A JAVA-BASED MULTI-PARTICIPANT 3D GRAPHICS APPLICATION INTERFACE USING JAVAGL[1] AND JAVANL[2]

Bing-Yu Chen and Ming Ouhyoung

Communications and Multimedia Laboratory,
Department of Computer Science and Information Engineering,
National Taiwan University, Taipei, Taiwan, ROC
Email:{robin,ming}@csie.ntu.edu.tw

ABSTRACT

This paper proposes a Java-based multi-participant 3D graphics application interface to provide 3D graphics and interactive capabilities over network. Two libraries support this interface, one is a 3D graphics library, called JavaGL, and the other is a network library, called JavaNL. The 3D graphics library is almost identical to OpenGL application interface but is written in Java to satisfy the 3D graphics requirement of the application model. The network library follows the concepts of Distributed Interactive Simulation (DIS), and is also written in Java. With these two libraries, we are able to provide an interactive application written in Java, gains the flexibility in cross-platform capability, while maintaining reasonable performance over Local Area Network (LAN).

1. INTRODUCTION

From the experience of developing 3D graphics and Virtual Reality (VR) projects on a stand-alone computer, we believe that after the 3D graphics and VR applications shift to the Internet, multi-participant 3D graphics and VR applications will be the next stream on Internet.

As Internet, World Wide Web (WWW), 3D Graphics and VR are getting more and more popular, there is increasing demand of 3D graphics over network. Because the Internet itself is a heterogeneous network environment, if we want to deliver WWW contents with 3D information, we need 3D graphics capability in each different platform. Observing the development of 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 in different platforms. Therefore, we decide to develop a 3D graphics library that is platform independent. Java [1] is chosen as our programming language for its hardware-neutral feature, and its wide availability on many hardware platforms, including embedded systems.

At the same time, it is desired that this 3D graphics library be easy to learn, so we define the Application Programming Interface (API) in a manner quite similar to that of OpenGL [2] , since OpenGL is a *de facto* industry standard, and many programmers have been familiar with OpenGL's API.

We also notice that a multi-participant interactive environment would be a idea for Internet applications. Hence, we have also developed a Java network library to help programmers to develop multi-participant applications easier, and followed the concepts of Distributed Interactive Simulation (DIS) [3] , since the DIS is a standard for the interactive simulation on the Internet.

To test the above idea, we have built a multi-participant building walk-through application to demonstrate its feasibility. Users who are on different machines while traversing the same building model can see each other as a symbol in the building and interact with each other.

2. JAVAGL - A 3D GRAPHICS LIBRARY IN JAVA

2.1. OpenGL vs. JavaGL

Functions of OpenGL can be divided into three categories: OpenGL Utility Library (GLU), OpenGL (GL), and the OpenGL Extension to native window systems (GLX or WGL), as shown in Figure 1(a). JavaGL follows the same function hierarchy as shown in Figure 1(b).

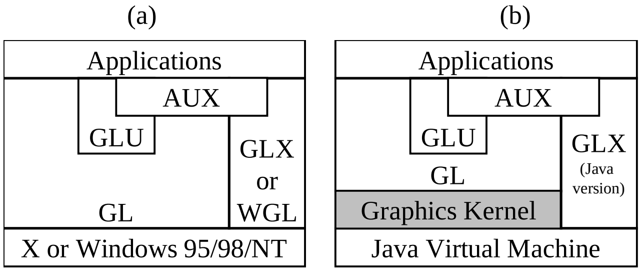


Figure 1 (a) The hierarchy of OpenGL modules. (b) The hierarchy of JavaGL modules.

GL implements a powerful but small set of drawing primitive 3D graphics operations, including rasterization, clipping, etc. GLU provides higher level OpenGL commands to programmers by encapsulating these OpenGL commands with a series of GL functions. GLX or WGL deals with function calls to native window systems and are implemented depending on each different platform.

1 Http://www.cmlab.csie.ntu.edu.tw/~robin/JavaGL
2 Http://www.cmlab.csie.ntu.edu.tw/~robin/JavaNL

Besides these three interfaces, there is an OpenGL Programming Guide Auxiliary Library, called AUX or GLAUX, which is not an official part of OpenGL API, but is widely used and familiar for the programmers. For this reason, we also include GLAUX in our JavaGL package.

The implementation of JavaGL is mainly based on the specifications of OpenGL [4] , while the GLAUX library is implemented according to OpenGL Programming Guide [5] . We also refer to Graphics Gems [6,7,8] for better implementation algorithms. Besides GL, GLU, GLX and AUX, there is an underlying graphics kernel which is transparent to programmers.

2.2. Implementation of JavaGL Graphics Kernel

We collect atomic 3D graphics routines into a graphics kernel, and its class hierarchy is shown in Figure 2.

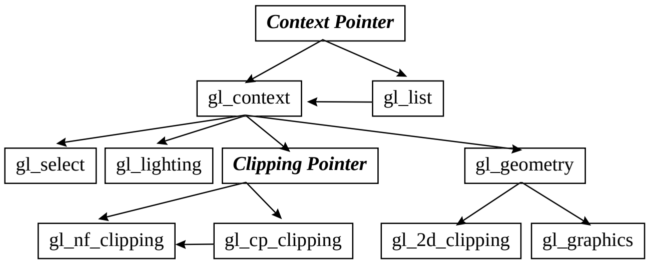


Figure 2 The hierarchy of JavaGL's graphics kernel.

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.

gl_context routes drawing commands into correct classes to complete drawing operations. These classes are gl_select for selection, gl_lighting for lighting calculation, Clipping Pointer for clipping, and gl_geometry for drawing all kinds of geometric objects, where gl_graphics is the lowest level of drawing functions in the graphics kernel.

2.3. Performance Enhancement Issues

Performance is a great challenge for both 3D graphics and Java applications, hence is also 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 philosophies to speed up JavaGL's performance.

1. **Make frequently used routines faster** – Polygon rasterization, shading, depth testing, clipping, etc., are frequently used routines, and optimized with faster algorithms and manual code optimization. This is a common but most useful strategy.

2. **Utilize class inheritance to avoid "if-then-else" statements** – OpenGL is a state machine, and it is usually necessary to determine if some status is enabled or not, which takes time to check. We utilize class inheritance to avoid these frequent checks. After deciding which status is enabled, we realize an object to its proper class type, so following rendering commands will be routed to proper functions automatically without any further checks.

   Take OpenGL's display list as an example. In a traditional "if-then-else" implementation, we might have the following code:

```
JavaGL_routine()
{
  if (use_display_list)
        render_with_display_list();
  else
        render_without_display_list();
}
```

   The "if-then-else" check takes time, and it is often to have a rendering routine called for a thousands times in a typical graphics application.

   In the case of "class inheritance" implementation, each rendering command will have two implementations, where one is to render with display list, and the other is to render without display list. Both classes are inherited from the same parent class. Once the flag of the display list is checked for the first time, a corresponding rendering object is realized, and all following rendering commands will be executed correctly without any further checks on the flag.

3. **Divide a routine into several smaller ones** – If a routine was very large and would be called in several situations, this routine must have some useless code segments for other situations, while it is just called for a simple situation. So, it is worth to divide this routine into several smaller ones.

   For example, to fill a polygon, we must do color interpolation if the polygon is to be filled using smooth shading. However if the polygon only requires flat shading, color interpolation would be unnecessary. Therefore, we divide the shading routines into two smaller ones.

4. **Group several routines into a larger one** – Contrarily, if two routines are similar, we combine these two routines into a larger one if the code size of the combined routine would much less than the total code size of the two original routines. The purpose is to reduce network transmission time.

   For example, we had two rendering routines with or without clipping originally. Since the second one is much simpler, we combine these two routines, and optimize the conditional test to redirect a rendering command to an appropriate code segment efficiently.

As the above considerations, if we divided a routine into several smaller ones, the code size of the library will be large. If we grouped several routines into a larger one, the performance will be worse. As an interactive program, the performance is important, but as an Internet program, the size is also important for the network transmission. So, the size and the performance is the trade-off when we develop JavaGL, since JavaGL offers interactive capability on the Internet.

3. JAVANL - A NETWORK LIBRARY IN JAVA

3.1. DIS vs. JavaNL

In our experiences, an Internet application will be more attractive if it provides several participants to interact with one another simultaneously. JavaNL, a multi-participant network library, is developed to fit the purpose.

JavaNL adopts some concepts of DIS with some modifications. DIS is originally designed for military exercise simulations over Wide-Area Network (WAN), and takes multi-participant interactions into account, hence we chose DIS as our design principles of 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 yet been standardized.

DIS defines a large set of data types for communications, and we use only a subset to develop our JavaNL. The principles of JavaNL complying DIS are listed in 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 Protocol Data Units (PDUs) to communicate with each other, and keeps all simulation information locally, as shown in Figure 3(a).

JavaNL modifies some PDUs' formats, and the detailed PDU formats can refer to [12]. 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 3(b).

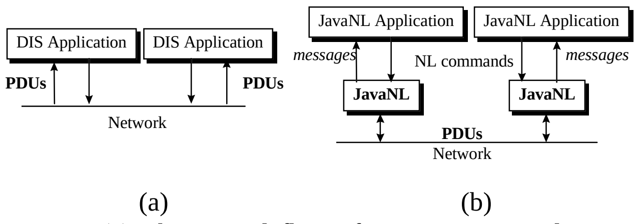


Figure 3 (a) The control flow of DIS. A DIS application needs to maintain all the simulation information necessary, and uses PDUs to communicate with each other. (b) The control flow of JavaNL that provides PDU transmission capability.

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 4.

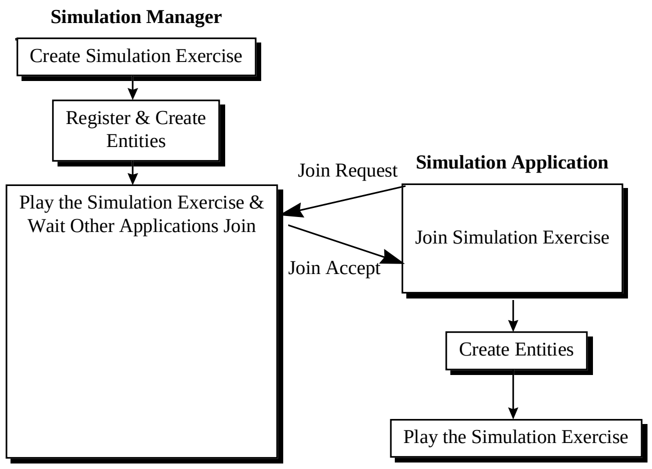

Simulation Manager
Create Simulation Exercise
Register & Create Entities
Play the Simulation Exercise & Wait Other Applications Join
Join Request
Join Accept
Simulation Application
Join Simulation Exercise
Create Entities
Play the Simulation Exercise

Figure 4 The control flow of PDUs in JavaNL.

A simulation manager is only needed when a simulation is to be created, or when an application wants to join the current simulation. Besides the above situations, a 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 5.

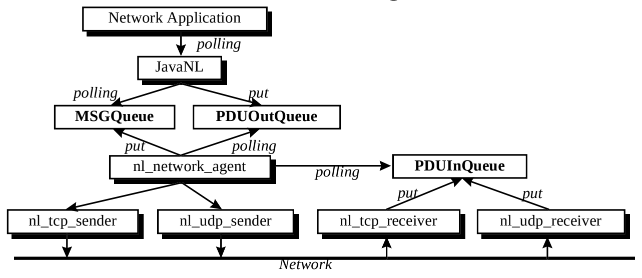


Figure 5 PDU sending and receiving in JavaNL.

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. It is our goal to design a network agent of network applications and offer a simple API to make programmers develop the network applications as easy as possible.

4. RESULTS

4.1. Performance Testing and Results of JavaGL

Currently, we have implemented over **180** OpenGL functions in JavaGL, including functions of GLAUX, GLU, and GL. These functions include 2D/3D transformation, 3D projection, depth buffer, smooth shading, lighting, material, display list, selection, etc. Functions not supported so far are mainly for anti-aliasing and texture mapping. To provide the *OpenGL Utility Toolkit* (GLUT) [13] using JavaGL is also our future work. Figure 6 shows a simple Java applet that draws a rectangle using JavaGL.

```
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 6 A simple Java applet that draws a rectangle using JavaGL.

To test JavaGL's capability, we have provided **16** examples on our JavaGL web page[3]. These examples are selected from the *OpenGL Programming Guide*, which is an official programming guide of OpenGL, and can be executed directly in Internet browsers supporting Java.

To evaluate JavaGL's performance, we used a test program that renders twelve spheres with different materials, where each sphere contains **256** polygons, as shown in Figure 7. The performance of the test program was measured on both a SUN Ultra-1 workstation and an Intel Pentium-200 PC. For comparison, we also rewrote the same program using C programming language with standard OpenGL APIs and compiled with Mesa 3-D graphics library [14] , which is a software-based OpenGL like graphics library. We also compiled the C version with hardware accelerated OpenGL on both platforms. The results are listed in Table 1 and Table 2.

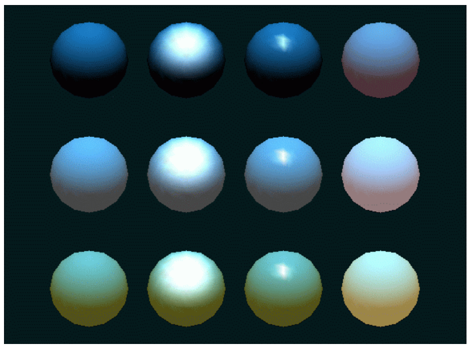


Figure 7 Twelve spheres are rendered to measure performance. Each sphere contains **256** polygons and has different material. This

3 Http://www.cmlab.csie.ntu.edu.tw/~robin/JavaGL/ Example.html

program is an example in *OpenGL Programming Guide* (code from Listing 6-3, pp. 183-184, Plate 16). This figure is rendered with JavaGL.

Table 1 lists the result measured on a SUN Ultra-1 workstation. JavaGL is about **4** times slower than Mesa, which is better than the performance claimed by SUN that Java is about 20 times slower than C [15,16].

Table 2 lists the result measured on a Pentium-200 PC. The test program using SUN JDK 1.0.2 [17] is **4** times slower than the one using Symantec Café 1.51 [18] with the support of Just-In-Time (JIT) [19] compiler.

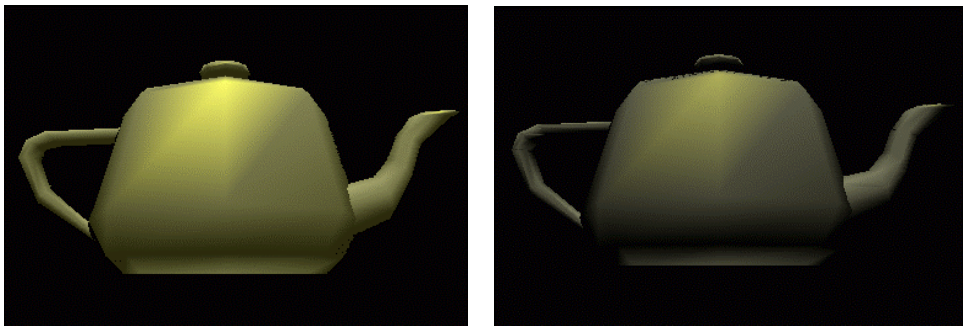
Graphics Library	Environment	Rendering Time (ms)
JavaGL 1.0 beta 3	SUN JDK 1.0.2 SUN JIT 1.0.2	**4984**
Mesa 2.1	GNU C 2.7.2.1	**1085**
OpenGL for Creator3D 1.0	GNU C 2.7.2.1 Hardware accelerated (Creator 3D)	**138**

Table 1 A performance comparison on a workstation. The workstation configuration is SUN Ultra-1 Model 170E, 128MB memory, Creator3D accelerated display card, 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 VC++ 4.2 Hardware accelerated (ET-6000)	**189**

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

Figure 8 is the quality comparisons between JavaGL and Mesa 3-D. There is a teapot which contains **604** triangles and is rendered by JavaGL and Mesa 3-D. It takes **550ms** for JavaGL on an Intel Pentium-200 PC as in Table 2.



(a) (b)

Figure 8 The teapot rendered (a) with *JavaGL* (**550ms**), and (b) with Mesa 3-D (**122ms**).

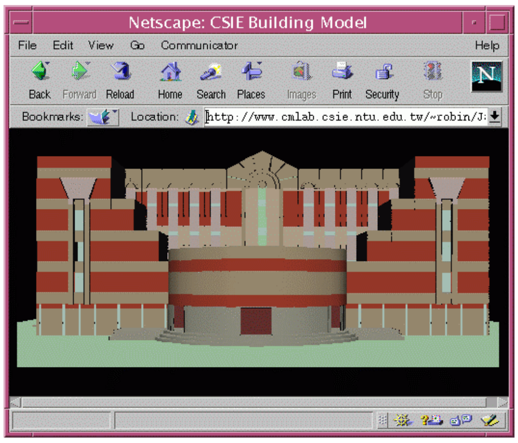

Netscape: CSIE Building Model
File Edit View Go Communicator Help
Back Forward Reload Home Search Places Images Print Security Stop
Bookmarks: Location: http://www.cmlab.csie.ntu.edu.tw/~robin/J

Figure 9 A complicated example rendered by JavaGL on Netscape Communicator 4.0pr2. This is the model of our department building which contains **5273** triangles and takes **6150ms** on a PC as shown in Table 2.

We also applied JavaGL to render complex models. Figure 9 shows our department building containing 5273 triangles, and the rendering time is **6150ms** on an Intel Pentium-200 PC with **64MB** memory. The department building is rendered by an applet using JavaGL, and all 3D graphics functions are obtained from a web server.

4.2. Performance Testing and Results of JavaNL

To test the performance of JavaNL, we have written two applications using JavaNL. One is put on the server side, and the other is put on the client site. When the server starts up, all it has to do is wait for others to join. When there is one client program joining the server, it will send a PDU which contains two integer numbers in it to the client site. When the client program runs, it will join the server and ready to receive the PDU, after receiving the PDU, it will send the same PDU returned to the server, then we can calculate the performance of JavaNL.

We measured the round-trip time of JavaNL PDU, Java UDP, and C UDP, and the results are listed in Table 3. Java introduces a little more overhead when sending the same UDP packet, and Java NL needs more time because JavaNL has to pack information into a PDU.

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

Table 3 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 192bytes. Note that JavaNL needs time to pack information into a PDU.

4.3. A Multi-participant Building Walk-through Application

To demonstrate the usage of JavaGL and JavaNL, we wrote a multi-participant building walk-through application that allows multi-participants interacting with one another in a LAN environment, as shown in Figure 10. The multi-participant building walk-through application is developed using Java programming language, the 3D graphics rendering work is done by JavaGL and the network transmission work is done by JavaNL, so this application can run on any Java enabled platforms.

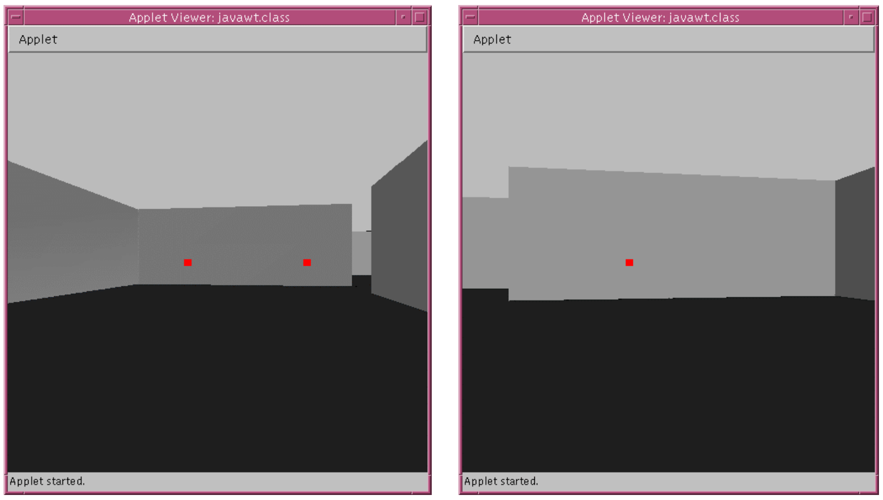


Figure 10 A multi-participant building walk-through application. There are three participants in the environment currently, and this figure shows one participant's view. The other two participants are represented by cubes.

In this walk-through 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 4.

Platform	**Workstation**	**PC**
Refresh Time (ms)	**230**	**130**
Refresh Rate (fps)	**4.3**	**7.7**
Environment	SUN Ultra-1 170E 128 MB memory Creator 3D SUN Solaris 2.5.1 10 Base 2 Ethernet	Intel Pentium-200 64 MB memory ET 6000 MS-Windows 95 10 Base T Ethernet
Interpreter	SUN JDK 1.0.2 SUN JIT 1.0.2	Symantec Café 1.51 Symantec JIT 2.0b3

Table 4 Performance of a multi-participant building walk-through 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.

If there is a participant changes his position, the participant on the other side will be notified after about **250ms**. The delay is due to the refresh rate and the network latency.

5. CONCLUSIONS AND FUTURE WORK

Since we upload JavaGL to our web server, there has been over **2000** people around the world visited 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.

SUN also revealed its Java3D [20] in 1997. Because Java3D is part of Java core packages, it can benefit from hardware acceleration, though this will need many efforts on porting Java3D to each platform.

At this moment, 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 is implemented purely by Java which 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 walk-through application, we only took 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.

ACKNOWLEDGMENTS

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

REFERENCES

[1] "Java Programming Language," Sun Microsystems, Inc., 1998. Http://www.javasoft.com.

[2] "OpenGL WWW Center," Silicon Graphics, Inc., 1998. 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] Bing-Yu Chen, "The JavaGL 3D Graphics Library & JavaNL Network Library," Master Thesis, Dept. of Computer Science and Information Engineering, National Taiwan University, 1997.

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

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

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

[16] David Flanagan, "Java in a Nutshell," O'Reilly & Associates, Inc., 1996.

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

[18] "Café for Windows 95/NT," Symantec, Co., 1996. Http://cafe.symantec.com/cafe.

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

[20] "Java3D API," Sun Microsystems, Inc., 1997. Http://www.javasoft.com/products/java-media/3D/.

[21] Bing-Yu Chen, Chee-Wen Shiah, Tzong-Jer Yang, and Ming Ouhyoung, "JavaGL - a 3D Graphics Library in Java for Internet Browser," in Proc. of IEEE International Conference on Consumer Electronics, Chicago, Illinois, U.S.A., 1997.

[22] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, "Computer Graphics Principles and Practice," Addison-Wesley, 1990.

AUTHORS

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 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 90 technical papers on consumer electronics , computer graphics, virtual reality and multimedia system. He is a member of ACM and IEEE.