if a certain number type supports certain operations and thus to determine which number type concept it is a model of. It also specifies if the number type supports some operations exactly.

```
#include <CGAL/Number_type_traits.h>
```

Types

The following types are each set to either *CGAL::Tag_true* or *CGAL::Tag_false* depending on whether the indicated operation is supported or not by the number type *NT*.

Number_type_traits<NT>::Has_gcd indicates if the number type *NT* supports the *gcd* operation or not.

Number_type_traits<NT>::Has_division indicates if the number type *NT* has *operator/* defined or not.

Number_type_traits<NT>::Has_sqrt indicates if the number type *NT* has *sqrt* defined or not.

Number_type_traits<NT>::Has_exact_ring_operations indicates if the number type *NT* supports ring operations exactly.

Number_type_traits<NT>::Has_exact_division indicates if the number type *NT* supports divisions exactly.

Number_type_traits<NT>::Has_exact_sqrt indicates if the number type *NT* supports the operation *sqrt* exactly.

CGAL::Quotient<NT>

Definition

An object of the class *Quotient<NT>* is an element of the field of quotients of the integral domain type *NT*. If *NT* behaves like an integer, *Quotient<NT>* behaves like a rational number. LEDA's class *rational* (see Section 44.7) has been the basis for *Quotient<NT>*. A *Quotient<NT>* *q* is represented as a pair of *NT*s, representing numerator and denominator.

```
#include <CGAL/Quotient.h>
```

Is Model for the Concepts

FieldNumberType

Creation

<i>Quotient<NT></i> <i>q</i> ;	introduces an uninitialized variable <i>q</i> .
<i>template <class T></i> <i>Quotient<NT></i> <i>q</i> (<i>T</i> <i>t</i>);	introduces the quotient <i>t/1</i> . <i>NT</i> needs to have a constructor from <i>T</i> .
<i>template <class T></i> <i>Quotient<NT></i> <i>q</i> (<i>Quotient<T></i> <i>t</i>);	introduces the quotient <i>NT(t.numerator())/NT(t.denominator())</i> . <i>NT</i> needs to have a constructor from <i>T</i> .
<i>Quotient<NT></i> <i>q</i> (<i>NT</i> <i>n</i> , <i>NT</i> <i>d</i>);	introduces the quotient <i>n/d</i> .

Operations

There are two access functions, namely to the numerator and the denominator of a quotient. Note that these values are not uniquely defined. It is guaranteed that *q.numerator()* and *q.denominator()* return values *nt_num* and *nt_den* such that *q = nt_num/nt_den*, only if *q.numerator()* and *q.denominator()* are called consecutively wrt *q*, i.e. *q* is not involved in any other operation between these calls.

<i>NT</i>	<i>q.numerator()</i>	returns a numerator of <i>q</i> .
<i>NT</i>	<i>q.denominator()</i>	returns a denominator of <i>q</i> .

The stream operations are available as well. They assume that corresponding stream operators for type *NT* exist.

<i>std::ostream&</i>	<i>std::ostream& out << q</i>	writes <i>q</i> to ostream <i>out</i> in format “n/d”, where <i>n</i> == <i>q.numerator()</i> and <i>d</i> == <i>q.denominator()</i> .
<i>std::istream&</i>	<i>std::istream& in >> & q</i>	reads <i>q</i> from istream <i>in</i> . Expected format is “n/d”, where <i>n</i> and <i>d</i> are of type <i>NT</i> . A single <i>n</i> which is not followed by a / is also accepted and interpreted as <i>n/1</i> .

The following functions are added to fulfill the CGAL requirements on number types.

<i>double</i>	<i>to_double(q)</i>	returns some double approximation to <i>q</i> .
<i>bool</i>	<i>is_valid(q)</i>	returns true, if numerator and denominator are valid.

bool *is_finite(q)*
Quotient<NT>

returns true, if numerator and denominator are finite.

sqrt(q)

returns the square root of q . This is supported if and only if NT supports the square root as well.

CGAL::Rational_traits<NT>

Definition

The class *Rational_traits<NT>* can be used to determine the type of the numerator and denominator of a rational number type as *Quotient*, *Gmpq*, *mpq_class* or *leda_rational*.

```
#include <CGAL/Number_type_traits.h>
```

Types

Rational_traits<NT>::RT the type of the numerator and denominator.

Operations

RT *t.numerator(NT r)* returns the numerator of *r*.

RT *t.denominator(NT r)* returns the denominator of *r*.

NT *t.make_rational(RT n, RT d)*
constructs a rational number.

NT *t.make_rational(NT n, NT d)*
constructs a rational number.

RingNumberType

Definition

The concept `RingNumberType` defines the syntactic requirements a number type must meet in order to be used in CGAL as a ring type. This implies that `CGAL::Number_type_traits<RingNumberType>::Has_division` is not required to be `CGAL::Tag_true`. Unsigned numbers are excluded due to semantical limitations in the ordering.

Refines

CopyConstructible, Assignable

Creation

`RingNumberType n1;`

Declaration of a variable.

`RingNumberType n1(int i);`

Declaration and initialization with a small integer constant i , $0 \leq i \leq 127$. The neutral elements for addition (zero) and multiplication (one) are needed quite often, but sometimes other small constants are useful too. The value 127 was chosen such that even signed 8 bit number types can fulfill this condition.

Operations

The comparison operators need to be provided.

```
bool      n1 == n2
bool      n1 != n2
bool      n1 < n2
bool      n1 > n2
bool      n1 <= n2
bool      n1 >= n2
```

In addition, the comparisons with small values of type `int` are also required.

```
bool      int n1 == n2
bool      int n1 != n2
bool      int n1 < n2
bool      int n1 > n2
bool      int n1 <= n2
bool      int n1 >= n2
bool      n1 == int n2
bool      n1 != int n2
bool      n1 < int n2
bool      n1 > int n2
bool      n1 <= int n2
bool      n1 >= int n2
```


The arithmetic operators for the addition, subtraction and multiplication are required as well.

<i>RingNumberType</i>	$n1 + n2$
<i>RingNumberType</i>	$n1 - n2$
<i>RingNumberType</i>	$n1 * n2$
<i>RingNumberType</i>	$-n$
<i>RingNumberType</i>	$n1 += n2$
<i>RingNumberType</i>	$n1 -= n2$
<i>RingNumberType</i>	$n1 *= n2$

And similarly, the mixed operators with small values of type *int* are also required.

<i>RingNumberType</i>	$int\ n1 + n2$
<i>RingNumberType</i>	$int\ n1 - n2$
<i>RingNumberType</i>	$int\ n1 * n2$
<i>RingNumberType</i>	$n1 + int\ n2$
<i>RingNumberType</i>	$n1 - int\ n2$
<i>RingNumberType</i>	$n1 * int\ n2$
<i>RingNumberType</i>	$n1 += int\ n2$
<i>RingNumberType</i>	$n1 -= int\ n2$
<i>RingNumberType</i>	$n1 *= int\ n2$

The following accessory functions are needed for special purposes :

<i>bool</i>	<i>is_valid(n)</i>	Not all values of a number type need be valid. The function <i>is_valid</i> checks this. For example, an expression like $NT(0)/NT(0)$ can result in an invalid number. Routines may have as a precondition that all numerical values are valid.
<i>bool</i>	<i>is_finite(n)</i>	When two large values are multiplied, the result may not fit in a <i>NT</i> . Some number types (e.g. the standard <i>float</i> and <i>double</i> types) have a way to represent a too big value as infinity. <i>is_finite</i> implies <i>is_valid</i> .
<i>double</i>	<i>to_double(n)</i>	gives the double value for a number type. This is usually an approximation for the real (stored) value. It can be used to send numbers to a renderer or to store them in a file.
<i>std::pair<double,double></i>	<i>to_interval(n)</i>	gives a double interval that encloses <i>n</i> .

Has Models

C++ built-in number types
CGAL::Filtered_exact<RingNumberType, ET>
CGAL::Fixed_precision_nt
CGAL::Gmpq
CGAL::Gmpz
CGAL::Interval_nt
CGAL::Interval_nt_advanced
CGAL::Lazy_exact_nt<RingNumberType>

CGAL::MP_Float
CGAL::Quotient<RingNumberType>
leda_integer
leda_rational
leda_bigfloat
leda_real

See Also

EuclideanRingNumberType page [2528](#)
FieldNumberType page [2530](#)
CGAL::Ring_tag page [2579](#)
Support Library Manual

CGAL::Ring_tag

Definition

The class *Ring_tag* is used as a tag in some algorithms. It indicates that a number type is to be considered as a model for `RingNumberType`. Only operations defined for `RingNumberType` are used. For example, no divisions are computed, even if the number type as such supports divisions.

```
#include <CGAL/Number_type_traits.h>
```

See Also

`RingNumberType` page [2576](#)
Euclidean_ring_tag page ??
Field_tag page ??
Sqrt_field_tag page ??

RootOf_2

Definition

Concept to represent algebraic numbers of degree up to 2 over a *RingNumberType* denoted as *RT*.

Creation

RootOf_2 *make_root_of_2*(*RT a*, *RT b*, *RT c*, *bool s*)

Returns the smaller (resp. larger) root of the equation $aX^2 + bX + c = 0$ if *s* is true (resp. false).

RootOf_2 *make_root_of_2*(*FT a*, *FT b*, *FT c*, *bool s*)

Returns the smaller (resp. larger) root of the equation $aX^2 + bX + c = 0$ if *s* is true (resp. false).

Operations

The comparison operators `==`, `!=`, `<`, `>`, `<=`, `>=` as well as the *sign* and *compare* functions need to be provided to compare elements of types *RootOf_2*, *RT* and *FT*.

In addition, the following operations must be provided:

RootOf_2 *RT a* + *RootOf_2 r*

RootOf_2 *FT a* + *RootOf_2 r*

RootOf_2 *RootOf_2 r* + *RT a*

RootOf_2 *RootOf_2 r* + *FT a*

and similarly for operators `-`, `*` and `/`.

RootOf_2 *square*(*RootOf_2 r*)

Has Models

double, *CGAL::Root_of_2*, etc

See Also

CGAL::make_root_of_2 page [2563](#)

CGAL::Root_of_2<*RT*> page [2582](#)

CGAL::Root_of_traits_2<*RT*> page [2581](#)

AlgebraicKernelForCircles::PolynomialForCircles_2_2 page [593](#)

AlgebraicKernelForCircles page [586](#)

CGAL::Root_of_traits_2<RT>

Definition

Associates types for algebraic numbers to RT , supposed to be a *RingNumberType*.

Types

Root_of_traits_2<RT>::Root_of_2

Model of *RootOf_2*.

CGAL::Root_of_2<RT>

#include <CGAL/Root_of_2.h>

Is Model for the Concepts

RootOf_2

See Also

CGAL::make_root_of_2 page [2563](#)

CGAL::sign

Definition

The function *sign* returns the sign of a number.

#include <CGAL/number_utils.h>

Sign *sign*(*NT nval*) returns the sign: *POSITIVE*, *ZERO*, or *NEGATIVE*.

CGAL::simplest_rational_in_interval

Definition

The function *simplest_rational_in_interval* computes the simplest rational number in an interval of two *double* values.

```
#include <CGAL/number_utils.h>
```

```
Rational simplest_rational_in_interval( double d1, double d2)
```

computes the rational number with the smallest denominator in the interval $[d1, d2]$.

Implementation

See Knuth, "Seminumerical algorithms", page 654, answer to exercise 4.53-39.

See Also

```
Rational CGAL::to_rational<Rational>(double d).
```


CGAL::sqrt

Definition

The function *sqrt* returns the square root of a number.

```
#include <CGAL/number_utils.h>
```

```
NT                                sqrt( NT ntval)
```

SqrtFieldNumberType

The concept `SqrtFieldNumberType` defines the syntactic requirements of a number type to be used as a template parameter for the Cartesian kernels. This number type supports the operations $+$, $-$, $*$, $/$, and $\sqrt{\cdot}$. This implies that both `CGAL::Number_type_traits<SqrtFieldNumberType>::Has_division` and `CGAL::Number_type_traits<SqrtFieldNumberType>::Has_sqrt` are `CGAL::Tag_true`.

Refines

`FieldNumberType`

Has Models

`float`

`double`

`CGAL::Filtered_exact<FieldNumberType, ET>`

`CGAL::Lazy_exact_nt<FieldNumberType>`

`leda_bigfloat`

`leda_real`

Operations

`SqrtFieldNumberType`

`sqrt(ntval)`

See Also

`RingNumberType` page [2576](#)

`EuclideanRingNumberType` page [2528](#)

`Kernel` page [35](#)

`CGAL::Sqrt_field_tag` page [2587](#)

Support Library Manual

CGAL::Sqrt_field_tag

Definition

The class *Sqrt_field_tag* is used as a tag in some algorithms. It indicates that a number type is to be considered as a model for *SqrtFieldNumberType*. All operations defined for *SqrtFieldNumberType* are used, including divisions and squareroot computations.

#include <CGAL/Number_type_traits.h>

See Also

SqrtFieldNumberType page [2586](#)
Ring_tag page ??
Euclidean_ring_tag page ??
Field_tag page ??

CGAL::square

Definition

The function *square* returns the square of a number.

```
#include <CGAL/number_utils.h>
```

```
NT          square( NT ntval)
```

CGAL::Sgn<NT>

Definition

The function object class *Sgn*<*NT*> computes the sign of a number. It is named *Sgn* to avoid a conflict with the type *Sign*.

```
#include <CGAL/number_utils_classes.h>
```

Is Model for the Concepts

AdaptableFunctor.....page [2656](#)

Sign *f*(*NT nval*) returns the sign: *POSITIVE*, *ZERO*, or *NEGATIVE*.

CGAL::Sqrt<NT>

Definition

The function object class *Sqrt*<*NT*> computes the square root of a number.

#include <*CGAL/number_utils.h*>

Is Model for the Concepts

AdaptableFunctor.....page [2656](#)

NT *f(NT nval)* returns the square root of *nval*.

CGAL::Square<NT>

Definition

The function *Square*<*NT*> computes the square root of a number.

#include <*CGAL/number_utils_classes.h*>

Is Model for the Concepts

AdaptableFunctor.....page [2656](#)

<i>NT</i>	<i>f(NT nval)</i>	returns the square root of <i>nval</i>
-----------	--------------------	----------------------------------------

CGAL::To_double<NT>

Definition

The function *To_double<NT>* computes an approximation of a number.

#include <CGAL/number_utils_classes.h>

Is Model for the Concepts

AdaptableFunctor.....page [2656](#)

double *f(NT ntval)* returns an approximation of *ntval*

CGAL::To_interval<NT>

Definition

The function *To_interval<NT>* computes an enclosing interval of a number.

```
#include <CGAL/number_utils_classes.h>
```

Is Model for the Concepts

AdaptableFunctor.....page [2656](#)

std::pair<double, double>

f(NT ntval) returns an enclosing interval of *ntval*

CGAL::to_rational

Definition

The function *to_rational* computes the rational number representing a given double precision floating point number.

```
#include <CGAL/number_utils.h>
```

Rational *to_rational*(*double d*) computes the rational number that equals *d*.

Implementation

See Also

CGAL::simplest_rational_in_interval<*Rational*>(*double d1*, *double d2*).

Chapter 45

STL Extensions for CGAL

Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Ron Wein

CGAL is designed in the spirit of the generic programming paradigm to work together with the Standard Template Library (STL) [C++98, Aus98]. This chapter documents non-geometric STL-like components that are not provided in the STL standard but in CGAL: a doubly-connected list managing items in place (where inserted items are not copied), a compact container, a multi-set class that uses three-valued comparisons and offers additional functionality, generic algorithms, iterators, functor adaptors for binding and swapping arguments and for composition, functors for projection and creation and adaptor classes around iterators and circulators. See also circulators in Chapter 46.

45.1 Doubly-Connected List Managing Items in Place

The class *In_place_list*<*T*,*bool*> manages a sequence of items in place in a doubly-connected list. Its goals are the flexible handling of memory management and performance optimization. The item type has to provide the two necessary pointers *&T::next_link* and *&T::prev_link*. One possibility to obtain these pointers is to inherit them from the base class *In_place_list_base*<*T*>.

The class *In_place_list*<*T*,*bool*> is a container quite similar to STL containers, with the advantage that it is able to handle the stored elements by reference instead of copying them. It is possible to delete an element only knowing its address and no iterator to it. This used to simplify mutually pointered data structures like a halfedge data structure for planar maps or polyhedral surfaces (the current design does not need this anymore). The usual iterators are also available.

45.2 Compact Container

The class *Compact_container*<*T*, *Allocator*> is an STL like container which provides a very compact storage for its elements. It achieves this goal by requiring *T* to provide access to a pointer in it, which is going to be used by *Compact_container*<*T*, *Allocator*> for its internal management. The traits class *Compact_container_traits*<*T*> specifies the way to access that pointer. The class *Compact_container_base* can be used as a base class to provide the pointer, although in this case you do not get the most compact representation. The values that this pointer can have during valid use of the object are valid pointer values to 4 bytes aligned objects (i.e., the two least significant bits of the pointer need to be zero when the object is constructed). Another interesting property of this container is that iterators are not invalidated during *insert* or *erase* operations.

The main deviation from the STL container concept is that the `++` and `--` operators of the iterator do not have a constant time complexity in all cases. The actual complexity is related to the maximum size that the container has had during its life time compared to its current size, because the iterator has to go over the "erased" elements as well, so the bad case is when the container used to contain lots of elements, but now has far less. In this case, we suggest to do a copy of the container in order to "defragment" the internal representation.

This container has been developed in order to efficiently handle large data structures like the triangulation and halfedge data structures. It can probably be useful for other kinds of graphs as well.

45.3 Multiset with Extended Functionality

The class `Multiset<Type, Compare, Allocator>` represents a multi-set of elements of type `Type`, represented as a red-black tree (see [CLRS01, Chapter 13] for an excellent introduction to red-black trees). It differs from the STL's `multiset` class-template mainly due to the fact that it is parameterized by a comparison functor `Compare` that returns the three-valued `Comparison_result` (namely it returns either `SMALLER`, `EQUAL`, or `LARGER`), rather than a `less` functor returning `bool`. Thus, it is possible to maintain the underlying red-black tree with less invocations of the comparison functor, which can considerably decrease running times, especially when comparing elements of type `Type` is an expensive operation.

`Multiset<Type, Compare, Allocator>` also guarantees that the order of elements sent to the comparison functor is fixed. For example, if we insert a new element `x` into the set (or erase an element from the set), then we always invoke `Compare()` (`x`, `y`) (and never `Compare()` (`y`, `x`)), where `y` is an element already stored in the set. This behavior, not supported by `std::multiset`, is sometimes crucial for designing more efficient comparison predicates.

The interface of `Multiset<Type, Compare, Allocator>` is in general derived from `std::multiset`. However, it extends the interface by offering some additional operations, such as: inserting of an element into the set given its *exact* position (and not just using an insertion hint); looking up keys whose type may differ from `Type`, as long as users supply a comparison functor `CompareKey`, between the keys and set elements; and catenating and splitting sets.

Functor Adaptors

The standard library contains some adaptors for binding functors, that is fixing one argument of a functor to a specific value thereby creating a new functor that takes one argument less than the original functor. Also, though non-standard, some STL implementations (such as SGI) provide adaptors to compose function objects. Unfortunately, these bind and compose adaptors are limited to unary and binary functors only, and these functors must not be overloaded.

Since there are a number of functors in CGAL that take more than two arguments, and since functors may also be overloaded, i.e., accept several different sets of arguments, we have to define our own adaptors to be used with CGAL functors.

STL Extensions for CGAL Reference Manual

Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Ron Wein

45.4 Classified Reference Pages

Doubly-Connected List Managing Items in Place

CGAL::In_place_list<T,bool> page [2603](#)
CGAL::In_place_list_base<T> page [2602](#)

Compact Container

CGAL::Compact_container<T, Allocator> page [2610](#)
CGAL::Compact_container_traits<T> page [2609](#)
CGAL::Compact_container_base page [2608](#)

Multiset with Extended Functionality

CGAL::Multiset<Type, Compare, Allocator> page [2614](#)

Generic Algorithms

CGAL::copy_n page [2623](#)
CGAL::min_max_element page [2624](#)
CGAL::predecessor page [2621](#)
CGAL::successor page [2622](#)

Iterators and Iterator/Circulator Adaptors.

CGAL::Emptyset_iterator page [2626](#)
CGAL::Oneset_iterator<T> page [2627](#)
CGAL::Insert_iterator<Container> page [2630](#)

<i>CGAL::Counting_iterator<Iterator, Value></i>	page 2629
<i>CGAL::N_step_adaptor<I, int N></i>	page 2631
<i>CGAL::Filter_iterator<Iterator, Predicate></i>	page 2632
<i>CGAL::Join_input_iterator_1<Iterator, Creator></i>	page 2633
<i>CGAL::Inverse_index<IC></i>	page 2634
<i>CGAL::Random_access_adaptor<IC></i>	page 2635
<i>CGAL::Random_access_value_adaptor<IC, T></i>	page 2636

Functor Adaptors

<i>CGAL::swap_1</i>	page 2637
<i>CGAL::swap_2</i>	page 2638
<i>CGAL::swap_3</i>	page 2639
<i>CGAL::swap_4</i>	page 2640
<i>CGAL::bind_1</i>	page 2641
<i>CGAL::bind_2</i>	page 2642
<i>CGAL::bind_3</i>	page 2643
<i>CGAL::bind_4</i>	page 2644
<i>CGAL::bind_5</i>	page 2645
<i>CGAL::compose</i>	page 2647
<i>CGAL::compose_shared</i>	page 2649
<i>CGAL::Swap<F, i></i>	page 2651
<i>CGAL::Bind<F, A, i></i>	page 2652
<i>CGAL::Compose<F0, F1, F2, F3></i>	page 2653
<i>CGAL::Compose_shared<F0, F1, F2, F3></i>	page 2655
<i>AdaptableFunctor</i>	page 2656
<i>CGAL::Arity_tag<int></i>	page 2658
<i>CGAL::Arity_traits<F></i>	page 2659
<i>CGAL::Set_arity<F, a></i>	page 2660
<i>CGAL::set_arity_0</i>	page 2661
<i>CGAL::set_arity_1</i>	page 2662
<i>CGAL::set_arity_2</i>	page 2663
<i>CGAL::set_arity_3</i>	page 2664
<i>CGAL::set_arity_4</i>	page 2665
<i>CGAL::set_arity_5</i>	page 2666

Projection Function Objects

<i>CGAL::Identity<Value></i>	page 2668
<i>CGAL::Dereference<Value></i>	page 2669
<i>CGAL::Get_address<Value></i>	page 2670
<i>CGAL::Cast_function_object<Arg, Result></i>	page 2671
<i>CGAL::Project_vertex<Node></i>	page 2672
<i>CGAL::Project_facet<Node></i>	page 2673
<i>CGAL::Project_point<Node></i>	page 2674
<i>CGAL::Project_normal<Node></i>	page 2675
<i>CGAL::Project_plane<Node></i>	page 2676
<i>CGAL::Project_next<Node></i>	page 2677
<i>CGAL::Project_prev<Node></i>	page 2678
<i>CGAL::Project_next_opposite<Node></i>	page 2679
<i>CGAL::Project_opposite_prev<Node></i>	page 2680

Creator Function Objects

<i>CGAL::Creator_1</i> <Arg, Result>	page 2681
<i>CGAL::Creator_2</i> <Arg1, Arg2, Result>	page 2682
<i>CGAL::Creator_3</i> <Arg1, Arg2, Arg3, Result>	page 2683
<i>CGAL::Creator_4</i> <Arg1, Arg2, Arg3, Arg4, Result>	page 2684
<i>CGAL::Creator_5</i> <Arg1, Arg2, Arg3, Arg4, Arg5, Result>	page 2685
<i>CGAL::Creator_uniform_2</i> <Arg, Result>	page 2686
<i>CGAL::Creator_uniform_3</i> <Arg, Result>	page 2687
<i>CGAL::Creator_uniform_4</i> <Arg, Result>	page 2688
<i>CGAL::Creator_uniform_5</i> <Arg, Result>	page 2689
<i>CGAL::Creator_uniform_6</i> <Arg, Result>	page 2690
<i>CGAL::Creator_uniform_7</i> <Arg, Result>	page 2691
<i>CGAL::Creator_uniform_8</i> <Arg, Result>	page 2692
<i>CGAL::Creator_uniform_9</i> <Arg, Result>	page 2693
<i>CGAL::Creator_uniform_d</i> <Arg, Result>	page 2694

Utilities

<i>CGAL::Twotuple</i> <T>	page 2695
<i>CGAL::Threetuple</i> <T>	page 2696
<i>CGAL::Fourtuple</i> <T>	page 2697
<i>CGAL::Sextuple</i> <T>	page 2698
<i>CGAL::Triple</i> <T1, T2, T3>	page 2699
<i>CGAL::Quadruple</i> <T1, T2, T3, T4>	page 2701

45.5 Alphabetical List of Reference Pages

<i>AdaptableFunctor</i>	page 2656
<i>Arity_tag</i> <int>	page 2658
<i>Arity_traits</i> <F>	page 2659
<i>Bind</i> <F,A,i>	page 2652
<i>bind_1</i>	page 2641
<i>bind_2</i>	page 2642
<i>bind_3</i>	page 2643
<i>bind_4</i>	page 2644
<i>bind_5</i>	page 2645
<i>Cast_function_object</i> <Arg, Result>	page 2671
<i>Compact_container</i> <T, Allocator>	page 2610
<i>Compact_container_base</i>	page 2608
<i>Compact_container_traits</i> <T>	page 2609
<i>Compare_to_less</i> <F>	page 2654
<i>compare_to_less</i>	page 2646
<i>Compose</i> <F0,F1,F2,F3>	page 2653
<i>Compose_shared</i> <F0,F1,F2,F3>	page 2655
<i>compose_shared</i>	page 2649
<i>compose</i>	page 2647
<i>Const_oneset_iterator</i> <T>	page 2628
<i>copy_n</i>	page 2623
<i>Counting_iterator</i> <Iterator, Value>	page 2629

<i>Creator_1</i> <Arg, Result>	page 2681
<i>Creator_2</i> <Arg1, Arg2, Result>	page 2682
<i>Creator_3</i> <Arg1, Arg2, Arg3, Result>	page 2683
<i>Creator_4</i> <Arg1, Arg2, Arg3, Arg4, Result>	page 2684
<i>Creator_5</i> <Arg1, Arg2, Arg3, Arg4, Arg5, Result>	page 2685
<i>Creator_uniform_2</i> <Arg, Result>	page 2686
<i>Creator_uniform_3</i> <Arg, Result>	page 2687
<i>Creator_uniform_4</i> <Arg, Result>	page 2688
<i>Creator_uniform_5</i> <Arg, Result>	page 2689
<i>Creator_uniform_6</i> <Arg, Result>	page 2690
<i>Creator_uniform_7</i> <Arg, Result>	page 2691
<i>Creator_uniform_8</i> <Arg, Result>	page 2692
<i>Creator_uniform_9</i> <Arg, Result>	page 2693
<i>Creator_uniform_d</i> <Arg, Result>	page 2694
<i>Dereference</i> <Value>	page 2669
<i>Emptyset_iterator</i>	page 2626
<i>Filter_iterator</i> <Iterator, Predicate>	page 2632
<i>Fourtuple</i> <T>	page 2697
<i>Get_address</i> <Value>	page 2670
<i>Identity</i> <Value>	page 2668
<i>Insert_iterator</i> <Container>	page 2630
<i>Inverse_index</i> <IC>	page 2634
<i>In_place_list</i> <T,bool>	page 2603
<i>In_place_list_base</i> <T>	page 2602
<i>Join_input_iterator_1</i> <Iterator, Creator>	page 2633
<i>min_max_element</i>	page 2624
<i>Multiset</i> <Type,Compare,Allocator>	page 2614
<i>negate</i>	page 2650
<i>N_step_adaptor</i> <I,int N>	page 2631
<i>Oneset_iterator</i> <T>	page 2627
<i>predecessor</i>	page 2621
<i>Projection_object</i>	page 2667
<i>Project_facet</i> <Node>	page 2673
<i>Project_next</i> <Node>	page 2677
<i>Project_next_opposite</i> <Node>	page 2679
<i>Project_normal</i> <Node>	page 2675
<i>Project_opposite_prev</i> <Node>	page 2680
<i>Project_plane</i> <Node>	page 2676
<i>Project_point</i> <Node>	page 2674
<i>Project_prev</i> <Node>	page 2678
<i>Project_vertex</i> <Node>	page 2672
<i>Quadruple</i> <T1, T2, T3, T4>	page 2701
<i>Random_access_adaptor</i> <IC>	page 2635
<i>Random_access_value_adaptor</i> <IC,T>	page 2636
<i>Set_arity</i> <F,a>	page 2660
<i>set_arity_0</i>	page 2661
<i>set_arity_1</i>	page 2662
<i>set_arity_2</i>	page 2663
<i>set_arity_3</i>	page 2664
<i>set_arity_4</i>	page 2665
<i>set_arity_5</i>	page 2666
<i>Sixtuple</i> <T>	page 2698
<i>successor</i>	page 2622
<i>Swap</i> <F,i>	page 2651

<i>swap_1</i>	page 2637
<i>swap_2</i>	page 2638
<i>swap_3</i>	page 2639
<i>swap_4</i>	page 2640
<i>Threetuple</i> < <i>T</i> >	page 2696
<i>Triple</i> < <i>T1</i> , <i>T2</i> , <i>T3</i> >	page 2699
<i>Twotuple</i> < <i>T</i> >	page 2695

CGAL::In_place_list_base<T>

Definition

The node base classes provides pointers to build linked lists. The class *In_place_sl_list_base*<*T*> provides a pointer *next_link* for a single linked list. The class *In_place_list_base*<*T*> provides an additional pointer *prev_link* for doubly linked lists. These names conform to the default parameters used in the template argument lists of the container classes. The pointers are public members.

```
#include <CGAL/In_place_list.h>
```

Variables

<i>T</i> *	<i>next_link</i> ;	forward pointer
<i>T</i> *	<i>prev_link</i> ;	backward pointer

CGAL::In_place_list<T,bool>

Definition

An object of the class *In_place_list*<*T*,*bool*> represents a sequence of items of type *T* that supports bidirectional iterators and allows constant time insert and erase operations anywhere within the sequence. The functionality is similar to the *list*<*T*> in the STL.

The *In_place_list*<*T*,*bool*> manages the items in place, i.e., inserted items are not copied. Two pointers of type *T** are expected to be reserved in *T* for the list management. The base class *In_place_list_base*<*T*> can be used to obtain such pointers.

The *In_place_list*<*T*,*bool*> does not copy element items during insertion (unless otherwise stated for a function). On removal of an item or destruction of the list the items are not deleted by default. The second template parameter *bool* is set to *false* in this case. If the *In_place_list*<*T*,*bool*> should take the responsibility for the stored objects the *bool* parameter could be set to *true*, in which case the list will delete removed items and will delete all remaining items on destruction. In any case, the *destroy()* member function deletes all items. Note that these two possible versions of *In_place_list*<*T*,*bool*> are not assignable to each other to avoid confusions between the different storage responsibilities.

```
#include <CGAL/In_place_list.h>
```

Parameters

The full class name is *In_place_list*<*T*, *bool managed* = *false*, *class Alloc* = *CGAL_ALLOCATOR*(*T*)>.

The parameter *T* is supposed to have a default constructor, a copy constructor and an assignment operator. The copy constructor and the assignment may copy the pointers in *T* for the list management, but they do not have to. The equality test and the relational order require the operators == and < for *T* respectively. These operators must not compare the pointers in *T*.

Types

```
In_place_list<T,bool>:: iterator  
In_place_list<T,bool>:: const_iterator
```

```
In_place_list<T,bool>:: value_type  
In_place_list<T,bool>:: reference  
In_place_list<T,bool>:: const_reference  
In_place_list<T,bool>:: size_type  
In_place_list<T,bool>:: difference_type
```

```
In_place_list<T,bool>:: reverse_iterator  
In_place_list<T,bool>:: const_reverse_iterator
```

```
In_place_list<T,bool>:: allocator_type
```

Creation

In_place_list<*T*,*bool*> *l*; introduces an empty list.

In_place_list<*T*,*bool*> *l*(*list*<*T*> *ll*); copy constructor. Each item in *ll* is copied.

In_place_list<*T*,*bool*> *l*(*size_type* *n*, *T* *t* = *T*());

introduces a list with *n* items, all initialized with copies of *t*.

template <*class InputIterator*>

In_place_list<*T*,*bool*> *l*(*InputIterator* *first*, *InputIterator* *last*);

a list with copies from the range [*first*,*last*).

In_place_list<*T*,*bool*> *l*(*const T** *first*, *const T** *last*);

non-member-template version.

In_place_list<*T*,*bool*> & *l* = *ll* assignment. Each item in *ll* is copied. Each item in *l* is deleted if the *bool* parameter is *true*.

void *l.swap*(*ll*) swaps the contents of *l* with *ll*.

void *l.destroy*() all items in *l* are deleted regardless of the *bool* parameter.

Comparison Operations

bool *l* == *ll* test for equality: Two lists are equal, iff they have the same size and if their corresponding elements are equal.

bool *l* < *ll* compares in lexicographical order.

Access Member Functions

iterator *l.begin*() returns a mutable iterator referring to the first element in *l*.

const_iterator *l.begin*() *const* returns a constant iterator referring to the first element in *l*.

iterator *l.end*() returns a mutable iterator which is the past-end-value of *l*.

const_iterator *l.end*() *const* returns a constant iterator which is the past-end-value of *l*.

bool *l.empty*() returns *true* if *l* is empty.

size_type *l.size*() returns the number of items in list *l*.

size_type *l.max_size*() returns the maximum possible size of the list *l*.

T& *l.front*() returns the first item in list *l*.

T& *l.back*() returns the last item in list *l*.

allocator_type

l.get_allocator() returns the allocator.

Insertion

void l.push_front(T&) inserts an item in front of list *l*.
void l.push_back(T&) inserts an item at the back of list *l*.

iterator l.insert(iterator pos, T& t)
iterator l.insert(T pos, T& t)* inserts *t* in front of *pos*. The return value points to the inserted item.

void l.insert(iterator pos, size_type n, T t = T())
void l.insert(T pos, size_type n, T t = T())*
inserts *n* copies of *t* in front of *pos*.

template <class InputIterator>
void l.insert(iterator pos, InputIterator first, InputIterator last)

template <class InputIterator>
void l.insert(T pos, InputIterator first, InputIterator last)*
inserts the range [*first*, *last*) in front of iterator *pos*.

As long as member templates are not supported, member functions using *T** instead of the general *InputIterator* are provided.

Removal

void l.pop_front() removes the first item from list *l*.
void l.pop_back() removes the last item from list *l*.
void l.erase(iterator pos) removes the item from list *l*, where *pos* refers to.
void l.erase(T pos)* removes the item from list *l*, where *pos* refers to.

void l.erase(iterator first, iterator last)
void l.erase(T first, T* last)*
removes the items in the range [*first*, *last*) from *l*.

Special List Operations

void l.splice(iterator pos, & x)
void l.splice(T pos, & x)* inserts the list *x* before position *pos* and *x* becomes empty. It takes constant time.
Precondition: & l != & x.

void l.splice(iterator pos, & x, iterator i)

<i>void</i>	<i>l.splice(T* pos, & x, T* i)</i>	inserts an element pointed to by <i>i</i> from list <i>x</i> before position <i>pos</i> and removes the element from <i>x</i> . It takes constant time. <i>i</i> is a valid dereferenceable iterator of <i>x</i> . The result is unchanged if <i>pos</i> == <i>i</i> or <i>pos</i> == ++ <i>i</i> .
<i>void</i> <i>void</i>	<i>l.splice(iterator pos, & x, iterator first, iterator last)</i> <i>l.splice(T* pos, & x, T* first, T* last)</i>	inserts elements in the range [<i>first</i> , <i>last</i>) before position <i>pos</i> and removes the elements from <i>x</i> . It takes constant time if & <i>x</i> == & <i>l</i> ; otherwise, it takes linear time. [<i>first</i> , <i>last</i>) is a valid range in <i>x</i> . <i>Precondition</i> : <i>pos</i> is not in the range [<i>first</i> , <i>last</i>).
<i>void</i>	<i>l.remove(T value)</i>	erases all elements <i>e</i> in the list <i>l</i> for which <i>e</i> == <i>value</i> . It is stable. <i>Precondition</i> : a suitable <i>operator</i> == for the type <i>T</i> .
<i>void</i>	<i>l.unique()</i>	erases all but the first element from every consecutive group of equal elements in the list <i>l</i> . <i>Precondition</i> : a suitable <i>operator</i> == for the type <i>T</i> .
<i>void</i>	<i>l.merge(& x)</i>	merges the list <i>x</i> into the list <i>l</i> and <i>x</i> becomes empty. It is stable. <i>Precondition</i> : Both lists are increasingly sorted. A suitable <i>operator</i> < for the type <i>T</i> .
<i>void</i>	<i>l.reverse()</i>	reverses the order of the elements in <i>l</i> in linear time.
<i>void</i>	<i>l.sort()</i>	sorts the list <i>l</i> according to the <i>operator</i> < in time $O(n \log n)$ where $n = \text{size}()$. It is stable. <i>Precondition</i> : a suitable <i>operator</i> < for the type <i>T</i> .

Example

```
// in_place_list_prog.C
// -----
#include <CGAL/basic.h>
#include <cassert>
#include <algorithm>
#include <CGAL/In_place_list.h>

using CGAL::In_place_list_base;

struct item : public In_place_list_base<item> {
    int key;
    item() {}
    item( const item& i) : In_place_list_base<item>(i), key(i.key) {}
    item( int i) : key(i) {}
    bool operator== (const item& i) const { return key == i.key;}
    bool operator!= (const item& i) const { return ! (*this == i);}
    bool operator== (int i) const          { return key == i;}
    bool operator!= (int i) const          { return ! (*this == i);}
    bool operator<  (const item& i) const { return key < i.key;}
};

int main() {
    typedef CGAL::In_place_list<item,true> List;
    List l;
    item* p = new item(1);
    l.push_back(*p);
    l.push_back(*new item(2));
    l.push_front(*new item(3));
    l.push_front(*new item(4));
    l.push_front(*new item(2));
    List::iterator i = l.begin();
    ++i;
    l.insert(i, *new item(5));
    l.insert(p, *new item(5));
    int a[7] = {2,5,4,3,5,1,2};
    bool ok = std::equal(l.begin(), l.end(), a);
    assert(ok);
    l.sort();
    l.unique();
    assert(l.size() == 5);
    int b[5] = {1,2,3,4,5};
    ok = std::equal(l.begin(), l.end(), b);
    assert(ok);
    return 0;
}
```

CGAL::Compact_container_base

Definition

The class *Compact_container_base* can be used as a base class for your own type *T*, so that *T* can be used directly within *Compact_container*<*T*, *Allocator*>. This class stores a *void ** pointer only for this purpose, so it may not be the most memory efficient way to achieve this goal. The other ways are to provide in *T* the necessary member functions so that the template *Compact_container_traits*<*T*> works, or to specialize it for the particular type *T* that you want to use.

```
#include <CGAL/Compact_container.h>
```

Operations

<code>void*</code>	<code>ccb.for_compact_container() const</code>	Returns the pointer necessary for <i>Compact_container_traits</i> < <i>T</i> >.
<code>void*&</code>	<code>ccb.for_compact_container()</code>	Returns a reference to the pointer necessary for <i>Compact_container_traits</i> < <i>T</i> >.

CGAL::Compact_container_traits<T>

Definition

The traits class *Compact_container_traits*<*T*> provides the way to access the internal pointer required for *T* to be used in a *Compact_container*<*T*, *Allocator*>. Note that this pointer needs to be accessible even when the object is not constructed, which means it has to reside in the same memory place as *T*.

You can specialize this class for your own type *T* if the default template is not suitable.

You can also use *Compact_container_base* as base class for your own types *T* to make them usable with the default *Compact_container_traits*<*T*>.

```
#include <CGAL/Compact_container.h>
```

Parameters

T is any type providing the following member functions:

```
void *t.for_compact_container() const;
void *&t.for_compact_container();
```

Operations

<i>static void*</i> <i>cct.pointer(const T &t)</i>	Returns the pointer hold by <i>t</i> . The template version defines this function as: <i>return t.for_compact_container();</i>
<i>static void*&</i> <i>cct.pointer(T &t)</i>	Returns a reference to the pointer hold by <i>t</i> . The template version defines this function as: <i>return t.for_compact_container();</i>

CGAL::Compact_container<T, Allocator>

Definition

An object of the class *Compact_container*<*T*, *Allocator*> is a container of objects of type *T*. It matches all the standard requirements for reversible containers, except that the complexity of its iterator increment and decrement operations is not always guaranteed to be amortized constant time.

This container is not a standard *sequence* nor *associative* container, which means the elements are stored in no particular order, and it is not possible to specify a particular place in the iterator sequence where to insert new objects. However, all dereferenceable iterators are still valid after calls to *insert()* and *erase()*, except those that have been erased (it behaves similarly to *std::list*).

The main feature of this container is that it is very memory efficient : its memory size is $N * \text{sizeof}(T) + o(N)$, where *N* is the maximum size that the container has had in its past history, its *capacity()* (the memory of erased elements is not deallocated until destruction of the container or a call to *clear()*). This container has been developed in order to store large graph-like data structures like the triangulation and the halfedge data structures.

It supports bidirectional iterators and allows a constant time amortized *insert()* operation. You cannot specify where to insert new objects (i.e. you don't know where they will end up in the iterator sequence, although *insert()* returns an iterator pointing to the newly inserted object). You can erase any element with a constant time complexity.

Summary of the differences with *std::list* : it is more compact in memory since it doesn't store two additional pointers for the iterator needs. It doesn't deallocate elements until the destruction or *clear()* of the container. The iterator does not have constant amortized time complexity for the increment and decrement operations in all cases, only when not too many elements have not been freed (i.e. when the *size()* is close to the *capacity()*). Iterating from *begin()* to *end()* takes $O(\text{capacity}())$ time, not *size()*. In the case where the container has a small *size()* compared to its *capacity()*, we advise to "defragment the memory" by copying the container if the iterator performance is needed.

The iterators themselves can be used as *T*, they provide the necessary functions to be used by *Compact_container_traits*<*T*>.

```
#include <CGAL/Compact_container.h>
```

Parameters

The parameter *T* is required to have a copy constructor and an assignment operator. It also needs to provide access to an internal pointer via *Compact_container_traits*<*T*>.

The equality test and the relational order require the operators == and < for *T* respectively.

The parameter *Allocator* has to match the standard allocator requirements, with value type *T*. This parameter has the default value *CGAL_ALLOCATOR*(*T*).

Types

Compact_container<*T*, *Allocator*>:: *value_type*

Compact_container<T, Allocator>:: reference
Compact_container<T, Allocator>:: const_reference
Compact_container<T, Allocator>:: pointer
Compact_container<T, Allocator>:: const_pointer
Compact_container<T, Allocator>:: size_type
Compact_container<T, Allocator>:: difference_type

Compact_container<T, Allocator>:: iterator
Compact_container<T, Allocator>:: const_iterator
Compact_container<T, Allocator>:: reverse_iterator
Compact_container<T, Allocator>:: const_reverse_iterator

Compact_container<T, Allocator>:: allocator_type

Creation

Compact_container<T, Allocator> c(Allocator a = Allocator());

introduces an empty container, eventually specifying a particular allocator *a* as well.

template <class InputIterator>

Compact_container<T, Allocator> c(InputIterator first, InputIterator last, Allocator a = Allocator());

a container with copies from the range [*first,last*), eventually specifying a particular allocator.

Compact_container<T, Allocator> c(cc);

copy constructor. Each item in *cc* is copied. The allocator is copied. The iterator order is preserved.

Compact_container<T, Allocator> & c = cc

assignment. Each item in *cc* is copied. The allocator is copied. Each item in *c* is deleted. The iterator order is preserved.

void c.swap(&cc)

swaps the contents of *c* and *cc* in constant time complexity. No exception is thrown.

Access Member Functions

iterator c.begin() returns a mutable iterator referring to the first element in *c*.

const_iterator c.begin() const returns a constant iterator referring to the first element in *c*.

iterator c.end() returns a mutable iterator which is the past-end-value of *c*.

<i>const_iterator</i>	<i>c.end()</i> <i>const</i>	returns a constant iterator which is the past-end-value of <i>c</i> .
<i>reverse_iterator</i>	<i>c.rbegin()</i>	
<i>const_reverse_iterator</i>	<i>c.rbegin()</i> <i>const</i>	
<i>reverse_iterator</i>	<i>c.rend()</i>	
<i>const_reverse_iterator</i>	<i>c.rend()</i> <i>const</i>	
<i>bool</i>	<i>c.empty()</i>	returns <i>true</i> iff <i>c</i> is empty.
<i>size_type</i>	<i>c.size()</i>	returns the number of items in <i>c</i> .
<i>size_type</i>	<i>c.max_size()</i>	returns the maximum possible size of the container <i>c</i> .
<i>size_type</i>	<i>c.capacity()</i>	returns the total number of elements that <i>c</i> can hold without requiring reallocation.
<i>Allocator</i>	<i>c.get_allocator()</i>	returns the allocator.
Insertion		
<i>iterator</i>	<i>c.insert(T t)</i>	inserts a copy of <i>t</i> in <i>c</i> and returns the iterator pointing to it.
<i>template <class InputIterator></i> <i>void</i>	<i>c.insert(InputIterator first, InputIterator last)</i>	inserts the range [<i>first</i> , <i>last</i>) in <i>c</i> .
<i>template <class InputIterator></i> <i>void</i>	<i>c.assign(InputIterator first, InputIterator last)</i>	erases all the elements of <i>c</i> , then inserts the range [<i>first</i> , <i>last</i>) in <i>c</i> .
<i>template < typename T1 ></i> <i>iterator</i>	<i>c.construct_insert(T1 t1)</i>	constructs an object of type <i>T</i> with the constructor that takes <i>t1</i> as argument, inserts it in <i>c</i> , and returns the iterator pointing to it. The same constructor exists for up to nine arguments.

Removal

<i>void</i>	<i>c.erase(iterator pos)</i>	removes the item pointed by <i>pos</i> from <i>c</i> .
<i>void</i>	<i>c.erase(iterator first, iterator last)</i>	removes the items from the range [<i>first</i> , <i>last</i>) from <i>c</i> .
<i>void</i>	<i>c.clear()</i>	all items in <i>c</i> are deleted, and the memory is deallocated. After this call, <i>c</i> is in the same state as if just default constructed.

Special Operation

<i>void</i>	<i>c.merge(&cc)</i>	adds the items of <i>cc</i> to the end of <i>c</i> and <i>cc</i> becomes empty. The time complexity is $O(c.capacity() - c.size())$. <i>Precondition:</i> <i>cc</i> must not be the same as <i>c</i> , and the allocators of <i>c</i> and <i>cc</i> need to be compatible : <i>c.get_allocator() == cc.get_allocator()</i> .
-------------	--------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Comparison Operations

<i>bool</i>	<i>c == cc</i>	test for equality: Two containers are equal, iff they have the same size and if their corresponding elements are equal.
<i>bool</i>	<i>c != cc</i>	test for inequality: returns $!(c == cc)$.
<i>bool</i>	<i>c < cc</i>	compares in lexicographical order.
<i>bool</i>	<i>c > cc</i>	returns $cc < c$.
<i>bool</i>	<i>c <= cc</i>	returns $!(c > cc)$.
<i>bool</i>	<i>c >= cc</i>	returns $!(c < cc)$.

CGAL::Multiset<Type,Compare,Allocator>

Definition

An instance s of the parametrized data type *Multiset*<*Type*,*Compare*,*Allocator*> is a multi-set of elements of type *Type*, represented as a red-black tree (see [CLRS01, Chapter 13] for an excellent introduction to red-black trees). The main difference between *Multiset*<*Type*,*Compare*,*Allocator*> and STL's *multiset* is that the latter uses a less-than functor with a Boolean return type, while our *Multiset*<*Type*,*Compare*,*Allocator*> class is parameterized by a comparison functor *Compare* that returns the three-valued *Comparison_result* (namely it returns either *SMALLER*, *EQUAL*, or *LARGER*). It is thus possible to maintain the underlying red-black tree with less invocations of the comparison functor. This leads to a speedup of about 5% even if we maintain a set of integers. When each comparison of two elements of type *Type* is an expensive operation (for example, when they are geometric entities represented using exact arithmetic), the usage of a three-valued comparison functor can lead to considerable decrease in the running times.

Moreover, *Multiset*<*Type*,*Compare*,*Allocator*> allows the insertion of an element into the set given its *exact* position, and not just using an insertion hint, as done by *std::multiset*. This can further reduce the running times, as additional comparison operations can be avoided.

In addition, the *Multiset*<*Type*,*Compare*,*Allocator*> guarantees that the order of elements sent to the comparison functor is fixed. For example, if we insert a new element x into the set (or erase an element from the set), then we always invoke *Compare*() (x , y) (and never *Compare*() (y , x)), where y is an element already stored in the set. This behavior, not supported by *std::multiset*, is sometimes crucial for designing more efficient comparison predicates.

Multiset<*Type*,*Compare*,*Allocator*> also allows for look-up of keys whose type may differ from *Type*, as long as users supply a comparison functor *CompareKey*, where *CompareKey*() (key , y) returns the three-valued *Comparison_result* (key is the look-up key and y is an element of type *Type*). Indeed, it is very convenient to look-up equivalent objects in the set given just by their key. We note however that it is also possible to use a key of type *Type* and to employ the default *Compare* functor for the look-up, as done when using the *std::multiset* class.

— advanced —

Finally, *Multiset*<*Type*,*Compare*,*Allocator*> introduces the *catenate*() and *split*() functions. The first function operates on s and accepts a second set s' , such that the maximum element in s is not greater than the minimal element in s' , and concatenates s' to s . The second function splits s into two sets, one containing all the elements that are less than a given key, and the other contains all elements greater than (or equal to) this key.

— advanced —

Parameters

The *Multiset* class-template has three parameters:

- *Type* — the type of the stored elements.
- *Compare* — the comparison-functor type. This type should provide the following operator for comparing two *Type* elements, namely:
`Comparison_result operator() (const Type& t1, const Type& t2) const;`
 The *CGAL::Compare*<*Type*> functor is used by default. In this case, *Type* must support an equality operator (*operator*==) and a less-than operator (*operator*<).

- *Allocator* — the allocator type.
CGAL_ALLOCATOR is used by default.

```
#include <CGAL/Multiset.h>
```

Assertions

The assertion and precondition flags for the *Multiset* class use *MULTISET* in their names (i.e., *CGAL_MULTISET_NO_ASSERTIONS* and *CGAL_MULTISET_NO_PRECONDITIONS*).

Types

In compliance with STL, the types *value_type* and *key_type* (both equivalent to *Type*), *reference* and *const_reference* (reference to a value-type), *key_compare* and *value_compare* (both equivalent to *Compare*), *size_type* and *difference_type* are defined as well.

```
Multiset<Type,Compare,Allocator>:: iterator  
Multiset<Type,Compare,Allocator>:: const_iterator
```

bi-directional iterators for the elements stored in the set.

```
Multiset<Type,Compare,Allocator>:: reverse_iterator  
Multiset<Type,Compare,Allocator>:: const_reverse_iterator
```

reverse bi-directional iterators for the elements stored in the set.

Creation

```
Multiset<Type,Compare,Allocator> s;
```

creates an an empty set *s* that uses a default comparison functor.

```
Multiset<Type,Compare,Allocator> s( Compare comp);
```

creates an an empty set *s* that uses the given comparison functor *comp*.

```
template <class InputIterator>  
Multiset<Type,Compare,Allocator> s( InputIterator first, InputIterator last, Compare comp = Compare());
```

creates a set *s* containing all elements in the range [*first*, *last*), that uses the comparison functor *comp*.

```
Multiset<Type,Compare,Allocator> s( other);
```

copy constructor.

```
Multiset<Type,Compare,Allocator>  
s = other
```

assignment operator.

<i>void</i>	<i>s.swap(& other)</i>	swaps the contents of <i>s</i> with those of the other set.
-------------	-----------------------------	-------------------------------------------------------------

Access Member Functions

<i>Compare</i>	<i>s.key_comp()</i>	the comparison functor used.
<i>Compare</i>	<i>s.value_comp()</i>	the comparison functor used (same as above). Both functions have a non-const version that return a reference to the comparison functor.

<i>bool</i>	<i>s.empty()</i>	returns <i>true</i> if the set is empty, <i>false</i> otherwise.
-------------	------------------	------------------------------------------------------------------

<i>size_t</i>	<i>s.size()</i>	returns the number of elements stored in the set.
---------------	-----------------	---------------------------------------------------

<i>size_t</i>	<i>s.max_size()</i>	returns the maximal number of elements the set can store (same as <i>size()</i>).
---------------	---------------------	------------------------------------------------------------------------------------

<i>iterator</i>	<i>s.begin()</i>	returns an iterator pointing to the first element stored in the set (a <i>const</i> version is also available).
-----------------	------------------	-----------------------------------------------------------------------------------------------------------------

<i>iterator</i>	<i>s.end()</i>	returns an iterator pointing beyond the last element stored in the set (a <i>const</i> version is also available).
-----------------	----------------	--------------------------------------------------------------------------------------------------------------------

<i>reverse_iterator</i>	<i>s.rbegin()</i>	returns a reverse iterator pointing beyond the last element stored in the set (a <i>const</i> version is also available).
-------------------------	-------------------	---------------------------------------------------------------------------------------------------------------------------

<i>reverse_iterator</i>	<i>s.rend()</i>	returns a reverse iterator pointing to the first element stored in the set (a <i>const</i> version is also available).
-------------------------	-----------------	------------------------------------------------------------------------------------------------------------------------

Comparison Operations

<i>bool</i>	<i>s == other</i>	returns <i>true</i> if the sequences of elements in the two sets are pairwise equal (using the comparison functor).
-------------	-------------------	---------------------------------------------------------------------------------------------------------------------

<i>bool</i>	<i>s < other</i>	returns <i>true</i> if the element sequence in <i>s</i> is lexicographically smaller than the element sequence of <i>other</i> .
-------------	---------------------	----------------------------------------------------------------------------------------------------------------------------------

Insertion Methods

<i>iterator</i>	<i>s.insert(Type x)</i>	inserts the element <i>x</i> into the set and returns an iterator pointing to the newly inserted element.
-----------------	--------------------------	-----------------------------------------------------------------------------------------------------------

<i>template <class InputIterator></i>	<i>s.insert(InputIterator first, InputIterator last)</i>	
<i>void</i>		inserts all elements in the range <i>[first, last)</i> into the set.

<i>iterator</i>	<i>s.insert(iterator position, Type x)</i>	inserts the element <i>x</i> with a given iterator used as a hint for the position of the new element. It Returns an iterator pointing to the newly inserted element.
-----------------	---------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------

<i>iterator</i>	<i>s.insert_before(iterator position, Type x)</i>	inserts the element <i>x</i> as the predecessor of the element at the given position. <i>Precondition:</i> The operation does not violate the set order — that is, <i>x</i> is not greater than the element pointed by <i>position</i> and not less than its current predecessor.
-----------------	----------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<i>iterator</i>	<i>s.insert_after(iterator position, Type x)</i>	inserts the element <i>x</i> as the successor of the element at the given position. <i>Precondition:</i> The operation does not violate the set order — that is, <i>x</i> is not less than the element pointed by <i>position</i> and not greater than its current successor.
-----------------	---------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Removal Methods

<i>size_t</i>	<i>s.erase(Type x)</i>	erases all elements equivalent to <i>x</i> from the set and returns the number of erased elements.
---------------	-------------------------	----------------------------------------------------------------------------------------------------

<i>void</i>	<i>s.erase(iterator position)</i>	erases the element pointed by <i>position</i> .
-------------	------------------------------------	-------------------------------------------------

<i>void</i>	<i>s.clear()</i>	clears the set (erases all stored elements).
-------------	------------------	----------------------------------------------

Look-up Methods

All methods listed in this section can also accept a *Type* element as a look-up key. In this case, it is not necessary to supply a *CompareKey* functor, as the *Compare* functor will be used by default.

<i>template <class Key, class CompareKey></i> <i>iterator</i>	<i>s.find(Key key, CompareKey comp_key)</i>	searches for the an element equivalent to <i>key</i> in the set. If the set contains objects equivalent to <i>key</i> , it returns an iterator pointing to the first one. Otherwise, <i>end()</i> is returned (a <i>const</i> version is also available).
------------------------------------------------------------------------	----------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

template <class Key, class CompareKey>

size_t *s.count(Key key, CompareKey comp_key)*

returns the number of elements equivalent to *key* in the set.

template <class Key, class CompareKey>
iterator *s.lower_bound(Key key, CompareKey comp_key)*

returns an iterator pointing to the first element in the set that is not less than *key*. If all set elements are less than *key*, *end()* is returned (a *const* version is also available).

template <class Key, class CompareKey>
iterator *s.upper_bound(Key key, CompareKey comp_key)*

returns an iterator pointing to the first element in the set that is greater than *key*. If no set element is greater than *key*, *end()* is returned (a *const* version is also available).

template <class Key, class CompareKey>
std::pair<iterator,iterator>

s.equal_range(Key key, CompareKey comp_key)

returns the range of set elements equivalent to the given key, namely (*lower_bound(key)*, *upper_bound(key)*) (a *const* version is also available).

template <class Key, class CompareKey>
std::pair<iterator,bool>

s.find_lower(Key key, CompareKey comp_key)

returns a pair comprised of *lower_bound(key)* and a Boolean flag indicating whether this iterator points to an element equivalent to the given key (a *const* version is also available).

— advanced —

Special Operations

void *s.replace(iterator position, Type x)*

replaces the element stored at the given position with *x*.
Precondition: The operation does not violate the set order — that is, *x* is not less than *position*'s predecessor and not greater than its successor.

void *s.swap(iterator pos1, iterator pos2)*

swaps places between the two elements given by *pos1* and *pos2*.

Precondition: The operation does not violate the set order — that is, *pos1* and *pos2* store equivalent elements.

void *s.catenate(Self& s_prime)*

concatenates all elements in *s_prime* into *s* and clears *s_prime*. All iterators to *s* and to *s_prime* remain valid.

Precondition: The maximal element in *s* is not greater than the minimal element in *s_prime*.

template <class Key, class CompareKey>
void *s.split(Key key, CompareKey comp_key, Self& s_prime)*

splits *s* such that it contains all elements that are less than the given *key* and such that *s_prime* contains all other elements.

Precondition: *s_prime* is initially empty.

void *s.split(iterator position, Self& s_prime)*

splits *s* such that it contains all set elements in the range [*begin*, *position*) and such that *s_prime* contains all elements in the range [*position*, *end*()).

Precondition: *s_prime* is initially empty.

└────────── *advanced* ─────────┘

Implementation

Multiset uses a proprietary implementation of a red-black tree data-structure. The red-black tree invariants guarantee that the height of a tree containing n elements is $O(\log n)$ (more precisely, it is bounded by $2 \log_2 n$). As a consequence, all methods that accept an element and need to locate it in the tree (namely *insert*(*x*), *erase*(*x*), *find*(*x*), *count*(*x*), *lower_bound*(*x*), *upper_bound*(*x*), *find_lower*(*x*) and *equal_range*(*x*)) take $O(\log n)$ time and perform $O(\log n)$ comparison operations.

On the other hand, the set operations that accept a position iterator (namely *insert_before*(*pos*, *x*), *insert_after*(*pos*, *x*) and *erase*(*pos*)) are much more efficient as they can be performed at a *constant* amortized cost (see [GS78] and [Tar83] for more details). More important, these set operations require *no* comparison operations. Therefore, it is highly recommended to maintain the set via iterators to the stored elements, whenever possible. The function *insert*(*pos*, *x*) is safer to use, but it takes amortized $O(\min\{d, \log n\})$ time, where d is the distance between the given position and the true position of *x*. In addition, it always performs at least two comparison operations.

└────────── *advanced* ─────────┘

The *catenate*() and *split*() functions are also very efficient, and can be performed in $O(\log n)$ time, where n is the total number of elements in the sets, and without performing any comparison operations (see [Tar83] for the details). Note however that the size of two sets resulting from a split operation is initially unknown, as it is

impossible to compute it in less than linear time. Thus, the first invocation of *size()* on such a set takes linear time, and *not* constant time.

————— *advanced* —————

The design is derived from the STL *multiset* class-template (see, e.g. [\[MS96\]](#)), where the main differences between the two classes are highlighted in the class definition above.

CGAL::predecessor

Definition

The function *predecessor* returns the previous iterator, i.e. the result of *operator--* on a bidirectional iterator.

```
#include <CGAL/algorithm.h>
```

```
template <class BidirectionalIterator>
```

```
BidirectionalIterator predecessor( BidirectionalIterator it)
```

returns *--it*.

See Also

CGAL::successor [page 2622](#)

CGAL::successor

Definition

The function *successor* returns the next iterator, i.e. the result of *operator++* on a forward iterator.

```
#include <CGAL/algorithm.h>
```

```
template <class ForwardIterator>
```

```
ForwardIterator    successor( ForwardIterator it)
```

returns $++it$.

See Also

CGAL::predecessor page [2621](#)

CGAL::copy_n

Definition

The function *copy_n* copies *n* items from an input iterator to an output iterator which is useful for possibly infinite sequences of random geometric objects.¹

```
#include <CGAL/algorithm.h>
```

```
template <class InputIterator, class Size, class OutputIterator>
OutputIterator copy_n( InputIterator first, Size n, OutputIterator result)
```

copies the first *n* items from *first* to *result*. Returns the value of *result* after inserting the *n* items.

See Also

CGAL::Counting_iterator<*Iterator*, *Value*> page [2629](#)

¹The STL release June 13, 1997, from SGI contains an equivalent function, but it is not part of the ISO standard.

CGAL::min_max_element

Definition

The function *min_max_element* computes the minimal and the maximal element of a range. It is modeled after the STL functions *min_element* and *max_element*. The advantage of *min_max_element* compared to calling both STL functions is that one only iterates once over the sequence. This is more efficient especially for large and/or complex sequences.

```
#include <CGAL/algorithm.h>
```

```
template < class ForwardIterator >
std::pair< ForwardIterator, ForwardIterator >
```

```
min_max_element( ForwardIterator first, ForwardIterator last)
```

returns a pair of iterators where the first component refers to the minimal and the second component refers to the maximal element in the range $[first, last)$. The ordering is defined by *operator<* on the value type of *ForwardIterator*.

```
template < class ForwardIterator, class CompareMin, class CompareMax >
std::pair< ForwardIterator, ForwardIterator >
```

```
min_max_element( ForwardIterator first,
                  ForwardIterator last,
                  CompareMin comp_min,
                  CompareMax comp_max)
```

returns a pair of iterators where the first component refers to the minimal and the second component refers to the maximal element in the range $[first, last)$.

Requirement: *CompareMin* and *CompareMax* are adaptable binary function objects: $VT \times VT \rightarrow bool$ where *VT* is the value type of *ForwardIterator*.

Example

The following example program computes the minimal and maximal element of the sequence (3, 6, 5). Hence the output is *min* = 3, *max* = 6.

```
#include <CGAL/algorithm.h>
#include <vector>
#include <iostream>

using std::vector;
using std::pair;
using std::cout;
```



```

using std::endl;
using CGAL::min_max_element;

int main()
{
    vector< int > v;
    v.push_back(3);
    v.push_back(6);
    v.push_back(5);
    typedef std::vector< int >::iterator iterator;
    pair< iterator, iterator > p = min_max_element(v.begin(), v.end());
    cout << "min = " << *p.first << ", max = " << *p.second << endl;
    return 0;
}

```

CGAL::Emptyset_iterator

Definition

The class defines an *OutputIterator* that ignores everything written to it. One can think of it as being connected to `/dev/null`.

```
#include <CGAL/iterator.h>
```

Is Model for the Concepts

OutputIterator

Creation

```
Emptyset_iterator i;
```

default constructor.

See Also

CGAL::Oneset_iteratorpage ??
CGAL::Const_oneset_iteratorpage ??

CGAL::Oneset_iterator<T>

Definition

The class *Oneset_iterator*<*T*> defines an *BidirectionalIterator* that always refers to one specific object of type *T*. Internally, *Oneset_iterator*<*T*> stores a pointer to the referred object.

```
#include <CGAL/iterator.h>
```

Is Model for the Concepts

BidirectionalIterator

Creation

```
Oneset_iterator<T> i( T& t);
```

creates an iterator referring to *t*.

See Also

CGAL::Emptyset_iterator page [2626](#)
CGAL::Const_oneset_iterator page ??

CGAL::Const_onese_iterator<T>

Definition

The class *Const_onese_iterator*<*T*> defines an *RandomAccessIterator* that always refers to a copy of a specific object of type *T*.

```
#include <CGAL/iterator.h>
```

Is Model for the Concepts

RandomAccessIterator

Creation

```
Const_onese_iterator<T> i( T& t);
```

creates an iterator that always refers to some copy of *t*. The copy is constructed by invoking *T*'s copy constructor and remains constant during *i*'s lifetime.

See Also

CGAL::Emptyset_iterator page [2626](#)
CGAL::Onese_iterator page ??

CGAL::Counting_iterator<Iterator, Value>

Definition

The iterator adaptor *Counting_iterator*<*Iterator*, *Value*> adds a counter to the internal iterator of type *Iterator* and defines equality of two instances in terms of this counter. It can be used to create finite sequences of possibly infinite sequences of values from input iterators.

```
#include <CGAL/iterator.h>
```

Is Model for the Concepts

InputIterator

Requirements

Iterator is a model for *InputIterator*.

Creation

```
Counting_iterator<Iterator, Value> i( std::size_t n = 0);
```

initializes the internal counter to *n* and *i* has a singular value.

```
Counting_iterator<Iterator, Value> i( Iterator j, std::size_t n = 0);
```

initializes the internal counter to *n* and *i* to *j*.

See Also

CGAL::copy_n page [2623](#)

CGAL::Insert_iterator<Container>

Definition

The output iterator *Insert_iterator*<Container> is similar to *std::insert_iterator*, but differs in that it calls the *insert()* function of the container without the iterator additional argument.

```
#include <CGAL/iterator.h>
```

Is Model for the Concepts

OutputIterator

Requirements

Container provides a member function *insert(const Container::const_reference&)*.

Creation

```
Insert_iterator<Container> i( Container &c);
```

initializes the internal container reference to *c*.

There is also a global function similar to *std::inserter*:

```
template < class Container >
```

```
Insert_iterator<Container>
```

```
inserter( Container &x)
```

Constructs *Insert_iterator*<Container>(x).

CGAL::N_step_adaptor<I,int N>

Definition

The adaptor *N_step_adaptor*<*I*,int *N*> changes the step width of the iterator or circulator class *I* to *N*. It is itself an iterator or circulator respectively. The behavior is undefined if the adaptor is used on a range [*i*, *j*) where *j* − *i* is not a multiple of *n*.

#include <CGAL/iterator.h>

Creation

N_step_adaptor<*I*,int *N*> *i*(*I* *j*); down cast.

CGAL::Filter_iterator<Iterator, Predicate>

Definition

The iterator adaptor *Filter_iterator*<*Iterator*, *Predicate*> acts as a filter on a given range. Whenever the iterator is in- or decremented, it ignores all iterators for which the given *Predicate* is true. The iterator category is the same as for *Iterator*.

Note: Boost also provides the same functionality via the *boost::filter_iterator* class. Unfortunately, the semantics chosen for accepting or rejecting elements based on the predicate's result are opposite as the semantic chosen here.

```
#include <CGAL/iterator.h>
```

Requirements

- *Iterator* is a model for *ForwardIterator*.
- *Predicate* is a functor: $Iterator \rightarrow bool$.

Creation

```
Filter_iterator<Iterator, Predicate> i;
```

```
Filter_iterator<Iterator, Predicate> i( Iterator e, Predicate p, Iterator c = e);
```

creates an iterator which filters values according to *p*. Initializes by taking the first valid iterator (according to *p*), starting at *c*, and stopping at *e* if none is found.

CGAL::Join_input_iterator_1<Iterator, Creator>

Definition

The class *Join_input_iterator_1*<*Iterator*, *Creator*> joins an iterator and a creator function object. The result is again an iterator (of the same iterator category type as the original iterator) that reads an object from the stream and applies a creator function object to that object.

```
#include <CGAL/iterator.h>
```

Is Model for the Concepts

InputIterator

Types

Join_input_iterator_1<*Iterator*, *Creator*>::*value_type*

typedef to *Creator*::*result_type*.

Creation

```
Join_input_iterator_1<Iterator, Creator> join( Iterator i, Creator creator);
```

creates a join iterator from the given iterator *i* and the functor *creator*. Applies *creator* to each item read from *i*.

```
Join_input_iterator_1<Iterator, Creator> join( Iterator i);
```

creates a join iterator from the given iterator *i* and a default constructed instance of *Creator*. The latter instance is applied to each item read from *i*.

See Also

CGAL::Creator_1<*Arg*, *Result*> page [2681](#)

CGAL::Inverse_index<IC>

Definition

The class *Inverse_index*<IC> constructs an inverse index for a given range $[i, j)$ of two iterators or circulators of type *IC*. The first element *I* in the range $[i, j)$ has the index 0. Consecutive elements are numbered incrementally. The inverse index provides a query for a given iterator or circulator *k* to retrieve its index number. *Precondition*: The iterator or circulator must be either of the random access category or the dereference operator must return stable and distinguishable addresses for the values, e.g. proxies or non-modifiable iterator with opaque values will not work.

```
#include <CGAL/iterator.h>
```

Creation

<i>Inverse_index</i> <IC> <i>inverse</i> ;	invalid index.
<i>Inverse_index</i> <IC> <i>inverse</i> (<i>IC</i> <i>i</i>);	empty inverse index initialized to start at <i>i</i> .
<i>Inverse_index</i> <IC> <i>inverse</i> (<i>IC</i> <i>i</i> , <i>IC</i> <i>j</i>);	inverse index initialized with range $[i, j)$.

Operations

<i>std::size_t</i> <i>inverse</i> [<i>IC</i> <i>k</i>]	returns inverse index of <i>k</i> . <i>Precondition</i> : <i>k</i> has been stored in the inverse index.
<i>void</i> <i>inverse.push_back</i> (<i>IC</i> <i>k</i>)	adds <i>k</i> at the end of the indices.

Implementation

For random access iterators or circulators, it is done in constant time by subtracting *i*. For other iterator categories, an STL *map* is used, which results in a $\log j - i$ query time. The comparisons are done using the operator *operator<* on pointers.

See Also

CGAL::Random_access_adaptor<IC> page [2635](#)
CGAL::Random_access_value_adaptor<IC,T> page [2636](#)

CGAL::Random_access_adaptor<IC>

Definition

The class *Random_access_adaptor*<*IC*> provides a random access for data structures. Either the data structure supports random access iterators or circulators where this class maps function calls to the iterator or circulator, or a STL *std::vector* is used to provide the random access. The iterator or circulator of the data structure are of type *IC*.

```
#include <CGAL/iterator.h>
```

Types

Random_access_adaptor<IC>::size_type size type of the STL *std::vector*.

Creation

Random_access_adaptor<*IC*> *random_access*; invalid index.

Random_access_adaptor $\langle IC \rangle$ *random_access*(*IC* *i*);

empty random access index initialized to start at *i*.

random access index initialized to the range $[i, j)$.

void

```
random_access.reserve( size_type r)
```

reserve *r* entries, if a *std::vector* is used internally.

Operations

IC *random_access*[*size_type* *n*] returns iterator or circulator to the *n*-th item.
Precondition: *n* < number of items in *random_access*.

<i>void</i>	
<i>random_access.push_back(IC k)</i>	adds <i>k</i> at the end of the indices.

See Also

<i>CGAL::Inverse_index<IC></i>	page 2634
<i>CGAL::Random_access_value_adaptor<IC,T></i>	page 2636

CGAL::Random_access_value_adaptor<IC,T>

Definition

The class *Random_access_value_adaptor*<*IC*,*T*> provides a random access for data structures. It is derived from *Random_access_adaptor*<*IC*>. Instead of returning iterators from the *operator[]* methods, it returns the dereferenced value of the iterator. The iterator or circulator of the data structure are of type *IC*. Their value type is *T*.

```
#include <CGAL/iterator.h>
```

Operations

Creation and operations see *Random_access_adaptor*<*IC*>, with the exception of:

T& *random_access*[*size_type* *n*] returns a reference to the *n*-th item.
Precondition: n < number of items in *random_access*.

See Also

CGAL::Inverse_index<*IC*> [page 2634](#)
CGAL::Random_access_adaptor<*IC*> [page 2635](#)

CGAL::swap_1

Definition

The function *swap_1* is used to swap the arguments of a functor. The result is a functor f' that calls the original functor f with the first two arguments exchanged, that is $f'(x, y, \dots) = f(y, x, \dots)$.

```
#include <CGAL/functional.h>
```

```
template < class F >
```

```
typename Swap<F,1>::Type
```

```
swap_1( F f)
```

returns a functor equivalent to f , but where the first two arguments are exchanged.

Requirement: F is a model for *AdaptableFunctor* with arity $2 \leq ar \leq 5$.

See Also

CGAL::Swap<F,i> page [2651](#)
CGAL::swap_2 page [2638](#)
CGAL::swap_3 page [2639](#)
CGAL::swap_4 page [2640](#)
AdaptableFunctor page [2656](#)

CGAL::swap_2

Definition

The function *swap_2* is used to swap the arguments of a functor. The result is a functor f' that calls the original functor f with the second and third argument exchanged, that is $f'(x, y, z, \dots) = f(x, z, y, \dots)$.

```
#include <CGAL/functional.h>
```

```
template < class F >
```

```
typename Swap<F,2>::Type
```

```
swap_2( F f)
```

returns a functor equivalent to f , but where the second and third argument are exchanged.

Requirement: F is a model for *AdaptableFunctor* with arity $3 \leq ar \leq 5$.

See Also

CGAL::Swap<F,i> page [2651](#)
CGAL::swap_1 page [2637](#)
CGAL::swap_3 page [2639](#)
CGAL::swap_4 page [2640](#)
AdaptableFunctor page [2656](#)

CGAL::swap_3

Definition

The function `swap_3` is used to swap the arguments of a functor. The result is a functor f' that calls the original functor f with the third and fourth argument exchanged, that is $f'(w, x, y, z, \dots) = f(w, x, z, y, \dots)$.

```
#include <CGAL/functional.h>
```

```
template < class F >
```

```
typename Swap<F,3>::Type
```

```
swap_3( F f)
```

returns a functor equivalent to f , but where the third and fourth argument are exchanged.

Requirement: F is a model for *AdaptableFunctor* with arity $4 \leq ar \leq 5$.

See Also

`CGAL::Swap<F,i>` page [2651](#)
`CGAL::swap_1` page [2637](#)
`CGAL::swap_2` page [2638](#)
`CGAL::swap_4` page [2640](#)
`AdaptableFunctor` page [2656](#)

CGAL::swap_4

Definition

The function *swap_4* is used to swap the arguments of a functor. The result is a functor f' that calls the original functor f with the fourth and fifth argument exchanged, that is $f'(v, w, x, y, z) = f(v, w, x, z, y)$.

```
#include <CGAL/functional.h>
```

```
template < class F >
```

```
typename Swap<F,4>::Type
```

```
    swap_4( F f)
```

returns a functor equivalent to f , but where the fourth and fifth argument are exchanged.

Requirement: F is a model for *AdaptableFunctor* with arity 5.

See Also

CGAL::Swap<F,i> page [2651](#)
 CGAL::swap_1 page [2637](#)
 CGAL::swap_2 page [2638](#)
 CGAL::swap_3 page [2639](#)
 AdaptableFunctor page [2656](#)

CGAL::bind_1

Definition

The function *bind_1* is used to bind the first argument of a functor to some specific value. The result is a functor that takes one argument less and calls the original functor where the first argument is set to the bound value.

```
#include <CGAL/functional.h>
```

```
template < class F, class A >
typename Bind< F, A, 1 >::Type    bind_1( F f, A a)
```

returns a functor equivalent to *f*, but where the first argument is bound (fixed) to *a*.

Requirement: F is a model for *AdaptableFunctor*.

See Also

<i>CGAL::Bind</i> < <i>F</i> , <i>A</i> , <i>i</i> >	page 2652
<i>CGAL::bind_2</i>	page 2642
<i>CGAL::bind_3</i>	page 2643
<i>CGAL::bind_4</i>	page 2644
<i>CGAL::bind_5</i>	page 2645
<i>AdaptableFunctor</i>	page 2656

CGAL::bind_2

Definition

The function *bind_2* is used to bind the second argument of a functor to some specific value. The result is a functor that takes one argument less and calls the original functor where the second argument is set to the bound value.

```
#include <CGAL/functional.h>
```

```
template < class F, class A >
typename Bind< F, A, 2 >::Type    bind_2( F f, A a)
```

returns a functor equivalent to *f*, but where the second argument is bound (fixed) to *a*.

Requirement: F is a model for *AdaptableFunctor*.

See Also

<i>CGAL::Bind</i> < <i>F</i> , <i>A</i> , <i>i</i> >	page 2652
<i>CGAL::bind_1</i>	page 2641
<i>CGAL::bind_3</i>	page 2643
<i>CGAL::bind_4</i>	page 2644
<i>CGAL::bind_5</i>	page 2645
<i>AdaptableFunctor</i>	page 2656

CGAL::bind_3

Definition

The function *bind_3* is used to bind the third argument of a functor to some specific value. The result is a functor that takes one argument less and calls the original functor where the third argument is set to the bound value.

```
#include <CGAL/functional.h>
```

```
template < class F, class A >
typename Bind< F, A, 3 >::Type    bind_3( F f, A a)
```

returns a functor equivalent to *f*, but where the third argument is bound (fixed) to *a*.

Requirement: F is a model for *AdaptableFunctor*.

See Also

<i>CGAL::Bind</i> < <i>F</i> , <i>A</i> , <i>i</i> >	page 2652
<i>CGAL::bind_1</i>	page 2641
<i>CGAL::bind_2</i>	page 2642
<i>CGAL::bind_4</i>	page 2644
<i>CGAL::bind_5</i>	page 2645
<i>AdaptableFunctor</i>	page 2656

CGAL::bind_4

Definition

The function *bind_4* is used to bind the fourth argument of a functor to some specific value. The result is a functor that takes one argument less and calls the original functor where the fourth argument is set to the bound value.

```
#include <CGAL/functional.h>
```

```
template < class F, class A >
typename Bind< F, A, 4 >::Type    bind_4( F f, A a)
```

returns a functor equivalent to *f*, but where the fourth argument is bound (fixed) to *a*.

Requirement: F is a model for *AdaptableFunctor*.

See Also

<i>CGAL::Bind</i> < <i>F</i> , <i>A</i> , <i>i</i> >	page 2652
<i>CGAL::bind_1</i>	page 2641
<i>CGAL::bind_2</i>	page 2642
<i>CGAL::bind_3</i>	page 2643
<i>CGAL::bind_5</i>	page 2645
<i>AdaptableFunctor</i>	page 2656

CGAL::bind_5

Definition

The function *bind_5* is used to bind the fifth argument of a functor to some specific value. The result is a functor that takes one argument less and calls the original functor where the fifth argument is set to the bound value.

```
#include <CGAL/functional.h>
```

```
template < class F, class A >
typename Bind< F, A, 5 >::Type    bind_5( F f, A a)
```

returns a functor equivalent to *f*, but where the fifth argument is bound (fixed) to *a*.

Requirement: F is a model for *AdaptableFunctor*.

See Also

<i>CGAL::Bind</i> < <i>F</i> , <i>A</i> , <i>i</i> >	page 2652
<i>CGAL::bind_1</i>	page 2641
<i>CGAL::bind_2</i>	page 2642
<i>CGAL::bind_3</i>	page 2643
<i>CGAL::bind_4</i>	page 2644
<i>AdaptableFunctor</i>	page 2656

CGAL::compare_to_less

Definition

The function *compare_to_less* is used to change a functor returning a *Comparison_result* to one which returns a *bool*. The returned functor will return *true* iff the original one returns *SMALLER*.

```
#include <CGAL/function_objects.h>
```

```
template < class F >
Compare_to_less< F >          compare_to_less( F f)
```

returns a functor equivalent to *f*, but which returns a *bool* instead of a *Comparison_result*.

Requirement: *F* is a model for *AdaptableFunctor*.

See Also

CGAL::Compare_to_less<F> page [2654](#)
AdaptableFunctor page [2656](#)

CGAL::compose

Definition

The function *compose* is used to compose functors f_0, \dots, f_n into each other, thereby creating a new functor f . The first argument f_0 always denotes the base functor, for which the remaining functors f_1, \dots, f_n provide the arguments. If we denote the arity of a functor f by $ar(f)$, then $ar(f) = \sum_{i=1}^n ar(f_i)$, i.e., the arguments of f are distributed among f_1, \dots, f_n according to their respective arity. Between one and three functors can be composed into the base functor, giving raise to a functor of arity at most five.

As an example, consider a binary functor f_0 and two functors f_1 and f_2 , with arity three and two, respectively. Composing f_1 and f_2 into f_0 yields a new functor

$$f : (x_0, x_1, x_2, x_3, x_4) \mapsto f_0(f_1(x_0, x_1, x_2), f_2(x_3, x_4))$$

with arity five.

```
#include <CGAL/functional.h>
```

```
template < class F0, class F1 >
typename Compose< F0, F1 >::Type
```

```
compose( F0 f0, F1 f1)
```

returns the functor $f0(f1(\cdot))$ with the same arity as $f1$.

Requirement: $f0$ is unary function (arity 1). $f0$ and $f1$ are models for *AdaptableFunctor*.

```
template < class F0, class F1, class F2 >
typename Compose< F0, F1, F2 >::Type
```

```
compose( F0 f0, F1 f1, F2 f2)
```

returns the functor $f0(f1(\cdot), f2(\cdot))$ with arity equal to $ar(f1) + ar(f2)$.

Requirement: $f0$ is binary function (arity 2). $f0, f1$, and $f2$ are models for *AdaptableFunctor*.

```
template < class F0, class F1, class F2, class F3 >
typename Compose< F0, F1, F2, F3 >::Type
```

```
compose( F0 f0, F1 f1, F2 f2, F3 f3)
```

returns the functor $f0(f1(\cdot), f2(\cdot), f3(\cdot))$ with arity equal to $ar(f1) + ar(f2) + ar(f3)$.

Requirement: $f0$ is ternary function (arity 3). $f0, f1, f2$, and $f3$ are models for *AdaptableFunctor*.

See Also

<i>CGAL::Compose<F0,F1,F2,F3></i>	page 2653
<i>CGAL::compose_shared</i>	page 2649
<i>AdaptableFunctor</i>	page 2656

CGAL::compose_shared

Definition

The function *compose_shared* is used to compose functors f_0, \dots, f_n into each other, thereby creating a new functor f . The first argument f_0 always denotes the base functor, for which the remaining functors f_1, \dots, f_n provide the arguments. Contrary to the function *compose*, the arguments of f are not split among f_1, \dots, f_n , but instead shared by all the functors. Therefore, all the functors f_1, \dots, f_n must have the same arity, which is also the arity of the composed functor f . Two or three functors can be composed into the base functor, giving raise to a functor of arity at most five.

As an example, consider a binary functor f_0 and two binary functors f_1 and f_2 . Composing f_1 and f_2 into f_0 yields a new binary functor

$$f : (x_0, x_1) \mapsto f_0(f_1(x_0, x_1), f_2(x_0, x_1)) .$$

```
#include <CGAL/functional.h>
```

```
template < class F0, class F1, class F2 >
typename Compose_shared< F0, F1, F2 >::Type
```

```
compose_shared( F0 f0, F1 f1, F2 f2)
```

returns the functor $f_0(f_1(\cdot), f_2(\cdot))$ with the same arity as f_1 (and f_2).

Requirement: f_0 is *AdaptableFunctor* of arity 2. f_1 and f_2 are *AdaptableFunctors* having the same arity.

```
template < class F0, class F1, class F2, class F3 >
typename Compose_shared< F0, F1, F2, F3 >::Type
```

```
compose_shared( F0 f0, F1 f1, F2 f2, F3 f3)
```

returns the functor $f_0(f_1(\cdot), f_2(\cdot), f_3(\cdot))$ with the same arity as f_1 (and f_2, f_3).

Requirement: f_0 is *AdaptableFunctor* of arity 3. f_1, f_2 , and f_3 are *AdaptableFunctors* having the same arity.

See Also

CGAL::Compose_shared<F0,F1,F2,F3> page [2655](#)
 CGAL::compose page [2647](#)
 AdaptableFunctor page [2656](#)

CGAL::negate

Definition

The function *negate* is a functor adaptor. For a given functor *f*, it creates a new functor *f'* which is the negation of *f*. That is, $f' = !f$.

```
#include <CGAL/functional.h>
```

```
template < typename F >
```

```
typename Compose< std::logical_not<typename F::result_type>, F >::Type
```

```
    negate( F f)
```

returns the functor $!f$ with the same arity as *f*.

Requirement: *f* is a model for *AdaptableFunctor*. Unary negation is defined for *F::result_type*.

See Also

CGAL::Compose<*F0*,*F1*,*F2*,*F3*> [page 2653](#)

CGAL::compose [page 2647](#)

AdaptableFunctor [page 2656](#)

Definition

The class *Bind*<*F*,*A*,*i*> is used to specify the type of a bound functor of type *F*, i.e., where the *i*-th argument is bound to some object of type *A*. The class is used in conjunction with the *bind* functions.

```
#include <CGAL/functional.h>
```

Types

$Bind\langle F, A, i \rangle :: Type$ the bound type.

Notes

This class encapsulates the differences in implementation of the binders across various platforms. But in any case, *Type* refers to a model of *AdaptableFunctor*.

See Also

<i>CGAL::bind_1</i>	page 2641
<i>CGAL::bind_2</i>	page 2642
<i>CGAL::bind_3</i>	page 2643
<i>CGAL::bind_4</i>	page 2644
<i>CGAL::bind_5</i>	page 2645
<i>AdaptableFunctor</i>	page 2656

CGAL::Compose<F0,F1,F2,F3>

Definition

The class *Compose*<*F0*,*F1*,*F2*,*F3*> is used to specify the type of a composed functor, i.e., the functor that results from composing functors of type *F1*, *F2*, and *F3*, into a functor of type *F0*. The arguments *F2* and *F3* are optional, such that between two and four functors can participate in the composition. The class is used in conjunction with the *compose* function; see there for an explanation on how exactly the functors are combined.

```
#include <CGAL/functional.h>
```

Types

Compose<*F0*,*F1*,*F2*,*F3*>:: *Type* type of the composed functor.

Notes

This class encapsulates the differences in implementation of the composers across various platforms. But in any case, *Type* refers to a model of *AdaptableFunctor*.

See Also

CGAL::compose page [2647](#)
AdaptableFunctor page [2656](#)

The class *Compare_to_less<F>* is used to convert a functor which returns a *Comparison_result* to a predicate (returning bool) : it will return true iff the return value of *F* is *SMALLER*. The class is used in conjunction with the *compare_to_less* function; see there for an explanation on how exactly the functors are combined.

Types

<i>Compare_to_less</i> < <i>F</i> >:: <i>Type</i>	type of the composed functor.
---------------------------------------------------	-------------------------------

<i>CGAL::compare_to_less</i>	page 2646
<i>AdaptableFunctor</i>	page 2656

CGAL::Compose_shared<F0,F1,F2,F3>

Definition

The class *Compose_shared*<*F0*,*F1*,*F2*,*F3*> is used to specify the type of a composed functor, i.e., the functor that results from composing functors of type *F1*, *F2*, and *F3*, into a functor of type *F0*. The arguments *F2* and *F3* are optional, such that between two and four functors can participate in the composition. The class is used in conjunction with the *compose_shared* function; see there for an explanation on how exactly the functors are combined.

```
#include <CGAL/functional.h>
```

Types

Compose_shared<*F0*,*F1*,*F2*,*F3*>:: *Type* type of the composed functor.

Notes

This class encapsulates the differences in implementation of the composers across various platforms. But in any case, *Type* refers to a model of *AdaptableFunctor*.

See Also

CGAL::compose_shared [page 2649](#)
AdaptableFunctor [page 2656](#)

AdaptableFunctor

Definition

The concept `AdaptableFunctor` defines an adaptable functor, i.e., a functor that can be used with function object adaptors such as binders and composers.

Types

AdaptableFunctor:: result_type

return type of the functor.

AdaptableFunctor:: Arity

defines the arity of the functor, i.e., the number of arguments it takes. The class has to be a specialization of `CGAL::Arity_tag<int>`, where the template parameter corresponds to the arity of the functor, e.g. `CGAL::Arity_tag<2>` for binary functors.

Operations

type0 *f.operator()(type1 a1, type2 a2, ..., typen an) const*

(as many arguments as defined by *Arity*)
returns *f(a1,...an)*.

Notes

Alternatively, the type *Arity* can be defined in a specialization of `CGAL::Arity_traits<>` for the functor. This is useful where existing classes cannot be changed easily, e.g. the functors from the standard library.

Has Models

All functors from the standard library, and all functors from the lower dimensional CGAL kernels. For all kernel functors, their arity is listed in the documentation. Some (few) of them are overloaded with operators of different arities; in this case one of these arities has been chosen as default arity. If you want to adapt the functor to a different arity, use the functor adaptor `CGAL::Set_arity<F,a>`.

See Also

`CGAL::Arity_tag<int>` page [2658](#)
`CGAL::Arity_traits<F>` page [2659](#)
`CGAL::Set_arity<F,a>` page [2660](#)
`CGAL::set_arity_0` page [2661](#)
`CGAL::set_arity_1` page [2662](#)

<i>CGAL::set_arity_2</i>	page 2663
<i>CGAL::set_arity_3</i>	page 2664
<i>CGAL::set_arity_4</i>	page 2665
<i>CGAL::set_arity_5</i>	page 2666

CGAL::Arity_tag<int>

Definition

The class *Arity_tag<int>* is used to define the arity of a functor, i.e., the number of arguments it takes. It is used as a compile time tag only, that is objects of this type are never created anywhere.

```
#include <CGAL/functional_base.h>
```

See Also

AdaptableFunctor.....page [2656](#)

CGAL::Arity_traits<F>

Definition

The class *Arity_traits*<*F*> is used to define the arity of a functor class *F*, i.e., the number of arguments it takes. It is used as a compile time tag only, that is objects of this type are never created anywhere. Specializations of *Arity_traits*<*F*> can be defined, where existing functors cannot be changed easily to contain their *Arity* type.

```
#include <CGAL/functional_base.h>
```

```
Arity_traits<F>:: Arity          F::Arity
```

See Also

AdaptableFunctor.....page [2656](#)
 CGAL::Arity_tag<int>.....page [2658](#)

Definition

The class *Set_arity*<*F*,*a*> is used to specify the type of a functor of type *F* whose arity has been set explicitly to *a*. The class is used in conjunction with the *set_arity* functions.

```
#include <CGAL/functional.h>
```

Types

$Set_arity\langle F, a \rangle :: Type$ the functor type.

Notes

This class encapsulates the differences in implementation across various platforms. But in any case, *Type* refers to a model of *AdaptableFunctor* with arity *a*.

See Also

<i>CGAL::set_arity_0</i>	page 2661
<i>CGAL::set_arity_1</i>	page 2662
<i>CGAL::set_arity_2</i>	page 2663
<i>CGAL::set_arity_3</i>	page 2664
<i>CGAL::set_arity_4</i>	page 2665
<i>CGAL::set_arity_5</i>	page 2666
<i>AdaptableFunctor</i>	page 2656

CGAL::set_arity_0

Definition

The function *set_arity_0* is used to set the arity of a functor to zero. The result is a functor that takes no arguments and calls the original functor with no arguments.

```
#include <CGAL/functional.h>
```

```
template < class F >
Set_arity< F, 0 >::Type    set_arity_0( F f)
```

returns a functor equivalent to *f*, but which has arity zero.

Requirement: F is a model for *AdaptableFunctor*.

See Also

<i>CGAL::Set_arity</i> < <i>F</i> , <i>a</i> >	page 2660
<i>CGAL::set_arity_1</i>	page 2662
<i>CGAL::set_arity_2</i>	page 2663
<i>CGAL::set_arity_3</i>	page 2664
<i>CGAL::set_arity_4</i>	page 2665
<i>CGAL::set_arity_5</i>	page 2666
<i>AdaptableFunctor</i>	page 2656

CGAL::set_arity_1

Definition

The function *set_arity_1* is used to set the arity of a functor to one. The result is a functor that takes one argument and calls the original functor with this argument.

```
#include <CGAL/functional.h>
```

```
template < class F >
Set_arity< F, 1 >::Type    set_arity_1( F f)
```

returns a functor equivalent to *f*, but which has arity one.

Requirement: F is a model for *AdaptableFunctor*.

See Also

<i>CGAL::Set_arity</i> < <i>F</i> , <i>a</i> >	page 2660
<i>CGAL::set_arity_0</i>	page 2661
<i>CGAL::set_arity_2</i>	page 2663
<i>CGAL::set_arity_3</i>	page 2664
<i>CGAL::set_arity_4</i>	page 2665
<i>CGAL::set_arity_5</i>	page 2666
<i>AdaptableFunctor</i>	page 2656

CGAL::set_arity_2

Definition

The function *set_arity_2* is used to set the arity of a functor to two. The result is a functor that takes two arguments and calls the original functor with these arguments.

```
#include <CGAL/functional.h>
```

```
template < class F >
Set_arity< F, 2 >::Type    set_arity_2( F f)
```

returns a functor equivalent to *f*, but which has arity two.

Requirement: F is a model for *AdaptableFunctor*.

See Also

<i>CGAL::Set_arity</i> < <i>F</i> , <i>a</i> >	page 2660
<i>CGAL::set_arity_0</i>	page 2661
<i>CGAL::set_arity_1</i>	page 2662
<i>CGAL::set_arity_3</i>	page 2664
<i>CGAL::set_arity_4</i>	page 2665
<i>CGAL::set_arity_5</i>	page 2666
<i>AdaptableFunctor</i>	page 2656

CGAL::set_arity_3

Definition

The function *set_arity_3* is used to set the arity of a functor to three. The result is a functor that takes three arguments and calls the original functor with these arguments.

```
#include <CGAL/functional.h>
```

```
template < class F >
Set_arity< F, 3 >::Type    set_arity_3( F f)
```

returns a functor equivalent to *f*, but which has arity three.

Requirement: F is a model for *AdaptableFunctor*.

See Also

<i>CGAL::Set_arity</i> < <i>F</i> , <i>a</i> >	page 2660
<i>CGAL::set_arity_0</i>	page 2661
<i>CGAL::set_arity_1</i>	page 2662
<i>CGAL::set_arity_2</i>	page 2663
<i>CGAL::set_arity_4</i>	page 2665
<i>CGAL::set_arity_5</i>	page 2666
<i>AdaptableFunctor</i>	page 2656

CGAL::set_arity_4

Definition

The function *set_arity_4* is used to set the arity of a functor to four. The result is a functor that takes four arguments and calls the original functor with these arguments.

```
#include <CGAL/functional.h>
```

```
template < class F >
Set_arity< F, 4 >::Type    set_arity_4( F f)
```

returns a functor equivalent to *f*, but which has arity four.

Requirement: F is a model for *AdaptableFunctor*.

See Also

<i>CGAL::Set_arity</i> < <i>F</i> , <i>a</i> >	page 2660
<i>CGAL::set_arity_0</i>	page 2661
<i>CGAL::set_arity_1</i>	page 2662
<i>CGAL::set_arity_2</i>	page 2663
<i>CGAL::set_arity_3</i>	page 2664
<i>CGAL::set_arity_5</i>	page 2666
<i>AdaptableFunctor</i>	page 2656

CGAL::set_arity_5

Definition

The function *set_arity_5* is used to set the arity of a functor to five. The result is a functor that takes five arguments and calls the original functor with these arguments.

```
#include <CGAL/functional.h>
```

```
template < class F >
Set_arity< F, 5 >::Type    set_arity_5( F f)
```

returns a functor equivalent to *f*, but which has arity five.

Requirement: F is a model for *AdaptableFunctor*.

See Also

<i>CGAL::Set_arity</i> < <i>F</i> , <i>a</i> >	page 2660
<i>CGAL::set_arity_0</i>	page 2661
<i>CGAL::set_arity_1</i>	page 2662
<i>CGAL::set_arity_2</i>	page 2663
<i>CGAL::set_arity_3</i>	page 2664
<i>CGAL::set_arity_4</i>	page 2665
<i>AdaptableFunctor</i>	page 2656

Projection_object

Definition

The concept `Projection_object` is modeled after the STL concept *UnaryFunction*, but takes also care of (const) references.

Projection_object:: argument_type argument type.

Projection_object:: result_type result type.

Creation

Projection_object o; default constructor.

Operations

result_type& *o.operator()(argument_type &) const*
const result_type& *o.operator()(const argument_type &) const*

Has Models

<i>CGAL::Identity<Value></i>	page 2668
<i>CGAL::Dereference<Value></i>	page 2669
<i>CGAL::Get_address<Value></i>	page 2670
<i>CGAL::Cast_function_object<Arg, Result></i>	page 2671
<i>CGAL::Project_vertex<Node></i>	page 2672
<i>CGAL::Project_facet<Node></i>	page 2673
<i>CGAL::Project_point<Node></i>	page 2674
<i>CGAL::Project_normal<Node></i>	page 2675
<i>CGAL::Project_plane<Node></i>	page 2676
<i>CGAL::Project_next<Node></i>	page 2677
<i>CGAL::Project_prev<Node></i>	page 2678
<i>CGAL::Project_next_opposite<Node></i>	page 2679
<i>CGAL::Project_opposite_prev<Node></i>	page 2680

CGAL::Identity<Value>

Definition

The class *Identity*<*Value*> represents the identity function on *Value*.

#include <*CGAL/function_objects.h*>

Is Model for the Concepts

Projection_object page [2667](#)

Identity<*Value*>::*argument_type* typedef to *Value*.

Identity<*Value*>::*result_type* typedef to *Value*.

Creation

Identity<*Value*> *o*; default constructor.

Operations

result_type& *o.operator()*(*argument_type*& *x*) *const*

returns *x*.

const result_type& *o.operator()*(*const argument_type*& *x*) *const*

returns *x*.

CGAL::Dereference<Value>

Definition

The class *Dereference*<Value> dereferences a pointer (*operator**).

#include <CGAL/function_objects.h>

Is Model for the Concepts

Projection_object page [2667](#)

Dereference<Value>::*argument_type* typedef to *Value**.

Dereference<Value>::*result_type* typedef to *Value*.

Creation

Dereference<Value> *o*; default constructor.

Operations

result_type& *o.operator()*(*argument_type*& *x*) *const*

returns **x*.

const result_type& *o.operator()*(*const argument_type*& *x*) *const*

returns **x*.

CGAL::Get_address<Value>

Definition

The class *Get_address<Value>* gets the address of an lvalue (*operator&*).

#include <CGAL/function_objects.h>

Is Model for the Concepts

Projection_object page [2667](#)

Get_address<Value>:: argument_type typedef to *Value*.

Get_address<Value>:: result_type typedef to *Value**.

Creation

Get_address<Value> o; default constructor.

Operations

result_type& *o.operator()(argument_type& x) const*

returns *&x*.

const result_type& *o.operator()(const argument_type& x) const*

returns *&x*.

CGAL::Cast_function_object<Arg, Result>

Definition

The class *Cast_function_object*<Arg, Result> applies a C-style type cast to its argument.

#include <CGAL/function_objects.h>

Is Model for the Concepts

Projection_object page [2667](#)

Cast_function_object<Arg, Result>::*argument_type*

typedef to Arg.

Cast_function_object<Arg, Result>::*result_type*

typedef to Result.

Creation

Cast_function_object<Arg, Result> o; default constructor.

Operations

result_type& o.operator()(*argument_type*& x) const

returns (Result)x.

const *result_type*& o.operator()(const *argument_type*& x) const

returns (Result)x.

CGAL::Project_vertex<Node>

Definition

The class *Project_vertex<Node>* calls the member function *vertex()* on an instance of type *Node*.

```
#include <CGAL/function_objects.h>
```

Is Model for the Concepts

Projection_object page [2667](#)

Project_vertex<Node>:: argument_type typedef to *Node*.

Project_vertex<Node>:: result_type typedef to *Node::Vertex*.

Creation

Project_vertex<Node> o; default constructor.

Operations

result_type& *o.operator()(argument_type& n) const*

returns *n.vertex()*.

const result_type& *o.operator()(const argument_type& n) const*

returns *n.vertex()*.

CGAL::Project_facet<Node>

Definition

The class *Project_facet<Node>* calls the member function *facet()* on an instance of type *Node*.

```
#include <CGAL/function_objects.h>
```

Is Model for the Concepts

Projection_object page [2667](#)

Project_facet<Node>:: argument_type typedef to *Node*.

Project_facet<Node>:: result_type typedef to *Node::Facet*.

Creation

Project_facet<Node> o; default constructor.

Operations

result_type& *o.operator()(argument_type& n) const*

returns *n.facet()*.

const result_type& *o.operator()(const argument_type& n) const*

returns *n.facet()*.

CGAL::Project_point<Node>

Definition

The class *Project_point*<*Node*> calls the member function *point()* on an instance of type *Node*.

```
#include <CGAL/function_objects.h>
```

Is Model for the Concepts

Projection_object page [2667](#)

Project_point<*Node*>::*argument_type* typedef to *Node*.

Project_point<*Node*>::*result_type* typedef to *Node*::*Point*.

Creation

Project_point<*Node*> *o*; default constructor.

Operations

result_type& *o.operator()(argument_type& n) const*

 returns *n.point()*.

const result_type& *o.operator()(const argument_type& n) const*

 returns *n.point()*.

CGAL::Project_normal<Node>

Definition

The class *Project_normal<Node>* calls the member function *normal()* on an instance of type *Node*.

```
#include <CGAL/function_objects.h>
```

Is Model for the Concepts

Projection_object page [2667](#)

Project_normal<Node>:: argument_type typedef to *Node*.

Project_normal<Node>:: result_type typedef to *Node::Normal*.

Creation

Project_normal<Node> *o*; default constructor.

Operations

result_type& *o.operator()(argument_type& n) const*

returns *n.normal()*.

const result_type& *o.operator()(const argument_type& n) const*

returns *n.normal()*.

CGAL::Project_plane<Node>

Definition

The class *Project_plane<Node>* calls the member function *plane()* on an instance of type *Node*.

```
#include <CGAL/function_objects.h>
```

Is Model for the Concepts

Projection_object page [2667](#)

Project_plane<Node>:: argument_type typedef to *Node*.

Project_plane<Node>:: result_type typedef to *Node::Plane*.

Creation

Project_plane<Node> o; default constructor.

Operations

result_type& *o.operator()(argument_type& n) const*

returns *n.plane()*.

const result_type& *o.operator()(const argument_type& n) const*

returns *n.plane()*.

CGAL::Project_next<Node>

Definition

The class *Project_next*<*Node*> calls the member function *next()* on an instance of type *Node*.

```
#include <CGAL/function_objects.h>
```

Is Model for the Concepts

Projection_object page [2667](#)

Project_next<*Node*>::*argument_type* typedef to *Node**.

Project_next<*Node*>::*result_type* typedef to *Node**.

Creation

Project_next<*Node*> *o*; default constructor.

Operations

result_type& *o.operator()(argument_type& n) const*

returns *n->next()*.

const result_type& *o.operator()(const argument_type& n) const*

returns *n->next()*.

CGAL::Project_prev<Node>

Definition

The class *Project_prev<Node>* calls the member function *prev()* on an instance of type *Node*.

```
#include <CGAL/function_objects.h>
```

Is Model for the Concepts

Projection_object page [2667](#)

Project_prev<Node>:: argument_type typedef to *Node**.

Project_prev<Node>:: result_type typedef to *Node**.

Creation

Project_prev<Node> o; default constructor.

Operations

result_type& *o.operator()(argument_type& n) const*

returns *n->prev()*.

const result_type& *o.operator()(const argument_type& n) const*

returns *n->prev()*.

CGAL::Project_next_opposite<Node>

Definition

The class *Project_next_opposite<Node>* calls the member functions *next()*->*opposite()* on an instance of type *Node*.

```
#include <CGAL/function_objects.h>
```

Is Model for the Concepts

Projection_object page [2667](#)

Project_next_opposite<Node>:: argument_type typedef to *Node**.

Project_next_opposite<Node>:: result_type typedef to *Node**.

Creation

Project_next_opposite<Node> *o*; default constructor.

Operations

result_type& *o.operator()(argument_type& n) const*

returns *n->next()*->*opposite()*.

const result_type& *o.operator()(const argument_type& n) const*

returns *n->next()*->*opposite()*.

CGAL::Project_opposite_prev<Node>

Definition

The class *Project_opposite_prev*<Node> calls the member functions *opposite()*->*prev()* on an instance of type *Node*.

```
#include <CGAL/function_objects.h>
```

Is Model for the Concepts

Projection_object page [2667](#)

Project_opposite_prev<Node>:: *argument_type*

typedef to *Node**.

Project_opposite_prev<Node>:: *result_type*

typedef to *Node**.

Creation

Project_opposite_prev<Node> *o*;

default constructor.

Operations

result_type& *o.operator()*(*argument_type*& *n*) *const*

returns *n*->*opposite()*->*prev()*.

const result_type& *o.operator()*(*const argument_type*& *n*) *const*

returns *n*->*opposite()*->*prev()*.

CGAL::Creator_1<Arg, Result>

Definition

The concept *Creator₁* $\langle Arg, Result \rangle$ defines types and operations for creating objects from one argument.

```
#include <CGAL/function_objects.h>
```

Requirements

Arg is convertible to *Result*.

Creator_1 $\langle Arg, Result \rangle :: argument_type$ type of argument.

Creator_I $\langle Arg, Result \rangle :: result_type$ type of object to create.

result_type *c.operator()*(*argument_type a*) *const*

returns *result_type(a)*.

CGAL::Creator_2<Arg1, Arg2, Result>

Definition

The concept *Creator_2<Arg1, Arg2, Result>* defines types and operations for creating objects from two arguments.

```
#include <CGAL/function_objects.h>
```

Requirements

Result defines a corresponding constructor.

Creator_2<Arg1, Arg2, Result>:: argument1_type type of first argument.

Creator_2<Arg1, Arg2, Result>:: argument2_type type of second argument.

Creator_2<Arg1, Arg2, Result>:: result_type type of object to create.

result_type *c.operator()(argument_type1 a1, argument_type2 a2) const*
returns *result_type(a1, a2)*.

CGAL::Creator_3<Arg1, Arg2, Arg3, Result>

Definition

The concept *Creator_3<Arg1, Arg2, Arg3, Result>* defines types and operations for creating objects from three arguments.

```
#include <CGAL/function_objects.h>
```

Requirements

Result defines a corresponding constructor.

```
Creator_3<Arg1, Arg2, Arg3, Result>:: argument1_type
```

type of first argument.

```
Creator_3<Arg1, Arg2, Arg3, Result>:: argument2_type
```

type of second argument.

```
Creator_3<Arg1, Arg2, Arg3, Result>:: argument3_type
```

type of third argument.

```
Creator_3<Arg1, Arg2, Arg3, Result>:: result_type
```

type of object to create.

```
result_type    c.operator()( argument_type1 a1, argument_type2 a2, argument_type3 a3) const
```

returns *result_type(a1, a2, a3)*.

CGAL::Creator_4<Arg1, Arg2, Arg3, Arg4, Result>

Definition

The concept *Creator_4<Arg1, Arg2, Arg3, Arg4, Result>* defines types and operations for creating objects from four arguments.

```
#include <CGAL/function_objects.h>
```

Requirements

Result defines a corresponding constructor.

Creator_4<Arg1, Arg2, Arg3, Arg4, Result>:: argument1_type
type of first argument.

Creator_4<Arg1, Arg2, Arg3, Arg4, Result>:: argument2_type
type of second argument.

Creator_4<Arg1, Arg2, Arg3, Arg4, Result>:: argument3_type
type of third argument.

Creator_4<Arg1, Arg2, Arg3, Arg4, Result>:: argument4_type
type of 4th argument.

Creator_4<Arg1, Arg2, Arg3, Arg4, Result>:: result_type
type of object to create.

```
result_type    c.operator()(
                argument_type1 a1,
                argument_type2 a2,
                argument_type3 a3,
                argument_type4 a4) const
```

returns *result_type(a1, a2, a3, a4)*.

CGAL::Creator_5<Arg1, Arg2, Arg3, Arg4, Arg5, Result>

Definition

The concept *Creator_5<Arg1, Arg2, Arg3, Arg4, Arg5, Result>* defines types and operations for creating objects from five arguments.

```
#include <CGAL/function_objects.h>
```

Requirements

Result defines a corresponding constructor.

```
Creator_5<Arg1, Arg2, Arg3, Arg4, Arg5, Result>:: argument1_type  
                                     type of first argument.
```

```
Creator_5<Arg1, Arg2, Arg3, Arg4, Arg5, Result>:: argument2_type  
                                     type of second argument.
```

```
Creator_5<Arg1, Arg2, Arg3, Arg4, Arg5, Result>:: argument3_type  
                                     type of third argument.
```

```
Creator_5<Arg1, Arg2, Arg3, Arg4, Arg5, Result>:: argument4_type  
                                     type of 4th argument.
```

```
Creator_5<Arg1, Arg2, Arg3, Arg4, Arg5, Result>:: argument5_type  
                                     type of 5th argument.
```

```
Creator_5<Arg1, Arg2, Arg3, Arg4, Arg5, Result>:: result_type  
                                     type of object to create.
```

```
result_type    c.operator()(  
                    argument_type1 a1,  
                    argument_type2 a2,  
                    argument_type3 a3,  
                    argument_type4 a4,  
                    argument_type5 a5) const  
                                     returns result_type(a1, a2, a3, a4, a5).
```

CGAL::Creator_uniform_2<Arg, Result>

Definition

The concept *Creator_uniform_2*<Arg, Result> defines types and operations for creating objects from two arguments of the same type.

```
#include <CGAL/function_objects.h>
```

Requirements

Result defines a constructor from two *Arg* arguments.

Creator_uniform_2<Arg, Result>:: *argument_type*

type of arguments; typedef to *Arg*.

Creator_uniform_2<Arg, Result>:: *result_type* type of object to create; typedef to *Result*.

result_type *c.operator()(argument_type a1, argument_type a2) const*

returns *result_type(a1, a2)*.

CGAL::Creator_uniform_3<Arg, Result>

Definition

The concept *Creator_uniform_3*<Arg, Result> defines types and operations for creating objects from three arguments of the same type.

```
#include <CGAL/function_objects.h>
```

Requirements

Result defines a constructor from three *Arg* arguments.

Creator_uniform_3<Arg, Result>:: *argument_type*

type of arguments; typedef to *Arg*.

Creator_uniform_3<Arg, Result>:: *result_type* type of object to create; typedef to *Result*.

result_type *c.operator()(argument_type a1, argument_type a2, argument_type a3) const*

returns *result_type(a1, a2, a3)*.

CGAL::Creator_uniform_4<Arg, Result>

Definition

The concept *Creator_uniform_4<Arg, Result>* defines types and operations for creating objects from four arguments of the same type.

```
#include <CGAL/function_objects.h>
```

Requirements

Result defines a constructor from four *Arg* arguments.

Creator_uniform_4<Arg, Result>:: argument_type

type of arguments; typedef to *Arg*.

Creator_uniform_4<Arg, Result>:: result_type type of object to create; typedef to *Result*.

```
result_type      c.operator()(
                    argument_type a1,
                    argument_type a2,
                    argument_type a3,
                    argument_type a4) const
```

returns *result_type(a1, a2, a3, a4)*.

CGAL::Creator_uniform_5<Arg, Result>

Definition

The concept *Creator_uniform_5<Arg, Result>* defines types and operations for creating objects from five arguments of the same type.

```
#include <CGAL/function_objects.h>
```

Requirements

Result defines a constructor from five *Arg* arguments.

```
Creator_uniform_5<Arg, Result>:: argument_type
                                     type of arguments; typedef to Arg.
```

```
Creator_uniform_5<Arg, Result>:: result_type    type of object to create; typedef to Result.
```

```
result_type    c.operator()(
                argument_type a1,
                argument_type a2,
                argument_type a3,
                argument_type a4,
                argument_type a5) const
                                     returns result_type(a1, a2, a3, a4, a5).
```

CGAL::Creator_uniform_6<Arg, Result>

Definition

The concept *Creator_uniform_6<Arg, Result>* defines types and operations for creating objects from six arguments of the same type.

```
#include <CGAL/function_objects.h>
```

Requirements

Result defines a constructor from six *Arg* arguments.

Creator_uniform_6<Arg, Result>:: argument_type

type of arguments; typedef to *Arg*.

Creator_uniform_6<Arg, Result>:: result_type type of object to create; typedef to *Result*.

```
result_type      c.operator()(
                    argument_type a1,
                    argument_type a2,
                    argument_type a3,
                    argument_type a4,
                    argument_type a5,
                    argument_type a6) const
```

returns *result_type(a1, a2, a3, a4, a5, a6)*.

CGAL::Creator_uniform_7<Arg, Result>

Definition

The concept *Creator_uniform_7*<Arg, Result> defines types and operations for creating objects from seven arguments of the same type.

```
#include <CGAL/function_objects.h>
```

Requirements

Result defines a constructor from seven *Arg* arguments.

```
Creator_uniform_7<Arg, Result>:: argument_type
                                     type of arguments; typedef to Arg.
```

```
Creator_uniform_7<Arg, Result>:: result_type  type of object to create; typedef to Result.
```

```
result_type  c.operator()(
               argument_type a1,
               argument_type a2,
               argument_type a3,
               argument_type a4,
               argument_type a5,
               argument_type a6,
               argument_type a7) const
                                     returns result_type(a1, a2, a3, a4, a5, a6, a7).
```

CGAL::Creator_uniform_8<Arg, Result>

Definition

The concept *Creator_uniform_8<Arg, Result>* defines types and operations for creating objects from eight arguments of the same type.

```
#include <CGAL/function_objects.h>
```

Requirements

Result defines a constructor from eight *Arg* arguments.

```
Creator_uniform_8<Arg, Result>:: argument_type
                                     type of arguments; typedef to Arg.
```

```
Creator_uniform_8<Arg, Result>:: result_type    type of object to create; typedef to Result.
```

```
result_type    c.operator()(
                argument_type a1,
                argument_type a2,
                argument_type a3,
                argument_type a4,
                argument_type a5,
                argument_type a6,
                argument_type a7,
                argument_type a8) const
                                     returns result_type(a1, a2, a3, a4, a5, a6, a7, a8).
```

CGAL::Creator_uniform_9<Arg, Result>

Definition

The concept *Creator_uniform_9<Arg, Result>* defines types and operations for creating objects from nine arguments of the same type.

```
#include <CGAL/function_objects.h>
```

Requirements

Result defines a constructor from nine *Arg* arguments.

Creator_uniform_9<Arg, Result>:: argument_type

type of arguments; typedef to *Arg*.

Creator_uniform_9<Arg, Result>:: result_type type of object to create; typedef to *Result*.

```
result_type      c.operator()(
    argument_type a1,
    argument_type a2,
    argument_type a3,
    argument_type a4,
    argument_type a5,
    argument_type a6,
    argument_type a7,
    argument_type a8,
    argument_type a9) const
```

returns *result_type(a1, a2, a3, a4, a5, a6, a7, a8, a9)*.

CGAL::Creator_uniform_d<Arg, Result>

Definition

The concept *Creator_uniform_d*<*Arg*, *Result*> defines types and operations for creating objects from two arguments of the same type.

```
#include <CGAL/function_objects.h>
```

Requirements

Result defines a constructor from three arguments: one *d* dimension and two *Arg* arguments.

Creator_uniform_d<*Arg*, *Result*>:: *argument_type*

type of arguments; typedef to *Arg*.

Creator_uniform_d<*Arg*, *Result*>:: *result_type* type of object to create; typedef to *Result*.

result_type *c.operator()*(*argument_type* *a1*, *argument_type* *a2*) *const*

returns *result_type*(*d*, *a1*, *a2*).

CGAL::Twotuple<T>

Definition

The *Twotuple*<*T*> class stores a homogeneous (same type) pair of objects of type *T*. A *Twotuple*<*T*> is much like a container, in that it "owns" its elements. It is not actually a model of container, though, because it does not support the standard methods (such as iterators) for accessing the elements of a container.

```
#include <CGAL/Twotuple.h>
```

Requirements

T must be *Assignable*.

Types

```
typedef T      value_type;
```

Variables

<i>T</i>	<i>e0</i> ;	first element
<i>T</i>	<i>e1</i> ;	second element

Creation

<i>Twotuple</i> < <i>T</i> > <i>t</i> ;	introduces a <i>Twotuple</i> < <i>T</i> > using the default constructor of the elements.
-----------------------------------------	------------------------------------------------------------------------------------------

<i>Twotuple</i> < <i>T</i> > <i>t</i> (<i>T</i> <i>x</i> , <i>T</i> <i>y</i>);	constructs a <i>Twotuple</i> < <i>T</i> > such that <i>e0</i> is constructed from <i>x</i> and <i>e1</i> is constructed from <i>y</i> .
----------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------

CGAL::Threetuple<T>

Definition

The *Threetuple*<*T*> class stores a homogeneous (same type) triple of objects of type *T*. A *Threetuple*<*T*> is much like a container, in that it "owns" its elements. It is not actually a model of container, though, because it does not support the standard methods (such as iterators) for accessing the elements of a container.

```
#include <CGAL/Threetuple.h>
```

Requirements

T must be *Assignable*.

Types

```
typedef T      value_type;
```

Variables

<i>T</i>	<i>e0</i> ;	first element
<i>T</i>	<i>e1</i> ;	second element
<i>T</i>	<i>e2</i> ;	third element

Creation

<i>Threetuple</i> < <i>T</i> > <i>t</i> ;	introduces a <i>Threetuple</i> < <i>T</i> > using the default constructor of the elements.
-------------------------------------------	--------------------------------------------------------------------------------------------

<i>Threetuple</i> < <i>T</i> > <i>t</i> (<i>T</i> <i>x</i> , <i>T</i> <i>y</i> , <i>T</i> <i>z</i>);	constructs a <i>Threetuple</i> < <i>T</i> > such that <i>e0</i> is constructed from <i>x</i> , <i>e1</i> is constructed from <i>y</i> and <i>e2</i> is constructed from <i>z</i> .
--------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

CGAL::Fourtuple<T>

Definition

The *Fourtuple<T>* class stores a homogeneous (same type) fourtuple of objects of type *T*. A *Fourtuple<T>* is much like a container, in that it "owns" its elements. It is not actually a model of container, though, because it does not support the standard methods (such as iterators) for accessing the elements of a container.

```
#include <CGAL/Fourtuple.h>
```

Requirements

T must be *Assignable*.

Types

```
typedef T      value_type;
```

Variables

<i>T</i>	<i>e0</i> ;	first element
<i>T</i>	<i>e1</i> ;	second element
<i>T</i>	<i>e2</i> ;	third element
<i>T</i>	<i>e3</i> ;	fourth element

Creation

<i>Fourtuple<T></i> <i>t</i> ;	introduces a <i>Fourtuple<T></i> using the default constructor of the elements.
--------------------------------------	---------------------------------------------------------------------------------------

<i>Fourtuple<T></i> <i>t</i> (<i>T</i> <i>x</i> , <i>T</i> <i>y</i> , <i>T</i> <i>z</i> , <i>T</i> <i>t</i>);	constructs a <i>Fourtuple<T></i> such that <i>e0</i> is constructed from <i>x</i> , <i>e1</i> from <i>y</i> , <i>e2</i> from <i>z</i> and <i>e3</i> from <i>t</i> .
-----------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------

CGAL::Sextuple<T>

Definition

The *Sextuple*<*T*> class stores a homogeneous (same type) sextuple of objects of type *T*. A *Sextuple*<*T*> is much like a container, in that it "owns" its elements. It is not actually a model of container, though, because it does not support the standard methods (such as iterators) for accessing the elements of a container.

```
#include <CGAL/Sextuple.h>
```

Requirements

T must be *Assignable*.

Types

```
typedef T          value_type;
```

Variables

<i>T</i>	<i>e0</i> ;	first element
<i>T</i>	<i>e1</i> ;	second element
<i>T</i>	<i>e2</i> ;	third element
<i>T</i>	<i>e3</i> ;	fourth element
<i>T</i>	<i>e4</i> ;	fifth element
<i>T</i>	<i>e5</i> ;	sixth element

Creation

Sextuple<*T*> *t*;

introduces a *Sextuple*<*T*> using the default constructor of the elements.

Sextuple<*T*> *t*(*T* *x*, *T* *y*, *T* *z*, *T* *t*, *T* *u*, *T* *v*);

constructs a *Sextuple*<*T*> such that *e0* is constructed from *x*, *e1* from *y*, *e2* from *z*, *e3* from *t*, *e4* from *u* and *e5* from *v*.

CGAL::Triple<T1, T2, T3>

Definition

The Triple class is an extension of *std::pair*. *Triple<T1, T2, T3>* is a heterogeneous triple: it holds one object of type *T1*, one of type *T2*, and one of type *T3*. A *Triple<T1, T2, T3>* is much like a container, in that it "owns" its elements. It is not actually a model of container, though, because it does not support the standard methods (such as iterators) for accessing the elements of a container.

```
#include <CGAL/utility.h>
```

Requirements

T1, *T2* and *T3* must be *Assignable*. Additional operations have additional requirements.

Types

```
typedef T1    first_type;
typedef T2    second_type;
typedef T3    third_type;
```

Variables

<i>T1</i>	<i>first</i> ;	first element
<i>T2</i>	<i>second</i> ;	second element
<i>T3</i>	<i>third</i> ;	third element

Creation

<i>Triple<T1, T2, T3> t</i> ;	introduces a triple using the default constructor of the three elements.
-------------------------------------	--------------------------------------------------------------------------

<i>Triple<T1, T2, T3> t(T1 x, T2 y, T3 z);</i>	constructs a triple such that <i>first</i> is constructed from <i>x</i> , <i>second</i> is constructed from <i>y</i> , and <i>third</i> is constructed from <i>z</i> .
-------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<i>template <class U, class V, class W></i> <i>Triple<T1, T2, T3> t(U u, V v, W w);</i>	constructs a triple such that <i>first</i> is constructed from <i>u</i> , <i>second</i> is constructed from <i>v</i> , and <i>third</i> is constructed from <i>w</i> . <i>Requirement:</i> Proper conversion operators exist from <i>U</i> to <i>T1</i> , <i>V</i> to <i>T2</i> , and <i>W</i> to <i>T3</i> .
---------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

```
template <class T1, class T2, class T3>
```

bool $x < y$

The comparison operator. It uses lexicographic comparison: the return value is true if the first element of x is less than the first element of y , and false if the first element of y is less than the first element of x . If neither of these is the case, then it returns true if the second element of x is less than the second element of y , and false if the second element of y is less than the second element of x . If neither of these is the case, then it returns the result of comparing the third elements of x and y . This operator may only be used if $T1$, $T2$ and $T3$ define the comparison operator.

template <*class* $T1$, *class* $T2$, *class* $T3$ >
bool $x == y$

The equality operator. The return value is true if and only the first elements of x and y are equal, the second elements of x and y are equal, and the third elements of x and y are equal. This operator may only be used if $T1$, $T2$ and $T3$ define the equality operator.

template <*class* $T1$, *class* $T2$, *class* $T3$ >
Triple< $T1$, $T2$, $T3$ >

make_triple($T1$ x , $T2$ y , $T3$ z)

Equivalent to *Triple*< $T1$, $T2$, $T3$ >(x , y , z).

CGAL::Quadruple<T1, T2, T3, T4>

Definition

The Quadruple class is an extension of *std::pair*. *Quadruple<T1, T2, T3, T4>* is a heterogeneous quadruple: it holds one object of type *T1*, one of type *T2*, one of type *T3*, and one of type *T4*. A *Quadruple<T1, T2, T3, T4>* is much like a container, in that it “owns” its elements. It is not actually a model of container, though, because it does not support the standard methods (such as iterators) for accessing the elements of a container.

```
#include <CGAL/utility.h>
```

Requirements

T1, *T2*, *T3* and *T4* must be *Assignable*. Additional operations have additional requirements.

Types

```
typedef T1    first_type;
typedef T2    second_type;
typedef T3    third_type;
typedef T4    fourth_type;
```

Variables

<i>T1</i>	<i>first</i> ;	first element
<i>T2</i>	<i>second</i> ;	second element
<i>T3</i>	<i>third</i> ;	third element
<i>T4</i>	<i>fourth</i> ;	fourth element

Creation

Quadruple<T1, T2, T3, T4> *t*;

introduces a quadruple using the default constructor of the four elements.

Quadruple<T1, T2, T3, T4> *t*(*T1* *x*, *T2* *y*, *T3* *z*, *T4* *w*);

constructs a quadruple such that *first* is constructed from *x*, *second* is constructed from *y*, *third* is constructed from *z*, and *fourth* is constructed from *w*.

```
template <class U, class V, class W, class X>
Quadruple<T1, T2, T3, T4> t( U u, V v, W w, X x );
```

constructs a quadruple such that *first* is constructed from *u*, *second* is constructed from *v*, *third* is constructed from *w*, and *fourth* is constructed from *x*.

Requirement: Proper conversion operators exist from *U* to *T1*, *V* to *T2*, *W* to *T3*, and *X* to *T4*.

template <class *T1*, class *T2*, class *T3*, class *T4*>

bool *x* < *y*

The comparison operator. It uses lexicographic comparison: the return value is true if the first element of *x* is less than the first element of *y*, and false if the first element of *y* is less than the first element of *x*. If neither of these is the case, then it returns true if the second element of *x* is less than the second element of *y*, and false if the second element of *y* is less than the second element of *x*. If neither of these is the case, then it returns true if the third element of *x* is less than the third element of *y*, and false if the third element of *y* is less than the third element of *x*. If neither of these is the case, then it returns the result of comparing the fourth elements of *x* and *y*. This operator may only be used if *T1*, *T2*, *T3*, and *T4* define the comparison operator.

template <class *T1*, class *T2*, class *T3*, class *T4*>

bool *x* == *y*

The equality operator. The return value is true if and only the first elements of *x* and *y* are equal, the second elements of *x* and *y* are equal, the third elements of *x* and *y* are equal, and the fourth elements of *x* and *y* are equal. This operator may only be used if *T1*, *T2*, *T3*, and *T4* define the equality operator.

template <class *T1*, class *T2*, class *T3*, class *T4*>

Quadruple<*T1*, *T2*, *T3*, *T4*>

make_quadruple(*T1* *x*, *T2* *y*, *T3* *z*, *T4* *w*)

Equivalent to *Quadruple*<*T1*, *T2*, *T3*, *T4*>(*x*, *y*, *z*, *w*).

Chapter 46

Handles and Circulators

Olivier Devillers, Lutz Kettner, Michael Seel, and Mariette Yvinec

46.1 Handles

Most data structures in CGAL use the concept of *Handle* in their user interface to refer to the elements they store. This concept describes what is sometimes called a trivial iterator. A *Handle* is akin to a pointer to an object providing the dereference operator *operator**() and member access *operator->*() but no increment or decrement operators like iterators. A *Handle* is intended to be used whenever the referenced object is not part of a logical sequence.

Model for a handle A simple pointer T^* , an iterator or a circulator with value type T , are also handles.

46.2 Circulators

An introduction to the concept of circulators is given here. A couple of adaptors are presented that convert between iterators and circulators. Some useful functions for circulators follow. This chapter concludes with a discussion of the design decisions taken. For the full description of the circulator requirements, the provided base classes, the circulator tags, and the support for generic algorithms that work for iterators as well as for circulators please refer to the reference pages. Note that circulators are not part of STL, but of CGAL.

46.2.1 Introduction

The concept of iterators in STL is tailored for linear sequences [C++98, MS96]. In contrast, circular sequences occur naturally in many combinatorial and geometric structures. Examples are polyhedral surfaces and planar maps, where the edges emanating from a vertex or the edges around a facet form a circular sequence.

Since circular sequences do not allow for efficient iterators, we have introduced the new concept of *circulators*. They share most of the requirements of iterators, while the main difference is the lack of a past-the-end position in the sequence. Appropriate adaptors are provided between iterators and circulators to integrate circulators smoothly into the framework of STL. An example of a generic `contains` function illustrates the use of circulators. As usual for circular structures, a `do-while` loop is preferable, such that for the specific input, `c == d`, all elements in the sequence are reached.

```

template <class Circulator, class T>
bool contains( Circulator c, Circulator d, const T& value) {
    if (c != 0) {
        do {
            if (*c == value)
                return true;
        } while (++c != d);
    }
    return false;
}

```

Three circulator categories are defined: forward, bidirectional and random-access circulators. Given a circulator c , the operation $*c$ denotes the item the circulator refers to. The operation $++c$ advances the circulator by one item and $--c$ steps a bidirectional circulator one item backwards. For random-access circulators $c+n$ advances the circulator n steps. Two circulators can be compared for equality.

Circulators have a different notion of reachability and ranges than iterators. A circulator d is called *reachable* from a circulator c if c can be made equal to d with finitely many applications of the operator $++$. Due to the circularity of the sequence this is always true if both circulators refer to items of the same sequence. In particular, c is always reachable from c . Given two circulators c and d , the range $[c, d)$ denotes all circulators obtained by starting with c and advancing c until d is reached, but does not include d , for $d \neq c$. So far it is the same range definition as for iterators. The difference lies in the use of $[c, c)$ to denote all items in the circular sequence, whereas for an iterator i the range $[i, i)$ denotes the empty range. As long as $c \neq d$ the range $[c, d)$ behaves like an iterator range and could be used in STL algorithms. For circulators however, an additional test $c == \text{NULL}$ is required that returns true if and only if the circular sequence is empty. As for C++, we recommend the use of 0 instead of NULL.

Besides the conceptual cleanness, the main reason for inventing a new concept with a similar intent as iterators is efficiency. An iterator is supposed to be a light-weight object – merely a pointer and a single indirection to advance the iterator. Although iterators could be written for circular sequences, we do not know of an efficient solution. The missing past-the-end situation in circular sequences can be solved with an arbitrary sentinel in the cyclic order, but this would destroy the natural symmetry in the structure (which is in itself a bad idea) and additional bookkeeping in the items and checking in the iterator advance method reduces efficiency. Another solution may use more bookkeeping in the iterator, e.g. with a start item, a current item, and a kind of winding-number that is zero for the `begin()`-iterator and one for the past-the-end situation¹. We have introduced the concept of circulators that allows light-weight implementations and the CGAL support library provides adaptor classes that convert between iterators and circulators (with the corresponding penalty in efficiency), so as to integrate this new concept into the framework of STL.

A serious design problem is the slight change of the semantic for circulator ranges as compared to iterator ranges. Since this semantic is defined by the intuitive operators $++$ and $==$, which we would like to keep for circulators as well, circulator ranges can be used in STL algorithms. This is in itself a useful feature, if there would not be the definition of a full range $[c, c)$ that an STL algorithm will treat as an empty range. However, the likelihood of a mistake may be overestimated, since for a container C supporting circulators there is no `end()` member function, and an expression such as `std::sort(C.begin(), C.end())` will fail. It is easy to distinguish iterators and circulators at compile time, which allows for generic algorithms supporting both as arguments. It is also possible to protect algorithms against inappropriate arguments using the same technique, see the reference pages for circulators, specifically the *Assert_iterator* and *is_empty_range* functions.

Warning: Please note that the definition of a range is different from that of iterators. An interface of a data structure must declare whether it works with iterators, circulators, or both. STL algorithms always specify only iterators in their interfaces. A range $[c, d)$ of circulators used in an interface for iterators will work as expected

¹This is currently implemented as the adaptor class which provides a pair of iterators for a given circulator.

as long as $c \neq d$. A range $[c, c)$ will be interpreted as the empty range like for iterators, which is different than the full range that it should denote for circulators.

46.2.2 Forward Circulator

A class *Circulator* that satisfies the requirements of a forward circulator with the value type T , supports the following operations. See the reference pages for the full set of requirements. Note that the stated return values are not required, only a return value that is convertible to the stated type is required. As for C++, we recommend the use of 0 instead of `NULL`.

Types

<i>Circulator::value_type</i>	the value type T .
<i>Circulator::reference</i>	either reference or const reference to T .
<i>Circulator::pointer</i>	either pointer or const pointer to T .
<i>Circulator::size_type</i>	unsigned integral type that can hold the size of the sequence.
<i>Circulator::difference_type</i>	signed integral type that can hold the distance between two circulators.
<i>Circulator::iterator_category</i>	circulator category <i>Forward_circulator_tag</i> .

Creation

<i>Circulator</i> c ;	a circulator equal to <i>NULL</i> denoting an empty sequence.
<i>Circulator</i> $c(d)$;	a circulator equal to d .

Operations

<i>Circulator</i> &	$c = d$	Assignment.
<i>bool</i>	$c == \text{NULL}$	Test for emptiness.
<i>bool</i>	$c \neq \text{NULL}$	Test for non-emptiness, i.e. $!(c == \text{NULL})$.
<i>bool</i>	$c == d$	c is equal to d if they refer to the same item.
<i>bool</i>	$c \neq d$	Test for inequality, i.e. $!(c == d)$.
<i>reference</i>	$*c$	Returns the value of the circulator. If <i>Circulator</i> is mutable $*c = t$ is valid. <i>Precondition:</i> c is dereferenceable.
<i>pointer</i>	$c \rightarrow$	Returns a pointer to the value of the circulator. <i>Precondition:</i> c is dereferenceable.
<i>Circulator</i> &	$++c$	Prefix increment operation. <i>Precondition:</i> c is dereferenceable. <i>Postcondition:</i> c is dereferenceable.
<i>Circulator</i>	$c++$	Postfix increment operation. The result is the same as that of: <i>Circulator</i> $tmp = c$; $++c$; <i>return</i> tmp ; .

46.2.3 Bidirectional Circulator

A class *Circulator* that satisfies the requirements of a bidirectional circulator with the value type *T*, supports the following operations in addition to the operations supported by a forward circulator.

Types

Circulator:: *iterator_category*

circulator category *Bidirectional_circulator_tag*.

Operations

<i>Circulator</i> &	$--c$	Prefix decrement operation. <i>Precondition</i> : <i>c</i> is dereferenceable. <i>Postcondition</i> : <i>c</i> is dereferenceable.
<i>Circulator</i>	$c--$	Postfix decrement operation. The result is the same as that of <i>Circulator tmp = c; --c; return tmp; .</i>

46.2.4 Random Access Circulator

A class *Circulator* that satisfies the requirements of a random access Circulator for the value type *T*, supports the following operations in addition to the operations supported by a bidirectional Circulator. In contrast to random access iterators, no comparison operators are available for random access circulators.

Types

Circulator:: *iterator_category*

circulator category *Random_access_circulator_tag*.

Operations

<i>Circulator</i> &	$c += \textit{difference_type } n$	The result is the same as if the prefix increment operation was applied <i>n</i> times, but it is computed in constant time.
<i>Circulator</i>	$c + \textit{difference_type } n$	Same as above, but returns a new circulator.
<i>Circulator</i>	$\textit{difference_type } n + c$	Same as above.
<i>Circulator</i> &	$c -= \textit{difference_type } n$	The result is the same as if the prefix decrement operation was applied <i>n</i> times, but it is computed in constant time.

<i>Circulator</i>	$c - \text{difference_type } n$	Same as above, but returns a new circulator.
<i>reference</i>	$c[\text{difference_type } n]$	Returns $*(c + n)$.
<i>difference_type</i>	$c - \text{Circulator } d$	returns the difference between the two circulators. The value will be in the interval $[1 - s, s - 1]$ if s is the size of the total sequence. The difference for a fixed circulator c (or d) with all other circulators d (or c) is a consistent ordering of the elements in the data structure. There has to be a minimal circulator d_{\min} for which the difference $c - d_{\min}$ to all other circulators c is non-negative.
<i>Circulator</i>	$c.\text{min_circulator}()$	Returns the minimal circulator c_{\min} in constant time.

46.2.5 Adaptors Between Iterators and Circulators

Algorithms working on iterator ranges can not be applied to circulator ranges in full generality, only to subranges (see the warning in Section 46.2.1). The following adaptors convert circulators to iterators and vice versa (with the unavoidable space and time penalty) to reestablish this generality.

```
#include <CGAL/circulator.h>
```

Container_from_circulator container-like class with iterators built from a circulator
Circulator_from_iterator circulator over a range of two iterators
Circulator_from_container circulator for a container

The following example applies the generic `std::reverse()` algorithm from STL to a sequence given by a bidirectional circulator c . It uses the *Container_from_circulator* adaptor.

```
Circulator c; // c must be at least bidirectional.
CGAL::Container_from_circulator<Circulator> container(c);
std::reverse( container.begin(), container.end());
```

Another example defines a circulator c for a vector of *int*'s. However, since there are no elements in the vector, the circulator denotes an empty sequence. If there were elements in the vector, the circulator would implement a random access modulus the size of the sequence.

```
std::vector<int> v;
typedef CGAL::Circulator_from_iterator<
    std::vector<int>::iterator > Circulator;
Circulator c( v.begin(), v.end());
```

46.2.6 Functions on Circulators

A few functions deal with circulators and circulator ranges. The type C denotes a circulator. The type IC denotes either a circulator or an iterator. More on algorithms that work with circulators as well with iterators can be found in the reference pages.

```
#include <CGAL/circulator.h>
```

<i>circulator_size</i> (<i>C c</i>)	size of the sequence reachable by <i>c</i>
<i>circulator_distance</i> (<i>C c</i> , <i>C d</i>)	number of elements in the range $[c, d)$
<i>iterator_distance</i> (<i>IC ic1</i> , <i>IC ic2</i>)	number of elements in the range $[ic2, ic1)$
<i>is_empty_range</i> (<i>IC ic1</i> , <i>IC ic2</i>)	test the range $[ic2, ic1)$ for emptiness

Handles and Circulators

Reference Manual

Olivier Devillers, Lutz Kettner, Michael Seel, and Mariette Yvinec

Most data structures in CGAL use the concept of *Handle* in their user interface to refer to the elements they store. This concept describes what is sometimes called a trivial iterator. A *Handle* is akin to a pointer to an object providing the dereference operator *operator**() and member access *operator->*() but no increment or decrement operators like iterators. A *Handle* is intended to be used whenever the referenced object is not part of a logical sequence.

The concept of iterators in STL is tailored for linear sequences. In contrast, circular sequences occur naturally in many combinatorial and geometric structures. Examples are polyhedral surfaces and planar maps, where the edges emanating from a vertex or the edges around a facet form a circular sequence.

We provide several functions, classes and macros to assist in working with circulators: distance computation, adaptor classes converting between circulators and iterators, base classes to ease the implementation of circulators, and support for generic algorithms that work with circulators as well as with iterators.

46.3 Classified Reference Pages

Concepts

Handle	page 2729
Circulator	page 2715
<i>Forward_circulator</i>	
<i>Bidirectional_circulator</i>	
<i>Random_access_circulator</i>	

Classes

<i>CGAL::Container_from_circulator<C></i>	page 2726
<i>CGAL::Circulator_from_iterator<I></i>	page 2720
<i>CGAL::Circulator_from_container<C></i>	page 2718
<i>CGAL::Const_circulator_from_container<C></i>	
<i>CGAL::Circulator_tag</i>	page 2722
<i>CGAL::Iterator_tag</i>	

<i>CGAL::Forward_circulator_tag</i>	
<i>CGAL::Bidirectional_circulator_tag</i>	
<i>CGAL::Random_access_circulator_tag</i>	
<i>CGAL::Circulator_base</i>	page 2722
<i>CGAL::Forward_circulator_base</i>	
<i>CGAL::Bidirectional_circulator_base</i>	
<i>CGAL::Random_access_circulator_base</i>	
<i>CGAL::Circulator_traits<C></i>	page 2725

Functions

<i>size_type</i>	<i>CGAL::circulator_size(C c)</i>	page 2714
<i>difference_type</i>	<i>CGAL::circulator_distance(C c, C d)</i>	page 2713
<i>difference_type</i>	<i>CGAL::iterator_distance(IC ic1, IC ic2)</i>	page 2731
<i>bool</i>	<i>CGAL::is_empty_range(IC i, IC j)</i>	page 2730
<i>CGAL::Circulator_tag</i>	<i>CGAL::query_circulator_or_iterator(C c)</i>	page 2732
<i>CGAL::Iterator_tag</i>	<i>CGAL::query_circulator_or_iterator(I i)</i>	
<i>void</i>	<i>CGAL::Assert_circulator(C c)</i>	page 2712
<i>void</i>	<i>CGAL::Assert_iterator(I i)</i>	
<i>void</i>	<i>CGAL::Assert_input_category(I i)</i>	
<i>void</i>	<i>CGAL::Assert_output_category(I i)</i>	
<i>void</i>	<i>CGAL::Assert_forward_category(IC ic)</i>	
<i>void</i>	<i>CGAL::Assert_bidirectional_category(IC ic)</i>	
<i>void</i>	<i>CGAL::Assert_random_access_category(IC ic)</i>	
<i>void</i>	<i>CGAL::Assert_circulator_or_iterator(IC i)</i>	

Macros

<i>CGAL_For_all(i,j)</i>	page 2728
<i>CGAL_For_all_backwards(i,j)</i>	

46.4 Alphabetical List of Reference Pages

<i>Assert_circulator</i>	page 2712
<i>CGAL_For_all</i>	page 2728
<i>circulator_distance</i>	page 2713

<i>Circulator_from_container<C></i>	page 2718
<i>Circulator_from_iterator<I></i>	page 2720
<i>circulator_size</i>	page 2714
<i>Circulator_tag</i>	page 2722
<i>Circulator_traits<C></i>	page 2725
<i>Circulator</i>	page 2715
<i>Container_from_circulator<C></i>	page 2726
<i>Handle</i>	page 2729
<i>is_empty_range</i>	page 2730
<i>iterator_distance</i>	page 2731
<i>query_circulator_or_iterator</i>	page 2732

CGAL::Assert_circulator

Definition

Each of the following assertions, applicable to an iterator or a circulator or both, checks at compile time if its argument is of the kind stated in the assertions name, i.e. a circulator, an iterator, or a particular category of either an iterator or a circulator. Note that neither input nor output circulators exists.

```
#include <CGAL/circulator.h>
```

```
void          Assert_circulator( C c)
void          Assert_iterator( I i)
void          Assert_circulator_or_iterator( IC i)

void          Assert_input_category( I i)
void          Assert_output_category( I i)
void          Assert_forward_category( IC ic)
void          Assert_bidirectional_category( IC ic)
void          Assert_random_access_category( IC ic)
```

See Also

Circulator_tag, *Circulator_traits*, *query_circulator_or_iterator*, *Circulator*.

CGAL::circulator_distance

Definition

The distance of a circulator c to a circulator d is the number of elements in the range $[c, d)$. It is defined to be zero for a circulator on an empty sequence and it returns the size of the data structure when applied to a range of the form $[c, c)$.

```
#include <CGAL/circulator.h>
```

```
template <class C>  
C::difference_type      circulator_distance( C c, C d)
```

See Also

circulator_size, *iterator_distance*, *is_empty_range*, *Circulator*.

CGAL::circulator_size

Definition

The size of a circulator is the size of the data structure it refers to. It is zero for a circulator on an empty sequence. The size can be computed in linear time for forward and bidirectional circulators, and in constant time for random access circulators using the minimal circulator. The function *circulator_size(c)* returns the circulator size. It uses the *c.min_circulator()* function if *c* is a random access circulator.

```
#include <CGAL/circulator.h>
```

```
template <class C>  
C::size_type          circulator_size( C c)
```

See Also

circulator_distance, *iterator_distance*, *is_empty_range*, *Circulator*.

Circulator

Definition

Note: This specification is a revised version based on the C++ Standard [C++98], which is available now. In particular, iterator traits are now assumed and required.

Iterators in the STL were tailored for linear sequences. The specialization for circular data structures leads to slightly different requirements which we will summarize in the *circulators* concept. The main difference is that a circular data structure has no natural past-the-end value. As a consequence, a container supporting circulators will not have an `end()`-member function. The semantic of a circulator range differs from the semantic of an iterator range. For a circulator c the range $[c, c)$ denotes the sequence of all elements in the data structure. For iterators, this range defines the empty sequence. A separate test for an empty sequence has been added to the circulator requirements: A comparison $c == \text{NULL}$ for a circulator c is true for an empty sequence. As for C++, we recommend the use of 0 instead of `NULL`.

Similar to STL iterators, we distinguish between forward, bidirectional, and random access circulators². Most requirements for circulators are equal to those for iterators. We present the changes, please refer to [MS96, chapter 18] or [C++98] for the iterator requirements.

Past-the-end value: There is no past-the-end value for circulators.

Singular values: There are no singular values for circulators³

Empty sequence: The comparison $c == \text{NULL}$ (or $c == 0$) for a circulator c is `true` if c denotes an empty sequence, and `false` otherwise.

Dereferenceable values: A circulator that does not denote an empty sequence is dereferenceable.

Reachability: Each dereferenceable circulator can reach itself with a finite and non-empty sequence of applications of `operator++`.

Ranges: For any circulator c the range $[c, c)$ is a valid range. If the circulator refers to an empty sequence, the range $[c, c)$ denotes the empty range. Otherwise the circulator is dereferenceable and the range $[c, c)$ denotes the sequence of all elements in the data structure. *Remark:* When a circulator is used in a place of an iterator, as, for example, with an STL algorithm, it will work as expected with the only exception that, in STL algorithms, the range $[c, c)$ denotes always the empty range. This is not a requirement, but a consequence of the requirements stated here and the fact that the STL requirements for iterator ranges are based on the `operator++` and the `operator==`, which we use for circulators as well. In principle, we face here the difference between a `while` loop and a `do-while` loop.

Types: For a circulator of type C the following local types are required:

<code>C::value_type</code>	value type the circulator refers to.
<code>C::reference</code>	reference type used for the return type of <code>C::operator*()</code> .
<code>C::pointer</code>	pointer type used for the return type of <code>C::operator->()</code> .
<code>C::size_type</code>	unsigned integral type that can hold the size of a sequence
<code>C::difference_type</code>	signed integral type that can hold the distance between two circulators.
<code>C::iterator_category</code>	circulator category.

²Input circulators are a contradiction, since any circulator is supposed to return once to itself. Output circulators are not supported since they would be indistinguishable from output iterators.

³Since circulators must be implemented as classes anyway, there is no need to allow singular values for them. An un-initialized circulator does not have a singular value, but is supposed to refer to an empty sequence.

Forward Circulators

In the following, we assume that a and b are circulators of type C , r is of type $C\&$ (is assignable), and T denotes the value type of C . Let D be the distance type of C . As for $C++$, we recommend the use of 0 instead of $NULL$.

<code>C()</code>	a circulator equal to <i>NULL</i> denoting an empty sequence.
<code>a == NULL</code>	Returns <i>true</i> if a denotes an empty sequence, <i>false</i> otherwise. For simplicity, <code>NULL == a</code> is not required. The behavior for comparisons with pointer-like values different than <i>NULL</i> is undefined. A runtime assertion is recommended.
<code>a != NULL</code>	Returns <code>!(a == NULL)</code> .
<code>++r</code>	Like for forward iterators, but a dereferenceable circulator r will always be dereferenceable after <code>++r</code> (no past-the-end value). <i>Precondition:</i> r does not denote an empty sequence.
<code>r++</code>	Same as for <code>++r</code> .
<code>C::iterator_category</code>	circulator category <code>CBP_Forward_circulator_tag</code> .

Bidirectional Circulators

The same requirements as for the forward circulators hold for bidirectional iterators with the following change of the iterator category:

`C::iterator_category` circulator category `CBP_Bidirectional_circulator_tag`.

Random Access Circulators

The same requirements as for the bidirectional circulators hold for random access iterators with the following changes and extensions.

The idea of random access extends naturally to circulators using equivalence classes modulus the length of the sequence. With this in mind, the additional requirements for random access iterators hold also for random access circulators. The only exception is that the random access iterator is required to provide a total order on the sequence, which a circulator cannot provide⁴.

The difference of two circulators is not unique as for iterators. A reasonable requirement demands that the result is in a certain range $[1 - \text{size}, \text{size} - 1]$, where *size* is the size of the sequence, and that whenever a circulator a is fixed that the differences with all other circulators of the sequence form a consistent ordering.

For the adaptor to iterators a minimal circulator d_{\min} is required for which the difference $c - d_{\min}$ to all other circulators c is non negative.

<code>b - a</code>	limited range and consistent ordering as explained above.
<code>a.min_circulator()</code>	returns the minimal circulator from the range $[a, a)$.
<code>C::iterator_category</code>	circulator category <code>CBP_Random_access_circulator_tag</code> .

⁴One might define an order by splitting the circle at a fixed point, e.g. the start circulator provided from the data structure. This is what the adaptor to iterators will do. Nonetheless, we do not require this for circulators.

Const Circulators

As with iterators, we distinguish between circulators and const circulators. The expression `*a = t` with `t` of type `T` is valid for mutable circulators. It is invalid for const circulators.

Circulators in Container Classes

For a container `x` of type `X` that supports circulators `c` the following naming convention is recommended:

<code>X::Circulator</code>	the type of the mutable circulator.
<code>X::Const_circulator</code>	the type of the const circulator.
<code>c = x.begin()</code>	the start circulator of the sequence. It is of type <code>X::Circulator</code> for a mutable container or <code>X::Const_circulator</code> for a const container. There must not be an <code>end()</code> member function.

If a container will support iterators and circulators, the member function `circulator_begin()` is proposed. However, the support of iterators and circulators simultaneously is not recommended, since it would lead to fat interfaces. The natural choice should be supported, the other concept will be available through adaptors.

Example

A generic `contains` function accepts a range of circulators and a value. It returns `true` if the value is contained in the sequence of items denoted by the range of circulators. As usual for circular structures, a `do-while` loop is preferable, such that for the specific input, `c == d`, all elements in the sequence are reached. Note that the example simplifies if the sequence is known to be non-empty, which is for example the common case in polyhedral surfaces where vertices and facets have at least one incident edge.

```
template <class Circulator, class T>
bool contains( Circulator c, Circulator d, const T& value) {
    if (c != 0) {
        do {
            if (*c == value)
                return true;
        } while (++c != d);
    }
    return false;
}
```

CGAL::Circulator_from_container<C>

Definition

The adaptor *Circulator_from_container*<*C*> provides a circulator for an STL container *C* of equal category as the iterator provided by the container. The iterator must be at least of the forward iterator category. The corresponding non-mutable circulator is called *Const_circulator_from_container*<*C*>.

The container type *C* is supposed to conform to the STL requirements for container (i.e. to have a *begin()* and an *end()* iterator as well as the local types *reference*, *const_reference*, *value_type*, *size_type*, and *difference_type*).

```
#include <CGAL/circulator.h>
```

Types

All types required for circulators are provided.

Creation

Circulator_from_container<C> c; a circulator *c* on an empty sequence.

```
Circulator_from_container<C> c( C* container);
```

a circulator c initialized to refer to the first element in *container*, i.e. *container.begin()*. The circulator c refers to an empty sequence if the *container* is empty.

```
Circulator_from_container<C> c( C* container, C::iterator i);
```

a circulator c initialized to refer to the element $*i$ in *container*.

Precondition: $*i$ is dereferenceable and refers to *container*.

Operations

The adaptor conforms to the requirements of the corresponding circulator category. An additional member function *current_iterator()* returns the current iterator pointing to the same position as the circulator does.

See Also

Container_from_circulator, Circulator_from_iterator, Circulator.

Example

The following program composes two adaptors – from a container to a circulator and back to an iterator. It applies an STL sort algorithm on a STL vector with three elements. The resulting vector will be [2 5 9] as it is checked by the assertions. The program is part of the CGAL distribution.

```

// file: examples/Circulator/circulator_prog2.C

#include <CGAL/basic.h>
#include <cassert>
#include <vector>
#include <algorithm>
#include <CGAL/circulator.h>

typedef CGAL::Circulator_from_container< std::vector<int> > Circulator;
typedef CGAL::Container_from_circulator<Circulator> Container;
typedef Container::iterator Iterator;

int main() {
    std::vector<int> v;
    v.push_back(5);
    v.push_back(2);
    v.push_back(9);
    Circulator c( &v);
    Container container( c);
    std::sort( container.begin(), container.end());
    Iterator i = container.begin();
    assert( *i == 2);
    i++;    assert( *i == 5);
    i++;    assert( *i == 9);
    i++;    assert( i == container.end());
    return 0;
}

```

CGAL::Circulator_from_iterator<I>

Definition

The adaptor *Circulator_from_iterator*<*I*> converts two iterators of type *I*, a begin and a past-the-end value, to a circulator of equal category. The iterator must be at least of the forward iterator category. The circulator will be mutable or non-mutable according to the iterator. Iterators provide no *size_type*. This adapter assumes *std::size_t* instead.

```
#include <CGAL/circulator.h>
```

Types

```
typedef I          iterator;
```

In addition all types required for circulators are provided.

Creation

Circulator_from_iterator<*I*> *c*; a circulator *c* on an empty sequence.

Circulator_from_iterator<*I*> *c*(*I* *begin*, *I* *end*, *I* *cur* = *begin*);

a circulator *c* initialized to refer to the element **cur* in a range [*begin*, *end*). The circulator *c* refers to a empty sequence if *begin*==*end*.

Circulator_from_iterator<*I*> *c*(*Circulator_from_iterator*<*I*,*T*,*Size*,*Dist*> *d*, *I* *cur*);

a copy of circulator *d* referring to the element **cur*. The circulator *c* refers to a empty sequence if *d* does so.

Operations

The adaptor conforms to the requirements of the respective circulator category. An additional member function *current_iterator*() returns the current iterator pointing to the same position as the circulator does.

See Also

Container_from_circulator, *Circulator_from_container*, *Circulator*.

Example

The following program composes two adaptors – from an iterator to a circulator and back to an iterator. It applies an STL sort algorithm on a STL vector containing three elements. The resulting vector will be [2 5 9] as it is checked by the assertions. The program is part of the CGAL distribution.


```

// file: examples/Circulator/circulator_progl.C

#include <CGAL/basic.h>
#include <cassert>
#include <vector>
#include <algorithm>
#include <CGAL/circulator.h>

typedef std::vector<int>::iterator      I;
typedef CGAL::Circulator_from_iterator<I>      Circulator;
typedef CGAL::Container_from_circulator<Circulator> Container;
typedef Container::iterator      Iterator;

int main() {
    std::vector<int> v;
    v.push_back(5);
    v.push_back(2);
    v.push_back(9);
    Circulator c( v.begin(), v.end());
    Container container( c);
    std::sort( container.begin(), container.end());
    Iterator i = container.begin();
    assert( *i == 2);
    i++;    assert( *i == 5);
    i++;    assert( *i == 9);
    i++;    assert( i == container.end());
    return 0;
}

```

Another example usage for this adaptor is a random access circulator over the built-in C arrays. Given an array of type T^* with a begin pointer b and a past-the-end pointer e the adaptor *Circulator_from_iterator* $\langle T^* \rangle c(b,e)$ is a random access circulator c over this array.

CGAL::Circulator_tag

Definition

Iterators and circulators as well as different categories of circulators can be distinguished with the use of discriminating functions and the following circulator tags. A couple of base classes simplify the task of writing own circulators. They declare the appropriate tags and the local types needed for circulators. To use the tags or base classes only it is sufficient to include:

```
#include <CGAL/circulator_bases.h>
#include <CGAL/circulator.h>
```

Compile Time Tags

<code>struct Circulator_tag {};</code>	any circulator.
<code>struct Iterator_tag {};</code>	any iterator.
 <code>struct Forward_circulator_tag {};</code>	 derived from <code>forward_iterator_tag</code> .
<code>struct Bidirectional_circulator_tag {};</code>	derived from <code>bidirectional_iterator_tag</code> .
<code>struct Random_access_circulator_tag {};</code>	derived from <code>random_access_iterator_tag</code> .

Base Classes

```
template < class Category, class T, class Dist = std::ptrdiff_t, class Size = std::size_t, class Ptr = T*, class Ref
= T& >
struct Circulator_base {};

struct Forward_circulator_base {};
struct Bidirectional_circulator_base {};
struct Random_access_circulator_base {};
```

See Also

`query_circulator_or_iterator`, `Circulator_traits`, `Assert_circulator`,
`CGAL_For_all`, `is_empty_range`, `Circulator`.

Example

The above declarations can be used to distinguish between iterators and circulators and between different circulator categories. The assertions can be used to protect a templated algorithm against instantiations that do not fulfill the requirements. The following example program illustrates both.

```
// file: examples/Circulator/circulator_prog3.C

#include <CGAL/basic.h>
#include <cassert>
```

```

#include <list>
#include <CGAL/circulator.h>

template <class C> inline int foo( C c, std::forward_iterator_tag) {
    CGAL::Assert_circulator( c);
    CGAL::Assert_forward_category( c);
    return 1;
}

template <class C> inline int foo( C c, std::random_access_iterator_tag) {
    CGAL::Assert_circulator( c);
    CGAL::Assert_random_access_category( c);
    return 2;
}

template <class I> inline int foo( I i, CGAL::Iterator_tag) {
    CGAL::Assert_iterator( i);
    return 3;
}

template <class C> inline int foo( C c, CGAL::Circulator_tag) {
    CGAL::Assert_circulator( c);
    typedef std::iterator_traits<C> Traits;
    typedef typename Traits::iterator_category iterator_category;
    return foo( c, iterator_category());
}

template <class IC> inline int foo( IC ic) {
    typedef CGAL::Circulator_traits<IC> Traits;
    typedef typename Traits::category category;
    return foo( ic, category());
}

int main() {
    typedef CGAL::Forward_circulator_base<int> F;
    typedef CGAL::Random_access_circulator_base<int> R;
    F f = F();
    R r = R();
    std::list<int> l;
    assert( foo( f) == 1);
    assert( foo( r) == 2);
    assert( foo( l.begin()) == 3);
    return 0;
}

```

Implementation

Since not all current compilers can eliminate the space needed for the compile time tags even when deriving from them, we implement a variant for each base class that contains a protected *void** data member called *_ptr*. Here, the allocated space in the derived classes can be reused.

<i>template <class T, class Dist, class Size></i>	
<i>class Forward_circulator_ptrbase {};</i>	forward circulator.
<i>template <class T, class Dist, class Size></i>	
<i>class Bidirectional_circulator_ptrbase {};</i>	bidirectional circulator.

```
template <class T, class Dist, class Size>  
class Random_access_circulator_ptrbase {};
```

random access circulator.

CGAL::Circulator_traits<C>

Definition

The circulator traits class distinguishes between circulators and iterators. It defines a local type *category* that is identical to the type *Circulator_tag* if the iterator category of the argument *C* is a circulator category. Otherwise it is identical to the type *Iterator_tag*.

The local type *iterator_category* gives the corresponding iterator category for circulators, i.e. one of *forward_iterator_tag*, *bidirectional_iterator_tag*, or *random_access_iterator_tag*.

The local type *circulator_category* gives the corresponding circulator category for iterators, i.e. one of *Forward_circulator_tag*, *Bidirectional_circulator_tag*, or *Random_access_circulator_tag*.

```
#include <CGAL/circulator.h>
```

Types

<i>Circulator_traits<C>::category</i>	either <i>Iterator_tag</i> or <i>Circulator_tag</i> .
<i>Circulator_traits<C>::iterator_category</i>	corresponding iterator category for circulators.
<i>Circulator_traits<C>::circulator_category</i>	corresponding circulator category for iterator

Example

A generic function *bar* that distinguishes between a call with a circulator range and a call with an iterator range:

```
template <class I>
void bar( I i, I j, CGAL::Iterator_tag) {
    CGAL::Assert_iterator(i);
    // This function is called for iterator ranges [i,j).
}
template <class C>
void bar( C c, C d, CGAL::Circulator_tag) {
    CGAL::Assert_circulator(c);
    // This function is called for circulator ranges [c,d).
}
template <class IC>
void bar( IC i, IC j) { // calls the correct function
    return bar( i, j, typename CGAL::Circulator_traits<IC>::category());
}
```

CGAL::Container_from_circulator<C>

Definition

The adaptor *Container_from_circulator<C>* is a class that converts any circulator type *C* to a kind of container class, i.e. a class that provides an *iterator* and a *const_iterator* type and two member functions – *begin()* and *end()* – that return the appropriate iterators. By analogy to STL container classes these member functions return a const iterator in the case that the container itself is constant and a mutable iterator otherwise.

```
#include <CGAL/circulator.h>
```

Types

```
typedef C          Circulator;
Container_from_circulator<C>:: iterator
Container_from_circulator<C>:: const_iterator
Container_from_circulator<C>:: value_type
Container_from_circulator<C>:: reference
Container_from_circulator<C>:: const_reference
Container_from_circulator<C>:: pointer
Container_from_circulator<C>:: const_pointer
Container_from_circulator<C>:: size_type
Container_from_circulator<C>:: difference_type
```

Creation

```
Container_from_circulator<C> container;
```

any iterator of *container* will have a singular value.

```
Container_from_circulator<C> container( C c);
```

any iterator of *container* will have a singular value if the circulator *c* corresponds to an empty sequence.

Operations

<i>iterator</i>	<i>container.begin()</i>	the start iterator.
<i>const_iterator</i>	<i>container.begin() const</i>	the start const iterator.
<i>iterator</i>	<i>container.end()</i>	the past-the-end iterator.
<i>const_iterator</i>	<i>container.end() const</i>	the past-the-end const iterator.

The *iterator* and *const_iterator* types are of the appropriate iterator category. In addition to the operations required for their category, they have a member function *current_circulator()* that returns a circulator pointing to the same position as the iterator does.

See Also

Circulator_from_iterator, *Circulator_from_container*, *Circulator*.

Example

The generic `reverse()` algorithm from the STL can be used with an adaptor if at least a bidirectional circulator `c` is given.

```
Circulator c; // c is assumed to be a bidirectional circulator.
CGAL::Container_from_circulator<Circulator> container(c);
reverse( container.begin(), container.end());
```

Implementation

The iterator adaptor keeps track of the number of rounds a circulator has done around the ring-like data structure (a kind of winding number). It is used to distinguish between the start position and the end position which will be denoted by the same circulator internally. This winding number is zero for the *begin()*-iterator and one for the *end()*-iterator. It is incremented whenever the internal circulator passes the *begin()* position. Two iterators are equal if their internally used circulators and winding numbers are equal. This is more general than necessary since an iterator equal to *end()*-iterator is not supposed to be incremented any more, which is here still possible in a defined manner.

The implementation is different for random access iterators. The random access iterator has to be able to compute the size of the data structure in constant time. This is for example needed if the difference of the past-the-end iterator and the begin iterator is taken, which is exactly the size of the data structure. Therefore, if the circulator is of the random-access category, the adapter chooses the minimal circulator for the internal anchor position. The minimal circulator is part of the random access circulator requirements, see Page [2716](#). For the random access iterator the adaptor implements a total ordering relation that is currently not required for random access circulators.

CGAL_For_all

Definition

In order to write algorithms that work with iterator ranges as well as with circulator ranges we have to consider the difference of representing an empty range. For iterators this is the range $[i, i)$, while for circulators it would be $c == \text{NULL}$, the empty sequence test. The function *is_empty_range* provides the necessary generic test which accepts an iterator range or a circulator range and says whether the range is empty or not.

```
#include <CGAL/circulator.h>
```

A macro *CGAL_For_all*(*i*, *j*) simplifies the writing of such simple loops as the one in the example of the function *is_empty_range*. *i* and *j* can be either iterators or circulators. The macro loops through the range $[i, j)$. It increments *i* until it reaches *j*. The implementation looks like:

```
CGAL_For_all(i,j)  ≡  for ( bool _circ_loop_flag = ! ::CGAL::is_empty_range(i, j);
                          _circ_loop_flag;
                          _circ_loop_flag = ((++i) != (j))
                          )
```

Note that the macro behaves like a *for*-loop. It can be used with a single statement or with a statement block. For bidirectional iterators or circulators, a backwards loop macro *CGAL_For_all_backwards*(*i*, *j*) exists that decrements *j* until it reaches *i*.

See Also

iterator_distance, *is_empty_range*, *Circulator_tag*, *Circulator_traits*,
Assert_circulator_or_iterator, *Circulator*.

Handle

Definition

Most data structures in CGAL use the concept of Handle in their user interface to refer to the elements they store. This concept describes what is sometimes called a trivial iterator. A Handle is akin to a pointer to an object providing the dereference operator *operator**() and member access *operator->*() but no increment or decrement operators like iterators. A Handle is intended to be used whenever the referenced object is not part of a logical sequence.

Like iterators, the handle can be passed as template argument to *std::iterators_traits* in order to extract its *value_type*, the type of the element pointed to. The *iterator_category* is *void*.

Refines

DefaultConstructible, CopyConstructible, Assignable, EqualityComparable

The default constructed object must be unique as far as the equality operator is concerned (this serves the same purpose as NULL for pointers). (Note that this is not a generally supported feature of iterators of standard containers.)

Creation

Dereference

<i>value_type</i> &	<i>*h</i>	returns the object pointed to.
<i>value_type</i> *	<i>h -></i>	returns a pointer to the object pointed to.

Has Models

pointers
const pointers
iterators
circulators

CGAL::is_empty_range

Definition

In order to write algorithms that work with iterator ranges as well as with circulator ranges we have to consider the difference of representing an empty range. For iterators this is the range $[i, i)$, while for circulators it would be $c == \text{NULL}$, the empty sequence test. The function *is_empty_range* provides the necessary generic test which accepts an iterator range or a circulator range and says whether the range is empty or not.

```
#include <CGAL/circulator.h>
```

```
template< class IC>
```

```
bool    is_empty_range( IC i, IC j)    is true if the range  $[i, j)$  is empty, false otherwise.  
                                             Precondition: IC is either a circulator or an iterator type. The range  $[i, j)$  is valid.
```

Example

The following function *process_all* accepts a range $[i, j)$ of an iterator or circulator *IC* and processes each element in this range:

```
template <class IC>
void process_all( IC i, IC j) {
    if (! CGAL::is_empty_range( i, j)) {
        do {
            process(*i);
        } while (++i != j);
    }
}
```

See Also

iterator_distance, *CGAL_For_all*, *Circulator_tag*, *Circulator_traits*, *Assert_circulator_or_iterator*, *Circulator*.

CGAL::iterator_distance

Definition

The following function returns the distance between either two iterators or two circulators. The return type is `ptrdiff_t` for compilers not supporting iterator traits yet.

```
#include <CGAL/circulator.h>
```

```
template <class IC>  
iterator_traits<IC>::difference_type    iterator_distance( IC ic1, IC ic2)
```

See Also

circulator_size, *circulator_distance*, *is_empty_range*, *Circulator_tag*,
Assert_circulator_or_iterator, *CGAL_For_all*, *Circulator*.

CGAL::query_circulator_or_iterator

Definition

The following function distinguishes between circulators and iterators. It is based on iterator traits [C++98, Mye95] and *Circulator_traits*.

```
#include <CGAL/circulator.h>
```

```
template <class I>
Iterator_tag    query_circulator_or_iterator( I i)
```

if the iterator category of *I* belongs to an iterator.

```
template <class C>
Circulator_tag  query_circulator_or_iterator( C c)
```

if the iterator category of *C* belongs to a circulator.

See Also

Circulator_tag, *Circulator_traits*, *Assert_circulator*, *Circulator*.

Chapter 47

Geometric Object Generators

Susan Hert, Michael Hoffmann, Lutz Kettner, and Sven Schönherr

A variety of generators for geometric objects are provided in CGAL. They are useful as synthetic test data sets, e.g. for testing algorithms on degenerate object sets and for performance analysis.

Two kinds of point generators are provided: first, random point generators and second deterministic point generators. Most random point generators and a few deterministic point generators are provided as input iterators. The input iterators model an infinite sequence of points. The function `CGAL::copy_n()` can be used to copy a finite sequence; see Section 45.4. The iterator adaptor *Counting_iterator* can be used to create finite iterator ranges; see Section 45.4. Other generators are provided as functions that write to output iterators. Further functions add degeneracies or random perturbations.

In 2D, we provide input iterators to generate random points in a disc (*Random_points_in_disc_2*), in a square (*Random_points_in_square_2*), on a circle (*Random_points_on_circle_2*), on a segment (*Random_points_on_segment*), and on a square (*Random_points_on_square_2*). For generating grid points we provide three functions, *points_on_segment_2*, *points_on_square_grid_2* that write to output iterators and an input iterator *Points_on_segment_2*.

For 3D points, input iterators are provided for random points uniformly distributed in a sphere (*Random_points_in_sphere_3*) or cube (*Random_points_in_cube_3*) or on the boundary of a sphere (*Random_points_on_sphere_3*). For generating 3D grid points, we provide the function *points_on_cube_grid_3* that writes to an output iterator.

We also provide two functions for generating more complex geometric objects. The function *random_convex_set_2* computes a random convex planar point set of a given size where the points are drawn from a specific domain and *random_polygon_2* generates a random simple polygon from points drawn from a specific domain.

Random Perturbations Degenerate input sets like grid points can be randomly perturbed by a small amount to produce *quasi*-degenerate test sets. This challenges numerical stability of algorithms using inexact arithmetic and exact predicates to compute the sign of expressions slightly off from zero. For this the function *perturb_points_2* is provided.

Adding Degeneracies For a given point set certain kinds of degeneracies can be produced by adding new points. The *random_selection()* function is useful for generating multiple copies of identical points. The function *random_collinear_points_2()* adds collinearities to a point set.

Support Functions and Classes for Generators The function *random_selection* chooses n items at random from a random access iterator range which is useful to produce degenerate input data sets with multiple entries of identical items.

47.1 Example Generating Degenerate Point Sets

We want to generate a test set of 1000 points, where 60% are chosen randomly in a small disc, 20% are from a larger grid, 10% are duplicates points, and 10% collinear points. A random shuffle removes the construction order from the test set. See Figure 47.1 for the example output.

```
// file: examples/Generator/generators_example1.C

#include <CGAL/Simple_cartesian.h>
#include <cassert>
#include <vector>
#include <algorithm>
#include <CGAL/point_generators_2.h>
#include <CGAL/copy_n.h>
#include <CGAL/random_selection.h>

using namespace CGAL;

typedef Simple_cartesian<double>          R;
typedef R::Point_2                        Point;
typedef Creator_uniform_2<double,Point>   Creator;
typedef std::vector<Point>                Vector;

int main() {
    // Create test point set. Prepare a vector for 1000 points.
    Vector points;
    points.reserve(1000);

    // Create 600 points within a disc of radius 150.
    Random_points_in_disc_2<Point,Creator> g( 150.0);
    CGAL::copy_n( g, 600, std::back_inserter(points));

    // Create 200 points from a 15 x 15 grid.
    points_on_square_grid_2( 250.0, 200, std::back_inserter(points),Creator());

    // Select 100 points randomly and append them at the end of
    // the current vector of points.
    random_selection( points.begin(), points.end(), 100,
        std::back_inserter(points));

    // Create 100 points that are collinear to two randomly chosen
    // points and append them to the current vector of points.
    random_collinear_points_2( points.begin(), points.end(), 100,
        std::back_inserter( points));

    // Check that we have really created 1000 points.
    assert( points.size() == 1000);
}
```

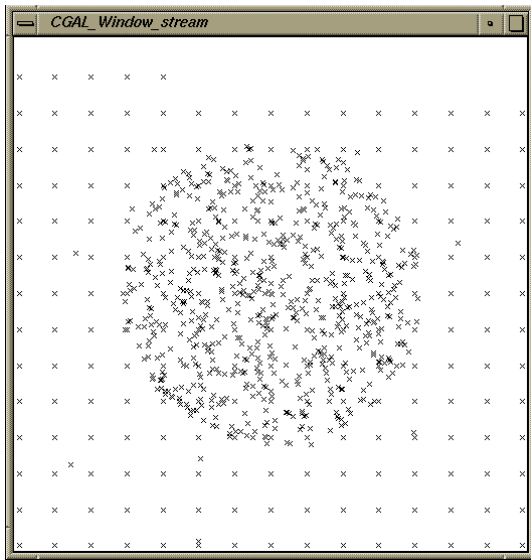


Figure 47.1: Output of example program for point generators.

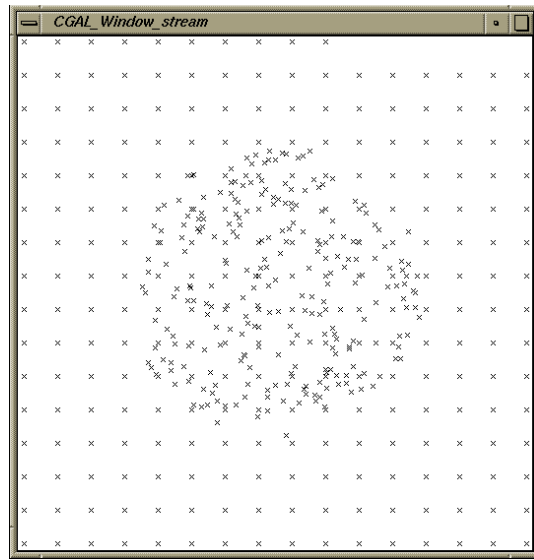


Figure 47.2: Output of example program for point generators working on integer points.

```
// Use a random permutation to hide the creation history
// of the point set.
std::random_shuffle( points.begin(), points.end(), default_random);

// Check range of values.
for ( Vector::iterator i = points.begin(); i != points.end(); i++){
assert( i->x() <= 251);
assert( i->x() >= -251);
assert( i->y() <= 251);
assert( i->y() >= -251);
}
return 0;
}
```

47.2 Example Generating Grid Points

The second example demonstrates the point generators with integer points. Arithmetic with *doubles* is sufficient to produce regular integer grids. See Figure 47.2 for the example output.

```
// file: examples/Generator/generators_example2.C

#include <CGAL/Simple_cartesian.h>
#include <cassert>
#include <vector>
#include <algorithm>
#include <CGAL/point_generators_2.h>
#include <CGAL/copy_n.h>
```

```

using namespace CGAL;

typedef Simple_cartesian<int>          R;
typedef R::Point_2                    Point;
typedef Creator_uniform_2<int,Point>  Creator;

int main() {
    // Create test point set. Prepare a vector for 400 points.
    std::vector<Point> points;
    points.reserve(400);

    // Create 250 points from a 16 x 16 grid. Note that the double
    // arithmetic _is_ sufficient to produce exact integer grid points.
    // The distance between neighbors is 34 pixel = 510 / 15.
    points_on_square_grid_2( 255.0, 250, std::back_inserter(points),Creator());

    // Lower, left corner.
    assert( points[0].x() == -255);
    assert( points[0].y() == -255);

    // Upper, right corner. Note that 6 points are missing to fill the grid.
    assert( points[249].x() == 255 - 6 * 34);
    assert( points[249].y() == 255);

    // Create 250 points within a disc of radius 150.
    Random_points_in_disc_2<Point,Creator> g( 150.0);
    CGAL::copy_n( g, 250, std::back_inserter(points));

    // Check that we have really created 500 points.
    assert( points.size() == 500);
    return 0;
}

```

47.3 Examples Generating Segments

The following two examples illustrate the use of the generic functions from Section 45.4 like *Join_input_iterator_2* to generate composed objects from other generators – here two-dimensional segments from two point generators.

We want to generate a test set of 200 segments, where one endpoint is chosen randomly from a horizontal segment of length 200, and the other endpoint is chosen randomly from a circle of radius 250. See Figure 47.3 for the example output.

```

// file: examples/Generator/Segment_generator_example1.C

#include <CGAL/Simple_cartesian.h>
#include <cassert>
#include <vector>
#include <algorithm>
#include <CGAL/Point_2.h>
#include <CGAL/Segment_2.h>

```

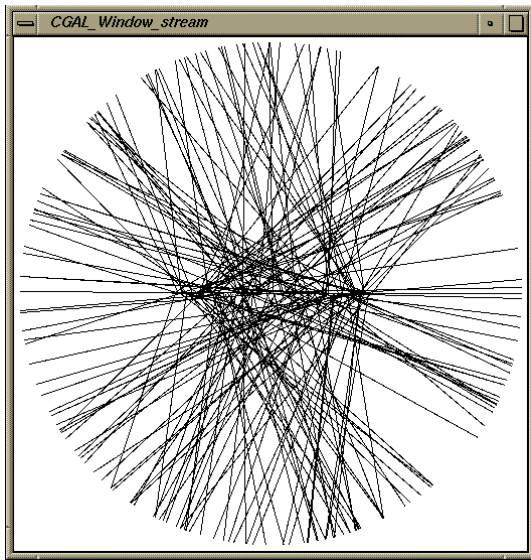



Figure 47.3: Output of the first example program for the generic generator.

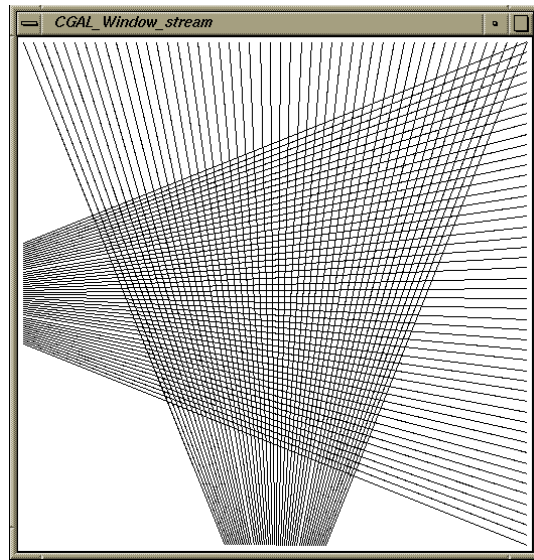


Figure 47.4: Output of the second example program for the generic generator without using intermediate storage.

```
#include <CGAL/point_generators_2.h>
#include <CGAL/function_objects.h>
#include <CGAL/Join_input_iterator.h>
#include <CGAL/copy_n.h>

using namespace CGAL;

typedef Simple_cartesian<double>          R;
typedef R::Point_2                        Point;
typedef Creator_uniform_2<double,Point>   Pt_creator;
typedef R::Segment_2                     Segment;
typedef std::vector<Segment>              Vector;

int main() {
    // Create test segment set. Prepare a vector for 200 segments.
    Vector segs;
    segs.reserve(200);

    // Prepare point generator for the horizontal segment, length 200.
    typedef Random_points_on_segment_2<Point,Pt_creator> P1;
    P1 p1( Point(-100,0), Point(100,0));

    // Prepare point generator for random points on circle, radius 250.
    typedef Random_points_on_circle_2<Point,Pt_creator> P2;
    P2 p2( 250);

    // Create 200 segments.
    typedef Creator_uniform_2< Point, Segment> Seg_creator;
    typedef Join_input_iterator_2< P1, P2, Seg_creator> Seg_iterator;
    Seg_iterator g( p1, p2);
```

```

CGAL::copy_n( g, 200, std::back_inserter(segs));

assert( segs.size() == 200);
for ( Vector::iterator i = segs.begin(); i != segs.end(); i++){
    assert( i->source().x() <= 100);
    assert( i->source().x() >= -100);
    assert( i->source().y() == 0);
    assert( i->target().x() * i->target().x() +
            i->target().y() * i->target().y() <= 251*251);
    assert( i->target().x() * i->target().x() +
            i->target().y() * i->target().y() >= 249*249);
}
return 0;
}

```

The second example generates a regular structure of 100 segments; see Figure 47.4 for the example output. It uses the *Points_on_segment_2* iterator, *Join_input_iterator_2* and *Counting_iterator* to avoid any intermediate storage of the generated objects until they are used, which in this example means copied to a window stream.

```

// file: examples/Generator/Segment_generator_example2.C

// CGAL example program for the generic segment generator
// using precomputed point locations.

#include <CGAL/Simple_cartesian.h>
#include <algorithm>
#include <vector>
#include <CGAL/point_generators_2.h>
#include <CGAL/function_objects.h>
#include <CGAL/Join_input_iterator.h>
#include <CGAL/Counting_iterator.h>

using namespace CGAL;

typedef Simple_cartesian<double>          R;
typedef R::Point_2                        Point;
typedef R::Segment_2                      Segment;
typedef Points_on_segment_2<Point>        PG;
typedef Creator_uniform_2< Point, Segment> Creator;
typedef Join_input_iterator_2< PG, PG, Creator> Segm_iterator;
typedef Counting_iterator<Segm_iterator,Segment> Count_iterator;
typedef std::vector<Segment>              Vector;

int main() {
    // Create test segment set. Prepare a vector for 100 segments.
    Vector segs;
    segs.reserve(100);

    // A horizontal like fan.
    PG p1( Point(-250, -50), Point(-250, 50),50); // Point generator.
    PG p2( Point( 250,-250), Point( 250,250),50);
    Segm_iterator t1( p1, p2); // Segment generator.
    Count_iterator t1_begin( t1); // Finite range.
}

```

```

Count_iterator t1_end( 50);
std::copy( t1_begin, t1_end, std::back_inserter(segs));

// A vertical like fan.
PG p3( Point( -50,-250), Point( 50,-250),50);
PG p4( Point(-250, 250), Point( 250, 250),50);
Segm_iterator t2( p3, p4);
Count_iterator t2_begin( t2);
Count_iterator t2_end( 50);
std::copy( t2_begin, t2_end, std::back_inserter(segs));

CGAL_assertion( segs.size() == 100);
for ( Vector::iterator i = segs.begin(); i != segs.end(); i++){
CGAL_assertion( i->source().x() <= 250);
CGAL_assertion( i->source().x() >= -250);
CGAL_assertion( i->source().y() <= 250);
CGAL_assertion( i->source().y() >= -250);
CGAL_assertion( i->target().x() <= 250);
CGAL_assertion( i->target().x() >= -250);
CGAL_assertion( i->target().y() <= 250);
CGAL_assertion( i->target().y() >= -250);
}
return 0;
}

```


Geometric Object Generators

Reference Manual

Susan Hert, Michael Hoffmann, Lutz Kettner, and Sven Schönherr

This chapter describes the functions and classes provided in CGAL that are useful for generating synthetic test data sets, *e.g.*, for testing algorithms on degenerate object sets and for performance analysis. These include a class for generating random numbers and function for selecting random items from a set of objects, generators for two-dimensional and three-dimensional points sets, a generator for random convex sets and one for simple polygons. The STL algorithm *std::random_shuffle* is useful with these functions and classes to to achieve random permutations for otherwise regular generators (*e.g.*, points on a grid or segment).

47.4 Classified Reference Pages

Concepts

PointGenerator	page 2750
RandomConvexSetTraits_2	page 2759
RandomPolygonTraits_2	page 2760

Functions

CGAL::default_random	page 2743
CGAL::perturb_points_2	page 2744
CGAL::points_on_segment_2	page 2745
CGAL::points_on_square_grid_2	page 2748
CGAL::points_on_cube_grid_3	page 2749
CGAL::random_collinear_points_2	page 2751
CGAL::random_convex_set_2	page 2752
CGAL::random_polygon_2	page 2754
CGAL::random_selection	page 2756

Classes

<i>CGAL::Random</i>	page 2757
<i>CGAL::Points_on_segment_2<Point_2></i>	page 2746
<i>CGAL::Random_points_in_cube_3<Point_3, Creator></i>	page 2762
<i>CGAL::Random_points_in_disc_2<Point_2, Creator></i>	page 2763
<i>CGAL::Random_points_in_sphere_3<Point_3, Creator></i>	page 2764
<i>CGAL::Random_points_in_square_2<Point_2, Creator></i>	page 2765
<i>CGAL::Random_points_on_circle_2<Point_2, Creator></i>	page 2766
<i>CGAL::Random_points_on_segment_2<Point_2, Creator></i>	page 2767
<i>CGAL::Random_points_on_sphere_3<Point_3, Creator></i>	page 2768
<i>CGAL::Random_points_on_square_2<Point_2, Creator></i>	page 2769
<i>CGAL::Random_points_in_iso_box_d<Point_d></i>	page 2770

Traits Class

<i>CGAL::Random_convex_set_traits_2<Kernel></i>	page 2761
-------------------------------------------------------------	-----------

47.5 Alphabetical List of Reference Pages

<i>default_random</i>	page 2743
<i>perturb_points_2</i>	page 2744
<i>PointGenerator</i>	page 2750
<i>points_on_cube_grid_3</i>	page 2749
<i>Points_on_segment_2<Point_2></i>	page 2746
<i>points_on_segment_2</i>	page 2745
<i>points_on_square_grid_2</i>	page 2748
<i>RandomConvexSetTraits_2</i>	page 2759
<i>RandomPolygonTraits_2</i>	page 2760
<i>random_collinear_points_2</i>	page 2751
<i>random_convex_set_2</i>	page 2752
<i>Random_convex_set_traits_2<Kernel></i>	page 2761
<i>Random_points_in_cube_3<Point_3, Creator></i>	page 2762
<i>Random_points_in_disc_2<Point_2, Creator></i>	page 2763
<i>Random_points_in_iso_box_d<Point_d></i>	page 2770
<i>Random_points_in_sphere_3<Point_3, Creator></i>	page 2764
<i>Random_points_in_square_2<Point_2, Creator></i>	page 2765
<i>Random_points_on_circle_2<Point_2, Creator></i>	page 2766
<i>Random_points_on_segment_2<Point_2, Creator></i>	page 2767
<i>Random_points_on_sphere_3<Point_3, Creator></i>	page 2768
<i>Random_points_on_square_2<Point_2, Creator></i>	page 2769
<i>random_polygon_2</i>	page 2754
<i>random_selection</i>	page 2756
<i>Random</i>	page 2757

CGAL::default_random

Definition

The variable *default_random* is the default random numbers generator used for the generator functions and classes.

```
#include <CGAL/Random.h>
```

```
Random                default_random;
```

CGAL::perturb_points_2

Definition

The function *perturb_points_2* perturbs each point in a given range of points by a random amount.

```
#include <CGAL/point_generators_2.h>
```

```
template <class ForwardIterator>
void perturb_points_2(
    ForwardIterator first,
    ForwardIterator last,
    double xeps,
    double yeps = xeps,
    Random& rnd = default_random,
    Creator creator = Creator_uniform_2<Kernel_traits<P>::Kernel::RT,P>)
```

perturbs the points in the range $[first, last)$ by replacing each point with a random point from the $xeps \times yeps$ rectangle centered at the original point. Two random numbers are needed from *rnd* for each point.

Requirements

- *Creator* must be a function object accepting two *double* values *x* and *y* and returning an initialized point (*x*,*y*) of type *P*.
Predefined implementations for these creators like the default are described in Section 45.4.
- The *value_type* of the *ForwardIterator* must be assignable to *P*.
- *P* is equal to the *value_type* of the *ForwardIterator* when using the default initializer.
- The expressions *to_double((*first).x())* and *to_double((*first).y())* must result in the respective coordinate values.

See Also

CGAL::points_on_segment_2 page 2745
 CGAL::points_on_square_grid_2 page 2748
 CGAL::random_selection page 2756
 CGAL::random_selection page 2756
 std::random_shuffle

CGAL::points_on_segment_2

Definition

The function *points_on_segment_2* generates a set of points equally spaced on a segment given the endpoints of the segment.

```
#include <CGAL/point_generators_2.h>
```

```
template <class P, class OutputIterator>
OutputIterator points_on_segment_2( P p, P q, std::size_t n, OutputIterator o)
```

creates n points equally spaced on the segment from p to q ,
i.e. $\forall i : 0 \leq i < n : o[i] := \frac{n-i-1}{n-1} p + \frac{i}{n-1} q$. Returns the value
of o after inserting the n points.

See Also

CGAL::points_on_segment_2 page [2745](#)
 CGAL::points_on_square_grid_2 page [2748](#)
 CGAL::random_collinear_points_2 page [2751](#)

CGAL::Points_on_segment_2<Point_2>

Definition

The class *Points_on_segment_2<Point_2>* is a generator for points on a segment whose endpoints are specified upon construction. The points are equally spaced

Is Model for the Concepts

PointGenerator page [2750](#)

Types

```
typedef std::input_iterator_tag      iterator_category;
typedef Point_2                     value_type;
typedef std::ptrdiff_t              difference_type;
typedef const Point_2*               pointer;
typedef Point_2                     reference;
```

Creation

Points_on_segment_2<Point_2> *g*(*Point_2* *p*, *Point_2* *q*, *std::size_t* *n*, *std::size_t* *i* = 0);

g is an input iterator creating points of type *P* equally spaced on the segment from *p* to *q*. *n* − *i* points are placed on the segment defined by *p* and *q*. Values of the index parameter *i* larger than 0 indicate starting points for the sequence further from *p*. Point *p* has index value 0 and *q* has index value *n* − 1. *Requirement:* The expressions *to_double(p.x())* and *to_double(p.y())* must result in the respective *double* representation of the coordinates of *p*, and similarly for *q*.

Operations

double *g.range()* returns the range in which the point coordinates lie, i.e. $\forall x : |x| \leq \text{range}()$ and $\forall y : |y| \leq \text{range}()$

.

Point_2 *g.source()* returns the source point of the segment.
Point_2 *g.target()* returns the target point of the segment.

See Also

CGAL::copy_n page [2623](#)
CGAL::Counting_iterator page ??

CGAL::points_on_segment<Point_2> page ??
CGAL::Random_points_in_disc_2<Point_2, Creator> page [2763](#)
CGAL::Random_points_in_square_2<Point_2, Creator> page [2765](#)
CGAL::Random_points_on_circle_2<Point_2, Creator> page [2766](#)
CGAL::Random_points_on_segment_2<Point_2, Creator> page [2767](#)
CGAL::Random_points_on_square_2<Point_2, Creator> page [2769](#)
CGAL::random_selection page [2756](#)
std::random_shuffle

CGAL::points_on_square_grid_2

Definition

The function *points_on_square_grid_2* generates a given number of points on a square grid whose size is determined by the number of points to be generated.

```
#include <CGAL/point_generators_2.h>
```

```
template <class OutputIterator, Creator creator>
OutputIterator      points_on_square_grid_2(
                    double a,
                    std::size_t n,
                    OutputIterator o,
                    Creator creator = Creator_uniform_2<Kernel_traits<P>::Kernel::RT,P>)
```

creates the first n points on the regular $\lceil \sqrt{n} \rceil \times \lceil \sqrt{n} \rceil$ grid within the square $[-a, a] \times [-a, a]$. Returns the value of o after inserting the n points.

Requirements

- *Creator* must be a function object accepting two *double* values x and y and returning an initialized point (x, y) of type P . Predefined implementations for these creators like the default can be found in Section 45.4.
- The *OutputIterator* must accept values of type P . If the *OutputIterator* has a *value_type* the default initializer of the *creator* can be used. P is set to the *value_type* in this case.

See Also

CGAL::perturb_points_2 page 2744
 CGAL::points_on_segment_2 page 2745
 CGAL::points_on_cube_grid_3 page 2749
 CGAL::random_collinear_points_2 page 2751
 CGAL::random_selection page 2756
 std::random_shuffle

CGAL::points_on_cube_grid_3

Definition

The function *points_on_cube_grid_3* generates a given number of points on a cubic grid whose size is determined by the number of points to be generated.

```
#include <CGAL/point_generators_3.h>
```

```
template <class OutputIterator, Creator creator>
OutputIterator      points_on_cube_grid_3(
    double a,
    std::size_t n,
    OutputIterator o,
    Creator creator = Creator_uniform_3<Kernel_traits<P>::Kernel::RT,P>)
```

creates the first n points on the regular $\lceil n^{1/3} \rceil \times \lceil n^{1/3} \rceil \times \lceil n^{1/3} \rceil$ grid within the cube $[-a, a] \times [-a, a] \times [-a, a]$. Returns the value of o after inserting the n points.

Requirements

- *Creator* must be a function object accepting three *double* values x , y , and z and returning an initialized point (x, y, z) of type P . Predefined implementations for these creators like the default can be found in Section 45.4.
- The *OutputIterator* must accept values of type P . If the *OutputIterator* has a *value_type* the default initializer of the *creator* can be used. P is set to the *value_type* in this case.

CGAL::points_on_square_grid_2 page [2748](#)
 CGAL::random_selection page [2756](#)

PointGenerator

Definition

The concept `PointGenerator` defines the requirements for a point generator, which can be used in places where input iterators are called for.

Refines

`InputIterator`

Has Models

<code>CGAL::Random_points_in_disc_2<Point_2, Creator></code>	page 2763
<code>CGAL::Random_points_in_square_2<Point_2, Creator></code>	page 2765
<code>CGAL::Random_points_on_circle_2<Point_2, Creator></code>	page 2766
<code>CGAL::Random_points_on_segment_2<Point_2, Creator></code>	page 2767
<code>CGAL::Random_points_on_square_2<Point_2, Creator></code>	page 2769
<code>CGAL::Random_points_in_cube_3<Point_3, Creator></code>	page 2762
<code>CGAL::Random_points_in_sphere_3<Point_3, Creator></code>	page 2764
<code>CGAL::Random_points_on_sphere_3<Point_3, Creator></code>	page 2768
<code>CGAL::Random_points_in_iso_box_d<Point_d></code>	page 2770

Types

`PointGenerator::value_type` the type of point being generated.

Operations

`double` `pg.range() const` return an absolute bound for the coordinates of all generated points.

CGAL::random_collinear_points_2

Definition

```
#include <CGAL/point_generators_2.h>
```

```
template <class RandomAccessIterator, class OutputIterator>
OutputIterator      random_collinear_points_2(
                    RandomAccessIterator first,
                    RandomAccessIterator last,
                    std::size_t n,
                    OutputIterator first2,
                    Random& rnd = default_random,
                    Creator creator = Creator_uniform_2<Kernel_traits<P>::Kernel::RT,P>)
```

randomly chooses two points from the range $[first, last)$, creates a random third point on the segment connecting these two points, writes it to $first2$, and repeats this n times, thus writing n points to $first2$ that are collinear with points in the range $[first, last)$. Three random numbers are needed from rnd for each point. Returns the value of $first2$ after inserting the n points.

Requirements

- *Creator* must be a function object accepting two *double* values x and y and returning an initialized point (x, y) of type P . Predefined implementations for these creators like the default can be found in Section 45.4.
- The *value_type* of the *RandomAccessIterator* must be assignable to P . P is equal to the *value_type* of the *RandomAccessIterator* when using the default initializer.
- The expressions $to_double((*first).x())$ and $to_double((*first).y())$ must result in the respective coordinate values.

See Also

CGAL::perturb_points_2 page 2744
 CGAL::points_on_segment_2 page 2745
 CGAL::points_on_square_grid_2 page 2748
 CGAL::random_selection page 2756
 std::random_shuffle

CGAL::random_convex_set_2

Definition

The function *random_convex_set_2* computes a random convex planar point set of given size where the points are drawn from a specific domain.

```
#include <CGAL/random_convex_set_2.h>
```

```
template < class OutputIterator, class PointGenerator, class Traits >
OutputIterator      random_convex_set_2(
                    int n,
                    OutputIterator o,
                    PointGenerator pg,
                    Traits t = Default_traits)
```

computes a random convex n -gon by writing its vertices (oriented counterclockwise) to o . The resulting polygon is scaled such that it fits into the bounding box as specified by pg . Therefore we cannot easily describe the resulting distribution.

Precondition: $n \geq 3$.

Requirements

1. *PointGenerator* is a model of the concept `PointGenerator`
2. *Traits* is a model of the concept `RandomConvexSetTraits_2`
3. *Point_generator::value_type* is equivalent to *Traits::Point_2* and *OutputIterator::value_type*.
4. if *Traits* is not specified, *Point_generator::value_type* must be *Point_2*< R > for some representation class R ,

The default traits class *Default_traits* is *Random_convex_set_traits_2*.

See Also

CGAL::Random_points_in_square_2<*Point_2*, *Creator*> page [2765](#)
CGAL::Random_points_in_disc_2<*Point_2*, *Creator*> page [2763](#)

Implementation

The implementation uses the centroid method described in [Sch96] and has a worst case running time of $O(r \cdot n + n \cdot \log n)$, where r is the time needed by *pg* to generate a random point.

Example

The following program displays a random convex 500-gon where the points are drawn uniformly from the unit square centered at the origin.

```
// file: examples/Generator/rcs_example.C

#include <CGAL/Cartesian.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_convex_set_2.h>

#include <iostream>
#include <iterator>

typedef CGAL::Cartesian< double >    K;
typedef K::Point_2                   Point_2;
typedef CGAL::Random_points_in_square_2<
    Point_2,
    CGAL::Creator_uniform_2< double, Point_2 > >
    Point_generator;

int main() {
    // create 500-gon and write it into a window:
    CGAL::random_convex_set_2(
        500,
        std::ostream_iterator<Point_2>(std::cout, "\n"),
        Point_generator( 0.5));
    return 0;
}
```

CGAL::random_polygon_2

Definition

The function *random_polygon_2* constructs a random simple polygon from points that are drawn from a specific domain. Though each simple polygon defined on this set of points has a non-zero probability of being constructed, some polygons may have higher probabilities than others. The overall distribution of the generated polygons is not known since it depends on the generated points.

```
#include <CGAL/random_polygon_2.h>
```

```
template < class OutputIterator, class PointGenerator, class Traits >
OutputIterator      random_polygon_2( int n,
                                     OutputIterator result,
                                     PointGenerator pg,
                                     Traits t = Default_traits)
```

computes a random simple polygon by writing its vertices (oriented counterclockwise) to *result*. The polygon generated will have a number of vertices equal to the number of unique points in the first n points generated by *pg*.

Requirements

1. *Traits* is a model of the concept `RandomPolygonTraits_2`
2. `PointGenerator::value_type` is equivalent to `Traits::Point_2` and `OutputIterator::value_type`.

The default traits class *Default_traits* is the kernel in which `Traits::Point_2` is defined.

See Also

`CGAL::Random_points_in_disc_2<Point_2, Creator>` [page 2763](#)
`CGAL::Random_points_in_square_2<Point_2, Creator>` [page 2765](#)

Implementation

The implementation is based on the method of eliminating self-intersections in a polygon by using so-called “2-opt” moves. Such a move eliminates an intersection between two edges by reversing the order of the vertices between the edges. No more than $O(n^3)$ such moves are required to simplify a polygon defined on n points [vLS82]. Intersecting edges are detected using a simple sweep through the vertices and then one intersection is chosen at random to eliminate after each sweep. The worse-case running time is therefore $O(n^4 \log n)$.

Example

The following program displays a random simple polygon with up to 50 vertices, where the vertex coordinates are drawn uniformly from the unit square centered at the origin.

```

// file: examples/Generator/random_poly_example.C

#include <CGAL/Cartesian.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/random_polygon_2.h>
#include <CGAL/Polygon_2.h>

typedef CGAL::Cartesian< double >          K;
typedef K::Point_2                         Point_2;
typedef std::list<Point_2>                 Container;
typedef CGAL::Polygon_2<K, Container>      Polygon_2;
typedef CGAL::Random_points_in_square_2< Point_2 > Point_generator;

int main() {
    Polygon_2 polygon;
    // create 50-gon and write it into a window:
    CGAL::random_polygon_2(50, std::back_inserter(polygon),
                          Point_generator(0.5));

    std::cout << polygon;
    return 0;
}

```

CGAL::random_selection

Definition

random_selection chooses n items at random from a random access iterator range which is useful to produce degenerate input data sets with multiple entries of identical items.

```
#include <CGAL/random_selection.h>
```

```
template <class RandomAccessIterator, class Size, class OutputIterator, class Random>
OutputIterator      random_selection( RandomAccessIterator first,
                                     RandomAccessIterator last,
                                     Size n,
                                     OutputIterator result,
                                     Random& rnd = default_random)
```

chooses a random item from the range $[first, last)$ and writes it to *result*, each item from the range with equal probability, and repeats this n times, thus writing n items to *result*. A single random number is needed from *rnd* for each item. Returns the value of *result* after inserting the n items.

Precondition: *Random* is a random number generator type as provided by the STL or by *Random*.

See Also

CGAL::perturb_points_2 page [2744](#)

CGAL::Random

Definition

The class *Randomis* is a random numbers generator. It generates uniformly distributed random *bools*, *ints* and *doubles*. It can be used as the random number generating function object in the STL algorithm *random_shuffle*.

Instances of \mathcal{S} can be seen as input streams. Different streams are *independent* of each other, i.e. the sequence of numbers from one stream does *not* depend upon how many numbers were extracted from the other streams.

It can be very useful, e.g. for debugging, to reproduce a sequence of random numbers. This can be done by either initialising deterministically or using the state functions as described below.

```
#include <CGAL/Random.h>
```

Types

<i>Random::State</i>	State type.
----------------------	-------------

Creation

Random random; introduces a variable *random* of type *Random*.

Random *random*(*long seed*); introduces a variable *random* of type *Random* and initializes its internal state using *seed*. Equal values for *seed* result in equal sequences of random numbers.

Random *random*(*State* *state*); introduces a variable *random* of type *Random* and initializes its internal state with *state*.

Operations

<i>bool</i>	<i>random.get_bool()</i>	returns a random <i>bool</i> .
-------------	--------------------------	--------------------------------

<i>template <int b></i>		
<i>int</i>	<i>random.get_bits()</i>	returns a random <i>int</i> value from the interval $[0, 2^b)$. This is supposed to be efficient.

int *random.get_int(int lower, int upper)*

returns a random *int* from the interval $[lower, upper)$.

double *random.get_double(double lower = 0.0, double upper = 1.0)*

returns a random *double* from the interval $[lower, upper)$.

int *random(int upper)* returns *random.get_int(0, upper)*.

State Functions

void *random.save_state(State& state)*
saves the current internal state in *state*.

void *random.restore_state(State state)*
restores the internal state from *state*.

Equality Test

bool *random == random2* returns *true*, iff *random* and *random2* have equal internal states.

Implementation

We use the C library function *erand48* to generate the random numbers, *i.e.*, the sequence of numbers depends on the implementation of *erand48* on your specific platform.

See Also

CGAL::default_random [page 2743](#)

RandomConvexSetTraits_2

Definition

The concept `RandomConvexSetTraits_2` describes the requirements of the traits class for the function `random_convex_set_2`.

Has Models

`CGAL::Random_convex_set_traits_2<Kernel>` page [2761](#)

Types

<code>RandomConvexSetTraits_2:: Point_2</code>	point class.
<code>RandomConvexSetTraits_2:: FT</code>	class used for doing computations on point and vector coordinates (has to fulfill field type requirements).
<code>RandomConvexSetTraits_2:: Sum</code>	AdaptableBinaryFunction class: $Point_2 \times Point_2 \rightarrow Point_2$. It returns the point that results from adding the vectors corresponding to both arguments.
<code>RandomConvexSetTraits_2:: Scale</code>	AdaptableBinaryFunction class: $Point_2 \times FT \rightarrow Point_2$. <code>Scale(p,k)</code> returns the point that results from scaling the vector corresponding to p by a factor of k .
<code>RandomConvexSetTraits_2:: Max_coordinate</code>	AdaptableUnaryFunction class: $Point_2 \rightarrow FT$. <code>Max_coordinate(p)</code> returns the coordinate of p with largest absolute value.
<code>RandomConvexSetTraits_2:: Angle_less</code>	AdaptableBinaryFunction class: $Point_2 \times Point_2 \rightarrow bool$. It returns <code>true</code> , iff the angle of the direction corresponding to the first argument with respect to the positive x -axis is less than the angle of the direction corresponding to the second argument.

Operations

<code>Point_2</code>	<code>t.origin() const</code>	return origin (neutral element for the <code>Sum</code> operation).
----------------------	-------------------------------	---------------------------------------------------------------------

RandomPolygonTraits_2

Definition

The concept `RandomPolygonTraits_2` describes the requirements for the traits class used by the function `random_polygon_2`.

Has Models

The CGAL kernels.

Types

<code>RandomPolygonTraits_2:: FT</code>	The coordinate type of the points of the polygon (<i>i.e.</i> , a field type)
<code>RandomPolygonTraits_2:: Point_2</code>	The point type of the polygon.
<code>RandomPolygonTraits_2:: Orientation_2</code>	Predicate object type that determines the orientation of three points. It must provide <i>Orientation operator</i> <code>(Point_2 p, Point_2 q, Point_2 r)</code> that returns <i>LEFT_TURN</i> , if <i>r</i> lies to the left of the oriented line <i>l</i> defined by <i>p</i> and <i>q</i> , returns <i>RIGHT_TURN</i> if <i>r</i> lies to the right of <i>l</i> , and returns <i>COLLINEAR</i> if <i>r</i> lies on <i>l</i> .
<code>RandomPolygonTraits_2:: Less_xy_2</code>	Binary predicate object type comparing <i>Point_2</i> s lexicographically. It must provide <i>bool operator</i> <code>(Point_2 p, Point_2 q)</code> that returns <i>true</i> iff $p <_{xy} q$. We have $p <_{xy} q$, iff $p_x < q_x$ or $p_x = q_x$ and $p_y < q_y$, where p_x and p_y denote the <i>x</i> and <i>y</i> coordinates of point <i>p</i> , resp.

Operations

The following two member functions returning instances of the above predicate object types are required.

<code>Less_xy_2</code>	<code>t.less_xy_2_object()</code>
<code>Orienation_2</code>	<code>t.orientation_2_object()</code>

CGAL::Random_convex_set_traits_2<Kernel>

Definition

The class *Random_convex_set_traits_2<Kernel>* serves as a traits class for the function *random_convex_set_2*.

`#include <CGAL/Random_convex_set_traits_2.h>`

Is Model for the Concepts

RandomConvexSetTraits_2 page [2759](#)

Types

`typedef Kernel::Point_2`

`Point_2;`
`typedef Kernel::FT FT;`

`Random_convex_set_traits_2<Kernel>:: Sum` function object class derived from `std::binary_function<Point_2, Point_2, Point_2>`

`Random_convex_set_traits_2<Kernel>:: Scale` function object class derived from `std::binary_function<Point_2, Point_2, Point_2>`

`Random_convex_set_traits_2<Kernel>:: Max_coordinate`

function object class derived from `std::unary_function<Point_2, FT>`

`Random_convex_set_traits_2<Kernel>:: Angle_less`

function object class derived from `std::binary_function<Point_2, Point_2, bool>`

Creation

`Random_convex_set_traits_2<Kernel> t;` default constructor

Operations

`Point_2 t.origin() const` returns CGAL::ORIGIN.

CGAL::Random_points_in_cube_3<Point_3, Creator>

Definition

The class *Random_points_in_cube_3<Point_3, Creator>* is an input iterator creating points uniformly distributed in a half-open cube. The default *Creator* is *Creator_uniform_3<Kernel_traits<P>::Kernel::RT,P>*.

Is Model for the Concepts

InputIterator

PointGenerator page [2750](#)

```
#include <CGAL/point_generators_3.h>
```

Types

```
typedef std::input_iterator_tag      iterator_category;
typedef Point_3                     value_type;
typedef std::ptrdiff_t              difference_type;
typedef const Point_3*               pointer;
typedef Point_3                     reference;
```

Operations

```
Random_points_in_cube_3<Point_3, Creator> g( double a, Random& rnd = default_random);
```

g is an input iterator creating points of type *Point_3* uniformly distributed in the half-open cube with side length $2a$, centered at the origin, i.e. $\forall p = *g : -a \leq p.x(), p.y(), p.z() < a$. Three random numbers are needed from *rnd* for each point.

See Also

CGAL::copy_n page [2623](#)
CGAL::Counting_iterator page ??
CGAL::Random_points_in_square_2<Point_2, Creator> page [2765](#)
CGAL::Random_points_in_sphere_3<Point_3, Creator> page [2764](#)
CGAL::Random_points_on_sphere_3<Point_3, Creator> page [2768](#) *std::random_shuffle*

CGAL::Random_points_in_disc_2<Point_2, Creator>

Definition

The class *Random_points_in_disc_2<Point_2, Creator>* is an input iterator creating points uniformly distributed in an open disc. The default *Creator* is *Creator_uniform_2<Kernel_traits<P>::Kernel::RT,P>*.

Is Model for the Concepts

InputIterator

PointGenerator page 2750

#include <CGAL/point_generators_2.h>

Types

```
typedef std::input_iterator_tag      iterator_category;
typedef Point_2                     value_type;
typedef std::ptrdiff_t              difference_type;
typedef const Point_2*              pointer;
typedef Point_2                     reference;
```

Operations

Random_points_in_disc_2<Point_2, Creator> *g*(double *r*, Random& *rnd* = *default_random*);

g is an input iterator creating points of type *Point_2* uniformly distributed in the open disc with radius *r*, i.e. $|*g| < r$. Two random numbers are needed from *rnd* for each point.

See Also

CGAL::copy_n page 2623
CGAL::Counting_iterator page ??
CGAL::Points_on_segment_2<Point_2> page 2746
CGAL::Random_points_in_square_2<Point_2, Creator> page 2765
CGAL::Random_points_on_circle_2<Point_2, Creator> page 2766
CGAL::Random_points_on_segment_2<Point_2, Creator> page 2767
CGAL::Random_points_on_square_2<Point_2, Creator> page 2769
CGAL::Random_points_in_sphere_3<Point_3, Creator> page 2764 *std::random_shuffle*

CGAL::Random_points_in_sphere_3<Point_3, Creator>

Definition

The class *Random_points_in_sphere_3<Point_3, Creator>* is an input iterator creating points uniformly distributed in an open sphere. The default *Creator* is *Creator_uniform_3<Kernel_traits<P>::Kernel::RT,P>*.

Is Model for the Concepts

InputIterator

PointGenerator page [2750](#)

```
#include <CGAL/point_generators_3.h>
```

Types

```
typedef std::input_iterator_tag    iterator_category;
typedef Point_3                   value_type;
typedef std::ptrdiff_t            difference_type;
typedef const Point_3*            pointer;
typedef Point_3                   reference;
```

Operations

```
Random_points_in_sphere_3<Point_3, Creator> g( double r, Random& rnd = default_random);
```

g is an input iterator creating points of type *Point_3* uniformly distributed in the open sphere with radius *r*, i.e. $|*g| < r$. Three random numbers are needed from *rnd* for each point.

See Also

CGAL::copy_n page [2623](#)
CGAL::Counting_iterator page [??](#)
CGAL::Random_points_in_disc_2<Point_2, Creator> page [2763](#)
CGAL::Random_points_in_cube_3<Point_3, Creator> page [2762](#)
CGAL::Random_points_on_sphere_3<Point_3, Creator> page [2768](#) *std::random_shuffle*

CGAL::Random_points_in_square_2<Point_2, Creator>

Definition

The class *Random_points_in_square_2<Point_2, Creator>* is an input iterator creating points uniformly distributed in a half-open square. The default *Creator* is *Creator_uniform_2<Kernel_traits<P>::Kernel::RT,P>*.

Is Model for the Concepts

InputIterator

PointGenerator page [2750](#)

`#include <CGAL/point_generators_2.h>`

Types

<code>typedef std::input_iterator_tag</code>	<code>iterator_category;</code>
<code>typedef Point_2</code>	<code>value_type;</code>
<code>typedef std::ptrdiff_t</code>	<code>difference_type;</code>
<code>typedef const Point_2*</code>	<code>pointer;</code>
<code>typedef Point_2</code>	<code>reference;</code>

Operations

Random_points_in_square_2<Point_2, Creator> *g*(*double a*, *Random& rnd* = *default_random*);

g is an input iterator creating points of type *Point_2* uniformly distributed in the half-open square with side length $2a$, centered at the origin, i.e. $\forall p = *g : -a \leq p.x() < a$ and $-a \leq p.y() < a$. Two random numbers are needed from *rnd* for each point.

See Also

CGAL::copy_n page [2623](#)
CGAL::Counting_iterator page [??](#)
CGAL::Points_on_segment_2<Point_2> page [2746](#)
CGAL::Random_points_in_disc_2<Point_2, Creator> page [2763](#)
CGAL::Random_points_on_segment_2<Point_2, Creator> page [2767](#)
CGAL::Random_points_on_square_2<Point_2, Creator> page [2769](#)
CGAL::Random_points_in_cube_3<Point_3, Creator> page [2762](#)
std::random_shuffle

CGAL::Random_points_on_circle_2<Point_2, Creator>

Definition

The class *Random_points_on_circle_2<Point_2, Creator>* is an input iterator creating points uniformly distributed on a circle. The default *Creator* is *Creator_uniform_2<Kernel_traits<P>::Kernel::RT,P>*.

Is Model for the Concepts

InputIterator

PointGenerator page [2750](#)

```
#include <CGAL/point_generators_2.h>
```

Types

```
typedef std::input_iterator_tag      iterator_category;
typedef Point_2                     value_type;
typedef std::ptrdiff_t              difference_type;
typedef const Point_2*              pointer;
typedef Point_2                     reference;
```

Operations

```
Random_points_on_circle_2<Point_2, Creator> g( double r, Random& rnd = default_random);
```

g is an input iterator creating points of type *Point_2* uniformly distributed on the circle with radius *r*, i.e. $|*g| == r$. A single random number is needed from *rnd* for each point.

See Also

See Also

```
CGAL::copy_n ..... page 2623
CGAL::Counting_iterator ..... page ??
CGAL::Points_on_segment_2<Point_2> ..... page 2746
CGAL::Random_points_in_disc_2<Point_2, Creator> ..... page 2763
CGAL::Random_points_in_square_2<Point_2, Creator> ..... page 2765
CGAL::Random_points_on_segment_2<Point_2, Creator> ..... page 2767
CGAL::Random_points_on_square_2<Point_2, Creator> ..... page 2769
CGAL::Random_points_on_sphere_3<Point_3, Creator> ..... page 2768
std::random_shuffle
```

CGAL::Random_points_on_segment_2<Point_2, Creator>

Definition

The class *Random_points_on_segment_2<Point_2, Creator>* is an input iterator creating points uniformly distributed on a segment. The default *Creator* is *Creator_uniform_2<Kernel_traits<P>::Kernel::RT,P>*.

Is Model for the Concepts

InputIterator

PointGenerator page [2750](#)

#include <CGAL/point_generators_2.h>

Types

```
typedef std::input_iterator_tag      iterator_category;
typedef Point_2                     value_type;
typedef std::ptrdiff_t              difference_type;
typedef const Point_2*              pointer;
typedef Point_2                     reference;
```

Operations

Random_points_on_segment_2<Point_2, Creator> *g*(*Point_2* *p*, *Point_2* *q*, *Random& rnd* = *default_random*);

g is an input iterator creating points of type *Point_2* uniformly distributed on the segment from *p* to *q* (excluding *q*), i.e. $*g == (1 - \lambda)p + \lambda q$ where $0 \leq \lambda < 1$. A single random number is needed from *rnd* for each point.

Requirement: The expressions *to_double(p.x())* and *to_double(p.y())* must result in the respective *double* representation of the coordinates of *p*, and similarly for *q*.

See Also

CGAL::copy_n page [2623](#)
CGAL::Counting_iterator page [??](#)
CGAL::Points_on_segment_2<Point_2> page [2746](#)
CGAL::Random_points_in_disc_2<Point_2, Creator> page [2763](#)
CGAL::Random_points_in_square_2<Point_2, Creator> page [2765](#)
CGAL::Random_points_on_circle_2<Point_2, Creator> page [2766](#)
CGAL::Random_points_on_square_2<Point_2, Creator> page [2769](#)
std::random_shuffle

CGAL::Random_points_on_sphere_3<Point_3, Creator>

Definition

The class *Random_points_on_sphere_3<Point_3, Creator>* is an input iterator creating points uniformly distributed on a sphere. The default *Creator* is *Creator_uniform_3<Kernel_traits<P>::Kernel::RT,P>*.

Is Model for the Concepts

InputIterator

PointGenerator page [2750](#)

```
#include <CGAL/point_generators_3.h>
```

Types

```
typedef std::input_iterator_tag    iterator_category;
typedef Point_3                   value_type;
typedef std::ptrdiff_t            difference_type;
typedef const Point_3*            pointer;
typedef Point_3                   reference;
```

Operations

```
Random_points_on_sphere_3<Point_3, Creator> g( double r, Random& rnd = default_random);
```

g is an input iterator creating points of type *Point_3* uniformly distributed on the boundary of a sphere with radius *r*, i.e. $|*g| == r$. Two random numbers are needed from *rnd* for each point.

See Also

CGAL::copy_n page [2623](#)
CGAL::Counting_iterator page [??](#)
CGAL::Random_points_on_circle_2<Point_2, Creator> page [2766](#)
CGAL::Random_points_in_cube_3<Point_3, Creator> page [2762](#)
CGAL::Random_points_in_sphere_3<Point_3, Creator> page [2764](#)
std::random_shuffle

CGAL::Random_points_on_square_2<Point_2, Creator>

Definition

The class *Random_points_on_square_2<Point_2, Creator>* is an input iterator creating points uniformly distributed in the boundary of a square. The default *Creator* is *Creator_uniform_2<Kernel_traits<P>::Kernel::RT,P>*.

Is Model for the Concepts

InputIterator

PointGenerator page 2750

```
#include <CGAL/point_generators_2.h>
```

Types

```
typedef std::input_iterator_tag      iterator_category;
typedef Point_2                     value_type;
typedef std::ptrdiff_t              difference_type;
typedef const Point_2*              pointer;
typedef Point_2                     reference;
```

Operations

```
Random_points_on_square_2<Point_2, Creator> g( double a, Random& rnd = default_random);
```

g is an input iterator creating points of type *Point_2* uniformly distributed on the boundary of the square with side length $2a$, centered at the origin, i.e. $\forall p = *g$: one coordinate is either a or $-a$ and for the other coordinate c holds $-a \leq c < a$. A single random number is needed from *rnd* for each point.

See Also

```
CGAL::copy_n ..... page 2623
CGAL::Counting_iterator ..... page ??
CGAL::Points_on_segment_2<Point_2> ..... page 2746
CGAL::Random_points_in_disc_2<Point_2, Creator> ..... page 2763
CGAL::Random_points_in_square_2<Point_2, Creator> ..... page 2765
CGAL::Random_points_on_circle_2<Point_2, Creator> ..... page 2766
CGAL::Random_points_on_segment_2<Point_2, Creator> ..... page 2767
std::random_shuffle
```

CGAL::Random_points_in_iso_box_d<Point_d>

Definition

The class *Random_points_in_iso_box_d<Point_d>* is an input iterator creating points uniformly distributed in a half-open d-dimensional iso box. The default *Creator* is *Creator_uniform_d<Kernel_traits<Point_d>::Kernel::RT, Point_d>*.

Is Model for the Concepts

InputIterator

PointGenerator page [2750](#)

```
#include <CGAL/point_generators_d.h>
```

Types

```
typedef std::input_iterator_tag    iterator_category;
typedef Point_d                   value_type;
typedef std::ptrdiff_t             difference_type;
typedef const Point_d*             pointer;
typedef Point_d                   reference;
```

Operations

```
Random_points_in_iso_box_d<Point_d> g( int dim, double a, Random& rnd = default_random);
```

g is an input iterator creating points of type *Point_d* uniformly distributed in the half-open d-dimensional iso box with side length $2a$, centered at the origin, i.e. $\forall p = *g : -a \leq p.x(), p.y(), p.z() < a$. d random numbers are needed from *rnd* for each point.

Chapter 48

Timers, Hash Map, Union-find, Modifiers

Lutz Kettner, Sylvain Pion, and Michael Seel

48.1 Timers

CGAL provides classes for measuring the user process time and the real time. The class *CGAL::Timer* is the version for the user process time and the class *CGAL::Real_timer* is the version for the real time.

Instantiations of both classes are objects with a state. The state is either *running* or it is *stopped*. The state of an object *t* is controlled with *t.start()* and *t.stop()*. The timer counts the time elapsed since its creation or last reset. It counts only the time where it is in the running state. The time information is given in seconds. The timer counts also the number of intervals it was running, i.e. it counts the number of calls of the *start()* member function since the last reset. If the reset occurs while the timer is running it counts as the first interval.

48.2 Memory Size

CGAL provides access to the memory size used by the program with the *CGAL::Memory_sizer* class. Both the virtual memory size and the resident size are available (the resident size does not account for swapped out memory nor for the memory which is not yet paged-in).

48.3 Unique Hash Map

The class *Unique_hash_map* implements an injective mapping between a set of unique keys and a set of data values. This is implemented using a chained hashing scheme and access operations take $O(1)$ expected time. Such a mapping is useful, for example, when keys are pointers, handles, iterators or circulators that refer to unique memory locations. In this case, the default hash function is *Handle_hash_function*.

48.4 Union-find

CGAL also provides a class *Union_find* that implements a partition of values into disjoint sets. This is implemented with union by rank and path compression. The running time for m set operations on n elements is

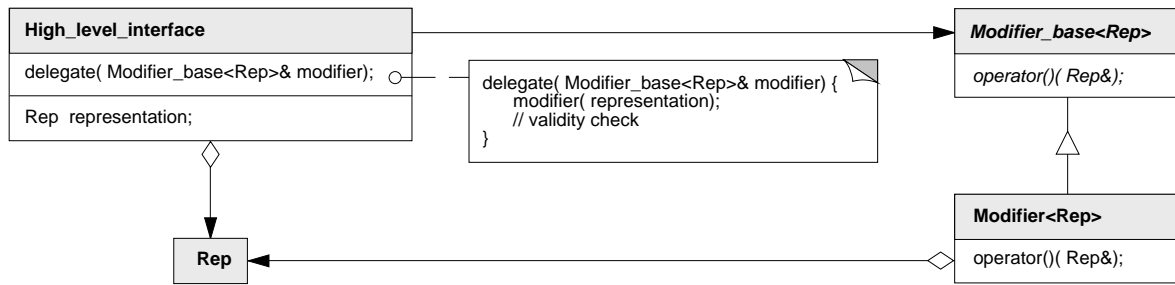


Figure 48.1: Class diagram for the modifier. It illustrates the safe access to an internal representation through an high-level interface.

$O(n\alpha(m,n))$ where $\alpha(m,n)$ is the extremely slowly growing inverse of Ackermann's function.

48.5 Protected Access to Internal Representations

High level data structures typically maintain integrity of an internal data representation, which they protect from the user. A minimal while complete interface of the data structure allows manipulations in the domain of valid representations. Additional operations might benefit from being allowed to access the internal data representation directly. An example are intermediate steps within an algorithm where the internal representation would be invalid. We present a general method to accomplish access in a safe manner, such that the high level data structures can guarantee validity after the possibly compromising algorithm has finished its work. An example are polyhedral surfaces in the Basic Library, where a construction process like for a file scanner could be performed more efficiently on the internal halfedge data structure than by using the high-level Euler operators of the polyhedron.

The solution provided here is inspired by the strategy pattern [GHJV95], though it serves a different intent, see Figure 48.1. The abstract base class *Modifier_base<R>* declares a pure virtual member function *operator()* that accepts a single reference parameter of the internal representation type. The member function *delegate()* of the high-level interface calls this *operator()* with its internal representation. An actual modifier implements this virtual function, thus gaining access to the internal representation. Once, the modifier has finished its work, the member function *delegate()* is back in control and can check the validity of the internal representation. Summarizing, a user can implement and apply arbitrary functions based on the internal representation and keeps the benefit if a protected high-level interface. User provided modifiers must in any case return a valid internal representation or the checker in the high-level interface is allowed (and supposed) to abort the program. The indirection via the virtual function invocation is negligible for operations that consists of more than a pointer update or integer addition.

Timers, Hash Map, Union-find, Modifiers Reference Manual

Lutz Kettner, Sylvain Pion, and Michael Seel

This chapter describes classes for measuring user process time and real time as well as the memory size.

A hash map `CGAL::Unique_hash_map` is offered that is specialized on unique hash values of type `std::size_t`, i.e., it is particularly useful for pointers, handles, iterators, and circulators as key values.

Furthermore, a union-find data structure and the modifier base class is documented.

48.6 Classified Reference Pages

Concepts

`UniqueHashFunction` page [2784](#)

Classes

`CGAL::Timer` page [2779](#)

`CGAL::Real_timer` page [2778](#)

`CGAL::Memory_sizer` page [2776](#)

`CGAL::Unique_hash_map<Key,Data,UniqueHashFunction>` page [2782](#)

`CGAL::Handle_hash_function` page [2775](#)

`CGAL::Union_find<T,A>` page [2780](#)

`CGAL::Modifier_base<R>` page [2777](#)

48.7 Alphabetical List of Reference Pages

`Handle_hash_function` page [2775](#)

`Memory_sizer` page [2776](#)

`Modifier_base<R>` page [2777](#)

<i>Real_timer</i>	page 2778
<i>Timer</i>	page 2779
<i>Union_find</i> < <i>T</i> , <i>A</i> >	page 2780
<i>UniqueHashFunction</i>	page 2784
<i>Unique_hash_map</i> < <i>Key</i> , <i>Data</i> , <i>UniqueHashFunction</i> >	page 2782

CGAL::Handle hash function

Definition

The class *Handle_hash_function* is a model for the *UniqueHasFunction* concept. It is applicable for all key types with pointer-like functionality, such as handles, iterators, and circulators. Specifically, for a *key* value the expression `&*key` must return a unique address.

```
#include <CGAL/Handle_hash_function.h>
```

Is Model for the Concepts

UniqueHashFunction.....page 2784

Creation

<i>Handle_hash_function</i>	<i>hash</i> ;	default constructor.
-----------------------------	---------------	----------------------

Operations

<i>template <class Handle></i>		
<i>std::size_t</i>	<i>hash(Handle key)</i>	returns unique hash value for any <i>Handle</i> type for which &*key gives a unique address.
		<i>Requirement:</i> The type <i>std::iterator_traits<Handle>::value_</i> <i>type</i> has to be defined (which it is already for pointers, han- dles, iterators, and circulators).

See Also

CGAL::Unique_hash_map<Key,Data,UniqueHashFunction>page 2782

Implementation

Plain type cast of `&*key` to `std::size_t` and divided by the size of the `std::iterator_traits<Handle>::value_type` to avoid correlations with the internal table size, which is a power of two.

CGAL::Memory_sizer

Definition

The class *Memory_sizer* allows to measure the memory size used by the process. Both the virtual memory size and the resident size are available (the resident size does not account for swapped out memory nor for the memory which is not yet paged-in).

```
#include <CGAL/Memory_sizer.h>
```

Types

The memory sizes are given in bytes.

```
typedef std::size_t    size_type;
```

Creation

```
Memory_sizer m;           Default constructor.
```

Operations

```
size_type
```

```
    m.virtual_size()
```

Returns the virtual memory size.

```
size_type
```

```
    m.resident_size()
```

Returns the resident memory size.

Implementation

Accessing this information requires the use of non-portable code. Currently, there is support for Linux platforms and the Microsoft and Intel compiler on Windows. If a platform is not supported, then the macro *CGAL_DONT_HAVE_MEMORY_SIZER* is defined by this file.

CGAL::Modifier_base<R>

Definition

Modifier_base<R> is an abstract base class providing the interface for any modifier. A modifier is a function object derived from *Modifier_base<R>* that implements the pure virtual member function *operator()*, which accepts a single reference parameter *R*& on which the modifier is allowed to work. *R* is the type of the internal representation that is to be modified.

```
#include <CGAL/Modifier_base.h>
```

Types

```
typedef R      Representation;           the internal representation type.
```

Operations

```
virtual void    modifier.operator()( R& rep)    Postcondition: rep is a valid representation.
```

Example

The following fragment defines a class *A* with an internal representation *i* of type *int*. It provides a member function *delegate()*, which gives a modifier access to the internal variable and checks validity thereafter. The example modifier sets the internal variable to 42. The example function applies the modifier to an instance of class *A*.

```
class A {
    int i; // protected internal representation
public:
    void delegate( CGAL::Modifier_base<int>& modifier) {
        modifier(i);
        CGAL_postcondition( i > 0); // check validity
    }
};

struct Modifier : public CGAL::Modifier_base<int> {
    void operator()( int& rep) { rep = 42;}
};

void use_it() {
    A a;
    Modifier m;
    a.delegate(m); // a.i == 42 and A has checked that A::i > 0.
}
```

CGAL::Real_timer

Definition

```
#include <CGAL/Real_timer.h>
```

The class *Real_timer* is a timer class for measuring real time. A timer *t* of type *Real_timer* is an object with a state. It is either *running* or it is *stopped*. The state is controlled with *t.start()* and *t.stop()*. The timer counts the time elapsed since its creation or last reset. It counts only the time where it is in the running state. The time information is given in seconds. The timer counts also the number of intervals it was running, i.e. it counts the number of calls of the *start()* member function since the last reset. If the reset occurs while the timer is running it counts as the first interval.

Creation

Real_timer t; state is *stopped*.

Operations

<i>void</i>	<i>t.start()</i>	<i>Precondition:</i> state is <i>stopped</i> .
<i>void</i>	<i>t.stop()</i>	<i>Precondition:</i> state is <i>running</i> .
<i>void</i>	<i>t.reset()</i>	reset timer to zero. The state is unaffected.
<i>bool</i>	<i>t.is_running()</i>	<i>true</i> if the current state is running.
<i>double</i>	<i>t.time()</i>	real time in seconds, or 0 if the underlying system call failed.
<i>int</i>	<i>t.intervals()</i>	number of start/stop-intervals since the last reset.
<i>double</i>	<i>t.precision()</i>	smallest possible time step in seconds, or -1 if the system call failed.
<i>double</i>	<i>t.max()</i>	maximal representable time in seconds.

Implementation

The timer class is based in the C function *gettimeofday()* on POSIX systems, the C function *_ftime()* on MS Visual C++, the C function *ftime()* on Borland C++, and *time()* on Metrowerks Codewarrior. The system calls to these timers might fail, in which case a warning message will be issued through the CGAL error handler and the functions return with the error codes indicated above. The *precision* method computes the precision dynamically at runtime at its first invocation.

CGAL::Timer

Definition

```
#include <CGAL/Timer.h>
```

The class *Timer* is a timer class for measuring user process time. A timer *t* of type *Timer* is an object with a state. It is either *running* or it is *stopped*. The state is controlled with *t.start()* and *t.stop()*. The timer counts the time elapsed since its creation or last reset. It counts only the time where it is in the running state. The time information is given in seconds. The timer counts also the number of intervals it was running, i.e. it counts the number of calls of the *start()* member function since the last reset. If the reset occurs while the timer is running it counts as the first interval.

Creation

Timer t; state is *stopped*.

Operations

<i>void</i>	<i>t.start()</i>	<i>Precondition:</i> state is <i>stopped</i> .
<i>void</i>	<i>t.stop()</i>	<i>Precondition:</i> state is <i>running</i> .
<i>void</i>	<i>t.reset()</i>	reset timer to zero. The state is unaffected.
<i>bool</i>	<i>t.is_running()</i>	<i>true</i> if the current state is running.
<i>double</i>	<i>t.time()</i>	user process time in seconds, or 0 if the underlying system call failed.
<i>int</i>	<i>t.intervals()</i>	number of start/stop-intervals since the last reset.
<i>double</i>	<i>t.precision()</i>	smallest possible time step in seconds, or -1 if the system call failed.
<i>double</i>	<i>t.max()</i>	maximal representable time in seconds.

Implementation

The timer class is based in the C function *std::clock()* on PC systems and the C function *getrusage()* on standard POSIX systems. The counter for the *std::clock()* based solution might wrap around (overflow) after only about 36 minutes. This won't happen on POSIX systems. The system calls to these timers might fail, in which case a warning message will be issued through the CGAL error handler and the functions return with the error codes indicated above. The *precision* method computes the precision dynamically at runtime at its first invocation.

CGAL::Union_find<T,A>

Definition

An instance P of the data type $Union_find<T,A>$ is a partition of values of type T into disjoint sets. The template parameter A has to be a model of the allocator concept as defined in the C++ standard. It has a default argument $CGAL_ALLOCATOR(T)$.

Types

$Union_find<T,A>::value_type$ values stored in items (equal to T).

$Union_find<T,A>::handle$ handle to values.

$Union_find<T,A>::iterator$ iterator over values.

There are also constant versions $const_handle$ and $const_iterator$.

$Union_find<T,A>::allocator$ allocator.

Creation

$Union_find<T,A> P;$ creates an instance P of type $Union_find<T,A>$ and initializes it to the empty partition.

Operations

$allocator$ $P.get_allocator()$ the allocator of P .

$std::size_t$ $P.number_of_sets()$ returns the number of disjoint sets of P .

$std::size_t$ $P.size()$ returns the number of values of P .

$std::size_t$ $P.bytes()$ returns the memory consumed by P .

$std::size_t$ $P.size(const_handle p)$ returns the size of the set containing p .

$void$ $P.clear()$ reinitializes P to an empty partition.

$handle$ $P.make_set(T x)$ creates a new singleton set containing x and returns a handle to it.

$handle$ $P.push_back(T x)$ same as $make_set(x)$.

<i>template</i>	<i><class Forward_iterator></i>	
<i>void</i>	<i>P.insert(Forward_iterator first, Forward_iterator beyond)</i>	insert the range of values referenced by <i>[first,beyond)</i> . <i>Requirement:</i> value type of <i>Forward_iterator</i> is <i>T</i> .
<i>handle</i>	<i>P.find(handle p)</i>	returns a canonical handle of the set that contains <i>p</i> , i.e., <i>P.same_set(p,q)</i> iff <i>P.find(p)</i> and <i>P.find(q)</i> return the same handle. <i>Precondition:</i> <i>p</i> is a handle in <i>P</i> .
<i>const_handle</i>	<i>P.find(const_handle p)</i>	
<i>void</i>	<i>P.unify_sets(handle p, handle q)</i>	unites the sets of partition <i>P</i> containing <i>p</i> and <i>q</i> . <i>Precondition:</i> <i>p</i> and <i>q</i> are in <i>P</i> .
<i>bool</i>	<i>P.same_set(const_handle p, const_handle q)</i>	returns true iff <i>p</i> and <i>q</i> belong to the same set of <i>P</i> . <i>Precondition:</i> <i>p</i> and <i>q</i> are in <i>P</i> .
<i>iterator</i>	<i>P.begin()</i>	returns an iterator pointing to the first value of <i>P</i> .
<i>iterator</i>	<i>P.end()</i>	returns an iterator pointing beyond the last value of <i>P</i> .

Implementation

Union_find $\langle T, A \rangle$ is implemented with union by rank and path compression. The running time for m set operations on n elements is $O(n\alpha(m, n))$ where $\alpha(m, n)$ is the extremely slow growing inverse of Ackermann's function.

CGAL::Unique_hash_map<Key,Data,UniqueHashFunction>

Definition

An instance *map* of the parameterized data type *Unique_hash_map<Key,Data,UniqueHashFunction>* is an injective mapping from the set of keys of type *Key* to the set of variables of type *Data*. New keys can be inserted at any time, however keys cannot be individually deleted.

An object *hash* of the type *UniqueHashFunction* returns a unique integer index *hash(key)* of type *std::size_t* for all objects *key* stored in *map*. The template parameter has as default the *Handle_hash_function* that hashes all types of pointers, handles, iterators, and circulators.

All variables are initialized to *default_data*, a value of type *Data* specified in the definition of *map*.

```
#include <CGAL/Unique_hash_map.h>
```

Types

<i>Unique_hash_map<Key,Data,UniqueHashFunction>::Key</i>	the <i>Key</i> type.
<i>Unique_hash_map<Key,Data,UniqueHashFunction>::Data</i>	the <i>Data</i> type.
<i>Unique_hash_map<Key,Data,UniqueHashFunction>::Hash_function</i>	the unique hash function type.

In compliance with STL, the types *key_type*, *data_type*, and *hasher* are defined as well.

Creation

```
Unique_hash_map<Key,Data,UniqueHashFunction> map( Data default = Data(),
                                                    std::size_t table_size = 1,
                                                    Hash_function fct = Hash_function())
```

creates an injective function *map* from *Key* to the set of unused variables of type *Data*, sets *default_data* to *default*, passes the *table_size* as argument to the internal implementation, and initializes the hash function with *fct*.

```
Unique_hash_map<Key,Data,UniqueHashFunction> map( Key first1,
                                                    Key beyond1,
                                                    Data first2,
                                                    Data default = Data(),
                                                    std::size_t table_size = 1,
                                                    Hash_function fct = Hash_function())
```

creates an injective function *map* from *Key* to the set of unused variables of type *Data*, sets *default_data* to *default*, passes the *table_size* as argument to the internal implementation, initializes the hash function with *fct*, and inserts all keys from the range *[first1,beyond1)*. The data variable for each inserted *key* is initialized with the corresponding value from the range *[first2,first2 + (beyond1-first1))*.

Precondition: The increment operator must be defined for values of type *Key* and for values of type *Data*. *beyond1* must be reachable from *first1* using increments.

Operations

<i>Data</i>	<i>map.default_value()</i>	the current <i>default_value</i> .
<i>Hash_function</i>	<i>map.hash_function()</i>	the current hash function.
<i>bool</i>	<i>map.is_defined(Key key)</i>	returns true if <i>key</i> is defined in <i>map</i> . Note that there can be keys defined that have not been inserted explicitly. Their variables are initialized to <i>default_value</i> .
<i>void</i>	<i>map.clear()</i>	resets <i>map</i> to the injective function <i>map</i> from <i>Key</i> to the set of unused variables of type <i>Data</i> . The <i>default_data</i> remains unchanged.
<i>void</i>	<i>map.clear(Data default)</i>	resets <i>map</i> to the injective function <i>map</i> from <i>Key</i> to the set of unused variables of type <i>Data</i> and sets <i>default_data</i> to <i>default</i> .
<i>Data&</i>	<i>map.operator[](const Key& key)</i>	returns a reference to the variable <i>map(key)</i> . If <i>key</i> has not been inserted into <i>map</i> before, <i>key</i> is inserted and initialized with <i>default_value</i> .
<i>const Data&</i>	<i>map.operator[](const Key& key) const</i>	returns a const reference to the variable <i>map(key)</i> . If <i>key</i> has not been inserted into <i>map</i> before, a const reference to the <i>default_value</i> is returned. However, <i>key</i> is not inserted into <i>map</i> .
<i>Data</i>	<i>map.insert(Key first1, Key beyond1, Data first2)</i>	<p>inserts all keys from the range $[first1, beyond1)$. The data variable for each inserted <i>key</i> is initialized with the corresponding value from the range $[first2, first2 + (beyond1 - first1))$. Returns $first2 + (beyond1 - first1)$.</p> <p><i>Precondition:</i> The increment operator must be defined for values of type <i>Key</i> and for values of type <i>Data</i>. <i>beyond1</i> must be reachable from <i>first1</i> using increments.</p>

See Also

UniqueHashFunction.....page [2784](#)
 CGAL::Handle_hash_function.....page [2775](#)

Implementation

Unique_hash_map is implemented via a chained hashing scheme. Access operations *map[i]* take expected time $O(1)$. The *table_size* parameter passed to chained hashing can be used to avoid unnecessary rehashing when set to the number of expected elements in the map. The design is derived from the STL *hash_map* and the LEDA type *map*. Its specialization on insertion only and unique hash values allow for a more time- and space-efficient implementation, see also [MN00, Chapter 5]. This implementation makes also use of sentinels that lead to defined keys that have not been inserted.

UniqueHashFunction

Definition

UniqueHashFunction is a concept for a hash function with unique hash values. An instance *hash* for a model of the UniqueHashFunction concept is a function object. It maps objects of its domain type *Key* to the integral image type *std::size_t*. The image values have to be unique for all keys in the domain type *Key*.

Refines

STL concept HashFunction.

Types

typedef std::size_t result_type; type of the hash value.

Creation

UniqueHashFunction hash(hash2); copy constructor.

UniqueHashFunction& hash = hash2 assignment.

Operations

std::size_t hash(Key key) returns unique hash value for the *key* value.

Has Models

CGAL::Handle_hash_function page [2775](#)

See Also

CGAL::Unique_hash_map<Key,Data,UniqueHashFunction> page [2782](#)

Chapter 49

IO Streams

Andreas Fabri, Geert-Jan Giezeman, and Lutz Kettner

All classes in the CGAL kernel provide input and output operators for IO streams. The basic task of such an operator is to produce a representation of an object that can be written as a sequence of characters on devices as a console, a file, or a pipe. In CGAL we distinguish between a raw ascii, a raw binary and a pretty printing format.

```
enum Mode { ASCII = 0, BINARY, PRETTY};
```

In *ASCII* mode, objects are written as a set of numbers, e.g. the coordinates of a point or the coefficients of a line, in a machine independent format. In *BINARY* mode, data are written in a binary format, e.g. a double is represented as a sequence of four byte. The format depends on the machine. The mode *PRETTY* serves mainly for debugging as the type of the geometric object is written, as well as the data defining the object. For example for a point at the origin with Cartesian double coordinates, the output would be *PointC2(0.0, 0.0)*. At the moment CGAL does not provide input operations for pretty printed data. By default a stream is in ASCII mode.

CGAL provides the following functions to modify the mode of an IO stream.

```
IO::Mode set_mode( std::ios& s, IO::Mode m)
```

```
IO::Mode set_ascii_mode( std::ios& s)
```

```
IO::Mode set_binary_mode( std::ios& s)
```

```
IO::Mode set_pretty_mode( std::ios& s)
```

The following functions allow to test whether a stream is in a certain mode.

```
IO::Mode get_mode( std::ios& s)
```

```
bool is_ascii( std::ios& s)
```

```
bool is_binary( std::ios& s)
```

```
bool is_pretty( std::ios& s)
```

49.1 Output Operator

CGAL defines output operators for classes that are derived from the class *ostream*. This allows to write to ostream as *cout* or *cerr*, as well as to strstreams and fstreams. The output operator is defined for all classes in the CGAL kernel and for the class *Color* as well. Let *os* be an output stream.

ostream& *ostream& os << Class c* Inserts object *c* in the stream *os*. Returns *os*.

Example

```
#include <CGAL/basic.h>
#include <iostream>
#include <fstream>

#include <CGAL/Cartesian.h>
#include <CGAL/Segment_2.h>

typedef CGAL::Point_2< CGAL::Cartesian<double> >    Point;
typedef CGAL::Segment_2< CGAL::Cartesian<double> >    Segment;

int main()
{
    Point p(0,1), q(2,2);
    Segment s(p,q);

    CGAL::set_pretty_mode(std::cout);
    std::cout << p << std::endl << q << std::endl;

    std::ofstream f("data.txt");
    CGAL::set_binary_mode(f);
    f << s << p ;

    return 1;
}
```

49.2 Input Operator

CGAL defines input operators for classes that are derived from the class *istream*. This allows to read from istreams as *cin*, as well as from strstreams and fstreams. The input operator is defined for all classes in the CGAL kernel. Let *is* be an input stream.

istream& *istream& is >> Class c* Extracts object *c* from the stream *is*. Returns *is*.

Example

```
#include <CGAL/basic.h>
#include <iostream>
```

```

#include <fstream>

#include <CGAL/Cartesian.h>
#include <CGAL/Segment_2.h>

typedef CGAL::Point_2< CGAL::Cartesian<double> >    Point;
typedef CGAL::Segment_2< CGAL::Cartesian<double> >    Segment;

int
main()
{
    Point p, q;
    Segment s;

    CGAL::set_ascii_mode(std::cin);
    std::cin >> p >> q;

    std::ifstream f("data.txt");
    CGAL::set_binary_mode(f);
    f >> s >> p;

    return 1;
}

```

49.3 Stream Support

Three classes are provided by CGAL as adaptors to input and output stream iterators. The class *Istream_iterator* is an input iterator adaptor and is particularly useful for classes that are similar but not compatible to *std::istream*. Similarly, the class *Ostream_iterator* is an output iterator adaptor. The class *Verbose_ostream* can be used as an output stream. The stream output operator << is defined for any type. The class stores in an internal state a stream and whether the output is active or not. If the state is active, the stream output operator << uses the internal stream to output its argument. If the state is inactive, nothing happens.

IO Streams

Reference Manual

Andreas Fabri, Geert-Jan Giezeman, and Lutz Kettner

All classes in the CGAL kernel provide input and output operators for IO streams. CGAL provides three different printing mode, defined in the enum *Mode*, as well as different functions to set and get the printing mode.

49.4 Classified Reference Pages

Enum

CGAL::Mode page [2798](#)

Functions

CGAL::get_mode page [2791](#)
CGAL::is_ascii page [2793](#)
CGAL::is_binary page [2794](#)
CGAL::is_pretty page [2795](#)
CGAL::set_mode page [2803](#)
CGAL::set_ascii_mode page [2801](#)
CGAL::set_binary_mode page [2802](#)
CGAL::set_pretty_mode page [2804](#)
CGAL::operator>> page [2792](#)
CGAL::operator<< page [2800](#)

Classes

CGAL::Istream_iterator<T,Stream> page [2796](#)
CGAL::Ostream_iterator<T,Stream> page [2799](#)
CGAL::Verbose_ostream page [2805](#)

49.5 Alphabetical List of Reference Pages

<i>get_mode</i>	page 2791
<i>Istream_iterator</i> < <i>T</i> , <i>Stream</i> >	page 2796
<i>is_ascii</i>	page 2793
<i>is_binary</i>	page 2794
<i>is_pretty</i>	page 2795
<i>Mode</i>	page 2798
<i>operator</i> <<	page 2800
<i>operator</i> >>	page 2792
<i>Ostream_iterator</i> < <i>T</i> , <i>Stream</i> >	page 2799
<i>set_ascii_mode</i>	page 2801
<i>set_binary_mode</i>	page 2802
<i>set_mode</i>	page 2803
<i>set_pretty_mode</i>	page 2804
<i>Verbose_ostream</i>	page 2805

CGAL::get_mode

Mode *get_mode(std::ios& s)*

returns the printing mode of the IO stream *s*.

See Also

CGAL::Mode page [2798](#)
CGAL::set_mode page [2803](#)
CGAL::set_ascii_mode page [2801](#)
CGAL::set_binary_mode page [2802](#)
CGAL::set_pretty_mode page [2804](#)
CGAL::is_ascii page [2793](#)
CGAL::is_binary page [2794](#)
CGAL::is_pretty page [2795](#)

CGAL::operator>>

Definition

CGAL defines input operators for classes that are derived from the class *istream*. This allows to read from istreams as *cin*, as well as from strstreams and fstreams. The input operator is defined for all classes in the CGAL kernel.

istream& *istream& is >> Class c*

Extracts object *c* from the stream *is*. Returns *is*.

See Also

CGAL::set_mode page [2803](#)
CGAL::set_ascii_mode page [2801](#)
CGAL::set_binary_mode page [2802](#)
CGAL::set_pretty_mode page [2804](#)
CGAL::get_mode page [2791](#)
CGAL::is_ascii page [2793](#)
CGAL::is_binary page [2794](#)
CGAL::is_pretty page [2795](#)
CGAL::operator<< page [2800](#)

Example

```
#include <CGAL/basic.h>
#include <iostream>
#include <fstream>

#include <CGAL/Cartesian.h>
#include <CGAL/Segment_2.h>

typedef CGAL::Point_2< CGAL::Cartesian<double> >   Point;
typedef CGAL::Segment_2< CGAL::Cartesian<double> >   Segment;

int
main()
{
    Point p, q;
    Segment s;

    CGAL::set_ascii_mode(std::cin);
    std::cin >> p >> q;

    std::ifstream f("data.txt");
    CGAL::set_binary_mode(f);
    f >> s >> p;

    return 1;
}
```


CGAL::is_ascii

bool *is_ascii(std::ios& s)* checks if the IO stream *s* is in *ASCII* mode.

See Also

CGAL::Mode page [2798](#)
CGAL::set_mode page [2803](#)
CGAL::set_ascii_mode page [2801](#)
CGAL::set_binary_mode page [2802](#)
CGAL::set_pretty_mode page [2804](#)
CGAL::get_mode page [2791](#)
CGAL::is_binary page [2794](#)
CGAL::is_pretty page [2795](#)

CGAL::is_binary

bool *is_binary(std::ios& s)* checks if the IO stream *s* is in *BINARY* mode.

See Also

CGAL::Mode page [2798](#)
CGAL::set_mode page [2803](#)
CGAL::set_ascii_mode page [2801](#)
CGAL::set_binary_mode page [2802](#)
CGAL::set_pretty_mode page [2804](#)
CGAL::get_mode page [2791](#)
CGAL::is_ascii page [2793](#)
CGAL::is_pretty page [2795](#)

CGAL::is_pretty

bool *is_pretty(std::ios& s)* checks if the IO stream *s* is in *PRETTY* mode.

See Also

CGAL::Mode page [2798](#)
CGAL::set_mode page [2803](#)
CGAL::set_ascii_mode page [2801](#)
CGAL::set_binary_mode page [2802](#)
CGAL::set_pretty_mode page [2804](#)
CGAL::get_mode page [2791](#)
CGAL::is_ascii page [2793](#)
CGAL::is_binary page [2794](#)

CGAL::Istream_iterator<T,Stream>

Definition

The class *Istream_iterator*<*T*,*Stream*> is an input iterator adaptor for the input stream class *Stream* and value type *T*. It is particularly useful for classes that are similar but not compatible to *std::istream*.

```
#include <CGAL/IO/Istream_iterator.h>
```

Creation

Istream_iterator<*T*,*Stream*> *i*; creates an end-of-stream iterator *i*. This is a past-the-end iterator, and it is useful when constructing a range.

Istream_iterator<*T*,*Stream*> *i*(*Stream*& *s*); creates an input iterator *i* reading from *s*. When *s* reaches end of stream, this iterator will compare equal to an end-of-stream iterator created using the default constructor.

Operations

i fulfills the requirements for an input iterator.

Example

The following program reads points from a *Window_stream* until the right mouse button gets clicked.

```
// Copyright (c) 2001, 2003 Utrecht University (The Netherlands),
// ETH Zurich (Switzerland), Freie Universitaet Berlin (Germany),
// INRIA Sophia-Antipolis (France), Martin-Luther-University Halle-Wittenberg
// (Germany), Max-Planck-Institute Saarbruecken (Germany), RISC Linz (Austria),
// and Tel-Aviv University (Israel). All rights reserved.
//
// This file is part of CGAL (www.cgal.org); you can redistribute it and/or
// modify it under the terms of the GNU Lesser General Public License as
// published by the Free Software Foundation; version 2.1 of the License.
// See the file LICENSE.LGPL distributed with CGAL.
//
// Licensees holding a valid commercial license may use this file in
// accordance with the commercial license agreement provided with the software.
//
// This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
// WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
//
// $URL: svn+ssh://scm.gforge.inria.fr/svn/cgal/branches/CGAL-3.2-branch/Stream_support/demo/Stream_support/
// $Id: Istream_iterator.C 29807 2006-03-29 14:31:13Z fcacciola $
//
//
// Author(s) : Sylvain Pion <Sylvain.Pion@sophia.inria.fr>
```

```

#include <CGAL/basic.h>

#ifdef CGAL_USE_LEDA
#include <iostream>
int main(){ std::cout << "This demo needs LEDA" << std::endl; return 0;}
#else

#include <CGAL/Cartesian.h>
#include <CGAL/IO/Istream_iterator.h>
#include <CGAL/IO/Window_stream.h>
#include <iostream>
#include <algorithm>

typedef CGAL::Cartesian<double>::Point_2          Point;
typedef CGAL::Istream_iterator<Point, CGAL::Window_stream> Iterator;

#ifdef CGAL_USE_CGAL_WINDOW
#define leda_window CGAL::window
#define leda_green  CGAL::green
#endif

void init_window( leda_window& W) {
    CGAL::cgalize( W);
    W.set_fg_color( leda_green);
    W.display();
    W.init(-1.0, 1.0, -1.0);
}

int main () {
    CGAL::Window_stream window( 512, 512);
    init_window(window);
    std::copy( Iterator(window), Iterator(),
              std::ostream_iterator<Point>(std::cout, "\n"));
    return 0;
}

#endif

```

CGAL::Mode

Definition

All classes in the CGAL kernel provide input and output operators for IOStreams. The basic task of such an operator is to produce a representation of an object that can be written as a sequence of characters on devices as a console, a file, or a pipe. The enum *Mode* distinguish between three different printing formats.

In *ASCII* mode, numbers e.g. the coordinates of a point or the coefficients of a line, are written in a machine independent format. In *BINARY* mode, data are written in a binary format, e.g. a double is represented as a sequence of four byte. The format depends on the machine. The mode *PRETTY* serves mainly for debugging as the type of the geometric object is written, as well as the data defining the object. For example for a point at the origin with Cartesian double coordinates, the output would be *PointC2(0.0, 0.0)*. At the moment CGAL does not provide input operations for pretty printed data. By default a stream is in *ASCII* mode.

```
enum Mode { ASCII = 0, BINARY, PRETTY};
```

See Also

<i>CGAL::set_mode</i>	page 2803
<i>CGAL::set_ascii_mode</i>	page 2801
<i>CGAL::set_binary_mode</i>	page 2802
<i>CGAL::set_pretty_mode</i>	page 2804
<i>CGAL::get_mode</i>	page 2791
<i>CGAL::is_ascii</i>	page 2793
<i>CGAL::is_binary</i>	page 2794
<i>CGAL::is_pretty</i>	page 2795

CGAL::Ostream_iterator<T,Stream>

Definition

The class *Ostream_iterator*<*T*,*Stream*> is an output iterator adaptor for the output stream class *Stream* and value type *T*.

```
#include <CGAL/IO/Ostream_iterator.h>
```

Creation

Ostream_iterator<*T*,*Stream*> *o*(*Stream*& *s*); creates an output iterator *o* writing to *s*.

Operations

o fulfills the requirements for an output iterator.

Implementation

The *operator**() in class *Ostream_iterator*<*T*,*Stream*> uses a proxy class.

CGAL::operator<<

Definition

CGAL defines output operators for classes that are derived from the class *ostream*. This allows to write to ostreams as *cout* or *cerr*, as well as to strstreams and fstreams. The output operator is defined for all classes in the CGAL kernel and for the class *Color* as well. Let *os* be an output stream.

```
ostream&          ostream& os >> Class c
```

Inserts object *c* in the stream *os*. Returns *os*.

See Also

CGAL::set_mode page [2803](#)
CGAL::set_ascii_mode page [2801](#)
CGAL::set_binary_mode page [2802](#)
CGAL::set_pretty_mode page [2804](#)
CGAL::get_mode page [2791](#)
CGAL::is_ascii page [2793](#)
CGAL::is_binary page [2794](#)
CGAL::is_pretty page [2795](#)
CGAL::operator>> page [2792](#)

Example

```
#include <CGAL/basic.h>
#include <iostream>
#include <fstream>

#include <CGAL/Cartesian.h>
#include <CGAL/Segment_2.h>

typedef CGAL::Point_2< CGAL::Cartesian<double> >    Point;
typedef CGAL::Segment_2< CGAL::Cartesian<double> >    Segment;

int main()
{
    Point p(0,1), q(2,2);
    Segment s(p,q);

    CGAL::set_pretty_mode(std::cout);
    std::cout << p << std::endl << q << std::endl;

    std::ofstream f("data.txt");
    CGAL::set_binary_mode(f);
    f << s << p ;

    return 1;
}
```


CGAL::set_ascii_mode

Mode *set_ascii_mode(std::ios& s)*

sets the mode of the IO stream *s* to be the *ASCII* mode. Returns the previous mode of *s*.

See Also

CGAL::Mode page [2798](#)
CGAL::set_mode page [2803](#)
CGAL::set_binary_mode page [2802](#)
CGAL::set_pretty_mode page [2804](#)
CGAL::get_mode page [2791](#)
CGAL::is_ascii page [2793](#)
CGAL::is_binary page [2794](#)
CGAL::is_pretty page [2795](#)

CGAL::set_binary_mode

Mode *set_binary_mode(std::ios& s)*

sets the mode of the IO stream *s* to be the *BINARY* mode. Returns the previous mode of *s*.

See Also

CGAL::Mode page [2798](#)
CGAL::set_mode page [2803](#)
CGAL::set_ascii_mode page [2801](#)
CGAL::set_pretty_mode page [2804](#)
CGAL::get_mode page [2791](#)
CGAL::is_ascii page [2793](#)
CGAL::is_binary page [2794](#)
CGAL::is_pretty page [2795](#)

CGAL::set_mode

Mode *set_mode(std::ios& s, IO::Mode m)*

sets the printing mode of the IO stream *s*.

See Also

CGAL::Mode page [2798](#)
CGAL::set_ascii_mode page [2801](#)
CGAL::set_binary_mode page [2802](#)
CGAL::set_pretty_mode page [2804](#)
CGAL::get_mode page [2791](#)
CGAL::is_ascii page [2793](#)
CGAL::is_binary page [2794](#)
CGAL::is_pretty page [2795](#)

CGAL::set_pretty_mode

Mode *set_pretty_mode(std::ios& s)*

sets the mode of the IO stream *s* to be the *PRETTY* mode. Returns the previous mode of *s*.

See Also

CGAL::Mode page [2798](#)
CGAL::set_mode page [2803](#)
CGAL::set_ascii_mode page [2801](#)
CGAL::set_binary_mode page [2802](#)
CGAL::get_mode page [2791](#)
CGAL::is_ascii page [2793](#)
CGAL::is_binary page [2794](#)
CGAL::is_pretty page [2795](#)

CGAL::Verbose_ostream

Definition

The class *Verbose_ostream* can be used as an output stream. The stream output operator << is defined for any type. The class *Verbose_ostream* stores in an internal state a stream and whether the output is active or not. If the state is active, the stream output operator << uses the internal stream to output its argument. If the state is inactive, nothing happens.

```
#include <CGAL/IO/Verbose_ostream.h>
```

Creation

```
Verbose_ostream verr( bool active = false, std::ostream& out = std::cerr);
```

creates an output stream with state set to *active* that writes to the stream *out*.

Operations

```
template < class T >
Verbose_ostream&      verr << T t
```

Example

The class *Verbose_ostream* can be conveniently used to implement for example the *is_valid()* member function for triangulations or other complex data structures.

```
bool is_valid( bool verbose = false, int level = 0) {
    Verbose_ostream verr( verbose);
    verr << "Triangulation::is_valid( level = " << level << ' ' << endl;
    verr << "    Number of vertices = " << size_of_vertices() << endl;
    // ...
}
```


Chapter 50

IO Streams Colors

50.1 Colors

An object of the class *CGAL::Color* is a color available for drawing operations in CGAL output streams. Each color is defined by a triple of integers (r, g, b) with $0 \leq r, g, b \leq 255$, the so-called *rgb-value* of the color. There are 11 predefined *Color* constants available: *BLACK*, *WHITE*, *GRAY*, *RED*, *GREEN*, *DEEPBLUE*, *BLUE*, *PURPLE*, *VIOLET*, *ORANGE*, and *YELLOW*.

IO Streams Colors Reference Manual

50.2 Colors

An object of the class *CGAL::Color* is a color available for drawing operations in CGAL output streams. Each color is defined by a triple of integers (r, g, b) with $0 \leq r, g, b \leq 255$, the so-called *rgb-value* of the color. There are 11 predefined *Color* constants available: *BLACK*, *WHITE*, *GRAY*, *RED*, *GREEN*, *DEEPBLUE*, *BLUE*, *PURPLE*, *VIOLET*, *ORANGE*, and *YELLOW*.

Chapter 51

Geomview

Andreas Fabri and Sylvain Pion

51.1 Definition

This chapter presents the CGAL interface to Geomview¹, which is a viewer for three-dimensional objects, originally developed at the Geometry Center in Minneapolis².

Geomview 1.8.1 is required.

Note: In releases up to and including 2.2, CGAL used to have the following requirement : the last line in the startup file *.geomview* must be (*echo "started"*). This is no longer necessary.

An object of the class *Geomview_stream* is a stream in which geometric objects can be inserted and where geometric objects can be extracted from. The constructor starts Geomview either on the local either on a remote machine.

Not all but most classes of the CGAL kernel have output operators for the *Geomview_stream*. 2D objects are embedded in the *xy*-plane. Input is only provided for points. Polyhedron and 2D and 3D triangulations have output operators for the *Geomview_stream*.

51.2 Implementation

The constructor forks a process and establishes two pipes between the processes. The forked process is then overlaid with Geomview. The file descriptors *stdin* and *stdout* of Geomview are hooked on the two pipes.

All insert operators construct expressions in *gcl*, the Geomview command language, which is a subset of LISP. These expressions are sent to Geomview via the pipe. The extract operators notify *interest* for a certain kind of events. When such an event happens Geomview sends a description of the event in *gcl* and the extract operator has to parse this expression.

In order to implement further insert and extract operators you should take a look at the implementation and at the Geomview manual.

¹<http://www.geomview.org/>

²<http://www.geom.umn.edu/>

51.3 Example

The following program outputs successively a 2D Delaunay triangulation (projected), a 3D Delaunay, and a terrain from the set of points.

```
// Copyright (c) 2000, 2001, 2004 Utrecht University (The Netherlands),
// ETH Zurich (Switzerland), Freie Universitaet Berlin (Germany),
// INRIA Sophia-Antipolis (France), Martin-Luther-University Halle-Wittenberg
// (Germany), Max-Planck-Institute Saarbruecken (Germany), RISC Linz (Austria),
// and Tel-Aviv University (Israel). All rights reserved.
//
// This file is part of CGAL (www.cgal.org); you can redistribute it and/or
// modify it under the terms of the GNU Lesser General Public License as
// published by the Free Software Foundation; version 2.1 of the License.
// See the file LICENSE.LGPL distributed with CGAL.
//
// Licensees holding a valid commercial license may use this file in
// accordance with the commercial license agreement provided with the software.
//
// This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
// WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
//
// $URL: svn+ssh://scm.gforge.inria.fr/svn/cgal/branches/CGAL-3.2-branch/Geomview/demo/Geomview/terrain.C
// $Id: terrain.C 28567 2006-02-16 14:30:13Z lsaboret $
//
//
// Author(s)      : Sylvain Pion

#include <CGAL/Cartesian.h>
#include <iostream>

#ifndef CGAL_USE_GEOMVIEW
int main()
{
    std::cout << "Geomview doesn't work on Windows, so..." << std::endl;
    return 0;
}
#else

#include <fstream>
#include <unistd.h> // for sleep()

#include <CGAL/Triangulation_euclidean_traits_xy_3.h>

#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/Delaunay_triangulation_3.h>

#include <CGAL/IO/Geomview_stream.h>
#include <CGAL/IO/Triangulation_geomview_ostream_2.h>
#include <CGAL/IO/Triangulation_geomview_ostream_3.h>

#include <CGAL/intersections.h>
```

```

typedef CGAL::Cartesian<double> K;

typedef K::Point_2 Point2;
typedef CGAL::Triangulation_euclidean_traits_xy_3<K> Gt3;
typedef Gt3::Point Point3;

typedef CGAL::Delaunay_triangulation_2<K> Delaunay;
typedef CGAL::Delaunay_triangulation_2<Gt3> Terrain;

typedef CGAL::Delaunay_triangulation_3<K> Delaunay3d;

int main()
{
    CGAL::Geomview_stream gv(CGAL::Bbox_3(-100, -100, -100, 600, 600, 600));
    gv.set_line_width(4);
    // gv.set_trace(true);
    gv.set_bg_color(CGAL::Color(0, 200, 200));
    // gv.clear();

    Delaunay D;
    Delaunay3d D3d;
    Terrain T;
    std::ifstream iFile("data/points3", std::ios::in);
    Point3 p;

    while ( iFile >> p )
    {
        D.insert( Point2(p.x(), p.y()) );
        D3d.insert( p );
        T.insert( p );
    }

    // use different colors, and put a few sleeps/clear.

    gv << CGAL::BLUE;
    std::cout << "Drawing 2D Delaunay triangulation in wired mode.\n";
    gv.set_wired(true);
    gv << D;

    #if 1 // It's too slow ! Needs to use OFF for that.
        gv << CGAL::RED;
        std::cout << "Drawing its Voronoi diagram.\n";
        gv.set_wired(true);
        D.draw_dual(gv);
    #endif

    sleep(5);
    gv.clear();

    std::cout << "Drawing 2D Delaunay triangulation in non-wired mode.\n";
    gv.set_wired(false);
    gv << D;
    sleep(5);
    gv.clear();
}

```

```

std::cout << "Drawing 3D Delaunay triangulation in wired mode.\n";
gv.set_wired(true);
gv << D3d;
sleep(5);
gv.clear();
std::cout << "Drawing 3D Delaunay triangulation in non-wired mode.\n";
gv.set_wired(false);
gv << D3d;
sleep(5);
gv.clear();

std::cout << "Drawing Terrain in wired mode.\n";
gv.set_wired(true);
gv << T;
sleep(5);
gv.clear();
std::cout << "Drawing Terrain in non-wired mode.\n";
gv.set_wired(false);
gv << T;

std::cout << "Enter a key to finish" << std::endl;
char ch;
std::cin >> ch;

return 0;
}
#endif

```

Geomview

Reference Manual

Andreas Fabri and Sylvain Pion

This chapter presents the CGAL interface to Geomview³, which is a viewer for three-dimensional objects, originally developed at the Geometry Center in Minneapolis⁴.

Geomview 1.8.1 is required.

Classes

CGAL::Geomview_stream page [2816](#)

51.4 Alphabetical List of Reference Pages

Geomview_stream page [2816](#)

³<http://www.geomview.org/>

⁴<http://www.geom.umn.edu/>

CGAL::Geomview_stream

Definition

An object of the class *Geomview_stream* is a stream in which geometric objects can be inserted and where geometric objects can be extracted from. The constructor starts Geomview either on the local either on a remote machine.

```
#include <CGAL/IO/Geomview_stream.h>
```

Creation

```
Geomview_stream gs( Bbox_3 bbox = Bbox_3(0,0,0, 1,1,1),
                   const char *machine = NULL,
                   const char *login = NULL)
```

Introduces a Geomview stream *gs* with a camera that sees the bounding box. The command *geomview* must be in the user's *PATH*. If *machine* and *login* are not *NULL*, Geomview is started on the remote machine using *rsh*.

Operations

Output Operators for CGAL Kernel Classes

At the moment not all classes of the CGAL kernel have output operators. 2D objects are embedded in the *xy*-plane.

```
template <class R>
Geomview_stream&      Geomview_stream& G << Point_2<R> p
```

Inserts the point *p* into the stream *gs*.


```
template <class R>
Geomview_stream& G << Point_3<R> p
```

Inserts the point p into the stream gs .

```
template <class R>
Geomview_stream& G << Segment_2<R> s
```

Inserts the segment s into the stream gs .

```
template <class R>
Geomview_stream& G << Segment_3<R> s
```

Inserts the segment s into the stream gs .

```
template <class R>
Geomview_stream& G << Ray_2<R> r
```

Inserts the ray r into the stream gs .

```
template <class R>
Geomview_stream& G << Ray_3<R> r
```

Inserts the ray r into the stream gs .

```
template <class R>
Geomview_stream& G << Line_2<R> l
```

Inserts the line l into the stream gs .

```
template <class R>
Geomview_stream& G << Line_3<R> l
```

Inserts the line l into the stream gs .

```
template <class R>
Geomview_stream& G << Triangle_2<R> t
```

Inserts the triangle t into the stream gs .

```
template <class R>
Geomview_stream& G << Triangle_3<R> t
```

Inserts the triangle t into the stream gs .

```
template <class R>
```

```
Geomview_stream&      Geomview_stream& G << Tetrahedron_3<R> t
```

Inserts the tetrahedron t into the stream gs .

```
template <class R>
Geomview_stream&      Geomview_stream& G << Sphere_3<R> s
```

Inserts the sphere s into the stream gs .

```
Geomview_stream&      Geomview_stream& G << Bbox_2 b
```

Inserts the bounding box b into the stream gs .

```
Geomview_stream&      Geomview_stream& G << Bbox_3 b
```

Inserts the bounding box b into the stream gs .

```
template < class InputIterator >
void      gs.draw_triangles( InputIterator begin, InputIterator end)
```

$[begin;end)$ is an iterator range with value type $Triangle_3<R>$. This method uses the OFF format to draw several triangles at once, which is much faster than drawing them one by one.

Input Operators for CGAL Kernel Classes

At the moment input is only provided for points. The user has to select a point on the *pick plane* with the right mouse button. The pick plane can be moved anywhere with the left mouse button, before a point is entered.

```
template <class R>
Geomview_stream&      Geomview_stream& G >> Point_3<R>& p
```

Extracts the point p from the stream gs . The point is echoed by default, and it depends on the stream echo mode status.

Output Operators for CGAL Basic Library Classes

```
#include <CGAL/IO/Polyhedron_geomview_ostream.h>
```

```
template <class Traits, class HDS>
Geomview_stream&      Geomview_stream &G << Polyhedron_3<Traits,HDS> P
```

Inserts the polyhedron P into the stream gs .

```
#include <CGAL/IO/Triangulation_geomview_ostream.2.h>
```

```
template <class GT, class TDS>
```

```
Geomview_stream&      Geomview_stream &G << Triangulation_2<GT,TDS> T
```

Inserts the 2D triangulation T into the stream gs . The actual output depends on whether the stream is in wired mode or not. Also note that in the case of terrains (when $GT::Point_2$ is $Point_3<R>$), then the 3D terrain is displayed.

```
#include <CGAL/IO/Triangulation_geomview_ostream_3.h>
```

```
template <class GT, class TDS>
```

```
Geomview_stream&      Geomview_stream &G << Triangulation_3<GT,TDS> T
```

Inserts the 3D triangulation T into the stream gs . The actual output depends on whether the stream is in wired mode or not.

Colors

Geomview distinguishes between edge and face colors. The edge color is at the same time the color of vertices.

```
Geomview_stream&      gs << Color c
```

Makes c the color of vertices, edges and faces in subsequent IO operations.

```
Color                  gs.set_bg_color( Color c)
```

Changes the background color. Returns the old value.

```
Color                  gs.set_vertex_color( Color c)
```

Changes the vertex color. Returns the old value.

```
Color                  gs.set_edge_color( Color c)
```

Changes the edge color. Returns the old value.

```
Color                  gs.set_face_color( Color c)
```

Changes the face color. Returns the old value.

Miscellaneous

```
void                  gs.clear()
```

Deletes all objects.

```
void                  gs.pickplane()
```

Creates a pickplane (useful after a clear).

```
void                  gs.look_recenter()
```

Positions the camera in a way that all objects can be seen.

<i>int</i>	<i>gs.get_line_width()</i>	Returns the line width.
<i>int</i>	<i>gs.set_line_width(int w)</i>	Sets the line width to <i>w</i> . Returns the previous value.
<i>double</i>	<i>gs.get_vertex_radius()</i>	Returns the radius of vertices.
<i>double</i>	<i>gs.set_vertex_radius(double r)</i>	Sets the radius of vertices to <i>d</i> . Returns the previous value.
<i>string</i>	<i>gs.get_new_id(string s)</i>	Used to obtain unique identifier names passed to Geomview. On successive calls with the same <i>s</i> value, it will return a string which is <i>s</i> appended with the numbers 0, then 1, then 2... Note that all counters are reset when <i>clear()</i> is called.
<i>bool</i>	<i>gs.set_wired(bool b)</i>	Sets wired mode. In wired mode, some structures output only there edges, not there surfaces. Returns the previous value. By default, wired mode is off.
<i>bool</i>	<i>gs.get_wired()</i>	Returns <i>true</i> iff wired mode is on.

— advanced —

Advanced and Developers Features

The following functions are helpful if you develop your own insert and extract functions. The following functions allow to pass a string from Geomview and to read data sent back by Geomview.

<i>Geomview_stream&</i>	<i>gs << string s</i>	Inserts string <i>s</i> into the stream.
<i>Geomview_stream&</i>	<i>gs >> char* s</i>	Extracts a string <i>s</i> from the stream. <i>Precondition:</i> You have to allocate enough memory.
<i>Geomview_stream&</i>	<i>gs << int i</i>	Inserts <i>i</i> into the stream. Puts whitespace around if the stream is in ascii mode.
<i>Geomview_stream&</i>	<i>gs << unsigned int i</i>	Inserts <i>i</i> into the stream. Puts whitespace around if the stream is in ascii mode.
<i>Geomview_stream&</i>	<i>gs << long i</i>	Inserts <i>i</i> into the stream. Puts whitespace around if the stream is in ascii mode. Currently implemented by converting to int, so it can be truncated on 64 bit platforms.

<i>Geomview_stream</i>	<i>gs << unsigned long i</i>	Inserts <i>i</i> into the stream. Puts whitespace around if the stream is in ascii mode. Currently implemented by converting to unsigned int, so it can be truncated on 64 bit platforms.
<i>Geomview_stream</i>	<i>gs << double d</i>	Inserts double <i>d</i> into the stream. Puts whitespace around if the stream is in ascii mode.
<i>bool</i>	<i>gs.set_trace(bool b)</i>	Sets tracing on. The data that are sent to <i>Geomview</i> are also sent to <i>cerr</i> . Returns the previous value. By default tracing is off.
<i>bool</i>	<i>gs.get_trace()</i>	Returns <i>true</i> iff tracing is on.
<i>bool</i>	<i>gs.set_raw(bool b)</i>	Sets raw mode. In raw mode, kernel points are output without headers and footers, just the coordinates (in binary or ascii mode). This allows the implementation of the stream functions for other objects to re-use the code for points internally, by temporary saving the raw mode to true, and restoring it after. Returns the previous value. By default, raw mode is off.
<i>bool</i>	<i>gs.get_raw()</i>	Returns <i>true</i> iff raw mode is on.
<i>bool</i>	<i>gs.set_echo(bool b)</i>	Sets echo mode. In echo mode, when you select a point in Geomview, the point is actually sent back to Geomview. Returns the previous value. By default, echo mode is on.
<i>bool</i>	<i>gs.get_echo()</i>	Returns <i>true</i> iff echo mode is on.
<i>bool</i>	<i>gs.set_binary_mode(bool b = true)</i>	Sets whether we are in binary mode.
<i>bool</i>	<i>gs.set_ascii_mode(bool b = true)</i>	Sets whether we are in ascii mode.
<i>bool</i>	<i>gs.get_binary_mode()</i>	Returns <i>true</i> iff <i>gs</i> is in binary mode.
<i>bool</i>	<i>gs.get_ascii_mode()</i>	Returns <i>true</i> iff <i>gs</i> is in ascii mode.

_____ *advanced* _____

Implementation

The constructor forks a process and establishes two pipes between the processes. The forked process is then overlaid with Geomview. The file descriptors *stdin* and *stdout* of Geomview are hooked on the two pipes.

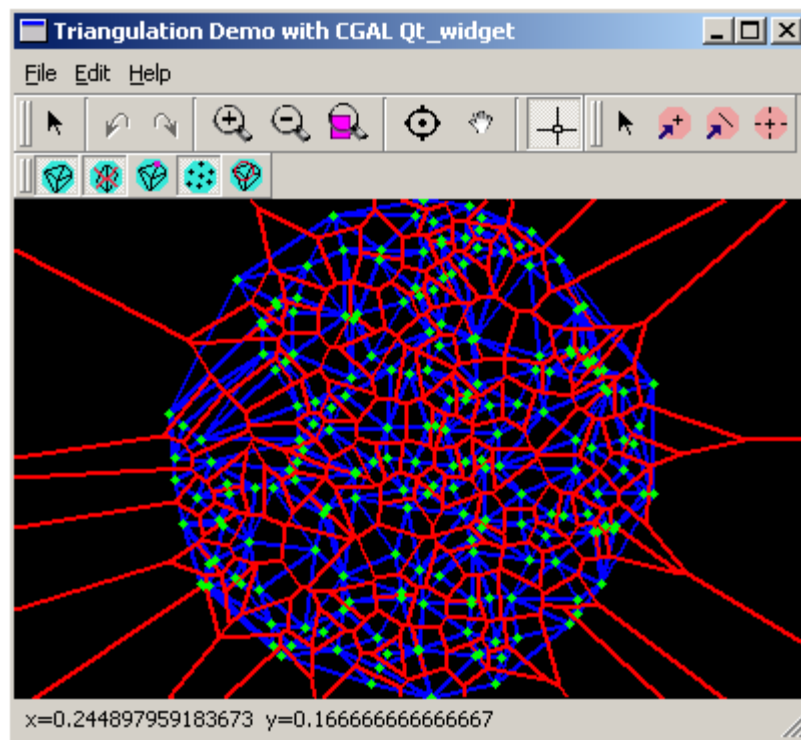
All insert operators construct expressions in *gcl*, the Geomview command language, which is a subset of LISP. These expressions are sent to Geomview via the pipe. The extract operators notify *interest* for a certain kind of events. When such an event happens Geomview sends a description of the event in *gcl* and the extract operator has to parse this expression.

In order to implement further insert and extract operators you should take a look at the implementation and at the Geomview manual.

Chapter 52

Qt_widget

Laurent Rineau and Radu Ursu



Qt is a GUI toolkit¹ for cross-platform application development.

52.1 Introduction

This chapter describes the *Qt_widget* package which provides an interface between CGAL and the GUI toolkit *Qt*. The *Qt_widget* package allows to build *Qt* applications showing two dimensional CGAL objects and algorithms.

¹<http://www.trolltech.com>

The atom of the *Qt* user interface is called a widget. A widget receives mouse, keyboard and other events from the window system, and paints a representation of itself on the screen. Widgets are rectangular, and the different widgets of an application are sorted in a Z-order. Widgets can have a parent widget and children. A widget is clipped by its parent and by the widgets in front of it.

The most important class in the package is the class *Qt_widget* which implements a widget providing a drawing area and output stream operators for CGAL two dimensional objects. *Qt_widget* also provides zooming and panning functionalities.

The *Qt_widget* allows to attach *layers*. Layers usually draw on the drawing area of the widget. Layers can be activated and deactivated, and what you see in the drawing area is the overlay of all attached activated layers. Layers can also be used for entering input, and CGAL provides input *layers* for the two-dimensional CGAL objects.

The package includes also the class *Qt_widget_standard_toolbar* providing a standard toolbar for controlling the basic functionality of the *Qt_widget*.

The following sections describe the main class as well as the helper classes in more detail and give examples that can be taken as starting points for new applications.

Remark: The *Qt_widget* is distributed under the QPL, which is Trolltech's open source license. For more details on the QPL see <http://www.trolltech.com/developer/licensing/qpl.html>.

52.2 Qt_widget

The class *Qt_widget* is derived from the class *QWidget* which is the base class of all *Qt* user interface objects.

The *Qt_widget* provides output operators for two dimensional CGAL objects. There are operators defined for output of: points, segments, lines, rays, circles, triangles, rectangles, polygons, conics, and all type of triangulations. Also some operators are defined to set *Qt_widget*'s properties, like background and fill color, as well as line width and point size.

As the following examples show, simple applications can be written without the layers.

52.2.1 Example: Hello Segment

The first example draws a red segment on an orange background.

```
// Copyright (c) 1997-2004 Utrecht University (The Netherlands),  
// ETH Zurich (Switzerland), Freie Universitaet Berlin (Germany),  
// INRIA Sophia-Antipolis (France), Martin-Luther-University Halle-Wittenberg  
// (Germany), Max-Planck-Institute Saarbruecken (Germany), RISC Linz (Austria),  
// and Tel-Aviv University (Israel). All rights reserved.  
//  
// This file is part of CGAL (www.cgal.org); you can redistribute it and/or  
// modify it under the terms of the GNU Lesser General Public License as  
// published by the Free Software Foundation; version 2.1 of the License.  
// See the file LICENSE.LGPL distributed with CGAL.  
//  
// Licensees holding a valid commercial license may use this file in
```



```

// accordance with the commercial license agreement provided with the software.
//
// This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
// WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
//
// $URL: svn+ssh://scm.gforge.inria.fr/svn/cgal/branches/CGAL-3.2-branch/Qt_widget/demo/Qt_widget/helloseg
// $Id: hellosegment.C 30999 2006-05-04 09:15:26Z lsaboret $
//
//
// Author(s)      : Laurent Rineau
//                  Radu Ursu <rursu@sophia.inria.fr>

#include <CGAL/basic.h>

#ifndef CGAL_USE_QT
#include <iostream>
int main(int, char*){
    std::cout << "Sorry, this demo needs QT..." << std::endl; return 0;}
#else
#include <CGAL/Cartesian.h>
#include <CGAL/IO/Qt_widget.h>

#include <qapplication.h>

typedef CGAL::Cartesian<int> Rep;
typedef CGAL::Point_2<Rep> Point_2;
typedef CGAL::Segment_2<Rep> Segment;

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    CGAL::Qt_widget *w = new CGAL::Qt_widget();
    app.setMainWidget( w );
    w->resize(600, 600);
    w->set_window(0, 600, 0, 600);
    w->show();
    w->lock();
    *w << CGAL::BackgroundColor(CGAL::ORANGE) << CGAL::RED;
    *w << Segment(Point_2(100,100), Point_2(400,400));
    w->unlock();
    return app.exec();
}
#endif

```

We follow the *Qt* naming conventions for material properties, for example, the `CGAL::BackgroundColor` above.

All the drawing code should be put between *Qt_Widget*'s `lock()` and `unlock()` functions. See the manual reference pages of *Qt_widget*. Doing like this, the window will be updated only once, when *Qt_widget* finds the last `unlock()`. This way you can avoid the window flickering.

This example has a severe drawback: when you resize the window it is empty, as nothing is redrawn. This style of programs makes only sense, if you quickly want to validate output of a geometric computation. As in any event driven GUI application, *Qt* provides a callback mechanism so that the window system can update the

drawing whenever necessary. This is the topic of the next example.

52.2.2 Example: Signals and Slots

This example is slightly more involved and uses the signal/slots mechanism of *Qt*.

The main widget shows a Delaunay triangulation. Every time the mouse button is pressed over the widget, a point is input and inserted in the Delaunay triangulation. The result of this insertion appears immediately. Furthermore, the drawing is updated every time the window is resized.

```
// Copyright (c) 1997-2004 Utrecht University (The Netherlands),
// ETH Zurich (Switzerland), Freie Universitaet Berlin (Germany),
// INRIA Sophia-Antipolis (France), Martin-Luther-University Halle-Wittenberg
// (Germany), Max-Planck-Institute Saarbruecken (Germany), RISC Linz (Austria),
// and Tel-Aviv University (Israel). All rights reserved.
//
// This file is part of CGAL (www.cgal.org); you can redistribute it and/or
// modify it under the terms of the GNU Lesser General Public License as
// published by the Free Software Foundation; version 2.1 of the License.
// See the file LICENSE.LGPL distributed with CGAL.
//
// Licensees holding a valid commercial license may use this file in
// accordance with the commercial license agreement provided with the software.
//
// This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
// WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
//
// $URL: svn+ssh://scm.gforge.inria.fr/svn/cgal/branches/CGAL-3.2-branch/Qt_widget/demo/Qt_widget/tutorial2.C $
// $Id: tutorial2.C 30999 2006-05-04 09:15:26Z lsaboret $
//
//
// Author(s) : Mariette Yvinec <Mariette.Yvinec@sophia.inria.fr>

#include <CGAL/basic.h>

#ifdef CGAL_USE_QT
#include <iostream>
int main(int, char*){
    std::cout << "Sorry, this demo needs QT..." << std::endl; return 0;}
#else
#include <CGAL/Cartesian.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/IO/Qt_widget_Triangulation_2.h>
#include <CGAL/IO/Qt_widget.h>
#include <qapplication.h>
#include <qmainwindow.h>

typedef CGAL::Cartesian<double> K;
typedef K::Point_2 Point_2;
typedef CGAL::Delaunay_triangulation_2<K> Delaunay;

Delaunay dt;
```

```

class My_window : public QMainWindow {
    Q_OBJECT
public:
    My_window(int x, int y)
    {
        widget = new CGAL::Qt_widget(this);
        widget->resize(x,y);
        widget->set_window(0, x, 0, y);

        connect(widget, SIGNAL(redraw_on_back()),
            this, SLOT(redraw_win()));

        connect(widget, SIGNAL(s_mousePressEvent(QMouseEvent*)),
            this, SLOT(mousePressEvent(QMouseEvent*)));

        setCentralWidget(widget);
    };
private slots:
    void redraw_win()
    {
        *widget << dt;
    }

    void mousePressEvent(QMouseEvent *e)
    {
        dt.insert(Point_2(widget->x_real(e->x()), widget->y_real(e->y())));
        widget->redraw();
    }

private: // private data member
    CGAL::Qt_widget* widget;
};

//moc_source_file : tutorial2.C
#include "tutorial2.moc"

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    My_window *w = new My_window(400,400);
    app.setMainWidget( w);
    w->show();
    return app.exec();
}
#endif

```

Qt applications are event driven and respond to user interaction. For example, when a user clicks on a menu item or on a toolbar button, the application executes some codes. The programmer of an application has to be able to relate events to the relevant code. *Qt* provide for that the signals/slots mechanism:

Signals. Each *Qt* widget declares a set of signals which, using the keyword *emit* can be emitted by member functions under some circumstances. Signals are declared by using the keyword *signals*: just like an access specifier in your class declaration.

Slots. A slot is just a member function declared under a public (or private) slots section.

Connect. Signals and slots can be connected together using the method *connect*. This method needs to know four things: the object that sends out the signal, the signal, the object to which belong the connected slot and the slot connected to the signal. For instance, the statement:

```
connect(widget, SIGNAL(redraw_on_back()),
        this, SLOT(redraw_win()));
```

connects the signal *redraw_on_back()* of the widget *widget* to the slot *redraw_win()* of the *QMainWindow* *his*. Signals and slots can have any type of arguments, but a signal and a slot connected together must have the same arguments types.

Every class that defines at least a signal or a slot must be derived from the class *QObject* and must use the macro *Q_OBJECT* inside the private section of its declaration. You also need to run *moc*, the *Meta Object Compiler* supplied with *Qt* on the file that contains the class declaration.

moc is a pre-compiler that produces the *meta object* code of each class that uses the macro *Q_OBJECT*. This *meta object* code is needed by the signal/slot mechanism. There are several methods to compile the outputs of *moc*. In CGAL, we have chosen to include the outputs of *moc* in a source files. See the *Qt* documentation on *moc* for other possibilities.

The line *//moc_source_file : tutorial2.C* is for users that use makefiles. This line tells to the CGAL makefile generator that *tutorial2.C* should be the file that *moc* should be run on.

Let us come back to the control flow in the above example. The main widget of the application is the widget *w* of class *My_window*. The creator of *My_window* triggers the creation of a *Qt_widget* accessible through the pointer *widget*. When the mouse button is pressed in its drawing area, the *Qt_widget* emits the signal *s_mousePressEvent(QMouseEvent*)* connected to the slot *mousePressEvent(QMouseEvent*)* of *My_window*. This slot inserts the point in the Delaunay triangulation and calls the method *redraw()* of *Qt_widget*. The *redraw()* method of *Qt_widget* emits the signal *redraw_on_back()*. This signal is connected to the slot *redraw_win()* of *My_window* which actually draws the triangulation. Note that the *Qt_widget* emits the same signal *redraw_on_back()* when the window is resized. Thus, the signals/slots connection ensures that the triangulation is redrawn each time the redrawing is needed.

— advanced —

There are several ways to draw something with *Qt_widget*. One way is to use the signals *redraw_on_back()*, *redraw_on_front()*. This way you can bring your drawings before all or after all the other drawings. An other option is to use the *QPainter* instance of *Qt_widget* that you can get calling *get_painter()* method. The most recommended way is to use layers, that are described in the next section.

— advanced —

Note that in that example, the *My_window* constructor calls *new* to allocate a new *Qt_widget* object but *delete* is never called to deallocate it. This does not mean that there is a memory leak. It is in Qt's responsibility to free widgets that have a parent. In that example the *My_window* object is the parent of *widget* and will deallocate it automatically at its destruction.

52.3 Layers

52.3.1 Using Layers to Draw

In the examples from the previous section the code for drawing on the widget was in the `redraw_win()` function. As soon as the applications are more involved it leads to a more modular design if one delegates the drawing task to *layers*. For example, if the application displays a Delaunay triangulation, the corresponding Voronoi diagram, and at the same time highlights the nearest vertex to the mouse coordinates, it makes sense to have three independent layers. Besides better code, layers have the advantage that they can be activated and deactivated at runtime. Finally, more modularity means a higher potential for reuse.

A layer can be *attached* to a *Qt_widget*. The `redraw()` member function of the *Qt_widget* calls the method `Qt_widget_layer::draw()` of all attached layers, in the order that they were attached. It is a very simple rule: the last layer attached will be drawn on top.

Also a layer can be *activated* and *deactivated*. Only active layers are drawn, and by default a layer is activated when it gets attached. Note that deactivating and activating do not influence the order of layers. You can change the order only by attaching and detaching it.

CGAL provides a base class so that users can write their own layers. All the layers have to derive from this base class `Qt_widget_layer` to have the functionality described.

52.3.2 Example: Using a Layer to Draw

Example

```
#include <CGAL/Cartesian.h>
#include <CGAL/Point_2.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/IO/Qt_widget_Delaunay_triangulation_2.h>
#include <CGAL/IO/Qt_widget.h>
#include <CGAL/IO/Qt_widget_layer.h>
#include <qapplication.h>

typedef CGAL::Cartesian<double>          Rep;
typedef CGAL::Point_2<Rep>              Point;
typedef CGAL::Delaunay_triangulation_2<Rep> Delaunay;

Delaunay dt;

class My_layer : public CGAL::Qt_widget_layer{
    void draw(){
        *widget << dt;
    }
};

class My_window : public CGAL::Qt_widget {
public:
    My_window(int x, int y)
    {
```

```

        resize(x,y);
        attach(&layer);
    };
private:
    //this method is called when the user presses the mouse
    void mousePressEvent(QMouseEvent *e)
    {
        Qt_widget::mousePressEvent(e);
        dt.insert(Point(x_real(e->x()), y_real(e->y())));
        redraw();
    }
    My_layer layer;
};

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    My_window *W = new My_window(400,400);
    app.setMainWidget(W);
    W->show();
    W->set_window(0, 400, 0, 400);
    return app.exec();
}

```

This example defines a class derived from *Qt_widget_layer*. In the member function *draw()* is the code for drawing the triangulation. In *My_Window* class you need an instance of *My_Layer* and you will have to attach it, if you want to see what the layer draws on the screen.

As you see, this example is very similar to the previous one, but the code for drawing the triangulation is no longer in the *redraw_win()* function, but in a layer.

52.3.3 Using Layers to Build New Objects

The main purpose of layers is to have more modular code for drawing on the widget. Things are similar for handling input. In the previous examples, input went through the *Qt_widget::mousePressedEvent()* callback, which interpreted the input. In applications where you have different kinds of input, e.g., segments and polygons in an arrangement demo, this quickly leads to unreadable, difficult to maintain code, especially as typically more than one event callback is involved. The proper way of decomposition, is delegation of the event handling to a *layer*.

A layer for *Qt_widget* receives all the events from *Qt_widget* if it is active and can provide some functionality like input objects for *Qt_widget*. Notice that the layers receive events in the order they have been attached. Layers can have internal state, because for entering complex objects it needs several events. Therefore layers must have functions to initialize state when they are activated and to clean up when they are deactivated.

CGAL provides some predefined input layers. You can have a lot of layers attached that could be active in the same time. You have to take care how you manage the events if you do not want to have conflicts. A conflict is when two attached layers that are active need both the same event and getting and using it might not have such a good effect in your application. For example the predefined layers that builds a new line and a new circle produce bad visual effects when are both active.

You can resolve conflicts by using *layers* exclusive. If you have several layers that can not be active at the same

time without creating conflicts, you can resolve that by letting the user not activate more than one of those at a time.

— *advanced* —

In all the CGAL demos that are provided, the layers are used with a toolbar and buttons. To activate and deactivate a layer you have to click one of the toolbar buttons. There are layers that need exclusive use. This is accomplished by grouping the buttons in one group, and making that group exclusive. The group class is *QButtonGroup* that comes with *Qt*.

— *advanced* —

We first show how to use layers and then how they work.

52.3.4 Example: How to Use a Layer

In the previous section, you could insert new points in the triangulation by clicking on the widget. This example shows how the same can be achieved with the help of a layer.

We attach the predefined layer *Qt_widget_get_point* to the widget, and connect the signal emitted by the widget to the function that handles the input. When the user clicks with the left mouse button, the layer creates a point and passes it to the widget. The widget then emits a signal that gets passed to the connected slot *My_Window::get_new_object(CGAL::Object)*.

```
// Copyright (c) 1997-2004 Utrecht University (The Netherlands),
// ETH Zurich (Switzerland), Freie Universitaet Berlin (Germany),
// INRIA Sophia-Antipolis (France), Martin-Luther-University Halle-Wittenberg
// (Germany), Max-Planck-Institute Saarbruecken (Germany), RISC Linz (Austria),
// and Tel-Aviv University (Israel). All rights reserved.
//
// This file is part of CGAL (www.cgal.org); you can redistribute it and/or
// modify it under the terms of the GNU Lesser General Public License as
// published by the Free Software Foundation; version 2.1 of the License.
// See the file LICENSE.LGPL distributed with CGAL.
//
// Licensees holding a valid commercial license may use this file in
// accordance with the commercial license agreement provided with the software.
//
// This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
// WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
//
// $URL: svn+ssh://scm.gforge.inria.fr/svn/cgal/branches/CGAL-3.2-branch/Qt_widget/demo/Qt_widget/layer.C
// $Id: layer.C 30999 2006-05-04 09:15:26Z lsaboret $
//
//
// Author(s) : Radu Ursu <rursu@sophia.inria.fr>

#include <CGAL/basic.h>

#ifndef CGAL_USE_QT
#include <iostream>
int main(int, char*){
    std::cout << "Sorry, this demo needs QT..." << std::endl; return 0;}
}
```

```

#else
#include <CGAL/Cartesian.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/IO/Qt_widget_Delaunay_triangulation_2.h>
#include <CGAL/IO/Qt_widget.h>
#include <CGAL/IO/Qt_widget_layer.h>
#include <CGAL/IO/Qt_widget_get_point.h>

#include <qapplication.h>
#include <qmainwindow.h>

typedef CGAL::Cartesian<double> Rep;
typedef CGAL::Point_2<Rep> Point_2;
typedef CGAL::Delaunay_triangulation_2<Rep> Delaunay;

Delaunay dt;

class My_Layer : public CGAL::Qt_widget_layer{
    void draw(){
        *widget << dt;
    }
};

class My_Window : public QMainWindow {
    Q_OBJECT
public:
    My_Window(int x, int y){
        widget = new CGAL::Qt_widget(this, "CGAL Qt_widget");
        setCentralWidget(widget);
        resize(x,y);
        widget->attach(&get_point);
        widget->attach(&v);
        connect(widget, SIGNAL(new_cgal_object(CGAL::Object)),
                this, SLOT(get_new_object(CGAL::Object)));
        widget->set_window(0, 600, 0, 600);
    };
private: //members
    CGAL::Qt_widget_get_point<Rep> get_point;
    My_Layer v;
    CGAL::Qt_widget *widget;
private slots:
    void get_new_object(CGAL::Object obj)
    {
        Point_2 p;
        if (CGAL::assign(p, obj)) {
            dt.insert(p);
        }
        widget->redraw();
    }
}; //endclass

// moc_source_file : layer.C
#include "layer.moc"

```



```

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    My_Window *W = new My_Window(600,600);
    app.setMainWidget( W );
    W->show();
    return app.exec();
}
#endif

```

The *Qt_widget* forwards all events that it receives to the attached and active layers. If a layer is attached but not active, it does not get the events. It is put in a passive state. Activating and deactivating the layer does not mean that the object is destructed.

52.3.5 Example: How Layers Work

The following is an example of a *layer* that creates CGAL points when the user clicks the left mouse button over the widget.

```

#include <CGAL/IO/Qt_widget.h>
#include <CGAL/IO/Qt_widget_layer.h>
#include <qcursor.h>

#ifndef CGAL_QT_WIDGET_GET_POINT_BUTTON
#define CGAL_QT_WIDGET_GET_POINT_BUTTON Qt::LeftButton
#endif

namespace CGAL {
template <class R>
class Qt_widget_get_point : public Qt_widget_layer
{
public:
    typedef typename R::Point_2    Point;
    typedef typename R::FT          FT;

    Qt_widget_get_point(const QCursor c=QCursor(Qt::crossCursor),
                        QObject* parent = 0, const char* name = 0) :
        Qt_widget_layer(parent, name), cursor(c) {};

private:
    bool is_pure(Qt::ButtonState s){
        if((s & Qt::ControlButton) ||
           (s & Qt::ShiftButton) ||
           (s & Qt::AltButton))
            return 0;
        else
            return 1;
    }
    void mousePressEvent(QMouseEvent *e)
    {
        if(e->button() == CGAL_QT_WIDGET_GET_POINT_BUTTON

```

```

        && is_pure(e->state()))
    {
        FT x, y;
        widget->x_real(e->x(), x);
        widget->y_real(e->y(), y);
        widget->new_object(make_object(Point(x, y)));
    }
};

void activating()
{
    oldcursor = widget->cursor();
    widget->setCursor(cursor);
};

void deactivating()
{
    widget->setCursor(oldcursor);
};

QCursor cursor;
QCursor oldcursor;
};
} // namespace CGAL

```

The *Qt_widget* forwards mouse and keyboard events to the attached layer. In the above example only the *mousePressEvent* member function is overloaded.

Tools that create new CGAL objects, must call the member function *Qt_widget::new_object(CGAL::Object)*. The *Qt_widget* then emits the signal *new_cgal_object(CGAL::Object)*. This signal can be routed to any slot of other object accepting a *CGAL::Object*, with the following connect statement:

```

connect(qt_widget_ptr, SIGNAL(new_cgal_object(CGAL::Object)),
        any_other_object_ptr, SLOT(any_other_slot(CGAL::Object)));

```

The first argument must be a pointer to an instance of *Qt_widget*. In the example we connect it to *MyWindow::get_new_object(CGAL::Object)*.

52.4 The Standard Toolbar

The *Qt_widget* allows to zoom and pan. This functionality is accessible through the class *Qt_widget_standard_toolbar*. The example further down shows how to use it in your application.



The functionality of the toolbar is as follows from the left to right:

History back: Go back into the transformation history

History forth: Go forth into the transformation history

Zoom In: The scaling factor is multiplied by two, keeping the same center.

Zoom Out: The scaling factor is divided by two, keeping the same center.

Point tool: Deactivate the layers corresponding to the three following buttons which form an exclusive group

Focus on Point: Lets you choose the center of the region where you want to focus.

Focus on the Region: The area in the rectangle that you selected will be magnified to best fit in the window.

Hand Tool: Used for translate. Click to select the first point of translation and drag to select the second point.

Mouse Coordinates Layer: Mouse coordinates are displayed on the status bar of your window. You can deactivate this layer if you click on it. To activate it again just click one more time.

Example

```
// Copyright (c) 1997-2004 Utrecht University (The Netherlands),
// ETH Zurich (Switzerland), Freie Universitaet Berlin (Germany),
// INRIA Sophia-Antipolis (France), Martin-Luther-University Halle-Wittenberg
// (Germany), Max-Planck-Institute Saarbruecken (Germany), RISC Linz (Austria),
// and Tel-Aviv University (Israel). All rights reserved.
//
// This file is part of CGAL (www.cgal.org); you can redistribute it and/or
// modify it under the terms of the GNU Lesser General Public License as
// published by the Free Software Foundation; version 2.1 of the License.
// See the file LICENSE.LGPL distributed with CGAL.
//
// Licensees holding a valid commercial license may use this file in
// accordance with the commercial license agreement provided with the software.
//
// This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
// WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
//
// $URL: svn+ssh://scm.gforge.inria.fr/svn/cgal/branches/CGAL-3.2-branch/Qt_widget/demo/Qt_widget/standard
// $Id: standard_toolbar.C 30999 2006-05-04 09:15:26Z lsaboret $
//
//
// Author(s)      : Laurent Rineau
//                  Radu Ursu <rursu@sophia.inria.fr>

#include <CGAL/basic.h>

#ifndef CGAL_USE_QT
#include <iostream>
int main(int, char*){
    std::cout << "Sorry, this demo needs QT..." << std::endl; return 0;}
#else
#include <CGAL/Cartesian.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/IO/Qt_widget/Delaunay_triangulation_2.h>
```

```

#include <qapplication.h>
#include <qmainwindow.h>

#include <CGAL/IO/Qt_widget.h>
#include <CGAL/IO/Qt_widget_standard_toolbar.h>
#include <CGAL/point_generators_2.h>

typedef CGAL::Cartesian<double> Rep;
typedef CGAL::Point_2<Rep> Point_2;
typedef CGAL::Delaunay_triangulation_2<Rep> Delaunay;

Delaunay dt;

class My_window : public QMainWindow{
    Q_OBJECT
public:
    My_window(int x, int y)
    {
        widget = new CGAL::Qt_widget(this);
        setCentralWidget(widget);
        resize(x,y);
        widget->show();
        widget->set_window(0, x, 0, y);

        CGAL::Random_points_in_disc_2<Point_2> g(500);
        for(int count=0; count<100; count++) {
            dt.insert(*g++);
        }

        //How to attach the standard toolbar
        std_toolbar = new CGAL::Qt_widget_standard_toolbar(widget, this,
            "Standard Toolbar");

        connect(widget, SIGNAL(redraw_on_back()),
            this, SLOT(redraw_win()) );
    }

private slots: //functions
    void redraw_win()
    {
        *widget << dt;
    }

private: //members
    CGAL::Qt_widget *widget;
    CGAL::Qt_widget_standard_toolbar *std_toolbar;
};

// moc_source_file: standard_toolbar.C
#include "standard_toolbar.moc"

int main( int argc, char **argv )
{

```

```

    QApplication app( argc, argv );
    My_window W(600,600);
    app.setMainWidget( &W );
    W.show();
    W.setCaption("Using the Standard Toolbar");
    return app.exec();
}
#endif

```

This example generates 100 points and inserts them in a Delaunay triangulation. Using the standard toolbar you can zoom in, zoom out, translate.

52.5 The Help Window

We provide a class in the *Qt_widget* library that was taken from an example of *Qt* and adapted to our needs. This class has the functionality of a rich text browser with hypertext navigation. You can also PRINT, GO BACK, GO FORWARD or GO HOME. This class is called *Qt_help_window* and you can use it to display hypertext support in your application. It is used in a lot of demos provided in the distribution.

Example

```

#include <CGAL/IO/Qt_help_window.h>
....
QString home = "help/index.html";
Qt_help_window *help = new Qt_help_window(home, ".", 0, "help viewer");
help->resize(400, 400);
help->setCaption("Demo HowTo");
help->show();

```

52.6 Some Predefined Icons

CGAL provides some icons defined in some header files. The icons are pixmaps, having the extension *.xpm*. Their location is */include/CGAL/IO/pixmaps*.

To use a pixmap in your code you have to include the right file, and to know the names of the pixmaps. The names of the pixmaps are composed of two parts, the name of the file and the tag *xpm*. So for example the arrow pixmap has the name *arrow_xpm*, the line pixmap has the name *line_xpm*, and so on. There are also pixmaps files that contain small icons. The name of the smaller pixmaps contain a *small* at the middle of it like *point_small_xpm*. In the tutorials and demos, almost all the pixmaps are used for the toolbar buttons, like this:

Example

```

#include <CGAL/IO/pixmaps/point.xpm>

QIconSet set(QPixmap( (const char**)point_small_xpm ),

```

```
QPixmap( (const char**)point_xpm ));

QToolButton *point_button;
point_button = new QToolButton(toolbar_ptr, "POINT INPUT BUTTON");
point_button->setIconSet(set);
point_button->setTextLabel("POINT INPUT LAYER");
```

52.7 What Shall I Use?

The previous sections presented different ways of writing *Qt* based applications. We recommend to use layers for the drawing task and for input handling, even if you write tiny applications, because in general they grow over time. Layers are a little bit more overhead, but it pays off in the long run, as you then do not have to completely reorganize your code, to add layers.

Qt_widget

Reference Manual

Laurent Rineau and Radu Ursu

Qt_widget is the base widget class that CGAL provide as support for 2D CGAL users.

52.8 Classified Reference Pages

Classes

<i>CGAL::Qt_widget</i>	page 2841
<i>CGAL::Qt_widget_layer</i>	page 2851
<i>CGAL::Qt_widget_standard_toolbar</i>	page 2864
<i>CGAL::Qt_widget_history</i>	page 2867
<i>CGAL::Qt_help_window</i>	page 2868
<i>CGAL::Qt_widget_get_point<T></i>	page 2854
<i>CGAL::Qt_widget_get_segment<T></i>	page 2855
<i>CGAL::Qt_widget_get_line<T></i>	page 2856
<i>CGAL::Qt_widget_get_circle<T></i>	page 2857
<i>CGAL::Qt_widget_get_iso_rectangle<T></i>	page 2858
<i>CGAL::Qt_widget_get_polygon<Polygon></i>	page 2859
<i>CGAL::Qt_widget_get_simple_polygon<Polygon></i>	page 2861

52.9 Alphabetical List of Reference Pages

<i>Custom_zoom_layer</i>	page 2863
<i>Navigation_layer</i>	page 2862
<i>Qt_help_window</i>	page 2868
<i>Qt_widget_get_circle<T></i>	page 2857
<i>Qt_widget_get_iso_rectangle<T></i>	page 2858
<i>Qt_widget_get_line<T></i>	page 2856
<i>Qt_widget_get_point<T></i>	page 2854
<i>Qt_widget_get_polygon<Polygon></i>	page 2859
<i>Qt_widget_get_segment<T></i>	page 2855
<i>Qt_widget_get_simple_polygon<Polygon></i>	page 2861

<i>Qt_widget_history</i>	page 2867
<i>Qt_widget_layer</i>	page 2851
<i>Qt_widget_standard_toolbar</i>	page 2864
<i>Qt_widget</i>	page 2841

CGAL::Qt_widget

Definition

An object of type *Qt_widget* is a two-dimensional window for graphical IO. It is a class that is designed to help CGAL users to visualize easily CGAL objects and for advanced users to interact with them. This widget is designed for 2D CGAL objects only.

```
#include <CGAL/IO/Qt_widget.h>
```

Inherits From

QWidget

Types

The widget class also defines the following enum type to specify which kind of point representation do you want to use:

```
enum PointStyle { PIXEL, CROSS, PLUS, CIRCLE, DISC, RECT, BOX};
```

Creation

```
Qt_widget win( QWidget *parent = 0, const char *name = );
```

Constructs a widget which is a child of *parent*, with the name *name*. The default visible area is between ranges *xmin* = -1, *xmax* = 1, *ymin* = -1, *ymax* = 1. By default the X and Y scale factors are equal.

Qt_widget provides scaling support. You can use a scaling factor for your objects as well as you can draw the objects with one scale and look at the objects with different scales. Also there is a possibility to tell the widget that you want that the visible area should be mapped to a certain interval, and the widget adjust the scaling factor according to that.

```
void win.set_window( double x_min, double x_max, double y_min, double y_max, bool u = false)
```

Map the widget coordinates to the interval defined by the rectangle with given coordinates. This method should be called after *resize()* from *QWidget*. If *u* is *true* the X and Y scale factors will not be equal and the widget will not keep the aspect ratio.

```
void win.set_x_scale( double xscale)
```

Set the current X scaling factor. Actually the scales are recomputed when you resize, so this method is not used very often.

void *win.set_y_scale(double yscale)*

Set the current Y scaling factor. Actually the scales are recomputed when you resize, so this method is not used very often.

— *advanced* —

You should have the same scaling factor for X and Y if you want to keep the aspect ratio of your objects.

— *advanced* —

void *win.zoom(double ratio)* Multiply the X and Y scaling factors by *ratio*, then calls *redraw*.
The center of the visible area remains the same.

void *win.zoom(double ratio, double xc, double yc)*

Multiply the X and Y scaling factors by *ratio*, then calls *redraw*.
The center of the visible area becomes (xc, yc).

void *win.move_center(const double distx, const double disty)*

Move the center of the widget with *distx* on the X axis and *disty* on the Y axis.

void *win.set_center(const double x, const double y)*

Set the center of the widget to (x, y).

void *win.add_to_history()* Deprecated: This function adds the current viewport and transformations to history. You should use *save()* public slot from *Qt_widget_history* object.

void *win.clear_history()* Deprecated: Clears the history. This means that there are no elements in the history list. Call this method when you want to reinitialize the history list. You should use *clear_history()* method from *Qt_widget_standard_toolbar* or the *clear()* public slot from *Qt_widget_history* object.

public slots:

bool *win.back()*
bool *win.forth()*

Deprecated: This slots permit to walk into history. Return *true* if succeeded. These slots are deprecated. You should use the *backward()* and *forward()* methods of *Qt_widget_history* object, or the *back()* and *forward()* public slots from *Qt_widget_standard_toolbar* object.

void *win.clear()*

Clear the screen. The properties of the widget remain the same after calling this member function. You can see a list of the properties that you can set at the properties description.

<i>void</i>	<i>win.lock()</i>	Locks the widget, keeping the widget from being refreshed. If you lock the widget you should verify that you unlock it somewhere. The number of <i>lock()</i> calls should be the same with the number of <i>unlock()</i> calls. Lock and unlock calls can be nested.
<i>void</i>	<i>win.unlock()</i>	Unlocks the widget, and calls <i>do_paint()</i> . The widget is only unlocked if the number of unlock calls is equal to the number of lock calls.
<i>void</i>	<i>win.do_paint()</i>	Refresh the widget calling <i>paintEvent(QPaintEvent *e)</i> for the <i>Qt_widget</i> only if the widget is unlocked. This mean that <i>redraw()</i> is called if and only if the widget is unlocked.
<i>void</i>	<i>win.redraw()</i>	If you derive from <i>Qt_widget</i> you have to overload this function and put your code for drawing here if you don't use <i>layers</i> . The best way is to attach layers to the widget and call this method. This method redraws the layers attached and active. Before drawing the layers, redraw clear the screen and emit <i>redraw_on_back()</i> . Emit <i>redraw_on_front()</i> at the end.
<i>void</i>	<i>win.print_to_ps()</i>	Redraws all the attached and active layers into a Postscript device. It could be a file or a printer. This method also use signals as <i>redraw_on_back()</i> or <i>redraw_on_front()</i> .

Properties

You can set the properties of the functions through this functions as well as with the help of manipulators described later. The function naming convention has changed for this member functions, to *Qt* convention.

<i>void</i>	<i>win.setColor(QColor c)</i>	Set the current pen color of the widget to be <i>c</i> .
<i>void</i>	<i>win.setBackgroundColor(QColor c)</i>	Set the current background color to be <i>c</i> .
<i>void</i>	<i>win.setFillColor(QColor c)</i>	Set the current fill color of the widget to be <i>c</i> .
<i>void</i>	<i>win.setFilled(bool f)</i>	Set the status of the widget to true or false concerning filling the objects: polygons, circles, rectangles ...
<i>void</i>	<i>win.setLineWidth(unsigned int i)</i>	Set the current line width of the widget.
<i>void</i>	<i>win.setPointSize(unsigned int i)</i>	Set the current point size of the widget.

void *win.setPointStyle(PointStyle s)*

Set the current point style of the widget to *s*. *PointStyle* is an enumeration declared in *Qt_widget*.

void *win.setRasterOp(RasterOp r)*

Set the current raster operation.

Layers

void *win.attach(Qt_widget_layer* s)*

Add the layer *s* in the list of layers. The last added will be on top of the screen. Also the events are forwarded to layers in the order they have been attached.

void *win.detach(Qt_widget_layer* s)*

Remove the layer *s* from the list. *s* is a pointer to an existing layer.

New CGAL Objects

void *win.new_object(CGAL::Object obj)*

This function should be called by the tools that create CGAL objects. It then emits the signal *new_cgal_object(CGAL::Object)*. Slots of other components can be connected to this signal.

Access Functions

Note that we use also types from *Qt* here.

QColor *win.color()* Returns the current pen color. The color returned is a Qt class.

QColor *win.backgroundColor()* Returns the current widget background color. The color returned is a Qt class.

QColor *win.fillColor()* Returns the current color used for filling the objects. The color returned is a Qt class.

PointStyle *win.pointStyle()* Returns the current point style. *PointStyle* is an enumeration declared in *Qt_widget*.

uint *win.pointSize()* Returns the current point size.

<i>uint</i>	<i>win.lineWidth()</i>	Returns the current line width.
<i>RasterOp</i>	<i>win.rasterOp()</i>	Return the current raster operation.
<i>double</i>	<i>win.x_min()</i>	Returns the left <i>x</i> coordinate of the widget.
<i>double</i>	<i>win.y_min()</i>	Returns the lower <i>y</i> coordinate of the widget.
<i>double</i>	<i>win.x_max()</i>	Returns the right <i>x</i> coordinate of the widget.
<i>double</i>	<i>win.y_max()</i>	Returns the upper <i>y</i> coordinate of the widget.

advanced

The coordinates of the screen are mapped to a certain interval that you can choose with *set_window* member function. The scale of the objects you can visualize is computed and maintained the same for both axes to keep the aspect ratio of the objects. You should use *x_real* or *y_real* to get the real world coordinates, for your screen coordinates. If you are using *gmp* you can use this functions with *Gmpq* as second parameter. You may need it when you work with rationals. The double from the other function could be more complex and can make you loose speed in computations. Computing with rational coordinates directly could increase the speed, rather than computing with complex double coordinates.

double *win.x_real(int x)*
double *win.y_real(int y)*

template<class FT>
void *x_real(int x, FT&)* Returns the *x* real world coordinate of the *Qt_widget*.

template<class FT>
void *y_real(int y, FT&)* Returns the *y* real world coordinate of the *Qt_widget*.

void *win.x_real(int, Gmpq&)*
void *win.y_real(int, Gmpq&)*

advanced

int *win.x_pixel(double x)* This method is the opposite of *x_real* method. Converts the world coordinates in screen coordinates. If you want to get an integer between 0 and width, you should pass an *x* between *xmin* and *xmax*.

int *win.y_pixel(double y)* This method is the opposite of *y_real* method. Converts the world coordinates in screen coordinates. If you want to get an integer between 0 and height, you should pass an *y* between *ymin* and *ymax*.

advanced

Painter and Pixmap

In order that layers can draw on the drawing area of a widget, they have to access the underlying pixmap and painter.

Qt_widget use as a backbuffer for drawing a pixmap defined inside the class, i.e. an object of type *QPixmap*. The *QPixmap* class is an off-screen pixel-based paint device. One common use of the *QPixmap* class is to enable smooth updating of widgets. The *QPainter* class paints on paint devices. There is an object of type *QPainter* defined in *Qt_widget* that uses as a paint device the *QPixmap* object.

QPixmap& *win.get_pixmap()* Returns the current pixmap.

QPainter& *win.get_painter()* Returns the current painter.

advanced

Signals

```
void      win.s_mousePressEvent( QMouseEvent *e)
void      win.s_mouseReleaseEvent( QMouseEvent *e)
void      win.s_mouseMoveEvent( QMouseEvent *e)
void      win.s_paintEvent( QPaintEvent *e)
void      win.s_resizeEvent( QResizeEvent *e)
void      win.s_wheelEvent( QMouseEvent *e)
void      win.s_mouseDoubleClickEvent( QMouseEvent *e)
void      win.s_keyPressEvent( QKeyEvent *e)
void      win.s_keyReleaseEvent( QKeyEvent *e)
void      win.s_enterEvent( QEvent *e)
void      win.s_leaveEvent( QEvent *e)
void      win.s_event( QEvent *e)
```

The *Qt_widget* receives the events through virtual functions. This is the mechanism that *Qt* offers for dispatching events. This signals are called every time an event is dispatched to a virtual function. For example if *Qt_widget* receives *mousePressEvent(QMouseEvent *e)* emits *s_mousePressEvent(e)*. This is very useful when you have only a pointer to *Qt_widget*. It is enough to connect this slot to your function to receive the event.

void *win.new_cgal_object(CGAL::Object)*

Triggered when a new object from a tool is received. The user should catch this signal if it's working with tools that provide CGAL objects as input.

void *win.custom_redraw()* Deprecated: Emitted in the *redraw()* function after the layers are drawn.

void *win.redraw_on_back()* Emmitted in *redraw()* method before calling layer's redraw.

void *win.redraw_on_front()* Emmitted in *redraw()* method after calling layer's redraw.

void *win.rangesChanged()* Emmitted each time (xmin, xmax) and (ymin, ymax) ranges are changed.

Operators for Output

The output operator is defined for all geometric classes in the CGAL kernel.

```
template<class R>
Qt_widget&      & win << Point_2<R> p
```

```
template<class R>
Qt_widget&      & win << Segment_2<R> s
```

```
template<class R>
Qt_widget&      & win << Line_2<R> l
```

```
template<class R>
Qt_widget&      & win << Ray_2<R> r
```

```
template<class R>
Qt_widget&      & win << Triangle_2<R> t
```

```
template<class R>
Qt_widget&      & win << Iso_rectangle_2<R> r
```

```
template<class R>
Qt_widget&      & win << Circle_2<R> c
```

To use the other operators described you have to include the right header.

```
#include <CGAL/IO/Qt_widget_Polygon_2.h>
template<class Tr, class Co>
Qt_widget&      & win << Polygon_2<Tr,Co> pol
```

```
#include <CGAL/IO/Qt_widget_Min_ellipse_2.h>
template<class Traits_>
Qt_widget&      & win << Min_ellipse_2<Traits_> min_ellipse
```

```
#include <CGAL/IO/Qt_widget_Optimisation_ellipse_2.h>
template<class Traits_>
Qt_widget&      & win << Optimisation_ellipse_2<Traits_> oe
```

```
#include <CGAL/IO/Qt_widget_Optimisation_circle_2.h>
template<class Traits>
Qt_widget&      & win << Optimisation_circle_2<Traits> oc
```

```
#include <CGAL/IO/Qt_widget_Conic_2.h>
template<class R>
Qt_widget&      & win << Conic_2<R> c
```

```
#include <CGAL/IO/Qt_widget_Triangulation_2.h>
template <class Gt, class Tds>
Qt_widget&      & win << Triangulation_2<Gt, Tds> t
```

```

#include <CGAL/IO/Qt_widget_Alpha_shape_2.h>
template <class Dt>
Qt_widget&      & win << Alpha_shape_2<Dt> As

#include <CGAL/IO/Qt_widget_Delaunay_triangulation_2.h>
template<class Gt, class Tds>
Qt_widget&      & win << Delaunay_triangulation_2<R> dt

#include <CGAL/IO/Qt_widget_Constrained_triangulation_2.h>
template<class Gt, class Tds>
Qt_widget&      & win << Constrained_triangulation_2<Gt, Tds> t

#include <CGAL/IO/Qt_widget-Regular_triangulation_2.h>
template<class Gt, class Tds>
Qt_widget&      & win << Regular_triangulation_2<Gt, Tds> t

```

Manipulators for Qt_widget

A *manipulator* is an object which can be inserted in the *Qt_widget* , via the operator <<, to change the context for further drawing.

Here, we simply document the use of these operators which is all the user needs to know to modify the state of a stream.

<i>Qt_widget&</i>	<i>& win << BackgroundColor(Color c)</i>	Sets the color used for background color.
<i>Qt_widget&</i>	<i>& win << FillColor(Color c)</i>	Sets the color used for filling the objects.
<i>Qt_widget&</i>	<i>& win << LineWidth(const unsigned int i)</i>	Sets the width of the line for drawing objects.
<i>Qt_widget&</i>	<i>& win << PointSize(const unsigned int i)</i>	Sets the size of the points.
<i>Qt_widget&</i>	<i>& win << noFill</i>	Sets the state of <i>Qt_widget</i> concerning filling the objects to be false.
<i>Qt_widget&</i>	<i>& win << Color c</i>	Sets the color used as the <i>Qt_widget</i> fillColor.
<i>Qt_widget&</i>	<i>& win << PointStyle ps</i>	Sets the point style for <i>Qt_widget</i> .

Example

In the given example, that is found in tutorial/Qt_widget/tutorial1/tutorial1.C, we create an object of type *Qt_widget* and then we use the operators for output and the manipulators to show some of the widget's functionality.

```

#include <CGAL/Cartesian.h>
#include <CGAL/Bbox_2.h>
#include <list>
#include <CGAL/Polygon_2.h>

```



```

#include <CGAL/IO/Qt_widget_Polygon_2.h>
#include <qapplication.h>
#include <CGAL/IO/Qt_widget.h>

typedef CGAL::Cartesian<int>    Rep;
typedef CGAL::Point_2<Rep>     Point;
typedef CGAL::Circle_2<Rep>    Circle;
typedef CGAL::Segment_2<Rep>   Segment;
typedef CGAL::Line_2<Rep>      Line;
typedef CGAL::Ray_2<Rep>       Ray;
typedef CGAL::Triangle_2<Rep>  Triangle;
typedef CGAL::Iso_rectangle_2<Rep>
                                Cgal_Rectangle;

typedef CGAL::Bbox_2           BBox;
typedef std::list<Point>       Container;
typedef CGAL::Polygon_2<Rep, Container>
                                Cgal_Polygon;

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    using namespace CGAL;
    CGAL::Qt_widget * W = new CGAL::Qt_widget();
    app.setMainWidget( W );
    W->resize(600, 600);
    W->set_window(0, 600, 0, 600);
    W->show();
    //painting something on the screen
    W->lock();

    *W << BackgroundColor(ORANGE) << RED <<
    LineWidth(3) << PointSize(3) << PointStyle(DISC);
    *W << Segment(Point(10,20),Point(300,400));
    *W << LineWidth(5) << GREEN << FillColor(BLACK) <<
        Circle(Point(400,400),50*50);
    *W << LineWidth(1) << noFill << Circle(Point(300,300),300*300);
    *W << BLUE << LineWidth(2);
    *W << Segment(Point(200,200),Point(400,400));
    *W << Segment(Point(200,400),Point(400,200));
    W->setFilled(TRUE);
    *W << RED << Triangle(Point(150,300),
                           Point(150,350),
                           Point(100,325));
    *W << FillColor(RED) << Cgal_Rectangle(Point(320,220),
                                           Point(350,240));

    *W << DEEPBLUE << BBox(100,80,260,140);
    Cgal_Polygon p;
    p.push_back(Point(300,30));
    p.push_back(Point(400,30));
    p.push_back(Point(500,130));
    p.push_back(Point(400,180));
    p.push_back(Point(300,130));
    *W << p;
    *W << Ray(Point(200,400), Point(180,430))

```

```
        << Ray(Point(200,400), Point(180,370));  
W->unlock();  
  
return app.exec();  
}
```

CGAL::Qt_widget_layer

Definition

The *Qt_widget_layer* serves as a base class for layers. Layers are classes that draw on the drawing area of a *Qt_widget* and receive events from it. You can attach and detach layers on the *Qt_widget*. All the attached layers will receive the events if they are active. The same for drawing, all the active layers that are attached will be drawn.

```
#include <CGAL/IO/Qt_widget_layer.h>
```

Inherits From

QObject

Creation

```
Qt_widget_layer layer( QObject *parent = 0, const char* name = 0);
```

The default constructor. The parameters *parent* and *name* are passed to the *QObject* constructor.

<i>virtual void</i>	<i>layer.mousePressEvent(QMouseEvent *)</i>	
<i>virtual void</i>	<i>layer.mouseReleaseEvent(QMouseEvent *)</i>	
<i>virtual void</i>	<i>layer.wheelEvent(QMouseEvent *)</i>	
<i>virtual void</i>	<i>layer.mouseDoubleClickEvent(QMouseEvent *)</i>	
<i>virtual void</i>	<i>layer.mouseMoveEvent(QMouseEvent *)</i>	
<i>virtual void</i>	<i>layer.keyPressEvent(QKeyEvent *)</i>	
<i>virtual void</i>	<i>layer.keyReleaseEvent(QMouseEvent *)</i>	
<i>virtual void</i>	<i>layer.enterEvent(QEvent *)</i>	
<i>virtual void</i>	<i>layer.leaveEvent(QEvent *)</i>	
<i>virtual bool</i>	<i>layer.event(QEvent *)</i>	These virtual functions can be overloaded in the derived class. They are called by the <i>Qt_widget</i> to which the layer is attached.

<i>bool</i>	<i>layer.is_active()</i>	Returns <i>true</i> if this layer is active.
-------------	--------------------------	----------------------------------------------

Public Slots

<i>virtual void</i>	<i>layer.draw()</i>	This virtual member function must be overloaded. This is where the drawing code goes.
---------------------	---------------------	---------------------------------------------------------------------------------------

<i>void</i>	<i>layer.stateChanged(int s)</i>	
-------------	-----------------------------------	--

This slot is provided to change the layer's state from activated to deactivated and reverse if it is triggered. The layer is activated if *s* is 2, or it is deactivated if *s* is 0. These values match with the signal *stateChanged(int)* in the *QButton* widget.

bool *layer.activate()* Activate and return *true* if it was not active.

bool *layer.deactivate()* Deactivate and return *true* if it was active.

Protected

*Qt_widget** *widget;* The widget a layer is attached to or 0 otherwise.

virtual void *layer.activating()* You should overload this function if you want to have initializing code for your layer. This function is called every time the layer is activated.

virtual void *layer.deactivating()* You should overload this function if you want to write clean up code for your layer. This function is called every time the layer is deactivated.

Signals

void *layer.activated(*l)* This signal is emitted every time this layer is activated.

void *layer.deactivated(*l)* This signal is emitted every time this layer is deactivated.

Example

The following example of a layer draws the points of a triangulation in green.

```
#include <CGAL/IO/Qt_widget_layer.h>

namespace CGAL {

template <class T>
class Qt_layer_show_points : public Qt_widget_layer {
public:
    typedef typename T::Point            Point;
    typedef typename T::Segment        Segment;
    typedef typename T::Vertex        Vertex;
    typedef typename T::Vertex_iterator    Vertex_iterator;

    Qt_layer_show_points(T &t) : tr(t){};

    void draw()
    {
        Vertex_iterator it = tr.vertices_begin(),
            beyond = tr.vertices_end();
        *widget << CGAL::GREEN << CGAL::PointSize (3)
            << CGAL::PointStyle (CGAL::DISC);
        while(it != beyond) {
            *widget << (*it).point();
            ++it;
        }
    }
};

}
```

```
    }  
};  
private:  
    T &tr;  
  
}; //end class  
  
} // namespace CGAL
```

CGAL::Qt_widget_get_point<T>

Definition

An object of type *Qt_widget_get_point<T>* creates a CGAL point, every time the left mouse button is clicked.

```
#include <CGAL/IO/Qt_widget_get_point.h>
```

Parameters

The full template declaration of *Qt_widget_get_point* states one parameter:

```
template < class T >
class Qt_widget_get_point;
```

If T is one of the CGAL kernels you don't need additional types. If not, the parameter T has to provide this types:

Types

```
typedef T::Point_2
```

Point_2; This should be a Point type

```
typedef T::FT    FT;
```

FT; This should be a Field type

Inherits From

Qt_widget_layer

Creation

```
Qt_widget_get_point<T> getpoint( const QCursor c=QCursor(Qt::crossCursor),
                                QObject* parent = 0,
                                const char* name = 0)
```

c is the cursor that this layer will use when is active. *parent* is the parent widget and *name* is the name you give to this layer.

CGAL::Qt_widget_get_segment<T>

Definition

An object of type *Qt_widget_get_segment<T>* creates a CGAL segment in this way: one left click on the mouse will be the first point and the second point of the segment will be considered at the coordinates where the left mouse button is pressed for the second time. You can always cancel the creation process by pressing the ESC key. The use of *Qt_widget_get_segment<T>* requires that the mouse tracking is enabled for widgets attaching it.

```
#include <CGAL/IO/Qt_widget_get_segment.h>
```

Parameters

The full template declaration of *Qt_widget_get_segment* states one parameter:

```
template < class T >
class Qt_widget_get_segment;
```

If T is one of the CGAL kernels you don't need additional types. If not, the parameter T has to provide this types:

Types

```
typedef T::Point_2
```

Point_2; This should be a Point type

```
typedef T::Segment_2
```

Segment_2; This should be a Segment type

```
typedef T::FT    FT;
```

This should be a Field type

Inherits From

Qt_widget_layer

Creation

```
Qt_widget_get_segment<T> getsegment( const QCursor c=QCursor(Qt::crossCursor),
                                     QObject* parent = 0,
                                     const char* name = 0)
```

c is the cursor that this layer will use when is active. *parent* is the parent widget and *name* is the name you give to this layer.

CGAL::Qt_widget_get_line<T>

Definition

An object of type *Qt_widget_get_line<T>* creates a CGAL line in this way: one left click on the mouse will be the first point and the second point that generate the line will be considered at the coordinates where the left mouse button is pressed for the second time. You can always cancel the creation process by pressing the ESC key. The use of *Qt_widget_get_line<T>* requires that the mouse tracking is enabled for widgets attaching it.

```
#include <CGAL/IO/Qt_widget_get_line.h>
```

Parameters

The full template declaration of *Qt_widget_get_line* states one parameter:

```
template < class T >
class Qt_widget_get_line;
```

If T is one of the CGAL kernels you don't need additional types. If not, the parameter T has to provide this types:

Types

```
typedef T::Point_2
```

Point_2; This should be a Point type

```
typedef T::Line_2
```

Line_2; This should be a Line type

```
typedef T::FT    FT;
```

This should be a Field type

Inherits From

Qt_widget_layer

Creation

```
Qt_widget_get_line<T> getline( const QCursor c=QCursor(Qt::crossCursor),
                               QObject* parent = 0,
                               const char* name = 0)
```

c is the cursor that this layer will use when is active. *parent* is the parent widget and *name* is the name you give to this layer.

CGAL::Qt_widget_get_circle<T>

Definition

An object of type *Qt_widget_get_circle<T>* creates a CGAL circle in this way: one left click on the mouse will be the center of the circle and the second point will be considered at the coordinates where the left mouse button is pressed for the second time, the distance between those 2 representing the radius of the new circle. You can always cancel the creation process by pressing the ESC key. The use of *Qt_widget_get_circle<T>* requires that the mouse tracking is enabled for widgets attaching it.

```
#include <CGAL/IO/Qt_widget_get_circle.h>
```

Parameters

The full template declaration of *Qt_widget_get_circle* states one parameter:

```
template < class T >
class Qt_widget_get_circle;
```

If T is one of the CGAL kernels you don't need additional types. If not, the parameter T has to provide this types:

Types

```
typedef T::Point_2
```

Point_2; This should be a Point type

```
typedef T::Circle_2
```

Circle_2; This should be a Circle type

```
typedef T::FT    FT;
```

This should be a Field type

Inherits From

Qt_widget_layer

Creation

```
Qt_widget_get_circle<T> getcircle( const QCursor c=QCursor(Qt::crossCursor),
                                   QObject* parent = 0,
                                   const char* name = 0)
```

c is the cursor that this layer will use when is active. *parent* is the parent widget and *name* is the name you give to this layer.

CGAL::Qt_widget_get_iso_rectangle<T>

Definition

An object of type *Qt_widget_get_iso_rectangle<T>* creates a CGAL iso_rectangle this way one left click will be the first generator point, and second point will be considered at the coordinates where the left mouse button is pressed for the second time. You can always cancel the creation process by pressing the ESC key. The use of *Qt_widget_get_iso_rectangle<T>* requires that the mouse tracking is enabled for widgets attaching it.

```
#include <CGAL/IO/Qt_widget_get_iso_rectangle.h>
```

Parameters

The full template declaration of *Qt_widget_get_iso_rectangle* states one parameter:

```
template < class T >
class Qt_widget_get_iso_rectangle;
```

If T is one of the CGAL kernels you don't need additional types. If not, the parameter T has to provide this types:

Types

```
typedef T::RT    RT;           This should be a Ring type
```

Inherits From

Qt_widget_layer

Creation

```
Qt_widget_get_iso_rectangle<T> getisor( const QCursor c=QCursor(Qt::crossCursor),
                                         QObject* parent = 0,
                                         const char* name = 0)
```

c is the cursor that this layer will use when is active. *parent* is the parent widget and *name* is the name you give to this layer.

CGAL::Qt_widget_get_polygon<Polygon>

Definition

An object of type *Qt_widget_get_polygon<Polygon>* creates a CGAL polygon. A new vertex is inserted every time the left mouse button is pressed. The polygon is returned on a right click. You can use the *Escape* key if you want to remove your last entered point in the polygon. The use of *Qt_widget_get_polygon<Polygon>* requires that the mouse tracking is enabled for widgets attaching it.

```
#include <CGAL/IO/Qt_widget_get_polygon.h>
```

Parameters

The full template declaration of *Qt_widget_get_polygon* states one parameter:

```
template < class Polygon >
class Qt_widget_get_polygon;
```

Polygon should be a CGAL Polygon, or should provide the following types:

Types

```
typedef Polygon::Point_2
```

```
Point_2;
```

```
typedef Polygon::Segment_2
```

```
Segment2;
```

```
typedef Polygon::Edge_const_iterator
```

```
ECI;
```

```
typedef Polygon::Vertex_iterator
```

```
VI;
```

```
typedef Polygon::FT
```

```
FT;
```

Inherits From

```
Qt_widget_tool
```

Creation

```
Qt_widget_get_polygon<Polygon> getpoly( const QCursor c=QCursor(Qt::crossCursor),  
                                           QObject* parent = 0,  
                                           const char* name = 0)
```

c is the cursor that this layer will use when is active. *parent* is the parent widget and *name* is the name you give to this layer.

CGAL::Qt_widget_get_simple_polygon<Polygon>

Definition

An object of type *Qt_widget_get_simple_polygon<Polygon>* creates a CGAL simple polygon. A new vertex is inserted every time the left mouse button is pressed, if the polyline entered so far is simple. A right click closes the polygon, if it is simple. You can use the *Escape* key if you want to remove your last entered point in the polygon.

```
#include <CGAL/IO/Qt_widget_get_simple_polygon.h>
```

Parameters

The full template declaration of *Qt_widget_get_simple_polygon* states one parameter:

```
template < class Polygon >
class Qt_widget_get_simple_polygon;
```

Polygon should be a CGAL Polygon, or it should provide the following types:.

Types

```
typedef Polygon::Point_2
```

```
Point_2;
```

```
typedef Polygon::Segment_2
```

```
Segment2;
```

```
typedef Polygon::Edge_const_iterator
```

```
ECI;
```

Inherits From

```
Qt_widget_layer
```

Creation

```
Qt_widget_get_simple_polygon<Polygon> getspoly( const QCursor c=QCursor(Qt::crossCursor),
                                                QObject* parent = 0,
                                                const char* name = 0)
```

c is the cursor that this layer will use when is active. *parent* is the parent widget and *name* is the name you give to this layer.

CGAL::Navigation_layer

```
#include <CGAL/IO/Navigation_layer.h>
```

Inherits From

Qt_widget_layer

Creation

```
Navigation_layer nav_layer( QObject* parent = 0, const char* name = 0);
```

parent is the parent widget and *name* is the name you give to this layer.

This layer is used to provide basic navigation control over the *Qt_widget*. When this layer is attached and active, you may use the arrows to navigate the equivalent in world coordinates of 10 pixels in the desired direction. You can also use the PageUp, PageDown keys to scroll the equivalent in world coordinates of half screenheight on the Y axes.

CGAL::Custom_zoom_layer

```
#include <CGAL/IO/Custom_zoom_layer.h>
```

Inherits From

Custom_zoom_layer

Creation

```
Custom_zoom_layer zoom_layer( QObject* parent = 0, const char* name = 0);
```

parent is the parent widget and *name* is the name you give to this layer.

The standard toolbar already provide zooming functionality, but this layer comes to complete the latest. When this layer is attached and active, zooming functionality could be used anytime. To zoom in x2 you must press the + key on your keyboard. To zoom out x2 you must press the - key on your keyboard. You can also use the mouse to zoom to a region defined by one rectangle. To define the rectangle you must press Ctrl+LeftMouseButton for the first corner of the rectangle, you may release, then move, then press Ctrl+LeftMouseButton once again to define the second corner.

CGAL::Qt_widget_standard_toolbar

Definition

The standard toolbar includes the basic tools used for zooming and translating in a *Qt_widget*.

```
#include <CGAL/IO/Qt_widget_standard_toolbar.h>
```

Inherits From

QToolBar

Creation

```
Qt_widget_standard_toolbar toolbar( Qt_widget *w,
                                   QMainWindow *mw,
                                   QWidget* parent,
                                   bool newline = true,
                                   const char* name = 0)
```

This constructor creates a new toolbar, called *name*, in your application, containing all the standard tools. The first parameter *w* is a non-null pointer to the *Qt_widget* object on which the toolbar functionalities act. The second parameter *mw* is a pointer to the *QMainWindow* that manages the toolbar. The later is added to the top area of the *QMainWindow*, unless *mw=0*. *mw*, *parent*, *newline*, *name* are passed to the *QMainWindow* constructor (with first parameter *label="Qt_widget standard toolbar"*²).

```
Qt_widget_standard_toolbar toolbar( Qt_widget *w, QMainWindow *parent = 0, const char* name = 0);
```

Simplified version of the previous one. The *parent* is the *QMainWindow* that manages the toolbar.

*const QToolBar**

toolbar.toolbar()

Deprecated: (in CGAL-2.4 the standard toolbar was not derived from *QToolBar*) Returns a pointer to the *QToolBar* *this* itself.

public slots:

void toolbar.back() Goes back in the history list of the standard toolbar.

void toolbar.forward() Goes forward in the history list of the standard toolbar.

void toolbar.clear_history() Clears the history list of the standard toolbar.

Example


```

#include <CGAL/Cartesian.h>
#include <CGAL/Point_2.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/IO/Qt_widget_Delaunay_triangulation_2.h>

#include <qapplication.h>
#include <qmainwindow.h>

#include <CGAL/IO/Qt_widget.h>
#include <CGAL/IO/Qt_widget_layer.h>
#include <CGAL/IO/Qt_widget_standard_toolbar.h>

typedef CGAL::Cartesian<double>          Rep;
typedef CGAL::Point_2<Rep>              Point;
typedef CGAL::Delaunay_triangulation_2<Rep> Delaunay;

Delaunay dt;

class My_layer : public CGAL::Qt_widget_layer{
    void draw(){
        *widget << CGAL::BLACK;
        *widget << dt;
    }
};

class My_widget : public CGAL::Qt_widget {
public:
    My_widget(QMainWindow* c) : CGAL::Qt_widget(c) {};
private:
    //this event is called only when the user presses the mouse
    void mousePressEvent(QMouseEvent *e)
    {
        Qt_widget::mousePressEvent(e);
        dt.insert(Point(x_real(e->x()), y_real(e->y())));
        redraw();
    }
};

class My_window : public QMainWindow{
public:
    My_window(int x, int y)
    {
        widget = new My_widget(this);
        setCentralWidget(widget);
        resize(x,y);
        widget->set_window(0, x, 0, y);

        //How to attach the standard toolbar
        stoolbar = new CGAL::Qt_widget_standard_toolbar(widget, this,
                                                         "Standard toolbar");
        widget->attach(&v);
    }
private:
    My_widget *widget;

```

```

    My_layer v;
    CGAL::Qt_widget_standard_toolbar *stoolbar;
};

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    My_window W(400,400);
    app.setMainWidget( &W );
    W.show();
    W.setCaption("Using the Standard Toolbar");
    return app.exec();
}

```

This example is implemented in the fifth tutorial. You can look over the code to see how the code works.

CGAL::Qt_widget_history

Definition

This class provides basic functionality to manipulate intervals of *Qt_widget* class. *Qt_widget* knows about the current visible area that is specified through an interval. The default one is (-1, 1, -1, 1). This class is mostly used by the *Qt_widget_standard_toolbar* to provide the backward and the forward navigation through the different intervals.

```
#include <CGAL/IO/Qt_widget_history.h>
```

Inherits From

QObject

Creation

```
Qt_widget_history history( Qt_widget* parent, const char *name = 0);
```

parent is the parent widget that has to be a *Qt_widget* and *name* is the name of the history. This will create a history object that can deal with the *Qt_widget* visualisation intervals. Each time the visible area will change, it's interval will be stored in the history list.

public slots:

<i>void</i>	<i>history.save()</i>	This saves the widget visible area in the history list.
<i>void</i>	<i>history.backward()</i>	This goes back in the list of saved intervals.
<i>void</i>	<i>history.forward()</i>	This goes forward in the list of saved intervals.
<i>void</i>	<i>history.clear()</i>	This will clear the entire history list.

signals:

<i>void</i>	<i>history.backwardAvailable(bool)</i>
<i>void</i>	<i>history.forwardAvailable(bool)</i>

This two signals are emitted when there are available items in the history list as regards to the current position in the list.

CGAL::Qt_help_window

Definition

This class is a specialization of the *QTextBrowser* class, that provides a rich text browser with hypertext navigation. It has couple more functionalities as navigation support and printing.

```
#include <CGAL/IO/Qt_help_window.h>
```

Inherits From

QMainWindow

Creation

```
Qt_help_window hw( QString home_, QString path, QWidget* parent = 0, const char *name=0);
```

home_ is the name of the file considered as home, *path* is the path to this file, *parent* is the parent widget, and *name* is the name of this help class.

public slots:

<i>void</i>	<i>hw.print()</i>	This prints the current displayed page to the printer device of your choice.
-------------	-------------------	------------------------------------------------------------------------------

Example

```
#include <CGAL/IO/Qt_help_window.h>
//...
QString home;
home = "help/index.html";
CGAL::Qt_help_window *help = new
    CGAL::Qt_help_window(home, ".", 0, "help viewer");
help->resize(400, 400);
help->setCaption("Demo HowTo");
help->show();
```

Bibliography

- [AA95] O. Aichholzer and F. Aurenhammer. Straight skeletons for general polygonal figures. Technical Report 432, Inst. for Theor. Comput. Sci., Graz Univ. of Technology, Graz, Austria, 1995.
- [AAAG95] O. Aichholzer, D. Alberts, F. Aurenhammer, and B. Gärtner. A novel type of skeleton for polygons. *J. Universal Comput. Sci.*, 1(12):752–761, 1995.
- [AB99] Nina Amenta and Marshall Bern. Surface reconstruction by Voronoi filtering. *Discrete Comput. Geom.*, 22(4):481–504, 1999.
- [ADS98] Pierre Alliez, Olivier Devillers, and Jack Snoeyink. Removing degeneracies by perturbing the problem or the world. In *Proc. 10th Canad. Conf. Comput. Geom.*, 1998.
- [AKM⁺87] A. Aggarwal, M. M. Klawe, S. Moran, P. W. Shor, and R. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2:195–208, 1987.
- [Ale01] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.
- [AM93a] S. Arya and D. M. Mount. Algorithms for fast vector quantization. In *Data Compression Conference*, pages 381–390. IEEE Press, 1993.
- [AM93b] S. Arya and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 271–280, 1993.
- [And78] K. R. Anderson. A reevaluation of an efficient algorithm for determining the convex hull of a finite planar set. *Inform. Process. Lett.*, 7(1):53–55, 1978.
- [And79] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Inform. Process. Lett.*, 9(5):216–219, 1979.
- [AS00] Pankaj K. Agarwal and Micha Sharir. Arrangements and their applications. In Jörg-Rüdiger Sack and Jorge Urrutia, editors, *Handbook of Computational Geometry*, pages 49–119. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.
- [AT78] S. G. Akl and G. T. Toussaint. A fast convex hull algorithm. *Inform. Process. Lett.*, 7(5):219–222, 1978.
- [Aus98] Matthew H. Austern. *Generic Programming and the STL*. Addison-Wesley, 1998.
- [Bau75] B. G. Baumgart. A polyhedron representation for computer vision. In *Proc. AFIPS Natl. Comput. Conf.*, volume 44, pages 589–596. AFIPS Press, Alrlington, Va., 1975.
- [BB97] F. Bernardini and C. Bajaj. Sampling and reconstructing manifolds using alpha-shapes. Technical Report CSD-TR-97-013, Dept. Comput. Sci., Purdue Univ., West Lafayette, IN, 1997.
- [BBP01] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Applied Mathematics*, 109:25–47, 2001.

- [BDH96] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The Quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, December 1996.
- [BDP⁺02] Jean-Daniel Boissonnat, Olivier Devillers, Sylvain Pion, Monique Teillaud, and Mariette Yvinec. Triangulations in CGAL. *Comput. Geom. Theory Appl.*, 22:5–19, 2002.
- [BDTY00] Jean-Daniel Boissonnat, Olivier Devillers, Monique Teillaud, and Mariette Yvinec. Triangulations in CGAL. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 11–18, 2000.
- [BEH⁺02] Eric Berberich, Arno Eigenwillig, Michael Hemmer, Susan Hert, Kurt Mehlhorn, and Elmar Schömer. A computational basis for conic arcs and boolean operations on conic polygons. In Rolf Möhring and Rajeev Raman, editors, *Algorithms - ESA 2002: 10th Annual European Symposium*, volume 2461 of *Lecture Notes in Computer Science*, pages 174–186, Rome, Italy, September 2002. Springer.
- [Ben75] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.
- [BF02] Jean-Daniel Boissonnat and Julia Flötotto. A local coordinate system on a surface. In *Proc. 7th ACM Symposium on Solid Modeling and Applications*, 2002.
- [BFH95] Heinzgerd Bendels, Dieter W. Fellner, and Sven Havemann. Modellierung der grundlagen: Erweiterbare datenstrukturen zur modellierung und visualisierung polygonaler welten. In D. W. Fellner, editor, *Modeling – Virtual Worlds – Distributed Graphics*, pages 149–157, Bad Honnef / Bonn, 27.–28. November 1995.
- [BGH97] Julien Basch, Leonidas Guibas, and John Hersherberger. Data structures for mobile data. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 747–756, 1997.
- [BMS94] C. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 16–23, 1994.
- [BO05] Jean-Daniel Boissonnat and Steve Oudot. Provably good sampling and meshing of surfaces. *Graphical Models*, 67:405–451, 2005.
- [BPP95] Gavin Bell, Anthony Parisi, and Mark Pesce. Vrm1 the virtual reality modeling language: Version 1.0 specification. <http://www.vrml.org/>, May 26 1995. Third Draft.
- [Bro97] J. L. Brown. Systems of coordinates associated with points scattered in the plane. *Comput. Aided Design*, 14:547–559, 1997.
- [Bur96] C. Burnikel. *Exact Computation of Voronoi Diagrams and Line Segment Intersections*. Ph.D thesis, Universität des Saarlandes, March 1996.
- [BY98] Jean-Daniel Boissonnat and Mariette Yvinec. *Algorithmic Geometry*. Cambridge University Press, UK, 1998. Translated by Hervé Brönnimann.
- [Byk78] A. Bykat. Convex hull of a finite set of points in two dimensions. *Inform. Process. Lett.*, 7:296–298, 1978.
- [C++98] International standard ISO/IEC 14882: Programming languages – C++. American National Standards Institute, 11 West 42nd Street, New York 10036, 1998.
- [CC78] E. Catmull and J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer Aided Design*, 10:350–355, 1978.
- [CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2nd edition, 2001.
- [CMS93] K. L. Clarkson, K. Mehlhorn, and R. Seidel. Four results on randomized incremental constructions. *Comput. Geom. Theory Appl.*, 3(4):185–212, 1993.

- [dBvKOS97] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.
- [dBvKOS00] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2nd edition, 2000.
- [Dev98] Olivier Devillers. Improved incremental randomized Delaunay triangulation. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 106–115, 1998.
- [Dev02] Olivier Devillers. The Delaunay hierarchy. *Internat. J. Found. Comput. Sci.*, 13:163–180, 2002.
- [DFMT00] Olivier Devillers, Alexandra Fronville, Bernard Mourrain, and Monique Teillaud. Algebraic methods and arithmetic filtering for exact predicates on circle arcs. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 139–147, 2000.
- [DFMT02] Olivier Devillers, Alexandra Fronville, Bernard Mourrain, and Monique Teillaud. Algebraic methods and arithmetic filtering for exact predicates on circle arcs. *Comput. Geom. Theory Appl.*, 22:119–142, 2002.
- [DLPT98] Olivier Devillers, Giuseppe Liotta, Franco P. Preparata, and Roberto Tamassia. Checking the convexity of polytopes and the planarity of subdivisions. *Comput. Geom. Theory Appl.*, 11:187–208, 1998.
- [DMA02] Mathieu Desbrun, Mark Meyer, and Pierre Alliez. Intrinsic parameterizations of surface meshes. *Computer Graphics Forum*, 21(3):209–218, September 2002.
- [DP03] Olivier Devillers and Sylvain Pion. Efficient exact geometric predicates for Delaunay triangulations. In *Proc. 5th Workshop Algorithm Eng. Exper.*, pages 37–44, 2003.
- [DPT02] Olivier Devillers, Sylvain Pion, and Monique Teillaud. Walking in a triangulation. *Internat. J. Found. Comput. Sci.*, 13:181–199, 2002.
- [DT03] Olivier Devillers and Monique Teillaud. Perturbations and vertex removal in a 3D Delaunay triangulation. In *Proc. 14th ACM-SIAM Sympos. Discrete Algorithms (SODA)*, pages 313–319, 2003.
- [Edd77] W. F. Eddy. A new convex hull algorithm for planar sets. *ACM Trans. Math. Softw.*, 3:398–403 and 411–412, 1977.
- [EDD⁺95] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution analysis of arbitrary meshes. In *Computer Graphics (Proc. SIGGRAPH '95)*, volume 29, pages 173–182, 1995. Examples in `file://ftp.cs.washington.edu/pub/graphics`.
- [Ede92] H. Edelsbrunner. Weighted alpha shapes. Technical Report UIUCDCS-R-92-1760, Dept. Comput. Sci., Univ. Illinois, Urbana, IL, 1992.
- [EE98] David Eppstein and Jeff Erickson. Raising roofs, crashing cycles, and playing pool: Applications of a data structure for finding pairwise interactions. In *Symposium on Computational Geometry*, pages 58–67, 1998.
- [EKP⁺04] Ioannis Z. Emiris, Athanasios Kakargias, Sylvain Pion, Monique Teillaud, and Elias P. Tsigaridas. Towards an open curved kernel. In *Proc. 20th Annu. ACM Sympos. Comput. Geom.*, pages 438–446, 2004.
- [EM94] H. Edelsbrunner and E. P. Mücke. Three-dimensional alpha shapes. *ACM Trans. Graph.*, 13(1):43–72, January 1994.
- [ES96] H. Edelsbrunner and N. R. Shah. Incremental topological flipping works for regular triangulations. *Algorithmica*, 15:223–241, 1996.

- [Far90] G. Farin. Surfaces over Dirichlet tessellations. *Comput. Aided Geom. Design*, 7:281–292, 1990.
- [FBF77] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3:209–226, 1977.
- [FH05] M. S. Floater and K. Hormann. Surface parameterization: a tutorial and survey. In N. A. Dodgson, M. S. Floater, and M. A. Sabin, editors, *Advances in Multiresolution for Geometric Modelling*, Mathematics and Visualization, pages 157–186. Springer, Berlin, Heidelberg, 2005.
- [FJ83] G. N. Frederickson and D. B. Johnson. Finding k th paths and p -centers by generating and searching good data structures. *J. Algorithms*, 4:61–80, 1983.
- [FJ84] G. N. Frederickson and D. B. Johnson. Generalized selection and ranking: sorted matrices. *SIAM J. Comput.*, 13:14–30, 1984.
- [Flo03a] Michael Floater. Mean value coordinates. *Computer Aided Design*, 20(1):19–27, 2003.
- [Flö03b] Julia Flötotto. *A coordinate system associated to a point cloud issued from a manifold: definition, properties and applications*. Thèse de doctorat en sciences, Université de Nice-Sophia Antipolis, France, 2003.
- [FO98] Petr Felkel and Stěpán Obdržálek. Straight skeleton implementation. In László Szirmay Kalos, editor, *14th Spring Conference on Computer Graphics (SCCG'98)*, pages 210–218, 1998.
- [Gär99] B. Gärtner. Fast and robust smallest enclosing balls. In *Proc. 7th annu. European Symposium on Algorithms (ESA)*, volume 1643 of *Lecture Notes in Computer Science*, pages 325–338. Springer-Verlag, 1999.
- [GGHT97] M. Goodrich, L. J. Guibas, J. Hershberger, and P. Tanenbaum. Snap rounding line segments efficiently in two and three dimensions. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 284–293, 1997.
- [GHH⁺03] Miguel Granados, Peter Hachenberger, Susan Hert, Lutz Ketter, Kurt Mehlhorn, and Michael Seel. Boolean operations on 3d selective nef complexes data structure, algorithms, and implementation. In Giuseppe Di Battista and Uri Zwick, editors, *Algorithms - ESA 2003: 11th Annual European Symposium*, volume 2832 of *Lecture Notes in Computer Science*, pages 174–186, Budapest, Hungary, September 2003. Springer.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [GKR04] Leonidas Guibas, Menelaos Karavelos, and Daniel Russel. A computational framework for handling motion. In *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments*, pages 129–141, 2004.
- [GM98] Leonidas Guibas and David Marimont. Rounding arrangements dynamically. *Internat. J. Comput. Geom. Appl.*, 8:157–176, 1998.
- [Gra] Torbjörn Granlund. GMP, the GNU multiple precision arithmetic library. <http://www.swox.com/gmp/>.
- [Gra72] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inform. Process. Lett.*, 1:132–133, 1972.
- [Gre83] Daniel H. Greene. The decomposition of polygons into convex parts. In Franco P. Preparata, editor, *Computational Geometry*, volume 1 of *Adv. Comput. Res.*, pages 235–259. JAI Press, Greenwich, Conn., 1983.
- [GS78] Leonidas J. Guibas and R. Sedgwick. A dichromatic framework for balanced trees. In *Proc. 19th Annu. IEEE Sympos. Found. Comput. Sci.*, Lecture Notes Comput. Sci., pages 8–21. Springer-Verlag, 1978.

- [GS85] Leonidas J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graph.*, 4(2):74–123, April 1985.
- [GS97a] B. Gärtner and S. Schönherr. Exact primitives for smallest enclosing ellipses. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 430–432, 1997.
- [GS97b] Bernd Gärtner and Sven Schönherr. Smallest enclosing ellipses – fast and exact. Serie B – Informatik B 97-03, Freie Universität Berlin, Germany, June 1997. URL <http://www.inf.fu-berlin.de/inst/pubs/tr-b-97-03.abstract.html>.
- [GS98a] Bernd Gärtner and Sven Schönherr. Smallest enclosing circles – an exact and generic implementation in C++. Serie B – Informatik B 98-04, Freie Universität Berlin, Germany, April 1998. URL <http://www.inf.fu-berlin.de/inst/pubs/tr-b-98-04.abstract.html>.
- [GS98b] Bernd Gärtner and Sven Schönherr. Smallest enclosing ellipses – an exact and generic implementation in C++. Serie B – Informatik B 98-05, Freie Universität Berlin, Germany, April 1998. URL <http://www.inf.fu-berlin.de/inst/pubs/tr-b-98-05.abstract.html>.
- [GS00] Bernd Gärtner and Svend Schönherr. An efficient, exact, and generic quadratic programming solver for geometric optimization. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 110–118, 2000.
- [Hal04] Dan Halperin. Arrangements. In Jacob E. Goodman and Joseph O’Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 24, pages 529–562. Chapman & Hall/CRC, 2nd edition, 2004.
- [Han91] E. N. Hanson. The interval skip list: a data structure for finding all intervals that overlap a point. In *Proc. 2nd Workshop Algorithms Data Struct.*, volume 519 of *Lecture Notes Comput. Sci.*, pages 153–164. Springer-Verlag, 1991.
- [HH05] Idit Haran and Dan Halperin. Efficient point location in cgal arrangements using landmarks, 2005.
- [HM83] S. Hertel and K. Mehlhorn. Fast triangulation of simple polygons. In *Proc. 4th Internat. Conf. Found. Comput. Theory*, volume 158 of *Lecture Notes Comput. Sci.*, pages 207–218. Springer-Verlag, 1983.
- [Hob99] J. D. Hobby. Practical segment intersection with finite precision output. *Comput. Geom. Theory Appl.*, 13(4):199–214, October 1999.
- [Hof89a] C. Hoffmann. *Geometric and Solid Modeling*. Morgan-Kaufmann, San Mateo, CA, 1989.
- [Hof89b] C. M. Hoffmann. The problems of accuracy and robustness in geometric computation. *IEEE Computer*, 22(3):31–41, March 1989.
- [Hof89c] Christoph M. Hoffmann. *Geometric and Solid Modeling - An Introduction*. Morgan Kaufmann, 1989.
- [Hof99] M. Hoffmann. A simple linear algorithm for computing rectangular three-centers. In *Proc. 11th Canad. Conf. Comput. Geom.*, pages 72–75, 1999.
- [Hof04] Christoph M. Hoffmann. Solid modeling. In Jacob E. Goodman and Joseph O’Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 56, pages 1257–1278. Chapman & Hall/CRC, 2nd edition, 2004.
- [HP02] D. Halperin and E. Packer. Iterated snap rounding. *Computational Geometry: Theory and Applications*, 23(2):209–225, 2002.
- [HS95] G. R. Hjaltason and H. Samet. Ranking in spatial databases. In M. J. Egenhofer and J. R. Herring, editors, *Advances in Spatial Databases - Fourth International Symposium*, number 951 in *Lecture Notes Comput. Sci.*, pages 83–95, August 1995.

- [HS00] Hisamoto Hiyoshi and Kokichi Sugihara. Voronoi-based interpolation with higher continuity. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 242–250, 2000.
- [HW96] Jed Hartman and Josie Wernecke. *The VRML 2.0 Handbook: Building Moving Worlds on the Web*. Addison-Wesley, 1996.
- [Jar73] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Inform. Process. Lett.*, 2:18–21, 1973.
- [Kar04] Menelaos I. Karavelas. A robust and efficient implementation for the segment Voronoi diagram. In *Proc. Internat. Symp. on Voronoi diagrams in Science and Engineering (VD2004)*, pages 51–62, 2004.
- [KE02] Menelaos I. Karavelas and Ioannis Z. Emiris. Predicates for the planar additively weighted Voronoi diagram. Technical Report ECG-TR-122201-01, INRIA Sophia-Antipolis, Sophia-Antipolis, May 2002.
- [KE03] Menelaos I. Karavelas and Ioannis Z. Emiris. Root comparison techniques applied to computing the additively weighted Voronoi diagram. In *Proc. 14th ACM-SIAM Sympos. Discrete Algorithms (SODA)*, pages 320–329, 2003.
- [Ket98] L. Kettner. Designing a data structure for polyhedral surfaces. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 146–154, 1998.
- [Ket99] L. Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *Comput. Geom. Theory Appl.*, 13:65–90, 1999.
- [Kha96] L. Khachiyan. Rounding of polytopes in the real number model of computation. *Mathematics of Operations Research*, 21(2):307–320, 1996.
- [Kle89] Rolf Klein. *Concrete and Abstract Voronoi Diagrams*, volume 400 of *Lecture Notes Comput. Sci.* Springer-Verlag, 1989.
- [KLPY99] Vijay Karamcheti, Chen Li, Igor Pechtchanski, and Chee Yap. *The CORE Library Project*, 1.2 edition, 1999. <http://www.cs.nyu.edu/exact/core/>.
- [KM76] K. Kuratowski and A. Mostowski. *Set Theory*. North-Holland Publishing Co., 1976.
- [Kob00] Leif Kobbelt. $\sqrt{3}$ -subdivision. In *Computer Graphics (Proc. SIGGRAPH '00)*, volume 34, pages 103–112, 2000.
- [KY02] Menelaos Karavelas and Mariette Yvinec. Dynamic additively weighted voronoi diagrams in 2d. In *Proc. 10th European Symposium on Algorithms*, pages 586–598, 2002.
- [LD03] R. G. Laycock and A. M. Day. Automatically generating roof models from building footprints. In *The 11-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision'2003. Journal of WSCG - FULL Papers*, volume 11, 2003.
- [Lev05] Bruno Levy. Numerical methods for digital geometry processing. In *Israel Korea Bi-National Conference - Invited talk, extended abstract (full paper will be available shortly)*, November 2005.
- [LPRM02] Bruno Lévy, Sylvain Petitjean, Nicolas Ray, and Jérôme Maillot. Least squares conformal maps for automatic texture atlas generation. In *Proceedings of the 29th Conference on Computer Graphics and Interactive Techniques SIGGRAPH*, volume 21(3) of *ACM Transactions on Graphics*, pages 362–371, 2002.
- [MAD05] Abdelkrim Mebarki, Pierre Alliez, and Olivier Devillers. Farthest point seeding for efficient placement of streamlines. In *Proceeding of IEEE Visualization*, 2005.
- [Män88] M. Mäntylä. *An Introduction to Solid Modeling*. Computer Science Press, Rockville, MD, 1988.

- [Meh84] Kurt Mehlhorn. *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*, volume 3 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, Germany, 1984.
- [Mel87] A. Melkman. On-line construction of the convex hull of a simple polyline. *Inform. Process. Lett.*, 25:11–12, 1987.
- [MN00] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.
- [MNS⁺96] Kurt Mehlhorn, Stefan Näher, Thomas Schilz, Stefan Schirra, Michael Seel, Raimund Seidel, and Christian Uhrig. Checking geometric programs or verification of geometric structures. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 159–165, 1996.
- [MNSU] K. Mehlhorn, S. Näher, M. Seel, and C. Uhrig. *The LEDA User Manual*. Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany. <http://www.mpi-sb.mpg.de/LEDA/leda.html>.
- [MP78] D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoret. Comput. Sci.*, 7:217–236, 1978.
- [MS96] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, 1996.
- [MSW92] J. Matoušek, Micha Sharir, and Emo Welzl. A subexponential bound for linear programming. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 1–8, 1992.
- [Mul90] K. Mulmuley. A fast planar partition algorithm, I. *J. Symbolic Comput.*, 10(3-4):253–280, 1990.
- [Mye95] Nathan C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.
- [Orl90] M. Orlowski. A new algorithm for the largest empty rectangle problem. *Algorithmica*, 5:65–73, 1990.
- [Phi96] Mark Phillips. *Geomview Manual, Version 1.6.1 for Unix Workstations*. The Geometry Center, University of Minnesota, 1996. <http://www.geom.umn.edu/software/download/geomview.html>.
- [Pip93] B. Piper. Properties of local coordinates based on dirichlet tessellations. *Computing Suppl.*, 8:227–239, 1993.
- [PP93] U. Pinkall and K. Polthier. Computing discrete minimal surfaces and their conjugates. *Experimental Mathematics*, 2(1):15–36, 1993.
- [PTVF02] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C++*. Cambridge University Press, 2nd edition, 2002.
- [Pug90] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.
- [Req80] Aristides G. Requicha. Representations for rigid solids: Theory, methods, and systems. *ACM Computing Surveys*, 12(4):437–464, 1980.
- [RY06] Laurent Rineau and Mariette Yvinec. A generic software design for Delaunay refinement meshing. *to be published*, 2006.
- [Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.
- [Sch96] Michael Schutte. Zufällige konvexe mengen. Master’s thesis, Freie Universität Berlin, Germany, 1996.

- [Sch00] Stefan Schirra. Robustness and precision issues in geometric computation. In Jörg-Rüdiger Sack and Jorge Urrutia, editors, *Handbook of Computational Geometry*, chapter 14, pages 597–632. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.
- [Sei91] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Comput. Geom. Theory Appl.*, 1(1):51–64, 1991.
- [She98] Jonathan R. Shewchuk. A condition guaranteeing the existence of higher-dimensional constrained delaunay triangulations. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 76–85, 1998.
- [She00] Jonathan R. Shewchuk. Mesh generation for domains with small angles. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 1–10, 2000.
- [Sib80] R. Sibson. A vector identity for the Dirichlet tessellation. *Math. Proc. Camb. Phil. Soc.*, 87:151–155, 1980.
- [Sib81] R. Sibson. A brief description of natural neighbour interpolation. In Vic Barnett, editor, *Interpreting Multivariate Data*, pages 21–36. John Wiley & Sons, Chichester, 1981.
- [Sil97] Silicon Graphics Computer Systems, Inc. Standard template library programmer’s guide. <http://www.sgi.com/Technology/STL/>, 1997.
- [Sk172] J. Sklansky. Measuring concavity on rectangular mosaic. *IEEE Trans. Comput.*, C-21:1355–1364, 1972.
- [SL95] Alexander Stepanov and Meng Lee. The standard template library. <http://www.cs.rpi.edu/~musser/doc.ps>, October 1995.
- [SM00] M. Seel and K. Mehlhorn. Infimaximal frames: A technique for making lines look like segments. Research Report MPI-I-2000-1-005, MPI für Informatik, Saarbrücken, Germany, December 2000. <http://www.mpi-sb.mpg.de/~mehlhorn/ftp/InfiFrames.ps>.
- [SP05] Le-Jeng Shiue and Jörg Peters. Mesh refinement based on euler encoding. In *Proceedings of the International Conference on Shape Modeling and Applications 2005*, pages 343–348, 2005.
- [STV⁺95] Christian Schwarz, Jürgen Teich, Alek Vainshtein, Emo Welzl, and Brian L. Evans. Minimal enclosing parallelogram with application. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C34–C35, 1995.
- [SW96] Micha Sharir and Emo Welzl. Rectilinear and polygonal p -piercing and p -center problems. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 122–132, 1996.
- [Tar83] R. E. Tarjan. *Data Structures and Network Algorithms*, volume 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.
- [Tei99] Monique Teillaud. Three dimensional triangulations in CGAL. In *Abstracts 15th European Workshop Comput. Geom.*, pages 175–178. INRIA Sophia-Antipolis, 1999.
- [Tou83] G. T. Toussaint. Solving geometric problems with the rotating calipers. In *Proc. IEEE MELECON '83*, pages A10.02/1–4, 1983.
- [Tut63] W. T. Tutte. How to draw a graph. *Proceedings London Mathematical Society*, 13(52):743–768, 1963.
- [Vai90] A. Vainshtein. Finding minimal enclosing parallelograms. *Diskretnaya Matematika*, 2:72–81, 1990. In Russian.

- [vLS82] J. van Leeuwen and Anneke A. Schoone. Untangling a travelling salesman tour in the plane. In J. R. Mühlbacher, editor, *Proc. 7th Internat. Workshop Graph-Theoret. Concepts Comput. Sci.*, pages 87–98, München, 1982. Hanser.
- [VRM96] The virtual reality modeling language specification: Version 2.0, ISO/IEC CD 14772. <http://www.vrml.org/>, August 4 1996.
- [Wei85] K. Weiler. Edge-based data structures for solid modeling in a curved surface environment. *IEEE Comput. Graph. Appl.*, 5(1):21–40, 1985.
- [Wel91] Emo Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *New Results and New Trends in Computer Science*, volume 555 of *Lecture Notes Comput. Sci.*, pages 359–370. Springer-Verlag, 1991.
- [Wer94] Josie Wernicke. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor, Release 2*. Addison-Wesley, 1994.
- [WW02] Joe Warren and Henrik Weimer. *Subdivision Methods for Geometric Design*. Morgan Kaufmann Publishers, New York, 2002.
- [Yap97] C. Yap. Towards exact geometric computation. *Comput. Geom. Theory Appl.*, 7(1):3–23, 1997.
- [YD95] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 4 of *Lecture Notes Series on Computing*, pages 452–492. World Scientific, Singapore, 2nd edition, 1995.
- [ZE02] Afra Zomorodian and Herbert Edelsbrunner. Fast software for box intersection. *Int. J. Comput. Geom. Appl.*, 12:143–172, 2002.

Index

Pages on which definitions are given are presented in **boldface**.

!

Nef_polyhedron_3, 1037

!=

Circle_2, 66

Circular_arc_point_2, 558

Circulator, 2705

Compact_container, 2613

const, 2576

Direction_2, 67

Direction_3, 92

int, 2576

Interval, 2033

Interval_nt, 2546

Iso_box_d, 472

Iso_cuboid_3, 95

Iso_rectangle_2, 69

Line_2, 73

Line_3, 97

Matrix, 444

Nef_polyhedron_3, 1036

Plane_3, 100

Point_2, 75

Point_3, 102

Polygon_2, 730

Ray_2, 79

Ray_3, 105

Segment_2, 81

Segment_3, 107

Sphere_3, 110

Sphere_point, 997

Tetrahedron_3, 112

Triangle_2, 83

Triangle_3, 114

Triangulation_3, 1506

Vector_2, 85

Vector_3, 116

%

EuclideanRingNumberType, 2528

%=

EuclideanRingNumberType, 2528

()

Abs, 2522

AdaptableFunctor, 2656

Aff_transformation_2, 62

Aff_transformation_3, 90

AlgebraicKernelForCircles::CompareX, 595

AlgebraicKernelForCircles::CompareXY, 597

AlgebraicKernelForCircles::CompareY, 596

AlgebraicKernelForCircles::ConstructPolynomial_1_2, 599

AlgebraicKernelForCircles::ConstructPolynomialForCircles_2_2, 600

AlgebraicKernelForCircles::SignAt, 598

AlgebraicKernelForCircles::Solve, 601

AlgebraicKernelForCircles::XCriticalPoints, 602

AlgebraicKernelForCircles::YCriticalPoints, 603

BasicMatrix, 2327

Cartesian_converter, 42

Cast_function_object, 2671

CircularKernel::CompareX_2, 570

CircularKernel::CompareXY_2, 572

CircularKernel::CompareY_2, 571

CircularKernel::CompareYatX_2, 573

CircularKernel::CompareYtoRight_2, 574

CircularKernel::ConstructBbox_2, 569

CircularKernel::ConstructCircle_2, 561

CircularKernel::ConstructCircularArc_2, 564

CircularKernel::ConstructCircularArcPoint_2, 562

CircularKernel::ConstructCircularMaxVertex_2, 566

CircularKernel::ConstructCircularMinVertex_2, 565

CircularKernel::ConstructCircularSourceVertex_2, 567

CircularKernel::ConstructCircularTargetVertex_2, 568

CircularKernel::ConstructLine_2, 560

CircularKernel::ConstructLineArc_2, 563

CircularKernel::DoOverlap_2, 580

CircularKernel::Equal_2, 578

CircularKernel::GetEquation, 585

CircularKernel::HasOn_2, 579

CircularKernel::Intersect_2, 576

CircularKernel::InXRange_2, 581

[CircularKernel::IsVertical_2, 582](#)
[CircularKernel::IsXMonotone_2, 583](#)
[CircularKernel::IsYMonotone_2, 584](#)
[CircularKernel::MakeXMonotone_2, 575](#)
[CircularKernel::Split_2, 577](#)
[Compare, 2524](#)
[ConstructFunction, 2484](#)
[Creator_1, 2681](#)
[Creator_2, 2682](#)
[Creator_3, 2683](#)
[Creator_4, 2684](#)
[Creator_5, 2685](#)
[Creator_uniform_2, 2686](#)
[Creator_uniform_3, 2687](#)
[Creator_uniform_4, 2688](#)
[Creator_uniform_5, 2689](#)
[Creator_uniform_6, 2690](#)
[Creator_uniform_7, 2691](#)
[Creator_uniform_8, 2692](#)
[Creator_uniform_9, 2693](#)
[Creator_uniform_d, 2694](#)
[Data_access, 2370](#)
[Dereference, 2669](#)
[Div, 2527](#)
[Dynamic_matrix, 2323](#)
[Filtered_predicate, 46](#)
[Function, 2491](#)
[Gcd, 2538](#)
[Get_address, 2670](#)
[Handle_hash_function, 2775](#)
[Homogeneous_converter, 49](#)
[Identity, 2668](#)
[ImplicitFunction, 1832](#)
[Integrator_2, 2402](#)
[Is_convex_2, 753](#)
[Is_negative, 2553](#)
[Is_one, 2554](#)
[Is_positive, 2555](#)
[Is_vacuously_valid, 754](#)
[Is_y_monotone_2, 755](#)
[Is_zero, 2556](#)
[Kernel::Affine_rank_d, 502](#)
[Kernel::Affinely_independent_d, 501](#)
[Kernel::Angle_2, 208](#)
[Kernel::Angle_3, 209](#)
[Kernel::AreOrderedAlongLine_2, 210](#)
[Kernel::AreOrderedAlongLine_3, 211](#)
[Kernel::AreParallel_2, 212](#)
[Kernel::AreParallel_3, 213](#)
[Kernel::AreStrictlyOrderedAlongLine_2, 214](#)
[Kernel::AreStrictlyOrderedAlongLine_3, 215](#)
[Kernel::Assign_2, 216](#)
[Kernel::Assign_3, 217](#)
[Kernel::BoundedSide_2, 218](#)

[Kernel::BoundedSide_3, 219](#)
[Kernel::Center_of_sphere_d, 504](#)
[Kernel::Collinear_2, 226](#)
[Kernel::Collinear_3, 227](#)
[Kernel::CollinearAreOrderedAlongLine_2, 221](#)
[Kernel::CollinearAreOrderedAlongLine_3, 222](#)
[Kernel::CollinearAreStrictlyOrderedAlongLine_2, 223](#)
[Kernel::CollinearAreStrictlyOrderedAlongLine_3, 224](#)
[Kernel::CollinearHasOn_2, 225](#)
[Kernel::Compare_lexicographically_d, 505](#)
[Kernel::CompareAngleWithXAxis_2, 228](#)
[Kernel::CompareDistance_2, 229](#)
[Kernel::CompareDistance_3, 230](#)
[Kernel::CompareSlope_2, 231](#)
[Kernel::CompareX_2, 237](#)
[Kernel::CompareX_3, 238](#)
[Kernel::CompareXAtY_2, 232](#)
[Kernel::CompareXY_2, 235](#)
[Kernel::CompareXY_3, 236](#)
[Kernel::CompareXYZ_3, 234](#)
[Kernel::CompareY_2, 241](#)
[Kernel::CompareY_3, 242](#)
[Kernel::CompareYAtX_2, 239, 240](#)
[Kernel::CompareZ_3, 243](#)
[Kernel::ComputeA_2, 244](#)
[Kernel::ComputeArea_2, 247](#)
[Kernel::ComputeArea_3, 248](#)
[Kernel::ComputeB_2, 245](#)
[Kernel::ComputeC_2, 246](#)
[Kernel::ComputeScalarProduct_2, 249](#)
[Kernel::ComputeScalarProduct_3, 250](#)
[Kernel::ComputeSquaredArea_3, 251](#)
[Kernel::ComputeSquaredDistance_2, 252](#)
[Kernel::ComputeSquaredDistance_3, 253](#)
[Kernel::ComputeSquaredLength_2, 254](#)
[Kernel::ComputeSquaredLength_3, 255](#)
[Kernel::ComputeSquaredRadius_2, 256](#)
[Kernel::ComputeSquaredRadius_3, 257](#)
[Kernel::ComputeVolume_3, 258](#)
[Kernel::ComputeX_2, 259](#)
[Kernel::ComputeXmax_2, 263](#)
[Kernel::ComputeXmin_2, 261](#)
[Kernel::ComputeY_2, 260](#)
[Kernel::ComputeYAtX_2, 265](#)
[Kernel::ComputeYmax_2, 264](#)
[Kernel::ComputeYmin_2, 262](#)
[Kernel::ConstructBaseVector_3, 266](#)
[Kernel::ConstructBbox_2, 267](#)
[Kernel::ConstructBbox_3, 268](#)
[Kernel::ConstructBisector_2, 269](#)

Kernel::ConstructBisector_3, [270](#)
 Kernel::ConstructCartesianConstIterator_2, [271](#)
 Kernel::ConstructCartesianConstIterator_3, [272](#)
 Kernel::ConstructCartesianConstIterator_d, [507](#)
 Kernel::ConstructCenter_2, [273](#)
 Kernel::ConstructCenter_3, [274](#)
 Kernel::ConstructCentroid_2, [275](#)
 Kernel::ConstructCentroid_3, [276](#)
 Kernel::ConstructCircle_2, [277](#)
 Kernel::ConstructCircumcenter_2, [278](#)
 Kernel::ConstructCircumcenter_3, [279](#)
 Kernel::ConstructCrossProductVector_3, [280](#)
 Kernel::ConstructDifferenceOfVectors_2, [281](#)
 Kernel::ConstructDirection_2, [282](#)
 Kernel::ConstructDirection_3, [283](#)
 Kernel::ConstructIsoCuboid_3, [284](#)
 Kernel::ConstructIsoRectangle_2, [285](#)
 Kernel::ConstructLiftedPoint_3, [286](#)
 Kernel::ConstructLine_2, [287](#)
 Kernel::ConstructLine_3, [288](#)
 Kernel::ConstructMaxVertex_2, [289](#)
 Kernel::ConstructMaxVertex_3, [290](#)
 Kernel::ConstructMidpoint_2, [291](#)
 Kernel::ConstructMidpoint_3, [292](#)
 Kernel::ConstructMinVertex_2, [293](#)
 Kernel::ConstructMinVertex_3, [294](#)
 Kernel::ConstructObject_2, [295](#)
 Kernel::ConstructObject_3, [296](#)
 Kernel::ConstructOppositeCircle_2, [297](#)
 Kernel::ConstructOppositeDirection_2, [298](#)
 Kernel::ConstructOppositeDirection_3, [299](#)
 Kernel::ConstructOppositeLine_2, [300](#)
 Kernel::ConstructOppositeLine_3, [301](#)
 Kernel::ConstructOppositePlane_3, [302](#)
 Kernel::ConstructOppositeRay_2, [303](#)
 Kernel::ConstructOppositeRay_3, [304](#)
 Kernel::ConstructOppositeSegment_2, [305](#)
 Kernel::ConstructOppositeSegment_3, [306](#)
 Kernel::ConstructOppositeSphere_3, [307](#)
 Kernel::ConstructOppositeTriangle_2, [308](#)
 Kernel::ConstructOppositeVector_2, [309](#)
 Kernel::ConstructOppositeVector_3, [310](#)
 Kernel::ConstructOrthogonalVector_3, [311](#)
 Kernel::ConstructPerpendicularDirection_2, [312](#)
 Kernel::ConstructPerpendicularLine_2, [313](#)
 Kernel::ConstructPerpendicularLine_3, [314](#)
 Kernel::ConstructPerpendicularPlane_3, [315](#)
 Kernel::ConstructPerpendicularVector_2, [316](#)
 Kernel::ConstructPlane_3, [317](#)

Kernel::ConstructPoint_2, [321](#)
 Kernel::ConstructPoint_3, [322](#)
 Kernel::ConstructPointOn_2, [319](#)
 Kernel::ConstructPointOn_3, [320](#)
 Kernel::ConstructProjectedPoint_2, [323](#)
 Kernel::ConstructProjectedPoint_3, [324](#)
 Kernel::ConstructProjectedXYPoint_2, [325](#)
 Kernel::ConstructRay_2, [326](#)
 Kernel::ConstructRay_3, [327](#)
 Kernel::ConstructScaledVector_2, [328](#)
 Kernel::ConstructScaledVector_3, [329](#)
 Kernel::ConstructSegment_2, [331](#)
 Kernel::ConstructSegment_3, [332](#)
 Kernel::ConstructSphere_3, [333](#)
 Kernel::ConstructSumOfVectors_2, [330](#)
 Kernel::ConstructSupportingPlane_3, [335](#)
 Kernel::ConstructTetrahedron_3, [336](#)
 Kernel::ConstructTranslatedPoint_2, [337](#)
 Kernel::ConstructTranslatedPoint_3, [338](#)
 Kernel::ConstructTriangle_2, [339](#)
 Kernel::ConstructTriangle_3, [340](#)
 Kernel::ConstructVector_2, [341](#)
 Kernel::ConstructVector_3, [342](#)
 Kernel::ConstructVertex_2, [343](#)
 Kernel::ConstructVertex_3, [344](#)
 Kernel::Contained_in_affine_hull_d, [508](#)
 Kernel::Contained_in_linear_hull_d, [509](#)
 Kernel::Contained_in_simplex_d, [510](#)
 Kernel::Coplanar_3, [349](#)
 Kernel::CoplanarOrientation_3, [347](#)
 Kernel::CoplanarSideOfBoundedCircle_3, [348](#)
 Kernel::CounterclockwiseInBetween_2, [350](#)
 Kernel::DoIntersect_2, [353](#)
 Kernel::DoIntersect_3, [354](#)
 Kernel::Equal_2, [361](#)
 Kernel::Equal_3, [362](#)
 Kernel::Equal_d, [511](#)
 Kernel::EqualX_2, [356](#)
 Kernel::EqualX_3, [357](#)
 Kernel::EqualXY_3, [355](#)
 Kernel::EqualY_2, [358](#)
 Kernel::EqualY_3, [359](#)
 Kernel::EqualZ_3, [360](#)
 Kernel::Has_on_positive_side_d, [512](#)
 Kernel::HasOn_2, [373](#)
 Kernel::HasOn_3, [374](#)
 Kernel::HasOnBoundary_2, [363](#)
 Kernel::HasOnBoundary_3, [364](#)
 Kernel::HasOnBoundedSide_2, [365](#)
 Kernel::HasOnBoundedSide_3, [366](#)
 Kernel::HasOnNegativeSide_2, [367](#)
 Kernel::HasOnNegativeSide_3, [368](#)
 Kernel::HasOnPositiveSide_2, [369](#)

Kernel::HasOnPositiveSide_3, 370
 Kernel::HasOnUnboundedSide_2, 371
 Kernel::HasOnUnboundedSide_3, 372
 Kernel::Intersect_2, 375
 Kernel::Intersect_3, 376
 Kernel::Intersect_d, 513
 Kernel::IsDegenerate_2, 377
 Kernel::IsDegenerate_3, 378
 Kernel::IsHorizontal_2, 380
 Kernel::IsVertical_2, 383
 Kernel::LeftTurn_2, 384
 Kernel::Less_lexicographically_d, 514
 Kernel::Less_or_equal_lexicographically_d, 515
 Kernel::LessDistanceToPoint_2, 385
 Kernel::LessDistanceToPoint_3, 386
 Kernel::LessRotateCCW_2, 387
 Kernel::LessSignedDistanceToLine_2, 388
 Kernel::LessSignedDistanceToPlane_3, 389
 Kernel::LessX_2, 393
 Kernel::LessX_3, 394
 Kernel::LessXY_2, 391
 Kernel::LessXY_3, 392
 Kernel::LessXYZ_3, 390
 Kernel::LessY_2, 396
 Kernel::LessY_3, 397
 Kernel::LessYX_2, 395
 Kernel::LessZ_3, 398
 Kernel::Lift_to_paraboloid_d, 516
 Kernel::Linear_base_d, 518
 Kernel::Linear_rank_d, 519
 Kernel::Linearly_independent_d, 517
 Kernel::Midpoint_d, 520
 Kernel::Orientation_2, 403
 Kernel::Orientation_3, 404
 Kernel::Orientation_d, 521
 Kernel::Oriented_side_d, 522
 Kernel::OrientedSide_2, 405
 Kernel::OrientedSide_3, 406
 Kernel::Orthogonal_vector_d, 523
 Kernel::Point_of_sphere_d, 524
 Kernel::Point_to_vector_d, 525
 Kernel::Project_along_d_axis_d, 526
 Kernel::Side_of_bounded_sphere_d, 527
 Kernel::Side_of_oriented_sphere_d, 528
 Kernel::SideOfBoundedCircle_2, 416
 Kernel::SideOfBoundedSphere_3, 417
 Kernel::SideOfOrientedCircle_2, 418
 Kernel::SideOfOrientedSphere_3, 419
 Kernel::Squared_distance_d, 529
 Kernel::Value_at_d, 530
 Kernel::Vector_to_point_d, 531
 Matrix, 443
 Max, 2566

Min, 2567
 Modifier_base, 2777
 MonotoneMatrixSearchTraits, 2325
 PolygonIsValid, 771
 Project_facet, 2673
 Project_next, 2677
 Project_next_opposite, 2679
 Project_normal, 2675
 Project_opposite_prev, 2680
 Project_plane, 2676
 Project_point, 2674
 Project_prev, 2678
 Project_vertex, 2672
 Projection_object, 2667
 Random, 2757
 Sgn, 2589
 Sqrt, 2590
 Square, 2591
 To_double, 2592
 To_interval, 2593
 UniqueHashFunction, 2784
 *
 Aff_transformation_2, 62
 Aff_transformation_3, 90
 Aff_transformation_d, 476
 Circulator, 2705
 const, 441, 444, 2577
 Handle, 2729
 int, 2577
 Matrix, 444
 Nef_polyhedron_3, 1037
 Vector, 441
 Vector_2, 86, 87, 189
 Vector_3, 117, 118, 189
 Vector_d, 453, 454
 * =
 Nef_polyhedron_3, 1037
 RingNumberType, 2577
 Vector, 441
 Vector_d, 453
 +
 Bbox_2, 64
 Bbox_3, 88
 Circulator, 2706
 const, 2577, 2580
 difference_type, 2706
 int, 2577
 Line_d, 459
 Matrix, 444
 Nef_polyhedron_3, 1037
 Point_2, 77, 187
 Point_3, 104, 187
 Point_d, 449
 Ray_d, 461

Segment_d, 463
Sphere_d, 471
Vector, 441
Vector_2, 86
Vector_3, 117
Vector_d, 453
++
Circulator, 2705
+=
Circulator, 2706
Nef_polyhedron_3, 1037
Point_d, 449
RingNumberType, 2577
Vector, 441
Vector_d, 453
—
Circulator, 2707
const, 2577
Direction_2, 68
Direction_3, 92
Direction_d, 456
int, 2577
Matrix, 444
Nef_polyhedron_3, 1037
Point_2, 77, 188
Point_3, 104, 188
Point_d, 449
Vector, 441
Vector_2, 86
Vector_3, 117
Vector_d, 454
--
Circulator, 2706
--=
Circulator, 2706
Nef_polyhedron_3, 1037
Point_d, 449
RingNumberType, 2577
Vector, 441
Vector_d, 454
—>
Circulator, 2705
Handle, 2729
/
const, 2530
int, 2530
Interval_nt, 2545
Vector_2, 87
Vector_3, 118
Vector_d, 453
/=
FieldNumberType, 2530
Vector, 441
Vector_d, 453
<
Circular_arc_point_2, 558
Compact_container, 2613
const, 2576
Direction_2, 67
In_place_list, 2604
int, 2576
Interval_nt, 2546
Multiset, 2616
Nef_polyhedron_3, 1036
Point_2, 77
Point_3, 104
Quadruple, 2702
Triple, 2699
<<
Alpha_shape_2, 1609, 2848
Alpha_shape_3, 1638
Apollonius_graph_2, 1719
Apollonius_graph_hierarchy_2, 1735
Apollonius_site_2, 1721
Arrangement_2, 1235
Arrangement_with_history_2, 1235
Bbox_2, 2818
Bbox_3, 2818
CGAL::General_polygon_2, 951
CGAL::General_polygon_with_holes_2, 951
CGAL::Nef_polyhedron_3, 1032, 1053
CGAL::Polygon_with_holes_2, 951
CGAL::Polyhedron_3, 789, 797, 832
Circle_2, 2847
Circular_arc_2, 555
Circular_arc_point_2, 558
Class, 2786
Color, 2848
Conic_2, 2847
Constrained_triangulation_2, 1391, 2848
Delaunay_triangulation_2, 2848
Event, 2487, 2493
Geomview_stream, 2819–2821
Interval_skip_list, 2032
Interval_skip_list_interval, 2034
Iso_rectangle_2, 2847
Kd_tree_rectangle, 2075
Lazy_exact_nt, 2558
Level_interval, 2035
Line_2, 2817, 2847
Line_3, 2817
Line_arc_2, 556
LineWidth(const), 2848
Min_annulus_d, 2249
Min_circle_2, 2197
Min_ellipse_2, 2207, 2847
Min_sphere_d, 2242
MP_Float, 2568

noFill, 2848
Number_type_checker, 2571
Optimisation_circle_2, 2847
Optimisation_ellipse_2, 2847
Plane_separator, 2089
Point_2, 2816, 2847
Point_3, 2817
PointSize(const), 2848
PointStyle, 2848
Polygon_2, 2847
Polyhedron_3, 2818
Polytope_distance_d, 2320
Quotient, 2573
Ray_2, 2817, 2847
Ray_3, 2817
Regular_triangulation_2, 2848
Segment_2, 2817, 2847
Segment_3, 2817
Segment_Delaunay_graph_2, 1668
Segment_Delaunay_graph_hierarchy_2, 1695
Sphere_3, 2818
Tetrahedron_3, 2817
Time, 2510
Triangle_2, 2817, 2847
Triangle_3, 2817
Triangulation_2, 1440, 2819, 2847
Triangulation_3, 1519, 2819
TriangulationDataStructure_2::Face, 1475
TriangulationDataStructure_2::Vertex, 1479
TriangulationDataStructure_3, 1470, 1584
TriangulationDSCellBase_3, 1591
TriangulationDSVertexBase_3, 1593
TriangulationVertexBase_2, 1427
TriangulationVertexBase_3, 1547
Verbose_ostream, 2805
Voronoi_diagram_2, 1756

<=
Circular_arc_point_2, 558
Compact_container, 2613
const, 2576
Direction_2, 67
int, 2576
Interval_nt, 2546
Nef_polyhedron_3, 1036
Point_2, 77
Point_3, 104

=
Apollonius_graph_2, 1714
Apollonius_graph_filtered_traits_2, 1732
Apollonius_graph_hierarchy_2, 1733
Apollonius_graph_traits_2, 1731
ApolloniusGraphTraits_2, 1730
Arrangement_2, 1203
Arrangement_with_history_2, 1319

ArrangementBasicTraits_2, 1258
ArrangementDirectionalXMonotoneTraits_2, 929
ArrangementLandmarkTraits_2, 1259
ArrangementTraits_2, 1263
ArrangementXMonotoneTraits_2, 1262
Circulator, 2705
Compact_container, 2611
Constrained_triangulation_plus_2, 1395
DelaunayTriangulationTraits_2, 1400
Fixed_precision_nt, 2534
GeneralPolygon_2, 924
GeneralPolygonSetTraits_2, 931
GeneralPolygonWithHoles_2, 926
HalfedgeDS, 850
In_place_list, 2604
Largest_empty_iso_rectangle_2, 2299
LargestEmptyIsoRectangleTraits_2, 2301
Multiset, 2615
Object, 120
Plane_separator, 2088
PolyhedronTraits_3, 827
RegularTriangulationTraits_2, 1409
SurfaceMeshTriangulation_3, 1858
Taucs_vector, 2000
Triangulation_2, 1431
Triangulation_3, 1506
Triangulation_euclidean_traits_xy_3, 1446
TriangulationDataStructure_2, 1464
TriangulationDataStructure_3, 1574
TriangulationTraits_2, 1425
UniqueHashFunction, 2784
Vector, 2003

==
AlgebraicKernelForCircles::const, 593
Arr_consolidated_curve_data_traits_2<Traits, Data>::Data_container, 1290

Circle_2, 66
Circular_arc_point_2, 558
Circulator, 2705
Compact_container, 2613
const, 589, 2576
Direction_2, 67
Direction_3, 92
In_place_list, 2604
int, 2576
Interval, 2033
Interval_nt, 2546
Iso_box_d, 472
Iso_cuboid_3, 95
Iso_rectangle_2, 69
Line_2, 72
Line_3, 97
Matrix, 444

Multiset, 2616
Nef_polyhedron_3, 1036
Plane_3, 100
Point_2, 75
Point_3, 102
Point_d, 450
Polygon_2, 730
Quadruple, 2702
Random, 2758
Ray_2, 79
Ray_3, 105
Segment_2, 81
Segment_3, 107
Sphere_3, 110
Sphere_point, 997
Tetrahedron_3, 112
Triangle_2, 83
Triangle_3, 114
Triangulation_3, 1506
Triple, 2700
Vector_2, 85
Vector_3, 116

>

Circular_arc_point_2, 558
Compact_container, 2613
const, 2576
Direction_2, 67
int, 2576
Interval_nt, 2546
Nef_polyhedron_3, 1036
Point_2, 77
Point_3, 104

>=

Circular_arc_point_2, 558
Compact_container, 2613
const, 2576
Direction_2, 67
int, 2576
Interval_nt, 2546
Nef_polyhedron_3, 1036
Point_2, 77
Point_3, 104

>>

Apollonius_graph_2, 1719
Apollonius_graph_hierarchy_2, 1735
Apollonius_site_2, 1721
Arrangement_2, 1234
Arrangement_with_history_2, 1234
CGAL::General_polygon_2, 952
CGAL::General_polygon_with_holes_2, 952
CGAL::Nef_polyhedron_3, 1032, 1054
CGAL::Polygon_with_holes_2, 952
CGAL::Polyhedron_3, 789, 798, 833
Circular_arc_2, 555
Circular_arc_point_2, 558
Class, 2786, 2792, 2800
Geomview_stream, 2820
Lazy_exact_nt, 2558
Line_arc_2, 556
Min_annulus_d, 2249
Min_circle_2, 2197
Min_ellipse_2, 2207
Min_sphere_d, 2242
MP_Float, 2568
Number_type_checker, 2571
Point_3, 2818
Polytope_distance_d, 2320
Quotient, 2573
Segment_Delaunay_graph_2, 1668
Segment_Delaunay_graph_hierarchy_2, 1695
Triangulation_2, 1440
Triangulation_3, 1519
TriangulationDataStructure_2::Face, 1475
TriangulationDataStructure_2::Vertex, 1479
TriangulationDataStructure_3, 1469, 1584
TriangulationDSCellBase_3, 1591
TriangulationDSVertexBase_3, 1593
TriangulationVertexBase_2, 1426
TriangulationVertexBase_3, 1547
Voronoi_diagram_2, 1756

[]

Arr_polyline_traits_2<SegmentTraits>::Curve_2, 1269
Circulator, 2707
Direction_d, 456
Hyperplane_d, 466
Inverse_index, 2634
Iso_cuboid_3, 95
Iso_rectangle_2, 70
Kinetic::ActiveObjectsTable, 2478
Point_2, 76
Point_3, 103
Point_d, 448
Polygon_2, 730
Random_access_adaptor, 2635
Random_access_value_adaptor, 2636
Segment_2, 81
Segment_3, 107
Segment_d, 463
Taucs_vector, 2000
Tetrahedron_3, 112
Triangle_2, 83
Triangle_3, 114
Unique_hash_map, 2783
Vector, 440, 2003
Vector_2, 86
Vector_3, 117
Vector_d, 452

- ^
 - Nef_polyhedron_3*, 1037
- ^=
 - Nef_polyhedron_3*, 1037
- a
 - Line_2*, 73
 - Plane_3*, 100
- ABORT, 8
- Abs, 2522
- abs, 2521
- ABSOLUTE_INDEXING, 818
- access
 - Approximate_min_ellipsoid_d*, 2267
 - Min_annulus_d*, 2245
 - Min_circle_2*, 2194
 - Min_ellipse_2*, 2205
 - Min_sphere_d*, 2239
 - Min_sphere_of_spheres_d*, 2253
 - Polytope_distance_d*, 2315
 - Width_3*, 2307
- access_coordinates_begin_d_object
 - Optimisation_d_traits_2*, 2280
 - Optimisation_d_traits_3*, 2282
 - Optimisation_d_traits_d*, 2284
 - OptimisationDTraits*, 2286
- access_dimension_d_object
 - Optimisation_d_traits_2*, 2280
 - Optimisation_d_traits_3*, 2282
 - Optimisation_d_traits_d*, 2284
 - OptimisationDTraits*, 2286
- access_site_2_object
 - AdaptationTraits_2*, 1769
- achieved_epsilon
 - Approximate_min_ellipsoid_d*, 2267
- activate
 - Qt_widget_layer*, 2851
- activated
 - Qt_widget_layer*, 2852
- activating
 - Qt_widget_layer*, 2852
- active_points_[123]_table_handle
 - Kinetic::SimulationTraits*, 2502
- ACUTE, 123
- AdaptableFunctor, 2656–2657
- Adaptation_policy, 1751
- adaptation_policy
 - Voronoi_diagram_2*, 1753
- Adaptation_traits, 1751
- adaptation_traits
 - Voronoi_diagram_2*, 1753
- AdaptationPolicy_2, 1770–1771
- AdaptationTraits_2, 1768–1769
- add_coef
 - Matrix*, 1954
 - Taucs_matrix*, 1994
- add_face_to_border
 - HalfedgeDS_decorator*, 867
- add_facet
 - Polyhedron_incremental_builder_3*, 820
- add_facet_to_border
 - Polyhedron_3*, 807
- add_hidden_site
 - ApolloniusGraphVertexBase_2*, 1725
- add_hole
 - ArrangementDcelFace*, 1245
- add_isolated_vertex
 - ArrangementDcelFace*, 1245
- add_to_complex
 - SurfaceMeshComplex_2InTriangulation_3*, 1849
- add_to_history
 - Qt_widget*, 2842
- add_vertex
 - Polyhedron_incremental_builder_3*, 819
- add_vertex_and_facet_to_border
 - Polyhedron_3*, 807
- add_vertex_to_facet
 - Polyhedron_incremental_builder_3*, 819
- Aff_transformation_2, 60–63
- Aff_transformation_3, 89–91
- Aff_transformation_d, 474–476, 2382
- affine_rank, 478
- affinely_independent, 477
- after_add_hole
 - Arr_observer*, 1315
- after_add_isolated_vertex
 - Arr_observer*, 1316
- after_assign
 - Arr_observer*, 1313
- after_attach
 - Arr_observer*, 1314
- after_clear
 - Arr_observer*, 1313
- after_create_edge
 - Arr_observer*, 1314
- after_create_vertex
 - Arr_observer*, 1314
- after_edge_flip
 - Kinetic::DelaunayTriangulationVisitor3*, 2436
 - Kinetic::RegularTriangulationVisitor3*, 2453
- after_facet_flip
 - Kinetic::DelaunayTriangulationVisitor3*, 2436
 - Kinetic::RegularTriangulationVisitor3*, 2453
- after_flip
 - Kinetic::RegularTriangulationVisitor3*, 2453

- Kinetic::DelaunayTriangulationVisitor2, 2435
- after_global_change
 - Arr_observer, 1313
- after_merge_edge
 - Arr_observer, 1316
- after_merge_face
 - Arr_observer, 1316
- after_merge_hole
 - Arr_observer, 1316
- after_modify_edge
 - Arr_observer, 1314
- after_modify_vertex
 - Arr_observer, 1314
- after_move_hole
 - Arr_observer, 1316
- after_move_isolated_vertex
 - Arr_observer, 1317
- after_remove_edge
 - Arr_observer, 1317
- after_remove_hole
 - Arr_observer, 1317
- after_remove_vertex
 - Arr_observer, 1317
- after_split_edge
 - Arr_observer, 1315
- after_split_face
 - Arr_observer, 1315
- after_split_hole
 - Arr_observer, 1315
- after_swap
 - Kinetic::SortVisitor, 2459
- Algebraic_kernel_for_circles_2_2, 588
- AlgebraicKernelForCircles, 586–587
- AlgebraicKernelForCircles::CompareX, 595
- AlgebraicKernelForCircles::CompareXY, 597
- AlgebraicKernelForCircles::CompareY, 596
- AlgebraicKernelForCircles::ConstructPolynomial_1_2, 599
- AlgebraicKernelForCircles::ConstructPolynomialForCircles_2_2, 600
- AlgebraicKernelForCircles::Polynomial_1_2, 591
- AlgebraicKernelForCircles::PolynomialForCircles_2_2, 593
- AlgebraicKernelForCircles::RootForCircles_2_2, 589
- AlgebraicKernelForCircles::SignAt, 598
- AlgebraicKernelForCircles::Solve, 601
- AlgebraicKernelForCircles::XCriticalPoints, 602
- AlgebraicKernelForCircles::YCriticalPoints, 603
- all_furthest_neighbors, 2303
- all_cells_begin
 - Triangulation_3, 1517
- all_cells_end

- Triangulation_3, 1517
- All_cells_iterator, 1505
- All_Delaunay_edges_iterator, 1770
- all_edges_begin
 - Apollonius_graph_2, 1715
 - DelaunayGraph_2, 1766
 - Segment_Delaunay_graph_2, 1663
 - Triangulation_2, 1437
 - Triangulation_3, 1517
- all_edges_end
 - Apollonius_graph_2, 1715
 - DelaunayGraph_2, 1766
 - Segment_Delaunay_graph_2, 1663
 - Triangulation_2, 1438
 - Triangulation_3, 1517
- All_edges_iterator, 1430, 1505, 1662, 1714
- all_faces_begin
 - Apollonius_graph_2, 1715
 - DelaunayGraph_2, 1766
 - Segment_Delaunay_graph_2, 1664
 - Triangulation_2, 1438
- all_faces_end
 - Apollonius_graph_2, 1715
 - DelaunayGraph_2, 1766
 - Segment_Delaunay_graph_2, 1664
 - Triangulation_2, 1438
- All_faces_iterator, 1430, 1662, 1714
- all_facets
 - Convex_hull_d, 692
- all_facets_begin
 - Triangulation_3, 1517
- all_facets_end
 - Triangulation_3, 1517
- All_facets_iterator, 1505
- all_furthest_neighbors_2, 2303–2304
- all_points
 - Convex_hull_d, 692
 - Delaunay_d, 704
- all_simplices
 - Convex_hull_d, 692
 - Delaunay_d, 703
- all_vertices
 - Convex_hull_d, 692
 - Delaunay_d, 703
- all_vertices_begin
 - Apollonius_graph_2, 1715
 - DelaunayGraph_2, 1766
 - Regular_triangulation_2, 1416
 - Segment_Delaunay_graph_2, 1663
 - Triangulation_2, 1437
 - Triangulation_3, 1516
- all_vertices_end
 - Apollonius_graph_2, 1715
 - DelaunayGraph_2, 1766

- Regular_triangulation_2*, 1416
- Segment_Delaunay_graph_2*, 1663
- Triangulation_2*, 1437
- Triangulation_3*, 1516
- All_vertices_iterator*, 1430, 1505, 1662, 1714
- AllFurthestNeighborsTraits_2*, 2305
- alpha*
 - Arr_circle_segment_traits_2<Kernel>*
 - ::CoordNT*, 1272
- alpha_begin*
 - Alpha_shape_2*, 1608
 - Alpha_shape_3*, 1637
- alpha_end*
 - Alpha_shape_2*, 1608
 - Alpha_shape_3*, 1637
- alpha_find*
 - Alpha_shape_2*, 1608
 - Alpha_shape_3*, 1637
- alpha_lower_bound*
 - Alpha_shape_2*, 1609
 - Alpha_shape_3*, 1637
- alpha_max*
 - Alpha_status*, 1641
- alpha_mid*
 - Alpha_status*, 1641
- alpha_min*
 - Alpha_status*, 1641
- Alpha_shape_2*, 1605–1610
- Alpha_shape_3*, 1633–1638
- Alpha_shape_cell_base_3*, 1639
- alpha_shape_edges_begin*
 - Alpha_shape_2*, 1607
- alpha_shape_edges_end*
 - Alpha_shape_2*, 1607
- Alpha_shape_face_base_2*, 1613
- Alpha_shape_vertex_base_2*, 1617
- Alpha_shape_vertex_base_3*, 1640
- alpha_shape_vertices_begin*
 - Alpha_shape_2*, 1607
- alpha_shape_vertices_end*
 - Alpha_shape_2*, 1607
- Alpha_status*, 1641–1642
- alpha_upper_bound*
 - Alpha_shape_2*, 1609
 - Alpha_shape_3*, 1637
- AlphaShapeCell_3*, 1629–1630
- AlphaShapeFace_2*, 1611–1612
- AlphaShapeTraits_2*, 1614
- AlphaShapeTraits_3*, 1631
- AlphaShapeVertex_2*, 1616
- AlphaShapeVertex_3*, 1632
- ambient_dimension*
 - Min_annulus_d*, 2245
 - Min_sphere_d*, 2239
- Polytope_distance_d*, 2315
- Angle*, 123
- angle*, 135
- angle_2_object*
 - ConformingDelaunayTriangulationTraits_2*, 1802
- annulus*
 - smallest enclosing, 2244
 - see also smallest enclosing sphere
- antipode*
 - Sphere_point*, 997
- Apollonius_graph_2*, 1713–1719
- Apollonius_graph_adaptation_traits_2*, 1772
- Apollonius_graph_caching_degeneracy_removal_policy_2*, 1781
- Apollonius_graph_degeneracy_removal_policy_2*, 1777
- Apollonius_graph_filtered_traits_2*, 1732
- Apollonius_graph_hierarchy_2*, 1733–1735
- Apollonius_graph_hierarchy_vertex_base_2*, 1738
- Apollonius_graph_traits_2*, 1731
- Apollonius_graph_vertex_base_2*, 1726
- Apollonius_site_2*, 1721
- ApolloniusGraphDataStructure_2*, 1722–1723
- ApolloniusGraphHierarchyVertexBase_2*, 1736–1737
- ApolloniusGraphTraits_2*, 1727–1730
- ApolloniusGraphVertexBase_2*, 1724–1725
- ApolloniusSite_2*, 1720
- approx*
 - Lazy_exact_nt*, 2557
- approx_convex_partition_2*, 736, 740–742
- postconditions, 737, 765
- traits class, 745
 - default, 769
- approximate_2_object*
 - ArrangementLandmarkTraits_2*, 1260
- approximate_division*, 2568
- Approximate_min_ellipsoid_d*, 2265–2273
 - creation, 2267
 - implementation, 2270
 - member functions, 2267–2270
 - access, 2267
 - miscellaneous, 2270
 - predicates, 2269
 - validity check, 2270
 - requirements, 2266
 - traits class
 - see also *Approximate_min_ellipsoid_d_traits_2*
 - see also *Approximate_min_ellipsoid_d_traits_3*
 - see also *Approximate_min_ellipsoid_d_traits_d*

- types, 2266
- Approximate_min_ellipsoid_d_traits_2*, 2276
- Approximate_min_ellipsoid_d_traits_3*, 2277
- Approximate_min_ellipsoid_d_traits_d*, 2278
 - traits class
 - requirements, 2275
- approximate_sqrt*, 2568
- ApproximateMinEllipsoid_d_Traits_d*, 2274–2275
- are_corners_parameterized*
 - Parameterization_polyhedron_adaptor_3*, 1977
 - ParameterizationPatchableMesh_3*, 1963
- are_equal*
 - Triangulation_3*, 1510
 - TriangulationDataStructure_3*, 1576
- are_mergeable_2_object*
 - ArrangementXMonotoneTraits_2*, 1262
- are_ordered_along_line*, 136
- are_parallel_2_object*
 - SegmentDelaunayGraphTraits_2*, 1686
- are_strictly_ordered_along_line*, 137
- are_strictly_ordered_along_line_2_object*
 - OptimalConvexPartitionTraits_2*, 760
- are_there_incident_constraints*
 - Constrained_triangulation_2*, 1390
- area*, 138
 - Iso_rectangle_2*, 70
 - Polygon_2*, 729
 - Triangle_2*, 84
- area_2*, 715
- Arity_tag*, 2658
- Arity_traits*, 2659
- Arr_accessor*, 1210–1215
- Arr_circle_segment_traits_2*, 1271–1276
- Arr_circle_segment_traits_2<Kernel>::CoordNT*, 1271–1272
- Arr_circle_segment_traits_2<Kernel>::Curve_2*, 1273–1275
- Arr_circle_segment_traits_2<Kernel>::Point_2*, 1272–1273
- Arr_circle_segment_traits_2<Kernel>::X_monotone_curve_2*, 1275–1276
- Arr_circular_arc_traits*, 604
- Arr_circular_line_arc_traits*, 606
- Arr_conic_traits_2*, 1277–1282
- Arr_conic_traits_2<RatKernel, AlgKernel, NtTraits>::X_monotone_curve_2*, 1282
- Arr_conic_traits_2<RatKernel, AlgKernel, NtTraits>::Curve_2*, 1278–1281
- Arr_consolidated_curve_data_traits_2*, 1289–1291
- Arr_consolidated_curve_data_traits_2<Traits, Data>::Data_container*, 1290–1291
- Arr_curve_data_traits_2*, 1286–1288
- Arr_curve_data_traits_2<Tr, XData, Mrg, CData, Cnv>::Curve_2*, 1287
- Arr_curve_data_traits_2<Tr, XData, Mrg, CData, Cnv>::X_monotone_curve_2*, 1288
- Arr_dcel_base*, 1248–1249
- Arr_default_dcel*, 1250
- Arr_default_overlay_traits*, 1232
- Arr_extended_dcel*, 1252
- Arr_extended_dcel_text_formatter*, 1302
- Arr_extended_face*, 1255
- Arr_extended_halfedge*, 1254
- Arr_extended_vertex*, 1253
- Arr_face_base*, 1248–1249
- Arr_face_extended_dcel*, 1251
- Arr_face_extended_text_formatter*, 1301
- Arr_face_overlay_traits*, 1233
- Arr_halfedge_base<Curve>*, 1248
- Arr_landmarks_point_location*, 1310
- Arr_line_arc_traits*, 605
- Arr_naive_point_location*, 1307
- Arr_non_caching_segment_basic_traits_2*, 1266
- Arr_non_caching_segment_traits_2*, 1267
- Arr_observer*, 1312–1317
- Arr_polyline_traits_2*, 1268–1270
- Arr_polyline_traits_2<SegmentTraits>::Curve_2*, 1268–1270
- Arr_polyline_traits_2<SegmentTraits>::X_monotone_curve_2*, 1270
- Arr_rational_arc_traits_2*, 1283–1285
- Arr_rational_arc_traits_2<AlgKernel, NtTraits>::Curve_2*, 1283–1285
- Arr_segment_traits_2*, 1265
- Arr_text_formatter*, 1300
- Arr_trapezoid_ric_point_location*, 1309
- Arr_vertex_base<Point>*, 1248
- Arr_walk_along_line_point_location*, 1308
- Arr_with_history_text_formatter*, 1327
- arrangement*
 - General_polygon_set_2*, 918
- Arrangement_2*, 1201–1209
- arrangement_type_2_object*
 - SegmentDelaunayGraphTraits_2*, 1686
- Arrangement_with_history_2*, 1318–1321
- ArrangementBasicTraits_2*, 1256–1258
- ArrangementDcel*, 1236–1238
- ArrangementDcelFace*, 1244–1245
- ArrangementDcelHalfedge*, 1241–1243
- ArrangementDcelHole*, 1246
- ArrangementDcelIsolatedVertex*, 1247
- ArrangementDcelVertex*, 1239–1240
- ArrangementDirectionalXMonotoneTraits_2*, 928–929

- ArrangementInputFormatter, 1292–1295
- ArrangementLandmarkTraits_2, 1259–1260
- ArrangementOutputFormatter, 1296–1299
- ArrangementPointLocation_2, 1303–1304
- ArrangementTraits_2, 1263–1264
- ArrangementVerticalRayShoot_2, 1305–1306
- ArrangementXMonotoneTraits_2, 1261–1262
- ArrWithHistoryInputFormatter, 1323–1324
- ArrWithHistoryOutputFormatter, 1325–1326
- ASCII, 2785, 2798
- aspect_ratio
 - Fair, 2058
 - Sliding_fair, 2099
 - Splitter, 2105
- Assert_bidirectional_category, 2712
- Assert_circulator, 2712
- Assert_circulator_or_iterator, 2712
- Assert_forward_category, 2712
- Assert_input_category, 2712
- Assert_iterator, 2712
- Assert_output_category, 2712
- Assert_random_access_category, 2712
- assertion flags
 - convex hull, 2D, 613
 - convex hull, 3D, 661
 - polygon, 713
 - polygon partitioning, 737
- assign, 120
 - Arr_extended_face, 1255
 - Arr_extended_halfedge, 1254
 - Arr_extended_vertex, 1253
 - Arrangement_2, 1203
 - Arrangement_with_history_2, 1319
 - ArrangementDcel, 1236
 - ArrangementDcelFace, 1244
 - ArrangementDcelHalfedge, 1241
 - ArrangementDcelVertex, 1239
 - Compact_container, 2612
- assign_2_object
 - ApolloniusGraphTraits_2, 1730
 - SegmentDelaunayGraphTraits_2, 1686
- associated_point
 - Convex_hull_d, 688
 - Delaunay_d, 702
- at
 - Kinetic::ActiveObjectsTable, 2478
- attach
 - Arr_observer, 1313
 - ArrangementPointLocation_2, 1304
 - ArrangementVerticalRayShoot_2, 1306
 - Qt_widget, 2844
- axes_lengths_begin
 - Approximate_min_ellipsoid_d, 2268
- axes_lengths_end
 - Approximate_min_ellipsoid_d, 2269
- axis_direction_cartesian_begin
 - Approximate_min_ellipsoid_d, 2269
- axis_direction_cartesian_end
 - Approximate_min_ellipsoid_d, 2269
- b
 - Line_2, 73
 - Plane_3, 100
- back
 - Arr_consolidated_curve_data_traits_2<Traits, Data>::Data_container, 1290
 - In_place_list, 2604
 - Qt_widget, 2842
 - Qt_widget_standard_toolbar, 2864
- backgroundColor
 - Qt_widget, 2844
- backward
 - Qt_widget_history, 2867
- backwardAvaillable
 - Qt_widget_history, 2867
- BAD, 1811
- Bare_point, 1412, 1528, 1542
- barycenter, 2342–2343
- Barycentric_mapping_parameterizer_3, 1934–1935
- Base, 1695
- base1
 - Plane_3, 100
- base2
 - Plane_3, 100
- Base_curve_2, 1286, 1289
- Base_traits_2, 1286, 1289
- Base_x_monotone_curve_2, 1286, 1289
- BasicMatrix, 2327
- bbox
 - Arr_circle_segment_traits_2<Kernel>::X_monotone_curve_2, 1276
 - Arr_conic_traits_2<RatKernel, AlgKernel, NtTraits>::Curve_2, 1281
 - Arr_polyline_traits_2<SegmentTraits>::Curve_2, 1269
 - Box_d, 2175
 - Box_with_handle_d, 2180
 - Circle_2, 66
 - Circular_arc_point_2, 558
 - Iso_cuboid_3, 96
 - Iso_rectangle_2, 70
 - Point_2, 76
 - Point_3, 103
 - Polygon_2, 729
 - Segment_2, 82
 - Segment_3, 108
 - Sphere_3, 111
 - Tetrahedron_3, 113

- Triangle_2*, 84
- Triangle_3*, 114
- VectorField_2*, 2410
- Bbox_2*, 64
- bbox_2*, 716
- Bbox_3*, 88
- before_add_hole*
 - Arr_observer*, 1315
- before_add_isolated_vertex*
 - Arr_observer*, 1315
- before_assign*
 - Arr_observer*, 1313
- before_attach*
 - Arr_observer*, 1314
- before_clear*
 - Arr_observer*, 1313
- before_create_edge*
 - Arr_observer*, 1314
- before_create_vertex*
 - Arr_observer*, 1314
- before_detach*
 - Arr_observer*, 1314
- before_edge_flip*
 - Kinetic::DelaunayTriangulationVisitor3*, 2436
 - Kinetic::RegularTriangulationVisitor3*, 2453
- before_facet_flip*
 - Kinetic::DelaunayTriangulationVisitor3*, 2436
 - Kinetic::RegularTriangulationVisitor3*, 2453
- before_flip*
 - Kinetic::DelaunayTriangulationVisitor2*, 2435
- before_global_change*
 - Arr_observer*, 1313
- before_merge_edge*
 - Arr_observer*, 1316
- before_merge_face*
 - Arr_observer*, 1316
- before_merge_hole*
 - Arr_observer*, 1316
- before_modify_edge*
 - Arr_observer*, 1314
- before_modify_vertex*
 - Arr_observer*, 1314
- before_move_hole*
 - Arr_observer*, 1316
- before_move_isolated_vertex*
 - Arr_observer*, 1317
- before_remove_edge*
 - Arr_observer*, 1317
- before_remove_hole*
 - Arr_observer*, 1317
- before_remove_vertex*
- Arr_observer*, 1317
- before_split_edge*
 - Arr_observer*, 1315
- before_split_face*
 - Arr_observer*, 1315
- before_split_hole*
 - Arr_observer*, 1315
- before_swap*
 - Kinetic::SortVisitor*, 2459
- begin*
 - Arr_consolidated_curve_data_traits_2<Traits, Data>::Data_container*, 1290
 - Arr_polyline_traits_2<SegmentTraits>::Curve_2*, 1269
 - Compact_container*, 2611
 - Container_from_circulator*, 2726
 - In_place_list*, 2604
 - Incremental_neighbor_search*, 2066
 - Interval_skip_list*, 2032
 - K_neighbor_search*, 2068
 - Kd_tree*, 2070
 - Kd_tree_node*, 2073
 - Kinetic::Delaunay_triangulation_recent_edges_visitor_2*, 2434
 - Largest_empty_iso_rectangle_2*, 2299
 - Matrix*, 444
 - Multiset*, 2616
 - Orthogonal_incremental_neighbor_search*, 2085
 - Orthogonal_k_neighbor_search*, 2087
 - SpatialTree*, 2103
 - Stream_lines_2*, 2408
 - Union_find*, 2781
 - Vector*, 440
- begin_facet*
 - Polyhedron_incremental_builder_3*, 819
- begin_surface*
 - Polyhedron_incremental_builder_3*, 819
- beta*
 - Arr_circle_segment_traits_2<Kernel>::CoordNT*, 1272
- Bidirectional_circulator*, 2715
- Bidirectional_circulator_base*, 2722
- Bidirectional_circulator_ptrbase*, 2723
- Bidirectional_circulator_tag*, 2722
- BINARY*, 2785, 2798
- Bind*, 2652
- bind_1*, 2641
- bind_2*, 2642
- bind_3*, 2643
- bind_4*, 2644
- bind_5*, 2645
- bisector*, 139
- border_edges_begin*

- Polyhedron_3*, 809
- border_halfedges_begin*
 - HalfedgeDS, 853
 - Polyhedron_3*, 809
- border_node*
 - CatmullClark_mask_3*, 1890
 - Loop_mask_3*, 1892
 - PQMask_3*, 1886
 - PTQMask_3*, 1887
- BorderParameterizer_3*, 1936
- bottom_vertex*
 - Polygon_2*, 729
- bottom_vertex_2*, 717
 - requirements, 715, 717–720, 722, 723, 732
- BOTTOMFRAME*, 973
- BOUNDARY*, 1847
- Boundary*, 960, 992, 1035
- boundary*
 - Nef_polyhedron_2*, 961
 - Nef_polyhedron_3*, 1036
 - Nef_polyhedron_S2*, 993
- boundary_edges_begin*
 - SurfaceMeshComplex_2InTriangulation_3*, 1850
- boundary_edges_end*
 - SurfaceMeshComplex_2InTriangulation_3*, 1850
- bounded_face*
 - Voronoi_diagram_2*, 1753
- bounded_faces_begin*
 - Voronoi_diagram_2*, 1754
- bounded_faces_end*
 - Voronoi_diagram_2*, 1754
- bounded_halfedge*
 - Voronoi_diagram_2*, 1754
- bounded_halfedges_begin*
 - Voronoi_diagram_2*, 1754
- bounded_halfedges_end*
 - Voronoi_diagram_2*, 1754
- Bounded_side*, 123
- bounded_side*
 - Circle_2*, 66
 - Convex_hull_d*, 690
 - Iso_box_d*, 472
 - Iso_cuboid_3*, 95
 - Iso_rectangle_2*, 70
 - Min_annulus_d*, 2247
 - Min_circle_2*, 2195
 - Min_ellipse_2*, 2205
 - Min_sphere_d*, 2240
 - Polygon_2*, 729
 - Sphere_3*, 110
 - Sphere_d*, 470
 - Tetrahedron_3*, 112
 - Triangle_2*, 83
- bounded_side_2*, 718
- bounding volumes
 - approximate smallest enclosing ellipsoid, 2265
 - smallest enclosing annulus, 2244
 - smallest enclosing circle, 2193
 - smallest enclosing ellipse, 2203
 - smallest enclosing sphere, 2238
 - smallest enclosing sphere of spheres, 2251
- bounding_box*, 2344
 - Kd_tree*, 2070
 - SpatialTree*, 2104
- BOX*, 2841
- Box_d*, 2174–2176
- Box_d::dimension*, 2175
- box_intersection_all_pairs_d*, 2164–2165
- box_intersection_d*, 2148, 2149, 2161–2163
- box_self_intersection_all_pairs_d*, 2169–2170
- box_self_intersection_d*, 2166–2168
- Box_traits_d*, 2177
- Box_with_handle_d*, 2178–2180
- Box_with_handle_d::dimension*, 2180
- BoxIntersectionBox_d*, 2171
- BoxIntersectionBox_d::dimension*, 2171
- BoxIntersectionTraits_d*, 2172–2173
- BoxIntersectionTraits_d::dimension*, 2172
- BoxIntersectionTraits_d::id*, 2172
- BoxIntersectionTraits_d::max_coord*, 2172
- BoxIntersectionTraits_d::min_coord*, 2172
- bucket_size*
 - Fair*, 2058
 - Median_of_max_spread*, 2078
 - Median_of_rectangle*, 2079
 - Midpoint_of_max_spread*, 2080
 - Midpoint_of_rectangle*, 2081
 - Sliding_fair*, 2099
 - Sliding_midpoint*, 2101
 - Splitter*, 2105
- bytes*
 - HalfedgeDS, 850
 - Polyhedron_3*, 802
 - Union_find*, 2780
- bytes_reserved*
 - HalfedgeDS, 850
 - Polyhedron_3*, 802
- c*
 - Line_2*, 73
 - Plane_3*, 100
- cached_number_of_components*
 - SurfaceMeshVertexBase_3*, 1864
- cached_number_of_incident_facets*
 - SurfaceMeshVertexBase_3*, 1864

- capacity*
 - Compact_container*, 2612
- capacity_of_faces*
 - HalfedgeDS*, 850
- capacity_of_facets*
 - Polyhedron_3*, 802
- capacity_of_halfedges*
 - HalfedgeDS*, 850
 - Polyhedron_3*, 802
- capacity_of_vertices*
 - HalfedgeDS*, 850
 - Polyhedron_3*, 802
- Cartesian*, 41
- cartesian*
 - Aff_transformation_2*, 62
 - Aff_transformation_3*, 91
 - Kernel::Component_accessor_d*, 506
 - Point_2*, 76
 - Point_3*, 103
 - Point_d*, 448
 - Vector_2*, 86
 - Vector_3*, 117
 - Vector_d*, 452
- cartesian_begin*
 - ApproximateMinEllipsoid_d_Traits_d*, 2274
 - Point_2*, 76
 - Point_3*, 103
 - Point_d*, 448
 - Vector_d*, 453
- Cartesian_const_iterator*, 2095
- Cartesian_const_iterator_d*, 2091, 2093, 2097
- Cartesian_converter*, 42
- cartesian_end*
 - Point_2*, 76
 - Point_3*, 103
 - Point_d*, 448
 - Vector_d*, 453
- cartesian_to_homogeneous*, 43
- Cast_function_object*, 2671
- catenate*
 - Multiset*, 2619
- CatmullClark_mask_3*, 1890–1891
- CatmullClark_subdivision*, 1884
- ccb*
 - Face*, 1762
 - Halfedge*, 1217, 1759
- Ccb_halfedge_circulator*, 1210, 1312
- ccb_halfedges*
 - Voronoi_diagram_2*, 1755
- ccw*, 1597
 - Triangulation_2*, 1439
 - Triangulation_cw_ccw_2*, 1442
 - TriangulationDataStructure_2*, 1469
 - TriangulationDataStructure_2::Face*, 1475
- ccw_permute*
 - ConstrainedTriangulationFaceBase_2*, 1379
 - TriangulationDSFaceBase_2*, 1472
- CELL*, 1505, 1558
- Cell*, 1585–1586
- Cell*, 1504
- cell*
 - TriangulationDSVertexBase_3*, 1592
 - Vertex*, 1587
- Cell_circulator*, 1505
- Cell_handle*, 1505, 1585, 1587, 1589, 1592, 1847
- cells_begin*
 - TriangulationDataStructure_3*, 1582
- cells_end*
 - TriangulationDataStructure_3*, 1582
- center*
 - rectangular, 2229
- center*
 - Circle_2*, 65
 - Min_annulus_d*, 2246
 - Min_sphere_d*, 2239
 - Sphere_3*, 110
 - Sphere_d*, 470
- center_cartesian_begin*
 - Approximate_min_ellipsoid_d*, 2268
 - Min_sphere_of_spheres_d*, 2253
 - Min_sphere_of_spheres_d_traits_2*, 2260
 - Min_sphere_of_spheres_d_traits_3*, 2262
 - Min_sphere_of_spheres_d_traits_d*, 2264
 - MinSphereOfSpheresTraits*, 2258
- center_cartesian_end*
 - Approximate_min_ellipsoid_d*, 2268
 - Min_sphere_of_spheres_d*, 2253
- center_coordinates_begin*
 - Min_annulus_d*, 2247
- center_coordinates_end*
 - Min_annulus_d*, 2247
- center_of_sphere*, 479
- center_vertex*
 - Halfedge*, 1040
 - SFace*, 1049
- centroid*, 140, 2345
- CGAL_CH_CHECK_EXPENSIVE*, 613, 661
- CGAL_For_all*, 2728
- CGAL_For_all_backwards*, 2728
- CGAL_HALFEDGEDS_DEFAULT*, 872
- ch_aki_toussaint*, 616–617
- ch_bykat*, 618–619
- ch_e_point*, 622
- ch_eddy*, 610, 620–621
- ch_graham_andrew*, 610, 623–624

- ch_graham_andrew_scan*, 611, 625–626
- ch_jarvis*, 610, 627–628
- ch_jarvis_march*, 611, 629–630
- ch_melkman*, 610, 631–632
- ch_n_point*, 635
- ch_ns_point*, 634
- ch_nswe_point*, 633
- ch_s_point*, 636
- ch_w_point*, 638
- ch_we_point*, 637
- check_integrity_and_topological_planarity*
 - Nef_polyhedron_S2*, 994
 - Topological_explorer*, 969
- check_parameterize_postconditions*
 - Fixed_border_parameterizer_3*, 1949
 - LSCM_parameterizer_3*, 1953
- check_parameterize_preconditions*
 - Fixed_border_parameterizer_3*, 1948
 - LSCM_parameterizer_3*, 1952
- check_tag*, 426
- check_unconnected_vertices*
 - Polyhedron_incremental_builder_3*, 820
- ChullTraits*, 2306, 2310
- CIRCLE*, 2841
- Circle*, 2013
- circle*
 - smallest enclosing, 2193
 - see also smallest enclosing ellipse
 - see also smallest enclosing sphere
 - see also smallest enclosing sphere of spheres
- circle*, 2200
 - Min_circle_2*, 2195
 - SHalfedge*, 1004, 1046
 - SHalfloop*, 1005, 1047
- Circle_2*, 65–66, 2857
- Circular_arc_2*, 548, 554–555
- Circular_arc_point_2*, 548, 558–559
- Circular_border_arc_length_parameterizer_3*, 1937–1938
- Circular_border_parameterizer_3*, 1939–1940
- Circular_border_uniform_parameterizer_3*, 1941–1942
- Circular_kernel_2*, 548
- CircularKernel*, 545–547
- CircularKernel::CircularArc_2*, 551
- CircularKernel::CircularArcPoint_2*, 553
- CircularKernel::CompareX_2*, 570
- CircularKernel::CompareXY_2*, 572
- CircularKernel::CompareY_2*, 571
- CircularKernel::CompareYatX_2*, 573
- CircularKernel::CompareYtoRight_2*, 574
- CircularKernel::ConstructBbox_2*, 569
- CircularKernel::ConstructCircle_2*, 561
- CircularKernel::ConstructCircularArc_2*, 564
- CircularKernel::ConstructCircularArcPoint_2*, 562
- CircularKernel::ConstructCircularMaxVertex_2*, 566
- CircularKernel::ConstructCircularMinVertex_2*, 565
- CircularKernel::ConstructCircularSourceVertex_2*, 567
- CircularKernel::ConstructCircularTargetVertex_2*, 568
- CircularKernel::ConstructLine_2*, 560
- CircularKernel::ConstructLineArc_2*, 563
- CircularKernel::DoOverlap_2*, 580
- CircularKernel::Equal_2*, 578
- CircularKernel::GetEquation*, 585
- CircularKernel::HasOn_2*, 579
- CircularKernel::Intersect_2*, 576
- CircularKernel::InXRange_2*, 581
- CircularKernel::IsVertical_2*, 582
- CircularKernel::IsXMonotone_2*, 583
- CircularKernel::IsYMonotone_2*, 584
- CircularKernel::LineArc_2*, 552
- CircularKernel::MakeXMonotone_2*, 575
- CircularKernel::Split_2*, 577
- Circulator*, 2715–2717
- Circulator*, 2705–2707, 2726
- Circulator_base*, 2722
- circulator_distance*, 2713
- Circulator_from_container*, 2718–2719
- Circulator_from_iterator*, 2720–2721
- circulator_size*, 2714
- Circulator_tag*, 2722–2724
- Circulator_traits*, 2725
- circumcenter*, 141
 - Triangulation_2*, 1440
- Classification_type*, 1605, 1633
- classify*
 - Alpha_shape_2*, 1608
 - Alpha_shape_3*, 1635, 1636
- clear*
 - AdaptationPolicy_2*, 1771
 - Alpha_shape_2*, 1606
 - Alpha_shape_3*, 1634
 - Apollonius_graph_2*, 1719
 - Apollonius_graph_hierarchy_2*, 1735
 - Arr_consolidated_curve_data_traits_2<Traits, Data>::Data_container*, 1291
 - Arr_polyline_traits_2<SegmentTraits>::Curve_2*, 1270
 - Arrangement_2*, 1203
 - Arrangement_with_history_2*, 1320
 - Compact_container*, 2613
 - Convex_hull_d*, 690
 - Delaunay_d*, 702
 - DelaunayGraph_2*, 1767

- General_polygon_2*, 932
- General_polygon_set_2*, 918
- Geomview_stream*, 2819
- HalfedgeDS*, 852
- Interval_skip_list*, 2032
- Kd_tree*, 2070
- Kinetic::ActiveObjectsTable*, 2478
- Largest_empty_iso_rectangle_2*, 2300
- Min_annulus_d*, 2248
- Min_circle_2*, 2196
- Min_ellipse_2*, 2206
- Min_sphere_d*, 2254
- Min_sphere_of_spheres_d*, 2254
- Multiset*, 2617
- Nef_polyhedron_2*, 961
- Nef_polyhedron_3*, 1037
- Nef_polyhedron_S2*, 992
- Polygon_2*, 728
- Polyhedron_3*, 808
- Polytope_distance_d*, 2317
- Qt_widget*, 2842
- Qt_widget_history*, 2867
- Segment_Delaunay_graph_2*, 1668
- SurfaceMeshTriangulation_3*, 1858
- Triangulation_2*, 1431
- Triangulation_3*, 1506
- TriangulationDataStructure_2*, 1464
- TriangulationDataStructure_3*, 1574
- Union_find*, 2780
- Unique_hash_map*, 2783
- Voronoi_diagram_2*, 1757
- clear_hidden_sites_container*
 - ApolloniusGraphVertexBase_2*, 1725
- clear_history*
 - Qt_widget*, 2842
 - Qt_widget_standard_toolbar*, 2864
- clear_seeds*
 - Delaunay_mesher_2*, 1806
- CLOCKWISE*, 126
- close_tip*
 - HalfedgeDS_items_decorator*, 882
- closure*
 - Nef_polyhedron_2*, 961
 - Nef_polyhedron_3*, 1036
 - Nef_polyhedron_S2*, 993
- code optimization*, 7
- coefficient*
 - Hyperplane_d*, 466
- coefficients_begin*
 - Hyperplane_d*, 466
- coefficients_end*
 - Hyperplane_d*, 467
- COLLINEAR*, 127
- collinear*, 142
 - collinear_are_ordered_along_line*, 143
 - collinear_are_ordered_along_line_2_object*
 - OptimalConvexPartitionTraits_2*, 760
 - collinear_are_strictly_ordered_along_line*, 144
 - collinear_has_on*
 - Ray_2*, 80
 - Segment_2*, 82
- color*
 - Qt_widget*, 2844
- column*
 - Matrix*, 443
- column_begin*
 - Matrix*, 443
- column_dimension*
 - Matrix*, 443, 1954
 - Taucs_matrix*, 1993
- column_end*
 - Matrix*, 444
- common_endpoint*, 464
- comp*
 - tree_interval_traits*, 2141
 - tree_point_traits*, 2143
- Compact_container*, 2610–2613
- Compact_container_base*, 2608
- Compact_container_traits*, 2609
- Compare*, 2524
- compare*, 2523, 2546
- Compare_distance_2*, 2384
- compare_distance_2_object*
 - DelaunayTriangulationTraits_2*, 1400
- compare_distance_3_object*
 - DelaunayTriangulationTraits_3*, 1537
- compare_distance_to_point*, 145
- compare_endpoints_xy_2_object*
 - ArrangementDirectionalXMonotoneTraits_2*, 929
- compare_lexicographically*, 480
- compare_non_strictly*
 - Sorted_matrix_search_traits_adaptor*, 2332
 - SortedMatrixSearchTraits*, 2333
- compare_power_distance_2_object*
 - RegularTriangulationTraits_2*, 1409
- compare_power_distance_3_object*
 - Regular_triangulation_euclidean_traits_3*, 1544
- compare_signed_distance_to_line*, 146
- compare_signed_distance_to_plane*, 147
- compare_slopes*, 148
- compare_strictly*
 - Sorted_matrix_search_traits_adaptor*, 2332
 - SortedMatrixSearchTraits*, 2333
- Compare_to_less*, 2654
- compare_to_less*, 2646
- compare_weight_2_object*

- ApolloniusGraphTraits_2, 1730
- compare_x*, 149–150
 - ExtendedKernelTraits_2, 975
- Compare_x_2*, 2385
- compare_x_2_object*
 - ApolloniusGraphTraits_2, 1730
 - ArrangementBasicTraits_2, 1258
 - LargestEmptyIsoRectangleTraits_2, 2302
 - PartitionTraits_2, 766
 - PolygonTraits_2, 725
 - SegmentDelaunayGraphTraits_2, 1686
 - SnapRoundingTraits_2, 1345
 - Triangulation_euclidean_traits_xy_3, 1446
 - TriangulationTraits_2, 1425
- compare_x_at_y*, 153–154
- compare_x_at_y_2_object*
 - YMonotonePartitionTraits_2, 778
- compare_xy*, 151
 - ExtendedKernelTraits_2, 975
- compare_xy_2_object*
 - ArrangementBasicTraits_2, 1258
- compare_xyz*, 152
- compare_xyz_3_object*
 - TriangulationTraits_3, 1535
- compare_y*, 155–156
 - ExtendedKernelTraits_2, 975
- Compare_y_2*, 2385
- compare_y_2_object*
 - ApolloniusGraphTraits_2, 1730
 - LargestEmptyIsoRectangleTraits_2, 2302
 - PartitionTraits_2, 766
 - PolygonTraits_2, 725
 - SegmentDelaunayGraphTraits_2, 1686
 - SnapRoundingTraits_2, 1345
 - Triangulation_euclidean_traits_xy_3, 1446
 - TriangulationTraits_2, 1425
- compare_y_at_x*, 157–158
- compare_y_at_x_2_object*
 - ArrangementBasicTraits_2, 1258
- compare_y_at_x_left_2_object*
 - ArrangementBasicTraits_2, 1258
- compare_y_at_x_right_2_object*
 - ArrangementBasicTraits_2, 1258
- compare_yx*, 159
- compare_z*, 160
- Comparison_result*, 124
- complement*, 939–940
 - General_polygon_set_2, 919
 - Nef_polyhedron_2, 961
 - Nef_polyhedron_3, 1036
 - Nef_polyhedron_S2, 993
 - Sphere_segment, 998
- COMPLETE, 960, 992, 1035
- Compose, 2653
- compose, 2647–2648
- Compose_shared, 2655
- compose_shared, 2649
- Compute_area_2, 2384
- compute_area_2_object*
 - PolygonTraits_2, 725
- compute_critical_squared_radius_3_object*
 - Regular_triangulation_euclidean_traits_3, 1544
- compute_edge_length*
 - Circular_border_arc_length_parameterizer_3, 1937
 - Circular_border_parameterizer_3, 1940
 - Circular_border_uniform_parameterizer_3, 1941
 - Square_border_arc_length_parameterizer_3, 1987
 - Square_border_parameterizer_3, 1990
 - Square_border_uniform_parameterizer_3, 1991
- compute_min_k_gon*
 - Extremal_polygon_area_traits_2, 2293
 - Extremal_polygon_perimeter_traits_2, 2295
 - ExtremalPolygonTraits_2, 2296
- compute_outer_frame_margin*, 1098–1099
- compute_power_product_3_object*
 - Regular_triangulation_euclidean_traits_3, 1544
- compute_scalar_product_3_object*
 - ImplicitSurfaceTraits_3, 1836
- compute_squared_distance_2_object*
 - AllFurthestNeighborsTraits_2, 2305
 - ConformingDelaunayTriangulationTraits_2, 1802
 - ConstrainedTriangulationTraits_2, 1382
- compute_squared_distance_3_object*
 - ImplicitSurfaceTraits_3, 1836
- Compute_squared_distance_d*, 2373, 2382
- compute_squared_distance_d_object*
 - InterpolationTraits, 2371
- compute_squared_radius_2_object*
 - AlphaShapeTraits_2, 1614
- compute_squared_radius_3_object*
 - AlphaShapeTraits_3, 1631
 - ImplicitSurfaceTraits_3, 1836
 - WeightedAlphaShapeTraits_3, 1643
- compute_squared_radius_smallest_orthogonal_sphere_3_object*
 - Regular_triangulation_euclidean_traits_3, 1544
- compute_w_ij*
 - Barycentric_mapping_parameterizer_3, 1935
 - Discrete_authalic_parameterizer_3, 1944

Discrete_conformal_map_parameterizer_3, 1946
Fixed_border_parameterizer_3, 1949
Mean_value_coordinates_parameterizer_3, 1957
concentric spheres
 see also annulus
ConformingDelaunayTriangulationTraits_2, 1801–1802
Const_circular_from_container<C>, 2718
Const_oneset_iterator, 2628
ConstrainedDelaunayTriangulation_2, 1383–1387
ConstrainedTriangulation_2, 1388–1392
ConstrainedTriangulation_face_base_2, 1393
ConstrainedTriangulation_plus_2, 1394–1398
ConstrainedDelaunayTriangulationTraits_2, 1378
ConstrainedTriangulationFaceBase_2, 1379–1380
ConstrainedTriangulationTraits_2, 1381–1382
Constraint, 1389
constraints_begin
 ConstrainedTriangulation_plus_2, 1397
constraints_end
 ConstrainedTriangulation_plus_2, 1397
construct_Apollonius_site_2_object
 ApolloniusGraphTraits_2, 1730
construct_Apollonius_vertex_2_object
 ApolloniusGraphTraits_2, 1730
construct_bisector_2_object
 DelaunayTriangulationTraits_2, 1400
Construct_cartesian_const_iterator, 2095
Construct_Cartesian_const_iterator_d, 2097
Construct_cartesian_const_iterator_d, 2091, 2093
construct_center_3_object
 ImplicitSurfaceTraits_3, 1836
Construct_center_d, 2092, 2093
construct_circular_arc_2_object
 CircularKernel, 547
construct_circumcenter_2_object
 DelaunayMeshTraits_2, 1804
 DelaunayTriangulationTraits_2, 1400
 TriangulationTraits_2, 1425
construct_circumcenter_3_object
 DelaunayTriangulationTraits_3, 1537
construct_curves_2_object
 GeneralPolygonSetTraits_2, 931
construct_direction
 ExtendedKernelTraits_2, 975
construct_direction_2_object
 DelaunayTriangulationTraits_2, 1400
construct_initial_points_object
 SurfaceMeshTraits_3, 1854
construct_insert
 Compact_container, 2612
Construct_iso_box_d, 2091
construct_iso_rectangle_2_above_left_point_2_object
 Rectangular_p_center_default_traits_2, 2234
 RectangularPCenterTraits_2, 2237
construct_iso_rectangle_2_above_right_point_2_object
 Rectangular_p_center_default_traits_2, 2234
 RectangularPCenterTraits_2, 2237
construct_iso_rectangle_2_below_left_point_2_object
 Rectangular_p_center_default_traits_2, 2234
 RectangularPCenterTraits_2, 2237
construct_iso_rectangle_2_below_right_point_2_object
 Rectangular_p_center_default_traits_2, 2234
 RectangularPCenterTraits_2, 2237
construct_iso_rectangle_2_object
 Rectangular_p_center_default_traits_2, 2234
 RectangularPCenterTraits_2, 2236
 SnapRoundingTraits_2, 1345
construct_line_2_object
 ConstrainedTriangulationTraits_2, 1381
 YMonotonePartitionTraits_2, 778
construct_max_vertex_2_object
 ArrangementBasicTraits_2, 1258
Construct_max_vertex_d, 2092, 2094, 2096, 2097
construct_midpoint_2_object
 ConformingDelaunayTriangulationTraits_2, 1802
construct_midpoint_3_object
 ImplicitSurfaceTraits_3, 1836
construct_min_vertex_2_object
 ArrangementBasicTraits_2, 1258
Construct_min_vertex_d, 2092, 2094, 2096, 2097
Construct_null_matrix_d, 2382
construct_null_matrix_d_object
 GradientFittingTraits, 2381
construct_object_2_object
 ApolloniusGraphTraits_2, 1730
 SegmentDelaunayGraphTraits_2, 1686
construct_object_3_object
 DelaunayTriangulationTraits_3, 1537
 RegularTriangulationTraits_3, 1541
construct_offset_contours
 Polygon_offset_builder_2, 1095
construct_opposite_2_object
 ArrangementDirectionalXMonotoneTraits_2, 929
Construct_opposite_plane_3, 828, 830
construct_opposite_plane_3_object
 Polyhedron_traits_3, 828
 Polyhedron_traits_with_normals_3, 830
 PolyhedronTraits_3, 827

construct_opposite_point
 ExtendedKernelTraits_2, 974
Construct_outer_product_d, 2382
construct_outer_product_d_object
 GradientFittingTraits, 2381
construct_perpendicular_line_object
 DelaunayTriangulationTraits_3, 1537
 RegularTriangulationTraits_3, 1541
construct_plane_3_object
 DelaunayTriangulationTraits_3, 1537
 RegularTriangulationTraits_3, 1541
construct_point
 ExtendedKernelTraits_2, 973
construct_point_d_object
 Kernel_d, 500
 Optimisation_d_traits_2, 2280
 Optimisation_d_traits_3, 2282
 Optimisation_d_traits_d, 2284
 OptimisationDTraits, 2286
construct_point_on_3_object
 ImplicitSurfaceTraits_3, 1836
construct_polygon_2_object
 GeneralPolygonSetTraits_2, 931
construct_radical_axis_2_object
 RegularTriangulationTraits_2, 1409
Construct_ray_2, 2384
construct_ray_2_object
 DelaunayTriangulationTraits_2, 1400
 OptimalConvexPartitionTraits_2, 760
construct_ray_3_object
 DelaunayTriangulationTraits_3, 1537
 RegularTriangulationTraits_3, 1541
construct_scaled_vector_2_object
 ConformingDelaunayTriangulationTraits_2, 1802
construct_scaled_vector_3_object
 ImplicitSurfaceTraits_3, 1836
Construct_scaled_vector_d, 2373, 2382
construct_scaled_vector_d_object
 GradientFittingTraits, 2381
 InterpolationTraits, 2371
Construct_scaling_matrix_d, 2382
construct_segment
 ExtendedKernelTraits_2, 975
construct_segment_2_object
 OptimalConvexPartitionTraits_2, 760
 PolygonTraits_2, 725
 SnapRoundingTraits_2, 1345
 Triangulation_euclidean_traits_xy_3, 1446
 TriangulationTraits_2, 1425
construct_segment_3_object
 ImplicitSurfaceTraits_3, 1836
 TriangulationTraits_3, 1535
construct_site_2
 SegmentDelaunayGraphSite_2, 1669
construct_skeleton
 Straight_skeleton_builder_2, 1091
Construct_squared_radius_d, 2092, 2094
construct_storage_site_2
 SegmentDelaunayGraphStorageSite_2, 1673, 1674
Construct_sum_matrix_d, 2382
construct_sum_matrix_d_object
 GradientFittingTraits, 2381
construct_svd_vertex_2_object
 SegmentDelaunayGraphTraits_2, 1686
construct_tetrahedron_3_object
 TriangulationTraits_3, 1535
construct_translated_point_2_object
 ConformingDelaunayTriangulationTraits_2, 1802
construct_translated_point_3_object
 ImplicitSurfaceTraits_3, 1836
Construct_triangle_2, 2384
construct_triangle_2_object
 Triangulation_euclidean_traits_xy_3, 1446
 TriangulationTraits_2, 1425
construct_triangle_3_object
 TriangulationTraits_3, 1535
construct_vector_2_object
 ConformingDelaunayTriangulationTraits_2, 1802
 Kernel, 40
 Min_quadrilateral_default_traits_2, 2224
 MinQuadrilateralTraits_2, 2227
construct_vector_3_object
 ImplicitSurfaceTraits_3, 1836
Construct_vector_d, 2373, 2382
construct_vector_d_object
 GradientFittingTraits, 2381
 InterpolationTraits, 2371
construct_vertex_2_object
 Rectangular_p_center_default_traits_2, 2234
 RectangularPCenterTraits_2, 2236
 SnapRoundingTraits_2, 1345
construct_Voronoi_point_2_object
 AdaptationTraits_2, 1769
Construct_weighted_circumcenter_2, 2385
construct_weighted_circumcenter_2_object
 RegularTriangulationTraits_2, 1409
construct_weighted_circumcenter_3_object
 Regular_triangulation_euclidean_traits_3, 1544
 RegularTriangulationTraits_3, 1541
construct_x_monotone_curve_2_object
 ArrangementLandmarkTraits_2, 1260
ConstructFunction, 2484
Construction_kernel, 1691, 1693

- Construction_traits_method_tag*, 1691, 1693
- contained_in_affine_hull*, 481
- contained_in_boundary*
 - Nef_polyhedron_2*, 962
 - Nef_polyhedron_S2*, 994
- contained_in_linear_hull*, 482
- contained_in_simplex*, 483
- container*
 - Polygon_2*, 730
- Container_from_circulator*, 2726–2727
- contains*
 - Delaunay_d*, 702
 - ExtendedKernelTraits_2*, 976
 - Fuzzy_iso_box*, 2060
 - Fuzzy_sphere*, 2062
 - FuzzyQueryItem*, 2059
 - Interval*, 2033
 - Kinetic::Delaunay_triangulation_recent_edges_visitor_2*, 2434
 - Nef_polyhedron_2*, 962
 - Nef_polyhedron_S2*, 994
- contains_interval*
 - Interval*, 2033
- Content*, 960, 992, 1035
- Context*, 1394
- context*
 - Constrained_triangulation_plus_2*, 1397
- Context_iterator*, 1395
- contexts_begin*
 - Constrained_triangulation_plus_2*, 1397
- contexts_end*
 - Constrained_triangulation_plus_2*, 1397
- CONTINUE*, 8
- Convert*, 1287
- convert_to_Polyhedron*
 - Nef_polyhedron_3*, 1037
- convex hull*, 609, 613, 661, 681, 683
- convex hull*, 2D, 639–640
 - Akl-Toussaint algorithm, 616–617
 - assertion flags, 613
 - Bykat’s algorithm, 610
 - Bykat algorithm, 618–619
 - Eddy algorithm, 620–621
 - gift-wrapping, 627, 629
 - Graham-Andrew scan, 623–626
 - Jarvis march, 627, 629
 - Melkman algorithm, 631–632
 - of polyline or polygon, 610, 631–632
 - postcondition, 612
 - quickhull, 610, 621
 - traits class
 - default, 647
 - requirements, 641
 - see also Convex_hull_constructive_traits_2*
 - see also Convex_hull_projective_xy_traits_2*
 - see also Convex_hull_projective_xz_traits_2*
 - see also Convex_hull_projective_yz_traits_2*
 - see also Convex_hull_traits_2*
- convex hull*, 3D, 655–660, 664–668
 - assertion flags, 661
 - dynamic, 659
 - incremental, 657, 667–668
 - quickhull, 655, 664–666
 - static, 655
- convex hull*, dD, 681–682
- convex polygon*
 - function object, 753
- convex_hull_2*, 610, 639–640
- convex_hull_3*, 655, 664–666
 - postcondition, 656, 678
 - traits class
 - default, 676
- Convex_hull_constructive_traits*, 611
- Convex_hull_constructive_traits_2*, 643
- Convex_hull_d*, 687–693
 - traits class
 - default, 694
- convex_hull_d_to_polyhedron_3*, 693
- convex_hull_incremental_3*, 657, 667–668
- Convex_hull_projective_xy_traits_2*, 611, 644
- Convex_hull_projective_xz_traits_2*, 611, 645
- Convex_hull_projective_yz_traits_2*, 611, 646
- Convex_hull_traits_2*, 611, 647
- Convex_hull_traits_3*, 656
- convex_partition_is_valid_2*, 743–744
 - preconditions, 743
 - traits class, 745
 - default, 743
- ConvexHullPolyhedron_3*, 672–673
- ConvexHullPolyhedronFacet_3*, 669
- ConvexHullPolyhedronHalfedge_3*, 670
- ConvexHullPolyhedronVertex_3*, 671
- ConvexHullTraits_2*, 641–642
 - model, 643–647
- ConvexHullTraits_3*, 664, 674–675
- ConvexHullTraits_d*, 685–686, 688
- convexity checking, 2D, 612, 648–649
- ConvexPartitionIsValidTraits_2*, 740, 743, 745, 756
 - model, 769
- COPLANAR*, 128
- coplanar*, 161
- coplanar_orientation*, 162
- coplanar_orientation_3_object*
 - TriangulationTraits_3*, 1535
- coplanar_side_of_bounded_circle*, 163
- coplanar_side_of_bounded_circle_3_object*
 - DelaunayTriangulationTraits_3*, 1537

- copy_n*, 2623
- copy_parallelogram_vertices_2*
 - Min_quadrilateral_default_traits_2*, 2224
 - MinQuadrilateralTraits_2*, 2227
- copy_rectangle_vertices_2*
 - Min_quadrilateral_default_traits_2*, 2224
 - MinQuadrilateralTraits_2*, 2227
- copy_strip_lines_2*
 - Min_quadrilateral_default_traits_2*, 2224
 - MinQuadrilateralTraits_2*, 2227
- copy_tds*
 - TriangulationDataStructure_2*, 1464
 - TriangulationDataStructure_3*, 1574
- CORE::Expr*, 2525
- corner_node*
 - DooSabin_mask_3*, 1893
 - DQMask_3*, 1888
- count*
 - Multiset*, 2617
- count_facet_vertices*
 - Parameterization_mesh_patch_3*, 1970
 - Parameterization_polyhedron_adaptor_3*, 1976
 - ParameterizationMesh_3*, 1960
- count_mesh_facets*
 - Parameterization_mesh_patch_3*, 1970
 - Parameterization_polyhedron_adaptor_3*, 1975
 - ParameterizationMesh_3*, 1960
- count_mesh_halfedges*
 - Parameterization_mesh_patch_3*, 1970
 - Parameterization_polyhedron_adaptor_3*, 1975
 - ParameterizationMesh_3*, 1960
- count_mesh_vertices*
 - Parameterization_mesh_patch_3*, 1969
 - Parameterization_polyhedron_adaptor_3*, 1975
 - ParameterizationMesh_3*, 1959
- COUNTERCLOCKWISE*, 126
- counterclockwise_in_between*
 - Direction_2*, 68
- Counting_iterator*, 2629
- create_cell*
 - TriangulationDataStructure_3*, 1581
- create_cells*
 - Kinetic::DelaunayTriangulationVisitor3*, 2436
 - Kinetic::RegularTriangulationVisitor3*, 2453
- create_center_vertex*
 - HalfedgeDS_decorator*, 869
 - Polyhedron_3*, 805
- create_edge*
 - OverlayTraits*, 1231
- create_face*
 - OverlayTraits*, 1231
 - TriangulationDataStructure_2*, 1468, 1469
- create_faces*
 - Kinetic::DelaunayTriangulationVisitor2*, 2435
- create_loop*
 - HalfedgeDS_decorator*, 865
- create_segment*
 - HalfedgeDS_decorator*, 865
- create_vertex*
 - Arr_accessor*, 1212
 - Kinetic::DelaunayTriangulationVisitor2*, 2435
 - Kinetic::DelaunayTriangulationVisitor3*, 2436
 - Kinetic::RegularTriangulationVisitor3*, 2453
 - Kinetic::SortVisitor*, 2459
 - OverlayTraits*, 1230, 1231
 - TriangulationDataStructure_2*, 1468
 - TriangulationDataStructure_3*, 1580
- Creator_1*, 2681
- Creator_2*, 2682
- Creator_3*, 2683
- Creator_4*, 2684
- Creator_5*, 2685
- Creator_uniform_2*, 2686
- Creator_uniform_3*, 2687
- Creator_uniform_4*, 2688
- Creator_uniform_5*, 2689
- Creator_uniform_6*, 2690
- Creator_uniform_7*, 2691
- Creator_uniform_8*, 2692
- Creator_uniform_9*, 2693
- Creator_uniform_d*, 2694
- CROSS*, 2841
- cross_product*, 164
- crossing_site*
 - SegmentDelaunayGraphSite_2*, 1670
 - SegmentDelaunayGraphStorageSite_2*, 1675
- ctp*, 1395
- current_dimension*
 - Convex_hull_d*, 688
 - Delaunay_d*, 701
- current_event_number*
 - Kinetic::Simulator*, 2508
- current_time*
 - Kinetic::Simulator*, 2507
- current_time_nt*
 - Kinetic::Simulator*, 2507
- curve*
 - ArrangementDcelHalfedge*, 1242
 - Halfedge*, 1217
 - Curve_2*, 1319, 1323, 1325

Curve_data, 1287
curves_begin
 Arrangement_with_history_2, 1320
 GeneralPolygon_2, 924
curves_end
 Arrangement_with_history_2, 1320
 GeneralPolygon_2, 924
custom_redraw
 Qt_widget, 2846
Custom_zoom_layer, 2863
cutting_dimension
 Plane_separator, 2088
 SpatialSeparator, 2102
cutting_value
 Plane_separator, 2088
 SpatialSeparator, 2102
cw, 1597
 Triangulation_2, 1440
 Triangulation_cw_ccw_2, 1442
 TriangulationDataStructure_2, 1469
 TriangulationDataStructure_2::Face, 1475
cw_permute
 ConstrainedTriangulationFaceBase_2, 1379
 TriangulationDSFaceBase_2, 1472
cyclic_adj_pred
 SHalfedge, 1004, 1046
 Topological_explorer, 967
cyclic_adj_succ
 SHalfedge, 1004, 1046
 Topological_explorer, 966

d
 Plane_3, 100
d2_map, 705
d2_show, 693, 705
d3_surface_map, 693
data
 Arr_curve_data_traits_2<Tr,
 XData,Mrg,CData,Cnv>::Curve_2,
 1287
 Arr_curve_data_traits_2<Tr,
 XData,Mrg,CData,Cnv>::X_monotone_-
 curve_2, 1288
 Arr_extended_face, 1255
 Arr_extended_halfedge, 1254
 Arr_extended_vertex, 1253
Data_access, 2370
Data_structure, 1661, 1713
data_structure
 Apollonius_graph_2, 1714
 Segment_Delaunay_graph_2, 1663
Data_type, 2370
dD Kernel
 traits class
 see also Linear_algebraCd
 see also Linear_algebraHd
deactivate
 Qt_widget_layer, 2852
deactivated
 Qt_widget_layer, 2852
deactivating
 Qt_widget_layer, 2852
DEFAULT, 962
default_random, 2743
default_value
 Unique_hash_map, 2783
defining_contour_edge
 StraightSkeletonHalfedge_2, 1079
defining_contour_halfedges_begin
 StraightSkeletonVertex_2, 1077
defining_matrix
 Approximate_min_ellipsoid_d, 2268
defining_scalar
 Approximate_min_ellipsoid_d, 2268
defining_vector
 Approximate_min_ellipsoid_d, 2268
degeneracies
 adding to input, 2733
 Min_annulus_d, 2244, 2248
 Min_circle_2, 2193, 2196
 Min_ellipse_2, 2203, 2206
 Min_sphere_d, 2240
 Polytope_distance_d, 2314, 2317
DEGENERATE, 128
degree
 Triangulation_3, 1518
 TriangulationDataStructure_2, 1469
 TriangulationDataStructure_3, 1583
 Vertex, 1216, 1760
Delaunay triangulation, dD, 682, 700–706
Delaunay_d, 682, 700–706
Delaunay_edge, 1751, 1758, 1768, 1770
Delaunay_edge_circulator, 1770
Delaunay_face_handle, 1751, 1760, 1768, 1770
Delaunay_geom_traits, 1751
Delaunay_graph, 1751
Delaunay_mesh_criteria_2, 1808
Delaunay_mesh_face_base_2, 1809
Delaunay_mesh_size_criteria_2, 1810
Delaunay_mesher_2, 1805–1807
Delaunay_triangulation_2, 1401–1405
Delaunay_triangulation_3, 1520–1526
Delaunay_triangulation_adaptation_traits_2, 1773
Delaunay_triangulation_caching_degeneracy_-
 removal_policy_2, 1782
Delaunay_triangulation_degeneracy_removal_-
 policy_2, 1778
Delaunay_vertex_handle, 1751, 1758, 1760, 1762,
 1768, 1770

Delaunay_voronoi_kind, 700
DelaunayGraph_2, 1764–1767
DelaunayLiftedTraits_d, 695–697, 701
DelaunayMeshFaceBase_2, 1803
DelaunayMeshTraits_2, 1804
DelaunayTraits_d, 698–699, 701
DelaunayTriangulationTraits_2, 1399–1400
DelaunayTriangulationTraits_3, 1536–1538
delegate
 Polyhedron_3, 810
delete_cell
 TriangulationDataStructure_3, 1581
delete_cells
 TriangulationDataStructure_3, 1581
delete_edge
 ArrangementDcel, 1237
delete_event
 Kinetic::Simulator, 2507
delete_face
 ArrangementDcel, 1237
 TriangulationDataStructure_2, 1469
delete_hole
 ArrangementDcel, 1237
delete_isolated_vertex
 ArrangementDcel, 1237
delete_vertex
 ArrangementDcel, 1237
 TriangulationDataStructure_2, 1469
 TriangulationDataStructure_3, 1581
delete_vertices
 TriangulationDataStructure_3, 1581
delta
 Direction_2, 67
 Direction_3, 92
 Direction_d, 456
deltas_begin
 Direction_d, 456
deltas_end
 Direction_d, 456
denominator
 *Arr_rational_arc_traits_2<AlgKernel, Nt-
 Traits>::Curve_2*, 1284
 Gmpq, 2542
 Quotient, 2573
 Rational_traits, 2575
Dereference, 2669
destroy
 In_place_list, 2604
detach
 Arr_observer, 1313
 ArrangementPointLocation_2, 1304
 ArrangementVerticalRayShoot_2, 1306
 Qt_widget, 2844
determinant
 LinearAlgebraTraits_d, 437, 438
difference, 941–942
 General_polygon_set_2, 920, 921
 Nef_polyhedron_2, 961
 Nef_polyhedron_3, 1037
 Nef_polyhedron_S2, 993
difference_type, 1429, 1505, 2746, 2762–2770
dimension
 Aff_transformation_d, 475
 Apollonius_graph_2, 1714
 Approximate_min_ellipsoid_d, 2268
 ApproximateMinEllipsoid_d_Traits_d, 2274
 Convex_hull_d, 688
 Delaunay_d, 701
 DelaunayGraph_2, 1765
 Direction_d, 456
 Hyperplane_d, 466
 Kd_tree_rectangle, 2075
 Kernel::Component_accessor_d, 506
 Line_d, 459
 Matrix, 443
 Point_2, 76
 Point_3, 103
 Point_d, 448
 Ray_d, 460
 Segment_d, 462
 Segment_Delaunay_graph_2, 1663
 Sphere_d, 469
 SurfaceMeshTriangulation_3, 1858
 Taucs_vector, 2000
 Triangulation_2, 1431
 Triangulation_3, 1507
 TriangulationDataStructure_2, 1464
 TriangulationDataStructure_3, 1575
 TriangulationDSFaceBase_2, 1472
 Vector, 440, 2003
 Vector_2, 86
 Vector_3, 117
 Vector_d, 452
direction
 ArrangementDcelHalfedge, 1241
 Halfedge, 1217
 Line_2, 73
 Line_3, 98
 Line_d, 459
 Ray_2, 79
 Ray_3, 105
 Ray_d, 461
 Segment_2, 81
 Segment_3, 107
 Segment_d, 463
 Vector_2, 86
 Vector_3, 117
 Vector_d, 453

Direction_2, 67–68
Direction_3, 92–93
Direction_d, 455–457
direction_of_time
 Kinetic::Simulator, 2508
DISC, 2841
Discrete_authalic_parameterizer_3, 1943–1944
Discrete_conformal_map_parameterizer_3, 1945–1946
discriminant
 Min_sphere_of_spheres_d, 2253
Distance, 1412
distance, 432
 squared, 432
distance, 19
 of polytopes, 2314
 squared, 19
Div, 2527
div, 2526, 2528
do_curves_intersect, 1336
do_intersect, 165, 484, 943–944
 General_polygon_set_2, 922
do_overlap, 166
 Interval_nt, 2545
do_paint
 Qt_widget, 2843
does_simplex_intersect_dual_support_3_object
 Regular_triangulation_euclidean_traits_3, 1544
DooSabin_mask_3, 1893
DooSabin_subdivision, 1884
double_coefficients
 Min_ellipse_2_traits_2, 2210
down
 ApolloniusGraphHierarchyVertexBase_2, 1736
 Halfedge, 1759
 SegmentDelaunayGraphHierarchyVertexBase_2, 1696
 TriangulationHierarchyVertexBase_2, 1423
 TriangulationHierarchyVertexBase_3, 1549
DQQ, 1884
DQQMask_3, 1888
draw
 Qt_widget_layer, 2851
draw_dual
 Apollonius_graph_2, 1718
 Delaunay_triangulation_2, 1404
 Delaunay_triangulation_3, 1525
 Regular_triangulation_2, 1416
 Regular_triangulation_3, 1533
 Segment_Delaunay_graph_2, 1667
draw_dual_edge
 Apollonius_graph_2, 1718
 Segment_Delaunay_graph_2, 1667
draw_primal
 Apollonius_graph_2, 1718
draw_primal_edge
 Apollonius_graph_2, 1718
draw_skeleton
 Segment_Delaunay_graph_2, 1667
draw_triangles
 Geomview_stream, 2818
dual
 Apollonius_graph_2, 1718
 Delaunay_triangulation_2, 1404
 Delaunay_triangulation_3, 1525
 Face, 1762
 Halfedge, 1759
 Regular_triangulation_2, 1416
 Regular_triangulation_3, 1533
 SurfaceMeshTriangulation_3, 1858
 Vertex, 1760
 Voronoi_diagram_2, 1753
dx
 Direction_2, 67
 Direction_3, 92
dy
 Direction_2, 67
 Direction_3, 92
Dynamic_matrix, 2323–2324
dz
 Direction_3, 92
e0, 2695–2698
e1, 2695–2698
e2, 2696–2698
e3, 2697, 2698
e4, 2698
e5, 2698
ECI, 2859, 2861
EDGE, 1406, 1430, 1505, 1558
Edge, 1081, 1084, 1429, 1463, 1504, 1574, 1661, 1713, 1764, 1847, 1857
edge
 Polygon_2, 730
Edge_circulator, 1662, 1714
edge_node
 CatmullClark_mask_3, 1890
 Loop_mask_3, 1892
 PQQMask_3, 1886
 PTQMask_3, 1887
edge_rejector_object
 AdaptationPolicy_2, 1771
edges_begin
 Arrangement_2, 1203
 Polygon_2, 728
 Polyhedron_3, 802

- SurfaceMeshComplex_2InTriangulation_3, 1850
- TriangulationDataStructure_2, 1465
- TriangulationDataStructure_3, 1582
- Voronoi_diagram_2, 1754
- edges_circulator*
 - Polygon_2, 728
- edges_clear*
 - HalfedgeDS, 852
- edges_end*
 - Arrangement_2, 1203
 - Polygon_2, 728
 - Polyhedron_3, 802
 - SurfaceMeshComplex_2InTriangulation_3, 1850
 - TriangulationDataStructure_2, 1466
 - TriangulationDataStructure_3, 1582
 - Voronoi_diagram_2, 1754
- edges_erase*
 - HalfedgeDS, 851, 852
- edges_pop_back*
 - HalfedgeDS, 851
- edges_pop_front*
 - HalfedgeDS, 851
- edges_push_back*
 - HalfedgeDS, 851
- EdgeTriple, 1081, 1084
- ellipse
 - approximate smallest enclosing, 2265
 - smallest enclosing, 2203
 - see also smallest enclosing circle
- ellipse, 2212
 - Min_ellipse_2, 2205
- EMPTY, 960, 992, 1035
- empty
 - Compact_container, 2612
 - Delaunay_d, 702
 - In_place_list, 2604
 - Kinetic::EventQueue, 2485
 - Kinetic::RootStack, 2500
 - Multiset, 2616
 - Polyhedron_3, 802
- Emptyset_iterator, 2626
- enclosing_query
 - Segment_tree_d, 2133
 - Segment_tree_k, 2135
 - Tree_anchor, 2144
- end
 - Arr_consolidated_curve_data_traits_2<Traits, Data>::Data_container, 1290
 - Arr_polyline_traits_2<SegmentTraits>::Curve_2, 1269
 - Compact_container, 2611
 - Container_from_circulator, 2726
 - In_place_list, 2604
 - Incremental_neighbor_search, 2066
 - Interval_skip_list, 2032
 - K_neighbor_search, 2068
 - Kd_tree, 2070
 - Kd_tree_node, 2073
 - Kinetic::Delaunay_triangulation_recent_edges_visitor_2, 2434
 - Largest_empty_iso_rectangle_2, 2299
 - Matrix, 444
 - Multiset, 2616
 - Orthogonal_incremental_neighbor_search, 2085
 - Orthogonal_k_neighbor_search, 2087
 - SpatialTree, 2103
 - Stream_lines_2, 2408
 - Union_find, 2781
 - Vector, 440
- end_facet
 - Polyhedron_incremental_builder_3, 819
- end_surface
 - Polyhedron_incremental_builder_3, 820
- end_time
 - Kinetic::Simulator, 2507
- enter_contour
 - Straight_skeleton_builder_2, 1091
- enterEvent
 - Qt_widget_layer, 2851
- enum, 818
- enum_cast, 167
- EQUAL, 124
- equal_2_object
 - ArrangementBasicTraits_2, 1258
 - PolygonTraits_2, 724
 - SegmentDelaunayGraphTraits_2, 1686
- equal_as_sets, 1001
- equal_range
 - Multiset, 2618
- erase
 - Arr_consolidated_curve_data_traits_2<Traits, Data>::Data_container, 1291
 - Compact_container, 2613
 - In_place_list, 2605
 - Kinetic::ActiveObjectsTable, 2478
 - Kinetic::EventQueue, 2485
 - Multiset, 2617
 - Polygon_2, 728
- erase_center_vertex
 - HalfedgeDS_decorator, 869
 - Polyhedron_3, 805
- erase_connected_component
 - HalfedgeDS_decorator, 866
 - Polyhedron_3, 808
- erase_face

- HalfedgeDS_decorator*, 866
- erase_facet*
 - Polyhedron_3*, 807
- erase_hole*
 - ArrangementDcelFace*, 1245
- erase_isolated_vertex*
 - ArrangementDcelFace*, 1245
- error*
 - Polyhedron_incremental_builder_3*, 820
- ERROR_CANNOT_SOLVE_LINEAR_SYSTEM*, 1981, 1983
- Error_code*, 1981, 1983
- ERROR_EMPTY_MESH*, 1981, 1983
- ERROR_INVALID_BORDER*, 1981, 1983
- ERROR_NO_1_TO_1_MAPPING*, 1981, 1983
- ERROR_NO_SURFACE_MESH*, 1981, 1983
- ERROR_NON_TRIANGULAR_MESH*, 1981, 1983
- ERROR_NOT_ENOUGH_MEMORY*, 1981, 1983
- ERROR_WRONG_PARAMETER*, 1981, 1983
- Euclidean_distance*, 2054–2055
- Euclidean_distance_sphere_point*, 2056–2057
- Euclidean_ring_tag*, 2529
- EuclideanRingNumberType*, 2528
- Euler_integrator_2*, 2401
- Event*, 2487
- event*
 - Kinetic::Simulator*, 2508
 - Qt_widget_layer*, 2851
- event_time*
 - Kinetic::Simulator*, 2508
- events_begin*
 - Kinetic::EventLogVisitor*, 2443
- events_end*
 - Kinetic::EventLogVisitor*, 2443
- exact*
 - Filtered_exact*, 2532
 - Lazy_exact_nt*, 2557
- Exact_circular_kernel_2*, 549
- Exact_kernel*, 1691, 1693
- Exact_predicates_exact_constructions_kernel*, 57
- Exact_predicates_exact_constructions_kernel_*
with_sqrt, 58
- Exact_predicates_inexact_constructions_kernel*, 59
- Exact_traits_method_tag*, 1691, 1693
- EXCLUDED*, 960, 992, 1035
- EXIT*, 8
- EXIT_WITH_SUCCESS*, 8
- Explorer*, 970–971
- explorer*
 - Nef_polyhedron_2*, 963
- extend*
 - Box_d*, 2175
 - Box_with_handle_d*, 2179
- extended kernel*, 2D
 - traits class*
 - requirements*, 972
- Extended_cartesian*, 977
- Extended_homogeneous*, 978
- EXTENDED_INTERNAL*, 2072
- ExtendedKernelTraits_2*, 972–976
 - model*, 977–979
- EXTERIOR*, 1605, 1633
- extract_all_even_rows*
 - Dynamic_matrix*, 2323
 - MonotoneMatrixSearchTraits*, 2325
- extremal_polygon_2*, 2291
- Extremal_polygon_area_traits_2*, 2292–2293
- Extremal_polygon_perimeter_traits_2*, 2294–2295
- ExtremalPolygonTraits_2*, 2296–2297
- extreme point*, 609, 613, 661, 681, 683
- extreme points*, 2D
 - between two points*, 629
 - in coordinate directions*, 611, 622, 633–638
 - right of line*, 625–626
 - traits class*
 - default*, 647
 - requirements*, 641
- FACE*, 1430
- Face*, 822, 1218, 1429, 1662, 1713, 1762–1763
- face*
 - ApolloniusGraphVertexBase_2*, 1724
 - ArrangementDcelHalfedge*, 1242
 - ArrangementDcelHole*, 1246
 - ArrangementDcelIsolatedVertex*, 1247
 - Halfedge*, 1217, 1758
 - HalfedgeDSHalfedge*, 858
 - SegmentDelaunayGraphVertexBase_2*, 1681
 - Topological_explorer*, 967
 - TriangulationDataStructure_2::Vertex*, 1478
 - Vertex*, 1216
- Face_circulator*, 1662, 1714
- Face_const_handle*, 1296
- face_cycle*
 - Topological_explorer*, 968
- Face_handle*, 1210, 1292, 1312, 1430, 1471, 1476, 1662, 1713
- face_handle*
 - Level_interval*, 2035
- face_rejector_object*
 - AdaptationPolicy_2*, 1771
- Face_status*, 1847
- face_status*
 - SurfaceMeshComplex_2InTriangulation_3*, 1849
- faces_begin*
 - Arrangement_2*, 1204
 - ArrangementDcel*, 1237

- HalfedgeDS, 850
- Topological_explorer, 967
- TriangulationDataStructure_2, 1465
- Voronoi_diagram_2, 1754
- faces_clear
 - HalfedgeDS, 852
- faces_end
 - Arrangement_2, 1204
 - ArrangementDcel, 1237
 - HalfedgeDS, 850
 - Topological_explorer, 967
 - TriangulationDataStructure_2, 1465
 - Voronoi_diagram_2, 1754
- faces_erase
 - HalfedgeDS, 852
 - HalfedgeDS_decorator, 866
- faces_pop_back
 - HalfedgeDS, 852
 - HalfedgeDS_decorator, 866
- faces_pop_front
 - HalfedgeDS, 852
 - HalfedgeDS_decorator, 866
- faces_push_back
 - HalfedgeDS, 851
 - HalfedgeDS_decorator, 865
- faces_splice
 - HalfedgeDS_list, 886
- FACET, 1406, 1505, 1558
- Facet, 811–812, 1504, 1574, 1847, 1857
- facet
 - ConvexHullPolyhedronHalfedge_3, 670
 - Halfedge, 815
 - SHalfedge, 1046
 - SHalfloop, 1048
- facet_begin
 - ConvexHullPolyhedronFacet_3, 669
 - ConvexHullPolyhedronHalfedge_3, 670
 - Facet, 811, 812
 - Halfedge, 814
- Facet_circulator, 1505
- facet_cycle_begin
 - Halffacet, 1043
- facet_cycle_end
 - Halffacet, 1043
- facet_degree
 - Facet, 812
 - Halfedge, 815
- Facet_handle, 818
- facet_node
 - CatmullClark_mask_3, 1890
 - PQMask_3, 1886
 - Sqrt3_mask_3, 1894
 - Sqrt3Mask_3, 1889
- facet_vertices_begin
 - Parameterization_mesh_patch_3, 1970
 - Parameterization_polyhedron_adaptor_3, 1975, 1976
 - ParameterizationMesh_3, 1960
- facets_begin
 - Convex_hull_d, 691
 - Polyhedron_3, 802
 - SurfaceMeshComplex_2InTriangulation_3, 1850
 - TriangulationDataStructure_3, 1582
- facets_end
 - Convex_hull_d, 691
 - Polyhedron_3, 802
 - SurfaceMeshComplex_2InTriangulation_3, 1850
 - TriangulationDataStructure_3, 1582
- facets_visible_from
 - Convex_hull_d, 690
- Failure_behaviour, 8
- failure_time
 - Kinetic::Certificate, 2483
- Fair, 2058
- farin_c1_interpolation, 2368
- Field_tag, 2531
- FieldNumberType, 2530
- file_input
 - Apollonius_graph_2, 1719
 - Apollonius_graph_hierarchy_2, 1735
 - Segment_Delaunay_graph_2, 1668
 - Voronoi_diagram_2, 1756
- file_output
 - Apollonius_graph_2, 1719
 - Apollonius_graph_hierarchy_2, 1735
 - Segment_Delaunay_graph_2, 1668
 - Voronoi_diagram_2, 1756
- fill_hole
 - HalfedgeDS_decorator, 866, 867
 - Polyhedron_3, 807
- fillColor
 - Qt_widget, 2844
- Filter_iterator, 2632
- Filtered_exact, 2532–2533
- Filtered_extended_homogeneous, 979
- Filtered_kernel, 44
- Filtered_kernel_adaptor, 45
- Filtered_predicate, 46–47
- Filtering_kernel, 1691, 1693
- Filtering_traits_method_tag, 1691, 1693
- filtration
 - Alpha_shape_3, 1637
- find
 - Arr_consolidated_curve_data_traits_2<Traits, Data>::Data_container, 1290
 - Multiset, 2617

- Union_find*, 2781
- find_conflicts*
 - Delaunay_triangulation_3*, 1523, 1524
 - Regular_triangulation_3*, 1532
 - SurfaceMeshTriangulation_3*, 1858
- find_intervals*
 - Interval_skip_list*, 2032
- find_lower*
 - Multiset*, 2618
- find_optimal_alpha*
 - Alpha_shape_2*, 1609
 - Alpha_shape_3*, 1638
- find_prev*
 - HalfedgeDS_items_decorator*, 882
- find_prev_around_vertex*
 - HalfedgeDS_items_decorator*, 882
- finite_cells_begin*
 - Triangulation_3*, 1516
- finite_cells_end*
 - Triangulation_3*, 1516
- Finite_Delaunay_edges_iterator*, 1770
- finite_edge_interior_conflict_2_object*
 - ApolloniusGraphTraits_2*, 1730
 - SegmentDelaunayGraphTraits_2*, 1686
- finite_edges_begin*
 - Apollonius_graph_2*, 1715
 - DelaunayGraph_2*, 1765
 - Segment_Delaunay_graph_2*, 1663
 - SurfaceMeshTriangulation_3*, 1859
 - Triangulation_2*, 1437
 - Triangulation_3*, 1516
- finite_edges_end*
 - Apollonius_graph_2*, 1715
 - DelaunayGraph_2*, 1765
 - Segment_Delaunay_graph_2*, 1663
 - SurfaceMeshTriangulation_3*, 1859
 - Triangulation_2*, 1437
 - Triangulation_3*, 1516
- finite_faces_begin*
 - Apollonius_graph_2*, 1715
 - DelaunayGraph_2*, 1765
 - Segment_Delaunay_graph_2*, 1663
 - Triangulation_2*, 1437
- finite_faces_end*
 - Apollonius_graph_2*, 1715
 - DelaunayGraph_2*, 1765
 - Segment_Delaunay_graph_2*, 1663
 - Triangulation_2*, 1437
- finite_facets_begin*
 - SurfaceMeshTriangulation_3*, 1859
 - Triangulation_3*, 1516
- finite_facets_end*
 - SurfaceMeshTriangulation_3*, 1859
 - Triangulation_3*, 1516

- finite_vertex*
 - Apollonius_graph_2*, 1715
 - DelaunayGraph_2*, 1765
 - Segment_Delaunay_graph_2*, 1663
 - Triangulation_2*, 1431
- finite_vertices_begin*
 - Apollonius_graph_2*, 1715
 - DelaunayGraph_2*, 1765
 - Regular_triangulation_2*, 1416
 - Segment_Delaunay_graph_2*, 1663
 - SurfaceMeshTriangulation_3*, 1859
 - Triangulation_2*, 1437
 - Triangulation_3*, 1516
- finite_vertices_end*
 - Apollonius_graph_2*, 1715
 - DelaunayGraph_2*, 1765
 - Regular_triangulation_2*, 1416
 - Segment_Delaunay_graph_2*, 1663
 - SurfaceMeshTriangulation_3*, 1859
 - Triangulation_2*, 1437
 - Triangulation_3*, 1516
- first*, 2699, 2701
- first_out_edge*
 - Topological_explorer*, 966
- first_pair_closer_than_second*
 - ExtendedKernelTraits_2*, 976
- first_type*, 2699, 2701
- Fixed_border_parameterizer_3*, 1947–1950
- Fixed_precision_nt*, 2534–2536
- flip*
 - Constrained_Delaunay_triangulation_2*, 1386
 - Triangulation_2*, 1433
 - Triangulation_3*, 1513
 - TriangulationDataStructure_2*, 1466
 - TriangulationDataStructure_3*, 1577
- flip_edge*
 - HalfedgeDS_items_decorator*, 883
 - Polyhedron_3*, 805
- flip_flippable*
 - Triangulation_3*, 1513
 - TriangulationDataStructure_3*, 1577
- for_compact_container*
 - Compact_container_base*, 2608
 - TriangulationDSCellBase_3*, 1591
 - TriangulationDSFaceBase_2*, 1473
 - TriangulationDSVertexBase_2*, 1477
 - TriangulationDSVertexBase_3*, 1593
- forth*
 - Qt_widget*, 2842
- forward*
 - Qt_widget_history*, 2867
 - Qt_widget_standard_toolbar*, 2864
- Forward_circulator*, 2715
- Forward_circulator_base*, 2722

Forward_circulator_ptrbase, 2723
Forward_circulator_tag, 2722
forwardAvaillable
 Qt_widget_history, 2867
fourth, 2701
fourth_type, 2701
Fourtuple, 2697
FPU_CW_t, 2546
FPU_get_and_set_cw, 2547
FPU_get_cw, 2546
FPU_set_cw, 2546
 Frederickson/Johnson matrix search, 2328
front
 Arr_consolidated_curve_data_traits_2<Traits, Data>::Data_container, 1290
 In_place_list, 2604
FT, 41, 48, 55, 56, 548, 726, 1605, 1853, 2054, 2056, 2058, 2060, 2062, 2067, 2069, 2072, 2074, 2076, 2084, 2086, 2091, 2093, 2097, 2099, 2106, 2373, 2382, 2384, 2403, 2407, 2409, 2761, 2854–2857, 2859
 Function, 2491
function_kernel_object
 Kinetic::Kernel, 2494
 Kinetic::SimulationTraits, 2502
 Kinetic::Simulator, 2507
FURTHEST, 700
furthest
 all neighbors, 2303
Fuzzy_iso_box, 2060–2061
Fuzzy_sphere, 2062–2063
FuzzyQueryItem, 2059

gamma
 Arr_circle_segment_traits_2<Kernel>::CoordNT, 1272
Gcd, 2538
gcd, 2528, 2537
GENERAL, 1606, 1633
General_polygon_2, 932, 935
General_polygon_set_2, 917–923
General_polygon_with_holes_2, 935
GeneralDistance, 2064
GeneralPolygon_2, 924–925
GeneralPolygonSetTraits_2, 930–931
GeneralPolygonWithHoles_2, 926–927
generator
 2D point, 2733
 3D point, 2733
 convex set, 2752
 segment, 2736–2739
 simple polygon, 2754
 generator classes, requirements, 2750

Geom_traits, 1429, 1504, 1661, 1677, 1713, 1805, 2407
geom_traits
 Apollonius_graph_2, 1714
 DelaunayGraph_2, 1765
 Segment_Delaunay_graph_2, 1662
 SurfaceMeshTriangulation_3, 1858
 Triangulation_2, 1431
 Triangulation_3, 1507
Geomview_stream, 2816–2822
get
 Kinetic::EventQueue, 2485
get_a, 2312
 Width_default_traits_3, 2310
get_adapted_mesh
 Parameterization_polyhedron_adaptor_3, 1974
Get_address, 2670
get_all_build_directions
 Width_3, 2307
get_allocator
 Compact_container, 2612
 HalfedgeDS, 850
 In_place_list, 2604
 Polyhedron_3, 802
 Union_find, 2780
get_alpha
 Alpha_shape_2, 1607
 Alpha_shape_3, 1635
 AlphaShapeCell_3, 1629
 AlphaShapeFace_2, 1612
get_alpha_shape_cells
 Alpha_shape_3, 1636
get_alpha_shape_edges
 Alpha_shape_3, 1636
get_alpha_shape_facets
 Alpha_shape_3, 1636
get_alpha_shape_vertices
 Alpha_shape_3, 1637
get_alpha_status
 AlphaShapeVertex_3, 1632
get_ascii_mode
 Geomview_stream, 2821
get_b, 2312
 Width_default_traits_3, 2310
get_binary_mode
 Geomview_stream, 2821
get_bits
 Random, 2757
get_bool
 Random, 2757
get_border
 Parameterization_mesh_patch_3, 1970

- Parameterization_polyhedron_adaptor_3*, 1975
- ParameterizationMesh_3*, 1960
- get_border_parameterizer*
 - Fixed_border_parameterizer_3*, 1950
 - LSCM_parameterizer_3*, 1953
- get_borders*
 - Parameterization_mesh_feature_extractor*, 1965
- get_boundary_of_conflicts*
 - Constrained_Delaunay_triangulation_2*, 1386
 - Delaunay_triangulation_2*, 1403
 - Regular_triangulation_2*, 1415
- get_boundary_of_conflicts_and_hidden_vertices*
 - Regular_triangulation_2*, 1414
- get_bounding_box*
 - Largest_empty_iso_rectangle_2*, 2299
- get_build_direction*
 - Width_3*, 2307
- get_c*, 2312
 - Width_default_traits_3*, 2310
- get_coef*
 - Matrix*, 1954
 - Taucs_matrix*, 1994
- get_conflicts*
 - Constrained_Delaunay_triangulation_2*, 1386
 - Delaunay_triangulation_2*, 1403
 - Regular_triangulation_2*, 1415
- get_conflicts_and_boundary*
 - Constrained_Delaunay_triangulation_2*, 1385
 - Delaunay_triangulation_2*, 1403
 - Regular_triangulation_2*, 1414
- get_conflicts_and_boundary_and_hidden_vertices*
 - Regular_triangulation_2*, 1413
- get_conflicts_and_hidden_vertices*
 - Regular_triangulation_2*, 1414
- get_corners_index*
 - Parameterization_polyhedron_adaptor_3*, 1977
 - ParameterizationPatchableMesh_3*, 1963
- get_corners_tag*
 - Parameterization_polyhedron_adaptor_3*, 1977
 - ParameterizationPatchableMesh_3*, 1963
- get_corners_uv*
 - Parameterization_polyhedron_adaptor_3*, 1977
 - ParameterizationPatchableMesh_3*, 1963
- get_criteria*
 - Delaunay_mesher_2*, 1806
- get_d*, 2313
 - Width_default_traits_3*, 2310
- get_decorated_mesh*
 - Parameterization_mesh_patch_3*, 1969
- get_dimension*
 - Regular_grid_2*, 2403
- get_double*
 - Random*, 2757
- get_echo*
 - Geomview_stream*, 2821
- get_error_message*
 - Parameterizer_traits_3*, 1984
- get_face*
 - HalfedgeDS_items_decorator*, 882
- get_face_halfedge*
 - HalfedgeDS_items_decorator*, 882
- get_facet_status*
 - AlphaShapeCell_3*, 1629
- get_facet_surface_center*
 - SurfaceMeshCellBase_3*, 1845
- get_field*
 - VectorField_2*, 2410
- get_genus*
 - Parameterization_mesh_feature_extractor*, 1966
- get_halfedge*
 - Parameterization_polyhedron_adaptor_3*, 1974
- get_halfedge_seaming*
 - Parameterization_polyhedron_adaptor_3*, 1977
 - ParameterizationPatchableMesh_3*, 1963
- get_hidden_vertices*
 - Regular_triangulation_2*, 1415
- get_hw*, 2312
 - Width_default_traits_3*, 2310
- get_hx*, 2312
 - Width_default_traits_3*, 2310
- get_hy*, 2312
 - Width_default_traits_3*, 2310
- get_hz*, 2312
 - Width_default_traits_3*, 2310
- get_in_conflict_flag*
 - TriangulationDSCellBase_3*, 1591
- get_int*
 - Random*, 2757
- get_integration_step*
 - VectorField_2*, 2410
- get_intersection_points*, 1334
- get_key*
 - tree_point_traits*, 2143
- get_largest_empty_iso_rectangle*
 - Largest_empty_iso_rectangle_2*, 2299
- get_left*
 - tree_interval_traits*, 2141
 - tree_point_traits*, 2143
- get_left_bottom_right_top*
 - Largest_empty_iso_rectangle_2*, 2299

get_left_win
 tree_interval_traits, 2141
get_line_width
 Geomview_stream, 2820
get_linear_algebra_traits
 Fixed_border_parameterizer_3, 1950
 LSCM_parameterizer_3, 1953
get_longest_border
 Parameterization_mesh_feature_extractor,
 1966
get_mode, 2785, 2791
 Alpha_shape_2, 1607
 Alpha_shape_3, 1635
get_nb_borders
 Parameterization_mesh_feature_extractor,
 1965
get_nb_connex_components
 Parameterization_mesh_feature_extractor,
 1966
get_new_id
 Geomview_stream, 2820
get_nth_alpha
 Alpha_shape_2, 1607
 Alpha_shape_3, 1635
get_number_of_optimal_solutions
 Width_3, 2307
getPainter
 Qt_widget, 2846
get_pixmap
 Qt_widget, 2846
get_plane_coefficients, 2313
 Width_default_traits_3, 2310
get_point_coordinates, 2312
 Width_default_traits_3, 2310
get_prev
 HalfedgeDS_items_decorator, 882
get_range
 AlphaShapeVertex_2, 1616
get_ranges
 AlphaShapeFace_2, 1611
get_raw
 Geomview_stream, 2821
get_relative_precision_of_to_double
 Lazy_exact_nt, 2557
get_right
 tree_interval_traits, 2141
 tree_point_traits, 2143
get_right_win
 tree_interval_traits, 2141
get_saturation_ratio
 Stream_lines_2, 2408
get_separating_distance
 Stream_lines_2, 2408
get_size
 Regular_grid_2, 2403
get_sphere_map
 Nef_polyhedron_3, 1036
get_squared_width
 Width_3, 2307
get_subcurves, 1335
get_taucs_matrix
 Taucs_matrix, 1994
get_taucs_vector
 Taucs_vector, 2000
get_trace
 Geomview_stream, 2821
get_traits
 Arrangement_2, 1203
get_vertex
 HalfedgeDS_items_decorator, 882
get_vertex_halfedge
 HalfedgeDS_items_decorator, 882
get_vertex_index
 Parameterization_mesh_patch_3, 1971
 Parameterization_polyhedron_adaptor_3,
 1976
 ParameterizationMesh_3, 1961
get_vertex_position
 Parameterization_mesh_patch_3, 1970
 Parameterization_polyhedron_adaptor_3,
 1976
 ParameterizationMesh_3, 1960
get_vertex_radius
 Geomview_stream, 2820
get_vertex_seaming
 Parameterization_polyhedron_adaptor_3,
 1976
 ParameterizationPatchableMesh_3, 1962
get_vertex_tag
 Parameterization_mesh_patch_3, 1971
 Parameterization_polyhedron_adaptor_3,
 1976
 ParameterizationMesh_3, 1961
get_vertex_uv
 Parameterization_mesh_patch_3, 1970
 Parameterization_polyhedron_adaptor_3,
 1976
 ParameterizationMesh_3, 1960
get_width_coefficients
 Width_3, 2307
get_width_planes
 Width_3, 2307
get_wired
 Geomview_stream, 2820
Gmpq, 2541–2542
Gmpz, 2543
Gps_circle_segment_traits_2, 937
Gps_segment_traits_2, 936

- Gps_traits_2*, 938
- GradientFittingTraits, 2378, 2380–2381
 - model, 2382
- Gray_level_image_3*, 1831
- greene_approx_convex_partition_2*, 736, 746–748
 - postconditions, 737, 765
 - traits class, 745
 - default, 769
- Halfedge*, 813–815, 1040–1041, 1217, 1758–1759
- halfedge*
 - ArrangementDcelFace, 1244
 - ArrangementDcelVertex, 1239
 - ConvexHullPolyhedronFacet_3, 669
 - Face, 1762
 - Facet, 811
 - HalfedgeDSFace, 855
 - HalfedgeDSVertex, 861
 - Topological_explorer, 967
 - Vertex, 817, 1760
- halfedge_around_vertex_begin*
 - StraightSkeletonVertex_2, 1077
- Halfedge_const_handle*, 1296
- halfedge_distance*
 - Arr_accessor, 1211
- Halfedge_handle*, 818, 1210, 1292, 1312
- HalfedgeDS, 847–854
- HalfedgeDS::face_handle*, 849
- HalfedgeDS::halfedge_handle*, 849
- HalfedgeDS::vertex_handle*, 849
- HalfedgeDS_const_decorator*, 863–864
- HalfedgeDS_decorator*, 865–871
- HalfedgeDS_default*, 872–873
- HalfedgeDS_face_base*, 874–875
- HalfedgeDS_face_min_base*, 876
- HalfedgeDS_halfedge_base*, 877
- HalfedgeDS_halfedge_min_base*, 878
- HalfedgeDS_items_2*, 879–880
- HalfedgeDS_items_decorator*, 881–884
- HalfedgeDS_list*, 886–887
- HalfedgeDS_min_items*, 885
- HalfedgeDS_vector*, 888
- HalfedgeDS_vertex_base*, 889–890
- HalfedgeDS_vertex_min_base*, 891
- HalfedgeDSFace, 855–856
- HalfedgeDSHalfedge, 857–858
- HalfedgeDSItems, 859–860
- HalfedgeDSVertex, 861–862
- halfedges_begin*
 - Arrangement_2, 1203
 - ArrangementDcel, 1237
 - HalfedgeDS, 850
 - Nef_polyhedron_3, 1036
 - Polyhedron_3, 802
- Topological_explorer*, 967
- Voronoi_diagram_2*, 1754
- halfedges_end*
 - Arrangement_2, 1203
 - ArrangementDcel, 1237
 - HalfedgeDS, 850
 - Nef_polyhedron_3, 1036
 - Polyhedron_3, 802
 - Topological_explorer, 967
 - Voronoi_diagram_2, 1754
- halfedges_splice*
 - HalfedgeDS_list, 886
- Halffacet*, 1042–1043
- Halffacet_cycle_iterator*, 1050
- halffacets_begin*
 - Nef_polyhedron_3, 1036
- halffacets_end*
 - Nef_polyhedron_3, 1036
- Handle, 2729
- handle*
 - Box_with_handle_d, 2180
- Handle_hash_function*, 2775
- has_audit_time*
 - Kinetic::Simulator, 2507
- has_certificates*
 - Kinetic::Delaunay_triangulation_3, 2428
- has_in_relative_interior*
 - Sphere_segment, 999
- Has_inserter*, 1776–1784
- has_larger_distance_to_point*, 168
- has_larger_signed_distance_to_line*, 169
- has_larger_signed_distance_to_plane*, 170
- Has_nearest_site_2*, 1772–1775
- has_neighbor*
 - TriangulationDSFaceBase_2, 1472
- has_neighbor*
 - Cell, 1585
 - TriangulationDataStructure_2::Face, 1474
 - TriangulationDSCellBase_3, 1590
 - TriangulationDSFaceBase_2, 1472
- has_on*
 - Hyperplane_d, 467
 - Line_2, 73
 - Line_3, 98
 - Line_d, 459
 - Plane_3, 101
 - Ray_2, 80
 - Ray_3, 106
 - Ray_d, 461
 - Segment_2, 82
 - Segment_3, 108
 - Segment_d, 463
 - Sphere_circle, 1000
 - Sphere_segment, 999

- [Triangle_3](#), [114](#)
- has_on_boundary*
 - [Circle_2](#), [66](#)
 - [Hyperplane_d](#), [467](#)
 - [Iso_box_d](#), [472](#)
 - [Iso_cuboid_3](#), [95](#)
 - [Iso_rectangle_2](#), [70](#)
 - [Min_annulus_d](#), [2248](#)
 - [Min_circle_2](#), [2195](#)
 - [Min_ellipse_2](#), [2205](#)
 - [Min_sphere_d](#), [2240](#)
 - [Polygon_2](#), [729](#)
 - [Sphere_3](#), [110](#)
 - [Sphere_d](#), [471](#)
 - [Tetrahedron_3](#), [113](#)
 - [Triangle_2](#), [84](#)
- has_on_bounded_side*
 - [Circle_2](#), [66](#)
 - [Iso_box_d](#), [472](#)
 - [Iso_cuboid_3](#), [95](#)
 - [Iso_rectangle_2](#), [70](#)
 - [Min_annulus_d](#), [2247](#)
 - [Min_circle_2](#), [2195](#)
 - [Min_ellipse_2](#), [2205](#)
 - [Min_sphere_d](#), [2240](#)
 - [Polygon_2](#), [729](#)
 - [Sphere_3](#), [110](#)
 - [Sphere_d](#), [471](#)
 - [Tetrahedron_3](#), [113](#)
 - [Triangle_2](#), [84](#)
- has_on_bounded_side_3_object*
 - [ImplicitSurfaceTraits_3](#), [1836](#)
- has_on_negative_side*
 - [Circle_2](#), [66](#)
 - [Hyperplane_d](#), [467](#)
 - [Line_2](#), [73](#)
 - [Plane_3](#), [101](#)
 - [Plane_separator](#), [2088](#)
 - [Polygon_2](#), [729](#)
 - [SpatialSeparator](#), [2102](#)
 - [Sphere_3](#), [110](#)
 - [Sphere_d](#), [470](#)
 - [Tetrahedron_3](#), [113](#)
 - [Triangle_2](#), [84](#)
- has_on_positive_side*
 - [Circle_2](#), [66](#)
 - [Hyperplane_d](#), [467](#)
 - [Line_2](#), [73](#)
 - [Plane_3](#), [101](#)
 - [Polygon_2](#), [729](#)
 - [Sphere_3](#), [110](#)
 - [Sphere_d](#), [470](#)
 - [Tetrahedron_3](#), [113](#)
 - [Triangle_2](#), [84](#)
- has_on_unbounded_side*
 - [Circle_2](#), [66](#)
 - [Iso_box_d](#), [472](#)
 - [Iso_cuboid_3](#), [95](#)
 - [Iso_rectangle_2](#), [70](#)
 - [Min_annulus_d](#), [2248](#)
 - [Min_circle_2](#), [2195](#)
 - [Min_ellipse_2](#), [2206](#)
 - [Min_sphere_d](#), [2240](#)
 - [Polygon_2](#), [730](#)
 - [Sphere_3](#), [110](#)
 - [Sphere_d](#), [471](#)
 - [Tetrahedron_3](#), [113](#)
 - [Triangle_2](#), [84](#)
- has_rational_current_time*
 - [Kinetic::Simulator](#), [2507](#)
- has_halfloop*
 - [Nef_polyhedron_S2](#), [995](#)
- has_smaller_distance_to_point*, [171](#)
- has_smaller_signed_distance_to_line*, [172](#)
- has_smaller_signed_distance_to_plane*, [173](#)
- has_source*
 - [Halfedge](#), [1759](#)
- has_target*
 - [Halfedge](#), [1759](#)
- has_vertex*
 - [Cell](#), [1585](#)
 - [Triangulation_3](#), [1509](#), [1510](#)
 - [TriangulationDataStructure_2::Face](#), [1474](#)
 - [TriangulationDataStructure_3](#), [1576](#)
 - [TriangulationDSCellBase_3](#), [1590](#)
 - [TriangulationDSFaceBase_2](#), [1472](#)
- hash_function*
 - [Unique_hash_map](#), [2783](#)
- hidden_points_begin*
 - [RegularTriangulationCellBase_3](#), [1550](#)
- hidden_points_end*
 - [RegularTriangulationCellBase_3](#), [1550](#)
- hidden_sites_begin*
 - [Apollonius_graph_2](#), [1716](#)
 - [ApolloniusGraphVertexBase_2](#), [1724](#)
- hidden_sites_end*
 - [Apollonius_graph_2](#), [1716](#)
 - [ApolloniusGraphVertexBase_2](#), [1725](#)
- hidden_vertices_begin*
 - [Regular_triangulation_2](#), [1416](#)
- hidden_vertices_end*
 - [Regular_triangulation_2](#), [1416](#)
- hide_point*
 - [RegularTriangulationCellBase_3](#), [1551](#)
- high_val*
 - [Kd_tree_node](#), [2073](#)
- hm*
 - [Aff_transformation_2](#), [62](#)

- Aff_transformation_3*, 91
- hole*
 - ArrangementDcelHalfedge*, 1242
- Hole_iterator*, 1246
- holes_begin*
 - ArrangementDcelFace*, 1244
 - Face*, 1218
 - GeneralPolygonWithHoles_2*, 926
 - Topological_explorer*, 968
- holes_end*
 - ArrangementDcelFace*, 1244
 - Face*, 1218
 - GeneralPolygonWithHoles_2*, 926
 - Topological_explorer*, 968
- Homogeneous*, 48
- homogeneous*
 - Aff_transformation_2*, 62
 - Aff_transformation_3*, 91
 - Kernel::Component_accessor_d*, 506
 - Point_2*, 76
 - Point_3*, 103
 - Point_d*, 448
 - Vector_2*, 86
 - Vector_3*, 117
 - Vector_d*, 452
- homogeneous_begin*
 - Point_d*, 449
 - Vector_d*, 453
- Homogeneous_converter*, 49
- homogeneous_end*
 - Point_d*, 449
 - Vector_d*, 453
- homogeneous_linear_solver*
 - LinearAlgebraTraits_d*, 439
- homogeneous_to_cartesian*, 50
- homogeneous_to_quotient_cartesian*, 51
- hull_points_begin*
 - Convex_hull_d*, 692
- hull_points_end*
 - Convex_hull_d*, 692
- hull_vertices_begin*
 - Convex_hull_d*, 692
- hull_vertices_end*
 - Convex_hull_d*, 692
- hw*
 - Point_2*, 76
 - Point_3*, 103
 - Vector_2*, 86
 - Vector_3*, 116
- hx*
 - Point_2*, 75
 - Point_3*, 102
 - Vector_2*, 85
 - Vector_3*, 116
- hy*
 - Point_2*, 75
 - Point_3*, 102
 - Vector_2*, 85
 - Vector_3*, 116
- Hyperplane_d*, 465–468
- hyperplane_supporting*
 - Convex_hull_d*, 690
- hz*
 - Point_3*, 103
 - Vector_3*, 116
- ID*, 2174, 2179
- id*
 - Box_d*, 2175
 - Box_with_handle_d*, 2180
 - BoxIntersectionBox_d*, 2171
 - StraightSkeletonVertex_2*, 1076
- Identity*, 2668
- Identity_policy_2*, 1776
- Identity_transformation*, 132
- IMPERATIVELY_BAD*, 1811
- Implicit_surface_3*, 1833
- ImplicitFunction*, 1832
- ImplicitSurfaceTraits_3*, 1834–1836
- in*
 - ArrangementInputFormatter*, 1292
- In_place_list*, 2603–2607
- In_place_list_base*, 2602
- in_smallest_orthogonal_sphere_3_object*
 - Regular_triangulation_euclidean_traits_3*, 1544
- incident_cells*
 - SurfaceMeshTriangulation_3*, 1859
 - Triangulation_3*, 1517, 1518
 - TriangulationDataStructure_3*, 1582, 1583
- incident_constraints*
 - Constrained_triangulation_2*, 1390
- incident_edges*
 - Apollonius_graph_2*, 1716, 1717
 - DelaunayGraph_2*, 1766
 - Segment_Delaunay_graph_2*, 1664, 1665
 - Triangulation_2*, 1439
 - TriangulationDataStructure_2*, 1466
- incident_faces*
 - Apollonius_graph_2*, 1716
 - DelaunayGraph_2*, 1766
 - Segment_Delaunay_graph_2*, 1664, 1665
 - Triangulation_2*, 1438, 1439
 - TriangulationDataStructure_2*, 1466
- incident_facets*
 - SurfaceMeshComplex_2InTriangulation_3*, 1850
 - Triangulation_3*, 1517, 1518

- TriangulationDataStructure_3, 1582, 1583
- incident_halfedges*
 - Vertex, 1216, 1761
 - Voronoi_diagram_2, 1755
- incident_sfce*
 - Halfedge, 1041
 - SHalfedge, 1004, 1046
 - SHalfloop, 1005, 1048
 - SVertex, 1002
- incident_vertices*
 - Apollonius_graph_2, 1716
 - DelaunayGraph_2, 1766
 - Segment_Delaunay_graph_2, 1664, 1665
 - Triangulation_2, 1439
 - Triangulation_3, 1518
 - TriangulationDataStructure_2, 1466
 - TriangulationDataStructure_3, 1583
- incident_volume*
 - Halffacet, 1043
- INCLUDED, 960, 992, 1035
- includes_edge*
 - Triangulation_2, 1432
- incremental algorithm*
 - Min_circle_2, 2197
 - Min_ellipse_2, 2208
 - Min_sphere_d, 2242
- Incremental_neighbor_search, 2065–2066
- independent_columns*
 - LinearAlgebraTraits_d, 439
- index*
 - Cell, 1585
 - Convex_hull_d, 689
 - Delaunay_d, 702
 - TriangulationDataStructure_2::Face, 1474
 - TriangulationDSCellBase_3, 1590
 - TriangulationDSFaceBase_2, 1472
- index_mesh_vertices*
 - Parameterization_mesh_patch_3, 1969
 - Parameterization_polyhedron_adaptor_3, 1975
 - ParameterizationMesh_3, 1959
- index_of_vertex_in_opposite_facet*
 - Convex_hull_d, 689
- index_of_vertex_in_opposite_simplex*
 - Convex_hull_d, 689
 - Delaunay_d, 702
- induced_edges_begin*
 - Arrangement_with_history_2, 1320
- induced_edges_end*
 - Arrangement_with_history_2, 1320
- inf*
 - Interval, 2033
 - Interval_nt, 2545
- inf_closed*
 - Interval_skip_list_interval, 2034
- inf_distance_2_object*
 - Rectangular_p_center_default_traits_2, 2234
 - RectangularPCenterTraits_2, 2236
- infinite_cell*
 - Triangulation_3, 1507
- infinite_edge_interior_conflict_2_object*
 - ApolloniusGraphTraits_2, 1730
 - SegmentDelaunayGraphTraits_2, 1686
- infinite_face*
 - Apollonius_graph_2, 1715
 - DelaunayGraph_2, 1765
 - Segment_Delaunay_graph_2, 1663
 - Triangulation_2, 1431
- infinite_vertex*
 - Apollonius_graph_2, 1715
 - DelaunayGraph_2, 1765
 - Segment_Delaunay_graph_2, 1663
 - Triangulation_2, 1431
 - Triangulation_3, 1507
- Info, 1449, 1453, 1553, 1555
- info*
 - Parameterization_polyhedron_adaptor_3, 1974
 - Triangulation_cell_base_with_info_3, 1553
 - Triangulation_face_base_with_info_2, 1449
 - Triangulation_vertex_base_with_info_2, 1453
 - Triangulation_vertex_base_with_info_3, 1555
- init*, 2535
 - Box_d, 2175
 - Box_with_handle_d, 2179
 - Delaunay_mesher_2, 1807
 - Extremal_polygon_area_traits_2, 2292
 - Extremal_polygon_perimeter_traits_2, 2295
 - ExtremalPolygonTraits_2, 2296
 - GeneralPolygon_2, 924
- init_Delaunay*
 - Triangulation_conformer_2, 1818
- init_Gabriel*
 - Triangulation_conformer_2, 1818
- initialize_system_from_mesh_border*
 - Fixed_border_parameterizer_3, 1948
 - LSCM_parameterizer_3, 1952
- inline*, 7
- inlining*, 7
- inner_range_intersects*
 - Fuzzy_iso_box, 2060
 - Fuzzy_sphere, 2062
 - FuzzyQueryItem, 2059
- inner_support_points_begin*
 - Min_annulus_d, 2246
- inner_support_points_end*
 - Min_annulus_d, 2246
- input*

- Min_annulus_d*, 2249
- Min_circle_2*, 2197
- Min_ellipse_2*, 2207
- Min_sphere_d*, 2242
- input_sites_begin*
 - Segment_Delaunay_graph_2*, 1664
- input_sites_end*
 - Segment_Delaunay_graph_2*, 1664
- insert*
 - Apollonius_graph_2*, 1717
 - Apollonius_graph_hierarchy_2*, 1734
 - Arr_consolidated_curve_data_traits_2*<Traits, Data>::Data_container, 1291
 - Compact_container*, 2612
 - Constrained_Delaunay_triangulation_2*, 1384, 1385
 - Constrained_triangulation_2*, 1390
 - Constrained_triangulation_plus_2*, 1396
 - Convex_hull_d*, 690
 - Delaunay_d*, 702
 - Delaunay_triangulation_2*, 1402
 - Delaunay_triangulation_3*, 1521
 - General_polygon_set_2*, 918, 919
 - In_place_list*, 2605
 - Interval_skip_list*, 2031
 - Kd_tree*, 2070
 - Kinetic::EventQueue*, 2485
 - Largest_empty_iso_rectangle_2*, 2299
 - Min_annulus_d*, 2248
 - Min_circle_2*, 2196
 - Min_ellipse_2*, 2206
 - Min_sphere_d*, 2241
 - Min_sphere_of_spheres_d*, 2254
 - Multiset*, 2616
 - Polygon_2*, 727, 728
 - Polytope_distance_d*, 2318
 - Regular_triangulation_2*, 1413
 - Regular_triangulation_3*, 1529
 - Segment_Delaunay_graph_2*, 1665, 1666
 - Triangulation_2*, 1433, 1434
 - Triangulation_3*, 1513, 1514
 - Union_find*, 2781
 - Unique_hash_map*, 2783
 - Voronoi_diagram_2*, 1756
- insert_after*
 - Multiset*, 2617
- insert_at_vertices*
 - Arrangement_2*, 1205–1207
- insert_at_vertices_ex*
 - Arr_accessor*, 1212
- insert_before*
 - Multiset*, 2617
- insert_constraint*
 - Constrained_Delaunay_triangulation_2*, 1385
 - Constrained_triangulation_2*, 1390, 1391
 - Constrained_triangulation_plus_2*, 1396
- insert_curve*, 1220
- insert_curves*, 1221
- insert_degree_2*
 - ApolloniusGraphDataStructure_2*, 1722
 - Triangulation_data_structure_2*, 1480
- insert_dim_up*
 - TriangulationDataStructure_2*, 1467
- insert_first*
 - Triangulation_2*, 1434
 - TriangulationDataStructure_2*, 1466
- insert_from_left_vertex*
 - Arrangement_2*, 1205, 1206
- insert_from_right_vertex*
 - Arrangement_2*, 1205, 1206
- insert_from_vertex_ex*
 - Arr_accessor*, 1212
- insert_halfedge*
 - HalfedgeDS_items_decorator*, 883
- insert_in_cell*
 - Triangulation_3*, 1514
 - TriangulationDataStructure_3*, 1578
- insert_in_edge*
 - Triangulation_2*, 1436
 - Triangulation_3*, 1514
 - TriangulationDataStructure_2*, 1466
 - TriangulationDataStructure_3*, 1578
- insert_in_face*
 - Triangulation_2*, 1436
 - TriangulationDataStructure_2*, 1466
- insert_in_face_interior*
 - Arrangement_2*, 1204
- insert_in_face_interior_ex*
 - Arr_accessor*, 1212
- insert_in_facet*
 - Triangulation_3*, 1514
 - TriangulationDataStructure_3*, 1578
- insert_in_hole*
 - SurfaceMeshTriangulation_3*, 1861
 - Triangulation_3*, 1515
 - TriangulationDataStructure_3*, 1579
- insert_increase_dimension*
 - TriangulationDataStructure_3*, 1578
- insert_isolated_vertex*
 - Arr_accessor*, 1213
- Insert_iterator*, 2630
- insert_non_intersecting_curve*, 1224
- insert_non_intersecting_curves*, 1225
- insert_object*
 - Kinetic::ActiveObjectsTable*, 2478
- insert_outside_affine_hull*
 - Triangulation_2*, 1436
 - Triangulation_3*, 1515

- insert_outside_convex_hull*
 - Triangulation_2*, 1436
 - Triangulation_3*, 1514
- insert_p*
 - Polytope_distance_d*, 2318, 2319
- insert_point*, 1226
- insert_q*
 - Polytope_distance_d*, 2318, 2319
- insert_second*
 - Triangulation_2*, 1436
 - TriangulationDataStructure_2*, 1466
- insert_tip*
 - HalfedgeDS_items_decorator*, 882
- insert_x_monotone_curve*, 1222
- insert_x_monotone_curves*, 1223
- inserter*, 2630
- inside_out*
 - HalfedgeDS_decorator*, 870
 - Polyhedron_3*, 809
- instantaneous_kernel_object*
 - Kinetic::SimulationTraits*, 2502
- integer_grid_point_2_object*
 - SnapRoundingTraits_2*, 1345
- Integrator_2*, 2402
- INTERIOR*, 1605, 1633
- interior*
 - Nef_polyhedron_2*, 961
 - Nef_polyhedron_3*, 1036
 - Nef_polyhedron_S2*, 993
- INTERNAL*, 2072
- Interpolation*
 - regular_neighbor_coordinates_2*, 2376–2377
 - surface_neighbor_coordinates_3*, 2386–2389
 - surface_neighbors_3*, 2390–2392
- Interpolation_gradient_fitting_traits_2*, 2382
- Interpolation_traits_2*, 2373
- InterpolationTraits*, 2366, 2371–2372
 - model*, 2373, 2382
- intersect_2_object*
 - ArrangementXMonotoneTraits_2*, 1262
 - ConstrainedTriangulationTraits_2*, 1381
- intersect_3_object*
 - SurfaceMeshTraits_3*, 1854
- intersection*, 432
- intersection*, 19
 - all pairs*, 2164, 2169
 - iso-oriented boxes*, 2161, 2164, 2169
 - self-intersection*, 2166
- intersection*, 174–176, 485–486, 945–946, 949
 - ExtendedKernelTraits_2*, 975
 - General_polygon_set_2*, 919–921
 - Nef_polyhedron_2*, 961
 - Nef_polyhedron_3*, 1037
 - Nef_polyhedron_S2*, 993
- Intersection_tag*, 1389, 1394
- Intersections_tag*, 1688–1690, 1692
- Interval*, 2033
- Interval*, 2123, 2124, 2128, 2130, 2131, 2134, 2137, 2138
- interval*
 - Filtered_exact*, 2532
 - Lazy_exact_nt*, 2557
- Interval_nt*, 2544–2548
- Interval_nt_advanced*, 2546
- Interval_skip_list*, 2031–2032
- Interval_skip_list_interval*, 2034
- intervals*
 - Real_timer*, 2778
 - Timer*, 2779
- invalidate_c2t3_cache*
 - SurfaceMeshVertexBase_3*, 1864
- inverse*
 - Aff_transformation_2*, 62
 - Aff_transformation_3*, 91
 - Aff_transformation_d*, 475
 - LinearAlgebraTraits_d*, 437
- Inverse_index*, 2634
- inverse_of_transformed_distance*
 - Euclidean_distance*, 2055
 - Euclidean_distance_sphere_point*, 2057
 - GeneralDistance*, 2064
 - Manhattan_distance_iso_box_point*, 2076
 - OrthogonalDistance*, 2083
 - Weighted_Minkowski_distance*, 2107
- is_active*
 - Qt_widget_layer*, 2851
- is_anchor*
 - Range_tree_d*, 2127
 - Segment_tree_d*, 2133
 - Tree_anchor*, 2144
- is_ascii*, 2785, 2793
- is_bad*
 - SurfaceMeshCriteria_3*, 1852
- is_bad_object*
 - MeshingCriteria_2*, 1814
- is_binary*, 2785, 2794
- is_bisector*
 - Halfedge*, 1759
 - StraightSkeletonHalfedge_2*, 1079
- is_bivalent*
 - Halfedge*, 815
 - Vertex*, 817
- is_border*
 - ConvexHullPolyhedronHalfedge_3*, 670
 - Halfedge*, 814
 - HalfedgeDSHalfedge*, 858
- is_border_convex*
 - BorderParameterizer_3*, 1936

- Circular_border_parameterizer_3*, 1940
- Square_border_parameterizer_3*, 1990
- Two_vertices_parameterizer_3*, 2002
- is_border_edge*
 - Halfedge*, 814
- is_c2t3_cache_valid*
 - SurfaceMeshVertexBase_3*, 1864
- is_ccw_strongly_convex_2*, 648
- is_cell*
 - Triangulation_3*, 1509
 - TriangulationDataStructure_3*, 1576
- is_circle*
 - Min_ellipse_2_traits_2*, 2210
- is_circular*
 - Arr_circle_segment_traits_2<Kernel>::Curve_2*, 1274
 - Arr_circle_segment_traits_2<Kernel>::X_monotone_curve_2*, 1276
- is_clockwise_oriented*
 - Polygon_2*, 729
- is_closed*
 - Polyhedron_3*, 803
- is_collinear_oriented*
 - Polygon_2*, 729
- is_conforming_Delaunay*
 - Triangulation_conformer_2*, 1818
- is_conforming_done*
 - Triangulation_conformer_2*, 1818
- is_conforming_Gabriel*
 - Triangulation_conformer_2*, 1818
- is_constrained*
 - Constrained_triangulation_2*, 1390
 - ConstrainedTriangulationFaceBase_2*, 1379
- is_contained*
 - Interval_skip_list*, 2032
- is_contour*
 - StraightSkeletonVertex_2*, 1077
- is_convex*
 - Polygon_2*, 729
- Is_convex_2*, 753, 757
- is_convex_2*, 719
- is_convex_2_object*
 - ConvexPartitionIsValidTraits_2*, 745
- is_counterclockwise_oriented*
 - Polygon_2*, 729
- is_cw_strongly_convex_2*, 649
- is_defined*
 - SegmentDelaunayGraphSite_2*, 1670
 - SegmentDelaunayGraphStorageSite_2*, 1674
 - Unique_hash_map*, 2783
- is_degenerate*
 - Circle_2*, 66
 - Direction_d*, 456
 - ExtendedKernelTraits_2*, 975
 - Iso_box_d*, 472
 - Iso_cuboid_3*, 95
 - Iso_rectangle_2*, 70
 - Line_2*, 73
 - Line_3*, 98
 - Min_annulus_d*, 2248
 - Min_circle_2*, 2195
 - Min_ellipse_2*, 2206
 - Min_sphere_d*, 2240
 - Plane_3*, 101
 - Polytope_distance_d*, 2317
 - Ray_2*, 80
 - Ray_3*, 106
 - Segment_2*, 82
 - Segment_3*, 108
 - Segment_d*, 463
 - Sphere_3*, 110
 - Sphere_d*, 470
 - Sphere_segment*, 999
 - Tetrahedron_3*, 112
 - Triangle_2*, 83
 - Triangle_3*, 114
- is_degenerate_edge_2_object*
 - ApolloniusGraphTraits_2*, 1730
- is_dimension_jump*
 - Convex_hull_d*, 690
- is_edge*
 - SurfaceMeshTriangulation_3*, 1860
 - Triangulation_2*, 1432
 - Triangulation_3*, 1509
 - TriangulationDataStructure_2*, 1465
 - TriangulationDataStructure_3*, 1575
- is_editing*
 - Kinetic::ActiveObjectsTable*, 2478
- is_empty*
 - Arrangement_2*, 1203
 - General_polygon_2*, 932
 - General_polygon_set_2*, 918
 - Min_annulus_d*, 2248
 - Min_circle_2*, 2195
 - Min_ellipse_2*, 2206
 - Min_sphere_d*, 2240
 - Min_sphere_of_spheres_d*, 2253
 - Nef_polyhedron_2*, 961
 - Nef_polyhedron_3*, 1036
 - Nef_polyhedron_S2*, 993
 - Object*, 120
 - Polygon_2*, 730
- is_empty_range*, 2730
- is_even*
 - Aff_transformation_2*, 62
 - Aff_transformation_3*, 91
- is_face*
 - Triangulation_2*, 1432

- TriangulationDataStructure_2, 1465
- is_facet*
 - Triangulation_3, 1509
 - TriangulationDataStructure_3, 1575, 1576
- is_facet_on_surface*
 - SurfaceMeshCellBase_3, 1844
- is_facet_visited*
 - SurfaceMeshCellBase_3, 1844
- is_feasible*
 - Sorted_matrix_search_traits_adaptor, 2332
 - SortedMatrixSearchTraits, 2333
- is_finite*, 2534, 2573, 2577
 - Polytope_distance_d, 2317
- is_flipable*
 - Constrained_Delaunay_triangulation_2, 1386
- is_frame_edge*
 - Explorer, 971
- is_full*
 - Arr_circle_segment_traits_2<Kernel>
::Curve_2, 1274
- is_full_conic*
 - Arr_conic_traits_2<RatKernel, AlgKernel, NtTraits>::Curve_2, 1281
- is_full_dimensional*
 - Approximate_min_ellipsoid_d, 2269
- is_Gabriel*
 - Alpha_status, 1641
 - Delaunay_triangulation_3, 1524
 - Regular_triangulation_3, 1532, 1533
- is_halfcircle*
 - Sphere_segment, 999
- is_halfedge_on_ccb*
 - Face, 1762
- is_hidden*
 - RegularTriangulationVertexBase_2, 1410
- is_hidden_2_object*
 - ApolloniusGraphTraits_2, 1730
- is_horizontal*
 - Line_2, 73
 - Ray_2, 80
 - Segment_2, 82
- is_horizontal_2_object*
 - YMonotonePartitionTraits_2, 779
- is_in_complex*
 - SurfaceMeshComplex_2InTriangulation_3, 1849, 1850
- is_in_domain*
 - DelaunayMeshFaceBase_2, 1803
 - VectorField_2, 2410
- is_incident_edge*
 - Vertex, 1761
- is_incident_face*
 - Vertex, 1761
- is_infinite*
 - Apollonius_graph_2, 1717
 - DelaunayGraph_2, 1766, 1767
 - Segment_Delaunay_graph_2, 1665
 - SurfaceMeshTriangulation_3, 1860
 - Triangulation_2, 1431, 1432
 - Triangulation_3, 1508
- is_inner_bisector*
 - StraightSkeletonHalfedge_2, 1079
- is_input*
 - SegmentDelaunayGraphSite_2, 1670
 - SegmentDelaunayGraphStorageSite_2, 1674
- is_inside*
 - Range_tree_d, 2127
 - Segment_tree_d, 2133
 - Tree_anchor, 2144
- is_inside_new_face*
 - Arr_accessor, 1211
- is_isolated*
 - ArrangementDcelVertex, 1239
 - Halfedge, 1040
 - SVertex, 1002
 - Topological_explorer, 966
 - Vertex, 1216
- is_leaf*
 - Kd_tree_node, 2073
- is_legal*
 - Sphere_d, 470
- is_linear*
 - Arr_circle_segment_traits_2<Kernel>
::Curve_2, 1274
 - Arr_circle_segment_traits_2<Kernel>::X-monotone_curve_2, 1276
- is_long*
 - Sphere_segment, 999
- is_mesh_triangular*
 - Parameterization_mesh_patch_3, 1970
 - Parameterization_polyhedron_adaptor_3, 1975
 - ParameterizationMesh_3, 1960
- Is_negative*, 2553
- is_negative*, 2549
- is_odd*
 - Aff_transformation_2, 62
 - Aff_transformation_3, 91
- is_on_chull*
 - Alpha_status, 1641
- is_on_hole*
 - ArrangementDcelHalfedge, 1241
- is_on_inner_boundary*
 - Arr_accessor, 1212
- is_on_outer_boundary*
 - Arr_accessor, 1212
- Is_one*, 2554
- is_one*, 2550

- is_one_to_one_mapping*
 - Barycentric_mapping_parameterizer_3*, 1935
 - Fixed_border_parameterizer_3*, 1950
 - LSCM_parameterizer_3*, 1953
 - Mean_value_coordinates_parameterizer_3*, 1957
- is_perturbed_incircle*, 2535
- is_perturbed_insphere*, 2535
- is_plane*
 - General_polygon_set_2*, 918
 - Nef_polyhedron_2*, 961
- is_point*
 - Interval_nt*, 2545
 - SegmentDelaunayGraphSite_2*, 1670
 - SegmentDelaunayGraphStorageSite_2*, 1674
- Is_positive*, 2555
- is_positive*, 2551
- is_pretty*, 2785, 2795
- is_pure_bivalent*
 - Polyhedron_3*, 803
- is_pure_quad*
 - Polyhedron_3*, 803
- is_pure_triangle*
 - Polyhedron_3*, 803
- is_pure_trivalent*
 - Polyhedron_3*, 803
- is_quad*
 - Facet*, 812
 - Halfedge*, 815
- is_rational*
 - Arr_circle_segment_traits_2<Kernel>::CoordNT*, 1272
- is_ray*
 - Halfedge*, 1759
- is_refinement_done*
 - Delaunay_mesher_2*, 1807
- is_regular_or_boundary_for_vertices*
 - SurfaceMeshComplex_2InTriangulation_3*, 1850
- is_running*
 - Real_timer*, 2778
 - Timer*, 2779
- is_same*
 - Interval_nt*, 2545
- is_segment*
 - Halfedge*, 1759
 - SegmentDelaunayGraphSite_2*, 1670
 - SegmentDelaunayGraphStorageSite_2*, 1674
- is_shalfedge*
 - Halffacet_cycle_iterator*, 1050
 - SFace_cycle_iterator*, 1008, 1051
- is_shalfloop*
 - Halffacet_cycle_iterator*, 1050
 - SFace_cycle_iterator*, 1008, 1051
- is_short*
 - Sphere_segment*, 998
- is_simple*
 - Nef_polyhedron_3*, 1035
 - Polygon_2*, 729
- is_simple_2*, 720
- is_simplex_of_furthest*
 - Delaunay_d*, 701
- is_simplex_of_nearest*
 - Delaunay_d*, 701
- is_skeleton*
 - StraightSkeletonVertex_2*, 1077
- is_solvable*
 - LinearAlgebraTraits_d*, 439
- is_space*
 - Nef_polyhedron_3*, 1036
- is_sphere*
 - Nef_polyhedron_S2*, 993
- is_standard*
 - Explorer*, 971
 - ExtendedKernelTraits_2*, 974
- is_strongly_convex_3*, 656, 678
- is_svertex*
 - SFace_cycle_iterator*, 1008, 1051
- is_tetrahedron*
 - Polyhedron_3*, 803
- is_triangle*
 - Facet*, 812
 - Halfedge*, 815
 - Polyhedron_3*, 803
- is_trivalent*
 - Halfedge*, 815
 - Vertex*, 817
- is_unbounded*
 - Face*, 1218, 1762
 - GeneralPolygonWithHoles_2*, 926
 - Halfedge*, 1759
- Is_vacuously_valid*, 754
- is_valid*, 1219, 2534, 2573, 2577
 - AdaptationPolicy_2*, 1771
 - Apollonius_graph_2*, 1719
 - Apollonius_graph_hierarchy_2*, 1735
 - ApolloniusGraphVertexBase_2*, 1725
 - Approximate_min_ellipsoid_d*, 2270
 - Arr_conic_traits_2<RatKernel, AlgKernel, NtTraits>::Curve_2*, 1280
 - Arr_rational_arc_traits_2<AlgKernel, NtTraits>::Curve_2*, 1284
 - Arrangement_2*, 1209
 - Cell*, 1586
 - Constrained_Delaunay_triangulation_2*, 1386
 - Constrained_triangulation_2*, 1391
 - ConstrainedTriangulationFaceBase_2*, 1380
 - Convex_hull_d*, 691

- Delaunay_triangulation_2*, 1404
- Delaunay_triangulation_3*, 1525
- DelaunayGraph_2*, 1767
- Face*, 1763
- General_polygon_set_2*, 923
- Halfedge*, 1759
- HalfedgeDS_const_decorator*, 864
- HalfedgeDS_decorator*, 870
- Key*, 2493
- Min_annulus_d*, 2249
- Min_circle_2*, 2196
- Min_ellipse_2*, 2206
- Min_sphere_d*, 2241
- Min_sphere_of_spheres_d*, 2254
- Nef_polyhedron_3*, 1035
- Number_type_checker*, 2570
- Polyhedron_3*, 809
- Polytope_distance_d*, 2319
- Range_tree_d*, 2126
- Regular_triangulation_2*, 1417
- Regular_triangulation_3*, 1533
- Segment_Delaunay_graph_2*, 1668
- Segment_tree_d*, 2133
- SegmentDelaunayGraphVertexBase_2*, 1681
- Tree_anchor*, 2144
- Triangulation_2*, 1440
- Triangulation_3*, 1519
- TriangulationDataStructure_2*, 1469
- TriangulationDataStructure_2::Face*, 1475
- TriangulationDataStructure_2::Vertex*, 1478
- TriangulationDataStructure_3*, 1584
- TriangulationDSCellBase_3*, 1590
- TriangulationDSFaceBase_2*, 1473
- TriangulationDSVertexBase_3*, 1593
- Vertex*, 1588
- Vertex*, 1761
- Voronoi_diagram_2*, 1756
- is_valid_2_object*
 - GeneralPolygonSetTraits_2*, 931
- is_valid_object*
 - Partition_is_valid_traits_2*, 767
 - PartitionIsValidTraits_2*, 764
- is_vertex*
 - SurfaceMeshTriangulation_3*, 1860
 - Triangulation_3*, 1508
 - TriangulationDataStructure_2*, 1465
 - TriangulationDataStructure_3*, 1575
- is_vertex_on_border*
 - Parameterization_mesh_patch_3*, 1971
 - Parameterization_polyhedron_adaptor_3*, 1976
 - ParameterizationMesh_3*, 1961
- is_vertex_on_main_border*
 - Parameterization_mesh_patch_3*, 1971
- Parameterization_polyhedron_adaptor_3*, 1976
- ParameterizationMesh_3*, 1961
- is_vertex_parameterized*
 - Parameterization_mesh_patch_3*, 1971
 - Parameterization_polyhedron_adaptor_3*, 1976
 - ParameterizationMesh_3*, 1961
- is_vertical*
 - Line_2*, 73
 - Line_arc_2*, 556
 - Ray_2*, 80
 - Segment_2*, 82
- is_vertical_2_object*
 - ArrangementBasicTraits_2*, 1258
- is_x_monotone*
 - Arr_conic_traits_2<RatKernel, AlgKernel, NtTraits>::Curve_2*, 1281
 - Circular_arc_2*, 554
- is_y_monotone*
 - Arr_conic_traits_2<RatKernel, AlgKernel, NtTraits>::Curve_2*, 1281
 - Circular_arc_2*, 554
- Is_y_monotone_2*, 755
- is_y_monotone_2*, 749–751
 - preconditions, 749
 - traits class, 752
 - default, 749
- is_y_monotone_2_object*
 - YMonotonePartitionIsValidTraits_2*, 777
- Is_zero*, 2556
- is_zero*, 2552
 - Polytope_distance_d*, 2317
 - Vector*, 440
 - Vector_d*, 454
- iso-oriented boxes
 - intersection, 2161
- Iso_box_d*, 472–473, 2091, 2093, 2095
- Iso_cuboid_3*, 94–96
- Iso_rectangle_2*, 69–71, 2298
- isolated_vertex*
 - ArrangementDcelVertex*, 1239
- Isolated_vertex_iterator*, 1247
- isolated_vertices_begin*
 - ArrangementDcelFace*, 1244
 - Face*, 1218
 - Topological_explorer*, 968
- isolated_vertices_end*
 - ArrangementDcelFace*, 1244
 - Face*, 1218
 - Topological_explorer*, 968
- IsStronglyConvexTraits_3*, 679–680
- Istream_iterator*, 2796–2797
- IsYMonotoneTraits_2*, 749, 752

iterator, 2720
 ArrangementDcelHole, 1246
 ArrangementDcelIsolatedVertex, 1247
iterator_category, 872, 886, 888, 2746, 2762–2770
iterator_distance, 2731
Iterator_tag, 2722

join, 947–948
 General_polygon_set_2, 920, 921
 Nef_polyhedron_2, 961
 Nef_polyhedron_3, 1037
 Nef_polyhedron_S2, 993
join_face
 HalfedgeDS_decorator, 868
join_facet
 Polyhedron_3, 804
Join_input_iterator_1, 2633
Join_input_iterator_2, 2736, 2738
join_loop
 HalfedgeDS_decorator, 870
 Polyhedron_3, 806
join_vertex
 HalfedgeDS_decorator, 868
 Polyhedron_3, 804
join_vertices
 SegmentDelaunayGraphDataStructure_2, 1678
 Triangulation_data_structure_2, 1480

K_neighbor_search, 2067–2068
Kd_tree, 2069–2071
Kd_tree_node, 2072–2073
Kd_tree_rectangle, 2074–2075
Kernel, 35–40
Kernel, 54, 1672, 1688, 1689, 1691, 1693
Kernel::Affine_rank_d, 502
Kernel::Affinely_independent_d, 501
Kernel::Angle_2, 208
Kernel::Angle_3, 209
Kernel::AreOrderedAlongLine_2, 210
Kernel::AreOrderedAlongLine_3, 211
Kernel::AreParallel_2, 212
Kernel::AreParallel_3, 213
Kernel::AreStrictlyOrderedAlongLine_2, 214
Kernel::AreStrictlyOrderedAlongLine_3, 215
Kernel::Assign_2, 216
Kernel::Assign_3, 217
Kernel::BoundedSide_2, 218
Kernel::BoundedSide_3, 219
Kernel::CartesianConstIterator_2, 345
Kernel::CartesianConstIterator_3, 346
Kernel::CartesianConstIterator_d, 503
Kernel::Center_of_sphere_d, 504
Kernel::Circle_2, 220
Kernel::Collinear_2, 226
Kernel::Collinear_3, 227
Kernel::CollinearAreOrderedAlongLine_2, 221
Kernel::CollinearAreOrderedAlongLine_3, 222
Kernel::CollinearAreStrictlyOrderedAlongLine_2, 223
Kernel::CollinearAreStrictlyOrderedAlongLine_3, 224
Kernel::CollinearHasOn_2, 225
Kernel::Compare_lexicographically_d, 505
Kernel::CompareAngleWithXAxis_2, 228
Kernel::CompareDistance_2, 229
Kernel::CompareDistance_3, 230
Kernel::CompareSlope_2, 231
Kernel::CompareX_2, 237
Kernel::CompareX_3, 238
Kernel::CompareXAtY_2, 232–233
Kernel::CompareXY_2, 235
Kernel::CompareXY_3, 236
Kernel::CompareXYZ_3, 234
Kernel::CompareY_2, 241
Kernel::CompareY_3, 242
Kernel::CompareYAtX_2, 239–240
Kernel::CompareZ_3, 243
Kernel::Component_accessor_d, 506
Kernel::ComputeA_2, 244
Kernel::ComputeArea_2, 247
Kernel::ComputeArea_3, 248
Kernel::ComputeB_2, 245
Kernel::ComputeC_2, 246
Kernel::ComputeScalarProduct_2, 249
Kernel::ComputeScalarProduct_3, 250
Kernel::ComputeSquaredArea_3, 251
Kernel::ComputeSquaredDistance_2, 252
Kernel::ComputeSquaredDistance_3, 253
Kernel::ComputeSquaredLength_2, 254
Kernel::ComputeSquaredLength_3, 255
Kernel::ComputeSquaredRadius_2, 256
Kernel::ComputeSquaredRadius_3, 257
Kernel::ComputeVolume_3, 258
Kernel::ComputeX_2, 259
Kernel::ComputeXmax_2, 263
Kernel::ComputeXmin_2, 261
Kernel::ComputeY_2, 260
Kernel::ComputeYAtX_2, 265
Kernel::ComputeYmax_2, 264
Kernel::ComputeYmin_2, 262
Kernel::ConstructBaseVector_3, 266
Kernel::ConstructBbox_2, 267
Kernel::ConstructBbox_3, 268
Kernel::ConstructBisector_2, 269
Kernel::ConstructBisector_3, 270
Kernel::ConstructCartesianConstIterator_2, 271
Kernel::ConstructCartesianConstIterator_3, 272
Kernel::ConstructCartesianConstIterator_d, 507

Kernel::ConstructCenter_2, [273](#)
 Kernel::ConstructCenter_3, [274](#)
 Kernel::ConstructCentroid_2, [275](#)
 Kernel::ConstructCentroid_3, [276](#)
 Kernel::ConstructCircle_2, [277](#)
 Kernel::ConstructCircumcenter_2, [278](#)
 Kernel::ConstructCircumcenter_3, [279](#)
 Kernel::ConstructCrossProductVector_3, [280](#)
 Kernel::ConstructDifferenceOfVectors_2, [281](#)
 Kernel::ConstructDirection_2, [282](#)
 Kernel::ConstructDirection_3, [283](#)
 Kernel::ConstructIsoCuboid_3, [284](#)
 Kernel::ConstructIsoRectangle_2, [285](#)
 Kernel::ConstructLiftedPoint_3, [286](#)
 Kernel::ConstructLine_2, [287](#)
 Kernel::ConstructLine_3, [288](#)
 Kernel::ConstructMaxVertex_2, [289](#)
 Kernel::ConstructMaxVertex_3, [290](#)
 Kernel::ConstructMidpoint_2, [291](#)
 Kernel::ConstructMidpoint_3, [292](#)
 Kernel::ConstructMinVertex_2, [293](#)
 Kernel::ConstructMinVertex_3, [294](#)
 Kernel::ConstructObject_2, [295](#)
 Kernel::ConstructObject_3, [296](#)
 Kernel::ConstructOppositeCircle_2, [297](#)
 Kernel::ConstructOppositeDirection_2, [298](#)
 Kernel::ConstructOppositeDirection_3, [299](#)
 Kernel::ConstructOppositeLine_2, [300](#)
 Kernel::ConstructOppositeLine_3, [301](#)
 Kernel::ConstructOppositePlane_3, [302](#)
 Kernel::ConstructOppositeRay_2, [303](#)
 Kernel::ConstructOppositeRay_3, [304](#)
 Kernel::ConstructOppositeSegment_2, [305](#)
 Kernel::ConstructOppositeSegment_3, [306](#)
 Kernel::ConstructOppositeSphere_3, [307](#)
 Kernel::ConstructOppositeTriangle_2, [308](#)
 Kernel::ConstructOppositeVector_2, [309](#)
 Kernel::ConstructOppositeVector_3, [310](#)
 Kernel::ConstructOrthogonalVector_3, [311](#)
 Kernel::ConstructPerpendicularDirection_2, [312](#)
 Kernel::ConstructPerpendicularLine_2, [313](#)
 Kernel::ConstructPerpendicularLine_3, [314](#)
 Kernel::ConstructPerpendicularPlane_3, [315](#)
 Kernel::ConstructPerpendicularVector_2, [316](#)
 Kernel::ConstructPlane_3, [317–318](#)
 Kernel::ConstructPoint_2, [321](#)
 Kernel::ConstructPoint_3, [322](#)
 Kernel::ConstructPointOn_2, [319](#)
 Kernel::ConstructPointOn_3, [320](#)
 Kernel::ConstructProjectedPoint_2, [323](#)
 Kernel::ConstructProjectedPoint_3, [324](#)
 Kernel::ConstructProjectedXYPoint_2, [325](#)
 Kernel::ConstructRay_2, [326](#)
 Kernel::ConstructRay_3, [327](#)
 Kernel::ConstructScaledVector_2, [328](#)
 Kernel::ConstructScaledVector_3, [329](#)
 Kernel::ConstructSegment_2, [331](#)
 Kernel::ConstructSegment_3, [332](#)
 Kernel::ConstructSphere_3, [333–334](#)
 Kernel::ConstructSumOfVectors_2, [330](#)
 Kernel::ConstructSupportingPlane_3, [335](#)
 Kernel::ConstructTetrahedron_3, [336](#)
 Kernel::ConstructTranslatedPoint_2, [337](#)
 Kernel::ConstructTranslatedPoint_3, [338](#)
 Kernel::ConstructTriangle_2, [339](#)
 Kernel::ConstructTriangle_3, [340](#)
 Kernel::ConstructVector_2, [341](#)
 Kernel::ConstructVector_3, [342](#)
 Kernel::ConstructVertex_2, [343](#)
 Kernel::ConstructVertex_3, [344](#)
 Kernel::Contained_in_affine_hull_d, [508](#)
 Kernel::Contained_in_linear_hull_d, [509](#)
 Kernel::Contained_in_simplex_d, [510](#)
 Kernel::Coplanar_3, [349](#)
 Kernel::CoplanarOrientation_3, [347](#)
 Kernel::CoplanarSideOfBoundedCircle_3, [348](#)
 Kernel::CounterclockwiseInBetween_2, [350](#)
 Kernel::Direction_2, [351](#)
 Kernel::Direction_3, [352](#)
 Kernel::DoIntersect_2, [353](#)
 Kernel::DoIntersect_3, [354](#)
 Kernel::Equal_2, [361](#)
 Kernel::Equal_3, [362](#)
 Kernel::Equal_d, [511](#)
 Kernel::EqualX_2, [356](#)
 Kernel::EqualX_3, [357](#)
 Kernel::EqualXY_3, [355](#)
 Kernel::EqualY_2, [358](#)
 Kernel::EqualY_3, [359](#)
 Kernel::EqualZ_3, [360](#)
 Kernel::Has_on_positive_side_d, [512](#)
 Kernel::HasOn_2, [373](#)
 Kernel::HasOn_3, [374](#)
 Kernel::HasOnBoundary_2, [363](#)
 Kernel::HasOnBoundary_3, [364](#)
 Kernel::HasOnBoundedSide_2, [365](#)
 Kernel::HasOnBoundedSide_3, [366](#)
 Kernel::HasOnNegativeSide_2, [367](#)
 Kernel::HasOnNegativeSide_3, [368](#)
 Kernel::HasOnPositiveSide_2, [369](#)
 Kernel::HasOnPositiveSide_3, [370](#)
 Kernel::HasOnUnboundedSide_2, [371](#)
 Kernel::HasOnUnboundedSide_3, [372](#)
 Kernel::Intersect_2, [375](#)
 Kernel::Intersect_3, [376](#)
 Kernel::Intersect_d, [513](#)
 Kernel::IsDegenerate_2, [377](#)
 Kernel::IsDegenerate_3, [378–379](#)

Kernel::IsHorizontal_2, 380
 Kernel::IsoCuboid_3, 381
 Kernel::IsoRectangle_2, 382
 Kernel::IsVertical_2, 383
 Kernel::LeftTurn_2, 384
 Kernel::Less_lexicographically_d, 514
 Kernel::Less_or_equal_lexicographically_d, 515
 Kernel::LessDistanceToPoint_2, 385
 Kernel::LessDistanceToPoint_3, 386
 Kernel::LessRotateCCW_2, 387
 Kernel::LessSignedDistanceToLine_2, 388
 Kernel::LessSignedDistanceToPlane_3, 389
 Kernel::LessX_2, 393
 Kernel::LessX_3, 394
 Kernel::LessXY_2, 391
 Kernel::LessXY_3, 392
 Kernel::LessXYZ_3, 390
 Kernel::LessY_2, 396
 Kernel::LessY_3, 397
 Kernel::LessYX_2, 395
 Kernel::LessZ_3, 398
 Kernel::Lift_to_paraboloid_d, 516
 Kernel::Line_2, 399
 Kernel::Line_3, 400
 Kernel::Linear_base_d, 518
 Kernel::Linear_rank_d, 519
 Kernel::Linearly_independent_d, 517
 Kernel::Midpoint_d, 520
 Kernel::Object_2, 401
 Kernel::Object_3, 402
 Kernel::Orientation_2, 403
 Kernel::Orientation_3, 404
 Kernel::Orientation_d, 521
 Kernel::Oriented_side_d, 522
 Kernel::OrientedSide_2, 405
 Kernel::OrientedSide_3, 406
 Kernel::Orthogonal_vector_d, 523
 Kernel::Plane_3, 407
 Kernel::Point_2, 408–409
 Kernel::Point_3, 410–411
 Kernel::Point_of_sphere_d, 524
 Kernel::Point_to_vector_d, 525
 Kernel::Project_along_d_axis_d, 526
 Kernel::Ray_2, 412
 Kernel::Ray_3, 413
 Kernel::Segment_2, 414
 Kernel::Segment_3, 415
 Kernel::Side_of_bounded_sphere_d, 527
 Kernel::Side_of_oriented_sphere_d, 528
 Kernel::SideOfBoundedCircle_2, 416
 Kernel::SideOfBoundedSphere_3, 417
 Kernel::SideOfOrientedCircle_2, 418
 Kernel::SideOfOrientedSphere_3, 419
 Kernel::Sphere_3, 420
 Kernel::Squared_distance_d, 529
 Kernel::Tetrahedron_3, 421
 Kernel::Triangle_2, 422
 Kernel::Triangle_3, 423
 Kernel::Value_at_d, 530
 Kernel::Vector_2, 424
 Kernel::Vector_3, 425
 Kernel::Vector_to_point_d, 531
 Kernel_archetype, 52–53
 Kernel_d, 499–500
 Kernel_traits, 54
 Key, 2493
 Key, 2123, 2124, 2128, 2130, 2131, 2134, 2137, 2138
 key_comp
 Multiset, 2616
 tree_point_traits, 2143
 Key_type, 2370
 keyPressEvent
 Qt_widget_layer, 2851
 keyReleaseEvent
 Qt_widget_layer, 2851
 Kinetic::Active_objects_listener_helper, 2476
 Kinetic::Active_objects_vector, 2479
 Kinetic::ActiveObjectsTable, 2477–2478
 Kinetic::Cartesian_instantaneous_kernel, 2480
 Kinetic::Cartesian_kinetic_kernel, 2481–2482
 Kinetic::Certificate, 2483
 Kinetic::Default_simulator, 2503
 Kinetic::Delaunay_triangulation_2, 2426–2427
 Kinetic::Delaunay_triangulation_3, 2428–2429
 Kinetic::Delaunay_triangulation_cell_base_3, 2430
 Kinetic::Delaunay_triangulation_event_log_visitor_2, 2431
 Kinetic::Delaunay_triangulation_event_log_visitor_3, 2432
 Kinetic::Delaunay_triangulation_face_base_2, 2433
 Kinetic::Delaunay_triangulation_recent_edges_visitor_2, 2434
 Kinetic::Delaunay_triangulation_visitor_base_2, 2438
 Kinetic::Delaunay_triangulation_visitor_base_3, 2439
 Kinetic::DelaunayTriangulationVisitor2, 2435
 Kinetic::DelaunayTriangulationVisitor3, 2436–2437
 Kinetic::Enclosing_box_2, 2440
 Kinetic::Enclosing_box_3, 2441
 Kinetic::Erase_event, 2442
 Kinetic::EventLogVisitor, 2443
 Kinetic::EventQueue, 2485–2486
 Kinetic::FunctionKernel, 2488–2490

Kinetic::Insert_event, 2444
Kinetic::InstantaneousKernel, 2492
Kinetic::Kernel, 2494
Kinetic::Qt_moving_points_2, 2445
Kinetic::Qt_triangulation_2, 2446
Kinetic::Qt_widget_2, 2447
Kinetic::Regular_triangulation_3, 2448
Kinetic::Regular_triangulation_cell_base_3, 2449
Kinetic::Regular_triangulation_event_log_visitor_3, 2450
Kinetic::Regular_triangulation_instantaneous_traits_3, 2451
Kinetic::Regular_triangulation_vertex_base_3, 2452
Kinetic::Regular_triangulation_visitor_base_3, 2455
Kinetic::RegularTriangulationVisitor3, 2453–2454
Kinetic::RootStack, 2500
Kinetic::SimulationTraits, 2501–2502
Kinetic::Simulator, 2506–2509
Kinetic::Simulator_kds_listener, 2504
Kinetic::Simulator_objects_listener, 2505
Kinetic::Sort, 2457
Kinetic::Sort_event_log_visitor, 2456
Kinetic::Sort_visitor_base, 2458
Kinetic::SortVisitor, 2459
kinetic_kernel_object
 Kinetic::SimulationTraits, 2502

LARGER, 124
 largest inscribed polygon, 2287, 2289
Largest_empty_iso_rectangle_2, 2298–2300
LargestEmptyIsoRectangleTraits_2, 2301–2302
last_out_edge
 Topological_explorer, 966
Lazy_exact_nt, 2557–2558
LEAF, 2072
leaveEvent
 Qt_widget_layer, 2851
leda_bigfloat, 2559
leda_integer, 2560
leda_rational, 2561
leda_real, 2562
left
 Arr_circle_segment_traits_2<Kernel>::X_monotone_curve_2, 1276
 Arr_conic_traits_2<RatKernel, AlgKernel, NtTraits>::X_monotone_curve_2, 1282
 Arr_rational_arc_traits_2<AlgKernel, NtTraits>::Curve_2, 1285
 Circular_arc_2, 554
 Halfedge, 1759
 Line_arc_2, 556
LEFT_TURN, 127

left_turn, 177
 ExtendedKernelTraits_2, 975
left_turn_2_object
 PartitionIsValidTraits_2, 764
 PartitionTraits_2, 766
left_vertex
 Polygon_2, 729
left_vertex_2, 721
 requirements, 721
LEFTFRAME, 973
less_x_2_object
 LargestEmptyIsoRectangleTraits_2, 2302
less_xy_2_object
 AllFurthestNeighborsTraits_2, 2305
 Extremal_polygon_area_traits_2, 2293
 Extremal_polygon_perimeter_traits_2, 2295
 ExtremalPolygonTraits_2, 2297
 PartitionIsValidTraits_2, 764
 PartitionTraits_2, 766
 PolygonTraits_2, 725
 RandomPolygonTraits_2, 2760
less_y_2_object
 LargestEmptyIsoRectangleTraits_2, 2302
less_yx_2_object
 IsYMonotoneTraits_2, 752
 PartitionTraits_2, 766
 PolygonTraits_2, 725
Level_interval, 2035
lexicographically_smaller, 487
lexicographically_smaller_or_equal, 488
lexicographically_xy_larger, 180
lexicographically_xy_larger_or_equal, 181
lexicographically_xy_smaller, 182
lexicographically_xy_smaller_or_equal, 183
lexicographically_xyz_smaller, 178
lexicographically_xyz_smaller_or_equal, 179
lift_to_paraboloid, 489
 lifting map, dD, 682
Line, 1412, 1520
Line_2, 72–74, 2384, 2856
Line_3, 97–98
Line_arc_2, 548, 556–557
Line_d, 458–459
line_walk
 Triangulation_2, 1438
 linear program
 Min_annulus_d, 2250
Linear_algebraCd, 445
Linear_algebraHd, 446
linear_base, 491
linear_interpolation, 2365
linear_least_squares_fitting_2, 2346
linear_least_squares_fitting_3, 2347–2348
linear_rank, 492

- linear_solver*
 - LinearAlgebraTraits_d*, 438
 - SparseLinearAlgebraTraits_d*, 1986
 - Taucs_solver_traits*, 1995
 - Taucs_symmetric_solver_traits*, 1998
- LinearAlgebraTraits_d*, 437–439
 - model, 445, 446
- LinearKernel*, 550
- linearly_independent*, 490
- lineWidth*
 - Qt_widget*, 2844
- Listener*, 2495–2497
- LMWT*, 962
- locate*, 1311
 - ArrangementPointLocation_2*, 1303
 - Delaunay_d*, 703
 - General_polygon_set_2*, 922
 - Nef_polyhedron_2*, 962
 - Nef_polyhedron_3*, 1036
 - Nef_polyhedron_S2*, 994
 - SurfaceMeshTriangulation_3*, 1860
 - Triangulation_2*, 1432
 - Triangulation_3*, 1510
 - Voronoi_diagram_2*, 1756
- locate_around_vertex*
 - Arr_accessor*, 1211
- Locate_result*, 1752
- Locate_type*, 1406, 1430, 1505, 1558
- Location_mode*, 962
- lock*
 - Qt_widget*, 2842
- look_recenter*
 - Geomview_stream*, 2819
- lookup*
 - Delaunay_d*, 703
 - Point_set_2*, 2014
- Loop_mask_3*, 1892
- Loop_subdivision*, 1884
- low_val*
 - Kd_tree_node*, 2073
- lower*
 - Kd_tree_node*, 2073
- lower_hull*, 2D, 650–651
- lower_bound*
 - Multiset*, 2618
- lower_hull_points_2*, 611, 650–651
- LSCM_parameterizer_3*, 1951–1953
- m*
 - Aff_transformation_2*, 62
 - Aff_transformation_3*, 91
- make_alpha_shape*
 - Alpha_shape_2*, 1606
 - Alpha_shape_3*, 1634
- make_conforming_Delaunay*
 - Triangulation_conformer_2*, 1817
- make_conforming_Delaunay_2*, 1812
- make_conforming_Gabriel*
 - Triangulation_conformer_2*, 1817
- make_conforming_Gabriel_2*, 1813
- make_hole*
 - HalfedgeDS_decorator*, 866
 - Polyhedron_3*, 807
 - TriangulationDataStructure_2*, 1468
- make_object*, 120
- make_plane*, 2313
 - Width_default_traits_3*, 2311
- make_point*, 2313
 - Width_default_traits_3*, 2310
- make_quadruple*, 2702
- make_rational*
 - Rational_traits*, 2575
- make_root_of_2*, 2563, 2580
- make_set*
 - Union_find*, 2780
- make_surface_mesh*, 1822, 1837–1838
- make_tetrahedron*
 - Polyhedron_3*, 802
- make_tree*
 - Range_tree_d*, 2126
 - Range_tree_k*, 2128
 - Segment_tree_d*, 2133
 - Segment_tree_k*, 2134
- make_triangle*
 - Polyhedron_3*, 802
- make_triple*, 2700
- make_vector*, 2313
 - Width_default_traits_3*, 2311
- make_x_monotone_2_object*
 - ArrangementTraits_2*, 1263
- Manhattan_distance_iso_box_point*, 2076–2077
- Manifold_tag*, 1839
- Manifold_with_boundary_tag*, 1840
- mark*
 - Halfedge*, 1040
 - Halffacet*, 1043
 - SFace*, 1007, 1049
 - SHalfedge*, 1004, 1046
 - SHalfloop*, 1005, 1047
 - SVertex*, 1002
 - Topological_explorer*, 968, 969
 - Vertex*, 1039
 - Volume*, 1044
- Matrix*, 442–444, 1954–1955
- matrix*
 - monotone, 2321
 - searching, 2321, 2328
 - sorted, 2328

- matrix*
 - Aff_transformation_d*, 475
- Max*, 2566
- max*, 2564
 - Iso_box_d*, 472
 - Iso_cuboid_3*, 95
 - Iso_rectangle_2*, 70
 - Real_timer*, 2778
 - Segment_2*, 81
 - Segment_3*, 107
 - Segment_d*, 463
 - Timer*, 2779
- max_coord*
 - Box_d*, 2175
 - Box_with_handle_d*, 2180
 - BoxIntersectionBox_d*, 2171
 - Iso_cuboid_3*, 95
 - Iso_rectangle_2*, 70
 - Kd_tree_rectangle*, 2074
- max_distance_to_rectangle*
 - Euclidean_distance*, 2054
 - Euclidean_distance_sphere_point*, 2056
 - GeneralDistance*, 2064
 - Manhattan_distance_iso_box_point*, 2077
 - OrthogonalDistance*, 2082
 - Weighted_Minkowski_distance*, 2107
- max_size*
 - Compact_container*, 2612
 - In_place_list*, 2604
 - Multiset*, 2616
- max_span*
 - Kd_tree_rectangle*, 2074
- max_span_coord*
 - Kd_tree_rectangle*, 2074
- max_vertex*, 184
- maximum_area_inscribed_k_gon_2*, 2287–2288
- maximum_perimeter_inscribed_k_gon_2*, 2289–2290
- Mean_value_coordinates_parameterizer_3*, 1956–1957
- Median_of_max_spread*, 2078
- Median_of_rectangle*, 2079
- Memory_sizer*, 2776
- Merge*, 1287
- merge*
 - Compact_container*, 2613
 - In_place_list*, 2606
- merge_2_object*
 - ArrangementXMonotoneTraits_2*, 1262
- merge_edge*
 - Arrangement_2*, 1208
 - Arrangement_with_history_2*, 1321
- Mesh_2::Face_badness*, 1811
- mesh_facets_begin*
 - Parameterization_mesh_patch_3*, 1970
 - Parameterization_polyhedron_adaptor_3*, 1975
 - ParameterizationMesh_3*, 1960
- mesh_facets_end*
 - Parameterization_mesh_patch_3*, 1970
 - Parameterization_polyhedron_adaptor_3*, 1975
 - ParameterizationMesh_3*, 1960
- mesh_main_border_vertices_begin*
 - Parameterization_mesh_patch_3*, 1969
 - Parameterization_polyhedron_adaptor_3*, 1975
 - ParameterizationMesh_3*, 1959, 1960
- mesh_main_border_vertices_end*
 - Parameterization_mesh_patch_3*, 1970
 - Parameterization_polyhedron_adaptor_3*, 1975
 - ParameterizationMesh_3*, 1960
- mesh_vertices_begin*
 - Parameterization_mesh_patch_3*, 1969
 - Parameterization_polyhedron_adaptor_3*, 1974
 - ParameterizationMesh_3*, 1959
- mesh_vertices_end*
 - Parameterization_mesh_patch_3*, 1969
 - Parameterization_polyhedron_adaptor_3*, 1975
 - ParameterizationMesh_3*, 1959
- MeshingCriteria_2*, 1814–1815
- Method_tag*, 1688, 1689, 1691, 1693
- midpoint*, 185, 493
- Midpoint_of_max_spread*, 2080
- Midpoint_of_rectangle*, 2081
- Min*, 2567
- min*, 2565
 - Iso_box_d*, 472
 - Iso_cuboid_3*, 95
 - Iso_rectangle_2*, 70
 - Segment_2*, 81
 - Segment_3*, 107
 - Segment_d*, 463
- Min_annulus_d*, 2244–2250
 - creation*, 2245
 - global functions*, 2249
 - input*, 2249
 - output*, 2249
 - implementation*, 2250
 - member functions*, 2245–2249
 - access*, 2245
 - miscellaneous*, 2249
 - modifiers*, 2248
 - predicates*, 2247
 - validity check*, 2249

- requirements, [2244](#)
- traits class
 - requirements, [2286](#)
 - see also* [Optimisation_d_traits_2](#)
 - see also* [Optimisation_d_traits_3](#)
 - see also* [Optimisation_d_traits_d](#)
- types, [2244](#)
- Min_circle_2*, [2193–2198](#)
 - creation, [2194](#)
 - example, [2197](#)
 - global functions, [2197](#)
 - input, [2197](#)
 - output, [2197](#)
 - window output, [2197](#)
 - implementation, [2197](#)
 - member functions, [2194–2196](#)
 - access, [2194](#)
 - miscellaneous, [2196](#)
 - modifiers, [2196](#)
 - predicates, [2195](#)
 - validity check, [2196](#)
 - requirements, [2193](#)
 - traits class
 - requirements, [2200](#)
 - see also* [Min_circle_2_traits_2](#)
 - types, [2193](#)
- Min_circle_2_traits_2*, [2199](#)
- min_circulator*
 - Circulator*, [2707](#)
- min_coord*
 - Box_d*, [2175](#)
 - Box_with_handle_d*, [2180](#)
 - BoxIntersectionBox_d*, [2171](#)
 - Iso_cuboid_3*, [95](#)
 - Iso_rectangle_2*, [70](#)
 - Kd_tree_rectangle*, [2074](#)
- min_distance_to_rectangle*
 - Euclidean_distance*, [2054](#)
 - Euclidean_distance_sphere_point*, [2056](#)
 - GeneralDistance*, [2064](#)
 - Manhattan_distance_iso_box_point*, [2077](#)
 - OrthogonalDistance*, [2082](#)
 - Weighted_Minkowski_distance*, [2107](#)
- Min_ellipse_2*, [2203–2209](#)
 - creation, [2203](#)
 - example, [2208](#), [2270](#)
 - global functions, [2207](#)
 - input, [2207](#)
 - output, [2207](#)
 - window output, [2207](#)
 - implementation, [2208](#)
 - member functions, [2205–2207](#)
 - access, [2205](#)
 - miscellaneous, [2207](#)
 - modifiers, [2206](#)
 - predicates, [2205](#)
 - validity check, [2206](#)
 - requirements, [2203](#)
 - traits class
 - requirements, [2212](#)
 - see also* [Min_ellipse_2_traits_2](#)
 - types, [2203](#)
- Min_ellipse_2_traits_2*, [2210–2211](#)
- min_k*
 - Extremal_polygon_area_traits_2*, [2292](#)
 - Extremal_polygon_perimeter_traits_2*, [2295](#)
 - ExtremalPolygonTraits_2*, [2296](#)
- min_max_element*, [2624–2625](#)
- min_parallelogram_2*, [2217–2218](#)
- Min_quadrilateral_default_traits_2*, [2221–2224](#)
- min_rectangle_2*, [2215–2216](#)
- Min_sphere_d*, [2238–2243](#)
 - creation, [2238](#)
 - global functions, [2242](#)
 - input, [2242](#)
 - output, [2242](#)
 - implementation, [2242](#)
 - member functions, [2239–2242](#)
 - access, [2239](#)
 - miscellaneous, [2241](#)
 - modifiers, [2240](#)
 - predicates, [2240](#)
 - validity check, [2241](#)
 - requirements, [2238](#)
 - traits class
 - requirements, [2286](#)
 - see also* [Optimisation_d_traits_2](#)
 - see also* [Optimisation_d_traits_3](#)
 - see also* [Optimisation_d_traits_d](#)
 - types, [2238](#)
- Min_sphere_of_spheres_d*, [2251–2256](#)
 - creation, [2252](#)
 - implementation, [2255](#)
 - member functions, [2253–2254](#)
 - access, [2253](#)
 - miscellaneous, [2254](#)
 - modifiers, [2254](#)
 - predicates, [2253](#)
 - validity check, [2254](#)
 - requirements, [2251](#)
 - types, [2251](#)
- Min_sphere_of_spheres_d_traits_2*, [2259–2260](#)
- Min_sphere_of_spheres_d_traits_3*, [2261–2262](#)
- Min_sphere_of_spheres_d_traits_d*, [2263–2264](#)
- min_strip_2*, [2219–2220](#)
- min_vertex*, [186](#)
- MinCircle2Traits*, [2200](#)
- MinEllipse2Traits*, [2212](#)

- minimum enclosing
 - see also* smallest enclosing
- minimum spanning
 - see also* smallest enclosing
- minkowski_sum_with_pixel_2_object*
 - SnapRoundingTraits_2*, 1345
- MinQuadrilateralTraits_2*, 2225–2228
- MinSphereOfSpheresTraits*, 2257–2258
- mirror_facet*
 - SurfaceMeshTriangulation_3*, 1860
 - Triangulation_3*, 1518
 - TriangulationDataStructure_3*, 1583
- mirror_index*
 - Triangulation_3*, 1518
 - TriangulationDataStructure_2*, 1466
 - TriangulationDataStructure_3*, 1583
- mirror_vertex*
 - Triangulation_3*, 1518
 - TriangulationDataStructure_2*, 1466
 - TriangulationDataStructure_3*, 1583
- miscellaneous
 - Approximate_min_ellipsoid_d*, 2270
 - Min_annulus_d*, 2249
 - Min_circle_2*, 2196
 - Min_ellipse_2*, 2207
 - Min_sphere_d*, 2241
 - Min_sphere_of_spheres_d*, 2254
 - Polytope_distance_d*, 2320
- Mode*, 1606, 1633, 2785, 2798
- Modifier_base*, 2777
- modifiers
 - Min_annulus_d*, 2248
 - Min_circle_2*, 2196
 - Min_ellipse_2*, 2206
 - Min_sphere_d*, 2240
 - Min_sphere_of_spheres_d*, 2254
 - Polytope_distance_d*, 2317
- modify_edge*
 - Arrangement_2*, 1208
- modify_edge_ex*
 - Arr_accessor*, 1214
- modify_vertex*
 - Arrangement_2*, 1207
 - Kinetic::DelaunayTriangulationVisitor2*, 2435
 - Kinetic::DelaunayTriangulationVisitor3*, 2436
 - Kinetic::RegularTriangulationVisitor3*, 2453
 - Kinetic::SortVisitor*, 2459
- modify_vertex_ex*
 - Arr_accessor*, 1214
- monotone matrix search, 2321
- monotone_matrix_search*, 2321–2322
- MonotoneMatrixSearchTraits*, 2325–2326
- mouseDoubleClickEvent*
 - Qt_widget_layer*, 2851
- mouseMoveEvent*
 - Qt_widget_layer*, 2851
- mousePressEvent*
 - Qt_widget_layer*, 2851
- mouseReleaseEvent*
 - Qt_widget_layer*, 2851
- move-to-front heuristic
 - Min_circle_2*, 2197
 - Min_ellipse_2*, 2208
 - Min_sphere_d*, 2242
- move_center*
 - Qt_widget*, 2842
- move_hole*
 - Arr_accessor*, 1213
- move_isolated_vertex*
 - Arr_accessor*, 1213
- move_point*
 - Delaunay_triangulation_3*, 1521
- MP_Float*, 2568–2569
- mpq_class*, 2539
- mpz_class*, 2540
- Multi_listener*, 2498
- Multiset*, 2614–2620
- n1*
 - Number_type_checker*, 2570
- n2*
 - Number_type_checker*, 2570
- N_step_adaptor*, 2631
- NAIVE*, 962
- natural_neighbor_coordinates_2*, 2374
 - Interpolation_traits_2*, 2374–2375
- Navigation_layer*, 2862
- NE*
 - ExtendedKernelTraits_2*, 974
- NEAREST*, 700
- nearest_neighbor*, 2017
 - Apollonius_graph_2*, 1717, 1718
 - Apollonius_graph_hierarchy_2*, 1734
 - Delaunay_d*, 703
 - Point_set_2*, 2014
 - Segment_Delaunay_graph_2*, 1666
- nearest_neighbors*, 2018–2019
 - Point_set_2*, 2014
- nearest_power_vertex*
 - Regular_triangulation_2*, 1415
 - Regular_triangulation_3*, 1531
- nearest_power_vertex_in_cell*
 - Regular_triangulation_3*, 1532
- nearest_site_2_object*
 - AdaptationTraits_2*, 1769
- nearest_vertex*

- Delaunay_triangulation_2*, [1403](#)
- Delaunay_triangulation_3*, [1523](#)
- nearest_vertex_in_cell*
 - Delaunay_triangulation_3*, [1523](#)
- NECORNER*, [973](#)
- Nef polyhedron*, 2D
 - traits class
 - see also* *Extended_cartesian*
 - see also* *Extended_homogeneous*
 - see also* *Filtered_extended_homogeneous*
- Nef_polyhedron_2*, [960–964](#)
- Nef_polyhedron_3*, [1033–1038](#)
- Nef_polyhedron_S2*, [991–996](#)
- negate*, [2650](#)
- NEGATIVE*, [124](#)
- neighbor*
 - all furthest, [2303](#)
- neighbor*
 - Cell, [1585](#)
 - TriangulationDataStructure_2::Face, [1474](#)
 - TriangulationDSCellBase_3, [1590](#)
 - TriangulationDSFaceBase_2, [1472](#)
- new_cgal_object*
 - Qt_widget*, [2846](#)
- new_distance*
 - Euclidean_distance*, [2055](#)
 - OrthogonalDistance*, [2083](#)
 - Weighted_Minkowski_distance*, [2107](#)
- new_edge*
 - ArrangementDcel, [1237](#)
- new_event*
 - Kinetic::Simulator, [2507](#)
- new_face*
 - ArrangementDcel, [1237](#)
- new_hole*
 - ArrangementDcel, [1237](#)
- new_isolated_vertex*
 - ArrangementDcel, [1237](#)
- new_notification*
 - Listener, [2496](#)
- new_object*
 - Qt_widget*, [2844](#)
- new_vertex*
 - ArrangementDcel, [1237](#)
- next*
 - ArrangementDcelHalfedge, [1241](#)
 - ConvexHullPolyhedronHalfedge_3, [670](#)
 - Halfedge, [814](#), [1217](#), [1758](#)
 - HalfedgeDSHalfedge, [858](#)
 - SHalfedge, [1046](#)
 - Topological_explorer, [967](#)
- next_around_edge*, [1597](#)
- next_event_time*
 - Kinetic::Simulator, [2507](#)
- next_link*, [2602](#)
- next_on_vertex*
 - Halfedge, [814](#)
- next_priority*
 - Kinetic::EventQueue, [2485](#)
- Node_handle*, [2072](#)
- Node_type*, [2072](#)
- non_const_handle*
 - Arrangement_2, [1204](#)
- Non_manifold_tag*, [1841](#)
- normalize_border*
 - HalfedgeDS, [852](#)
 - Polyhedron_3, [808](#)
- normalized_border_is_valid*
 - HalfedgeDS_const_decorator, [864](#)
 - HalfedgeDS_decorator, [870](#)
 - Polyhedron_3, [809](#)
- NOT_BAD*, [1811](#)
- NOT_IN_COMPLEX*, [1847](#)
- notifier*
 - Listener, [2496](#)
- notify_after_global_change*
 - Arr_accessor, [1211](#)
- notify_before_global_change*
 - Arr_accessor, [1211](#)
- NT*, [1633](#), [2065](#), [2095](#)
- null_event*
 - Kinetic::Simulator, [2507](#)
- NULL_VECTOR*, [129](#)
- Null_vector*, [129](#)
- Numb_type*, [2013](#)
- number_of_alphas*
 - Alpha_shape_2, [1607](#)
 - Alpha_shape_3, [1635](#)
- number_of_cells*
 - Triangulation_3, [1507](#)
 - TriangulationDataStructure_3, [1575](#)
- number_of_columns*
 - BasicMatrix, [2327](#)
 - Dynamic_matrix, [2323](#)
 - MonotoneMatrixSearchTraits, [2325](#)
- number_of_connected_components*
 - Nef_polyhedron_S2, [994](#)
 - Topological_explorer, [969](#)
 - Voronoi_diagram_2, [1753](#)
- number_of_curves*
 - Arrangement_with_history_2, [1320](#)
- number_of_edges*
 - Arrangement_2, [1203](#)
 - Nef_polyhedron_3, [1036](#)
 - Topological_explorer, [969](#)
 - Triangulation_3, [1507](#)
 - TriangulationDataStructure_2, [1465](#)
 - TriangulationDataStructure_3, [1575](#)

- number_of_enclosing_constraints*
 - Constrained_triangulation_plus_2*, 1397
- number_of_face_cycles*
 - Topological_explorer*, 969
- number_of_faces*
 - Apollonius_graph_2*, 1715
 - Arrangement_2*, 1204
 - DelaunayGraph_2*, 1765
 - Segment_Delaunay_graph_2*, 1663
 - Topological_explorer*, 969
 - Triangulation_2*, 1431
 - TriangulationDataStructure_2*, 1464
 - Voronoi_diagram_2*, 1753
- number_of_facets*
 - Convex_hull_d*, 691
 - Nef_polyhedron_3*, 1036
 - SurfaceMeshComplex_2InTriangulation_3*, 1849
 - Triangulation_3*, 1507
 - TriangulationDataStructure_3*, 1575
- number_of_failures*, 2546
- number_of_finite_cells*
 - Triangulation_3*, 1507
- number_of_finite_edges*
 - Triangulation_3*, 1507
- number_of_finite_facets*
 - Triangulation_3*, 1507
- number_of_full_dim_faces*
 - TriangulationDataStructure_2*, 1465
- number_of_halfedges*
 - Arrangement_2*, 1203
 - Nef_polyhedron_3*, 1035
 - Topological_explorer*, 969
 - Voronoi_diagram_2*, 1753
- number_of_halffacets*
 - Nef_polyhedron_3*, 1036
- number_of_hidden_sites*
 - Apollonius_graph_2*, 1714
 - ApolloniusGraphVertexBase_2*, 1724
- number_of_hidden_vertices*
 - Regular_triangulation_2*, 1416
- number_of_holes*
 - ArrangementDcelFace*, 1244
- number_of_induced_edges*
 - Arrangement_with_history_2*, 1320
- number_of_inner_support_points*
 - Min_annulus_d*, 2245
- number_of_input_sites*
 - Segment_Delaunay_graph_2*, 1663
- number_of_isolated_vertices*
 - Arrangement_2*, 1203
 - ArrangementDcelFace*, 1244
- number_of_originating_curves*
 - Arrangement_with_history_2*, 1320

- number_of_outer_support_points*
 - Min_annulus_d*, 2246
- number_of_output_sites*
 - Segment_Delaunay_graph_2*, 1663
- number_of_points*
 - Approximate_min_ellipsoid_d*, 2267
 - Min_annulus_d*, 2245
 - Min_circle_2*, 2194
 - Min_ellipse_2*, 2205
 - Min_sphere_d*, 2239
 - Polytope_distance_d*, 2315
- number_of_points_p*
 - Polytope_distance_d*, 2315
- number_of_points_q*
 - Polytope_distance_d*, 2315
- number_of_polygons_with_holes*
 - General_polygon_set_2*, 918
- number_of_rows*
 - BasicMatrix*, 2327
 - Dynamic_matrix*, 2323
 - MonotoneMatrixSearchTraits*, 2325
- number_of_sedges*
 - Nef_polyhedron_S2*, 993
- number_of_sets*
 - Union_find*, 2780
- number_of_sface_cycles*
 - Nef_polyhedron_S2*, 994
- number_of_sfases*
 - Nef_polyhedron_S2*, 993
- number_of_shalfedges*
 - Nef_polyhedron_S2*, 993
- number_of_shalfloops*
 - Nef_polyhedron_S2*, 993
- number_of_simplices*
 - Convex_hull_d*, 691
- number_of_sloops*
 - Nef_polyhedron_S2*, 993
- number_of_solid_components*
 - Alpha_shape_2*, 1609
 - Alpha_shape_3*, 1638
- number_of_support_points*
 - Min_annulus_d*, 2245
 - Min_circle_2*, 2194
 - Min_ellipse_2*, 2205
 - Min_sphere_d*, 2239
 - Polytope_distance_d*, 2315
- number_of_support_points_p*
 - Polytope_distance_d*, 2316
- number_of_support_points_q*
 - Polytope_distance_d*, 2316
- number_of_svertices*
 - Nef_polyhedron_S2*, 993
- number_of_vertices*
 - Apollonius_graph_2*, 1714

- Arrangement_2*, 1203
- Convex_hull_d*, 691
- DelaunayGraph_2*, 1765
- Nef_polyhedron_3*, 1035
- Regular_triangulation_2*, 1416
- Segment_Delaunay_graph_2*, 1663
- Topological_explorer*, 969
- Triangulation_2*, 1431
- Triangulation_3*, 1507
- TriangulationDataStructure_2*, 1464
- TriangulationDataStructure_3*, 1575
- Voronoi_diagram_2*, 1753
- number_of_visible_sites*
 - Apollonius_graph_2*, 1714
- number_of_volumes*
 - Nef_polyhedron_3*, 1036
- Number_type_checker*, 2570–2571
- Number_type_traits*, 2572
- numerator*
 - Arr_rational_arc_traits_2*<AlgKernel, Nt-Traits>::*Curve_2*, 1284
 - Gmpq*, 2542
 - Quotient*, 2573
 - Rational_traits*, 2575
- NW
 - ExtendedKernelTraits_2*, 974
- NWCORNER, 973
- Object*, 120–122, 1520
- object_cast*, 120, 121
- OBTUSE*, 123
- OFF_to_nef_3*, 1031, 1052
- OK*, 1981, 1983
- ON_BOUNDARY*, 123
- ON_BOUNDED_SIDE*, 123
- ON_NEGATIVE_SIDE*, 125
- ON_ORIENTED_BOUNDARY*, 125
- ON_POSITIVE_SIDE*, 125
- ON_UNBOUNDED_SIDE*, 123
- Oneset_iterator*, 2627
- operation*
 - Extremal_polygon_area_traits_2*, 2293
 - Extremal_polygon_perimeter_traits_2*, 2295
 - ExtremalPolygonTraits_2*, 2296
- operator**, 189
- operator+*, 187
- operator-*, 188
- operator<<*, 832, 951, 1053, 2800
- operator>>*, 833, 952, 1054, 2792
- opposite*, 190
 - ArrangementDcelHalfedge*, 1241
 - Circle_2*, 66
 - ConvexHullPolyhedronHalfedge_3*, 670
 - Direction_d*, 456
 - Halfedge*, 814, 1758
 - HalfedgeDSHalfedge*, 858
 - Line_2*, 73
 - Line_3*, 98
 - Line_d*, 459
 - Plane_3*, 100
 - Ray_2*, 80
 - Ray_3*, 106
 - Ray_d*, 461
 - Segment_2*, 82
 - Segment_3*, 108
 - Segment_d*, 463
 - Sphere_3*, 110
 - Sphere_circle*, 1000
 - Sphere_d*, 471
 - Sphere_segment*, 998
 - Triangle_2*, 84
- opposite_facet*
 - Convex_hull_d*, 689
- opposite_simplex*
 - Convex_hull_d*, 689
 - Delaunay_d*, 702
- optimal distances*
 - distance of polytopes, 2314
 - width of 3D point set, 2306
- optimal_convex_partition_2*, 736, 756–758
- postconditions, 737, 765
- traits class, 745, 759–760
 - default, 769
- OptimalConvexPartitionTraits_2*, 756, 759–760
 - model, 769
- Optimisation_d_traits_2*, 2279–2280
- Optimisation_d_traits_3*, 2281–2282
- Optimisation_d_traits_d*, 2283–2284
- OptimisationDTraits*, 2285–2286
- Orientation*, 125
- orientation*, 192, 494
 - Arr_circle_segment_traits_2*<Kernel>::*Curve_2*, 1274
 - Arr_circle_segment_traits_2*<Kernel>::*X_monotone_curve_2*, 1276
 - Arr_conic_traits_2*<RatKernel, AlgKernel, NtTraits>::*Curve_2*, 1281
 - Circle_2*, 66
 - ExtendedKernelTraits_2*, 975
 - General_polygon_2*, 932
 - Min_circle_2_traits_2*, 2199
 - MinCircle2Traits*, 2200
 - Polygon_2*, 729
 - Sphere_3*, 110
 - Sphere_d*, 470
 - Tetrahedron_3*, 112
 - Triangle_2*, 83
- Orientation_2*, 2385

- orientation_2*, 722
- orientation_2_object*
 - AllFurthestNeighborsTraits_2, 2305
 - ApolloniusGraphTraits_2, 1730
 - Extremal_polygon_area_traits_2, 2293
 - Extremal_polygon_perimeter_traits_2, 2295
 - ExtremalPolygonTraits_2, 2297
 - PartitionIsValidTraits_2, 764
 - PartitionTraits_2, 766
 - PolygonTraits_2, 725
 - RandomPolygonTraits_2, 2760
 - SegmentDelaunayGraphTraits_2, 1686
 - Triangulation_euclidean_traits_xy_3, 1446
 - TriangulationTraits_2, 1425
- orientation_3_object*
 - TriangulationTraits_3, 1535
- orientation_d_object*
 - Kernel_d, 500
- orientationC2*, 2535
- orientationC3*, 2536
- Oriented_side*, 125
- oriented_side*
 - Circle_2, 66
 - General_polygon_set_2, 922, 923
 - Hyperplane_d, 467
 - Line_2, 73
 - Plane_3, 101
 - Polygon_2, 729
 - Sphere_3, 110
 - Sphere_d, 470
 - Tetrahedron_3, 112
 - Triangle_2, 83
 - Triangulation_2, 1433
- oriented_side_2*, 723
- oriented_side_2_object*
 - SegmentDelaunayGraphTraits_2, 1686
- oriented_side_of_bisector_test_2_object*
 - ApolloniusGraphTraits_2, 1730
 - SegmentDelaunayGraphTraits_2, 1686
- ORIGIN*, 130–131
- Origin*, 130
- origin*
 - Random_convex_set_traits_2, 2761
 - RandomConvexSetTraits_2, 2759
- originating_curves_begin*
 - Arrangement_with_history_2, 1320
- originating_curves_end*
 - Arrangement_with_history_2, 1320
- orthogonal_direction*
 - Hyperplane_d, 467
 - Plane_3, 100
- Orthogonal_incremental_neighbor_search*, 2084–2085
- Orthogonal_k_neighbor_search*, 2086–2087
- orthogonal_pole*
 - Sphere_circle, 1000
- orthogonal_transform*
 - Circle_2, 66
 - Sphere_3, 111
- orthogonal_vector*, 191
 - Hyperplane_d, 467
 - Plane_3, 100
- OrthogonalDistance*, 2082–2083
- Ostream_iterator*, 2799
- out*
 - ArrangementOutputFormatter, 1297
- out_edges*
 - Topological_explorer, 967
- out_sedge*
 - Halfedge, 1041
 - SVertex, 1002
- outer_boundary*
 - GeneralPolygonWithHoles_2, 926
- outer_ccb*
 - Face, 1218
- outer_range_contains*
 - Fuzzy_iso_box, 2060
 - Fuzzy_sphere, 2062
 - FuzzyQueryItem, 2059
- outer_support_points_begin*
 - Min_annulus_d, 2246
- outer_support_points_end*
 - Min_annulus_d, 2246
- output*
 - Min_annulus_d, 2249
 - Min_circle_2, 2197
 - Min_ellipse_2, 2207
 - Min_sphere_d, 2242
 - Polytope_distance_d, 2320
- output_identifier*
 - ExtendedKernelTraits_2, 976
- output_sites_begin*
 - Segment_Delaunay_graph_2, 1664
- output_sites_end*
 - Segment_Delaunay_graph_2, 1664
- OUTSIDE_AFFINE_HULL*, 1406, 1430, 1505, 1558
- OUTSIDE_CONVEX_HULL*, 1406, 1430, 1505, 1558
- overlay*, 1229
- OverlayTraits*, 1230–1231
- parallel*, 193, 459, 461, 464
- parallelogram*
 - smallest enclosing, 2217
- Parameterization_mesh_feature_extractor*, 1965–1966
- Parameterization_mesh_patch_3*, 1967–1971

Parameterization_polyhedron_adaptor_3, 1972–1978
ParameterizationMesh_3, 1958–1961
ParameterizationPatchableMesh_3, 1962–1964
parameterize, 1897, 1900, 1979–1980
 Fixed_border_parameterizer_3, 1948
 LSCM_parameterizer_3, 1952
 Parameterizer_traits_3, 1984
 ParameterizerTraits_3, 1982
parameterize_border
 BorderParameterizer_3, 1936
 Circular_border_parameterizer_3, 1939
 Square_border_parameterizer_3, 1990
 Two_vertices_parameterizer_3, 2002
Parameterizer_traits_3, 1983–1985
ParameterizerTraits_3, 1981–1982
partition, 735, 737
 valid, 737
partition_is_valid_2, 761–762
 traits class, 767–768
 default, 761
Partition_is_valid_traits_2, 767–768
Partition_traits_2, 769–770
PartitionIsValidTraits_2, 761, 763–764
 model, 767
PartitionTraits_2, 740, 759, 760, 765–766, 778
 model, 769
perpendicular
 Line_2, 74
 Vector_2, 86
perpendicular_line
 Plane_3, 100
perpendicular_plane
 Line_3, 98
perturb_incircle, 2535
perturb_insphere, 2535
perturb_points_2, 2744
pickplane
 Geomview_stream, 2819
PIXEL, 2841
Plane, 824, 1520
plane
 ConvexHullPolyhedronFacet_3, 669
 Face, 822
 Facet, 811
 HalfedgeDS_face_base, 874
 Halffacet, 1043
 Sphere_circle, 1000
Plane_3, 99–101, 828, 830, 2306, 2310
Plane_separator, 2088–2089
planes_begin
 Polyhedron_3, 803
planes_end
 Polyhedron_3, 803
PLUS, 2841
Point, 824, 826, 1429, 1454, 1504, 1554, 2013
point, 1559
 ApolloniusSite_2, 1720
 ArrangementDcelVertex, 1239
 ConvexHullPolyhedronVertex_3, 671
 Explorer, 971
 Halfedge, 1040
 HalfedgeDS_vertex_base, 889
 Line_2, 73
 Line_3, 97
 Line_d, 459
 Plane_3, 100
 Ray_2, 79
 Ray_3, 105
 Ray_d, 460
 Segment_2, 81
 Segment_3, 107
 Segment_d, 463
 SegmentDelaunayGraphSite_2, 1670
 SegmentDelaunayGraphStorageSite_2, 1675
 Sphere_d, 469
 StraightSkeletonVertex_2, 1076
 SVertex, 1002
 Topological_explorer, 968
 TriangulationDataStructure_2::Vertex, 1478
 TriangulationVertexBase_2, 1426
 TriangulationVertexBase_3, 1547
 Vertex, 822, 1587
 Vertex, 816, 1039, 1216, 1760
 Weighted_point, 1454
point, 2D
 generator, 2733
point, 3D
 generator, 2733
point set
 3D width of, 2306
Point_2, 75–78, 726, 1201, 1210, 1292, 1296, 1312, 1318, 1445, 1661, 1713, 1751, 2298, 2384, 2403, 2407, 2409, 2761, 2854–2857, 2859, 2861
Point_3, 102–104, 828, 830, 2306, 2310
point_container
 Segment_Delaunay_graph_2, 1663
Point_d, 447–450, 2054, 2056, 2060, 2062, 2065, 2067, 2069, 2072, 2076, 2084, 2086, 2091, 2093, 2095, 2097, 2106, 2373, 2382
Point_d_iterator, 2072
Point_handle, 1661, 1673
point_is_in
 Arr_accessor, 1211
point_of_facet
 Convex_hull_d, 689

- point_of_simplex*
 - Convex_hull_d*, 689
 - Delaunay_d*, 702
- Point_set_2*, 2013–2015
- Point_type*, 973
- Point_with_transformed_distance*, 2065, 2067, 2084, 2086
- pointer*, 2746, 2762–2770
 - Compact_container_traits*, 2609
- PointGenerator*, 2750
- points*
 - Arr_polyline_traits_2<SegmentTraits>::Curve_2*, 1269
- points_begin*
 - Convex_hull_d*, 692
 - Delaunay_d*, 704
 - Min_annulus_d*, 2246
 - Min_circle_2*, 2194
 - Min_ellipse_2*, 2205
 - Min_sphere_d*, 2239
 - Polyhedron_3*, 802
 - Sphere_d*, 469
 - Triangulation_2*, 1437
 - Triangulation_3*, 1517
- points_end*
 - Convex_hull_d*, 692
 - Delaunay_d*, 704
 - Min_annulus_d*, 2246
 - Min_circle_2*, 2195
 - Min_ellipse_2*, 2205
 - Min_sphere_d*, 2239
 - Polyhedron_3*, 803
 - Sphere_d*, 470
 - Triangulation_2*, 1437
 - Triangulation_3*, 1517
- points_on_cube_grid_3*, 2749
- Points_on_segment_2*, 2738, 2746–2747
- points_on_segment_2*, 2745
- points_on_square_grid_2*, 2748
- points_p_begin*
 - Polytope_distance_d*, 2316
- points_p_end*
 - Polytope_distance_d*, 2316
- points_q_begin*
 - Polytope_distance_d*, 2316
- points_q_end*
 - Polytope_distance_d*, 2316
- PointSetTraits*, 2016
- pointSize*
 - Qt_widget*, 2844
- PointStyle*, 2841
- pointStyle*
 - Qt_widget*, 2844
- polygon*
 - largest inscribed, 2287, 2289
 - strongly convex, 612, 648–649
 - valid, 754
- polygon convex*
 - see also* convex polygon
- polygon partitioning*, 737
 - assertion flags, 737
 - convex, 736
 - approximately optimal, 736, 740, 746
 - optimal, 736, 756
 - valid, 743, 761
 - valid, 754, 761, 771
 - y-monotone, 735, 749, 772
 - valid, 761, 775
- polygon y-monotone*
 - see also* y-monotone polygon
- Polygon_2*, 726–731, 936
- polygon_area_2*, 732
- Polygon_offset_builder_2*, 1095–1097
- Polygon_offset_builder_traits_2*, 1094
- Polygon_set_2*, 934
- Polygon_with_holes_2*, 933
- PolygonIsValid*, 771
 - model, 753–755
- PolygonOffsetBuilderTraits_2*, 1084–1085
- polygons*
 - assertion flags, 713
- polygons_with_holes*
 - General_polygon_set_2*, 918
- PolygonTraits_2*, 715–725, 732–734
- polyhedron*
 - strongly convex, 678
 - see also* width of 3D point set
- Polyhedron_3*, 799–810
- Polyhedron_incremental_builder_3*, 818–821
- Polyhedron_items_3*, 824–825
- Polyhedron_min_items_3*, 826
- Polyhedron_traits_3*, 828–829
- Polyhedron_traits_with_normals_3*, 830–831
- PolyhedronItems_3*, 822–823
- PolyhedronTraits_3*, 827
- Polynomial_1_2*, 592
- Polynomial_for_circles_2_2*, 594
- polytope*
 - distance of polytopes, 2314
- Polytope_distance_d*, 2314–2320
 - creation, 2315
 - global functions, 2320
 - output, 2320
 - implementation, 2320
 - member functions, 2315–2320
 - access, 2315
 - miscellaneous, 2320
 - modifiers, 2317

- predicates, [2317](#)
 - validity check, [2319](#)
- requirements, [2314](#)
- traits class
 - requirements, [2286](#)
 - see also [Optimisation_d_traits_2](#)
 - see also [Optimisation_d_traits_3](#)
 - see also [Optimisation_d_traits_d](#)
- types, [2314](#)
- pop*
 - Kinetic::RootStack, [2500](#)
- pop_back*
 - In_place_list, [2605](#)
- pop_failure_time*
 - Kinetic::Certificate, [2483](#)
- pop_front*
 - In_place_list, [2605](#)
- POSITIVE*, [124](#)
- post_move*
 - Kinetic::RegularTriangulationVisitor3, [2454](#)
- post_pop*
 - Kinetic::RegularTriangulationVisitor3, [2454](#)
- post_push*
 - Kinetic::RegularTriangulationVisitor3, [2454](#)
- Power diagram, [1533](#)
- power diagram, [1361](#), [1416](#)
- power_test*
 - Regular_triangulation_2, [1417](#)
- Power_test_2*, [2385](#)
- power_test_2_object*
 - RegularTriangulationTraits_2, [1409](#)
- power_test_3_object*
 - Regular_triangulation_euclidean_traits_3, [1544](#)
 - RegularTriangulationTraits_3, [1541](#)
- PQQ*, [1883](#)
- PQQMask_3*, [1886](#)
- pre_move*
 - Kinetic::RegularTriangulationVisitor3, [2454](#)
- pre_pop*
 - Kinetic::RegularTriangulationVisitor3, [2454](#)
- pre_push*
 - Kinetic::RegularTriangulationVisitor3, [2454](#)
- precision*
 - Real_timer, [2778](#)
 - Timer, [2779](#)
- predecessor*, [2621](#)
- predicates
 - Approximate_min_ellipsoid_d, [2269](#)
 - Min_annulus_d, [2247](#)
 - Min_circle_2, [2195](#)
 - Min_ellipse_2, [2205](#)
 - Min_sphere_d, [2240](#)
 - Min_sphere_of_spheres_d, [2253](#)
 - Polytope_distance_d, [2317](#)
- PRETTY*, [2785](#), [2798](#)
- prev*
 - ArrangementDcelHalfedge, [1241](#)
 - ConvexHullPolyhedronHalfedge_3, [670](#)
 - Halfedge, [814](#), [1217](#)
 - HalfedgeDSHalfedge, [858](#)
 - SHalfedge, [1046](#)
- prev_link*, [2602](#)
- prev_on_vertex*
 - Halfedge, [814](#)
- previous*
 - Halfedge, [1758](#)
 - Topological_explorer, [967](#)
- primary_bisector*
 - StraightSkeletonVertex_2, [1076](#)
- print*
 - Qt_help_window, [2868](#)
- print_statistics*
 - Convex_hull_d, [691](#)
 - Nef_polyhedron_S2, [994](#)
 - Topological_explorer, [969](#)
- print_stream_lines*
 - Stream_lines_2, [2408](#)
- print_to_ps*
 - Qt_widget, [2843](#)
- priority*
 - Kinetic::EventQueue, [2485](#)
- process*
 - Event, [2487](#)
- process_next*
 - Kinetic::EventQueue, [2486](#)
- project_along_d_axis*, [495](#)
- Project_facet*, [2673](#)
- Project_next*, [2677](#)
- Project_next_opposite*, [2679](#)
- Project_normal*, [2675](#)
- Project_opposite_prev*, [2680](#)
- Project_plane*, [2676](#)
- Project_point*, [2674](#)
- Project_prev*, [2678](#)
- project_triangle*
 - LSCM_parameterizer_3, [1952](#)
- Project_vertex*, [2672](#)
- projection*
 - Line_2, [73](#)
 - Line_3, [97](#)
 - Line_d, [459](#)
 - Plane_3, [100](#)
- Projection_object, [2667](#)
- propagating_flip*
 - Constrained_Delaunay_triangulation_2, [1387](#)
- PTQ*, [1883](#)
- PTQMask_3*, [1887](#)

- push_back*
 - Arr_polyline_traits_2*<*SegmentTraits*>
::*Curve_2*, 1269
 - Constrained_Delaunay_triangulation_2*,
1384, 1385
 - Constrained_triangulation_2*, 1390, 1391
 - Constrained_triangulation_plus_2*, 1396
 - Delaunay_triangulation_2*, 1402
 - In_place_list*, 2605
 - Inverse_index*, 2634
 - Largest_empty_iso_rectangle_2*, 2299
 - Polygon_2*, 728
 - Random_access_adaptor*, 2635
 - Regular_triangulation_2*, 1413
 - Triangulation_2*, 1434
 - Union_find*, 2780
 - VertexContainer_2*, 1086
- push_front*
 - In_place_list*, 2605
- Qt_help_window*, 2868
- Qt_widget*, 2841–2850
- Qt_widget_get_circle*, 2857
- Qt_widget_get_iso_rectangle*, 2858
- Qt_widget_get_line*, 2856
- Qt_widget_get_point*, 2854
- Qt_widget_get_polygon*, 2859–2860
- Qt_widget_get_segment*, 2855
- Qt_widget_get_simple_polygon*, 2861
- Qt_widget_history*, 2867
- Qt_widget_layer*, 2851–2853
- Qt_widget_Nef_3*, 1055–1056
- Qt_widget_Nef_S2*, 1009–1010
- Qt_widget_standard_toolbar*, 2864–2866
- quadratic program
 - Polytope_distance_d*, 2320
- quadratic_interpolation, 2369
- Quadruple, 2701–2702
- query_circulator_or_iterator, 2732
- Query_item, 2054, 2065, 2067, 2076, 2084, 2086
- quickhull, 2D, 621
- quickhull, 3D, 655, 664–666
- Quotient, 2573–2574
- quotient_cartesian_to_homogeneous, 194
- r*
 - Arr_conic_traits_2*<*RatKernel*, *AlgKernel*,
NtTraits>::*Curve_2*, 1281
- radius
 - Min_sphere_of_spheres_d*, 2253
 - Min_sphere_of_spheres_d_traits_2*, 2260
 - Min_sphere_of_spheres_d_traits_3*, 2262
 - Min_sphere_of_spheres_d_traits_d*, 2264
 - MinSphereOfSpheresTraits*, 2258
- Random*, 2757–2758
- random convex set, 2752
- random perturbations, 2733
- random simple polygon, 2754
- Random_access_adaptor*, 2635
- Random_access_circulator*, 2715
- Random_access_circulator_base*, 2722
- Random_access_circulator_ptrbase*, 2724
- Random_access_circulator_tag*, 2722
- Random_access_value_adaptor*, 2636
- random_collinear_points_2*, 2751
- random_convex_set*
 - preconditions, 2752
 - traits class, 2761
 - default, 2752
 - traits requirements, 2759
- random_convex_set_2*, 2752–2753
- Random_convex_set_traits_2*, 2761
- Random_points_in_cube_3*, 2762
- Random_points_in_disc_2*, 2763
- Random_points_in_iso_box_d*, 2770
- Random_points_in_sphere_3*, 2764
- Random_points_in_square_2*, 2765
- Random_points_on_circle_2*, 2766
- Random_points_on_segment_2*, 2767
- Random_points_on_sphere_3*, 2768
- Random_points_on_square_2*, 2769
- random_polygon_2*, 2754–2755
 - traits class
 - default, 2754
 - traits requirements, 2760
- random_selection*, 2756
- RandomConvexSetTraits_2*, 2759
 - model, 2761
- randomization
 - Min_circle_2*, 2197
 - Min_ellipse_2*, 2208
- RandomPolygonTraits_2*, 2760
- range
 - PointGenerator*, 2750
 - Points_on_segment_2*, 2746
- range_search, 2020–2024
 - Delaunay_d*, 703
 - Point_set_2*, 2014, 2015
- Range_segment_tree_traits_set_2*, 2123
- Range_segment_tree_traits_set_3*, 2124
- Range_tree_d*, 2125–2127
- Range_tree_k*, 2128–2129
- Range_tree_traits_map_2*, 2130
- Range_tree_traits_map_3*, 2131
- rangesChanged
 - Qt_widget*, 2846
- RangeSegmentTreeTraits_k*, 2121–2122
- rank
 - LinearAlgebraTraits_d*, 439

rasterOp
 Qt_widget, 2845
rational_rotation_approximation, 195
Rational_traits, 2575
raw_cells_begin
 TriangulationDataStructure_3, 1582
raw_cells_end
 TriangulationDataStructure_3, 1582
Ray, 1412, 1520
ray
 Explorer, 971
Ray_2, 79–80, 2384
Ray_3, 105–106
Ray_d, 460–461
ray_shoot
 Nef_polyhedron_2, 962
 Nef_polyhedron_S2, 995
ray_shoot_down
 ArrangementVerticalRayShoot_2, 1306
ray_shoot_to_boundary
 Nef_polyhedron_2, 962
 Nef_polyhedron_S2, 995
ray_shoot_up
 ArrangementVerticalRayShoot_2, 1305
rbegin
 Arr_polyline_traits_2<SegmentTraits>
 ::Curve_2, 1269
 Compact_container, 2612
 Multiset, 2616
read, 1234
read_arrangement_begin
 ArrangementInputFormatter, 1293
read_arrangement_end
 ArrangementInputFormatter, 1293
read_ccb_halfedges_begin
 ArrangementInputFormatter, 1295
read_ccb_halfedges_end
 ArrangementInputFormatter, 1295
read_curve
 ArrWithHistoryInputFormatter, 1323
read_curve_begin
 ArrWithHistoryInputFormatter, 1323
read_curve_end
 ArrWithHistoryInputFormatter, 1323
read_curves_begin
 ArrWithHistoryInputFormatter, 1323
read_curves_end
 ArrWithHistoryInputFormatter, 1323
read_edge_begin
 ArrangementInputFormatter, 1294
read_edge_end
 ArrangementInputFormatter, 1294
read_edges_begin
 ArrangementInputFormatter, 1293
 ArrangementInputFormatter, 1293
read_edges_end
 ArrangementInputFormatter, 1293
read_face_begin
 ArrangementInputFormatter, 1294
read_face_data
 ArrangementInputFormatter, 1295
read_face_end
 ArrangementInputFormatter, 1294
read_faces_begin
 ArrangementInputFormatter, 1293
read_faces_end
 ArrangementInputFormatter, 1293
read_halfedge_index
 ArrangementInputFormatter, 1294
read_halfedge_data
 ArrangementInputFormatter, 1294
read_holes_begin
 ArrangementInputFormatter, 1294
read_holes_end
 ArrangementInputFormatter, 1294
read_induced_edges_begin
 ArrWithHistoryInputFormatter, 1323
read_induced_edges_end
 ArrWithHistoryInputFormatter, 1323
read_inner_ccb_begin
 ArrangementInputFormatter, 1294
read_inner_ccb_end
 ArrangementInputFormatter, 1295
read_isolated_vertices_begin
 ArrangementInputFormatter, 1295
read_isolated_vertices_end
 ArrangementInputFormatter, 1295
read_outer_ccb_begin
 ArrangementInputFormatter, 1294
read_outer_ccb_end
 ArrangementInputFormatter, 1294
read_point
 ArrangementInputFormatter, 1293
read_size
 ArrangementInputFormatter, 1293
read_vertex_begin
 ArrangementInputFormatter, 1293
read_vertex_data
 ArrangementInputFormatter, 1293
read_vertex_end
 ArrangementInputFormatter, 1293
read_vertex_index
 ArrangementInputFormatter, 1293
read_vertices_begin
 ArrangementInputFormatter, 1293
read_vertices_end
 ArrangementInputFormatter, 1293
read_x_monotone_curve
 ArrangementInputFormatter, 1294

- Real_timer*, 2778
- realizing_point_p*
 - Polytope_distance_d*, 2316
- realizing_point_p_coordinates_begin*
 - Polytope_distance_d*, 2317
- realizing_point_p_coordinates_end*
 - Polytope_distance_d*, 2317
- realizing_point_q*
 - Polytope_distance_d*, 2316
- realizing_point_q_coordinates_begin*
 - Polytope_distance_d*, 2317
- realizing_point_q_coordinates_end*
 - Polytope_distance_d*, 2317
- RECT*, 2841
- rectangle*
 - smallest enclosing, 2215
- rectangular centers*, 2229
- rectangular_p_center_2*, 2229–2231
- Rectangular_p_center_default_traits_2*, 2232–2234
- RectangularPCenterTraits_2*, 2235–2237
- rectilinear centers*, 2229
- redraw*
 - Qt_widget*, 2843
- redraw_on_back*
 - Qt_widget*, 2846
- redraw_on_front*
 - Qt_widget*, 2846
- Ref_counted*, 2499
- reference*, 2746, 2762–2770
- refine_Delaunay_mesh_2*, 1816
- refine_mesh*
 - Delaunay_mesher_2*, 1806
- Reflection*, 132
- REGULAR*, 1605, 1633, 1847
- Regular_grid_2*, 2403–2404
- regular_neighbor_coordinates_2*, 2376–2377
- Regular_triangulation_2*, 1411–1417
- Regular_triangulation_3*, 1528–1533
- Regular_triangulation_adaptation_traits_2*, 1774
- Regular_triangulation_caching_degeneracy_removal_policy_2*, 1783
- Regular_triangulation_cell_base_3*, 1557
- Regular_triangulation_degeneracy_removal_policy_2*, 1779
- Regular_triangulation_euclidean_traits_2*, 1418
- Regular_triangulation_euclidean_traits_3*, 1542–1544
- Regular_triangulation_face_base_2*, 1420
- Regular_triangulation_filtered_traits_2*, 1419
- Regular_triangulation_filtered_traits_3*, 1545
- Regular_triangulation_vertex_base_2*, 1421
- regularization*
 - Nef_polyhedron_2*, 961
 - Nef_polyhedron_3*, 1037
 - Nef_polyhedron_S2*, 993
- REGULARIZED*, 1606, 1633
- RegularTriangulationCellBase_3*, 1550–1551
- RegularTriangulationFaceBase_2*, 1407
- RegularTriangulationTraits_2*, 1408–1409
 - model, 2384
- RegularTriangulationTraits_3*, 1539–1541
- RegularTriangulationVertexBase_2*, 1410
- RELATIVE_INDEXING*, 818
- relocate_holes_in_new_face*
 - Arr_accessor*, 1214
- relocate_in_new_face*
 - Arr_accessor*, 1213
- relocate_isolated_vertices_in_new_face*
 - Arr_accessor*, 1214
- remove*
 - Apollonius_graph_2*, 1717
 - Apollonius_graph_hierarchy_2*, 1734
 - Constrained_Delaunay_triangulation_2*, 1385
 - Constrained_triangulation_2*, 1391
 - Delaunay_triangulation_2*, 1402
 - Delaunay_triangulation_3*, 1522
 - In_place_list*, 2606
 - Interval_skip_list*, 2031
 - Largest_empty_iso_rectangle_2*, 2300
 - Regular_triangulation_2*, 1413
 - Regular_triangulation_3*, 1530
 - Triangulation_2*, 1434
- remove_cells*
 - Kinetic::DelaunayTriangulationVisitor3*, 2436
 - Kinetic::RegularTriangulationVisitor3*, 2453
- remove_constrained_edge*
 - Constrained_triangulation_2*, 1391
- remove_constraint*
 - Constrained_Delaunay_triangulation_2*, 1385
 - Constrained_triangulation_plus_2*, 1396
- remove_curve*, 1322
- remove_decrease_dimension*
 - TriangulationDataStructure_3*, 1580
- remove_degree_2*
 - ApolloniusGraphDataStructure_2*, 1723
 - Triangulation_data_structure_2*, 1480
- remove_degree_3*
 - Triangulation_2*, 1436
 - TriangulationDataStructure_2*, 1467
- remove_dim_down*
 - TriangulationDataStructure_2*, 1467
- remove_edge*, 1227
 - Arrangement_2*, 1208
 - Arrangement_with_history_2*, 1321
- remove_edge_ex*
 - Arr_accessor*, 1215
- remove_faces*

Kinetic::DelaunayTriangulationVisitor2, [2435](#)
remove_first
 Triangulation_2, [1436](#)
 TriangulationDataStructure_2, [1467](#)
remove_from_complex
 SurfaceMeshComplex_2InTriangulation_3, [1849](#)
remove_from_maximal_dimension_simplex
 TriangulationDataStructure_3, [1580](#)
remove_halfedge
 HalfedgeDS_items_decorator, [883](#)
remove_incident_constraints
 Constrained_Delaunay_triangulation_2, [1385](#)
 Constrained_triangulation_2, [1391](#)
remove_isolated_vertex
 Arrangement_2, [1207](#)
remove_second
 Triangulation_2, [1436](#)
 TriangulationDataStructure_2, [1467](#)
remove_tip
 HalfedgeDS_items_decorator, [882](#)
remove_unconnected_vertices
 Polyhedron_incremental_builder_3, [820](#)
remove_vertex, [1228](#)
 Kinetic::DelaunayTriangulationVisitor2, [2435](#)
 Kinetic::DelaunayTriangulationVisitor3, [2436](#)
 Kinetic::RegularTriangulationVisitor3, [2453](#)
 Kinetic::SortVisitor, [2459](#)
rend
 Arr_polyline_traits_2<SegmentTraits>
 ::Curve_2, [1269](#)
 Compact_container, [2612](#)
 Multiset, [2616](#)
reorient
 ConstrainedTriangulationFaceBase_2, [1379](#)
 TriangulationDataStructure_3, [1580](#)
 TriangulationDSFaceBase_2, [1472](#)
replace
 Multiset, [2618](#)
replace_column
 Dynamic_matrix, [2323](#)
 MonotoneMatrixSearchTraits, [2325](#)
Representation, [2777](#)
reserve
 HalfedgeDS, [850](#)
 Polyhedron_3, [801](#)
 Random_access_adaptor, [2635](#)
reset
 Real_timer, [2778](#)
 Timer, [2779](#)
resident_size

Memory_sizer, [2776](#)
restore_state
 Random, [2758](#)
result_type, [46](#), [2784](#)
reverse
 In_place_list, [2606](#)
reverse_orientation
 General_polygon_2, [932](#)
 Polygon_2, [728](#)
RIGHT, [123](#)
right
 Arr_circle_segment_traits_2<Kernel>::X_monotone_curve_2, [1276](#)
 Arr_conic_traits_2<RatKernel, AlgKernel, NtTraits>::X_monotone_curve_2, [1282](#)
 Arr_rational_arc_traits_2<AlgKernel, NtTraits>::Curve_2, [1285](#)
 Circular_arc_2, [554](#)
 Halfedge, [1759](#)
 Line_arc_2, [556](#)
RIGHT_TURN, [127](#)
right_turn, [196](#)
right_vertex
 Polygon_2, [729](#)
right_vertex_2, [733](#)
 requirements, [733](#)
RIGHTFRAME, [973](#)
Ring_tag, [2579](#)
RingNumberType, [2576–2578](#)
rollback
 Polyhedron_incremental_builder_3, [820](#)
root
 Kd_tree, [2070](#)
 SpatialTree, [2103](#)
Root_for_circles_2_2, [590](#)
Root_of_2, [2582](#)
Root_of_traits_2, [2581](#)
RootOf_2, [2580](#)
rotating caliper, [2215](#), [2217](#), [2219](#)
Rotation, [133](#)
row
 Matrix, [443](#)
row_begin
 Matrix, [443](#)
row_dimension
 Matrix, [443](#), [1954](#)
 Taucs_matrix, [1993](#)
row_end
 Matrix, [443](#)
RT, [41](#), [48](#), [55](#), [56](#), [548](#), [1559](#), [2306](#), [2310](#), [2858](#)
Runge_kutta_integrator_2, [2405](#)

s

Arr_conic_traits_2<RatKernel, AlgKernel, NtTraits>::Curve_2, 1281
s_enterEvent
 Qt_widget, 2846
s_event
 Qt_widget, 2846
s_keyPressEvent
 Qt_widget, 2846
s_keyReleaseEvent
 Qt_widget, 2846
s_leaveEvent
 Qt_widget, 2846
s_mouseDoubleClickEvent
 Qt_widget, 2846
s_mouseMoveEvent
 Qt_widget, 2846
s_mousePressEvent
 Qt_widget, 2846
s_mouseReleaseEvent
 Qt_widget, 2846
s_paintEvent
 Qt_widget, 2846
s_resizeEvent
 Qt_widget, 2846
s_wheelEvent
 Qt_widget, 2846
same_set
 Union_find, 2781
save
 Qt_widget_history, 2867
save_state
 Random, 2758
Scaling, 133
SE
 ExtendedKernelTraits_2, 974
search
 Kd_tree, 2070
 Kd_tree_node, 2072
 SpatialTree, 2103
Search_traits, 2097–2098
Search_traits_2, 2091–2092
Search_traits_3, 2093–2094
Search_traits_d, 2095–2096
searching
 in monotone matrices, 2321
 in sorted matrices, 2328
SearchTraits, 2090
second, 2699, 2701
second_type, 2699, 2701
SECORNER, 973
seeds_begin
 Delaunay_mesher_2, 1806
seeds_end
 Delaunay_mesher_2, 1806
Segment, 1429, 1504, 2013
segment
 SegmentDelaunayGraphSite_2, 1670
 Triangulation_2, 1440
 Triangulation_3, 1508
Segment2, 2859, 2861
Segment_2, 81–82, 726, 1445, 2384, 2855
Segment_3, 107–108
Segment_d, 462–464
Segment_Delaunay_graph_2, 1661–1668
Segment_Delaunay_graph_adaptation_traits_2, 1775
Segment_Delaunay_graph_caching_degeneracy_removal_policy_2, 1784
Segment_Delaunay_graph_degeneracy_removal_policy_2, 1780
Segment_Delaunay_graph_filtered_traits_2, 1690–1691
Segment_Delaunay_graph_filtered_traits_without_intersections_2, 1692–1693
Segment_Delaunay_graph_hierarchy_2, 1694–1695
Segment_Delaunay_graph_hierarchy_vertex_base_2, 1697
Segment_Delaunay_graph_site_2, 1672
Segment_Delaunay_graph_storage_site_2, 1677
Segment_Delaunay_graph_traits_2, 1688
Segment_Delaunay_graph_traits_without_intersections_2, 1689
Segment_Delaunay_graph_vertex_base_2, 1682
Segment_tree_d, 2132–2133
Segment_tree_k, 2134–2136
Segment_tree_traits_map_2, 2137
Segment_tree_traits_map_3, 2138
SegmentDelaunayGraphDataStructure_2, 1678–1679
SegmentDelaunayGraphHierarchyVertexBase_2, 1696
SegmentDelaunayGraphSite_2, 1669–1671
SegmentDelaunayGraphStorageSite_2, 1673–1676
SegmentDelaunayGraphTraits_2, 1683–1687
SegmentDelaunayGraphVertexBase_2, 1680–1681
Segments_in_hierarchy_tag, 1695
Self, 1201
self-intersection
 iso-oriented boxes, 2166
Separator, 2072
separator
 Kd_tree_node, 2073
set
 Kinetic::ActiveObjectsTable, 2478
 Kinetic::EventQueue, 2485
 Min_annulus_d, 2248
 Min_sphere_d, 2240

Min_sphere_of_spheres_d, 2254
Polygon_2, 727
Polytope_distance_d, 2317
set_alpha
 Alpha_shape_2, 1606
 Alpha_shape_3, 1635
 AlphaShapeCell_3, 1630
 AlphaShapeFace_2, 1612
set_alpha_max
 Alpha_status, 1641
set_alpha_mid
 Alpha_status, 1641
set_alpha_min
 Alpha_status, 1641
Set_arity, 2660
set_arity_0, 2661
set_arity_1, 2662
set_arity_2, 2663
set_arity_3, 2664
set_arity_4, 2665
set_arity_5, 2666
set_ascii_mode, 2785, 2801
 Geomview_stream, 2821
set_bad_faces
 Delaunay_mesher_2, 1806
set_bg_color
 Geomview_stream, 2819
set_binary_mode, 2785, 2802
 Geomview_stream, 2821
set_cell
 TriangulationDSVertexBase_3, 1593
 Vertex, 1587
set_center
 Qt_widget, 2842
set_coef
 Matrix, 1954
 Taucs_matrix, 1994
set_constraint
 ConstrainedTriangulationFaceBase_2, 1379
set_constraints
 ConstrainedTriangulationFaceBase_2, 1379
set_corners_index
 Parameterization_polyhedron_adaptor_3, 1977
 ParameterizationPatchableMesh_3, 1963
set_corners_parameterized
 Parameterization_polyhedron_adaptor_3, 1977
 ParameterizationPatchableMesh_3, 1963
set_corners_tag
 Parameterization_polyhedron_adaptor_3, 1977
 ParameterizationPatchableMesh_3, 1963
set_corners_uv
 Parameterization_polyhedron_adaptor_3, 1977
 ParameterizationPatchableMesh_3, 1963
set_criteria
 Delaunay_mesher_2, 1806
set_current_event_number
 Kinetic::Simulator, 2508
set_current_time
 Kinetic::Simulator, 2507
set_curve
 ArrangementDcelHalfedge, 1242
set_cutting_dimension
 Plane_separator, 2088
 SpatialSeparator, 2102
set_cutting_value
 Plane_separator, 2088
 SpatialSeparator, 2102
set_data
 Arr_curve_data_traits_2<Tr, XData,Mrg,CData,Cnv>::Curve_2, 1287
 Arr_curve_data_traits_2<Tr, XData,Mrg,CData,Cnv>::X_monotone_curve_2, 1288
 Arr_extended_face, 1255
 Arr_extended_halfedge, 1254
 Arr_extended_vertex, 1253
set_dimension
 TriangulationDataStructure_2, 1465
 TriangulationDataStructure_3, 1575
set_direction
 ArrangementDcelHalfedge, 1242
set_direction_of_time
 Kinetic::Simulator, 2508
set_down
 ApolloniusGraphHierarchyVertexBase_2, 1736
 SegmentDelaunayGraphHierarchyVertexBase_2, 1696
 TriangulationHierarchyVertexBase_2, 1423
 TriangulationHierarchyVertexBase_3, 1549
set_echo
 Geomview_stream, 2821
set_edge_color
 Geomview_stream, 2819
set_end_priority
 Kinetic::EventQueue, 2486
set_end_time
 Kinetic::Simulator, 2507
set_error_behaviour, 8
set_error_handler, 10
set_event
 Kinetic::Simulator, 2508
set_face

- ApolloniusGraphVertexBase_2, 1725
- ArrangementDcelHalfedge, 1242
- ArrangementDcelHole, 1246
- ArrangementDcelIsolatedVertex, 1247
- HalfedgeDS_items_decorator, 883
- HalfedgeDSHalfedge, 858
- SegmentDelaunayGraphVertexBase_2, 1681
- TriangulationDataStructure_2::Vertex, 1478
- set_face_color
 - Geomview_stream, 2819
- set_face_halfedge
 - HalfedgeDS_items_decorator, 883
- set_face_in_face_loop
 - HalfedgeDS_items_decorator, 883
- set_facet_on_surface
 - SurfaceMeshCellBase_3, 1844
- set_facet_status
 - AlphaShapeCell_3, 1630
- set_facet_surface_center
 - SurfaceMeshCellBase_3, 1845
- set_facet_visited
 - SurfaceMeshCellBase_3, 1844
- set_halfedge
 - ArrangementDcelFace, 1245
 - ArrangementDcelVertex, 1239
 - Facet, 812
 - HalfedgeDSFace, 855
 - HalfedgeDSVertex, 861
 - Vertex, 817
- set_halfedge_seaming
 - Parameterization_polyhedron_adaptor_3, 1977
 - ParameterizationPatchableMesh_3, 1963
- set_has_certificates
 - Kinetic::Delaunay_triangulation_3, 2429
- set_hidden
 - RegularTriangulationVertexBase_2, 1410
- set_hole
 - ArrangementDcelHalfedge, 1242
- set_in
 - ArrangementInputFormatter, 1292
- set_in_conflict_flag
 - TriangulationDSCellBase_3, 1591
- set_in_domain
 - DelaunayMeshFaceBase_2, 1803
- set_infinite_vertex
 - Triangulation_2, 1440
- set_is_editing
 - Kinetic::ActiveObjectsTable, 2478
- set_is_Gabriel
 - Alpha_status, 1641
- set_is_on_chull
 - Alpha_status, 1641
- set_isolated_vertex
 - ArrangementDcelVertex, 1239
- set_iterator
 - ArrangementDcelHole, 1246
 - ArrangementDcelIsolatedVertex, 1247
- set_line_width
 - Geomview_stream, 2820
- set_lower_bound
 - Kd_tree_rectangle, 2074
- set_mesh_uv_from_system
 - Fixed_border_parameterizer_3, 1949
 - LSCM_parameterizer_3, 1953
- set_mode, 2785, 2803
 - Alpha_shape_2, 1607
 - Alpha_shape_3, 1635
- set_neighbor
 - Cell, 1586
 - TriangulationDataStructure_2::Face, 1475
 - TriangulationDSCellBase_3, 1590
 - TriangulationDSFaceBase_2, 1472
- set_neighbors
 - Cell, 1586
 - TriangulationDataStructure_2::Face, 1475
 - TriangulationDSCellBase_3, 1590
 - TriangulationDSFaceBase_2, 1472
- set_next
 - ArrangementDcelHalfedge, 1242
 - HalfedgeDSHalfedge, 858
- set_opposite
 - ArrangementDcelHalfedge, 1242
 - HalfedgeDSHalfedge, 858
- set_out
 - ArrangementOutputFormatter, 1296
- set_p
 - Polytope_distance_d, 2318
- set_point
 - ArrangementDcelVertex, 1240
 - TriangulationDataStructure_2::Vertex, 1478
 - TriangulationVertexBase_2, 1426
 - TriangulationVertexBase_3, 1547
 - Vertex, 1587
- set_pretty_mode, 2785, 2804
- set_prev
 - ArrangementDcelHalfedge, 1242
 - HalfedgeDS_items_decorator, 883
 - HalfedgeDSHalfedge, 858
- set_q
 - Polytope_distance_d, 2318
- set_range
 - AlphaShapeVertex_2, 1616
- set_ranges
 - AlphaShapeFace_2, 1612
- set_raw
 - Geomview_stream, 2821
- set_relative_precision_of_to_double

Lazy_exact_nt, 2557
set_saturation_ratio
 Stream_lines_2, 2407
set_seeds
 Delaunay_mesher_2, 1806
set_separating_distance
 Stream_lines_2, 2407
set_site
 ApolloniusGraphVertexBase_2, 1725
 SegmentDelaunayGraphVertexBase_2, 1681
set_source
 Arr_conic_traits_2<RatKernel, AlgKernel, NtTraits>::Curve_2, 1281
set_target
 Arr_conic_traits_2<RatKernel, AlgKernel, NtTraits>::Curve_2, 1281
set_time
 Kinetic::InstantaneousKernel, 2492
set_trace
 Geomview_stream, 2821
set_up
 ApolloniusGraphHierarchyVertexBase_2, 1736
 SegmentDelaunayGraphHierarchyVertexBase_2, 1696
 TriangulationHierarchyVertexBase_2, 1423
 TriangulationHierarchyVertexBase_3, 1549
set_upper_bound
 Kd_tree_rectangle, 2074
set_vertex
 ArrangementDcelHalfedge, 1242
 Cell, 1586
 HalfedgeDS_items_decorator, 883
 HalfedgeDSHalfedge, 858
 TriangulationDataStructure_2::Face, 1475
 TriangulationDSCellBase_3, 1590
 TriangulationDSFaceBase_2, 1472
set_vertex_color
 Geomview_stream, 2819
set_vertex_halfedge
 HalfedgeDS_items_decorator, 883
set_vertex_in_vertex_loop
 HalfedgeDS_items_decorator, 883
set_vertex_index
 Parameterization_mesh_patch_3, 1971
 Parameterization_polyhedron_adaptor_3, 1976
 ParameterizationMesh_3, 1961
set_vertex_parameterized
 Parameterization_mesh_patch_3, 1971
 Parameterization_polyhedron_adaptor_3, 1976
 ParameterizationMesh_3, 1961
set_vertex_radius
 Geomview_stream, 2820
set_vertex_seaming
 Parameterization_polyhedron_adaptor_3, 1977
 ParameterizationPatchableMesh_3, 1962
set_vertex_tag
 Parameterization_mesh_patch_3, 1971
 Parameterization_polyhedron_adaptor_3, 1976
 ParameterizationMesh_3, 1961
set_vertex_uv
 Parameterization_mesh_patch_3, 1970
 Parameterization_polyhedron_adaptor_3, 1976
 ParameterizationMesh_3, 1961
set_vertices
 Cell, 1586
 TriangulationDataStructure_2::Face, 1475
 TriangulationDSCellBase_3, 1590
 TriangulationDSFaceBase_2, 1472
set_warning_behaviour, 9
set_warning_handler, 10
set_window
 Qt_widget, 2841
set_wired
 Geomview_stream, 2820
set_x_scale
 Qt_widget, 2841
set_xy
 Regular_grid_2, 2403
set_y_scale
 Qt_widget, 2841
setBackgroundColor
 Qt_widget, 2843
setColor
 Qt_widget, 2843
setFillColor
 Qt_widget, 2843
setFilled
 Qt_widget, 2843
setLineWidth
 Qt_widget, 2843
setPointSize
 Qt_widget, 2843
setPointSize
 Qt_widget, 2844
setRasterOp
 Qt_widget, 2844
setup_inner_vertex_relations
 Fixed_border_parameterizer_3, 1949
setup_triangle_relations
 LSCM_parameterizer_3, 1953
SFace, 1007, 1049
sface_cycle_begin

SFace, 1007, 1049
sface_cycle_end
SFace, 1007, 1049
SFace_cycle_iterator, 1008, 1051
Sgn, 2589
SHalfedge, 1003–1004, 1045–1046
SHalfloop, 1005–1006, 1047–1048
shalfloop
Nef_polyhedron_S2, 995
shells_begin
Volume, 1044
shells_end
Volume, 1044
shrink_to_quadratic_size
Dynamic_matrix, 2324
MonotoneMatrixSearchTraits, 2325
sibson_c1_interpolation, 2366–2367
sibson_c1_interpolation_square, 2367
sibson_gradient_fitting, 2378–2379
sibson_gradient_fitting_nn_2, 2378
sibson_gradient_fitting_rm_2, 2379
side_of_bounded_circle, 197
side_of_bounded_circle_2_object
AlphaShapeTraits_2, 1614
side_of_bounded_orthogonal_sphere_3_object
Regular_triangulation_euclidean_traits_3, 1544
side_of_bounded_sphere, 198, 496
side_of_cell
Triangulation_3, 1511
side_of_circle
Delaunay_triangulation_3, 1522, 1523
side_of_edge
Triangulation_3, 1511, 1512
side_of_facet
Triangulation_3, 1511
side_of_oriented_circle, 199
Delaunay_triangulation_2, 1404
Triangulation_2, 1433
side_of_oriented_circle_2_object
DelaunayTriangulationTraits_2, 1400
Triangulation_euclidean_traits_xy_3, 1446
TriangulationTraits_2, 1425
side_of_oriented_circleC2, 2536
side_of_oriented_sphere, 200, 497
side_of_oriented_sphere_3_object
DelaunayTriangulationTraits_3, 1537
side_of_oriented_sphereC3, 2536
side_of_power_circle
Regular_triangulation_3, 1530, 1531
side_of_power_segment
Regular_triangulation_3, 1531
side_of_power_sphere
Regular_triangulation_3, 1530
side_of_sphere
Delaunay_triangulation_3, 1522
Sign, 124
sign, 2546, 2583
sign_of_determinant
LinearAlgebraTraits_d, 438
signed_inf_distance_2_object
Rectangular_p_center_default_traits_2, 2234
RectangularPCenterTraits_2, 2236
Simple_cartesian, 55
Simple_homogeneous, 56
simplest_rational_in_interval, 2584
simplex
Convex_hull_d, 689
Delaunay_d, 702
simplices_begin
Convex_hull_d, 691
Delaunay_d, 704
simplices_end
Convex_hull_d, 691
Delaunay_d, 704
simulator_handle
Kinetic::SimulationTraits, 2502
SINGULAR, 1605, 1633, 1847
site
ApolloniusGraphVertexBase_2, 1724
SegmentDelaunayGraphStorageSite_2, 1675
SegmentDelaunayGraphVertexBase_2, 1681
Vertex, 1761
Site_2, 1661, 1713, 1751
site_inserter_object
AdaptationPolicy_2, 1771
sites_begin
Apollonius_graph_2, 1715
Voronoi_diagram_2, 1755
sites_end
Apollonius_graph_2, 1716
Voronoi_diagram_2, 1755
Sixtuple, 2698
Size, 1201, 1292, 1296
size
Arr_consolidated_curve_data_traits_2<Traits, Data>::Data_container, 1290
Arr_polyline_traits_2<SegmentTraits>::Curve_2, 1269
Compact_container, 2612
General_polygon_2, 932
In_place_list, 2604
Kd_tree, 2070
Kd_tree_node, 2073
Multiset, 2616
Polygon_2, 730
SpatialTree, 2104
Union_find, 2780

- VertexContainer_2, 1086
- size_of_border_edges
 - HalfedgeDS, 853
 - Polyhedron_3, 808
- size_of_border_halfedges
 - HalfedgeDS, 852
 - Polyhedron_3, 808
- size_of_faces
 - ArrangementDcel, 1236
 - HalfedgeDS, 850
- size_of_facets
 - Polyhedron_3, 802
- size_of_halfedges
 - ArrangementDcel, 1236
 - HalfedgeDS, 850
 - Polyhedron_3, 802
- size_of_holes
 - ArrangementDcel, 1236
- size_of_isolated_vertices
 - ArrangementDcel, 1237
- size_of_vertices
 - ArrangementDcel, 1236
 - HalfedgeDS, 850
 - Polyhedron_3, 802
- size_type, 1429, 1505, 1661, 1714, 1751, 1847, 2776
- Sliding_fair, 2099–2100
- Sliding_midpoint, 2101
- SMALLER, 124
- smallest enclosing
 - annulus, 2244
 - circle, 2193
 - ellipse, 2203
 - parallelogram, 2217
 - rectangle, 2215
 - sphere, 2238
 - sphere of spheres, 2251
 - strip, 2219
- snap_2_object
 - SnapRoundingTraits_2, 1345
- snap_rounding_2, 1342–1343
- Snap_rounding_traits_2, 1347
- SnapRoundingTraits_2, 1344–1346
- snext
 - SHalfedge, 1004, 1046
- sort
 - In_place_list, 2606
- sorted matrix search, 2328
- sorted_matrix_search, 2328–2330
- Sorted_matrix_search_traits_adaptor, 2331–2332
- SortedMatrixSearchTraits, 2333–2334
- source
 - Arr_circle_segment_traits_2<Kernel>::X_monotone_curve_2, 1275
 - Arr_conic_traits_2<RatKernel, AlgKernel, NtTraits>::Curve_2, 1281
 - Arr_rational_arc_traits_2<AlgKernel, NtTraits>::Curve_2, 1285
 - Circular_arc_2, 554
 - ExtendedKernelTraits_2, 974
 - Halfedge, 1041, 1217, 1759
 - Line_arc_2, 556
 - Points_on_segment_2, 2746
 - Ray_2, 79
 - Ray_3, 105
 - Ray_d, 460
 - Segment_2, 81
 - Segment_3, 107
 - Segment_d, 462
 - SegmentDelaunayGraphSite_2, 1670
 - SHalfedge, 1004, 1046
 - Sphere_segment, 998
 - Topological_explorer, 966
- source_of_crossing_site
 - SegmentDelaunayGraphSite_2, 1671
 - SegmentDelaunayGraphStorageSite_2, 1675
- source_of_supporting_site
 - SegmentDelaunayGraphSite_2, 1671
 - SegmentDelaunayGraphStorageSite_2, 1675
- source_site
 - SegmentDelaunayGraphSite_2, 1670
 - SegmentDelaunayGraphStorageSite_2, 1675
- SparseLinearAlgebraTraits_d, 1986
- SpatialSeparator, 2102
- SpatialTree, 2103–2104
- sphere
 - smallest enclosing, 2238
 - smallest enclosing sphere of spheres, 2251
 - see also smallest enclosing annulus
 - see also smallest enclosing circle
- Sphere_3, 109–111
- Sphere_circle, 1000–1001
- sphere_circle
 - Sphere_segment, 998
- Sphere_d, 469–471, 2056, 2091, 2093, 2095
- Sphere_point, 997
- Sphere_segment, 998–999
- splice
 - In_place_list, 2605, 2606
- split
 - Kd_tree_rectangle, 2075
 - Multiset, 2619
- split_2_object
 - ArrangementXMonotoneTraits_2, 1262
- split_edge
 - Arrangement_2, 1208

- Arrangement_with_history_2*, 1321
- Polyhedron_3*, 805
- split_edge_ex*
 - Arr_accessor*, 1214
- split_face*
 - HalfedgeDS_decorator*, 868
- split_facet*
 - Polyhedron_3*, 804
- split_loop*
 - HalfedgeDS_decorator*, 870
 - Polyhedron_3*, 806
- split_vertex*
 - HalfedgeDS_decorator*, 868
 - Polyhedron_3*, 804
 - SegmentDelaunayGraphDataStructure_2*, 1679
 - Triangulation_data_structure_2*, 1480
- Splitter*, 2105
- sprev*
 - SHalfedge*, 1004, 1046
- Sqrt*, 2590
- sqrt*, 2545, 2574, 2585–2586
- Sqrt3*, 1884
- Sqrt3_mask_3*, 1894
- Sqrt3_subdivision*, 1884
- Sqrt3Mask_3*, 1889
- Sqrt_field_tag*, 2587
- SqrtFieldNumberType*, 2586
- Square*, 2591
- square*, 2580, 2588
- Square_border_arc_length_parameterizer_3*, 1987–1988
- Square_border_parameterizer_3*, 1989–1990
- Square_border_uniform_parameterizer_3*, 1991–1992
- squared_area*
 - Triangle_3*, 114
- squared_distance*, 201, 498
 - Polytope_distance_d*, 2316
- squared_distance_denominator*
 - Polytope_distance_d*, 2317
- squared_distance_numerator*
 - Polytope_distance_d*, 2317
- squared_inner_radius*
 - Min_annulus_d*, 2246
- squared_inner_radius_numerator*
 - Min_annulus_d*, 2247
- squared_length*
 - Segment_2*, 81
 - Segment_3*, 107
 - Segment_d*, 463
 - Vector_d*, 453
- squared_outer_radius*
 - Min_annulus_d*, 2247
- squared_outer_radius_numerator*
 - Min_annulus_d*, 2247
- squared_radii_denominator*
 - Min_annulus_d*, 2247
- squared_radius*, 202
 - Circle_2*, 65
 - Min_sphere_d*, 2239
 - Sphere_3*, 110
 - Sphere_d*, 470
- STANDARD*, 973
- standard_line*
 - ExtendedKernelTraits_2*, 974
- standard_point*
 - ExtendedKernelTraits_2*, 974
- standard_ray*
 - ExtendedKernelTraits_2*, 974
- star_hole*
 - Triangulation_2*, 1437
 - TriangulationDataStructure_2*, 1468
- start*
 - Real_timer*, 2778
 - Timer*, 2779
- stateChanged*
 - Qt_widget_layer*, 2851
- static_object*
 - Kinetic::InstantaneousKernel*, 2492
- statistics*
 - Incremental_neighbor_search*, 2066
 - K_neighbor_search*, 2068
 - Kd_tree*, 2071
 - Orthogonal_incremental_neighbor_search*, 2085
 - Orthogonal_k_neighbor_search*, 2087
- step_by_step_conforming_Delaunay*
 - Triangulation_conformer_2*, 1818
- step_by_step_conforming_Gabriel*
 - Triangulation_conformer_2*, 1818
- step_by_step_refine_mesh*
 - Delaunay_mesher_2*, 1807
- stop*
 - Real_timer*, 2778
 - Timer*, 2779
- storage_site*
 - SegmentDelaunayGraphVertexBase_2*, 1681
- Straight_skeleton_2*, 1087
- Straight_skeleton_builder_2*, 1091–1093
- Straight_skeleton_builder_traits_2*, 1090
- Straight_skeleton_halfedge_base_2*, 1089
- Straight_skeleton_vertex_base_2*, 1088
- StraightSkeleton_2*, 1075
- StraightSkeletonBuilderTraits_2*, 1081–1083
- StraightSkeletonHalfedge_2*, 1079–1080
- StraightSkeletonVertex_2*, 1076–1078
- Stream_lines_2*, 2407–2408

StreamLinesTraits_2, [2406](#)
strictly_ordered_along_line
 ExtendedKernelTraits_2, [976](#)
strictly_ordered_ccw
 ExtendedKernelTraits_2, [976](#)
 strip
 smallest enclosing, [2219](#)
 strongly convex, [609](#), [649](#), [655](#), [681](#)
 polygon, [648–649](#)
 polyhedron, [678](#)
subconstraints_begin
 ConstrainedTriangulation_plus_2, [1397](#)
subconstraints_end
 ConstrainedTriangulation_plus_2, [1397](#)
Subdivision_method_3, [1883–1885](#)
 Sublayer, [2139](#)
 successor, [2622](#)
 sup
 Interval, [2033](#)
 Interval_nt, [2545](#)
sup_closed
 Interval_skip_list_interval, [2034](#)
 support set
 Min_annulus_d, [2244](#), [2245](#)
 Min_circle_2, [2193](#), [2194](#)
 Min_ellipse_2, [2203–2205](#)
 Min_sphere_d, [2239](#)
 Polytope_distance_d, [2314](#), [2316](#)
support_begin
 Min_sphere_of_spheres_d, [2253](#)
support_end
 Min_sphere_of_spheres_d, [2253](#)
support_point
 Min_circle_2, [2195](#)
 Min_ellipse_2, [2205](#)
support_points_begin
 Min_annulus_d, [2246](#)
 Min_circle_2, [2195](#)
 Min_ellipse_2, [2205](#)
 Min_sphere_d, [2239](#)
support_points_end
 Min_annulus_d, [2246](#)
 Min_circle_2, [2195](#)
 Min_ellipse_2, [2205](#)
 Min_sphere_d, [2239](#)
support_points_p_begin
 Polytope_distance_d, [2316](#)
support_points_p_end
 Polytope_distance_d, [2316](#)
support_points_q_begin
 Polytope_distance_d, [2316](#)
support_points_q_end
 Polytope_distance_d, [2316](#)
supporting_circle
 Arr_circle_segment_traits_2<Kernel>
 ::Curve_2, [1274](#)
 Arr_circle_segment_traits_2<Kernel>::X_monotone_curve_2, [1276](#)
 Circular_arc_2, [554](#)
supporting_line
 Arr_circle_segment_traits_2<Kernel>
 ::Curve_2, [1274](#)
 Arr_circle_segment_traits_2<Kernel>::X_monotone_curve_2, [1276](#)
 Line_arc_2, [556](#)
 Ray_2, [79](#)
 Ray_3, [105](#)
 Ray_d, [461](#)
 Segment_2, [82](#)
 Segment_3, [108](#)
 Segment_d, [463](#)
supporting_plane
 Triangle_3, [114](#)
supporting_site
 SegmentDelaunayGraphSite_2, [1670](#)
 SegmentDelaunayGraphStorageSite_2, [1674](#), [1675](#)
Supports_face_plane, [824](#), [826](#)
Supports_removal, [872](#), [886](#), [888](#)
Supports_vertex_point, [824](#), [826](#)
 Surface_3, [1842](#)
Surface_mesh_cell_base_3, [1843](#)
Surface_mesh_complex_2_in_triangulation_3, [1846](#)
Surface_mesh_default_criteria_3, [1853](#)
Surface_mesh_traits_generator_3, [1856](#)
Surface_mesh_vertex_base_3, [1863](#)
surface_neighbor_coordinates_3, [2386](#), [2388](#)
 Voronoi_intersection_2_traits_3, [2386–2389](#)
surface_neighbor_coordinates_certified_3, [2387](#), [2388](#)
surface_neighbors_3, [2390–2392](#)
surface_neighbors_certified_3, [2390](#), [2391](#)
 SurfaceMeshCellBase_3, [1844–1845](#)
 SurfaceMeshComplex_2InTriangulation_3, [1847–1851](#)
 SurfaceMeshCriteria_3, [1852](#)
 SurfaceMeshTraits_3, [1854–1855](#)
 SurfaceMeshTriangulation_3, [1857–1862](#)
 SurfaceMeshVertexBase_3, [1864–1865](#)
 SVertex, [1002](#)
 SW
 ExtendedKernelTraits_2, [974](#)
 Swap, [2651](#)
 swap
 AdaptationPolicy_2, [1771](#)
 Apollonius_graph_2, [1719](#)
 Apollonius_graph_hierarchy_2, [1735](#)
 Compact_container, [2611](#)

Constrained_triangulation_plus_2, 1395
DelaunayGraph_2, 1767
In_place_list, 2604
Multiset, 2616, 2618
Segment_Delaunay_graph_2, 1668
Triangulation_2, 1431
Triangulation_3, 1506
TriangulationDataStructure_2, 1464
TriangulationDataStructure_3, 1574
Voronoi_diagram_2, 1757
swap_1, 2637
swap_2, 2638
swap_3, 2639
swap_4, 2640
swap_columns
 Matrix, 443
swap_rows
 Matrix, 443
SWCORNER, 973
symmetric_difference, 949–950
 General_polygon_set_2, 920, 921
 Nef_polyhedron_2, 961
 Nef_polyhedron_3, 1037
 Nef_polyhedron_S2, 993
t
 Arr_conic_traits_2<RatKernel, AlgKernel, NtTraits>::Curve_2, 1281
Tag_false, 426
Tag_true, 426
target
 Arr_circle_segment_traits_2<Kernel>::Curve_2, 1274
 Arr_circle_segment_traits_2<Kernel>::X_monotone_curve_2, 1276
 Arr_conic_traits_2<RatKernel, AlgKernel, NtTraits>::Curve_2, 1281
 Arr_rational_arc_traits_2<AlgKernel, NtTraits>::Curve_2, 1285
 Circular_arc_2, 554
 ExtendedKernelTraits_2, 974
 Halfedge, 1041, 1217, 1759
 Line_arc_2, 556
 Points_on_segment_2, 2746
 Segment_2, 81
 Segment_3, 107
 Segment_d, 462
 SegmentDelaunayGraphSite_2, 1670
 SHalfedge, 1004, 1046
 Sphere_segment, 998
 Topological_explorer, 966
target_of_crossing_site
 SegmentDelaunayGraphSite_2, 1671
 SegmentDelaunayGraphStorageSite_2, 1676
target_of_supporting_site
 SegmentDelaunayGraphSite_2, 1671
 SegmentDelaunayGraphStorageSite_2, 1675
target_site
 SegmentDelaunayGraphSite_2, 1671
 SegmentDelaunayGraphStorageSite_2, 1675
Taucs_matrix, 1993–1994
Taucs_solver_traits, 1995–1996
Taucs_symmetric_matrix, 1997
Taucs_symmetric_solver_traits, 1998–1999
Taucs_vector, 2000
tds
 Apollonius_graph_2, 1714
 DelaunayGraph_2, 1765
 Segment_Delaunay_graph_2, 1663
 Triangulation_2, 1431
 Triangulation_3, 1507
tds.file_input, 1469
tds.file_output, 1469
test_facet
 Polyhedron_incremental_builder_3, 820
Tetrahedron, 1504
tetrahedron
 Triangulation_3, 1507
Tetrahedron_3, 112–113
third, 2699, 2701
third_type, 2699, 2701
Threetuple, 2696
Time, 2510
time
 Kinetic::InstantaneousKernel, 2492
 Real_timer, 2778
 StraightSkeletonVertex_2, 1076
 Timer, 2779
Timer, 2779
to_2d
 Plane_3, 101
to_3d
 Plane_3, 101
To_double, 2592
to_double, 2510, 2535, 2545, 2571, 2573, 2577
To_interval, 2593
to_interval, 2510, 2571, 2577
to_rational, 2594
to_vector
 Line_2, 73
 Line_3, 98
 Ray_2, 79
 Ray_3, 105
 Segment_2, 81
 Segment_3, 107
toolbar
 Qt_widget_standard_toolbar, 2864
top
 Kinetic::RootStack, 2500

- top_vertex*
 - Polygon_2*, 729
- top_vertex_2*, 734
 - requirements, 734
- TOPFRAME*, 973
- Topological_explorer*, 965–969
- Traits*, 2298
- traits*
 - Approximate_min_ellipsoid_d*, 2268
 - General_polygon_set_2*, 918
 - Kinetic::Sort*, 2457
 - Largest_empty_iso_rectangle_2*, 2299
 - Min_annulus_d*, 2249
 - Min_circle_2*, 2196
 - Min_ellipse_2*, 2207
 - Min_sphere_d*, 2241
 - Min_sphere_of_spheres_d*, 2254
 - Polyhedron_3*, 803
 - Polytope_distance_d*, 2320
- transform*, 730
 - Aff_transformation_2*, 62
 - Aff_transformation_3*, 90
 - Direction_2*, 68
 - Direction_3*, 92
 - Direction_d*, 456
 - Hyperplane_d*, 467
 - Iso_cuboid_3*, 96
 - Iso_rectangle_2*, 70
 - Line_2*, 74
 - Line_3*, 98
 - Line_d*, 459
 - Nef_polyhedron_3*, 1037
 - Plane_3*, 101
 - Point_2*, 77
 - Point_3*, 104
 - Point_d*, 449
 - Ray_2*, 80
 - Ray_3*, 106
 - Ray_d*, 461
 - Segment_2*, 82
 - Segment_3*, 108
 - Segment_d*, 463
 - Tetrahedron_3*, 113
 - Triangle_2*, 84
 - Triangle_3*, 114
 - Vector_2*, 86
 - Vector_3*, 117
 - Vector_d*, 453
- transformed_distance*
 - Euclidean_distance*, 2054, 2055
 - Euclidean_distance_sphere_point*, 2056, 2057
 - GeneralDistance*, 2064
 - Manhattan_distance_iso_box_point*, 2076
 - OrthogonalDistance*, 2082, 2083
 - Weighted_Minkowski_distance*, 2106, 2107
- Translation*, 134
- transpose*
 - LinearAlgebraTraits_d*, 437
- Tree*, 2065, 2067, 2084, 2086
- Tree_anchor*, 2144
- tree_interval_traits*, 2140–2141
- tree_point_traits*, 2142–2143
- tree_points*
 - Kd_tree_node*, 2073
- Triangle*, 1429, 1504
- triangle*
 - largest inscribed, 2287, 2289
- triangle*
 - Triangulation_2*, 1440
 - Triangulation_3*, 1507
- Triangle_2*, 83–84, 1445, 2384
- Triangle_3*, 114–115
- Triangular_field_2*, 2409
- Triangulation*, 1394
- triangulation*
 - Kinetic::Delaunay_triangulation_2*, 2427
 - Kinetic::Delaunay_triangulation_3*, 2428
 - SurfaceMeshComplex_2InTriangulation_3*, 1849
- Triangulation_2*, 1428–1441
- Triangulation_2<Traits, Tds>*
 - Locate_type*, 1406
- Triangulation_3*, 1504–1519
 - Locate_type*, 1558
- Triangulation_cell_base_3*, 1552
- Triangulation_cell_base_with_info_3*, 1553
- Triangulation_conformer_2*, 1817–1819
- Triangulation_cw_ccw_2*, 1442–1443
- Triangulation_data_structure*, 1429, 1471, 1476, 1504, 1585, 1587, 1589, 1592, 1661, 1713
- Triangulation_data_structure_2*, 1480–1481
- Triangulation_data_structure_3*, 1594
- Triangulation_ds_cell_base_3*, 1595
- Triangulation_ds_face_base_2*, 1482
- Triangulation_ds_vertex_base_2*, 1483
- Triangulation_ds_vertex_base_3*, 1596
- Triangulation_euclidean_traits_2*, 1444
- Triangulation_euclidean_traits_xy_3*, 1445–1447
- Triangulation_face_base_2*, 1448
- Triangulation_face_base_with_info_2*, 1449
- Triangulation_hierarchy_2*, 1450
- Triangulation_hierarchy_3*, 1527
- Triangulation_hierarchy_vertex_base_2*, 1451
- Triangulation_hierarchy_vertex_base_3*, 1556
- Triangulation_utils_3*, 1597
- Triangulation_vertex_base_2*, 1452
- Triangulation_vertex_base_3*, 1554

- Triangulation_vertex_base_with_info_2*, 1453
- Triangulation_vertex_base_with_info_3*, 1555
- TriangulationCellBase_3*, 1546
- TriangulationDataStructure_2*, 1463–1470
- TriangulationDataStructure_2::Face*, 1474–1475
- TriangulationDataStructure_2::Vertex*, 1478–1479
- TriangulationDataStructure_3*, 1573–1584
- TriangulationDSCellBase_3*, 1589–1591
- TriangulationDSFaceBase_2*, 1471–1473
- TriangulationDSVertexBase_2*, 1476–1477
- TriangulationDSVertexBase_3*, 1592–1593
- TriangulationFaceBase_2*, 1422
- TriangulationHierarchyVertexBase_2*, 1423
- TriangulationHierarchyVertexBase_3*, 1549
- TriangulationTraits_2*, 1424–1425
- TriangulationTraits_3*, 1534–1535
- TriangulationVertexBase_2*, 1426–1427
- TriangulationVertexBase_3*, 1547–1548
- Triple*, 2699–2700
- twin*
 - Halfedge*, 1041, 1217, 1758
 - Halffacet*, 1043
 - SHalfedge*, 1004, 1046
 - SHalfloop*, 1005, 1047
 - SVertex*, 1002
 - Topological_explorer*, 966
- Two_vertices_parameterizer_3*, 2001–2002
- Twotuple*, 2695
- type*
 - ExtendedKernelTraits_2*, 974
 - Object*, 120
- u*
 - Arr_conic_traits_2<RatKernel, AlgKernel, NtTraits>::Curve_2*, 1281
- unbounded_face*
 - Arrangement_2*, 1204
 - Voronoi_diagram_2*, 1753
- unbounded_faces_begin*
 - Voronoi_diagram_2*, 1754
- unbounded_faces_end*
 - Voronoi_diagram_2*, 1754
- unbounded_halfedge*
 - Voronoi_diagram_2*, 1754
- unbounded_halfedges_begin*
 - Voronoi_diagram_2*, 1754
- unbounded_halfedges_end*
 - Voronoi_diagram_2*, 1754
- unify_sets*
 - Union_find*, 2781
- Union_find*, 2780–2781
- unique*
 - In_place_list*, 2606
- Unique_hash_map*, 2782–2783
- UniqueHashFunction*, 2784
- unit_value*, 2535
- unlock*
 - Qt_widget*, 2843
- unperturb_incircle*, 2535
- unperturb_insphere*, 2535
- unsafe_comparison*, 2544
- up*
 - ApolloniusGraphHierarchyVertexBase_2*, 1736
 - Halfedge*, 1759
 - SegmentDelaunayGraphHierarchyVertexBase_2*, 1696
 - TriangulationHierarchyVertexBase_2*, 1423
 - TriangulationHierarchyVertexBase_3*, 1549
- update*
 - Stream_lines_2*, 2408
- upper*
 - Kd_tree_node*, 2073
- upper hull, 2D*, 652–653
- upper_bound*, 2535
 - Multiset*, 2618
- upper_hull_points_2*, 611, 652–653
- v*
 - Arr_conic_traits_2<RatKernel, AlgKernel, NtTraits>::Curve_2*, 1281
- validity check*
 - Approximate_min_ellipsoid_d*, 2270
 - Min_annulus_d*, 2249
 - Min_circle_2*, 2196
 - Min_ellipse_2*, 2206, 2270
 - Min_sphere_d*, 2241
 - Min_sphere_of_spheres_d*, 2254
 - Polytope_distance_d*, 2319
- Value*, 2031, 2035, 2333
- value*
 - Filtered_exact*, 2532
- value_comp*
 - Multiset*, 2616
- value_type*, 2544, 2695–2698, 2746, 2762–2770
- Vector*, 440–441, 2003
- vector*
 - Direction_2*, 68
 - Direction_3*, 92
 - Direction_d*, 456
 - Segment_d*, 463
- Vector_2*, 85–87, 2384, 2403, 2407, 2409
- Vector_3*, 116–118, 2306, 2310
- Vector_d*, 451–454, 2373, 2382
- VectorField_2*, 2410
- Verbose_ostream*, 2805
- verify_determinant*
 - LinearAlgebraTraits_d*, 438

VERTEX, [1406](#), [1430](#), [1505](#), [1558](#)
 Vertex, [1587](#)–[1588](#)
 Vertex, [816](#)–[817](#), [822](#), [1039](#), [1081](#), [1084](#), [1216](#),
 [1429](#), [1504](#), [1661](#), [1713](#), [1760](#)–[1761](#)
 vertex
 ArrangementDcelHalfedge, [1242](#)
 Cell, [1585](#)
 ConvexHullPolyhedronHalfedge_3, [670](#)
 Halfedge, [815](#)
 HalfedgeDSHalfedge, [858](#)
 Iso_cuboid_3, [95](#)
 Iso_rectangle_2, [70](#)
 Polygon_2, [730](#)
 Polyhedron_incremental_builder_3, [820](#)
 Segment_2, [81](#)
 Segment_3, [107](#)
 Segment_d, [462](#)
 Tetrahedron_3, [112](#)
 Triangle_2, [83](#)
 Triangle_3, [114](#)
 TriangulationDataStructure_2::Face, [1474](#)
 TriangulationDSCellBase_3, [1590](#)
 TriangulationDSFaceBase_2, [1472](#)
 vertex_begin
 Halfedge, [814](#)
 Vertex, [817](#)
 Vertex_circulator, [1662](#), [1713](#)
 vertex_conflict_2_object
 ApolloniusGraphTraits_2, [1730](#)
 SegmentDelaunayGraphTraits_2, [1686](#)
 Vertex_const_handle, [1296](#)
 vertex_degree
 Halfedge, [815](#)
 Vertex, [817](#)
 Vertex_handle, [818](#), [1210](#), [1292](#), [1312](#), [1430](#), [1471](#),
 [1476](#), [1504](#), [1585](#), [1587](#), [1589](#), [1592](#),
 [1662](#), [1713](#), [1847](#)
 Vertex_list, [1407](#)
 vertex_list
 RegularTriangulationFaceBase_2, [1407](#)
 vertex_node
 CatmullClark_mask_3, [1890](#)
 Loop_mask_3, [1892](#)
 PQQMask_3, [1886](#)
 PTQMask_3, [1887](#)
 Sqrt3_mask_3, [1894](#)
 Sqrt3Mask_3, [1889](#)
 vertex_of_facet
 Convex_hull_d, [689](#)
 vertex_of_simplex
 Convex_hull_d, [688](#)
 Delaunay_d, [702](#)
 vertex_triple_index
 SurfaceMeshTriangulation_3, [1860](#)
 VertexContainer_2, [1086](#)
 vertices_around_vertex_begin
 Parameterization_mesh_patch_3, [1971](#)
 Parameterization_polyhedron_adaptor_3,
 [1976](#)
 ParameterizationMesh_3, [1961](#)
 vertices_begin
 Arrangement_2, [1203](#)
 ArrangementDcel, [1237](#)
 Convex_hull_d, [691](#)
 HalfedgeDS, [850](#)
 Nef_polyhedron_3, [1036](#)
 Polygon_2, [728](#)
 Polyhedron_3, [802](#)
 SurfaceMeshComplex_2InTriangulation_3,
 [1850](#)
 Topological_explorer, [967](#)
 TriangulationDataStructure_2, [1465](#)
 TriangulationDataStructure_3, [1582](#)
 Voronoi_diagram_2, [1755](#)
 vertices_circulator
 Polygon_2, [728](#)
 vertices_clear
 HalfedgeDS, [852](#)
 vertices_end
 Arrangement_2, [1203](#)
 ArrangementDcel, [1237](#)
 Convex_hull_d, [691](#)
 HalfedgeDS, [850](#)
 Nef_polyhedron_3, [1036](#)
 Polygon_2, [728](#)
 Polyhedron_3, [802](#)
 SurfaceMeshComplex_2InTriangulation_3,
 [1850](#)
 Topological_explorer, [967](#)
 TriangulationDataStructure_2, [1465](#)
 TriangulationDataStructure_3, [1582](#)
 Voronoi_diagram_2, [1755](#)
 vertices_erase
 HalfedgeDS, [851](#)
 HalfedgeDS_decorator, [866](#)
 vertices_in_conflict
 Delaunay_triangulation_3, [1524](#)
 vertices_in_constraint_begin
 Constrained_triangulation_plus_2, [1397](#)
 vertices_in_constraint_end
 Constrained_triangulation_plus_2, [1398](#)
 vertices_pop_back
 HalfedgeDS, [851](#)
 HalfedgeDS_decorator, [866](#)
 vertices_pop_front
 HalfedgeDS, [851](#)
 HalfedgeDS_decorator, [866](#)
 vertices_push_back

- HalfedgeDS, 850
- HalfedgeDS_decorator, 865
- vertices_splice
 - HalfedgeDS_list, 886
- VI, 2859
- virtual_size
 - Memory_sizer, 2776
- visible_sites_begin
 - Apollonius_graph_2, 1716
- visible_sites_end
 - Apollonius_graph_2, 1716
- visit_all_facets
 - Convex_hull_d, 692
- visit_shell_objects
 - Nef_polyhedron_3, 1037
- visitor
 - Kinetic::Delaunay_triangulation_2, 2427
 - Kinetic::Delaunay_triangulation_3, 2429
 - Kinetic::Sort, 2457
- Volume, 1044
- volume, 203
 - Iso_box_d, 473
 - Iso_cuboid_3, 96
 - SFace, 1049
 - Tetrahedron_3, 113
- volumes_begin
 - Nef_polyhedron_3, 1036
- volumes_end
 - Nef_polyhedron_3, 1036
- Voronoi diagram, 1360, 1404, 1525
- Voronoi_diagram_2, 1751–1757
- Voronoi_intersection_2_traits_3, 2384
- w
 - Arr_conic_traits_2<RatKernel, AlgKernel, NtTraits>::Curve_2, 1281
- weak_equality, 459, 463, 468, 471
- Weight, 1454, 2384
- weight, 1559
 - ApolloniusSite_2, 1720
 - Weighted_point, 1454
- Weighted Alpha Shapes 2, 1602
- Weighted_alpha_shape_euclidean_traits_2, 1615
- Weighted_alpha_shape_euclidean_traits_3, 1644
- Weighted_alpha_shapes_2, 1605
- Weighted_alpha_shapes_3, 1633
- weighted_circumcenter
 - Regular_triangulation_2, 1416
- Weighted_Minkowski_distance, 2106–2107
- Weighted_point, 1412, 1454, 1528
- Weighted_point_3, 1542
- WeightedAlphaShapeTraits_3, 1643
- WeightedPoint, 1559
- wheelEvent
 - Qt_widget_layer, 2851
- widget, 2852
- width
 - of 3D point set, 2306
- Width_3, 2306–2309
 - creation, 2306
 - example, 2308
 - implementation, 2308
 - member functions, 2307
 - access, 2307
 - requirements, 2306
 - traits class
 - requirements, 2313
 - see also Width_default_traits_3
 - types, 2306
- Width_default_traits_3, 2310–2311
- WidthTraits_3, 2312–2313
- window output
 - Min_circle_2, 2197
 - Min_ellipse_2, 2207
- window_query
 - Range_tree_d, 2126
 - Range_tree_k, 2129
 - Segment_tree_d, 2133
 - Segment_tree_k, 2135
 - Tree_anchor, 2144
- write, 1235
- write_arrangement_begin
 - ArrangementOutputFormatter, 1297
- write_arrangement_end
 - ArrangementOutputFormatter, 1297
- write_ccb_halfedges_begin
 - ArrangementOutputFormatter, 1299
- write_ccb_halfedges_end
 - ArrangementOutputFormatter, 1299
- write_curve
 - ArrWithHistoryOutputFormatter, 1325
- write_curve_begin
 - ArrWithHistoryOutputFormatter, 1325
- write_curve_end
 - ArrWithHistoryOutputFormatter, 1325
- write_curves_begin
 - ArrWithHistoryOutputFormatter, 1325
- write_curves_end
 - ArrWithHistoryOutputFormatter, 1325
- write_edge_begin
 - ArrangementOutputFormatter, 1298
- write_edge_end
 - ArrangementOutputFormatter, 1298
- write_edges_begin
 - ArrangementOutputFormatter, 1297
- write_edges_end
 - ArrangementOutputFormatter, 1297
- write_eps

- Approximate_min_ellipsoid_d*, 2270
- write_face_begin*
 - ArrangementOutputFormatter*, 1298
- write_face_data*
 - ArrangementOutputFormatter*, 1299
- write_face_end*
 - ArrangementOutputFormatter*, 1298
- write_faces_begin*
 - ArrangementOutputFormatter*, 1297
- write_faces_end*
 - ArrangementOutputFormatter*, 1297
- write_halfedge_index*
 - ArrangementOutputFormatter*, 1298
- write_halfedge_data*
 - ArrangementOutputFormatter*, 1298
- write_holes_begin*
 - ArrangementOutputFormatter*, 1298
- write_holes_end*
 - ArrangementOutputFormatter*, 1299
- write_induced_edges_begin*
 - ArrWithHistoryOutputFormatter*, 1325
- write_induced_edges_end*
 - ArrWithHistoryOutputFormatter*, 1325
- write_isolated_vertices_begin*
 - ArrangementOutputFormatter*, 1299
- write_isolated_vertices_end*
 - ArrangementOutputFormatter*, 1299
- write_outer_ccb_begin*
 - ArrangementOutputFormatter*, 1298
- write_outer_ccb_end*
 - ArrangementOutputFormatter*, 1298
- write_point*
 - ArrangementOutputFormatter*, 1298
- write_size*
 - ArrangementOutputFormatter*, 1297
- write_vertex_begin*
 - ArrangementOutputFormatter*, 1297
- write_vertex_data*
 - ArrangementOutputFormatter*, 1298
- write_vertex_end*
 - ArrangementOutputFormatter*, 1297
- write_vertex_index*
 - ArrangementOutputFormatter*, 1298
- write_vertices_begin*
 - ArrangementOutputFormatter*, 1297
- write_vertices_end*
 - ArrangementOutputFormatter*, 1297
- write_x_monotone_curve*
 - ArrangementOutputFormatter*, 1298
- x*
 - Arr_circle_segment_traits_2<Kernel>::Point_2*, 1273
 - Circular_arc_point_2*, 558
- Point_2*, 76
- Point_3*, 103
- Sphere_point*, 997
- Vector_2*, 86
- Vector_3*, 117
- x_at_y*
 - Line_2*, 73
- x_equal*, 204
- x_max*
 - Qt_widget*, 2845
- x_min*
 - Qt_widget*, 2845
- X_monotone_curve_2*, 1201, 1210, 1292, 1296, 1312, 1318
- X_monotone_curve_data*, 1287
- x_pixel*
 - Qt_widget*, 2845
- x_real*, 2845
 - Qt_widget*, 2845
- xmax*
 - Bbox_2*, 64
 - Bbox_3*, 88
 - Iso_cuboid_3*, 95
 - Iso_rectangle_2*, 70
- xmin*
 - Bbox_2*, 64
 - Bbox_3*, 88
 - Iso_cuboid_3*, 95
 - Iso_rectangle_2*, 70
- y*
 - Arr_circle_segment_traits_2<Kernel>::Point_2*, 1273
 - Circular_arc_point_2*, 558
 - Point_2*, 76
 - Point_3*, 103
 - Sphere_point*, 997
 - Vector_2*, 86
 - Vector_3*, 117
- y-monotone polygon*, 735, 749
 - function object, 755
- y_at_x*
 - Line_2*, 73
- y_equal*, 205
- y_max*
 - Qt_widget*, 2845
- y_min*
 - Qt_widget*, 2845
- y_monotone_partition_2*, 735, 772–774
 - postconditions, 737, 765
 - traits class, 777–779
 - default, 769
- y_monotone_partition_is_valid*
 - traits class, 777

- y_monotone_partition_is_valid_2*, [775–776](#)
 - traits class
 - default, [775](#)
- y_pixel*
 - Qt_widget*, [2845](#)
- y_real*, [2845](#)
 - Qt_widget*, [2845](#)
- y_max*
 - Bbox_2*, [64](#)
 - Bbox_3*, [88](#)
 - Iso_cuboid_3*, [95](#)
 - Iso_rectangle_2*, [70](#)
- y_min*
 - Bbox_2*, [64](#)
 - Bbox_3*, [88](#)
 - Iso_cuboid_3*, [95](#)
 - Iso_rectangle_2*, [70](#)
- YMonotonePartitionIsValidTraits_2*, [775](#), [777](#)
 - model, [769](#)
- YMonotonePartitionTraits_2*, [778–779](#)
 - model, [769](#)
- z*
 - Point_3*, [103](#)
 - Sphere_point*, [997](#)
 - Vector_3*, [117](#)
- z_equal*, [206](#)
- ZERO*, [124](#)
- z_max*
 - Bbox_3*, [88](#)
 - Iso_cuboid_3*, [95](#)
- z_min*
 - Bbox_3*, [88](#)
 - Iso_cuboid_3*, [95](#)
- zoom*
 - Qt_widget*, [2842](#)