JavaGL - A 3D Graphics Library in Java for Internet Browsers

Bing-Yu Chen, Tzong-Jer Yang, and Ming Ouhyoung

Communications and Multimedia Laboratory,
Department of Computer Science and Information Engineering,
National Taiwan University, Taipei, Taiwan, R.O.C.

Abstract

This paper presents a 3D graphics library, or JavaGL[1], written in Java to provide 3D graphics capabilities over network. To make the 3D graphics library easy to learn and use, we define the application programming interface (API) in a manner quite similar to that of OpenGL, since OpenGL is a *de facto* industry standard. Furthermore, we have also developed a network library, or JavaNL[2], and combined it into JavaGL, so that a programmer can develop multi-participant 3D graphics applications easier using JavaGL and JavaNL. Implementation issues and performance evaluations are addressed.

1. Introduction

As the Internet and World Wide Web (WWW) are getting more and more popular, many Internet-based consumer electronic products, including Network Computers [1] and Web TVs, have been developed. However the Internet itself is a heterogeneous network environment, if we want to deliver WWW contents with 3D graphics information across the Internet, we will need a 3D graphics capability in each different platform. Furthermore, observing the development of the Internet, we believe that the software "pay-per-use" concept will be realized in the near future. Under this new paradigm, a 3D graphics application may be distributed from a server to a client with a different hardware architecture. Therefore, we decide to develop a 3D graphics library that needs to be platform independent, and Java is chosen as our programming language for its hardware-neutral feature.

We also notice that a multi-participant interactive environment would be a potential requirement for Internet applications, hence we developed a network library in Java, called JavaNL, to help programmers developing multi-participant applications easier.

We begin in section 2 and 3 with descriptions of some implementation issues when developing JavaGL and JavaNL, and show some results in section 4. The conclusions and future work are presented in section 5.

[1] Http://www.cmlab.csie.ntu.edu.tw/~robin/JavaGL.

[2] Http://www.cmlab.csie.ntu.edu.tw/~robin/JavaNL.

2. JavaGL - A 3D graphics library in Java

JavaGL is designed to have an API similar to that of OpenGL [2], since OpenGL is a *de facto* industry standard, and many programmers have been familiar with OpenGL's API.

The functions of OpenGL can be divided into three categories: OpenGL Utility Library (glu), OpenGL (gl), and OpenGL Extensions to native window Systems (glX or wgl), as shown in Figure 1.

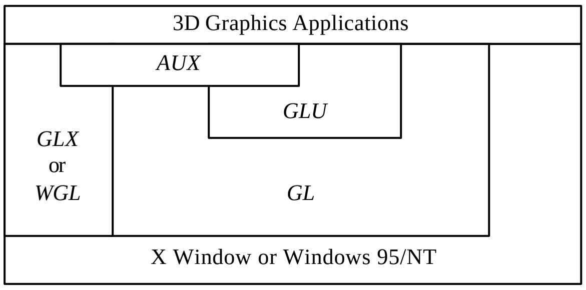


Figure 1 The hierarchy of OpenGL modules.

gl implements primitive 3D graphics operations, including rasterization, clipping, etc.; glu provides higher level OpenGL commands to programmers, and encapsulates these OpenGL commands as a series of gl functions; glX or wgl deals with function calls to native window systems.

Besides these three interfaces, there is an OpenGL Programming Guide Auxiliary Library, called aux or glaux, which is not an official OpenGL API, but is widely used. We also implement glaux in our JavaGL package.

The implementation of JavaGL is mainly based on the specifications of OpenGL [4], where the OpenGL Programming Guide Auxiliary Library (glaux) is implemented according to the definitions in the OpenGL Programming Guide [5]. We also refer to Graphics Gems for better implementation algorithms [6][7][8]. The hierarchy of JavaGL modules is shown in Figure 2.

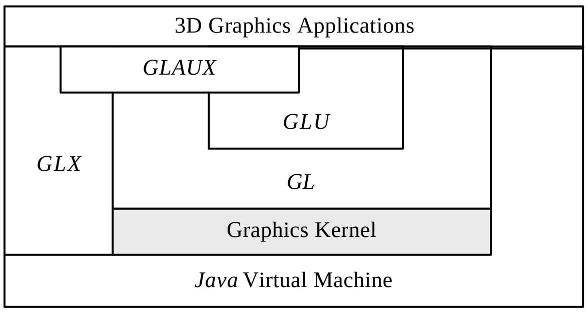


Figure 2 The hierarchy of JavaGL modules.

The graphics kernel shown in Figure 2 contains a more compact set of primitive 3D graphics operations, and is illustrated in the following section.

2.1 Implementation of JavaGL graphics kernel

The graphics kernel is transparent to programmers, which means if there is a better implementation, the graphics kernel can be substituted silently. Figure 3 shows the hierarchy of the graphics kernel, and each box represents a Java class.

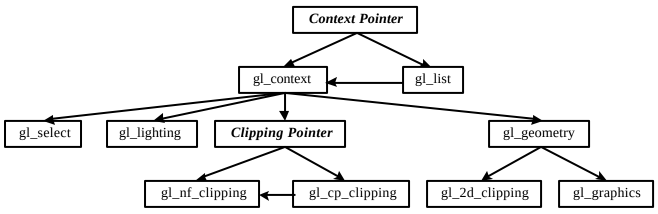


Figure 3. The hierarchy of JavaGL's graphics kernel.

When a rendering command is issued to the context pointer, the context pointer will check the state of OpenGL. If the state of OpenGL is normal, the rendering command is sent to gl_context directly; if the state of OpenGL is stalling to the display list, the rendering command is sent to gl_list. gl_list records a sequence of rendering commands, and eventually calls gl_context for rendering.

The gl_nf_clipping, gl_cp_clipping, and Clipping Pointer have the same relationship with that between gl_context, gl_list, and Context Pointer. gl_nf_clipping is the clipping class for near and far clipping planes, while gl_cp_clipping is the clipping class for user defined clipping planes.

The other classes are gl_select for selection, gl_lighting for lighting calculation, gl_geometry for drawing all kinds of geometric objects, gl_2d_clipping for 2D clipping functions, and gl_graphics is the lowest level of drawing functions of the graphics kernel.

2.2 Performance enhancement issues

Performance is a great challenge for both 3D graphics and Java, hence a great challenge for JavaGL. Moreover, JavaGL is designed to operate over the Internet, where network bandwidth affects the overall performance significantly. These considerations make the implementation of JavaGL complex.

According to our experiences, we develop the following design philosophies to speed up JavaGL's performance.

1. **Utilize class inheritance to avoid "if-then-else" statements** – OpenGL is a state machine, and it's usually necessary to determine if some status is enable, which takes time to check. We utilize class inheritance to avoid these frequent checks. After deciding which status is enable, we cast an object to its proper class type, and the following rendering commands will be routed to proper functions automatically without any further checks.

   Use the implement of the display list as an example. When implementing the display list, we set a flag to indicate whether the rendering commands are to be stored in a display list or to be executed immediately.

   In the case of "if-then-else," for each rendering command, we need to check if the flag of the display list is set or not using many "if-then-else" statements, and these many "if-then-else" statements will slow down the execution speed.

   In the case of "class inheritance," each rendering command has two class implementations, one with the display list, and the other without the display list. Both classes are inherited from the same parent class. After the flag of the display list is checked for the first time, all the following rendering commands are realized automatically without any further checks of the flag.

2. **Make frequently used routines faster** – Polygon rasterization, shading, depth testing, clipping, etc., are frequently used routines. These routines are always bottlenecks for 3D graphics libraries, so we put our most efforts on optimizing these routines with faster algorithms and manual code optimization.

3. **Divide frequently used routines into smaller ones** – For a frequently used routine, we would like several smaller and simple ones, rather than a larger but powerful one. The purpose is to reduce unnecessary network transmissions for unused code segments.

   For example, to fill a polygon, we must do the color interpolation if the polygon is filled by smooth shading. If the polygon only needs a flat shading, the color interpolation is not required, and does not need to be transmitted. Therefore, we categorize all the drawing functions into several smaller ones, such as drawing functions with or without depth testing, drawing functions with flat shading or smooth shading, etc., and optimize these functions.

4. **Group rarely used routines into a larger one** – When we divide frequently used routines into smaller ones, the total code size of the JavaGL library will increase. A large size of file will increase the overhead for network transmission. To reduce the total code size of the JavaGL library, we re-examine all routines, and combine some similar routines that are rarely used into a larger one, contrarily.

   For example, we had two routines for rendering, including one with clipping and the other one without clipping originally. Since the former routine is mostly used, we combine these two routines, and optimize the conditional testing to redirect a rendering command to an appropriate code segment efficiently.

3. JavaNL - A network library in Java

In our experiences, an Internet application will be more attractive if it provides several participants to interact with each other.

JavaNL, a multi-participant interactive network library, is developed to remove most of the programming burdens on maintaining multi-participant interactions over the Internet.

The JavaNL adopts the concepts of Distributed Interactive Simulation (DIS) [9][10][11] with some modifications. DIS is originally designed for military exercise simulations over WAN (Wide-Area Network), and takes multi-participant interactions into account, hence we chose DIS as our design principles of JavaNL.

3.1 DIS vs. JavaNL

DIS is a set of IEEE standards including IEEE Std 1278.1-1995 [9][10] for application protocols and IEEE Std 1278.2-1995 [11] for communication services and profiles. IEEE P1278.3 is for exercise management and feedback, and has not been standardized so far.

DIS defines a large set of data types for communications, and we use a subset of the data types to develop JavaNL. The principles of JavaNL complying DIS are listed as the following, where a simulation entity represents a data unit with some data type.

1. There is no central computer that controls the entire simulation.
2. Autonomous simulation applications are responsible for maintaining the state of one or more simulation entities.
3. Changes in the state of an entity are communicated by its controlling simulation application.
4. Perception of events of other entities is determined by the receiving application.

In DIS, each application uses PDUs (Protocol Data Units) to communicate with each other, and keeps all simulation information locally, as shown in Figure 4.

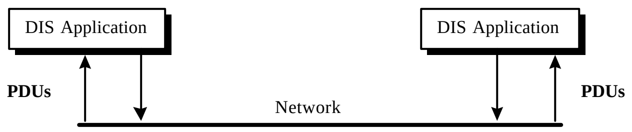


Figure 4 The control flow of DIS. A DIS application needs to maintain all the simulation information necessary, and uses PDUs to communicate with each other.

JavaNL modifies some PDUs' formats, and the detailed PDU formats can refer to [18]. In general, an application can call JavaNL's functions to send and receive data, and the multi-participant simulation is automatically maintained by JavaNL. Using JavaNL, an application needs not to implement the complex DIS, but instead of a simple set of function calls. The modified control flow of JavaNL is shown in Figure 5.

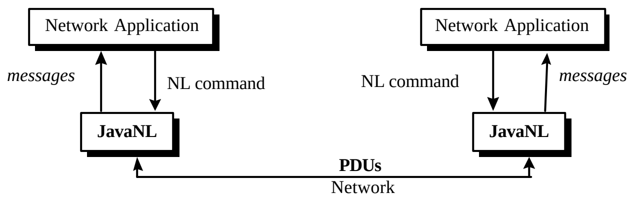


Figure 5 The control flow of JavaNL that provides PDU transmission capability.

3.2 The control flow of PDUs between applications

Four additional PDUs, Join Request, Join Accept, Join Reject, and Disconnect, are defined for JavaNL only. These four additional PDUs are used in communication with a simulation manager. When a simulation application creates a simulation, it becomes a simulation manager, and waits for other simulation applications to join. If there is a simulation application that wants to join the simulation, it sends a Join Request PDU to the simulation manager. If the simulation manager agrees the request, it sends a Join Accept PDU with all the simulation information to the simulation application that requests to join; if the simulation manager denies the request, it sends a Join Reject PDU to the simulation application that requests to join. The whole process is shown in Figure 6.

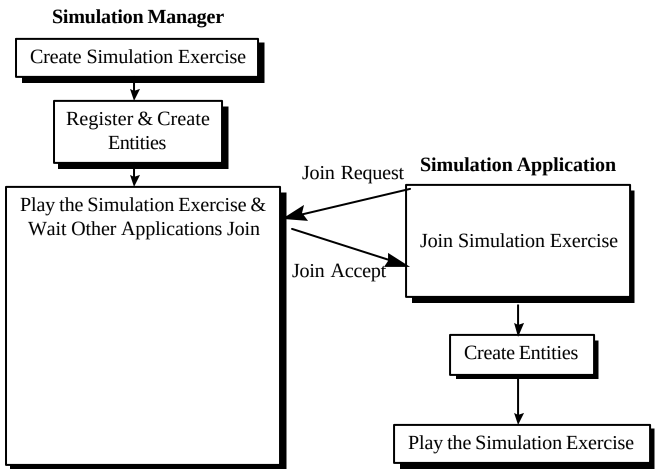

Simulation Manager
Create Simulation Exercise
Register & Create Entities
Play the Simulation Exercise & Wait Other Applications Join
Join Request
Join Accept
Simulation Application
Join Simulation Exercise
Create Entities
Play the Simulation Exercise

Figure 6 The control flow of PDUs in JavaNL.

A simulation manager is necessary when a simulation is to be created, or when an application wants to join the current simulation. Besides the above situations, the simulation manager behaves like other simulation applications, and all simulation information packed into PDUs are exchanged between all simulation applications automatically.

3.3 The control flow of PDUs in JavaNL

JavaNL provides applications a simple interface to access PDUs from the network. When an application uses JavaNL to create a client or server thread, another thread, or nl_network_agent, is created automatically. The nl_network_agent maintains several PDU queues. PDUInQueue stores PDUs received from the network; PDUOutQueue stores PDUs to be sent out from the application; MSGQueue holds messages to inform the application that there are events or PDUs to handle. The control flow of PDUs is shown in Figure 7.

If an application wants to communicate with other applications, it calls JavaNL functions to write PDUs to PDUOutQueue. The nl_network_agent constantly polls the PDUOutQueue, and if there are PDUs in the queue, it will call nl_udp_sender or nl_tcp_sender to send the PDUs out.

If a PDU arrives, it will be received by nl_tcp_receiver or nl_udp_receiver, and will be buffered in PDUInQueue. The nl_network_agent constantly polls the PDUInQueue, and if there are PDUs in the queue, it will write a message to MSGQueue to inform the application, or will process the PDUs locally. The application needs to poll the MSGQueue via JavaNL functions, and retrieves PDUs if necessary.

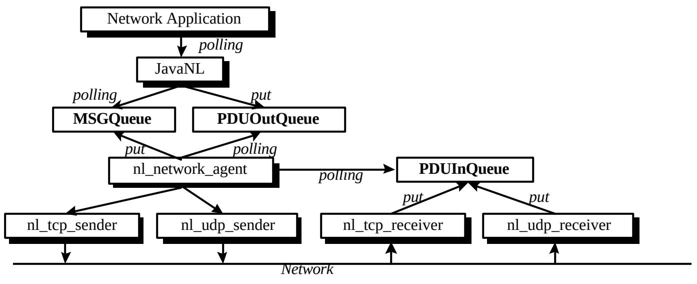


Figure 7 PDU sending and receiving in JavaNL.

4. Results

Currently, we have implemented over **160** OpenGL functions in JavaGL, including functions of GLAUX, GLU, and GL. The functionality provided contains functions for 2D/3D model transformation, 3D object projection, depth buffer, smooth shading, lighting, material, display list and selection. Functions not supported so far are mainly for anti-aliasing and texture mapping. In the future, OpenGL Utility Toolkit (GLUT) [12] using JavaGL will be provided, too.

We also provide 16 examples on WWW. These examples are selected from the OpenGL Programming Guide [5], and can be executed directly in Internet browsers supporting Java. Figure 8 shows a simple Java applet that draws a rectangle using JavaGL.

To evaluate JavaGL's performance, we use a testing program that renders 12 spheres with different materials, and each sphere contains **256** polygons, as shown in Figure 9 The performance of the testing program is measured on a SUN Ultra-1 workstation and an Intel Pentium-200 PC. For comparison, we also rewrote the same program with Mesa 3-D graphics library [13], that is a software-based 3D graphics library with an API similar to that of OpenGL using C programming language, and measured the rendering time. We also rewrote the same program with hardware accelerated OpenGL on both platforms. The performance comparisons are listed in

Table 1 and Table 2.

On the SUN workstation, the testing program with Mesa is about **4** times faster than that with JavaGL, which is better than the performance claimed by SUN that Java is about 20 times slower than C [14]. The performance can be further improved if a better Java interpreter or compiler exists.

On the PC platform, we execute the testing program using the SUN JDK 1.0.2 [15] and the Symantec Café 1.51 [16] with JIT 2.0 beta 3. By using the Just-In-Time (JIT) [17] compiler, we obtain an over 4 times performance speedup.

```
import java.applet.Applet;
import java.awt.*;

// must import packages of JavaGL.
import javagl.GL;
import javagl.GLAUX;

public class simple extends Applet
{
      GL  myGL = new GL();
      GLAUX   myAUX = new GLAUX(myGL);

      public void init()
      {
         myAUX.auxInitPosition(0,   0,   500,
500);
         myAUX.auxInitWindow(this);
      }

     public void paint(Graphics g)
      {
         //     JavaGL     only     supports
double-buffer.
         myGL.glXSwapBuffers(g, this);
      }

     public void start()
      {
         myGL.glClearColor((float)0.0,
(float)0.0,

(float)0.0, (float)0.0);

myGL.glClear(GL.GL_COLOR_BUFFER_BIT);
         myGL.glColor3f((float)1.0,
(float)1.0,

(float)1.0);

myGL.glMatrixMOde(GL.GL_PROJECTION);
         myGL.glLoadIdentity();
         myGL.glOrtho((float)-1.0,
(float)1.0,
                                (float)-1.0,
(float)1.0,
                                (float)-1.0,
(float)1.0);
         myGL.glBegin(GL.GL_POLYGON);
            myGL.glVertex2f((float)-0.5,
(float)-0.5);
            myGL.glVertex2f((float)-0.5,
(float)0.5);
            myGL.glVertex2f((float)0.5,
(float)0.5);
            myGL.glVertex2f((float)0.5,
(float)-0.5);
         myGL.glEnd();
         myGL.glFlush();
      }
}
```

Figure 8. A simple Java applet that draws a rectangle using JavaGL.

Graphics Library	Environment	Rendering Time (ms)
JavaGL 1.0 beta 3	SUN JDK 1.0.2 SUN JIT 1.0.2	**4984**
Mesa 2.1	GNU C 2.7.2.1	**1085**
OpenGL for Creator3D 1.0	GNU C 2.7.2.1 Hardware accelerated (Sun Creator3D)	**138**

Table 1 A performance comparison on a workstation. The workstation configuration is SUN Ultra-1 Model 170E, 128 MB memory, 24-bit display, Sun Solaris 2.5.1.

Graphics Library	Environment	Rendering Time (ms)
JavaGL 1.0 beta 3	Sun JDK 1.0.2	**16700**
JavaGL 1.0 beta 3	Symantec Café 1.51 Symantec JIT 2.0 beta 3	**4070**
OpenGL for Windows 95 1.0	Microsoft Visual C++ 4.2 Hardware accelerated (ET-6000)	**189**

Table 2 A performance comparison on a PC. The PC configuration is Intel Pentium-200 CPU, 64 MB memory, 24-bit display, Microsoft Windows 95.

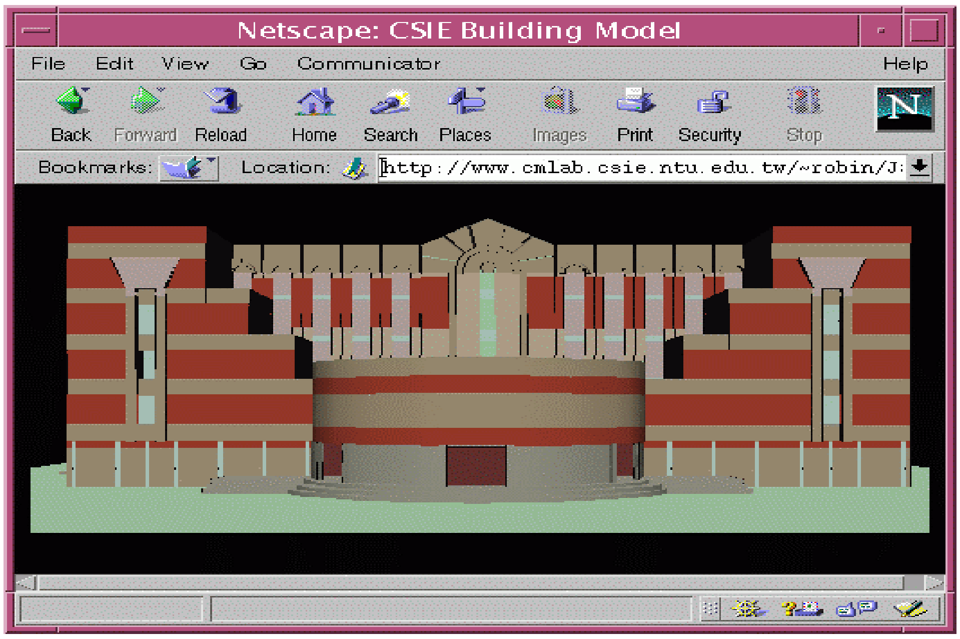


Figure 10. Our department building rendered with JavaGL on Netscape Navigator 4.0pr2. This model contains **5273** triangles and takes **6150 ms** on a PC with Intel Pentium-200 CPU and 64 MB memory.

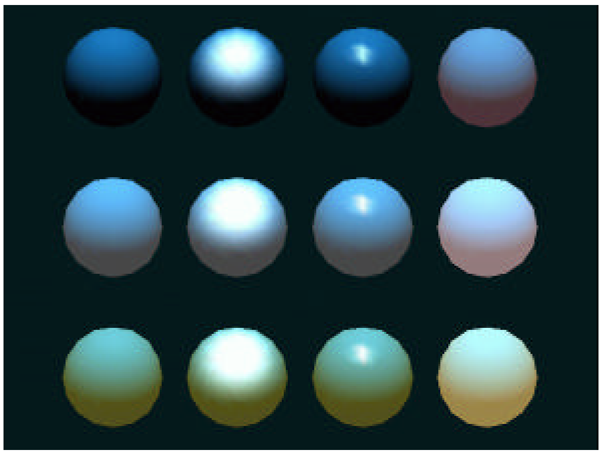


Figure 9 Twelve spheres are rendered to measure performance. Each sphere contains **256** polygons. This program is an example in OpenGL Programming Guide [5] (code from Listing 6-3, pp. 183-184, Plate 16). This figure is rendered with JavaGL.

Figure 10 is a complex model that contains **5273** triangles, and the rendering time is **6150 ms** on an Intel Pentium-200 PC with 64MB memory. The complex model is rendered by an applet running on a Netscape web browser, where all the 3D graphics functions are obtained directly from a server.

To demonstrate the usage of JavaNL, we developed a multi-participant building walkthrough application, as shown in Figure 11. The multi-participant building walkthrough application renders a building model, and allows multi-participants interacting with each other in a Local Area Network (LAN) environment. The system hierarchy is shown in Figure 12. In this application, participants are represented as cubes, and if one participant changes his position, other participants will notice a position change of a cube. The performance is listed in Table 3.

We also compare the round trip time of a PDU with an UDP packet, and the result is listed in Table 4. Java introduces a little more overhead when sending the same UDP packet, and JavaNL needs more time because JavaNL has to pack information into a PDU.

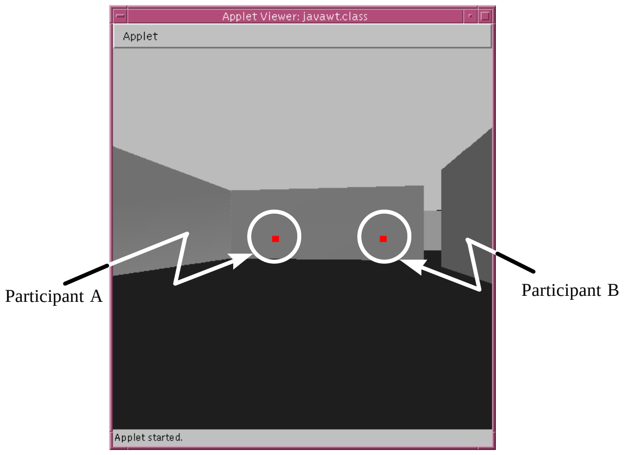


Figure 11. A multi-participant building walkthrough application. There are 3 participants in the environment currently, and this figure shows one participant's view. The other 2 participants are represented by cubes.

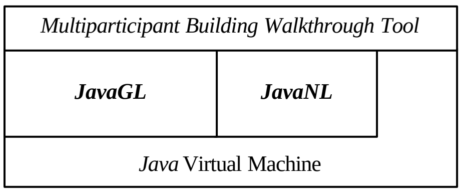


Figure 12. The system hierarchy of a multi-participant building walkthrough application using JavaGL and JavaNL.

Platform	**Workstation**	**PC**
Refresh Time (ms)	**230**	**130**
Refresh Rate (frames/sec)	**4.3**	**7.7**
Environment	SUN Ultra-1 170E 128 MB memory 24-bit display (Creator 3D) SUN Solaris 2.5.1 10 Base 2 Ethernet	Intel Pentium-200 64 MB memory 24-bit display (ET 6000) Microsoft Windows 95 10 Base T Ethernet
Interpreter	SUN JDK 1.0.2 SUN JIT 1.0.2	Symantec Café 1.51 Symantec JIT 2.0 beta 3

Table 3. Performance of a multi-participant building walkthrough application. The model used contains 84 triangles, and one cube representing one participant takes additional 12 triangles. The total number of triangles rendered is 120 triangles.

Round trip time of	PDU in JavaNL	UDP packet in Java	UDP packet in C
Time (ms)	**338**	**4**	**1**

Table 4. The round trip time of different packets. This evaluation is measured by sending a packet to another host and receiving the packet from the host. The packet is of length 192 bytes. Note that JavaNL needs time to pack information into a PDU.

5. Conclusions and Future Work

Since we upload JavaGL to our web server, there have been over **1000** people around the world visit our web page. We also received dozens of e-mails concerning the use of JavaGL. Some would like to collaborate with us, and some want to use JavaGL to develop their applications. This encourages us to further improve JavaGL and JavaNL.

JavaGL is being applied to develop a Java-based VRML 2.0 browser in our laboratory. The goal of this VRML browser is to provide users all the necessary functions from servers so that users do not have to install additional hardware or software for 3D graphics applications. JavaGL meets this requirement because it's implemented purely by Java that is designed for Internet.

Using JavaNL to develop a multi-participant interactive application is much easier than before. To add a chat function in the multi-participant building walkthrough application, we only take less than 10 minutes to finish this work with JavaNL.

Performance is a great challenge for any Java applications. We expect that the performance will be improved by better Java interpreters and Java compilers, and will be greatly improved by new Java chips and faster CPUs.

All the demo codes and examples are available in our web site at Http://www.cmlab.csie.ntu.edu.tw/~robin/JavaGL, and visitors are welcome.

Acknowledgments

This work is a part of the Multimedia Digital Classroom (MDC) project developed at the Communications and Multimedia Laboratory, National Taiwan University. The MDC project is sponsored by National Science Council (NSC) under the grant NSC 85-2622-E-002-015.

References

[1] "*Network Computer*," Network Computer, Inc., 1997. Http://www.nc.com.

[2] "*OpenGL WWW Center*," Silicon Graphics, Inc., 1997. Http://www.sgi.com/Technology/openGL.

[3] "*Distributed Interactive Simulation*," Institute for Simulation and Training, University of Central Florida, 1997. Http://www.ist.ucf.edu/labsproj/projects/dis.htm.

[4] Mark Segal, and Kurt Akeley, "*The OpenGL Graphics Systems: A Specification (Version 1.1)*," Silicon Graphics, Inc., 1996. Http://www.sgi.com/Technology/ openGL/glspec/glspec.html.

[5] Jackie Neider, Tom Davis, and Mason Woo, "*OpenGL Programming Guide*," Addison-Wesley, 1993.

[6] Andrew S. Glassner, "*Graphics Gems*," Academic Press, Inc., 1990.

[7] James Arvo, "*Graphics Gems II*," Academic Press, Inc., 1991.

[8] David Kirk, "*Graphics Gems III*," Academic Press, Inc., 1992.

[9] "*IEEE Standard for Distributed Interactive Simulation – Application Protocols (IEEE Std 1278.1-1995)*," Institute of Electrical and Electronics Engineers, 1996.

[10] "*Enumeration and Bit-encoded Values for Use with IEEE Std 1278.1-1995, Standard for Distributed Interactive Simulation – Application Protocols*," Institute for Simulation and Training, University of Central Florida, 1996. Http://ftp.sc.ist.ucf.edu/SISO/ dis/library/enumerat.doc.

[11] "*IEEE Standard for Distributed Interactive Simulation – Communication Services and Profiles (IEEE Std 1278.2-1995)*," Institute of Electrical and Electronics Engineers, 1996.

[12] Mark J. Kilgard, "*Graphics Library Utility Toolkit*," Silicon Graphics, Inc., 1996. Http://www.sgi.com/ Technology/openGL/glut.html.

[13] Brian Paul, "*The Mesa 3-D Graphics Library*," 1997. Http://www.ssec.wisc.edu/~brianp/Mesa.html.

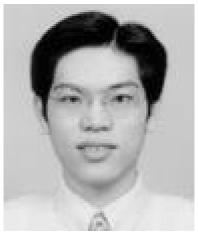
[14] Arthur van Hoff, Sami Shaio, and Orca Starbuck, "*Hooked on Java*," Addison-Wesley, 1996.

[15] "*The Java Developers Kit Version 1.0.2*," Sun Microsystems, Inc., 1996. Http://www.javasoft.com/ products/jdk/1.0.2.

[16] "Symantec *Café*," Symantec, Co., 1997. Http:// www.symantec.com/cafe.

[17] "*The JIT Compiler Interface Specification*," Sun Microsystems, Inc., 1996. Http://www.javasoft.com/ doc/jit_interface.html.

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



Bing-Yu Chen received the BS and MS degree in Computer Science and Information Engineering from the National Taiwan University, Taipei, in 1995 and 1997, respectively. He is currently a research assistant in Communications and Multimedia Laboratory at the National Taiwan University. His research interests include computer human interface, computer graphics, virtual reality, Java programming language, and Intemet technologies.

Tzong-Jer Yang received the BS degree in the Mathematics from the National Tsing-Hua University, Hsin-Chu, in 1992. He received the MS degree in the Computer Science and Information Engineering from the National Taiwan University, Taipei, in 1994. He is now a Ph.D. candidate there. His research interests include computer graphics, virtual reality, and Internet technologies.

Ming Ouhyoung received the BS and MS degree in Electrical Engineering from the National Taiwan University, Taipei, in 1981 and 1985, respectively. He received the Ph.D. degree in Computer Science from the University of North Carolina at Chapel Hill in 1990. He was a member of the technical staff at AT&T Bell Laboratories, middle-town, during 1990 and 1991. Since August 1991, he has been an associate professor in the Department of Computer Science and Information Engineering at the National Taiwan University and later became a professor in 1995. He has published 87 technical papers on consumer electronics , computer graphics, virtual reality and multimedia system. He is a member of ACM and IEEE.