Java-based Multiresolution Streaming Mesh with QoS-like Controlling for Web Graphics

by

Bing-Yu Chen

A Doctoral Dissertation

Submitted to
The Graduate School of Science
The University of Tokyo
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Science
in Information Science

December 2002

Abstract

Web Graphics has become an important and new platform today, since there are more and more people that would like to have 3D graphics supports on the Web. In order to distribute 3D graphics applications from servers to clients on different platforms, a general-purpose 3D graphics library for Java called jGL is developed as the base of the 3D graphics applications for Web Graphics. To make jGL to be easy to learn and use, its API (Application Programming Interface) is designed in a manner quite similar to that of OpenGL. Therefore, the programmers who are familiar with the usage of OpenGL can use jGL intuitively since they can find one-to-one mapping functions in both of them. Besides the functions of OpenGL, jGL also supports other useful functionalities, such as Phong shading, bump mapping, environment mapping, procedural solid texturing, and VRML (Virtual Reality Modeling Language) as its extensions.

How to transmit geometric 3D models efficiently on the Internet is an important task in Web Graphics, since there are many users using geometric 3D models on the Web. The data size of a geometric 3D model is usually large to enable more detail to be represented, although the user on the Internet usually does not need to use such a detailed model in most cases. Hence, it is necessary to represent the 3D model while keeping the data size small and preserving the shape and features, even if the meshes that constitute the model are unstructured. Therefore, we proposed a reversible multiresolution streaming mesh for these requirements. The mesh simplification algorithm used for generating the streaming mesh is different from other mesh simplification methods. In order to preserve the shape and features of the original model, we first detect the features of the 3D model before simplifying it, so our approach has no over-simplification problem, which is a common problem of other methods. Even the meshes consisted by the model is unstructured; our method can also get a good result efficiently. Moreover, since the network bandwidth of the Internet is not stable actually, how much data is suitable for representing a 3D model on the Web is also a question. Hence, a QoS-like (Quality of Service)

QoS-like (Quality of Service) controlling mechanism is also proposed with the reversible multiresolution streaming mesh.

Additionally, a VRML library and an adaptive method of procedural solid texturing for supporting the applications of Web Graphics are also provided to be the extensions of jGL. The main idea of the proposed procedural solid texturing method is to use the cache mechanism to store the calculated texture data; hence the computational cost can be reduced. This method solves two main problems of previous procedural solid texturing methods. One is the run-time performance problem due to the computational cost, and the other is the storage problem since to create a 3D-texture image as texture mapping needs a lot of storage spaces. Therefore, our approach can show a 3D model with procedural solid texturing with no such problems.

In this dissertation, a prototype of transmitting 3D geometric models on the Internet has been proposed. In order to provide this client/server system for Web Graphics, we contributed from a 3D graphics library as a development environment to a total solution to establish the system including mesh simplification, transmission, and reconstruction. Furthermore, a real-time shading method of procedural solid texturing which can be used on the Web and two model deformation methods are also proposed to support this model transmitting prototype.

Acknowledgements

First of all, I would like to deeply appreciate Prof. Tomoyuki Nishita, my advisor. It is my pleasure and so lucky to be his first PhD. student in the University of Tokyo. He taught me not only for how to do the researches, but also for how a good researcher and person should be. To be a good researcher as him is my next dream. Of course, I would also like to thank all of the members and past members in our laboratory, especially for Prof. Jieqing Feng, Yosuke Bando, and Yutaka Ono, my colleagues. We did really enjoy the co-research life so much. Prof. Yoshinori Dobashi usually comes to our laboratory and gives me a lot of suggestions; I would also like to thank him here.

Then, I wish to express my best appreciations for Prof. Ming Ouhyoung, the advisor of my Master thesis, Prof. Ja-Ling Wu, and Prof. Wen-Chin Chen. They do really give me many suggestions and commends for researches and personalities even when I left the National Taiwan University. I was so lucky to have the chances to work with them and also several ex-members of the Communications and Multimedia Laboratory, especially the Graphics Group ex-members and assistants.

The members of the judging committee of this dissertation, Prof. Katsushi Ikeuchi, Prof. Masami Hagiya, Prof. Hiroshi Imai, Prof. Takeo Igarashi, and Prof. Takashi Kanai, gave me a lot of useful suggestions to modify this dissertation. I also wish to express my deeply appreciations to them.

Furthermore, I can not forget to thank my families, my grandmother, father, mother, and brother. Without them, I think I can not get the degree so smoothly and successfully.

Lastly, this dissertation is for Yao and NoNo, my wife and baby.

Table of Contents

1. Introduction 1

1.1. Motivation 1

1.2. Problems of Web Graphics 2

1.3. Development Environment 2

1.4. jGL - a 3D Graphics Library for Java 3

1.5. jVL – a VRML Support for jGL 3

1.6. Streaming Mesh with Shape Preserving 4

1.7. Adaptive Solid Texturing for Web Graphics 5

1.8. Contributions 5

1.9. Organization 6

2. jGL - a 3D Graphics Library for Java 9

2.1. Introduction 9

2.2. Related Projects 10

2.3. Introduction to OpenGL 12

2.4. OpenGL vs. jGL 13

2.5. Graphics Context 14

2.6. Implementation Problems 17

2.7. Performance Enhancement Issues 18

2.8. Results 21

2.8.1. Supported Functions and Usage of jGL 22

2.8.2. Performance Evaluation 25

2.9. Summary 30

3. Streaming Mesh with Shape Preserving 31

3.1. Introduction 31

3.2. Related Work 33

3.3. Notation 35

3.4. Feature edge detection 36

3.4.1. Sharp edge definition 36

3.4.2. Base edge detection 37

3.5. Mesh Simplification 41

3.5.1. Simplification with sharp edge 42

3.5.2. Selected removable edge testing 43

3.5.3. Half Edge Collapse 46

3.6. Mesh Reconstruction 48

3.6.1. Vertex Split 48

3.6.2. QoS-like Controlling 50

3.7. Results 54

3.7.1. Evaluation of Mesh Simplification and Reconstruction 54

3.7.2. Comparisons with Previous Works 69

3.7.3. Evaluation of QoS-like Controlling 75

3.8. Summary 78

4. VRML and Solid Texturing Supports 81

4.1. Introduction 81

4.2. jVL – a VRML Support for jGL 81

4.2.1. System Hierarchy 82

4.2.2. Performance Enhancement Issues 83

4.2.3. Result 84

4.3. Adaptive Solid Texturing for Web Graphics 87
4.3.1. Cache Technology for Solid Texturing 88
4.3.2. Cache Cube Conception 88
4.3.3. Storage Methodology 90
4.3.4. Proper Resolution Detection....................................... 91
4.3.5. Result 92
4.4. Summary 98
5. Conclusions and Future Work 99
5.1. jGL - a 3D Graphics Library for Java....................................... 99
5.2. Streaming Mesh with Shape Preserving 100
5.3. jVL – a VRML Support for jGL 101
5.4. Adaptive Solid Texturing for Web Graphics 101
5.5. Contribution 102
Appendix A. Usage of jGL 105
A.1. Differences between jGL and OpenGL 105
A.2. Functionalities of jGL 107
A.3. How to Use jGL 111
Appendix B. jGL for Mobile Phone 113
Appendix C. Model Deformation 115
C.1. Deformation along a Parametric Surface 115
C.1.1. Subdivision on a Flattened Surface 116
C.1.2. Result 117
C.2. Deformation with Auto-generated Lattices 118
C.2.1. Octree Subdivision Lattices...................................... 118
C.2.2. Hierarchical Deformation 119
C.2.3. Result 120

Bibliography 123

List of Figures

Figure 2-1: OpenGL API hierarchy.. 13

Figure 2-2: jGL API hierarchy 14

Figure 2-3: The hierarchy of OpenGL modules 15

Figure 2-4: jGL implementation hierarchy 15

Figure 2-5: The graphics context of jGL 16

Figure 2-6: The example of using function-overriding 21

Figure 2-7: A simple OpenGL program using GLUT 23

Figure 2-8: The source code of a simple jGL program using GLUT 24

Figure 2-9: The programs used to measure jGL and Java 3D 26

Figure 2-10: 12 spheres are rendered to measure the performance............ 27

Figure 2-11: 24 teapots are rendered to measure the performance 27

Figure 3-1: The edge collapse operation 33

Figure 3-2: The vertex decimation method 34

Figure 3-3: The representation of a geometric 3D model 35

Figure 3-4: A wedge is defined as a pair of vertex and face 36

Figure 3-5: The examples of sharp edges.. 37

Figure 3-6: The SOD operator 38
Figure 3-7: The ESOD operator 38
Figure 3-8: Base edge detection by finding the virtual feature edges 39
Figure 3-9: The examples of base edges 40
Figure 3-10: The procedure of our 3D mesh simplification method 42
Figure 3-11: The example of mesh inversion 43
Figure 3-12: The example for 2-manifolds and non-2-manifold 44
Figure 3-13: The example of choosing the endpoint for removing 45
Figure 3-14: The examples of half edge collapse and vertex split 46
Figure 3-15: Patch data structure 47
Figure 3-16: Using the transmitted patch to do the vertex split 49
Figure 3-17: System hierarchy and network communication diagram 53
Figure 3-18: Comparisons of original and simplified bunny models 55
Figure 3-19: Simplified and reconstructed bunny models 57
Figure 3-20: Original and simplified dragon and hand models 59
Figure 3-21: Original and simplified cessna, fandisk, and tiger models 61
Figure 3-22: Comparisons of subdivided and reconstructed tiger models 65
Figure 3-23: Applying to models with different number of faces 67
Figure 3-24: Similarity of geometric approximation of bunny model 70
Figure 3-25: Images from different viewpoints 71
Figure 3-26: Similarity of appearance of bunny model 73

Figure 3-27: Similarity of appearance of tiger model 74

Figure 3-28: Network layout of testing environment 76

Figure 4-1: The system hierarchy of jVL 82

Figure 4-2: A simple table is rendered to measure the performance........... 85

Figure 4-3: The source code using jVL for displaying a VRML file.......... 86

Figure 4-4: System diagram of cache cube conception.............................. 89

Figure 4-5: The cache cube is stored in a hashtable................................... 90

Figure 4-6: A cube rendered with a marble texturing function.................. 94

Figure 4-7: The results of a wood face model and a clipped one................ 95

Figure 4-8: Two marble Atenea models are rendered................................ 95

Figure A-1: Implemented OpenGL functionalities in jGL....................... 107

Figure A-2: Implemented OpenGL functionalities in jGL (cont.)............ 108

Figure A-3: Implemented GLUT functionalities in jGL.......................... 109

Figure A-4: Implemented GLU functionalities in jGL 110

Figure A-5: Implemented VRML nodes in jVL....................................... 110

Figure B-1: A robot arm is rendered on a mobile phone.......................... 114

Figure C-1: A dolphin model and its deformed result.............................. 117

Figure C-2: A dolphin model deformed along a Bézier surface 118

Figure C-3: Hierarchical deformation of a chimpanzee model................ 121

List of Tables

Table 2-1: The performance testing of jGL using 12 spheres..................... 25

Table 2-2: The performance testing of jGL using 24 teapots...................... 26

Table 2-3: The performance comparisons for jGL and Java 3D................. 29

Table 2-4: The questionnaire result of jGL.. 30

Table 3-1: Comparisons of file sizes and run-time performances............... 63

Table 3-2: Comparisons of transmitting performances.............................. 77

Table 3-3: Comparisons of transmitting and reconstructing performances 77

Table 3-4: The transmitted 3D models' resolution comparisons................ 78

Table 4-1: The performance testing of jVL using a simple table................ 84

Table 4-2: The performance testing of jVL using several cubes 85

Table 4-3: Comparisons of adaptive and two previous methods 93

Table 4-4: Comparisons of using multiple cache cube 93

Chapter 1
Introduction

1.1. Motivation

Since the end of 20th century, the Internet [48] is getting more and more popular, and there are many people that would like to use the services on the Internet everyday, such as the Web, E-mail, etc. The Web is the most important service of these Internet services, because almost all kinds of media have been included into the Web, such as text, image, animation, sound, etc. Some new and powerful Web browsers, like Netscape Navigator[1] or Microsoft Internet Explorer[2], have already integrated the usages of several Internet services in them, such as the Web, E-Mail, FTP (File Transfer Protocol), etc., so that the user can use all of the services by just clicking a mouse button.

To produce more realistic and interactive contents and also to enhance the abilities of the Web, people have rediscovered 3D computer graphics recently. Because the machine performance is improving all the time, several 3D graphics applications can be realized on a cheaper computer, like a laptop PC with a low-power mobile CPU or a low-cost workstation. Therefore, there are more and more people on the Internet want to have 3D graphics supports on the Web recently. This is given rise to a new platform called Web Graphics. There are several related researches and products for Web Graphics, such as Apple QuickTime[3], VRML (Virtual Reality Modeling Language) [6], MPEG-4 SNHC (Synthetic-Natural Hybrid Coding), and

---

1 *http://channels.netscape.com/ns/browsers/default.jsp*

2 *http://www.microsoft.com/windows/ie/default.htm*

3 *http://www.apple.com/quicktime/*

DIS (Distributed Interactive Simulation). Web Graphics has changed the Web world from flat to be three-dimensional and from one way static output to two way interactive display.

1.2. Problems of Web Graphics

There are several problems of Web Graphics. The first problem is the performance, since the performance of the client machine does always not be known. Hence, we do not know what kind of the program is suitable for both hi-end and low-cost machines, but the users on the Internet always want to have quick or real-time feedback and maybe the users using the hi-end machines wish to get more complex results and effects.

The second problem is about the data and code size, because the narrow network bandwidth is always the biggest problem for network utilities. As the performance of the computer increases, people may want to use or show more complex scenes on the Web, but this implies more data or description codes are needed. Since the network bandwidth has no increasing as the computer, the users must pay more time for downloading; even they do always hate this.

The third is the platform dependent problem, due to there are several kinds of platforms linked with the Internet, and we also do not know what types of the client machines are used by the users. Therefore, what is the suitable program for all kinds of machines is also the problem.

1.3. Development Environment

Although many applications could be used for Web Graphics, there is only a small number available today. One of the main problems is that the appli-

cation provider must offer several versions of the same applications for different platforms, since the Internet itself is a heterogeneous network environment. Observing the development of the Internet, we believe that "pay-per-use" software will be realized in the near future. Under this new paradigm, we may need to distribute applications from servers to clients on different platforms. Therefore, Java [2] [30] is chosen as our programming language for its hardware-neutral features, and wide availability on many hardware platforms, even for embedded systems, such as mobile phone, PDA (Personal Data Assistant), etc.

1.4. jGL - a 3D Graphics Library for Java

To develop 3D graphics applications on a stand-alone computer, programmers always hope to use a powerful 3D graphics library, like OpenGL [70], but there is no such a library for Web Graphics. Moreover, to jump to the Web world from a stand-alone computer, people may not like to learn how to use an entirely new 3D graphics library. Therefore, we develop a 3D graphics library called jGL using pure Java programming language and define its API (Application Programming Interface) in a manner quite similar to that of OpenGL, since OpenGL is a de-facto industry standard and many programmers are familiar with its API. Besides the functionalities of OpenGL, we also implement Phong shading, bump mapping, environment mapping, procedural solid texturing, and VRML in jGL as its extensions.

1.5. jVL – a VRML Support for jGL

Besides the 3D graphics library, jGL, as a programming environment provided to 3D graphics programmers when developing applications for Web Graphics, it is also needed to display 3D models or scenes in the applications. Since VRML is a well-known standard for describing 3D models and

there are several 3D models on the Internet which are coded as VRML formats, we follow the API of jGL to develop a VRML library called jVL as an extension of jGL.

1.6. Streaming Mesh with Shape Preserving

For Web Graphics, geometric 3D models are widely used. Therefore, how to efficiently transmit the mesh data, which constitutes the 3D model, through the Internet has become an important task, since the size of the mesh data is usually large. If a user wants to use a geometric 3D model on a Web page, to retrieve the data set of the model is time-consuming. However, the user on the Internet usually does not need to use such a detailed model in most cases. Hence, to offer a simplified model which has easily recognizable shape and features of the original model is necessary. Moreover, if the user then decides that a model with more details is needed, then the subsequent download needs to be quick and with no retransmission of information. Therefore, we propose a client/server architecture called Streaming Mesh with the above requirements for Web Graphics which includes 3D model simplification, transmission, and reconstruction.

Additionally, since the network bandwidth of the Internet is not stable actually, how much data is suitable to represent a 3D model for Internet uses is also a question. Therefore, we also present a QoS-like (Quality of Service) [4] [5] control mechanism with the proposed multiresolution streaming mesh to determine how many mesh data should be sent according to current network bandwidth, so that the user with different qualities of network connections could receive different resolutions of geometric 3D models.

1.7. Adaptive Solid Texturing for Web Graphics

Rendering a 3D model with procedural solid texturing [19] [61] [62] is a useful method of showing a 3D model realistically. Since the corresponding texture data is calculated "on the fly" when rendering a 3D model with procedural solid texturing, it can offer highly visual effects of the model that is being rendered. However, to calculate the corresponding texture data on the fly is time-consuming, since the texturing function could involve a lot of computational time in order to present a complex textured appearance. Therefore, we offer a new method of quickly rendering a 3D model using procedural solid texturing for Web Graphics. This approach called Adaptive Solid Texturing does not need any support from the hardware and could still achieve an almost real-time response. As jVL, this method also forms an extension of jGL to enhance the rendering abilities of it.

1.8. Contributions

In this dissertation, we present from a 3D graphics library, jGL, as a development environment for Web Graphics to multiresolution streaming mesh, an efficient geometric 3D model transmission system, which includes 3D mesh simplification, transmission, and reconstruction. Moreover, an adaptive method of procedural solid texturing called Adaptive Solid Texturing which can be used on the Web and two model deformation methods are also proposed to support this model transmitting prototype.

jGL is the only general-purpose and software-based 3D graphics library for Java. Because it is developed with pure Java programming language, users who wish to use the Java applets or applications based on jGL do not need to install any package before executing them. When starting the Java applets, all of the necessary byte-codes, including jGL, are downloaded from the server. Moreover, the API of jGL is defined as that of OpenGL, so

that the programmers who are familiar with the usage of OpenGL can use jGL without any study. Since we offer jGL onto our Web server, many people around the world are using jGL for their works, such as providing their previous OpenGL works on the Web, or using jGL to teach and learn computer graphics algorithms.

The mesh simplification algorithm used in the proposed multiresolution streaming mesh is different from other previous methods. In order to preserve the shape and features of the original model after simplifying, we first segment the model to be several parts due to its features, even the meshes which constitute the 3D model are unstructured. Then, we simplify each part of the model recursively by removing its edges and vertices iteratively until there are no removable edges and vertices. Therefore, our approach could preserve the shape and features of the original model after the model has been simplified. Furthermore, since the operator used for detecting the features of the model is simple, the performance of our approach is also better than other previous methods. Moreover, with our QoS-like control mechanism, the user with different quality of network connections can receive different resolution of geometric 3D models due to the current network bandwidth.

The main idea of Adaptive Solid Texturing is to utilize the knowledge of the cache mechanism used in computer hardware. Using the cache mechanism, the computational cost could be reduced, since the texture date which has been calculated is not re-calculated again. Furthermore, because the visible portion of a 3D model is not so large, we can show the model with high-resolution appearance using only a few storage spaces for the cached texture data.

1.9. Organization

The rest parts of this dissertation are organized as follows: Chapter 2 introduces the jGL, a 3D graphics library for Java, as a programming environment of Web Graphics which is also our working space of whole works. Chapter 3 describes the streaming mesh; the generation, transmission, and

reconstruction of it are also included in this chapter. Chapter 4 shows two other works for Web Graphics which are a VRML library and an adaptive method of procedural solid texturing. These two works could also be used for developing 3D graphics applications running on the Web, and form as extensions of jGL. Finally, there are some conclusions and future work in Chapter 5. Additionally, the usage of jGL, the porting experimentation of jGL to mobile phones, and two FFD (Free-Form Deformation) methods to support the editing of the streaming mesh are described in the Appendix.

Chapter 2
jGL - a 3D Graphics Library for Java

2.1. Introduction

jGL is a 3D graphics library for Java [2] [30]. Unlike Java 3D [72] or other similar libraries, jGL does not need any native codes or pre-installed libraries, it is developed with exclusively the Java programming language. Users who run any programs developed with it do not need to install any package before using them; all required codes will be downloaded at run time. Because jGL is written in pure Java only, it is really platform independent, and could be executed on any Java enabled machine. Moreover, since OpenGL [70] is so famous and known by many 3D graphics programmers, we followed its specifications to develop jGL. Therefore, programmers who want to use jGL to develop their own programs for Web Graphics do not need to learn how to use a new library; they can find one-to-one mapping functions in jGL and those in OpenGL.

As the development experience of jGL, we find that the run-time performance is not the only important problem, the code size is also a problem needed to be concerned. In practice, although the performance of computer hardware improved several times, but the network bandwidth does hardly be improved. For example, more than five years ago, we used a PC with an Intel Pentium 200MHz CPU as our best testing platform and a 100Base-TX (IEEE 802.3u) Ethernet. Now, we are using a PC with an Intel Pentium 4 2.8GHz CPU as our best machine, but the network cable is just improved a little, and many people are still using 100Base-TX Ethernets to access the Internet through their local area network (LAN). Hence, we develop jGL with the minimum code size while keeping its run-time performance to be usable.

2.2. Related Projects

For Web Graphics, several companies are trying to combine the 3D graphics capability with the Java programming language. In this section, some of them are introduced: two are 3D graphics libraries for Java; others are used for displaying 3D models or scenes. Although there are several Java bindings for OpenGL, we just introduce one of them, since the advantages and disadvantages of those implementations are quite the same.

1. **Java 3D**

   Java 3D is provided by Sun Microsystems, Inc., but has not become a part of core Java package. Java 3D is based on OpenGL or Microsoft DirectX, which is needed to be pre-installed. Since it can get the benefits from the graphics hardware, the performance is good, but the platform dependent problem is occurred. To use a program based on Java 3D, the users have to install the run-time package of Java 3D before using it. Moreover, Java 3D has its own API (Application Programming Interface), so people who want to develop some 3D graphics programs with Java 3D may spend much time to learn how to use it.

2. **JSparrow**

   JSparrow[1] is an implementation of Java binding for OpenGL provided by PFU, Ltd. Since it needs the support of native OpenGL, it also has the platform dependent problem as Java 3D. Moreover, to use a program developed with JSparrow, the users also have to install the JSparrow package before using it.

3. **Eyematic Shout3D**

   Eyematic Shout3D[2] is a commercial product of Eyematic Interfaces, Inc. It uses pure Java and can display 3D models on the Web. Although the file format it used is not VRML (Virtual Reality Modeling Lan-

---

[1] *http://home.att.ne.jp/red/aruga/jsparrow/*

[2] *http://www.shout3d.com/products_shout3d.html*

guage) [6], it provides a converter from VRML to its own file format (not one-to-one mapping). Like VRML, Shout3D also provides its own API to let people be able to program animations of the 3D models.

4. **blaxxun3D**

   blaxxun3D[3] is a commercial product of Blaxxun Interactive, Inc. As Eyematic Shout3D, it was also developed using pure Java and can display 3D models on the Web. blaxxun3D can read VRML and draft X3D (Extensible 3D) [78] files, although it does not fully support both. Following the concepts of VRML EAI (External Authoring Interface), blaxxun3D also provides an API to let people be able to program their 3D models.

5. **Cortona Jet**

   Cortona Jet[4] is a commercial product of Parallel Graphics, Ltd. As Eyematic Shout3D and blaxxun3D, it was also developed using pure Java and can display 3D models on the Web. Cortona Jet can read VRML file format, but does not provide any API for programming.

6. **Xj3D**

   Xj3D[5] is an open-source project of the Web3D Consortium, Inc. Xj3D is also developed using Java, but the 3D graphics rendering is based on Java 3D. It can read VRML and draft X3D file formats exactly, and is the testing platform of the X3D file format, which is the next generation of VRML.

Since Java 3D is not platform independent and programmers need to pay more time to study how to use it, a real platform independent 3D graphics library with a familiar API is useful.

---

[3] http://www.blaxxun.com/products/blaxxun3d/

[4] http://www.parallelgraphics.com/products/jet/

[5] http://www.web3d.org/TaskGroups/source/xj3d.html

2.3. Introduction to OpenGL

OpenGL (for "Open Graphics Library") graphics system is a software interface to graphics hardware. The interface consists of a set of several hundred procedures and functions that allow a programmer to specify the objects and operations involved in producing high-quality graphical images, specifically color images of 3D objects.

To the programmer, OpenGL is a set of commands that allow the specification of geometric objects in two or three dimensions, together with commands that control how these objects are rendered into the framebuffer. For the most part, OpenGL provides an immediate-mode interface, meaning that specifying an object causes it to be drawn.

OpenGL allows a programmer to create interactive 3D graphics applications that produce color images of moving 3D objects. With OpenGL, a programmer can control computer graphics technology to produce realistic pictures or ones that depart from reality in imaginative ways.

OpenGL is designed as a streamlined, hardware independent interface to be implemented on many different hardware platforms. To achieve these qualities, no commands for performing windowing tasks or obtaining user input are included in OpenGL; instead, a programmer must work through whatever windowing system controls the particular hardware he or she is using. Similarly, OpenGL does not provide high-level commands for describing models of 3D objects. Such commands might allow programmers to specify relatively complicated shapes such as automobiles, parts of the body, airplanes, or molecules. With OpenGL, a programmer must build up his desired model from a small set of geometric primitives: points, lines, and polygons.

2.4. OpenGL vs. jGL

The functions of OpenGL as shown in Figure 2-1 can be divided into two main categories: GLU (OpenGL Utility Library) [16] and GL. There is also an additional part for specific native systems. For the X Window System, it is GLX (OpenGL Extension for the X Window System) [76] and a similar component called WGL is for Microsoft Windows systems. There are also other extensions for other window and operating systems; we just use GLX and WGL as the examples.

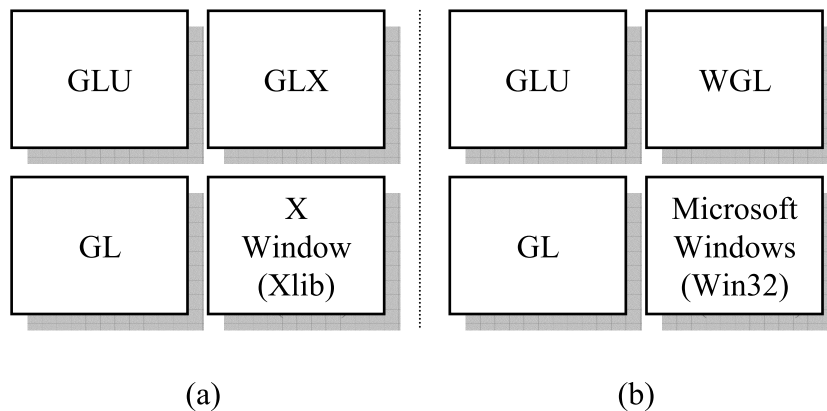


Figure 2-1: OpenGL API hierarchy for (a) the X Window System and (b) Microsoft Windows systems. GLU and GL are platform independent modules, but GLX and WGL are used to link different native window systems.

GL implements a powerful but small set of drawing primitive 3D graphics operations, such as rasterization, clipping, etc., and all higher-level drawing must be done in terms of these. GLU provides higher-level OpenGL commands to programmers by encapsulating these commands with a series of GL functions. GLX and WGL are the OpenGL extensions for the X Window System (Xlib) and Microsoft Windows systems (Win32); they are implemented depending on different platforms. Besides these main interfaces, there is an OpenGL Utility Toolkit (GLUT) [45], which also provides higher-level commands and makes programmers to not worry about which native system is used or will be used by users, like GLX or WGL. GLUT is

not an official part of OpenGL API, but is widely used and familiar to many 3D graphics programmers. For this reason, we also include GLUT in jGL. Therefore, we follow the above function hierarchy to develop the API hierarchy of jGL as shown in Figure 2-2.

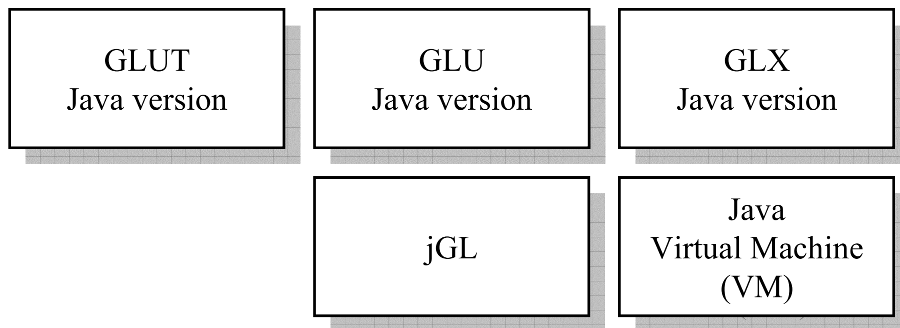


Figure 2-2: jGL API hierarchy. GLU and jGL are correspondence to GLU and GL as Figure 2-1. GLX is defined for Java VM, and GLUT is also provided as a part of jGL.

The implementation of jGL is mainly based on the specifications of OpenGL [16] [70]. Because the Java virtual machine (VM) is neither the X Window System nor Microsoft Windows systems, to make the programmers use jGL similar to using OpenGL, we still follow the specification of GLX [76] to design the API of this part. For the purpose of easy using, we also provide GLUT in Java version. Besides GL, GLU, GLX, and GLUT, which are just interfaces for programmers, there is an underlying graphics context, which is transparent to programmers, and they cannot use this part directly.

2.5. Graphics Context

The hierarchy of OpenGL modules is shown in Figure 2-3. There are four interfaces which will be used by the 3D graphics applications in OpenGL: GLUT, GLX or WGL, GLU, and GL. The applications will call the functions in the four modules. The hierarchy of jGL as shown in Figure 2-4 is

designed to follow this hierarchy.

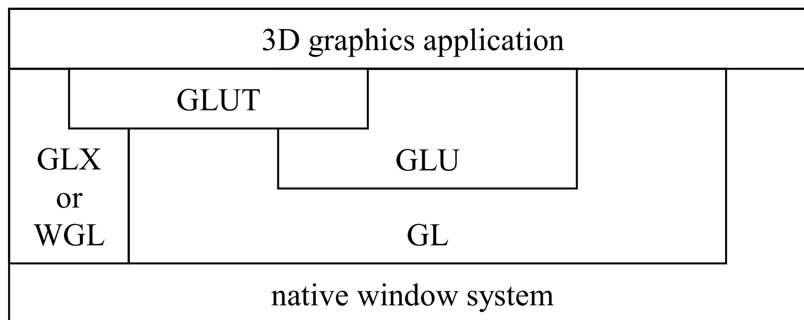


Figure 2-3: The hierarchy of OpenGL modules.

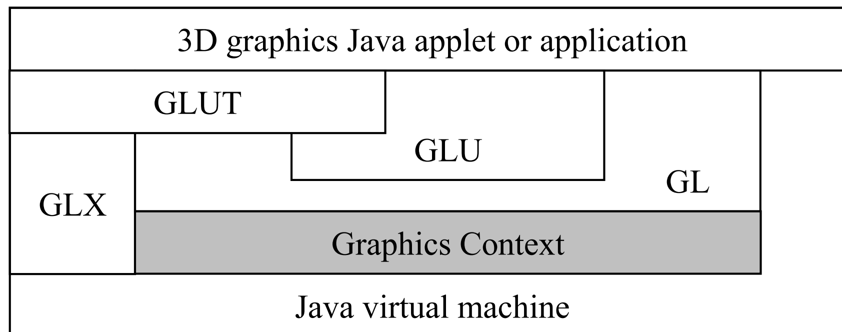


Figure 2-4: jGL implementation hierarchy. Besides GLUT, GLU, and GL, there is an underlying graphics context, which is transparent to programmers.

There are three components which could be used by the 3D graphics Java applets or applications: GLUT, GLU, and GL. For the Java applets or applications, these three components are three classes like other Java classes. What the Java applet or application needs to do is just “new” the classes then using the member functions of these classes as using the commands of OpenGL. For the platform dependent GLX or WGL, we follow the specification of GLX to provide programmers some similar functions to bind Java virtual machine with jGL, but we strongly recommend them to use GLUT instead of GLX, since GLUT is platform independent and provides the same mechanism of GLX or WGL. As Figure 2-4, GLUT, GLU, and GL are only

programming interfaces for programmers, the main part of jGL is the graphics context which is transparent to programmers, and provides a set of primitive rendering routines.

To reduce the byte-code size and keep or enhance the run-time performance of jGL in the same time, we utilize the class inheritance characteristics to design the system hierarchy of graphics context as shown in Figure 2-5. The graphics context of jGL can be divided into two parts. One is for display list: all of the commands from jGL will not be executed but just stored as a sequence of rendering commands. The other is for real graphics context, all of the commands from jGL will be executed immediately.

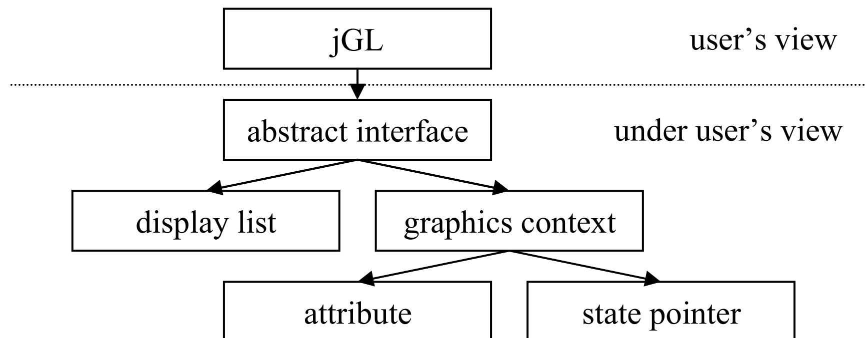


Figure 2-5: The graphics context of jGL.

The commands that will be executed in the real graphics context also can be categorized into two types. One is just for changing some information stored in the graphics context. For example, when the user calls the glClearColor command, the color value to clear the framebuffer will just be stored; no real clear actions will be performed. Once the user calls the glClear command (with GL_COLOR_BUFFER_BIT), the real clear action will be executed by using the clear color value, which has been stored before.

The other type is like the previous glClear command. The commands in this type will perform some actions and directly produce some changes. Since jGL is defined as a state machine as OpenGL, we utilize class inheritance to avoid frequent checks here. The states of jGL have been classified into several states; including selection or not, flat or smooth shading, with depth test or without, texture mapping enabled or disabled, clipping for

clip-plane or not. Therefore, running in different states will need different clipping, geometry, and rendering routines.

2.6. Implementation Problems

Developing such a 3D graphics library using pure Java programming language seems a tough task. During the design phase, we have some problems as following:

1. **Performance challenge**

   On the underlying graphics context, we put most of our efforts here, since the performance is a great challenge for Java applets or applications and also for general-purpose software-based 3D graphics engines. For instance, we subdivide drawing functions into several smaller ones to optimize these functions and class inheritance is used to avoid the use of “if-then-else” statements in using these functions. We also refer to Graphics Gems [3] [28] [33] [46] for performance enhancements. For the detail descriptions of the performance enhancement, please refer to Section 2.7.

2. **Byte-code size consideration**

   The byte-code size of jGL is also a serious problem, because it will be downloaded at the run-time when the users at the client site use the 3D graphics Java applets developed with it. If the size of jGL is too large, the users will wait for a long time to download it. However, the byte-code size of the library and the performance of it seems be constrained, sometimes the byte-code size and the performance considerations are trade-off.

3. **Differences between Java and C**

   The API of OpenGL is defined for the C programming language, not for Java which is an OOP (Object-Oriented Programming) language. We must design the API of jGL based on the OOP philosophy of Java that

implies the graphics engine should be designed as several classes.

4. **No pointer data type in Java**

   The pointer is very useful for programming. For instance, a drawing command in OpenGL may be executed immediately or be postponed in a display list depending on the state of the graphics context. In C, we can simply use function pointers to solve this problem. In Java, however, since there is no pointer, we use class inheritance instead. Therefore, we use class reference instead of using structure pointer, and use the Vector class in Java instead of using linked list which is also a useful data structure usually used by programmers.

5. **Java is just running on a virtual machine**

   Java is not running on a real machine, but just a virtual machine. The machines which the Java applets or applications run on are like the devices of this virtual machine. Therefore, we can not write any assembly routines for performance enhancement because we do not know which machine is the base of the program. Even if the real machine has a hardware accelerator for 3D graphics algorithms, we can not use the feature, yet. Like our performance evaluation, when we test some examples on a machine with a hardware accelerated display card, all of the 3D graphics enhancement features of the card are negligible for the examples.

2.7. Performance Enhancement Issues

Performance is a great challenge for both 3D graphics and Java; hence it is also a great challenge for jGL witch is a Java-based general-purpose 3D graphics library. Moreover, jGL is designed to operate over the Internet, where network bandwidth affects the overall performance significantly. Since we want to enhance the capabilities and minimize the byte-code size without making the run-time performance worse, these considerations make the implementation of jGL complex. In some cases, the byte-code size and the run-time performance are trade-off. Based on our experiences, we de-

veloped jGL with the following policies to speed up the run-time performance and minimize the byte-code size of it.

1. **Utilize Java functions**

   If there is a well-designed Java function, we utilize this function instead of re-writing another one by ourselves. Because Java plays the rule as a bridge between jGL to several native platforms like the device drivers of several adapters, if some of the Java functions are implemented by using native codes or system calls, there will be a shorter path between jGL to several native platforms. Then, the benefits of the native platforms could be used in jGL.

2. **Utilize class inheritance to avoid "if-then-else" statements**

   OpenGL is a state machine, so it is usually necessary to determine if some statuses are enabled or not, which take time to check. Since many conditions are always needed to be determined only once, after deciding which statuses are the current ones, they will not be changed and the graphics library will not need to determine them again. For example, when implementing the display list, we set a flag to indicate whether the rendering commands are to be stored or to be executed. Therefore, for each rendering command, we need to check if the flag is set or not, and it will slow down the execution speed.

   Hence, we utilize class inheritance to avoid these frequent checks, so that the status checks will be realized only when the statuses changed. When a status changed, the status pointer as shown in Figure 2-5 will be changed to point to its proper class type, so all of the rendering commands followed will be routed to proper functions automatically without any further checks. For example, the class for the display list and the class for the executed functions are both inherited from the same parent class. After the dispatcher checks the flag, all of the following rendering commands are realized in either the display list or the executed functions without any further checks of the flag.

3. **Use variables or tables to avoid re-calculation**

   This is a common method which is always used by programmers. Some parameters are calculated with complex algorithms, but only needed to be calculated once, like the parameters used in the 3D transformation or

the lighting calculation. Using variables or tables to catch the calculated parameters will reduce the re-calculation time.

4. Make frequently used routines faster

Polygon rasterization, shading, depth testing, clipping, etc., are frequently used routines. They are always the bottlenecks of the 3D graphics library, so we put most of our efforts to optimize these routines by introducing faster algorithms and manual code optimization.

5. Divide a routine into several smaller ones

Some routines will be called several times, to optimize all of the drawing functions is therefore important. However, many such routines have much unnecessary codes in them. If a routine was very large and would be called in several situations, this routine must have some useless code segments for some statuses, while it is just called for a simple situation. So, it is worthwhile to divide this routine into several smaller ones.

For example, to fill a polygon, we must do color interpolation if the polygon is filled using smooth shading. However, if the polygon only requires flat shading, the color interpolation would be unnecessary. Therefore, we divide the shading routine into two smaller ones by categorizing all of the drawing functions, such as the drawing functions with or without depth testing, the drawing functions with flat shading or smooth shading. Then, we make these functions to be optimized and use class inheritance to use them.

6. Use function overriding to minimize code size

Once we divide some frequently used routines into several smaller ones, the total byte-code size of the library will be larger than before, because there are too many duplicated codes. This is the side effect of the performance enhancement. Since we have utilized class inheritance to avoid “if-then-else” statements, we also use this method to structure all of the small routines and then use function overriding to reduce the duplicated codes.

The same example as the one above, since we have divided the shading routine into two smaller ones: one is for flat shading and the other is for smooth shading. Because all of them need the polygon scan procedure

to fill the polygon, we can put the two routines in the parent and child classes, and use the same code base to do the polygon scan. The difference between the two small routines is just for interpolation. In the flat shading, we only need to interpolate the vertex positions in the polygon, but in the smooth shading, we also need to calculate the color interpolation. Therefore, the function-overriding example is like Figure 2-6.

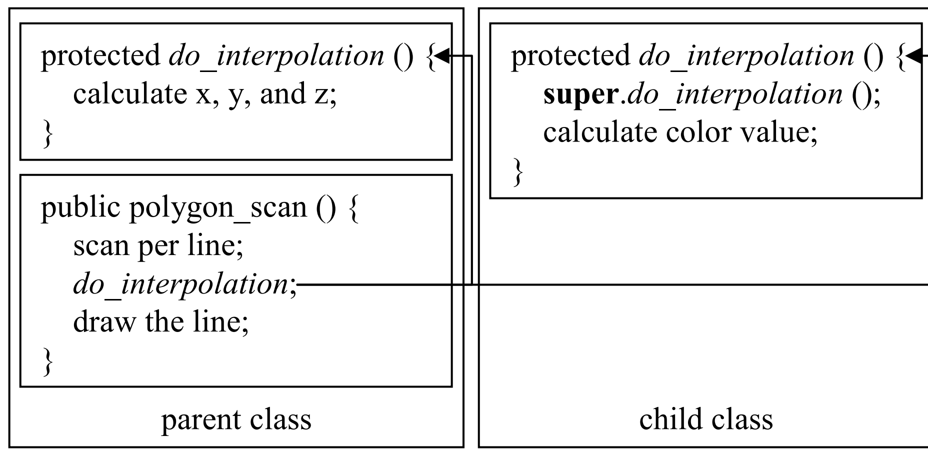


Figure 2-6: The example of using function-overriding. The *do_interpolcation* procedure has been divided into two smaller ones for different requirements, and the main procedure can call exact *do_interpolcation* procedure according to current status, since we are using function-overriding.

2.8. Results

The newest version of jGL is version 2.4 beta 2.3, which has been released in December 17th, 2002, and we still keep the development going. The following performance evaluations and descriptions are all for this version. Since jGL is developed in pure Java, it can be used on all Java-enabled machines. We have performed tests on all of major operating systems including Microsoft Windows 98/NT/2000/XP, Sun Solaris 7/8/9 (SPARC and Intel

platform editions), SGI IRIX 6.3/6.4/6.5, and Linux. Although we are using Java 2 SDK, Standard Edition (J2SE) v 1.4.1_01 as the development environment, jGL is still compatible with Java Development Kit (JDK) 1.1.x.

2.8.1. Supported Functions and Usage of jGL

Currently, we have implemented more than 300 OpenGL functions in jGL, including functions of GLUT, GLU, and GL. These functions include 2D/3D model transformation, 3D object projection, depth buffer, smooth shading, lighting, material property, display list, selection, texture-mapping, mip-mapping, evaluator, NURBS (Non-Uniform Rational B-Splines), stippled geometry, etc. All of the functionalities supported by jGL are listed in Appendix A.2. Besides these functionalities, jGL can also support Phong shading, bump mapping, environment mapping, procedural solid texturing, and VRML file format. Because these functions are not included in OpenGL, we make them to be the extensions of jGL, and the details of two of them are described in Chapter 4.

Figure 2-8 shows the source code of a Java applet as a simple jGL program using GLUT to show a white rectangle. The same program written with OpenGL is listed in Figure 2-7 that is an example provided in the OpenGL Programming Guide [77] (code from Example 1-2, pages 18-19, Figure 1-1). For the details of the usage of jGL, please refer to Appendix A. We also tried to port jGL to the mobile phones; the porting experimentation is described in Appendix B.

```
#include <GL/glut.h>

void display (void) {
    glClear (GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);
    glBegin (GL_POLYGON);
        glVertex3f (0.25, 0.25, 0.0);
        glVertex3f (0.75, 0.25, 0.0);
        glVertex3f (0.75, 0.75, 0.0);
        glVertex3f (0.25, 0.75, 0.0);
    glEnd ();
    glFlush ();
}

void init (void) {
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    glOrtho (0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
}

int main (int argc, char** argv) {
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (250, 250);
    glutInitWindowPosition (100, 100);
    glutCreateWindow ("hello");
    init ();
    glutDisplayFunc (display);
    glutMainLoop ();
    return 0;
}
```

Figure 2-7: A simple OpenGL program using GLUT to show a white rectangle. This program is an example in the OpenGL Programming Guide (code from Example 1-2, pages 18-19, Figure 1-1).

```
import jgl.GL;
import jgl.GLApplet;

public class hello extends GLApplet {

public void display () {
    myGL.glClear (GL.GL_COLOR_BUFFER_BIT);
    myGL.glColor3f (1.0f, 1.0f, 1.0f);
    myGL.glBegin (GL.GL_POLYGON);
        myGL.glVertex3f (0.25f, 0.25f, 0.0f);
        myGL.glVertex3f (0.75f, 0.25f, 0.0f);
        myGL.glVertex3f (0.75f, 0.75f, 0.0f);
        myGL.glVertex3f (0.25f, 0.75f, 0.0f);
    myGL.glEnd ();
    myGL.glFlush ();
}

private void myinit () {
    myGL.glClearColor (0.0f, 0.0f, 0.0f, 0.0f);
    myGL.glMatrixMode (GL.GL_PROJECTION);
    myGL.glLoadIdentity ();
    myGL.glOrtho (0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 1.0f);
}

public void init () {
    myUT.glutInitWindowSize (500, 500);
    myUT.glutInitWindowPosition (0, 0);
    myUT.glutCreateWindow (this);
    myinit ();
    myUT.glutDisplayFunc ("display");
    myUT.glutMainLoop ();
}

}
```

Figure 2-8: The source code of a simple jGL program using GLUT to show a white rectangle according to Figure 2-7.

2.8.2. Performance Evaluation

To test the capabilities of jGL, we have provided 30 examples on our jGL Web page[6]. These examples are selected from the OpenGL Programming Guide which is an official programming guide of OpenGL. All of these examples can be executed directly on Java-enabled Web browsers.

rendering time	platform
270 ms	Intel Mobile Pentium III 850MHz, 512 MB memory, Microsoft Windows XP Professional
109 ms	Intel Pentium 4 2.8GHz, 1 GB memory, Microsoft Windows XP Professional
904 ms	Sun UltraSPARC IIi 360MHz, 256MB memory, Sun Solaris 9

Table 2-1: The performance testing of jGL on different platforms by using 12 spheres drawn with different material properties as shown in Figure 2-10.

To evaluate the run-time performance of jGL, we used two test programs. One renders 12 spheres drawn with different material properties, where each sphere contains 256 polygons, as shown in Figure 2-10. This test program is an example in the OpenGL Programming Guide (code from Example 5-8, pages 205-206, Plate 16). The other test program renders 24 lighted, smooth-shaded teapots drawn with different material properties that approximate real materials, where each teapot is generated by jGL Bésier surface generating functions (evaluator functionality of OpenGL) and contains 12,544 polygons, as shown in Figure 2-11. This test program is also an example in the OpenGL Programming Guide (Plate 17). The run-time performances are measured on a Sun Ultra 10 Workstation with a Sun UltraSPARC IIi 360MHz CPU (256MB memory, Sun Solaris 9), a laptop PC with

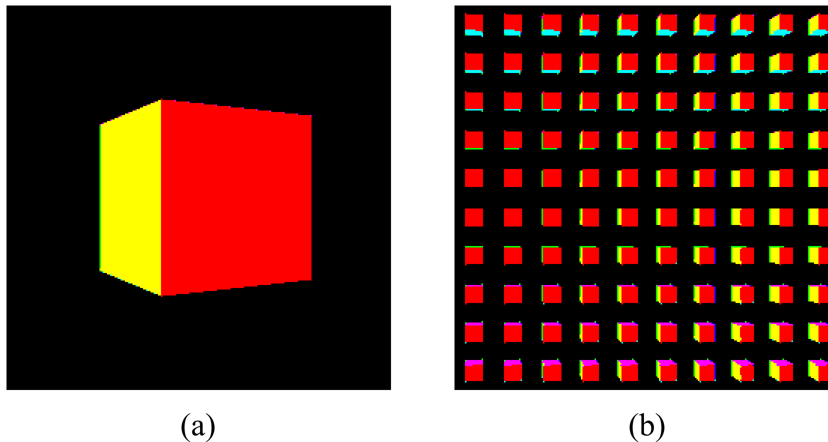
---

6 *http://nis-lab.is.s.u-tokyo.ac.jp/~robin/jGL/*

an Intel Mobile Pentium III 850MHz CPU (512MB memory, Microsoft Windows XP Professional), and a desktop PC with an Intel Pentium 4 2.8GHz CPU (1GB memory, Microsoft Windows XP Professional). The results are listed in Table 2-1 and Table 2-2.

rendering time	platform
1,682 ms	laptop PC as Table 2-1
609 ms	desktop PC as Table 2-1
4,507 ms	workstation as Table 2-1

Table 2-2: The performance testing of jGL on different platforms by using 24 lighted, smooth-shaded teapots drawn with different material properties that approximate real materials as shown in Figure 2-11.



(a) (b)

Figure 2-9: The programs used to measure the performances of jGL and Java 3D. (a) A rotating cube drawn with different color values similar to the sample Java 3D example in the Java 3D API Specification (Section 1.6.3, pages 9-10). (b) An example to repeat the same cube 100 times for testing the performance. These figures are rendered with jGL.

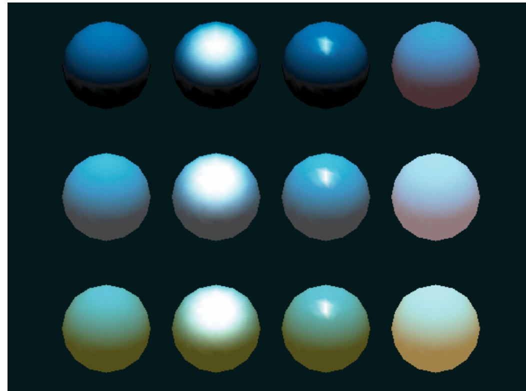


Figure 2-10: 12 spheres, each with different material properties, are rendered to measure the performance. This program is an example in the OpenGL Programming Guide (code from Example 5-8, pages 205-206, Plate 16). This figure is rendered with jGL.

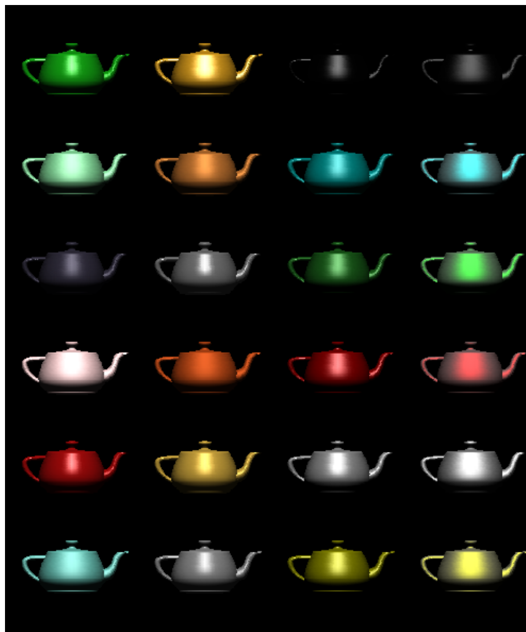


Figure 2-11: 24 lighted, smooth-shaded teapots drawn with different material properties that approximate real materials are rendered to measure the performance. This program is an example in the OpenGL Programming Guide (Plate 17). This figure is rendered with jGL.

To compare jGL with Java 3D, we wrote a program that draws a rotating cube drawn with different color values similar to the “HelloUniverse”, which is a simple Java 3D example in the Java 3D API Specification [72] (Section 1.6.3, pages 9-10) as shown in Figure 2-9 (a). We tested it and the Java 3D example on the laptop and desktop PCs as used in Table 2-1. There is no graphics accelerator installed in the laptop PC, and the graphics accelerator installed in the desktop PC is using an NVIDIA Quadro 4 900XGL GPU (Graphics Processing Unit), which is a high-level graphics accelerator. The results show that both of them are rendered in real-time even on the laptop PC.

library / platform	jGL	Java 3D
laptop PC as Table 2-1.	1,202 ms	
desktop PC as Table 2-1, NVIDIA Quadro 4 900XGL	406 ms	353 ms

Table 2-3: The performance comparisons for jGL and Java 3D on different platforms by using 6,400 rotating cubes drawn with different color values as shown in Figure 2-9 (b).

Since the run-time performances can not be estimated by using a simple model, we use the same cube but repeat it 6,400 times to measure their run-time performances. Figure 2-9 (b) shows the similar example of repeating the same cube only 100 times. The results are listed in Table 2-3. To use more cubes for testing is possible for jGL, but it caused a “memory overflow” error while using Java 3D. Because there is no graphics accelerator installed in the laptop PC, the program using Java 3D can not be run on it[7]. Furthermore, Java 3D currently only supports Microsoft Windows systems and Sun Solaris operation systems, but the programs developed with jGL can be run on more machines than those with Java 3D and can be used on the Web directly. Moreover, the byte-code size of jGL is less than 150KB (jar[8]-compressed), using jGL for Web Graphics is more suitable than using

[7] We only used the OpenGL version of the Java 3D for testing.

[8] “jar” is used for compressing and archiving the Java byte-codes to be a “jar-ball”.

other libraries which need the supports from the native 3D graphics libraries.

	excellent	good	bad	very bad
easy to use	5	45	0	0
reliability	8	42	0	0

Table 2-4: The questionnaire result of jGL.

Since we offer jGL onto our Web server, many people around the world are using it to provide their previous OpenGL works to be Web-enabled versions or to teach and learn the computer graphics algorithms. In order to get the feedbacks from the users, we make a questionnaire of jGL and mail to 50 users around the world. The result is shown in Table 2-4. Most of the users thought that jGL is easy to use and has good reliability.

2.9. Summary

This Chapter presented a new programming platform for Web Graphics. To develop 3D graphics programs on the Web is not very easy, because there is no high-quality library like OpenGL. For this purpose, jGL is developed as a 3D graphics library for the Java programming language using pure Java. jGL is a general-purpose software-based 3D graphics library, and its API is defined in a manner quite similar to that of OpenGL. Today, the computer hardware is better, but the network bottleneck is still the same as before, so we enhance the capabilities and run-time performance of jGL and also try to minimize its byte-code size to make it more suitable for running on the Web.

Chapter 3

Streaming Mesh with Shape Preserving

3.1. Introduction

How to transmit geometric 3D models efficiently through the Internet is an important topic for Web Graphics, since 3D models are widely used and the data sizes of them are usually large. Therefore, we present a new multiresolution streaming mesh for Internet transmission with a QoS-like controlling. When a user needs to use a 3D model encoded with the streaming mesh format on the Web page, the server first delivers a simplified model which has easily recognizable shape and features with the data size according to a QoS-like controlling. QoS (Quality of Service) [4] [5] is a protocol of computer network technologies, which provides the mechanics to differentiate traffic, so that users can get different quality of video or audio due to different bandwidth of the network environment with the same continuity. In our approach, we use the same concepts to provide the geometric 3D models with different resolutions to the users at heterogeneous client sides, and the users will pay the same waiting time to get the 3D models with different resolutions. Since this is not the definition of the original QoS, we call this approach as "QoS-like" in our method.

There are three processes to construct the streaming mesh. The first process is the 3D mesh simplification that is used for simplifying the mesh data which constitutes the geometric 3D model. The second one is the storage methodology for the simplified model and the patches which include the removed information during the simplification process and will be used to reconstruct the original model. The last one includes the transmission of the streaming mesh with a QoS-like controlling and the reconstruction of the original 3D model.

Although there are many methods for simplifying geometric 3D models [24] [34] [55] [57], most of them are generally time-consuming due to model optimization. These time-consuming methods are generally expected to be used directly if the model provider wishes to preview the simplified model by changing some of the parameters interactively. Moreover, the simplified models created by some previous methods can not be easily recognized nor used to reconstruct the original model.

The basic idea of our method is to segment the unstructured meshes of a geometric 3D model into several parts first using feature detection methods, and then simplifying each part of the meshes iteratively. When a model provider wishes to upload a geometric 3D model onto a Web server using our approach, the provider could first use our system to preview what kind of simplified model is to be used by the users, and could change some of the parameters to obtain interactively a better simplified model. On the client site, the user first receives the simplified model from the Web server, which is the base part of the streaming mesh. Of course, with the QoS-like controlling, the wider network bandwidth the user uses, the better 3D model could be received from the server.

Then, if the user needs to use the model with more details, the server will then transmit some additional necessary information to the client also with the QoS-like controlling, so that the client program can show increasing model detail progressively and the user also gets different progressive patches due to the current network bandwidth. Finally, if the user really needs the original model, after receiving all of the patches, the system is then able to reconstruct the original 3D model with no losses and no retransmission of information.

Moreover, to make our approach widely used in the Web world, we have developed all of the algorithms using exclusively the Java programming language [2] [30] for its hardware-neutral features and wide availability on many hardware platforms. Additionally, the 3D graphics rendering is done by jGL.

3.2. Related Work

Many researches have been carried out on simplifying and transmitting the meshes of geometric 3D models on the Internet. Some of them transmit almost optimized meshes which can represent fine shape and preserve the features of the original model with a small data size [18] [27]. Others simplify the meshes iteratively and store the removed information which can then be used to progressively reconstruct the lossless original model. Since our motivation is to transmit the geometric 3D model through the Internet, it is necessary to reconstruct the original model with a small amount of data transmission. Therefore, our approach belongs to the latter category. However, to allow it to be used for Web Graphics, where the run-time performance is more important than providing an almost perfect model, a more efficient method is needed. In this section, some related methods are introduced briefly.

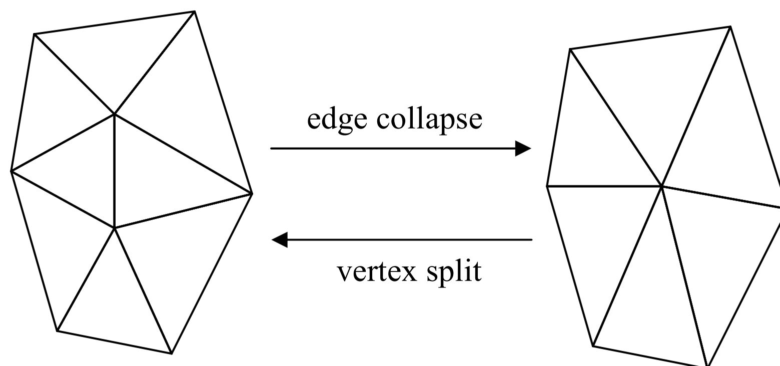


Figure 3-1: The edge collapse operation. This operation is also called edge contraction and its reverse operation is vertex split.

PM (Progressive Meshes) is a famous method for 3D mesh simplification and is based on the “edge collapse” or “edge contraction” operation provided by Hoppe [35] [37], Hoppe et al. [41], and Sander et al. [66] as shown in Figure 3-1. Simply speaking, this mesh simplification method is to find the minimum value of an energy function which is based on the geometric

distance between the original 3D model and the simplified one. Although this method could result in an almost optimized simplified model, it is well known to be time-consuming.

A derived method, QEM (Quadric Error Metrics), has been provided by Garland and Heckbert [25] [26], and Hoppe [39] to make the calculation faster by calculating the error quadrics of newly generated vertices due to the edge collapse or edge contraction operation. The heuristic function used by QEM is geometry-based, since it calculates the geometric distances between the newly generated vertex and the faces which are deformed before generating it. Although QEM could enhance the run-time performance of the 3D mesh simplification process, the simplified model is sometimes hardly recognizable due to the over-simplification. An image-based heuristic function is presented by Lindstrom and Turk [54]. It captures 20 images in every simplifying step and is time-consuming obviously.

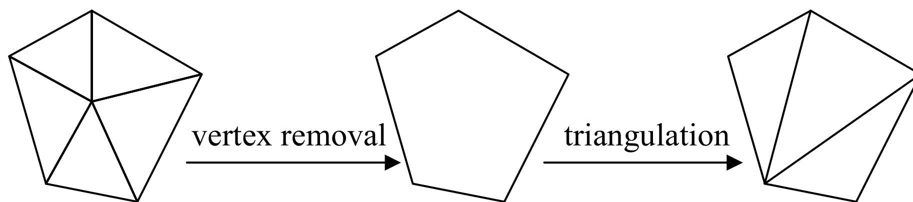


Figure 3-2: The vertex decimation method. The vertex removal operation is first performed to remove one vertex and then the remaining hole is re-triangulated.

Other algorithms are based on the “vertex decimation” method, which are provided by Alliez et al. [1], Schroeder et al. [67], and Turk [73]. The main processes of these algorithms are finding the removable vertices, and using the “vertex removal” operation to remove all of the removable vertices as shown in Figure 3-2. Finally, re-triangulating the remaining holes left by removing the vertices. Hence, the vertices of the simplified model are the subset of the vertices of the original model, i.e. the vertex positions are the same, but the vertex connectivity is maybe changed. These algorithms are different from the previous ones; it could simplify a 3D model fast, but the shape of the simplified model maybe be changed and is hardly recognizable.

Therefore, an efficient 3D mesh simplification method with feature detec-

tion for making the simplified model remain recognizable is necessary. Such a method will be described in the following sections.

3.3. Notation

A geometric 3D model is usually represented as triangular meshes[1]. Each triangular mesh is associated by three vertices. Like other 3D model representations, a geometric 3D model M is represented as a formula containing 4 components as shown in Figure 3-3.

$$
\begin{aligned}
&M = (V, F, D, S)\\
&V = \{v_i\}_{i=1}^{m}, v_i \in \Re^3\\
&F = \{f = \{j, k, l\} \mid v_j, v_k, v_l \in V\},\ |F| \subset \Re^m\\
&D = \{d_f \mid f \in F\}\\
&S = \{s_{(v_i, f)} \mid i \in f\}
\end{aligned}
$$

Figure 3-3: The representation of a geometric 3D model M.

In Figure 3-3, V is the set of vertex positions v_i, $i \in [1, m]$, where m is the number of vertices, and defining the shape of the triangular meshes in $\Re^3$. F is the vertex connectivity of the meshes. D is the set of discrete attributes d_f, like the material property, associated with the face f, and S is the set of scalar attributes $s_{(v_i, f)}$, like the normal vector, associated with the wedge $w = (v_i, f)$ as shown in Figure 3-4. Hence, the geometry of the 3D meshes could be represented as the image $\phi_V(|F|)$, where $\phi_V : \Re^m \to \Re^3$ is a linear mapping.

[1] We convert all non-triangular meshes into triangular ones before doing the 3D mesh simplification process.

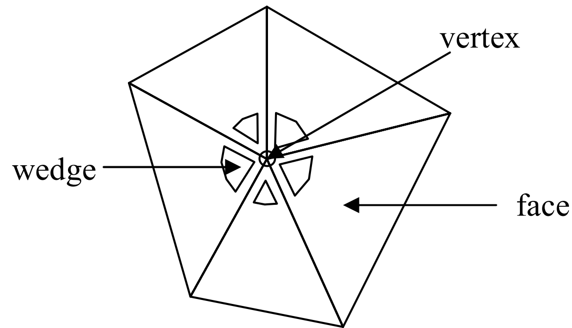


Figure 3-4: A wedge is defined as a pair of vertex and face. A vertex may be separate to be several wedges due to the different scalar attributes according to different faces connected with the same vertex.

3.4. Feature edge detection

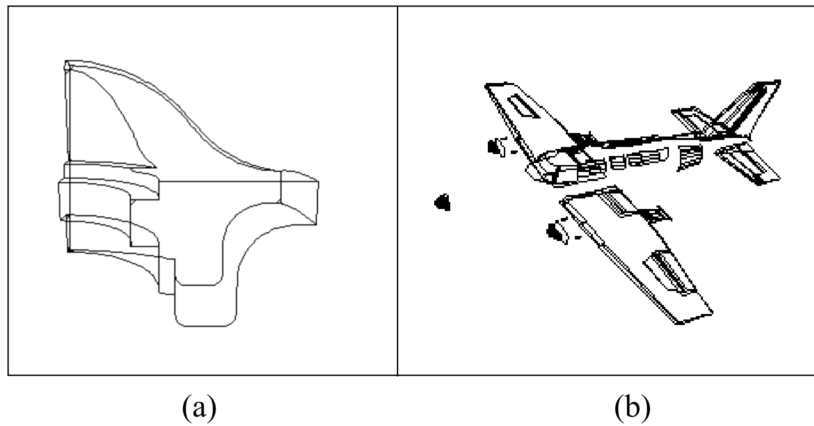
In our algorithm, to simplify a geometric 3D model efficiently and at the same time preserve its features, it is necessary to find out the feature edges of the model so that the model may be simplified by removing the non-feature edges while preserving its features.

There are two kinds of feature edges in our approach. One is the “sharp edge” due to the sharpness of the geometric differences, and also the edges of adjacent faces containing different material properties. The other is the “base edge” detected from the unstructured meshes if there is no sharp edge contained in the meshes.

3.4.1. Sharp edge definition

If a geometric 3D model is constructed with several material properties or contains some pre-defined sharp edges, which are specified when the model

is generated as shown in Figure 3-5, the sharp edges of the model could be detected easily. An edge $e = \{i, j\}$ is called a "boundary edge" if there is only one face $f = \{i, j, k\}$ with $e \subset f$. An edge $\{i, j\}$ is called a "sharp edge" if either (1) it is a boundary edge, (2) its two adjacent faces f_l and f_r have different discrete attributes, i.e. $d_{f_l} \neq d_{f_r}$, or (3) its adjacent wedges have different scalar attributes, i.e. $s_{(v_i, f_l)} \neq s_{(v_i, f_r)}$ or $s_{(v_j, f_l)} \neq s_{(v_j, f_r)}$.



(a) (b)

Figure 3-5: The examples of sharp edges due to (a) pre-defined feature edges and (b) different material properties.

3.4.2. Base edge detection

If there is no sharp edge or there are some features hidden in the meshes, the detection of base edges from the unstructured meshes is necessary. To detect the base edges, we use the ESOD (Extended Second Order Difference) operator as described in [42]. In this section, an operator called SOD (Second Order Difference) and the ESOD operator are described first.

The SOD operator is the simplest method to detect the features from unstructured meshes. It assigns a weight to every edge $e = \{i, j\}$ defined by the normal vectors of its adjacent faces as $w(e) = \mathbf{n}_{f_a} \cdot \mathbf{n}_{f_b}$. For example, in Figure 3-6, $f_a = \{i, j, k\}$ and $f_b = \{i, l, j\}$ are the adjacent faces of the

edge e, and the normal vector for face $f = \{i, j, k\}$ is defined as

$$\mathbf{n}_f = (v_j - v_i) \times (v_k - v_i) / \|(v_j - v_i) \times (v_k - v_i)\|.$$

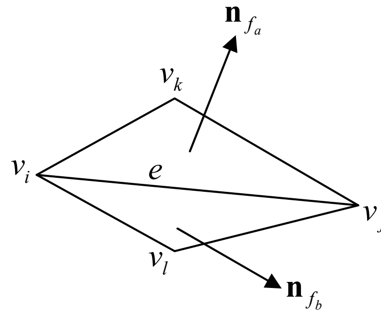


Figure 3-6: The SOD operator. The weight of edge is defined by the normal vectors of its adjacent faces.

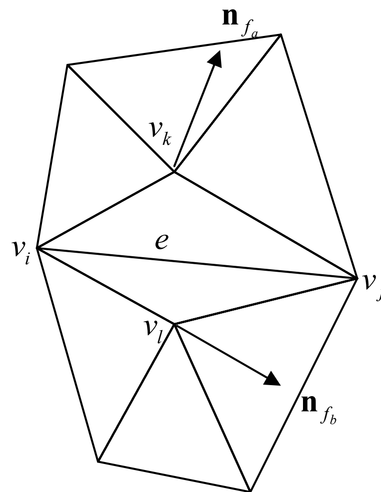


Figure 3-7: The ESOD operator. The weight of edge is defined by the average normal vectors computed from the faces on the 1-ring of the vertices v_k and v_l.

The ESOD operator extends the SOD operator. Instead of using the normal vectors of the adjacent faces of edge e, it uses the average normal vectors computed from the faces on the 1-ring of the vertices v_k and v_l as shown in Figure 3-7. Therefore, the weight of edge $e = \{i, j\}$ is defined as $w(e) = \mathbf{n}_{v_k} \cdot \mathbf{n}_{v_l}$, where the normal vector for the vertex v is defined as

$$\mathbf{n}_v = \sum_{v \in f} area(f) \cdot \mathbf{n}_f \Big/ \sum_{v \in f} area(f),$$

and $area(f)$ means the area of face f.

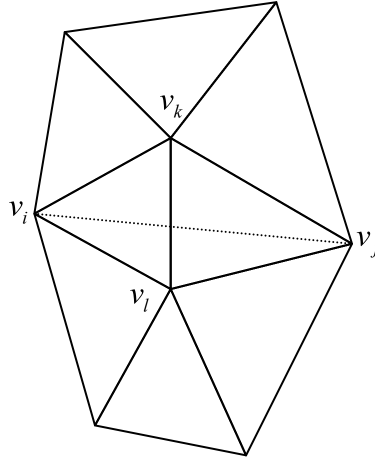
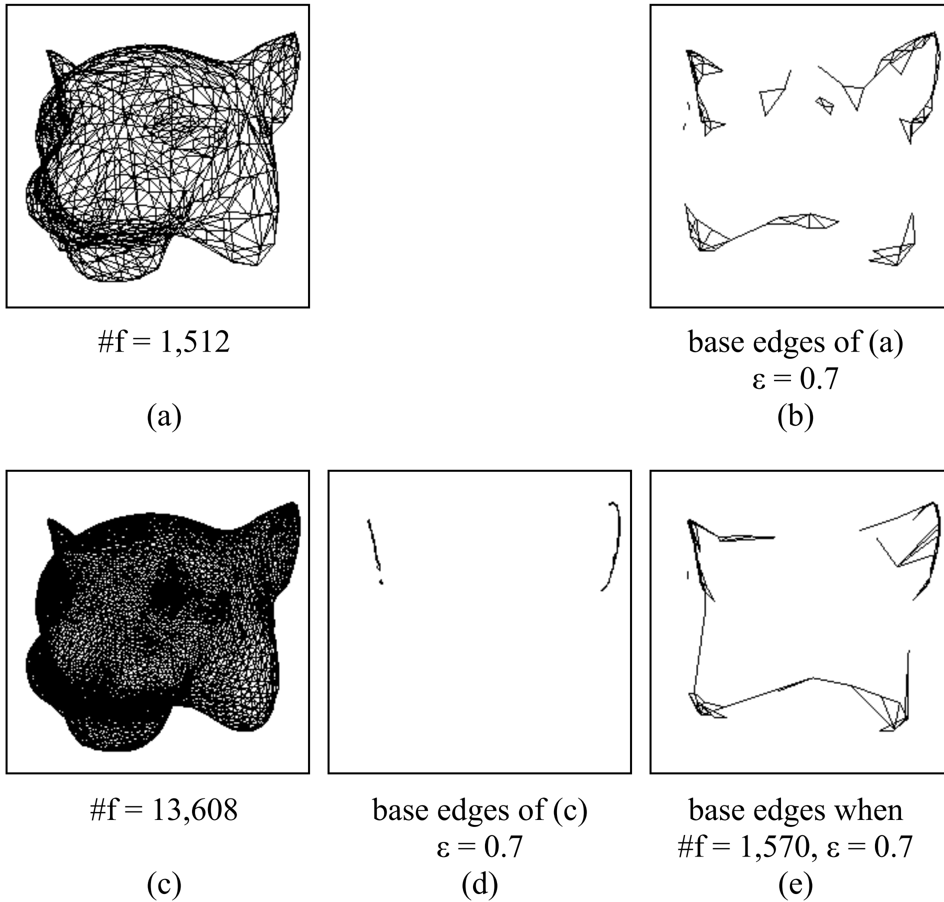


Figure 3-8: Base edge detection by finding the virtual feature edges using the ESOD operator.

Therefore, to define a proper threshold $\varepsilon = [-1,1]$, it is possible to determine the features from unstructured meshes. However, to use the SOD or ESOD operator to segment the unstructured meshes is also a time-consuming task if we use the method described in [50]. Instead of using the SOD or ESOD operator to extract the features, we use the ESOD operator to find out the “virtual feature edge”. As shown by the dotted line in Figure 3-8, the edge $e = \{i, j\}$ is called the “virtual edge” of edge $\{k, l\}$ if the edge does not exist, and there exists two faces $\{i, l, k\} \subset F$ and $\{j, k, l\} \subset F$. Furthermore, a virtual edge is called a “virtual feature edge” if

its weight calculated by the ESOD operator satisfies $w(e)=\mathbf{n}_{v_k}\cdot\mathbf{n}_{v_l}<\varepsilon$. In this case, and then the edge $\{k,l\}$ is called a "base edge".



#f = 1,512
(a)

base edges of (a)
ε = 0.7
(b)

#f = 13,608
(c)

base edges of (c)
ε = 0.7
(d)

base edges when
#f = 1,570, ε = 0.7
(e)

Figure 3-9: The examples of base edges: (a) line segments of tiger model (refer to Figure 3-22 (c) for its smooth shading representation); (b) the base edges of (a) when setting $\varepsilon=0.7$; (c) line segments of another tiger model subdivided from (a) (refer to Figure 3-21 (e) for its smooth shading representation); (d) the base edges of (c) when setting $\varepsilon=0.7$; (e) the base edges of (c) when its number of faces has been simplified to be similar with the number of faces of (a).

To make the base edges un-removable, the faces linked with the different endpoints of the base edges will not be merged during the mesh simplification process. Hence, the features of the original model could be preserved

even the model has been simplified. Someone may concern that the definition of the base edge is a little weak when the model contains a large number of faces. Actually, when the size of each face becomes small, the differences between the normal vectors of the adjusted faces will also become small, and the base edge will become hardly detected. To apply the ESOD operator to the faces belong a larger area would be a good idea to solve the problem, however it also implies the performance will become worse. Instead, we perform the base edge detection not only once, but after every loop of the mesh simplification process iteratively. Therefore, some hardly detected parts will become detectable after merging some nearby flat faces.

As an example shown in Figure 3-9, some parts of the 3D model “tiger”, which has 1,512 faces and is shown in Figure 3-9 (a), can be detected when setting $\varepsilon = 0.7$ as shown in Figure 3-9 (b), but if we use a similar model, which has more faces as shown in Figure 3-9 (c), the detected parts are only a few as shown in Figure 3-9 (d). However, after some loops of the mesh simplification process, the number of faces of the model shown in Figure 3-9 (c) has been decreased to be similar with the number of faces of the model shown in Figure 3-9 (a). Then, as shown in Figure 3-9 (e), the detected parts become as much as Figure 3-9 (b).

3.5. Mesh Simplification

The procedure for our mesh simplification method is listed in Figure 3-10. The first step is to search for sharp edges. Since the sharp edges are the edges with pre-specified features, they are used to segment the meshes into several parts. Each part of the meshes is simplified independently, and the simplification for the geometric 3D model with sharp edges is described in Section 3.5.1.

Then, the normal vector of each vertex is calculated, along with the weight of each edge besides the sharp edges. The weight of the edge is assigned the weight of its virtual edge using the ESOD operator, so that by using a threshold $\varepsilon = [-1,1]$, it is possible to detect the base edges of the meshes.

As is shown in the example in Figure 3-8, if the edge $\{i, j\}$ (the dotted line) is a virtual feature edge, the faces above the dotted line and the faces opposite the line could not be merged during the simplification process. Therefore, the endpoints of the base edges, edge $\{k, l\}$ in Figure 3-8 for instance, are specified as un-removable vertices. Otherwise, the vertices and the edges used to connect them are removable. These removable edges are pushed into a priority queue with their weights being candidate edges. Therefore, the edge with larger weight will be removed earlier.

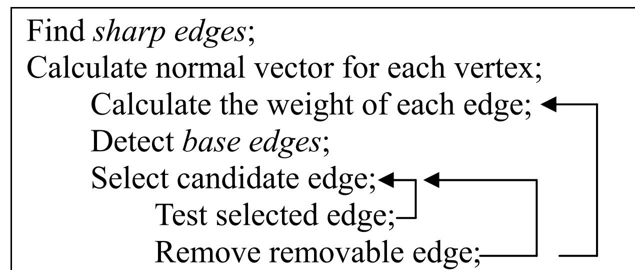


Figure 3-10: The procedure of our 3D mesh simplification method.

After removing the current removable edges, the weights of the edges which the connectivities have been changed due to the simplification process should be re-calculated, and also need to be checked to see if they become base edges or not. Then, attempts are made again to remove the removable edges, as described above, until there are no removable edges.

3.5.1. Simplification with sharp edge

To simplify a geometric 3D model with pre-defined sharp edges, which are specified when the model is generated, while preserving its features, we first search for the pre-defined sharp edges due to sharp geometric differences and also for the edges which are adjacent to faces which contain different material properties. Then, the endpoints of the sharp edges are marked as un-removable vertices. Therefore, the model is segmented into several parts due to the sharp edges. To simplify each part independently does not make the pre-defined features disappear.

If a vertex belongs to two sharp edges, the vertex is set to be removable with one of the two sharp edges. To remove a vertex which belongs to two sharp edges, the two sharp edges are made into one. Furthermore, to preserve the pre-defined features of the model, when removing the vertex between two sharp edges, we still calculate the weight of these two sharp edges, as described in the previous section. Therefore, only the sharp edges which are not significant are removed.

3.5.2. Selected removable edge testing

After pushing all of the removable edges into a priority queue, all of the edges in the priority queue are candidates for removal. Before removing the edges, it is necessary to do some tests to check if the removing process passes the following tests.

1. **Preserving mesh inversion**

   To remove an edge from meshes implies that the neighboring faces on the 1-ring of the endpoints of the edge are deformed. This action may cause the faces to fold over on each other. To avoid this type of mesh inversion, it is necessary to test the edge before removing it. When we get one removable edge from the priority queue, we compare the normal vector of each of the neighboring faces before and after removing. If the normal vector flips, this edge removing is not performed.

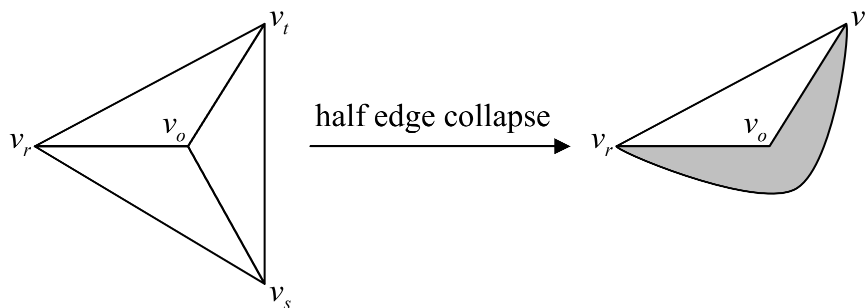


Figure 3-11: The example of mesh inversion.

As the example shown in Figure 3-11, once we get a removable edge $\overline{v_t v_s}$ from the priority queue, to remove this edge implies to remove the vertex v_s, and deform the triangle $\Delta v_o v_s v_r$ to be $\Delta v_o v_t v_r$, which is the inverse triangle of $\Delta v_o v_r v_t$, In this case, the mesh inversion is occurred.

2. **Preserving topology**

   To remove an edge from the meshes also implies that one or two of its adjacent faces are removed. This action may cause the vertex connection of the neighboring faces of the removing edge to change. Since our algorithm is working for 2-manifolds with boundary, it is necessary to test if this edge removing changed the topology of the meshes or not. When we get one removable edge, we check the neighborhood relationship of the neighboring faces of the removing faces before and after the edge removing. If it causes the topology to change, then this edge removing is not allowed.

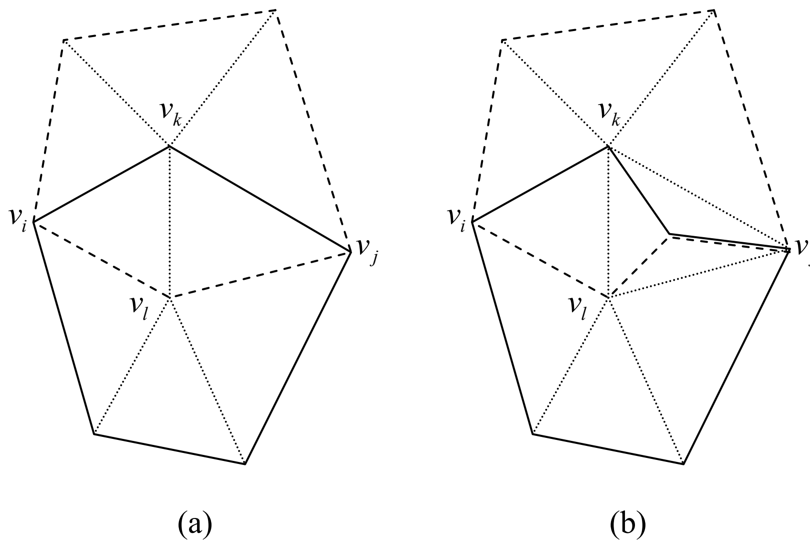


Figure 3-12: The example for (a) 2-manifolds and (b) non-2-manifold. The edges of the link of vertex v_l are shown as the solid lines and those of the link of vertex v_k are shown as the dashed lines.

On a 2-manifold, the link of each vertex is a circle. As the example

shown in Figure 3-12 (a), the two circles of the endpoints of edge $\{k,l\}$ intersect in two vertices v_i and v_j, and to remove such an edge could preserve the topological type. On the other hand, in the case shown in Figure 3-12 (b), the two circles of the endpoints of edge $\{k,l\}$ do not only intersect in two vertices v_i and v_j, but also another vertex and edge, to remove such an edge will pinch the manifold along a newly formed edge.

3. **Preserving model shape**

   Once the edge obtained from the priority queue passes the above tests, it is necessary to test which vertex could be removed if we used the half edge collapse operation to remove the edge, as will be described in Section 3.5.3. Hence, it is necessary to test which endpoint is the better one to remove when removing one edge as shown in Figure 3-13. If one of the endpoints is un-removable, i.e. it is also the endpoint of a sharp edge or a base edge, we remove the other one. If both of the endpoints of the edge are removable, we compare the deviation of the neighboring faces' normal vectors which are on the 1-ring of the endpoints, and then we remove the vertex which is located on the faces that are flatter than the faces located by the other endpoint.

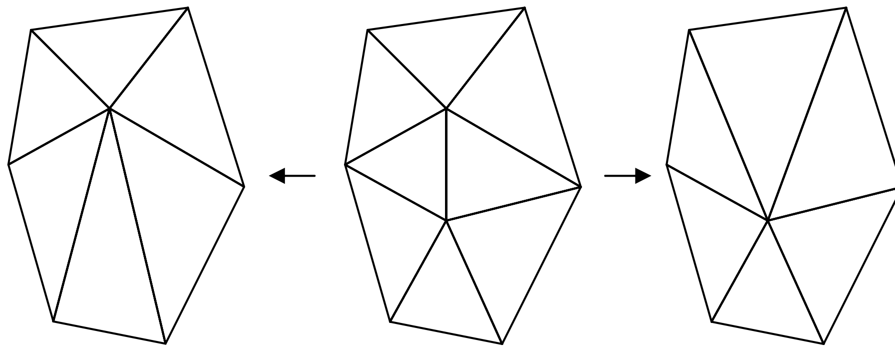


Figure 3-13: The example of choosing the endpoint for removing. When removing the edge located in the center, one of its endpoints is removed also. Here we remove the endpoint due to the deviation of its neighboring faces' normal vectors.

If the removable edge has passed all of the tests, then the half edge collapse operation is operated as described in the following section.

3.5.3. Half Edge Collapse

The "half edge collapse" operation as shown in Figure 3-14, is a special case of the "edge collapse" operation. If a vertex removed with a particular re-triangulation of the remaining hole when using the "vertex decimation" operation, the resulting mesh is also the same as the one after doing the half edge collapse operation.

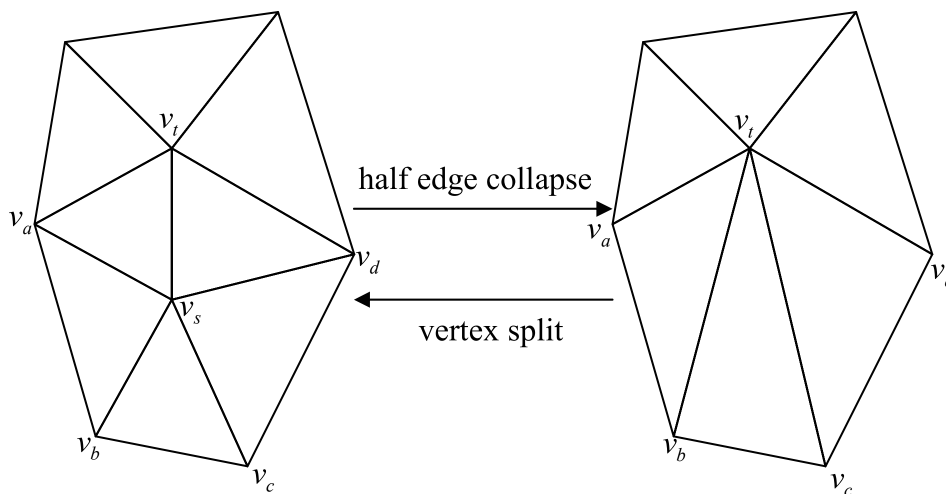


Figure 3-14: The examples of half edge collapse and vertex split operations.

The information used for reconstructing the original model is stored as a "patch". To minimize the data size of the patch which will be sent to the client side to reconstruct the original model and significantly affects the network transmission, we use the half edge collapse operation instead of using the edge collapse operation. Using the half edge collapse operation reduces the size of the patch compared to the edge collapse operation. This is because, when using the half edge collapse operation, there is only one vertex removed and no vertex is added into the meshes, but the edge collapse operation removes two vertices and adds one vertex.

Before doing the half edge collapse operation, it is necessary to store the information of the removing vertex for reconstructing the original model. Then, collapse the edge, remove one endpoint of the edge, and deform the faces associated with the removed vertex as shown in Figure 3-14. In Figure 3-14, the faces below the removed vertex v_s is deformed after collapsing edge $\overline{v_t v_s}$, i.e. combining vertex v_s with vertex v_t, and removing vertex v_s, face $\Delta v_t v_a v_s$, and face $\Delta v_t v_s v_t$.

Compare with the vertex split (Vsplit) data structure described in [37], since we detect the feature edges before doing the mesh simplification process, the size of our patch data structure listed in Figure 3-15 is only half of the size of the Vsplit data structure even when we use the edge collapse operation. For example, if we assume "int" is a 4 bytes primitive data type, 2 bytes are used for "short", and "float" is also using 4 bytes. The VertexAttribD and WedgeAttribD can be assumed to be 12 bytes data structures, since they both need a three-dimensional "float" array to store vertex position and normal vectors (texture mapping coordinates are ignored in this case). Then, the Vsplit data structure described in [37] needs approximately 60 bytes for storage if we use the binary mode to store it, but by using our approach, it just needs less than 30 bytes.

```
struct patch {
    int flclw;                  // a face in neighborhood of the patch
    struct {
        short vlr_rot:6;        // encoding to find another vertex
        short vs_index:2;       // index (0..2) within the patch
    } code;
    VertexAttribD vad_l;
    WedgeAttribD wad_l;
};
```

Figure 3-15: Patch data structure written as the vertex split data structure in [37] for comparison.

Because by using the half edge collapse operation, we still could get as good a simplified model as the one generated using the edge collapse operation. Therefore, in our algorithm, we still use the half edge collapse operation to minimize the size of the patch, although the difference in the patch's size is only a few as a result of using the two operations.

3.6. Mesh Reconstruction

The simplified model and the patches which have been stored when doing the 3D mesh simplification process are the components of the streaming mesh which will be uploaded onto the Web server and provided for the users to use at the client site. When using the streaming mesh on the Internet, the simplified model is sent first. After sending the simplified model, some patches are sent progressively with the QoS-like controlling, so that the users at the client site could get different 3D models with different number of faces due to the network bandwidth they are using. The "vertex slip" operation, as shown in Figure 3-14, are used to reconstruct the original 3D model without loss of data.

3.6.1. Vertex Split

If there are n steps for reconstructing the original 3D model from the simplified model, n patches are needed. Each of the patches contains the geometric position and attributes of one vertex which is removed from the model during the 3D mesh simplification process. As described in Section 3.5.3, when doing the mesh simplification from the meshes of step $i+1$ to the meshes of step i, the "half edge collapse" operation has been used. When doing the mesh reconstruction process from the meshes of step i to the meshes of step $i+1$, we use the "vertex split" operation on the other hand.

While doing the half edge collapse operation, we remove one vertex from the model, therefore one vertex is added into the model while doing the vertex split operation. Besides the geometric position and attributes of the vertex which will be added into the model, the other necessary information for the vertex split operation is used for finding the vertex which will be split. All of the necessary information contained in the patch is shown as the patch data structure listed in Figure 3-15.

When the meshes of step i becomes the meshes of step $i+1$, we first find out the vertex which will be split by using the transmitted patch as shown in Figure 3-16. Vertex v_t is such a vertex which will be split and the faces (gray triangles) which are below the edge $\overline{v_a v_t}$ and edge $\overline{v_t v_d}$ will be deformed. Hence, each patch also contains the triangle index of f_l (*flclw* in Figure 3-15), the vertex index of vertex v_t in face f_l (*vs_index*), and the number of triangles from the face f_l to face f_r (*vlr_rot*).

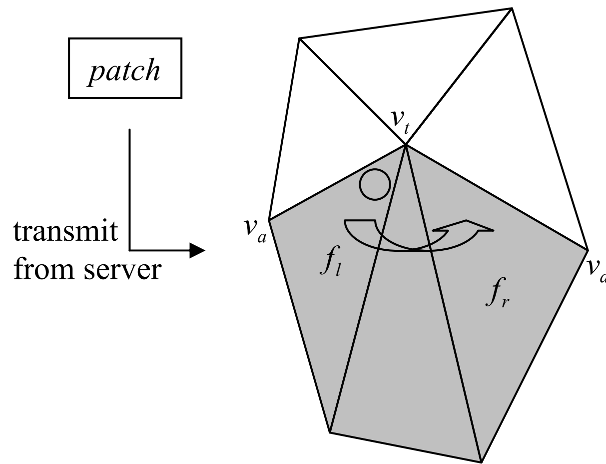


Figure 3-16: Using the transmitted patch to do the vertex split operation. The vertex which will be split is found by using the patch. Moreover, the triangles which will be deformed due to the vertex split operation will also be decided by using the same patch.

Therefore, as shown in Figure 3-16 we can use the triangle index of f_l and the vertex index to find out vertex v_t (the small circle), which will be split, and use the neighborhood information to find out all of the gray triangles from face f_l to face f_r, which will be deformed after doing the vertex split operation. After finding out vertex v_t, vertex v_a, and vertex v_d, we add a new vertex into the meshes according to *vad_l* and *wad_l* shown in Figure 3-15 as the geometric position and attributes of the newly added vertex, and deform the gray triangles shown in Figure 3-16. Then, the vertex split operation is done.

3.6.2. QoS-like Controlling

Although we have decoded a geometric 3D model with a streaming mesh which contains one simplified model and several patches. To transmit the streaming mesh through the Internet efficiently is still a problem, since the real network bandwidth between the server and the client is unknown and unstable. If the server delivers all of the patches in the same time, the users at the client side must still wait for downloading them as downloading the original 3D model. On the other hand, if the server sends only one patch at one time, the overhead of the network package's header and the synchronization between the server and the client will make the transmission rate worse as shown in Table 3-2. Hence, to make a flow control for monitoring the patches' transmissions is necessary. In our experiment, we use HTTP (Hypertext Transfer Protocol) [22] as the transmission protocol, since it could pass trough almost all kinds of the firewall limitations, although the synchronization between the server and the client costs a lot of time when making the connection.

A useful and interesting concept of QoS is to provide different qualities of media over the flexible network bandwidth. Here, we use the same concept to provide the geometric 3D models with different resolutions to the users at the client side, and the users will pay the same waiting time to get different progressive models with different number of faces due to the current network bandwidth.

The system hierarchy and the network communication diagram are shown in Figure 3-17. When the model provider creates a model encoded with streaming mesh method which contains a simplified model and several patches, all the provider has to do is to upload it onto the Web server and the users will use it via the inline Java applet embedded into a HTML (Hypertext Markup Language) file by a Web browser. The Web browser first sends a HTTP request to the Web server to download the Java applet which is the tool to display the streaming mesh. After the applet is running on the client machine, the applet will then send a HTTP request to the Web server for downloading the simplified model which is the first and base part of the streaming mesh, and calculate the effective network bandwidth between the server and the client as $W = S_{trans} / T_{trans}$, where W is the effective network

bandwidth and S_{trans} and T_{trans} are currently downloaded data size and transmission time.

Then, the applet will make a HTTP connection again with a Java servlet on the Web server to download proper number of patches according to the calculated network bandwidth. The number of patches is decided by $T_{wait} \times W / S_{patch}$, where T_{wait} is the waiting time set by the users and S_{patch} is the current patch data size. Because we encoded the patch by using gzipped[2] ASCII file format, the data size of a patch is not fixed. Hence, we have also to approximate the patch data size and the default value of S_{patch} is set to be the average value of the patch data size listed in Table 3-1.

The effective network bandwidth will be recalculated again while each connection defined as

$$W' = \frac{S'_{patch} \times N'_{patch}}{\frac{S_{patch} \times N_{patch}}{W} + T_{trans}},$$

where N_{patch} means the transmitted patch number and W' is the updated network bandwidth. $N'_{patch} = N_{patch} + N_{trans}$ is the updated transmitted patch number defined by previous N_{patch} and currently transmitted patch number N_{trans}. $S'_{patch} = S_{patch} \times N_{patch} + S_{trans} / N'_{patch}$ is the updated patch data size.

Therefore, the client applet could show the 3D model with proper data size due to the current network bandwidth. If the updated network bandwidth is much less or larger than the current network bandwidth, the network bandwidth is not updated, since the slowdown or speedup is due to the network noise.

Then, if the user needs to use the model with more details, the client applet will send a HTTP request again to the servlet running on the Web server again, and the servlet will transmit the patches to the client also according to

[2] "gzipped" means the file is compressed by "gzip" which is a well-known compressing method and program.

the calculated network bandwidth, so that the client applet could show the detail model progressively and recalculate the network bandwidth again. Finally, if the user really needs the original model, after the Java servlet at the server site sending all of the rest patches to the Java applet at the client site, the applet can reconstruct the original 3D model with no loss and no retransmission. Additionally, the Java servlet is established by using Sun Java 2 SDK, Enterprise Edition (J2EE) v 1.3.1 [71].

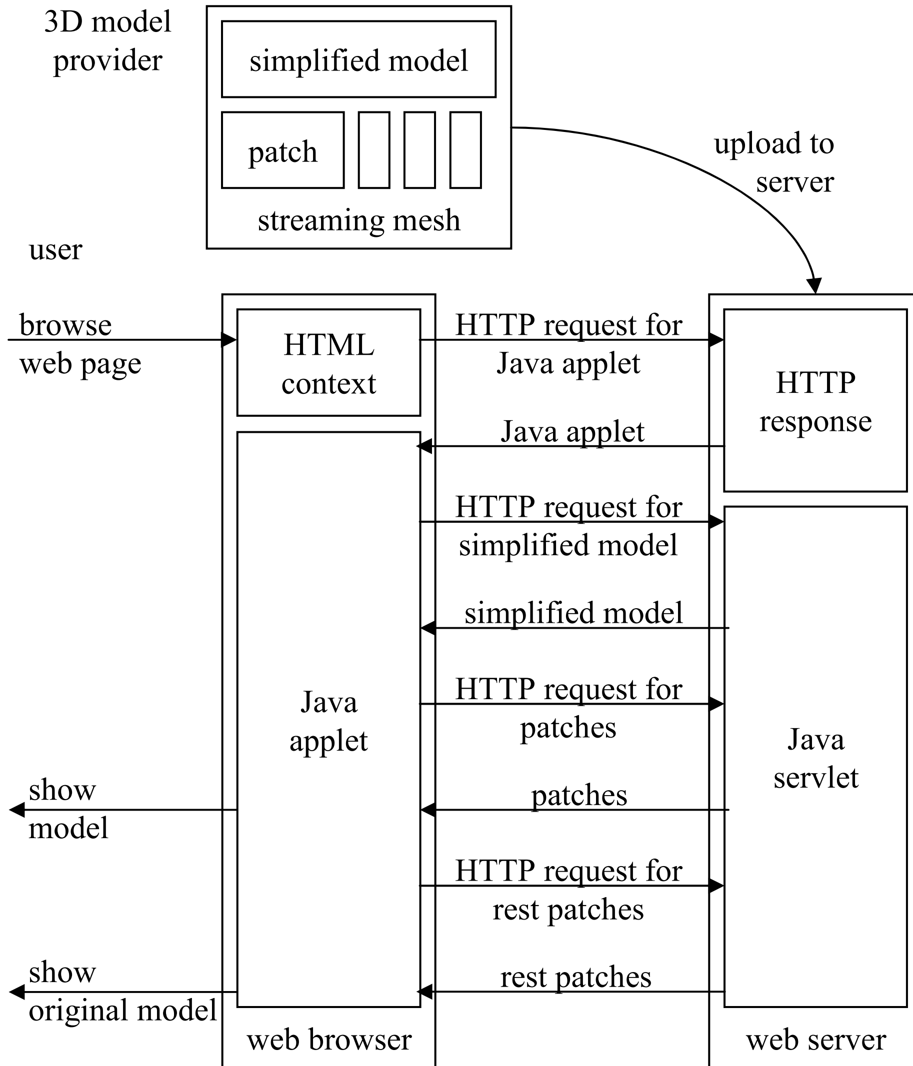


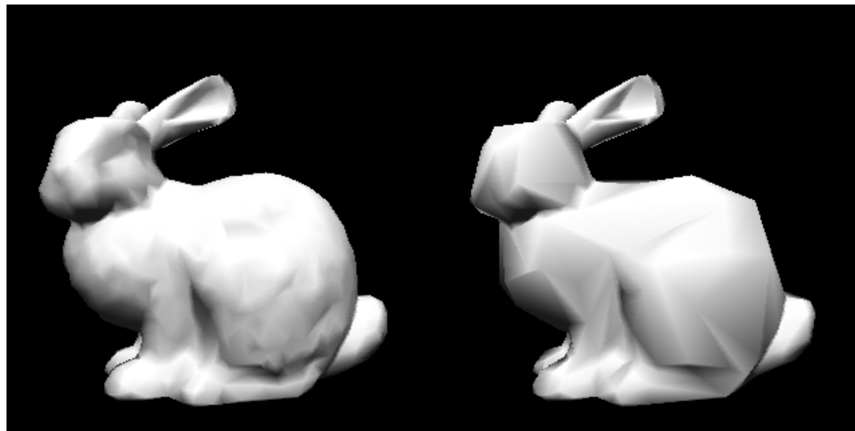
Figure 3-17: System hierarchy and network communication diagram.

3.7. Results

In this section, we will first show the evaluation of mesh simplification and reconstruction using several geometric 3D models with several conditions. Then, the comparison with the QEM and PM methods including the discussion of the algorithm differences is described. Finally, the effects and evaluation of the QoS-like controlling are shown in the last sub-section.

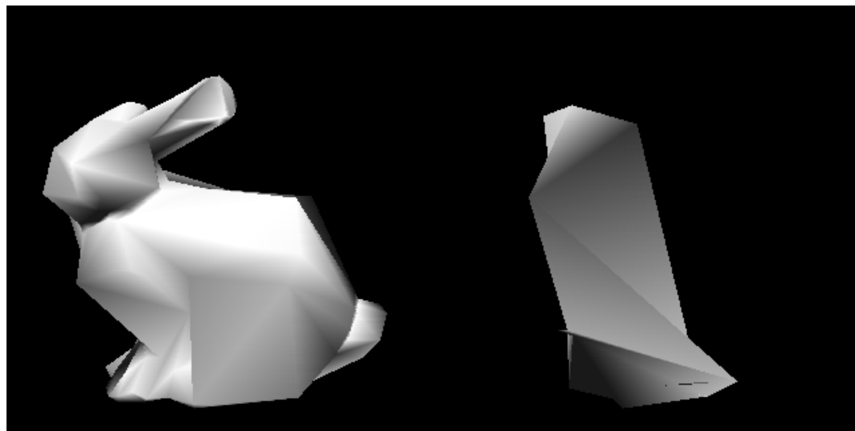
3.7.1. Evaluation of Mesh Simplification and Reconstruction

Figures 3-18 (b), (c), and 3-19 (a) show the simplified models, which is generated with different thresholds ε, of a 3D model "bunny" as shown in Figure 3-18 (a). Figure 3-18 (d) shows the simplified model when we ignore the base edge detection, since there are some boundary edges at the bottom of the model, which will be detected as the sharp edges. The shape of this part remains recognizable. Without detecting the base edges of the 3D model, the model may be over-simplified. As a result, the shape of the simplified model could not be recognized. This is a common problem of other previous methods. Figures 3-20 and 3-21 show comparisons of the original models and simplified results of other geometric 3D models. Even for the simplified models, the shapes and features could still be recognized easily. Since there are several sharp edges contained in the models shown in Figures 3-21 (a) and (c), we ignore the base edge detection during the 3D mesh simplification process. The number of faces of each model and the thresholds used for simplification are also shown in Figures 3-18, 3-19, 3-20, and 3-21.



original model
#f = 2,915
(a)

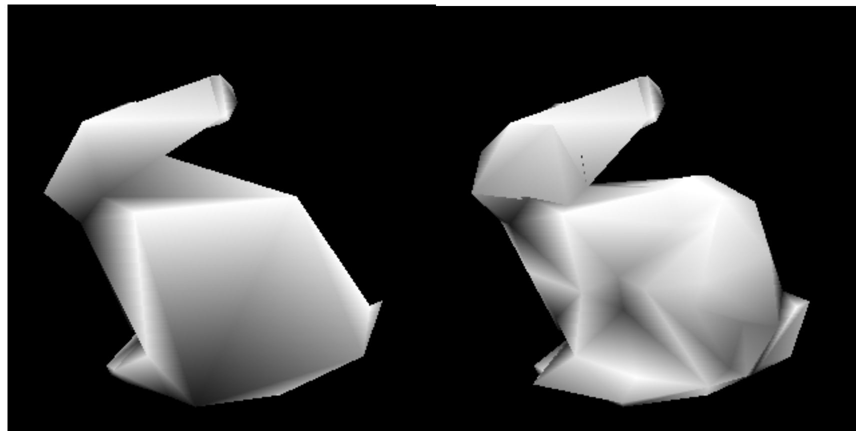
ε = 0.8: #f = 1,145
(b)



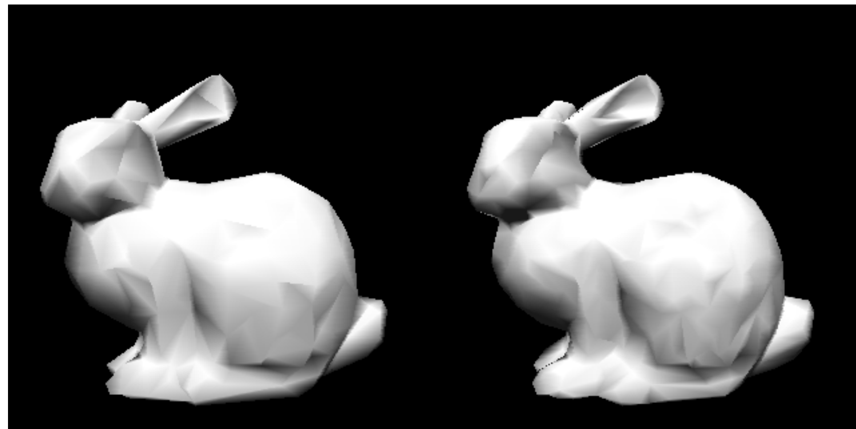
ε = 0.5: #f = 487
(c)

no base edge detection
#f = 103
(d)

Figure 3-18: Comparisons of (a) original bunny model, (b) (c) simplified models with different thresholds, and (d) simplified model without base edge detection.



ε = -0.2: #f = 185 #f = 419
(a) (b)



#f = 1,039 #f = 1,943
(c) (d)

Figure 3-19: (a) simplified bunny model, (b) ~ (d) reconstructed models from (a).

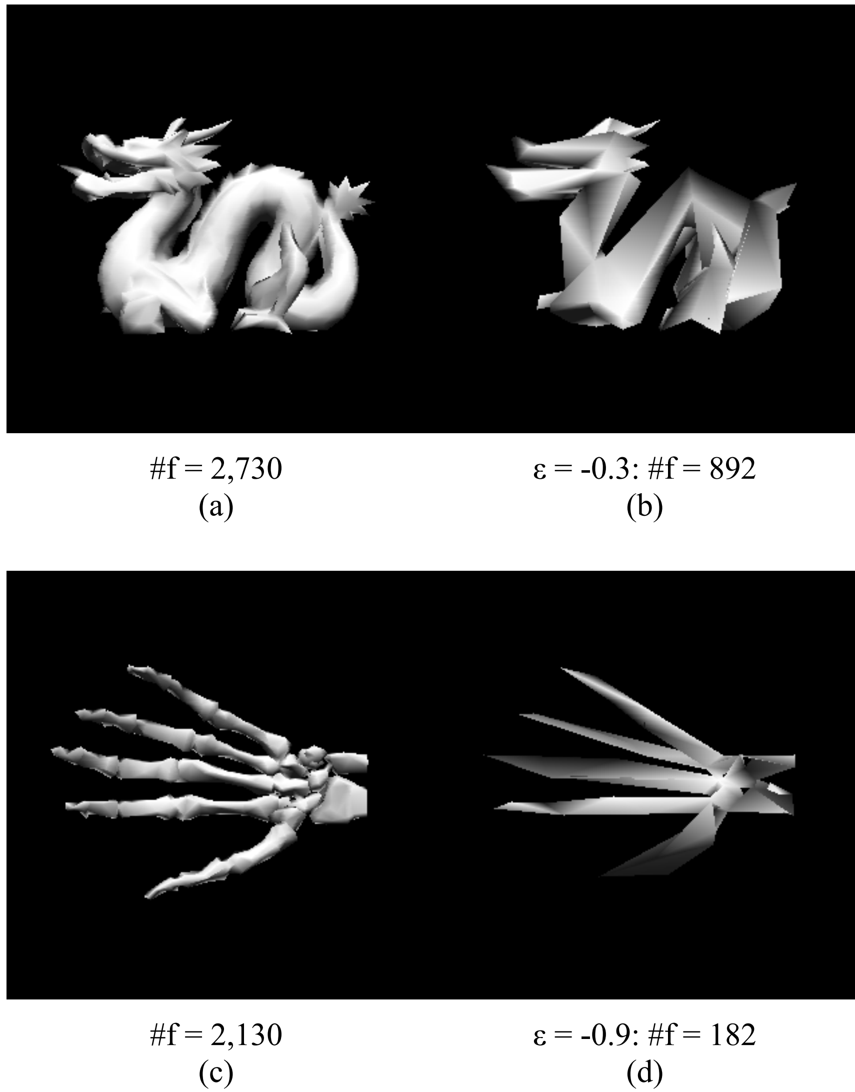


Figure 3-20: Comparisons of (a) (c) original models, dragon and hand (up to down), and (b) (d) their simplified results.

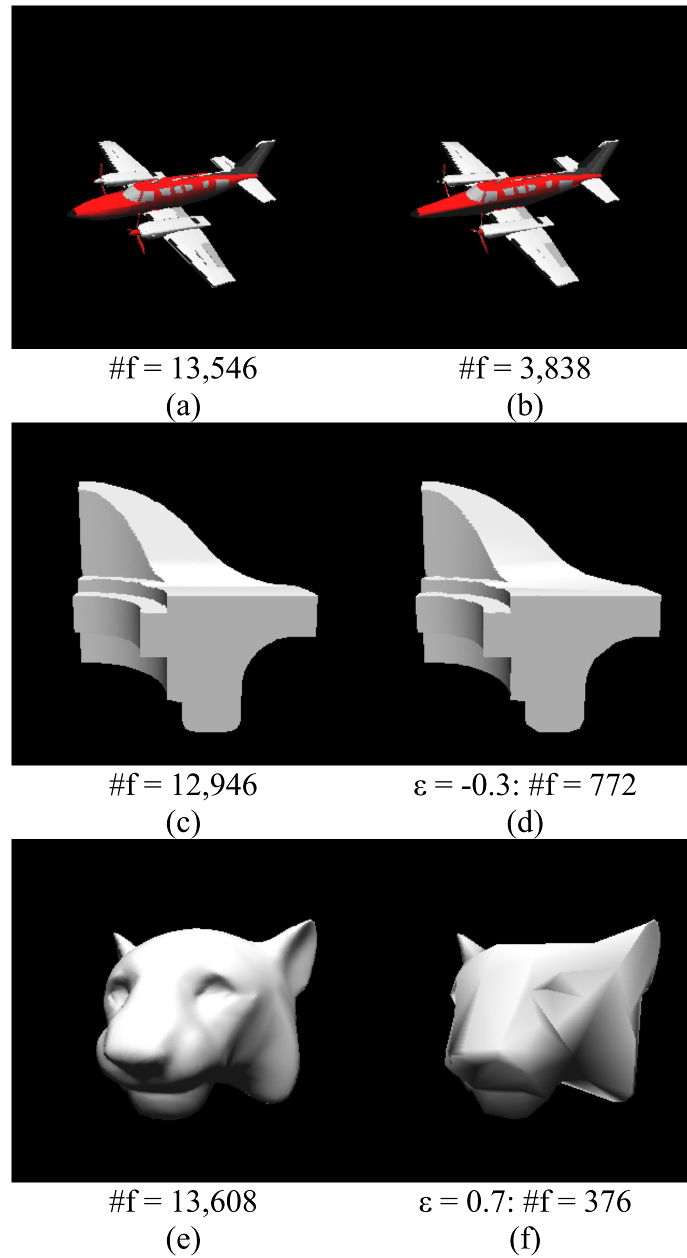


Figure 3-21: Comparisons of (a) (c) (e) original models, cessna, fandisk, and tiger (up to down), and (b) (d) (f) their simplified results, where cessna is a 3D model with different material properties and fandisk is a 3D model with pre-defined sharp edges.

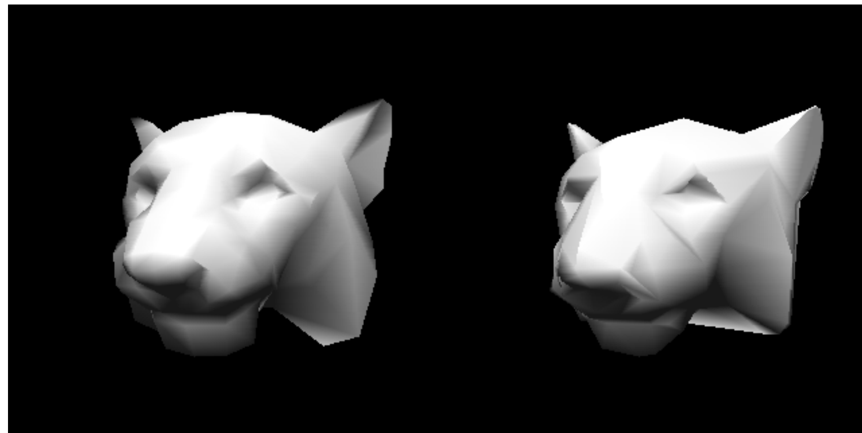
Table 3-1 lists the file sizes of the original models and the streaming mesh, which contains a simplified model and several patches, of different geometric 3D models shown in Figures 3-18 (a), 3-20 (a), (c), 3-21 (a), (c), and (e), respectively. The corresponding simplified models are shown in Figures 3-19 (a), 3-20 (b), (d), 3-21 (b), (d), and (f). The run-time performances required generating the simplified models and the average patch sizes for different 3D models are also shown in Table 3-1. The testing platform is a laptop PC with an Intel Mobile Pentium III 850MHz CPU (512MB memory, Microsoft Windows XP Professional), and the Java environment is Java 2 SDK, Standard Edition (J2SE) v 1.4.1_01. Since we wish to generate a simplified model with shape and features that are easily recognized, the compression rate of the model with several pre-defined features is worse than other models, for example the 3D model "cessna" shown in Figure 3-21 (a).

model	original model (bytes)	simplified model (bytes)	one patch (bytes)	time (ms)
bunny	75,142	5,746	34.424	1,111
dragon	65,586	18,137	35.939	961
hand	53,519	3,991	35.684	911
cessna	553,685	148,320	30.379	6,159
fandisk	295,092	35,765	29.083	15,001
tiger	347,587	9,099	34.925	14,971

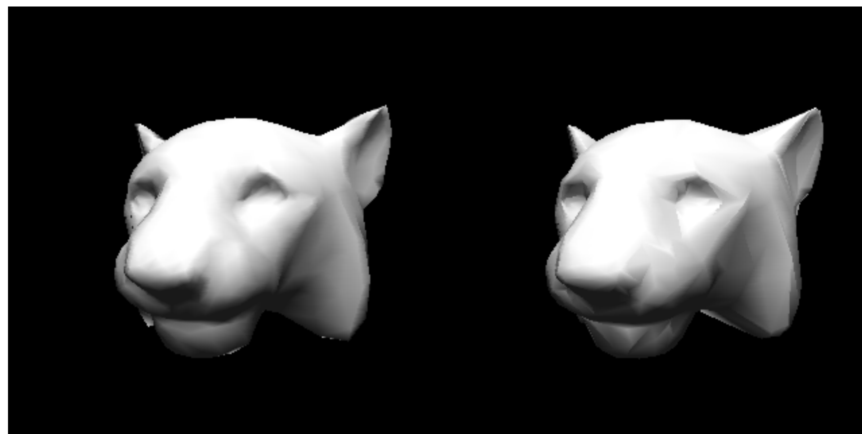
Table 3-1: Comparisons of file sizes and run-time performances.

For comparison, we have converted the file format of the original model to be the same as the simplified model (a gzipped ASCII file). The number of patches for reconstructing the original model from the simplified one is the difference between the vertex numbers of the original model and the simplified one, and approximately equals to half of the difference between the number of faces of the original model and the simplified one as shown in Figures 3-18, 3-19, 3-20, and 3-21. All of the patches are also stored as a gzipped ASCII file. Furthermore, since we use only pure Java programming language to develop all of the algorithms, it is possible to use our testing program via our Web site[3]. Moreover, the 3D graphics engine jGL is used which is also developed with pure Java.

[3] *http://nis-lab.is.s.u-tokyo.ac.jp/~robin/jSM/*

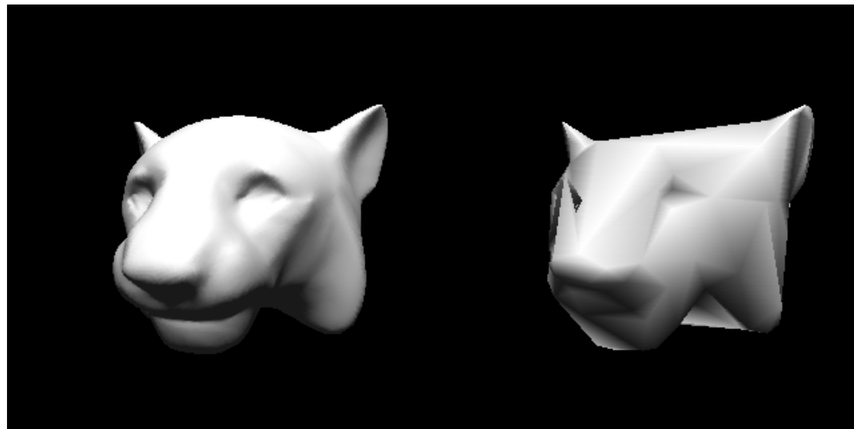


original model	reconstructed from Figure 3-21 (f)
#f = 504	#f = 504
(a)	(b)



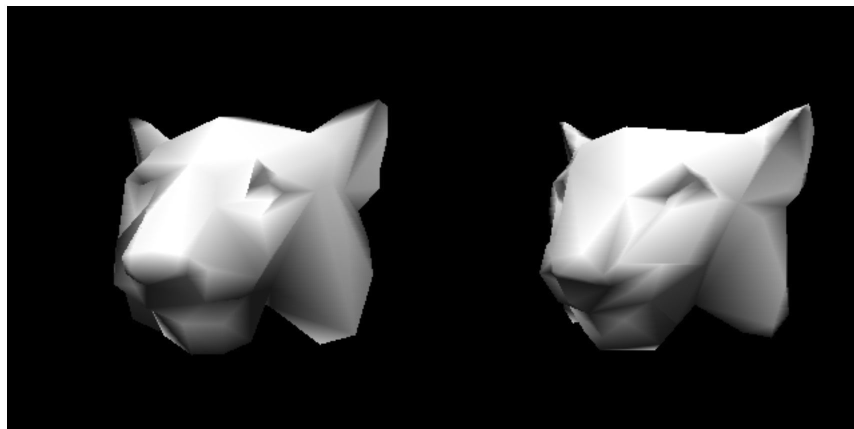
subdivided from (a)	reconstructed from Figure 3-21 (f)
#f = 1,512	#f = 1,512
(c)	(d)

Figure 3-22: Comparisons of (a) original tiger model, (c) subdivided model from (a), and (b) (d) reconstructed models from Figure 3-21 (f).



subdivided from Figure 3-21 (e)
#f = 40,824
(a)

simplified from (a)
ε = 0.7: #f = 348
(b)



simplified from Figure 3-22 (a)
ε = 0.7: #f = 250
(c)

simplified from Figure 3-22 (c)
ε = 0.7: #f = 352
(d)

Figure 3-23: Applying our method to the tiger models with different number of faces: (a) subdivided model from Figure 3-21 (e); (b) simplified model from (a); (c) (d) simplified models from Figures 3-22 (a) and (c) with the some condition as (b).

The model shown in Figure 3-21 (e) is generated from the model shown in Figure 3-22 (a) by applying the $\sqrt{3}$-subdivision algorithm provided by Kobbelt [47] three times. The model shown in Figure 3-22 (c) is the model generated by applying the same subdivision algorithm only once. If the models reconstructed from the simplified model shown in Figure 3-21 (f) have the same number of faces as the models shown in Figures 3-22 (a) and (c), the results shown in Figures 3-22 (b) and (d) are similar with them. These comparisons show that our algorithm could result in a well-reconstructed model even there are only a few patches applied to the simplified model.

To show our approach could be applied to the model with a large number of faces, we subdivide the model shown in Figure 3-21 (e) again to get the model shown in Figure 3-23 (a), which is equivalent of applying the $\sqrt{3}$-subdivision algorithm four times to the model shown in Figure 3-22 (a). Then, we simplify the model using our method with the same conditions as those used for generating the model shown in Figure 3-21 (f) to get the simplified model shown in Figure 3-23 (b). Comparing the simplified models shown in Figures 3-21 (f) and 3-23 (b), the features remained in them are similar. Furthermore, we also simplify the models shown in Figures 3-22 (a) and (c) with the same conditions to generate the simplified models shown in Figures 3-23 (c) and (d). The models shown in Figures 3-21 (e), 3-22 (a), (c), and 3-23 (a) are the same model, these simplified models show that our approach can get similar results even the scale of the number faces is not the same. This implies that the simple base edge detection is enough for preventing the features of the original model, and further global feature detections are not necessary even the model contains several small faces.

3.7.2. Comparisons with Previous Works

For comparing the models reconstructed from the simplified model with the original one, we use the simplified model shown in Figure 3-19 (a), which is generated by our method with $\varepsilon = -0.2$, and its original model is shown in Figure 3-18 (a). The results are as the curves (a) shown in Figures 3-24 and 3-26. In order to compare with other previous methods, we also used the QEM method to simplify the same model, and made the same

comparisons as those of our approach. The results are as the curves (b) shown in Figures 3-24 and 3-26.

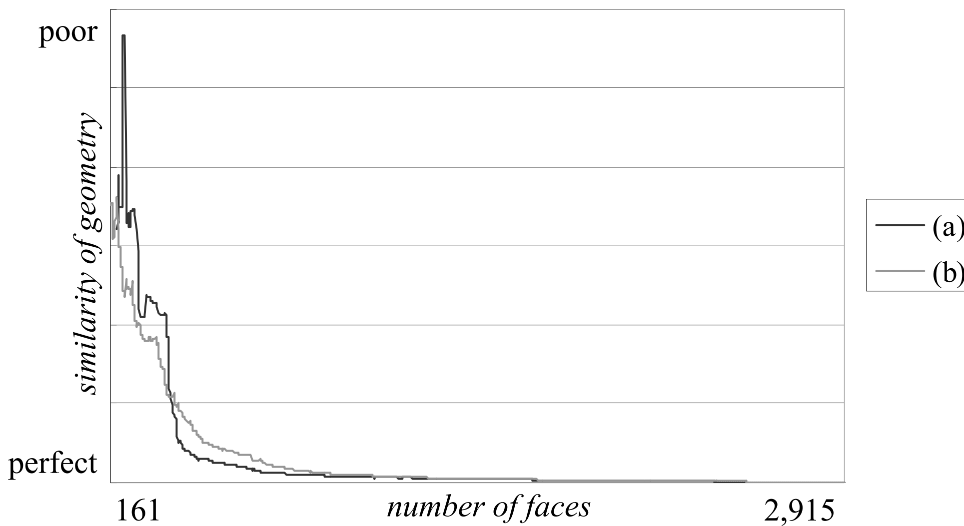


Figure 3-24: Similarity of geometric approximation of 3D model "bunny" for (a) our approach and (b) simplified with the QEM method.

Each curve of Figure 3-24 shows the similarity of the geometrical approximation of each reconstructed model and the original one. The measurement used in Figure 3-24 is based on the L_2 norm: the average squared distance from the vertices of the original model to the surface of the reconstructed one used to compare the difference between the two models. Each curve of Figure 3-26 shows the comparisons for similarity in appearance of each reconstructed models and the original one. To compare the similarity of appearance, we compute the differences between the rendered images as shown in Figure 3-25 of the reconstructed models and the original one by setting the camera at 6 different positions as the method described in [54].

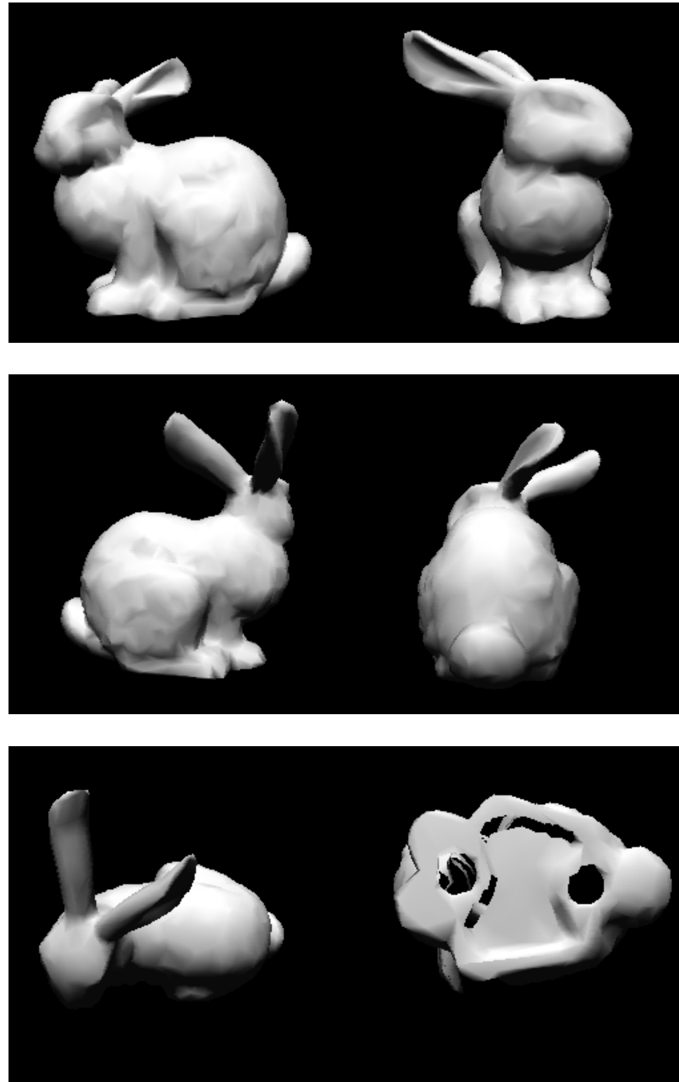


Figure 3-25: Images from different viewpoints for comparing the similarity of appearance. Each image is captured by setting the camera at different positions. The image at the right-bottom shows several “holes”, which are the features of the original “bunny” model.

Using the QEM method, we could get a simplified model that contains less faces (number of faces is 161 in this case) than using our approach, but the qualities of the corresponding models reconstructed from each simplified one are almost the same. However, to generate the simplified model using the QEM method is much more time-consuming than using our method. For example to simply the 3D model “bunny” shown in Figure 3-18 (a) took more than 4.7 seconds by using the QEM method, but it only took less than 1.2 seconds by using our approach as shown in Table 3-1. Moreover, the data size of the patch used for reconstructing the original model in our method is much smaller than the similar data structure used in other previous methods.

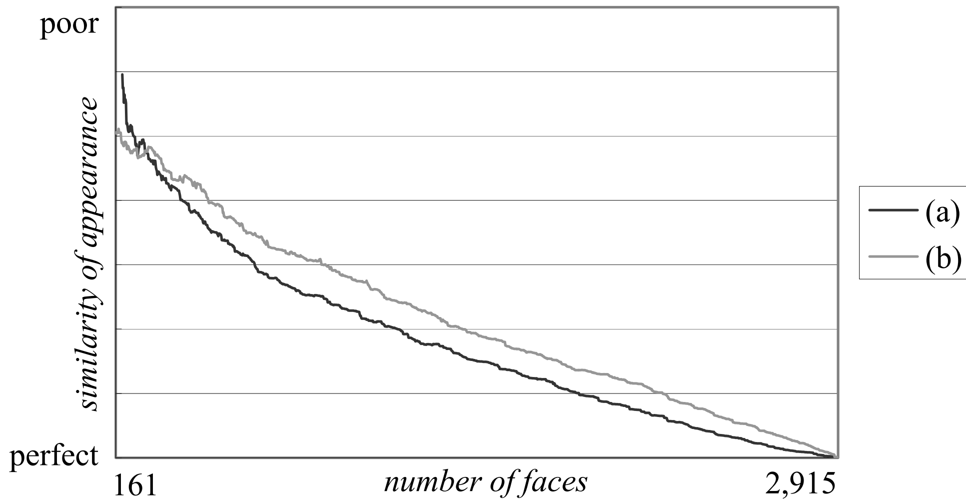


Figure 3-26: Similarity of appearance of 3D model “bunny” for (a) our approach and (b) simplified with the QEM method.

We also use the 3D model “tiger” shown in Figures 3-21 (e) and (f) to compare the similarity of appearance, since the model “tiger” only contains unstructured meshes. The result is as the curve (a) shown in Figure 3-27. The curve (b) of Figure 3-27 is the result of using the QEM method to simplify the same model. Since the model “tiger” contains a lot of faces, refer to Table 3-1 it took less than 15 seconds to get the simplified model by using our approach. However, by using the QEM method, it took more than 73 seconds, although it could get a simplified model that contains less faces (number of faces is 78 in this case). The curves of Figure 3-27 show that our approach can get the reconstructed models which qualities are as good as

those of the QEM method.

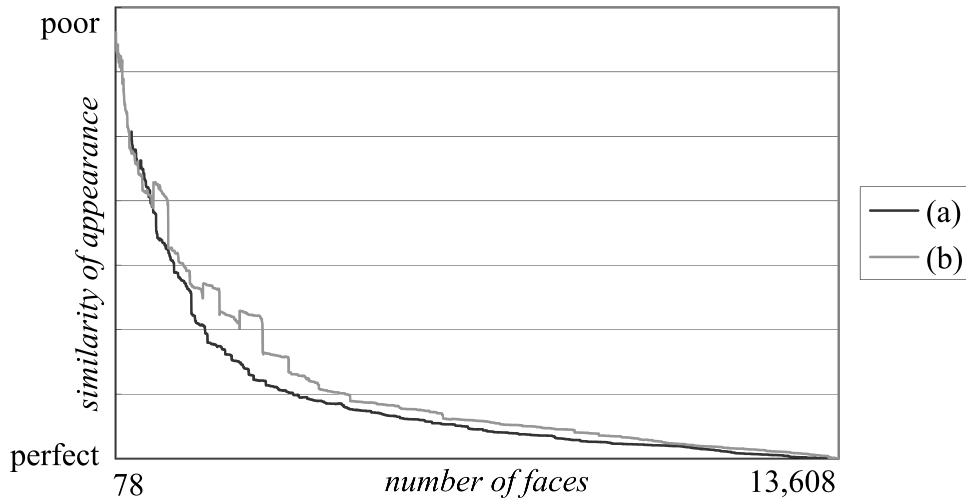


Figure 3-27: Similarity of appearance of 3D model “tiger” for (a) our approach and (b) simplified with the QEM method.

To use the priority queue for sorting the removable edges is a common science for 3D mesh simplification algorithm. The QEM and PM methods are also using the same mechanism as our approach. However, using the priority queue implies to get the local optimization for the processing efficiency. Without some necessary constraints, the previous mesh simplification algorithms let the simplified model to be over-simplified, because some edges with lower priority are removed earlier than some ones with higher priority due to the simplification loops. Since our approach uses the constraints formed as sharp edges and base edges, although we also use the priority queue to get the local optimization as others, the edges with the priority lower than the threshold are not removed.

The experiences show that our approach is much faster than the QEM method, which is also much faster than the PM method. One of the reasons about the run-time performance efficiency is because of the half edge collapse operation. To use the original edge collapse operation as the QEM and PM methods, it is necessary to arrange the position for the newly generated vertex due to the edge collapse operation. This problem can be ignored by replacing the edge collapse operation used by the QEM and PM methods to the half edge collapse operation. To fit the newly generated vertex to one of

the endpoints of the removed edge (same as half edge collapse) or the mid-point of the edge is also a good solution. However, even we use the above solution, the performances of the QEM and PM methods are also worse than our method.

Another reason about the run-time performance efficiency is because of the feature detection process. Without this process, all of the removable edges will be added into the priority queue with their weights even the edge is not necessary to be removed. In our method, we prevented this by using the sharp edge and the base edge concepts, hence in each mesh simplification loop, only qualified removable edges are added into the priority queue. Therefore, in our method, it is not necessary to sort too many edges in the priority queue and the number of items in the priority queue is also not too large in each iterative loop.

The main reason for the run-time performance efficiency is about the heuristic function using to weight the removable edges. The heuristic function used in our approach is based on the ESOD operation which is only an inner dot of two normal vectors. On the other hand, the heuristic function used by the QEM method is based on a complex energy function, and so is the PM method. To calculate the heuristic function for each removable edge takes a lot of time, so that the performance of our approach is much better than the QEM and PM methods.

3.7.3. Evaluation of QoS-like Controlling

Figure 3-28 shows the network layout of our testing environment for the QoS-like mechanism. For testing the performances for different network environments, we use the same laptop PC with an Intel Mobile Pentium III 850MHz CPU (512MB memory, Microsoft Windows XP Professional). The Web server is Apache HTTP Server 1.3.26 with Tomcat 3.2.3 running on a Sun Ultra 10 Workstation with a Sun UltraSPARC IIi 360MHz CPU (256MB memory, Sun Solaris 9). The Java environment is J2SE v 1.4.1_01 and J2EE v 1.3.1. For comparing, we use 5 types of network connections. 100Base-TX (IEEE 802.3u) and 10Base-T (IEEE 802.3) Ethernets are typical local area network (LAN) connections which could provide maximum 100Mbps (bits per second) and 10Mbps connections. The wireless station

provides IEEE 802.11b standard for 11Mbps connection. PHS (Personal Handyphone System) is a common mobile data communication method in Japan with 64Kbps. Finally, we also use a specific mobile data communication kit called “b-mobile” provided by an ISP (Internet Service Provider), although it is also a kind of PHS, its maximum communication speed is 32Kbps, and the heavy Internet traffic jam decays the speed to be very slow.

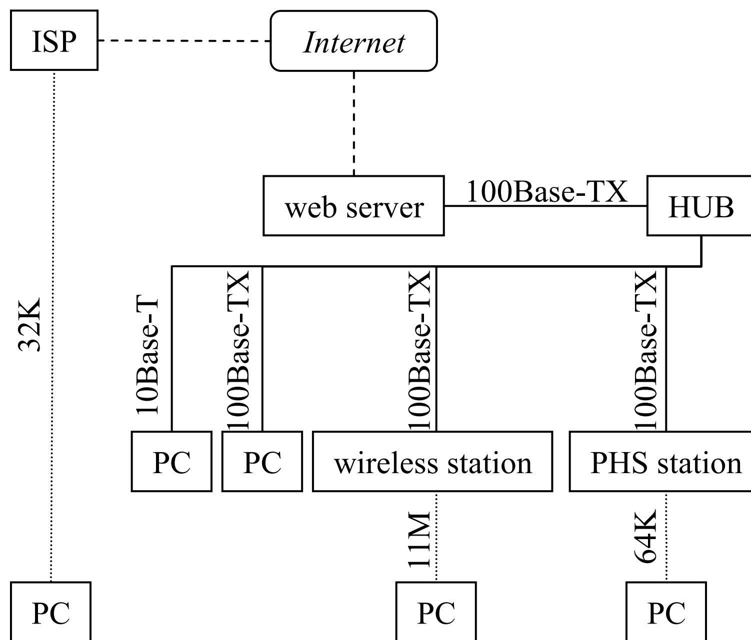


Figure 3-28: Network layout of testing environment.

The transmitting result for downloading the original and simplified modes of the 3D model “bunny” is shown in Table 3-2. The difference of downloading all of the patches at once and the sum of the time for transmitting each patch separately could also be known obviously. Since the latter one needs additional overhead for network package’s header, compressed file’s header, and the synchronization between the server and the client, the file size and the transmission rate is much worse than the other one. This is also a proving for the necessary of the QoS-like mechanism. To transmit each separately by using PHS (64K) or “b-mobile” (32K) took too much time, so we ignored these tests.

type of network connection	time (ms)			
	original model	simplified model	all patches at once	one patch at once
100Base-TX	271	30	270	211,670
10Base-T	418	30	470	211,628
wireless (11M)	361	40	351	201,925
PHS (64K)	10,875	1,052	7,291	
b-mobile (32K)	15,973	2,003	10,695	
file size (bytes)	74,695	5,808	48,237	146,967

Table 3-2: Comparisons of transmitting performances with different types of network connections by using 3D model "bunny".

The performance which includes the transmission and reconstruction for 3D model "bunny" by using PHS (64K) is shown in Table 3-3. The performances for getting the reconstructed models are also listed in it. The time for showing the simplified model is much less than showing the original one. The time for getting the reconstructed models is counted from getting the simplified model and includes the time for transmitting the data and reconstructing the model. To show the reconstructed model as shown in Figure 3-19 (c) does take less than half of the time to show the original one, and the differences between the two models are hardly recognized.

model	#faces	time (ms)
original model	2,915	10,875
simplified model	185	1,052
reconstructed models	419	2,814
	1,039	4,717
	1,943	6,740
	2,915	8,402

Table 3-3: Comparisons of transmitting and reconstructing performances of 3D model "bunny" with PHS (64K).

Table 3-4 shows the experimented results of the QoS-like controlling. When transmitting the 3D model "bunny" with different network connec-

tions, the client side receives difference 3D models with difference resolutions. If we set the waiting time to be around 1 second, the user will get the original model at once when using 100Base-TX and wireless (11M), since to get the whole data needs much less than the waiting time. However, the users using PHS (64K) and "b-mobile" (32K) could only see the simplified one, because even only downloading the simplified model it takes more than the waiting time. While using 10Base-T, the user receives the proper resolution of the 3D model much better than the PHS and "b-mobile" users but worse than the 100Base-TX and wireless (11M) users.

type of network connection	transmitted 3D models	
	#faces	time (ms)
100Base-TX	2,915	300
10Base-T	1,027	1,152
wireless (11M)	2,915	391
PHS (64K)	185	1,052
b-mobile (32K)	185	2,003

Table 3-4: The transmitted 3D models' resolution comparisons of 3D model "bunny" with different types of network connections.

3.8. Summary

This Chapter presents a multiresolution streaming mesh for Internet transmission with a QoS-like controlling. While transmitting the streaming mesh with our system, the server first delivers a simplified model with the data size according to the current network bandwidth. If the user at the client site needs to get more details, by transmitting some necessary patches, the system could show the progressively increasing model detail and finally the original model could be reconstructed without losses. The simplified model of the streaming mesh is generated by an efficient method of 3D mesh simplification for geometric 3D models. Our mesh simplification

method is to obtain an adequate simplified model in a short amount of time while keeping the data size small and preserving the shape and features of the original model. Therefore, the model provider is able to check the simplified model on the Web while changing some parameters for simplification to guarantee the shape of the simplified model before uploading to the Web server.

Chapter 4

VRML and Solid Texturing Supports

4.1. Introduction

Besides the 3D graphics library, jGL, and streaming mesh, we also provide some other projects for Web Graphics. Two of them are described in this Chapter. One is jVL, a VRML (Virtual Reality Modeling Language) library as a VRML support for jGL, and the other is an adaptive method for showing procedure solid texturing to support the applications running for Web Graphics. It is also an extension of jGL.

4.2. jVL – a VRML Support for jGL

jVL is a VRML library for Java [2] [30] and also a VRML support for jGL as its extension. To provide a programming environment for Web Graphics, besides a good 3D graphics library, a library for displaying 3D models is also important and necessary. Therefore, also to test the capabilities of jGL, we follow the specification of VRML [6] to develop a VRML library using jGL called jVL, since VRML is a well-known standard for representing 3D models and scenes for Web Graphics and used by many people. Although there are some tools for showing VRML objects or scenes as described in Section 2.2, we still need a library to support more interactive mechanisms and to be programmable. This is facilitated if the VRML library is implemented on top of a powerful API (Application Programming Interface).

To provide such a library by using pure Java programming language is not very easy, but fortunately we have jGL to be our 3D graphics engine and programming with jGL is almost the same as with OpenGL [70]. Since VRML itself is object oriented, we can follow the development policies of jGL to design jVL by using Java, but the run-time performance and byte-code size are still huge problems of it.

4.2.1. System Hierarchy

The development of jVL is following the specification of VRML. There are two main parts, which are nodes and fields, in the VRML specification. The 3D models or scenes are associated with several nodes with a tree structure, and the parameters of the nodes are stored in some fields. Since VRML is object oriented, we can also use the same performance enhancement methods as described in Section 2.7. Therefore, we use class inheritance to organize fields in a class tree. For nodes, since there are several differences between the nodes, we classify all of the nodes into six main categories, and four sub-categories, and also organize them in a class tree.

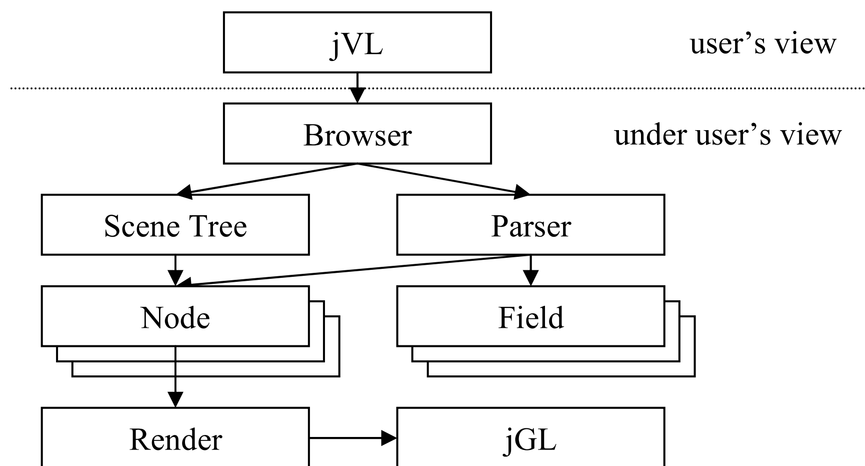


Figure 4-1: The system hierarchy of jVL. The Node and Field are two main parts in the VRML specification, and jVL is the interface provided for the programmers.

The system hierarchy of jVL is shown in Figure 4-1, and the Node and Field are the two main parts for constructing 3D models and scenes, and jVL is only an interface for programmers. We isolate the rendering routines (the Render in Figure 4-1) as a stand-alone class located between jVL and jGL, hence we can enhance the rendering performance by optimizing this one class. The Browser and Parser as its class name are used for all of the browser and parser functions. The Parser will be called only when loading the VRML file, and is used to parse the VRML file format and store all of the information into the Fields of Nodes with a tree structure. When rendering a 3D model or scene, jGL is used to generate the image.

4.2.2. Performance Enhancement Issues

From the experiences of developing jGL, we also utilize class inheritance and function-overriding to minimize the byte-code size and enhance the run-time performance. Besides these, we also use the following policies.

1. **Use display list mechanism of jGL**

   In OpenGL, display lists are used to store rendering commands, so that programmers could pre-calculate necessary functions and store the results with rendering commands into display lists to enhance the rendering performance. Because jGL has the same mechanism as OpenGL, we utilize display list of jGL in the same way.

2. **Pre-process the constant parameters**

   Since almost all of the attributes of the nodes will not be changed while re-drawing, we pre-process or pre-calculate all of the static information and store it in the nodes to enhance the rendering performance.

3. **Combine arbitrary geometric mesh data**

   Arbitrary geometric mesh data is more important than other nodes, since there are only few primitive geometric nodes supported by VRML. Most people prefer to use arbitrary geometric meshes instead of well-defined primitive geometric nodes. Therefore, it is important to display arbitrary geometric mesh data efficiently.

In VRML, to show arbitrary geometric mesh data with different material properties, we must use several Shape nodes with IndexedFaceSet nodes. Hence, there will be a huge branch in the 3D scene tree. Therefore, we combine such nodes to be just one node.

4.2.3. Result

As of this writing, we have implemented more than 70% of all VRML nodes in jVL. Besides the nodes, route and event transmission mechanisms have also been implemented. All of the implemented functionalities in jVL are listed in Appendix A.2.

rendering time	platform
10 ms = 100 fps	Intel Mobile Pentium III 850MHz, 512 MB memory, Microsoft Windows XP Professional
47 ms = 21.3 fps	Sun UltraSPARC IIi 360MHz, 256MB memory, Sun Solaris 9

Table 4-1: The performance testing of jVL on different platforms by using a simple table with variable colors for the legs and top as shown in Figure 4-2.

To evaluate the run-time performance of jVL and also to demonstrate that it can read 3D models described as VRML files, we use a simple table with variable colors for the legs and top might be prototyped, which is selected from the ISO/IEC 14772-1:1997 Virtual Reality Modeling Language (code from Section D.4, pages 211-213, Figure D.3), the specification of VRML. This model contains 204 polygons and is shown in Figure 4-2. The performance of the test file was measured on both a Sun Ultra 10 Workstation with a Sun UltraSPARC IIi 360MHz CPU (256MB memory, Sun Solaris 9) and a laptop PC with an Intel Mobile Pentium III 850MHz CPU (512MB memory, Microsoft Windows XP Professional). The results are listed in Table 4-1.

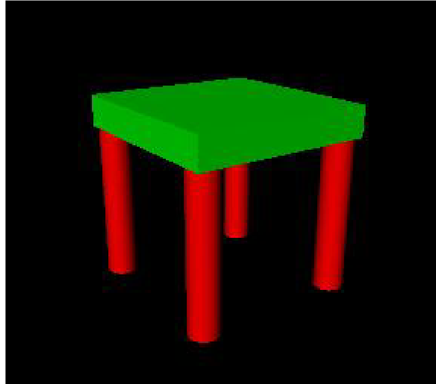


Figure 4-2: A simple table with variable colors for the legs and top might be prototyped is rendered to measure the performance. This program is an example in the ISO/IEC 14772-1:1997 Virtual Reality Modeling Language (code from Section D.4, pages 211-213, Figure D.3).

cube number	rendering time
100	30 ms
400	90 ms
900	190 ms
1,600	320 ms
2,500	490 ms
3,600	691 ms
4,900	931 ms
6,400	1,202 ms
8,100	1,512 ms
10,000	1,853 ms

Table 4-2: The performance testing of jVL for evaluating the rendering time according to model data size by using several rotating cubes, where each cube is drawn with different color values as shown in Figure 2-9 (a).

Since the performance could not be estimated by using a simple model, we use more than 100 rotating cubes drawn with different color values as shown in Figure 2-9 (b), which is the example of only 100 rotating cubes. Table 4-2 lists the performance of evaluating the rendering time according to the model data size on the laptop PC as Table 4-1.

Figure 4-3 shows the source code for displaying a VRML file by using jVL and jGL. This is just a simple example to show the usage of jVL, to change or control the objects, lights, and sensors included in the VRML file is also possible.

```
import jvl.VL;
import jgl.GLApplet;

public class viewer extends GLApplet {

    VL myVL = new VL (myGL);

public void display () {
    myVL.vlRenderWorld ();
}

public void init () {
    myUT.glutInitWindowSize (500, 500);
    myUT.glutInitWindowPosition (0, 0);
    myUT.glutCreateWindow (this);
    myVL.vlSetWorldURL (getDocumentBase(), “exampled.4.wrz”);
    myVL.vlInit ();
    myVL.vlViewpoint (getSize ().width, getSize ().height);
    myUT.glutDisplayFunc ("display");
    myUT.glutMainLoop ();
}

}
```

Figure 4-3: The source code using jVL for displaying a VRML file.

4.3. Adaptive Solid Texturing for Web Graphics

Rendering a 3D model with procedural solid texturing [19] [61] [62] is a useful method of showing a 3D model realistically. Obviously, a 3D model rendered with procedural solid texturing offers highly visual effects rather than one that is rendered with texture mapping, since the procedural solid texturing uses three-dimensional coordinates as the parameters to represent the appearance of the shape of the model, but texture mapping only maps a two-dimensional texture image onto the surface of it. Hence, procedural solid texturing has none of the distortion and discontinuity problems associated with texture mapping. Procedural solid texturing generally calculates the corresponding texture data "on the fly", so that it can offer highly visual effects of the model that is being rendered. Unfortunately, to calculate the corresponding texture data on the fly is time-consuming, since the texturing function could involve a lot of computational time in order to present a fancy textured appearance.

Many people wish to pre-generate a 3D-texture image by using a texturing function in the beginning so that they can use 3D-texture mapping functions in graphics libraries, such as OpenGL, in order to get the benefits available from hardware accelerators. However, if the resolution of the 3D-texture image is low, there is an artifact problem; i.e. every pixel on the 3D model surface looks like a lattice. If we increase the resolution of the image, the storage requirement becomes a big problem to handle, since the size of the image grows as a cubic function and is difficult to load into the memory space in a machine or on a graphics accelerator. Moreover, for a procedural solid texturing program running on the Internet, it is also difficult to download a huge 3D-texture image if there is already a pre-generated one on the server. Furthermore, to generate a 3D-texture image before rendering is also not a good solution, since it is also time-consuming according to the resolution of the image and the complexity of the texturing function, although most of the texture data is never used.

4.3.1. Cache Technology for Solid Texturing

Utilizing the cache technology to store some information for re-using is a common hardware technique in computer science. For procedural solid texturing, people may use a lookup table to store a sequence of random numbers rather than generating them on the fly to enhance the performance [51]. Some hardware providers have also noticed that the cache mechanism could be used to accelerate rendering with procedural solid texturing. However, there are several problems to be overcome if people want to simulate this hardware technology by using only software programming. The first and main problem is the memory storage requirement. Before explaining this, the cache access conception of our approach is first introduced.

4.3.2. Cache Cube Conception

Basically, in order to cache the calculated texture data of a texturing function in memory, we use a "cache cube" just like other 3D-texture mapping methods require. For some specific 3D models, the use of a specific shape which could cover the model would be tighter than using a cube, but since we wish to offer a general solution for procedural solid texturing, a cube is the most suitable shape for all categories of 3D models. Because the texture data at unused portion is not stored and computed, to use a cube does not waste the memory and time. Moreover, the coordinates of the cache cube are defined in the object coordinates of the model, which is going to be rendered, so that even if the model is transformed, the cache cube could offer the exact texture data corresponding to it.

Initially, the cache cube is empty, since there is no texture data has been calculated. When rendering a 3D model with procedural solid texturing, the system checks the cache cube first to see if there is already any corresponding texture data for the drawing pixel, which is contained in a voxel, as shown in Figure 4-4. If the voxel does exist, the pixel is rendered using the existing voxel, but otherwise the desired texture data is calculated by using the texturing function and stored into the cache cube at its corresponding position as a new voxel, then the pixel is then rendered with the calculated

data. Additionally, here we use just four bytes to store the voxel as an RGBA (red, green, blue, and alpha) structure, because Java uses a four-bytes integer to store the color value. Since the texture data has no alpha value, we could use this byte to be the flag to check if the corresponding voxel exists or not, although just one bit is sufficient.

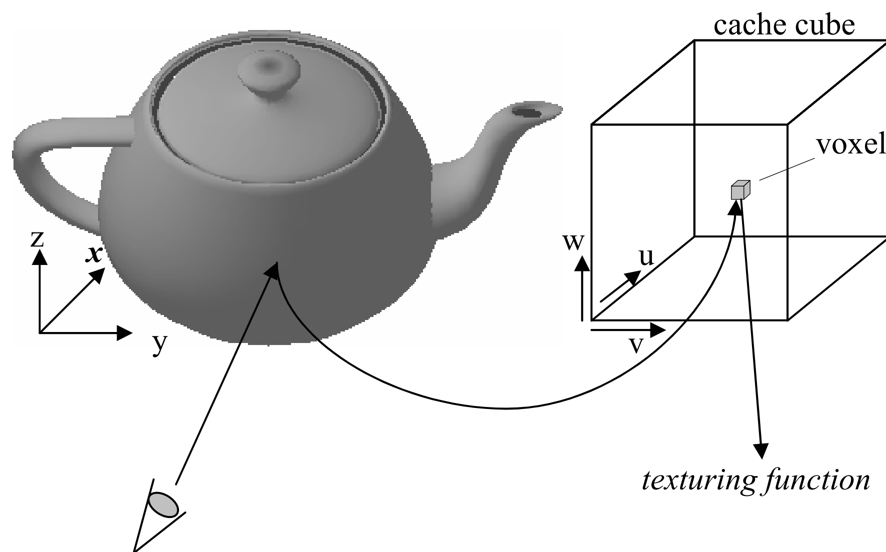


Figure 4-4: System diagram of cache cube conception. When rendering an object using our system for procedural solid texturing, the system first checks if the required voxel is already in the cache cube. If the voxel exists, it simply draws the pixel with the existing voxel; otherwise it calculates the voxel by using the texturing function, stores it in the cache cube, and then draws the pixel.

Once the texture data is calculated and stored in the cache cube, it is not necessary to re-calculate again when re-drawing. Only the portion that has never been seen needs to be calculated; i.e. only voxels that do not exist in the cache cube need to be generated on the fly. Since the required texture data is calculated on demand, it is not necessary to generate a cache cube before the raster process. Moreover, if the cache cube is pre-generated, even if we assume that it is a low-resolution cache cube containing just $128 \times 128 \times 128$ voxels, we still have an un-compressed file size of more than 8MB. This kind of huge data size is difficult to transmit through the

narrow Internet bandwidth and requires a lot of memory to store, although it cannot even offer good quality visual effects.

4.3.3. Storage Methodology

Fortunately, when rendering a 3D model, the visible area of the model is only a small portion of the whole shape, because the model itself is effectively just a shell, even if it is a solid model, i.e. the inner portion of the model can be considered as being empty. Therefore, there is only a relatively small amount of texture data that needs to be calculated and stored in the cache cube, so that the cache cube is sparse, i.e. the rest of the cache cube remains empty.

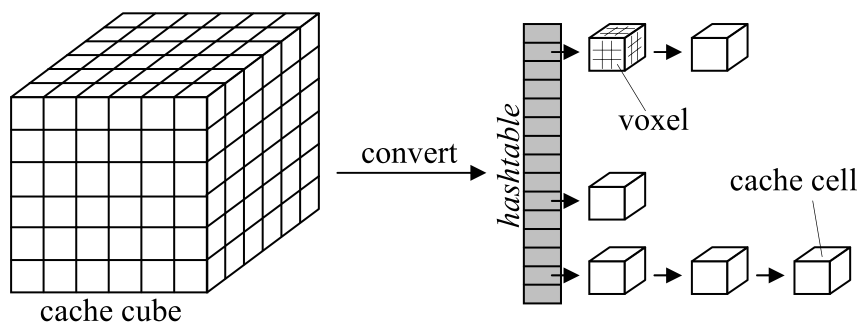


Figure 4-5: The cache cube is stored in a hashtable. The cache cube is subdivided into several cells and only the cache cells which contain several voxels are stored in the hashtable, the empty cells are ignored.

To store the calculated texture data in a “sparse cache cube”, we separate each texture coordinate into two parts, one of which is the “global index”, and the other is the “local index”. Therefore, the cache cube is subdivided into several cells due to the global index, as shown in Figure 4-5. Each cell is classified as either a “cache cell” which contains one or more voxels according to the size of the local index or just as an “empty cell” if there is no voxel located within it. To make the retrieval of the voxel efficient, we take the size of the cache cube to the power of two as the requirement for the texture image in some graphics libraries such as OpenGL, so that we can

use the bit calculation method rather than using integer or floating-point calculation. Therefore, a cache cell contains $2^{n'} \times 2^{n'} \times 2^{n'}$ voxels if we take the lower n' bits for the local index. If the size of the cache cube is n, the cache cube itself is subdivided into $2^{n-n'} \times 2^{n-n'} \times 2^{n-n'}$ cells, where some of the cells are cache cells and others are empty cells. Hence, the total data size in the memory is decreased because the empty cell does not need to be stored.

The global index is used to point to the cells. Although the texture coordinates and also the global index are three-dimensional, in order to make the cache cell retrieval efficient, we transform the three-dimensional index to be a one-dimensional index. Therefore, we can use a hashtable to store the cache cells by using the converted global index, since it could give us good performance for insertion and traversal. Although the number of the cache cells could be decreased by increasing the number of local index bits, the size of each cache cell is then also increased to have more voxels, so that saving more space or obtaining better retrieval performance seems to be a trade-off. Hence, to retrieve the desired texture data now becomes a two-step process, firstly to find the corresponding cache cell by using the global index, and then to get the voxel due to the local index or to insert the calculated one if there is none presented.

4.3.4. Proper Resolution Detection

Following the well-known definition of MIP-Mapping [74], we define the LOD (Level-of-Detail) parameter $\lambda(x, y)$ of our system as the following formula:

$$\lambda(x,y) = \begin{cases} n, & \lambda'(x,y) > n \\ \lambda'(x,y), & n \le \lambda'(x,y) \le n' \\ n', & \lambda'(x,y) < n' \end{cases},$$

where n is the maximum size of the cache cube, n' is the number of local index bits, and (x, y) are the viewport coordinates. Moreover, $\lambda'(x, y) = \log_2(\rho(x, y))$, where $\rho(x, y)$ is the scale factor and defined by

the following formula:

$$\rho(x,y)=\max\left\{\sqrt{\left(\frac{\partial u}{\partial x}\right)^2+\left(\frac{\partial v}{\partial x}\right)^2+\left(\frac{\partial w}{\partial x}\right)^2},\sqrt{\left(\frac{\partial u}{\partial y}\right)^2+\left(\frac{\partial v}{\partial y}\right)^2+\left(\frac{\partial w}{\partial y}\right)^2}\right\},$$

where (u,v,w) are the texture coordinates corresponding to the viewport coordinates (x,y), so that different pixels on the screen will not get the same texture data, and the un-used space in the cache cell is also minimized.

According to the LOD parameter, there are $(n-n'+1)$ levels of the cache cubes, so the most suitable level of the cache cube for each pixel on the screen is decided by the LOD parameter. In our system, if there is no LOD, the result will be worse if we set the size of the cache cube to be small. Otherwise, if we set the size of the cache cube to be large, the un-used space in the cache cell will be increased, since the adjacent pixels on the screen will be located in different cache cells, and the system may not be able to use a cache cube with high-resolution due to the memory space limitation. Moreover, since we also provide the LOD mechanism in our system, multiple cache cubes with different resolutions are used.

4.3.5. Result

To measure the performance of our approach with respect to the traditional procedural solid texturing methods, we used a marble texturing function and a simple cube that only contains six polygons as our 3D model in order to reduce the effects of the model complexity as shown in Figure 4-6. In Table 4-3, we make comparisons of our new method (using single cache cube) and two previous methods, one of which calculates the texture data on the fly (the previous solid texturing in Table 4-3), and the other generates a 3D-texture image first and then renders with 3D-texture mapping which is also done by software (the 3D-texture mapping in Table 4-3). For accelerating the performance of the previous solid texturing, we also use a lookup table to cache the random numbers. Obviously, both of the previous solid texturing and our approach do not require the preparation of a 3D-texture

image, but this is essential for the 3D-texture mapping, so users of the 3D-texture mapping technique have to be patient while they wait for it to be prepared.

<table>
<tr><th></th><th>previous solid texturing</th><th>3D-texture mapping</th><th>single cache cube</th></tr>
<tr><td>fill 3D image</td><td>0 sec</td><td>51.91 sec</td><td>0 sec</td></tr>
<tr><td>first loop</td><td rowspan="2">0.56 fps</td><td rowspan="2">18.27 fps</td><td>14.53 fps</td></tr>
<tr><td>rest loops</td><td>17.23 fps</td></tr>
<tr><td>resolution</td><td>∞</td><td colspan="2">128×128×128</td></tr>
</table>

Table 4-3: Comparisons of adaptive solid texturing and two previous methods. The frame rates of the previous methods are un-changed, whatever the first or rest loops. In our approach, the performance of the rest loops is better than the first loop, since the texture data is cached.

About the appearance result, since the previous solid texturing calculates the texture data on the fly, there is no resolution problem. However, the 3D-texture mapping needs a huge space to store the 3D-texture image, so that the size of it is only $128 \times 128 \times 128$ and it cannot achieve a higher resolution one on our testing platform, which is a laptop PC with an Intel Mobile Pentium III 850MHz CPU (512MB memory, Microsoft Windows XP Professional) and Sun Java 2 SDK, Standard Edition (J2SE) v 1.4.1_01.

first loop	7.18 fps	4.73 fps
rest loops	10.15 fps	10.68 fps
resolution	128×128×128	1024×1024×1024

Table 4-4: Comparisons of using multiple cache cubes with different maximum resolutions. Using multiple cache cubes to show a higher resolution result is possible.

During the rendering state, the 3D-texture mapping and using single cache cube in our approach could achieve a real-time response, but the previous solid texturing could not. To get a similar quality of the previous solid texturing, we use multiple cache cubes with a LOD controlling as shown in

Table 4-4. Although the performance is a little slower than using a single cache cube, but the appearance result is much better than using a low-resolution one. In our approach, the performances of the first loop and other rest loops are not the same, since there is no cached texture data available to be used in the beginning and our method have a slight delay when showing the first frame.

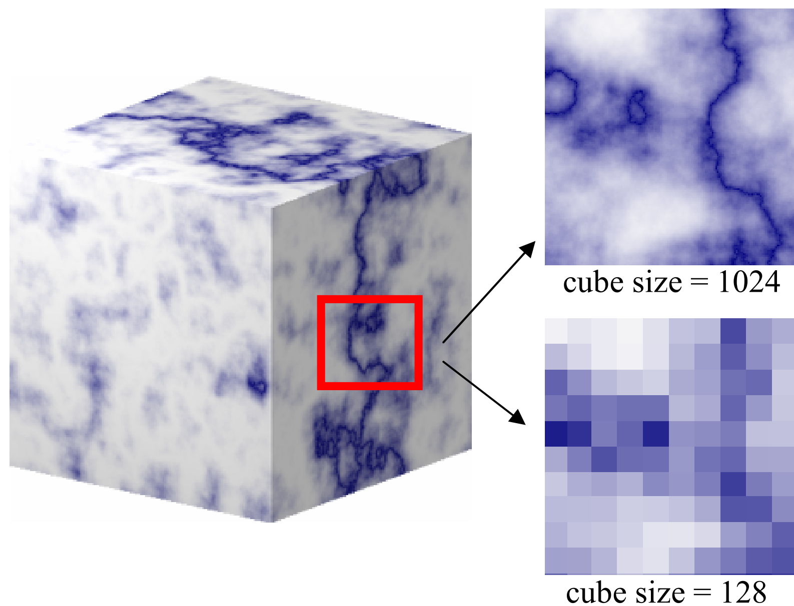
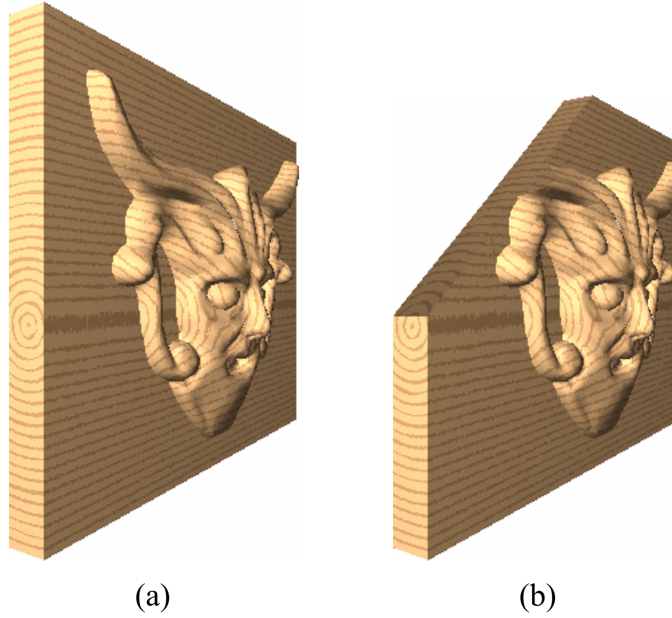


Figure 4-6: A cube rendered with a marble texturing function for performance testing. If a low-resolution cache cube is used, the appearance is worse. Otherwise, a high-resolution one is needed.



(a) (b)

Figure 4-7: The rendering results of (a) a 3D wood "ornamental face" model and (b) a clipped one.

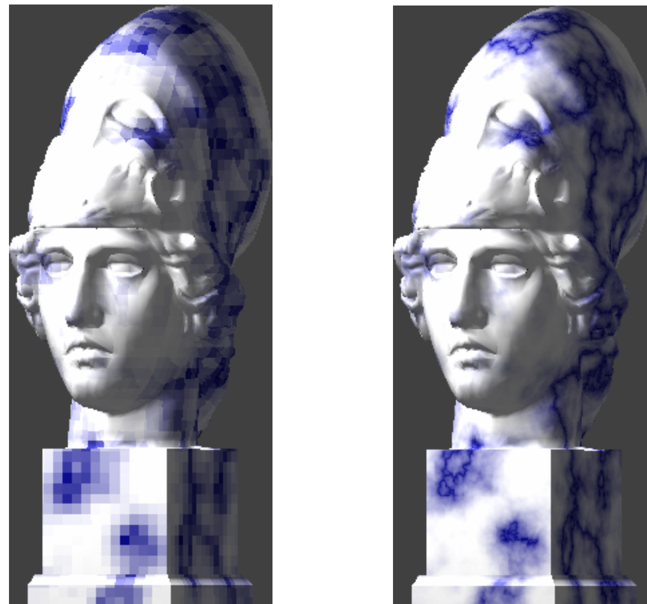


Figure 4-8: Two 3D marble "Atenea" models are rendered with different resolution cache cubes.

The differences in appearance when using a low-resolution cache cube and a high-resolution one are shown in Figure 4-6. Using a low-resolution cache cube, $128\times128\times128$ voxels for example, the appearance is blurred and looks like several grids, although this is also a limitation of the 3D-texture mapping method. On the other hand, using higher resolution one or multiple cache cubes set up with the appropriate LOD parameter; there is no such a problem.

The result of viewing the textured appearance of the cross section is shown in Figure 4-7. Two 3D wood "ornamental face" models, one of which is clipped by a clipping plane, are rendered. Because our method does not only simulate the surface of the 3D model, but also reserves a space for storing the interior texture data, showing the inner texture of a 3D model is akin to showing an invisible portion. The user will only experience a small delay due to the clipped plane calculation. There are two 3D marble "Atenea" models rendered with different resolution cache cubes in Figure 4-8.

For the Internet users, since we used only pure Java programming language to develop all of the algorithms and the executable byte-code size is less than 40KB (jar[1]-compressed) for the whole kernel, the use of our system on the Web[2] is possible. Moreover, unlike 3D-texture mapping, our approach has no requirement to generate any cache cube at the out-set, and all of the calculation and resolution adjusting is done on the fly, so users will not need to take up a lot of time downloading huge image data files or waiting for something to be prepared.

---

[1] "jar" is used for compressing and archiving the Java byte-codes to be a "jar-ball".

[2] *http://nis-lab.is.s.u-tokyo.ac.jp/~robin/jST/*

4.4. Summary

A VRML library for supporting jGL and an adaptive method for procedural solid texturing are described in this Chapter. VRML is a standard 3D model scripting language and very popular. To represent a 3D model or scene on the Internet, people would like to use the VRML format, and use a VRML browser plug-in of a Web browser to use it. Unfortunately, the support for displaying VRML models is not enough. Hence, a real platform independent VRML library as an extension of jGL is presented, and can be used on any Java enabled Web browsers.

Procedural solid texturing is a well-known computer graphics technology. However, it still has several problems, because it consumes too much time if every pixel is calculated on the fly or has a very high memory requirement if all of the pixels are stored in the beginning. Although some methods have recently been proposed to solve these problems, almost all of them need the support of hardware accelerators. Hence, these methods could not be applied to all kinds of machines, since not all of the machines have such support, especially the low-cost ones available over the Internet. Therefore, adaptive solid texturing is presented. The basic idea of it is similar to the cache mechanism used for main memory control. Therefore, it could almost render a 3D model with procedural solid texturing in real-time using only a software solution.

Chapter 5
Conclusions and Future Work

5.1. jGL - a 3D Graphics Library for Java

Since we offer jGL on our Web server for download, many people around the world have visited our Web page. We also received dozens of E-mails concerning the use of jGL. Some would like to collaborate with us, and some want to use jGL to develop their applications for Web Graphics. This encourages us to further improve jGL. Performance is still the great challenge for Java applications. Sun Microsystems, Inc. has combined its Java 2D into Java 2 SDK, Standard Edition (J2SE). Because Java 2D is part of Java core packages, it can benefit from hardware acceleration, though this will need many efforts on porting it to each platform. Using Java 2D to be the base of jGL may be a solution of the performance problem. But the main contribution of jGL is not only to provide a 3D graphics library for Java, but also to offer a familiar Web Graphics programming environment for all 3D programmers.

From our comparisons, although Java 3D uses OpenGL or DirectX as its graphics engine, the performance of jGL is not worse for a simple model. Hence, jGL seems more suitable for small 3D models on the Web, since the user does not need to install any run-time library before using the program which is developed with it. Moreover, the API (Application Programming Interface) of jGL is similar to that of OpenGL, so the programmers can use it intuitively.

At this moment, jGL is being applied to many Java-based projects. The goal of these Java-based projects is to provide users all of the necessary

functionalities by direct downloading from the Web server, so that the users do not have to install any additional hardware or software for 3D graphics applications on the Web. jGL meets this requirement because it is implemented purely in Java, which is designed for mobile code on the Internet.

5.2. Streaming Mesh with Shape Preserving

A multiresolution streaming mesh for Internet transmission with QoS-like controlling is presented. It also includes an efficient 3D mesh simplification method for unstructured meshes. Although our approach is not able to generate an almost optimized simplified model, to make the shape and features of the simplified model still recognizable is the main purpose of our method. However, a simplified model generated by some previous methods sometimes becomes only a polyhedron. A polyhedron is hardly to image the shape of the original model and is useless in practice. Since our method could provide a simplified model of fewer than 20KB on average, this kind of small size could be transmitted efficiently through the Internet.

Moreover, the size of the patch used for reconstructing the original 3D model is less than that used in other previous methods. For example, when using the data structure as the PM (Progressive Meshes) method, the data size is twice as large as our patch size, if both of them are stored as binary files. Therefore, the client site could receive more mesh data from the server by using our approach compared to other methods. In practice, by using the QoS-like transmission method, the user using our system could get a better model than using other systems, since he or she could receive more mesh data if the transmission time is set the same. Finally, if the user at the client site really needs the original model, after receiving all of the patches, he or she could still get the lossless original model without any redundant data transmission.

By using this client/server system, the users could receive different 3D models in different resolutions according to their current network bandwidth. Moreover, the model provider does not need to prepare so many 3D models with different resolutions on the server side. The provider is asked to put

only one 3D model with the streaming mesh format on the server, and the server program will talk with the client program to offer a proper 3D model.

Additionally, since our method is efficient, the model provider can change the threshold interactively to get a proper simplified model with the appropriate data size so that the provider can guarantee what resolution of simplified model will be transmitted to the users.

5.3. jVL – a VRML Support for jGL

Although the implementation of jVL is not completed yet, to fully support all of the VRML (Virtual Reality Modeling Language) set is not our original goal. jVL and jGL are developed as useful tools to help us to do some advanced projects for Web Graphics, so we only develop the necessary parts in these packages when we need them.

5.4. Adaptive Solid Texturing for Web Graphics

A simple method to render a 3D model with procedural solid texturing for almost all kinds of machines over the Internet is also proposed. Although the implementation only uses pure Java programming language, the user could also achieve an almost real-time interactive response. Since there are several low-cost machines over the Internet, we also provide a mechanism to control the resolution of the cache cubes automatically in accordance with the capability of the client machine. Finally, to utilize other methods of traditional cache technology, like pre-fetching or swapping, to get better performance in our approach is also a future work.

5.5. Contribution

In this dissertation, a prototype of transmitting 3D geometric models on the Internet has been proposed. In order to provide this client/server system for Web Graphics, we contributed from a 3D graphics library as a development environment to a total solution to establish the system including 3D mesh simplification, transmission, and reconstruction. Furthermore, a real-time shading method of procedural solid texturing which can be used on the Web and two model deformation methods are also proposed to support this model transmitting prototype.

jGL is provided as a basic development environment for Web Graphics. It is the only general-purpose 3D graphics library for Java and has no necessary to have the support from any other native library. Moreover, its API is defined as that of OpenGL, so that people who are familiar with the usage of OpenGL can use it intuitively since they can find one-to-one mapping functions in both of OpenGL and jGL. Now, many people around the world are using jGL, such as to provide previous OpenGL works on the Web, or use it to teach and learn computer graphics algorithms. Besides the functionalities of OpenGL, jGL also supports other useful functionalities, such as Phong shading, bump mapping, environment mapping, procedural sold texturing, and VRML. Furthermore, the VRML support is provided as an independent library called jVL.

The main contributions of the proposed multiresolution streaming mesh are an efficient mesh simplification algorithm for unstructured meshes and a QoS-like flow control mechanism. The mesh simplification algorithm is different from other previous methods: it has no over-simplification problem, which is a common problem of other methods, since we detect the features of the model before simplifying it. Even if the meshes consisted by the model is unstructured; our method can also preserve the shape and features of the original model. Although the method used for detecting the features of the model is simple, it can contribute a good result efficiently. The detailed discussion of the proposed mesh simplification algorithm and other previous methods is described in Section 3.7.2. With the QoS-like control mechanism, the users with different qualities of network connections can

receive different resolutions of geometric 3D models due to the current network bandwidth.

The main idea of the proposed procedural solid texturing, which is an extension of jGL, is to utilize the knowledge of the cache mechanism used in computer hardware. Using the cache mechanism, the computational cost can be reduced, since the calculated texture data will not be re-calculated again. Because the visible portion of a 3D model is not so large, we can show a 3D model with high-resolution appearance using only a few storage spaces for the cached texture data. This method solved two main problems of previous procedural solid texturing methods. One is the run-time performance problem due to the computational cost, and the other is the storage problem since to create a 3D-texture image as texture mapping needs a lot of storage spaces. Therefore, our approach can show a 3D model with procedural solid texturing with no such problems.

Appendix A. Usage of jGL

In this Appendix, there are three sub-sections for describing the usage of jGL. The first is to point out the differences between jGL and OpenGL. Then, the functionalities of jGL are listed. Finally, the rest sub-section describes how to use jGL from a programmer's view.

A.1. Differences between jGL and OpenGL

Although the development of jGL follows the specifications of OpenGL [16] [45] [70], there are still some differences between jGL and OpenGL due to the constraints of the Java programming language [2] [30].

1. **jGL has no structure data type**

   Because Java is an OOP (Object-Oriented Programming) language, there is no structure data type as in the C programming language. If there is a structure defined in OpenGL, there will be a class in jGL similar to the field definition of the structure. For example, there is a structure called GLUquadricObj in GLU for describing how a quadric should be constructed and rendered. Since jGL has no structure, there is a class with the same name and is put in jgl.glu.GLUquadricObj. Users can use this class instead of using the structure.

2. **jGL is platform independent**

   OpenGL is a platform independent 3D graphics library, the primitive data types defined in OpenGL, for example the GLint and GLfloat, are not the real primitive data types of the native platform, so OpenGL will convert them to the real primitive data types in the compile phase. On

the other hand, Java is also platform independent, but jGL does not need to convert the platform independent primitive data types defined in jGL to the primitive data types defined in Java. Therefore, jGL uses the primitive data types defined in Java directly, and does not use the primitive data types defined in OpenGL.

3. jGL does not allow address reference

In OpenGL, to use address reference is a common technique for some unknown data structure. For example, when users call glDrawPixels, they can change the type of the input array, but this is not allowed in Java. Therefore, we change the function definition to let users convert the input array to be an Object class in Java, so that they can have similar usage of jGL as using OpenGL.

4. jGL has no dithering mode

There is a rendering mode called dithering, which is used on the machine that does not support true color mode, but it is not necessary for jGL to concern this problem because Java will handle it. When we want to draw something on the screen, all we have to do is just writing the color values onto the screen as in true color mode, if the real machine does not support the true color mode, Java will show the dithered color on the screen.

5. jGL has no single buffer mode

Because jGL can not access real window framebuffer of the display card, jGL does not support single buffer mode. For the performance concern, if all of the drawing works are done in the background, the drawing works will be faster than done in the foreground directly.

6. jGL has several classes

jGL is designed for the Java programming language, which is an OOP language. jGL has three main classes: GLUT, GLU, and GL, so that to use the constants defined in OpenGL, programmers must use GL.GL_CONSTANTS instead of using GL_CONSTANTS as in the C programming language. Moreover, we also offer GLCanvas and GLApplet classes for users, so that they can inherit one of them to ignore the concern of the Java environment.

A.2. Functionalities of jGL

To make the functionalities of jGL to be easy to know, we list all of the implemented functionalities as listed in Figures A-1 ~ A-4 which are listed as the section names of the OpenGL specifications.

2. OpenGL Operation
    2.5 GL Errors
    2.6 Begin/End Paradigm
        2.6.1 Begin and End Objects
    2.7 Vertex Specification
    2.9 Rectangles
    2.10 Coordinate Transformations
        2.10.1 Controlling the Viewport
        2.10.2 Matrices
        2.10.3 Normal Transformation
        2.10.4 Generating Texture Coordinates
    2.11 Clipping
    2.13 Colors and Coloring
        2.13.1 Lighting
        2.13.2 Lighting Parameter Specification
        2.13.3 ColorMaterial
        2.13.4 Lighting State
        2.13.6 Clamping or Masking
        2.13.7 Flatshading
        2.13.8 Color and Texture Coordinate Clipping
        2.13.9 Final Color Processing
3. Rasterization
    3.3 Points
    3.4 Line Segments
        3.4.1 Basic Line Segment Rasterization
        3.4.2 Other Line Segment Features
        3.4.3 Line Rasterization State

Figure A-1: Implemented OpenGL functionalities in jGL.

- 3.5 Polygons
  - 3.5.1 Basic Polygon Rasterization
  - 3.5.2 Stippling
  - 3.5.4 Options Controlling Polygon Rasterization
  - 3.5.7 Polygon Rasterization State
- 3.6 Pixel Rectangles
  - 3.6.1 Pixel Storage Modes
- 3.8 Texturing
  - 3.8.1 Texture Image Specification
  - 3.8.4 Texture Parameters
  - 3.8.7 Texture Wrap Modes
  - 3.8.8 Texture Minification
  - 3.8.9 Texture Magnification
  - 3.8.11 Texture State and Proxy State
  - 3.8.12 Texture Objects
  - 3.8.13 Texture Environments and Texture Functions
  - 3.8.15 Texture Application

4. Pre-Fragment Operations and the Framebuffer
   - 4.1 Pre-Fragment Operations
     - 4.1.6 Depth Buffer Test
   - 4.2 Whole Framebuffer Operations
     - 4.2.3 Clearing the Buffers
   - 4.3 Drawing, Reading, and Copying Pixels
     - 4.3.2 Reading Pixels
     - 4.3.3 Copying Pixels
5. Special Functions
   - 5.1 Evaluators
   - 5.2 Selection
   - 5.4 Display Lists
   - 5.5 Flush and Finish
6. State and State Requests
   - 6.1 Querying GL State
     - 6.1.1 Simple Queries
     - 6.1.2 Data Conversions
     - 6.1.3 Enumerated Queries
     - 6.1.12 Saving and Restoring State

Figure A-2: Implemented OpenGL functionalities in jGL (cont.).

3. Beginning Event Processing
   3.1 glutMainLoop
4. Window Management
   4.1 glutCreateWindow
   4.5 glutPostRedisplay
   4.6 glutSwapBuffers
6. Menu Management
   6.1 glutCreateMenu
   6.2 glutSetMenu, glutGetMenu
   6.3 glutDestroyMenu
   6.4 glutAddMenuEntry
   6.5 glutAddSubMenu
   6.9 glutAttachMenu, glutDetachMenu
7. Callback Registration
   7.1 glutDisplayFunc
   7.3 glutReshapeFunc
   7.4 glutKeyboardFunc
   7.5 glutMouseFunc
   7.6 glutMotionFunc, glutPassiveMotionFunc
   7.18 glutIdleFunc
11. Geometric Object Rendering
   11.1 glutSolidSphere, glutWireSphere
   11.2 glutSolidCube, glutWireCube
   11.3 glutSolidCone, glutWireCone
   11.4 glutSolidTorus, glutWireTorus
   11.9 glutSolidTeapot, glutWireTeapot

Figure A-3: Implemented GLUT functionalities in jGL.

The implemented VRML functionalities and nodes are listed in Figure A-5, which are listed as the section names of the VRML specification [6]. Besides OpenGL, GLU, GLUT, and VRML, we also enhance jGL's abilities by adding Phong shading, bump mapping, environment mapping, and procedural solid texturing into jGL, although these parts are not supported by OpenGL.

4. Matrix Manipulation
   4.1 Matrix Setup
   4.2 Coordinate Projection
6. Quadrics
   6.1 The Quadrics Object
   6.3 Rendering Styles
   6.4 Quadrics Primitives
7. NURBS
   7.1 The NURBS Object
   7.3 NURBS Curves
   7.4 NURBS Surfaces
   7.6 NURBS Properities
8. Errors

Figure A-4: Implemented GLU functionalities in jGL.

4. Concepts
   4.6 Node semantics
      4.6.2 DEF/USE semantics
      4.6.3 Shapes and geometry
         4.6.3.1 Introduction - Box, Cone, Cylinder IndexedFaceSet, IndexedLineSet, PointSet, Sphere
         4.6.3.2 Geometric property nodes - Coordinate, Color, Normal
         4.6.3.3 Appearance nodes - Material
         4.6.3.5 Grouping and children nodes - Group, Transform
      4.6.6 Light sources - DirectionalLight, PointLight, SpotLight
      4.6.7 Sensor nodes
         4.6.7.2 Environment sensors - TimeSensor
      4.6.8 Interpolator nodes - ColorInterpolator, CoordinateInterpolator, NormalInterpolator, OrientationInterpolator, PositionInterpolator
   4.7 Field, eventIn, and eventOut semantics
   4.8 Prototype semantics
   4.10 Event processing
5. Field and event reference
6. Node reference

Figure A-5: Implemented VRML nodes in jVL.

A.3. How to Use jGL

1. **jGL at the server site**

   The Web master must put jGL in the same directory of the Java applets which are developed with it. Then, the users across the Internet can use the applets by downloading jGL at the same time when they download the byte-code of the applets. The size of jGL is less than 150KB, so this is very small and does only need to be downloaded only once. If the users use other Java applets which are also developed with jGL, jGL will not be downloaded again.

2. **jGL at the client site**

   If the users do not want to download jGL every time when they use the Java applets developed with jGL, they can download the newest jGL run-time library before using these applets from jGL's Web site, and add it into the environment variable CLASSPATH. Then, when the users use the Java applets, they will only need to download the byte-code of these applets. Moreover, jGL could also be used in Java application mode, so that users could also use it to develop some programs locally as using other 3D graphics libraries.

3. **To the programmers**

   Please download the jar-ball of jGL run-time library from jGL's Web site. Then, you can use this library like using OpenGL. To use jGL in the Java applets, the programmers must import the classes of jGL first, and then declare the instances of the classes, finally use the member functions of the classes as calling the functions of OpenGL. Figure 2-8 shows a Java applet source code as a simple jGL program using GLUT to show a white rectangle. The same program written with OpenGL is listed in Figure 2-7 that is an example provided in the OpenGL Programming Guide (code from Example 1-2, pages 18-19, Figure 1-1) [77].

   To use jGL, the Java applet or application must import the jGL class

named “jgl.GL”, if the programmers want to use GLU or GLUT, they must also import “jgl.GLU” or “jgl.GLUT”, like including “GL/gl.h”, “GL/glu.h” and “GL/glut.h” when using the C programming language and OpenGL. Because GL, GLU, and GLUT are three different classes, the programmers must declare the instances of these classes. GLU and GLUT are two classes which will call the rendering functions of GL, when declare these two instances of them, we must give them the instance of GL. Then, they can use all of the jGL functions as using the OpenGL functions. Like the source codes shown in Figures 2-7 and 2-8, all of the functions in jGL and OpenGL are one-to-one mapping. Besides the three main classes, we also offer GLCanvas and GLApplet classes for programmers, so that they can inherit one of them to ignore the concern of the Java environment as the example shown in Figure 2-8.

Appendix B. jGL for Mobile Phone

Recently, the Java applet can be run on some mobile phones, for example i-αppli of the 503i and 504i series provided by NTT DoCoMo[1] in Japan. Since i-αppli is not standard and based on Java 2 SDK, Micro Edition (J2ME) which is different from the platform of jGL, jGL can not be run on the mobile phones directly. To make the mobile phones to have the 3D graphics supports, we ported some basic parts of jGL onto the i-αppli platform.

The i-αppli platform is different from the common Java platforms, such as Java 2 SDK, Standard Edition (J2SE) and Java 2 SDK, Enterprise Edition (J2EE). To port jGL onto it, we have some limitations due to the specific platform: (1) there is no floating-point number, (2) the data size of the jar[2]-ball must be smaller than 10KB, and (3) we can only use the functions provided by i-αppli to draw something onto the screen.

Therefore, to port onto the i-αppli platform, we implement all of the necessary calculation by using only the integer numbers. To minimize the jar-ball size, we remove un-necessary constants and error checks. Finally, we have implemented more than 30 OpenGL functions in the i-αppli version of jGL, including 3D model transformation, 3D object projection, hidden-surface removal, primitive geometry, etc. Moreover, the jar-ball size is about 4,194 bytes.

To test the i-αppli version of jGL, we use a movable robot arm as shown in Figure B-1. This test program is an example in the OpenGL Programming Guide (code from Example 3-7, pages 148-150, Figure 3-25) [77]. Moreover, to use the button on the mobile phone, the robot arm can be moved as the

---

1 *http://www.nttdocomo.co.jp/p_s/imode/java/index.html*

2 “jar” is used for compressing and archiving the Java byte-codes to be a “jar-ball”.

example in the OpenGL Programming Guide.

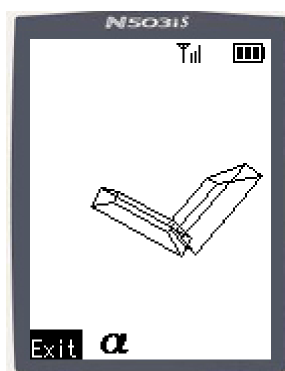


Figure B-1: A robot arm is rendered on a mobile phone. This program is an example in the OpenGL Programming Guide (code from Example 3-7, pages 148-150, Figure 3-25). This figure is rendered with i-αppli version of jGL running on the simulation of i-αppli.

Appendix C. Model Deformation

In this Appendix, two model deformation methods are presented to support the 3D model design for the streaming mesh. FFD (Free-Form Deformation) [68] is an efficient technique for editing the shapes of 3D models and widely used. The basic idea of some famous previous FFD approaches deforms a target 3D model by adjusting some control points of a 3D lattice surrounding the model. It is a tedious work, especially when the lattice contains too many control points. Moreover, how to place the control points of the lattice to make them cover the region of the target model is also a problem. Therefore, we provide two methods for model deformation. One is deforming the model along a parametric surface, so that the user does not need to set or control the control points of a lattice. The other is to generate proper lattices automatically if the user wishes to use some previous FFD techniques; hence to place the control points is also not needed.

C.1. Deformation along a Parametric Surface

To use a given parametric surface to do the 3D model deformation, the user first selects a well-defined surface, such as the surface of a cone or a cylinder, or generates a parametric one. The selected well-defined surface or generated parametric one is the desired shape after the target model is deformed, i.e. the deformed result will be like the shape along the given surface. The mapping relationship between the given parametric surface and the target 3D model is like the mapping between a free-form surface and a 2D-texture image, because in texture-mapping, the texture image is mapped onto the curved surface, and in our approach, the target model is mapped onto the given parametric surface, if we think of the texture image as a plane

in a three-dimensional space.

C.1.1. Subdivision on a Flattened Surface

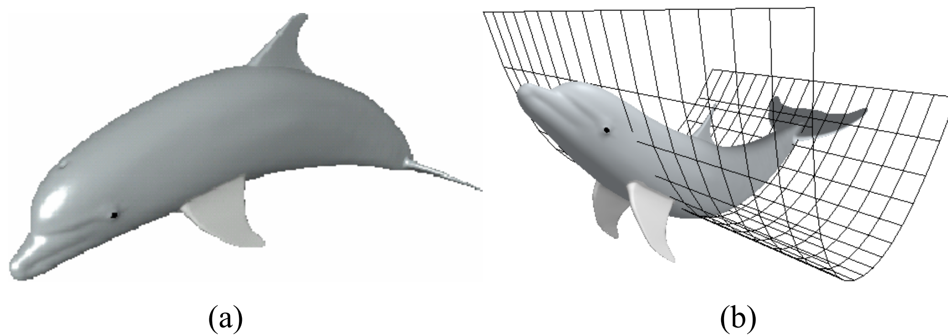
Ma and Lin proposed an approximate non-distortion texture-mapping method [58]. In this method, the curved surface is flattened to a 2D plane. Except for some kinds of 3D curved surfaces such as the surface of a cylinder or a corn which is called "developable surface", other general surfaces like the Bézier surface cannot be completely flattened to 2D planes without any distortion, so they could only have approximate flattened surfaces. By using a given parametric surface to do the 3D model deformation, we first flatten the surface, and then put the model onto the flattened surface, so that the model can get a proper corresponding mapping with the parametric surface. Moreover, unlike the other conventional FFD methods, the proposed method is done without generating any lattice-like structure, and the deformation task is also easier than before.

To get a smooth deformed object, we have to subdivide the target 3D model, if the size of the polygon of the model is larger than the size of the grid of the flattened surface. We first project all of the vertices, edges, and faces of the target model to the flattened surface, and then compute the following three types of points with the grid of the flattened surface of the given parametric surface: (1) The "vertex points" which are defined as the original projected vertices of the target model. (2) The "edge points" which are generated by intersected the original projected edges of the target model with the grid of the flattened surface. (3) The "face points" which are generated by inserted the cross points of the grid of the flattened surface inside the original projected faces of the target model.

Then, the edge points and face points are projected backward to the original parametric surface, and the corresponding points on the target model are also generated by using bi-linear interpolation. Finally, the generated corresponding points are displaced along the normal vectors of the parametric surface. Hence, there are two corresponding points for each vertex point, edge point, and face point: one is on the parametric surface, and the other is on the surface of the target model. Although there are several generated edge points and face points, to make the deformed model not to contain too

much newly generated vertices, the curvatures of the corresponding points on the parametric surface of the edge points and face points are computed, so that only the bumpy region due to the parametric surface needs to be subdivided.

C.1.2. Result



(a) (b)

Figure C-1: (a) an original dolphin model and (b) its deformed result along a Bézier surface.

Figure C-1 (a) shows a 3D model "dolphin" and its deformed result is shown in Figure C-1 (b). The deformation is along a Bézier surface. The number of grids of the Bézier surface is 16×16 and to flatten the surface needs 777 ms which is a pre-process and can be done at off-line. To deform the model along the surface costs 168.2 ms on a desktop PC with an Intel Xeon 2GHz CPU (1GB memory, Microsoft Windows 2000 Professional) and a 3Dlabs Wildcat II 5110 GPU (Graphics Processing Unit). Since our method can get a real-time response, it is suitable for doing animation. For example, the dolphin model shown in Figure C-1 (a) is deformed to simulate the swimming dolphin as shown in Figure C-2. To do the animation, the dolphin is put onto a parametric surface, and then the dolphin can be deformed along the surface in real-time, although it contains 73,665 polygons.

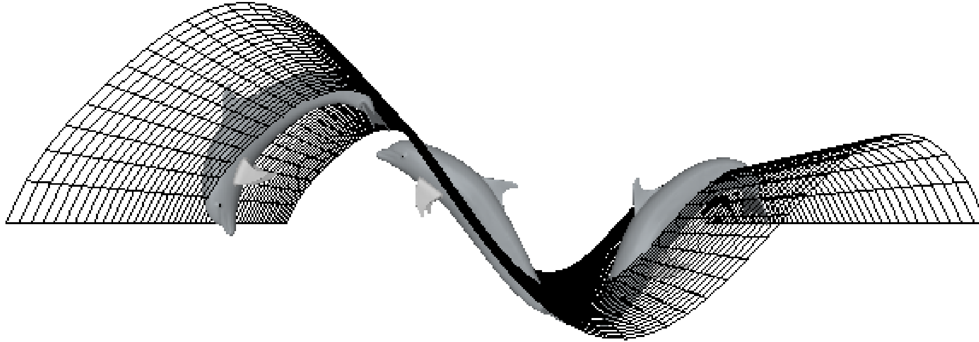


Figure C-2: A dolphin model deformed along a Bézier surface to make a swimming dolphin animation.

C.2. Deformation with Auto-generated Lattices

Before describing the details of our auto-generated lattices method, we first define the terms as [59]. (1) A “lattice” is defined as a set of control points and an associated set of pairs that specifies the connectivity of the control points. (2) An “edge” of the lattice is defined by two control points that are connected in the lattice. (3) A “face” of the lattice is defined by a minimal connected loop of control points. (4) A “cell” of the lattice is the region of space bounded by a closed set of faces. Our deformation approach allows lattices with the following properties, called “valid lattices”: (1) the lattice is well-connected, (2) all cells of the lattice are closed, not containing any holes, and (3) the lattice is not self-intersecting.

C.2.1. Octree Subdivision Lattices

When deforming a 3D model, the region of the model to be deformed is first determined by the user. Although the region maybe covers the whole model or just only some parts of it, our approach can be applied to both types. Given the region to be deformed, our method makes the lattices hierarchically approximate to the shape of the region by repeating 3D subdivi-

sions of a bounding box, which is identified with the lowest-level lattice of the multiresolution lattices. After defining the lowest-level lattice, we generate multiresolution lattices using octree subdivision rules that are similar to the rules for octree subdivision of 3D space. Since the lattices can approximate the shape of the model hierarchically and are comprised of uniformly-located control points, the user can do any level of deformation with ease and intuition. Moreover, since octree subdivision rules can be applied to any valid lattice, this method can be applied not only to unmodified lattices but also to the user-modified ones. This allows the user to do hierarchical deformation.

C.2.2. Hierarchical Deformation

The process for hierarchical deformation using more than one lattice of the multiresolution lattices is slightly different from the process of just using one lattice, since the user could change the deformation levels from global to local and vice versa. Hierarchical deformation from global to local proceeds as follows: (1) the user globally deforms the model with a low-level lattice, (2) a finer lattice is generated by applying the octree subdivision rules to the modified lattice, (3) the user locally deforms the model with the finer lattice, and (4) repeat steps (2) and (3) if a finer deformation is needed.

To achieve this kind of hierarchical deformation is rather simple, but there is one problem caused by the difference between the octree subdivision and the Catmull-Clark [7] subdivision rules. Although we can solve this problem by using the Catmull-Clark subdivision rules instead of the octree subdivision ones when we generate multiresolution lattices, the lattices generated by using the Catmull-Clark subdivision rules sometimes produce rather scattered control points and not easy to deform. Moreover, it is almost impossible to create high-level lattices because of its computational cost. Therefore, we make a lookup table here to correct the gap.

Hierarchical deformation from local to global is rather difficult because the low-level lattices regenerated by applying the inverse operation of octree subdivision rules to the high-level lattice may not contain whole the region to be deformed. Therefore, we provide this kind of deformation by defining a new low-level lattice that encloses both the model and the high-level lat-

tice. The control points of the high-level lattice are also deformed by the modification of the new low-level lattice as the model. Therefore, the user can do the global deformation even if some local deformations are performed and vice versa.

C.2.3. Result

Figure C-3 shows a process of hierarchical deformation. The model is first subjected to global deformation, and then successively to local deformations without redefining the lattices from the bounding box. Since refined lattices generated from a deformed lattice keep track of the features of the deformed model, the user can intuitively continue to deform the model, which is one of the main advantages of our method.

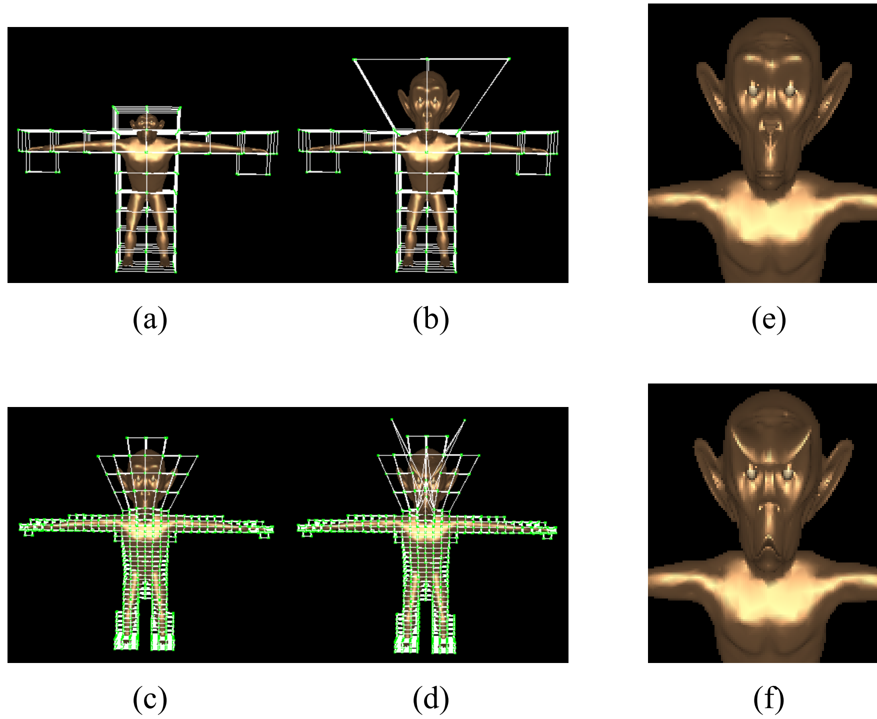


Figure C-3: Hierarchical deformation of a chimpanzee model: (a) original model with a low-level lattice; (b) deformed model with deformed low-level lattice; (c) deformed model with a high-level lattice generated from that of (b); (d) further deformed model with deformed high-level lattice; (e) closed up view of (b) and (c); (f) closed up view of (d).

Bibliography

[1] Pierre Alliez and Mathieu Desbrun. Progressive compression for lossless transmission of triangle meshes. *ACM SIGGRAPH 2001 Conference Proceedings*, pages 195-202, 2001.

[2] Ken Arnold, James Gosling, and David Holmes. *The Java™ Programming Language*. Addison-Wesley, 3rd. edition, 2000.

[3] James Arvo. *Graphics Gems II*. Academic Press, 1994.

[4] Darryl P. Black. *Building Switched Networks: Multilayer Switching, QoS, IP Multicast, Network Policy, and Service Level Agreements*. Addison-Wesley, 1999.

[5] Bob Braden, David Clark, and Scott Shenker. *Integrated Services in the Internet Architecture: an Overview*. RFC 1633, 1994.

[6] Rikk Carey, Gavin Bell, and Chris Marrin. *ISO/IEC 14772-1:1997 Virtual Reality Modeling Language (VRML97)*. The VRML Consortium, Inc., 1997.

[7] Edwin E. Catmull and James H. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. Computer-Aided Design, Vol. 10, No. 6, pages 350-355, 1978.

[8] Bing-Yu Chen. *The JavaGL 3D Graphics Library & JavaNL Network Library*. Master Thesis. Department of Computer Science and Information Engineering, National Taiwan University, 1997.

[9] Bing-Yu Chen and Tomoyuki Nishita. jGL and its applications as a Web3D platform. *ACM Web3D 2001 Conference Proceedings*, pages 85-91, 2001.

[10] Bing-Yu Chen and Tomoyuki Nishita. Multiresolution streaming mesh with shape preserving and QoS-like controlling. *ACM Web3D 2002 Conference Proceedings*, pages 35-42, 2002.

[11] Bing-Yu Chen and Tomoyuki Nishita. The development of 3D graphics and VRML libraries for Web3D platform by using Java. *IEICE Transactions on Information and Systems*, Vol. J85-D-II, No. 6, pages 1047-1054, 2002.

[12] Bing-Yu Chen and Tomoyuki Nishita. Adaptive solid texturing for Web3D applications. *Pacific Graphics 2002 Conference Proceedings*, pages 433-434, 2002.

[13] Bing-Yu Chen and Tomoyuki Nishita. An efficient mesh simplification method with feature detection for unstructured meshes and web graphics. *Computer Graphics International 2003 Conference Proceedings*, 2003.

[14] Bing-Yu Chen, Yutaka Ono, Henry Johan, Masaaki Ishii, Tomoyuki Nishita, and Jieqing Feng. 3D model deformation along a parametric surface. *IASTED Visualization, Imaging and Image Processing 2002 Conference Proceedings*, pages 282-287, 2002.

[15] Bing-Yu Chen, Tzong-Jer Yang, and Ming Ouhyoung. JavaGL - a 3D graphics library in Java for Internet browsers. *IEEE Transactions on Consumer Electronics*, Vol. 43, No. 3, pages 271-278, 1997.

[16] Norman Chin, Chris Frazier, Paul Ho, Zicheng Liu, and Kevin P. Smith. *The OpenGL® Graphics System Utility Library (Version 1.3)*. Silicon Graphics, Inc., 1998.

[17] Paolo Cignoni, Claudio Rocchini, and Roberto Scopigno. Metro: measuring error on simplified surfaces. *Computer Graphics Forum*, Vol. 17, No. 2, pages 167-174, 1998.

[18] Jonathan Cohen, Amitabh Varshney, Dinesh Manocha, Greg Turk, Hans Weber, Pankaj Agarwal, Frederick Brooks, and William Wright. Simplification envelopes. *ACM SIGGRAPH 96 Conference Proceedings*, pages 119-128, 1996.

[19] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steve Worley. *Texturing and Modeling: A Procedural Approach*. Academic Press, 2nd. edition, 1998.

[20] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution analysis of arbitrary meshes. *ACM SIGGRAPH 95 Conference Proceedings*, pages 173-182, 1995.

[21] Herbert Edelsbrunner. *Geometry and Topology for Mesh Generation*. Cambridge University Press, 2001.

[22] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Frystyk, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. *Hypertext Transfer Protocol -- HTTP/1.1*. RFC 2616, 1999.

[23] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice in C*. Addison-Wesley, 2nd. edition, 1996.

[24] Michael Garland. Multiresolution modeling: Survey & future opportunities. *Eurographics 99 State of the Art Report*, 1999.

[25] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. *ACM SIGGRAPH 97 Conference Proceedings*, pages 209-216, 1997.

[26] Michael Garland and Paul S. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. *IEEE Visualization 98 Conference Proceedings*, pages 263-269, 1998.

[27] Michael Garland, Andrew Willmott, and Paul S. Heckbert. Hierarchical face clustering on polygonal surfaces. *ACM Interactive 3D Graphics 2001 Conference Proceedings*, pages 49-58, 2001.

[28] Andrew S. Glassner. *Graphics Gems*. Academic Press, 1993.

[29] Meenakshisundaram Gopi and Dinesh Manocha. A unified approach for simplifying polygonal and spline models. *IEEE Visualization 98 Conference Proceedings*, pages 271-278, 1998.

[30] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification*. Addison-Wesley, 2nd. edition, 2000.

[31] André Guéziec, Gabriel Taubin, Bill Horn, and Francis Lazarus. A framework for streaming geometry in VRML. *IEEE Computer Graphics and Applications*, Vol. 19, No. 2, pages 68-78, 1999.

[32] Fred Halsall. *Data Communications, Computer Networks, and Open Systems*. Addison-Wesley, 4th edition, 1996.

[33] Paul S. Heckbert. *Graphics Gems IV*. Academic Press, 1994.

[34] Paul S. Heckbert and Michael Garland. Survey of polygonal surface

simplification algorithms. *Multiresolution Surface Modeling (ACM SIGGRAPH 97 Course Notes #25)*, 1997.

[35] Hugues Hoppe. Progressive meshes. *ACM SIGGRAPH 96 Conference Proceedings*, pages 99-108, 1996.

[36] Hugues Hoppe. View-dependent refinement of progressive meshes. *ACM SIGGRAPH 97 Conference Proceedings*, pages 189-198, 1997.

[37] Hugues Hoppe. Efficient implementation of progressive meshes. *Computer & Graphics*, Vol. 22, No. 1, pages 27-36, 1998.

[38] Hugues Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. *IEEE Visualization 98 Conference Proceedings*, pages 35-42, 1998.

[39] Hugues Hoppe. New quadric metric for simplifying meshes with appearance attributes. *IEEE Visualization 99 Conference Proceedings*, pages 59-66, 1999.

[40] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Surface reconstruction from unorganized points. *ACM Computer Graphics (SIGGRAPH 92 Conference Proceedings)*, Vol. 26, No. 2, pages 71-78, 1992.

[41] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. *ACM SIGGRAPH 93 Conference Proceedings*, pages 19-26, 1993.

[42] Andreas Hubeli and Markus Gross. Multiresolution feature extraction from unstructured meshes. *IEEE Visualization 2001 Conference Proceedings*, pages 287-294, 2001.

[43] Alan D. Kalvin and Russell H. Taylor. Superfaces: polygonal mesh simplification with bounded error. *IEEE Computer Graphics and Applications*, Vol. 16, No. 3, pages 64-77, 1996.

[44] Aadrei Khodakovsky, Peter Schröder, and Wim Sweldens. Progressive geometry compression. *ACM SIGGRAPH 2000 Conference Proceedings*, pages 271-278, 2000.

[45] Mark J. Kilgard. *The OpenGL Utility Toolkit (GLUT) Programming Interface API Version 3*. Silicon Graphics, Inc., 1996.

[46] David Kirk. *Graphics Gems III*. Academic Press, 1994.

[47] Leif Kobbelt. $\sqrt{3}$-Subdivision. *ACM SIGGRAPH 2000 Conference Proceedings*, pages 103-112, 2000.

[48] Ed Krol. *The Whole Internet User's Guide & Catalog*. O'Reilly & Associates, Inc., 2nd. edition, 1994.

[49] Aaron W. F. Lee, Wim Sweldens, Peter Schoröder, Lawrence Cowsar, and David Dobkin. MAPS: Multiresolution adaptive parameterization of surfaces. *ACM SIGGRAPH 98 Conference Proceedings*, pages 95-104, 1998.

[50] Bruno Lévy, Sylvain Petitjean, Nicolas Ray, and Jérome Maillot. Least squares conformal maps for automatic texture atlas generation. *ACM Transactions on Graphics (SIGGRAPH 2002 Conference Proceedings)*, Vol. 21, No. 3, pages 362-371, 2002.

[51] John P. Lewis. Algorithms for solid noise synthesis. *ACM Computer Graphics (SIGGRAPH 89 Conference Proceedings)*, Vol. 23, No. 3, pages 263-270, 1989.

[52] Peter Lindstrom and Greg Turk. Fast and memory efficient polygonal simplification. *IEEE Visualization 98 Conference Proceedings*, pages 279-286, 1998.

[53] Peter Lindstrom and Greg Turk. Evaluation of memoryless simplification. *IEEE Transactions on Visualization and Computer Graphics*, Vol. 5, No. 2, pages 98-115, 1999.

[54] Peter Lindstrom and Greg Turk. Image-driven simplification. *ACM Transactions on Graphics*, Vol. 19, No. 3, pages 204-241, 2000.

[55] David P. Luebke. A developer's survey of polygonal simplification algorithms. *IEEE Computer Graphics and Applications*, Vol. 21, No. 3, pages 24-35, 2001.

[56] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. *ACM SIGGRAPH 97 Conference Proceedings*, pages 199-208, 1997.

[57] David Luebke, Martin Reddy, Jonathan Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. *Level of Detail for 3D Graphics*. Morgan-Kaufmann, 2002.

[58] Song De Ma and Hong Lin, Optimal texture mapping, *Eurographics*

88 Conference Proceedings, pages 421-428, 1988.

[59] Ron MacCracken and Kenneth I. Joy. Free-form deformation with lattices of arbitrary topology. *ACM SIGGRAPH 96 Conference Proceedings*, pages 181-188, 1996.

[60] Yutaka Ono, Bing-Yu Chen, Tomoyuki Nishita, and Jieqing Feng. Free-form deformation with automatically generated multiresolution lattices. *Cyber Worlds 2002 Conference Proceedings*, pages 472-479, 2002.

[61] Darwyn R. Peachey. Solid texturing of complex surfaces. *ACM Computer Graphics (SIGGRAPH 85 Conference Proceedings)*, Vol. 19, No. 3, pages 279-286, 1985.

[62] Ken Perlin. An image synthesizer. *ACM Computer Graphics (SIGGRAPH 85 Conference Proceedings)*, Vol. 19, No. 3, pages 287-296, 1985.

[63] Ken Perlin and Eric M. Hoffert. Hypertexture. *ACM Computer Graphics (SIGGRAPH 89 Conference Proceedings)*, Vol. 23, No. 3, pages 253-262, 1989.

[64] Jovan Popović and Hugues Hoppe. Progressive simplicial complexes. *ACM SIGGRAPH 97 Conference Proceedings*, pages 217-224, 1997.

[65] Rémi Ronfard and Jarek Rossignac. Full-range approximation of triangulated polyhedra. *Computer Graphics Forum (Eurographics 96 Conference Proceedings)*, Vol. 15, No. 3, pages 67-76, 1996.

[66] Pedro V. Sander, John Snyder, Steven J. Gortler, and Hugues Hoppe. Texture mapping progressive meshes. *ACM SIGGRAPH 2001 Conference Proceedings*, pages 409-416, 2001.

[67] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. *ACM Computer Graphics (SIGGRAPH 92 Conference Proceedings)*, Vol. 26, No. 2, pages 65-70, 1992.

[68] Thomas M. Sederberg and Scott R. Parry. Free-form deformation of solid geometric models. *ACM Computer Graphics (SIGGRAPH 86 Conference Proceedings)*, Vol. 20, No. 4, pages 151-160, 1986.

[69] Robert Sedgewick. *Algorithms in C++*. Addison-Wesley, 1992.

[70] Mark Segal and Kurt Akeley. *The OpenGL® Graphics Systems: A*

Specification (Version 1.4). Silicon Graphics, Inc., 2002.

[71] Bill Shannon, Mark Hapner, Vlada Matena, Eduardo Pelegri-Llopart, James Davidson, Larry Cable, and The Enterprise Team. *Java™ 2 Platform, Enterprise Edition: Platform and Component Specifications*. Addison-Wesley, 2000.

[72] Henry Sowizral, Kevin Rushforth, and Michael Deering. *The Java 3D™ API Specification*. Addison-Wesley, 2nd. edition, 2000.

[73] Greg Turk. Re-tiling polygonal surfaces. *ACM Computer Graphics (SIGGRAPH 92 Conference Proceedings)*, Vol. 26, No. 2, pages 55-64, 1992.

[74] Lance Williams. Pyramidal parametrics. *ACM Computer Graphics (SIGGRAPH 83 Conference Proceedings)*, Vol. 17, No. 3, pages 1-11, 1983.

[75] Alan Watt. *3D Computer Graphics*. Addison-Wesley, 3rd. edition, 1999.

[76] Paula Womack and Jon Leech. *OpenGL® Graphics with the X Window System® (Version 1.3)*. Silicon Graphics, Inc., 1998.

[77] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL® Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley, 3rd. edition, 1999.

[78] X3D Task Group. *Extensible 3D (X3D) International Standard ISO/IEC 19775:200x*. The Web3D Consortium, Inc., 2002.