JavaGL and Its Applications for Web3D Platform

Bing-Yu Chen Tomoyuki Nishita
University of Tokyo

ABSTRACT

This paper proposes a new platform for the 3D graphics on the Internet (or for Web3D as a new platform recently). To develop 3D graphics programs on the Internet is not very easy, because there is no good enough tool, like OpenGL, to be used. For this purpose, we have developed a 3D graphics library, called JavaGL, by using pure Java since the end of 1996. At that time, we ignored some functions, such like texture mapping, because these functions are too complex to be realized on low-cost machines, even on the fastest machine three years ago. JavaGL is a general-purpose 3D graphics library, and its application-programming interface is defined in a manner quite similar to that of OpenGL. Today, the hardware is better, but the network bottleneck is still the same as before, so we almost re-wrote all the code to enhance its capabilities and performance and minimized its code size to make it more suitable for running on the Internet.

Moreover, to display or play a 3D model or scene on the Internet, people would like to use the VRML file format to describe it, and use the VRML browser plug-in of the Internet browsers to view it, because VRML is a standard 3D graphics file format and very popular. Unfortunately, besides the Microsoft Windows and SGI workstation environments, the supports for showing VRML file format are not enough. Hence, we also develop a real platform independent VRML browser applet by using JavaGL, so it could be used on any Java enabled Internet browsers. It is also a good example for JavaGL.

Keywords

OpenGL, VRML, Web3D, Java.

1. Introduction

The Internet is getting more and more popular since the end of 20th century. More and more people use the services on the Internet everyday, like WWW and E-Mail. To fulfill the contents and enhance the abilities of the Internet, people have noticed the 3D graphics recently. Although there is many 3D graphics applications could be used on the Internet, but the main problem is that they must offer several versions 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 in different platforms. Therefore, we decide to develop a 3D graphics library that is platform independent. Java 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 or PDA (personal data assistant).

E-mail: {robin, nis}@is.s.u-tokyo.ac.jp
Web: http://nis-lab.is.s.u-tokyo.ac.jp/~{robin, nis}/

To develop 3D graphics applications on a stand-alone computer, people always hope to use a powerful 3D graphics library, like OpenGL, for the detail rendering works. But, to develop such applications on the Internet by using Java, there is only Java3D, which provided by Sun Microsystems Inc., only supports few systems, and has its own API (application-programming interface). People who want to write some 3D graphics programs by using Java3D may pay much time to learn how to use it. So that, we also desire that this 3D graphics library is easy to be learn and used. Therefore, we define the API of JavaGL in a manner quite similar to that of OpenGL, since OpenGL is an industry standard, and many programmers are familiar with OpenGL's API.

Unless other commercial products; JavaGL does not need any native codes or pre-installed libraries, it is developed with only pure Java. Users run any programs developed with it do not need to install any packaged before using them, all the necessary codes will be down-loaded at the run time.

We released the first version of JavaGL in the end of 1997. In that version, we could only do the basic rendering routines, but lack of some complex functions, such like texture mapping. That is because those functions are time wasting and could not be realized on a low-end machine, even the machine is the best one at that time. But today, the hardware is much better than before, and more and more fancy applications have been needed on the Internet. Hence, we decided to enhance the capabilities of JavaGL. Unfortunately, when the hardware is getting faster and faster day-by-day, the network bandwidth is still the same as, or even worse then before. To increase the capabilities of JavaGL may make the code to be too large to be not suitable for the Internet transmission. To enhance the capabilities and minimize the code size at the same time, we decided to re-write the kernel to fulfill the requests.

Besides the 3D graphics library, to be a platform for Web3D used, we also need a 3D object and scene browsers. Hence, we also develop a VRML browser applet by using JavaGL, since people would like to use the VRML file format to display or play a 3D model or scene on the Internet. More than that, since we use JavaGL for the 3D graphics rendering, the VRML browser applet is also a good example for proving the capability of JavaGL.

2. JavaGL - A 3D Graphics Library in Java

JavaGL is a 3D graphics library for Java virtual machine. Unlike Java3D or other libraries, JavaGL does not need to be pre-installed; all the necessary codes will be downloaded at run time. JavaGL is written by pure Java, so it is really platform independent, and could be executed on any Java enabled machine. Since OpenGL is so famous and known by almost all 3D programmers, we follow the specifications of it to develop JavaGL. Therefore programmers who want to use JavaGL to develop their own 3D programs on the Internet do not need to learn how to use the new library, they can find a one-to-one mapping function in JavaGL as they call it in OpenGL.

We began to develop JavaGL since the end of 1996, and released several versions in 1997. At that time, the performance is just 4 times slower than the Mesa-3D, which is developed by using the C language. Although the performance is not too bad, we still skipped some issues, which need heavy calculations, such as texture mapping. We began to re-develop the JavaGL since the summer of 1999, almost all the codes have been re-written. The class inheritance tree structures of JavaGL have also been re-designed again.

As the development experience of JavaGL, we find that the performance is not the most important problem, but the code size is. Within the four years, the performance of the computer hardware is improved several times, but the network bandwidth is still the same. For example, four years ago, we use the PC with Intel Pentium 200MHz as our best testing platform and use 100BaseT as our Ethernet line, and 57600bps modem for dialing-up accesses. Now, we are using the PC with Intel PentiumIII 1.13GHz as our best testing platform, but the Ethernet line and modem are the same. Hence, we tried to develop JavaGL with the minimum code size.

2.1 OpenGL vs. JavaGL

Functions of OpenGL can be divided into two main categories: OpenGL Utility Library (GLU) and OpenGL (GL) as shown in Figure 1(a). JavaGL follows the same function hierarchy as shown in Figure 1(b).

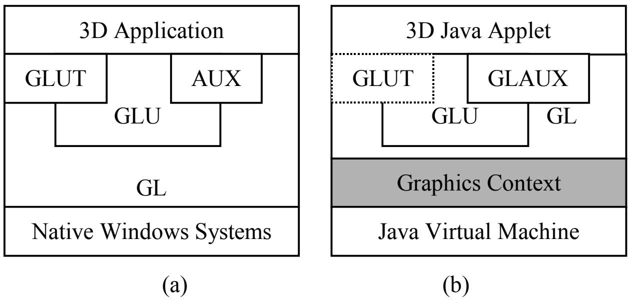


Figure 1 The hierarchy of (a) OpenGL and (b) JavaGL modules.

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. Besides these two main 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. Recently, GLUT has been widely used by programmers also, but we have not implemented this part yet.

The implementation of JavaGL is mainly based on the specifications of OpenGL, while the GLAUX library is implemented according to the *OpenGL Programming Guide*. Besides GL, GLU, and GLAUX, which are just interfaces for the programmers, there is an underlying graphics context, which is transparent to programmers, and people cannot use this part directly. Hence, to enhance the performance and minimize the code size, we rewrite all the codes in the graphics context several times.

2.2 Implementation of Graphics Context

To reduce the code size and keep or enhance the performance of JavaGL, we utilize the class inheritance characteristic to re-design the system hierarchy of graphics context as Figure 2.

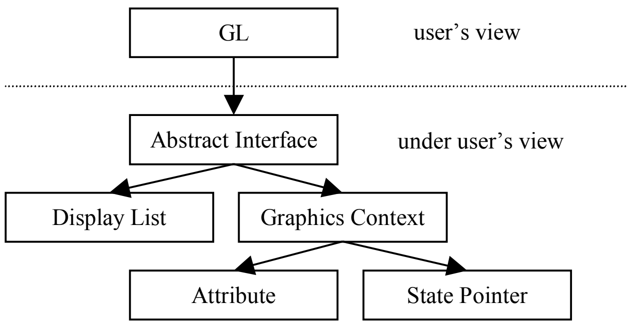


Figure 2 The graphics context of JavaGL.

The graphics context of JavaGL can be divided into two parts. One part is for display list, all the commands from the GL will not be executed and just stored as a sequence of rendering commands. The other part is for real graphics context, all the commands from the GL will be executed immediately.

The commands that will be executed in the real graphics context also can be categorized into two types. One type is just for changing some information stored in the graphics context. For example, when the users call the glClearColor command, the color value for clear the display will just be stored, no really clear actions will be occurred. Once the users call the glClear command (with GL_CLEAR_COLOR_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 occur some actions and make something changed. Since JavaGL is defined as a state machine as OpenGL, we utilize the class inheritance to avoid the frequently checks here. The states of JavaGL have been classified into several states; include 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.

2.3 Performance Enhancement Issues

Performance is a great challenge for both 3D graphics and Java, 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. Since we want to upgrade the capabilities and minimize the code size without making the run-time performance worse, these considerations make the implementation of JavaGL complex.

According to our experiences, we develop the following policies to speed up the performance and minimize the code size of JavaGL.

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 rendering commands followed will be routed to proper functions automatically without any further checks.

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 some states, 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 filled using smooth shading. However if the polygon only requires constant shading, color interpolation would be unnecessary. Therefore, we divide the shading routines into two smaller ones.

Use function overriding to minimize code size

Once we divided some routines into several smaller ones, the total code size will be larger than before, because there are too many duplicated codes. Although we have utilized class inheritance to avoid "if-than-else" statements, here we also use this method to structure all the small routines to be some hierarchy relationships, then use function-overriding feature to reduce the duplicated codes.

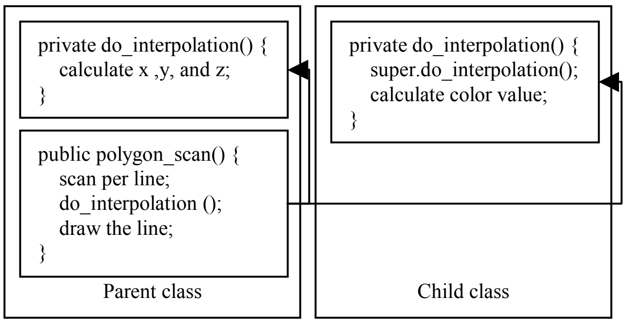


Figure 3 The example of using function overriding.

The same example as the above, since we have divided the shading routines into two smaller ones, one for flat shading and the other for smooth shading. Because all of them need polygon scan procedure to fill the polygon, we can make the two routines as 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 point 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 3.

3. VRML Browser Applet by Using JavaGL

To test the capabilities of JavaGL, we also develop a VRML browser applet by using JavaGL, since VRML is also a standard for modeling 3D models and scenes on the Internet, and used by most people. Moreover, to provide the solutions for Web3D, we will not only need to deliver the JavaGL, but also a testing platform, such as this VRML browser applet.

To provide such a browser applet by using pure Java is not very easy, but fortunately we have JavaGL to be our 3D graphics engine, and programming with JavaGL is almost the same as with OpenGL. Since VRML itself is object oriented, following the development policies of JavaGL to design the VRML browser applet by using Java is not too difficult, but the performance and code size problems are still the huge problems for it.

3.1 System Hierarchy

We follow the specification of VRML to develop the VRML browser applet. 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 could use class inheritance to construct the fields to be a class tree. For the nodes, since there are several differences between the nodes, we could only classify all the nodes into 6 main categories, and 4 sub-categories, and also make the nodes to be a class tree.

The system hierarchy of the VRML browser applet is as Figure 4, and the Node and Field are the two main parts as the above. We isolate the rendering routines (the Render Interface in the Figure) as a stand-alone class located between the VRML browser applet and the JavaGL, hence we can enhance the rendering performance with only this one class.

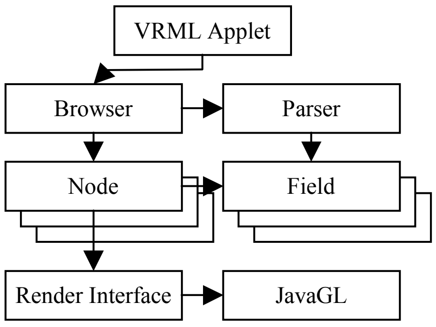


Figure 4 The system hierarchy of VRML browser applet.

The Browser and Parser as its class name are for all 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 the information into Fields of Nodes.

3.2 Performance Enhancement Issues

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

Use display list mechanism of JavaGL

Programming with OpenGL, to use a display list to store the rendering commands is reasonable. Because JavaGL has the same mechanism as OpenGL, we could utilize display list of JavaGL as programming with OpenGL.

Pre-process the constant parameters

Since almost all the parameters of nodes will not be changed while re-drawing, we could pre-process or pre-calculate all the non-changed information and store as parameters of the nodes to enhance the performance.

Combine arbitrary geometric mesh data

The arbitrary geometric mesh data is more important than other nodes, since there are only few primitive geometric nodes supported by VRML. Most people do not like to use the well-defined primitive geometric nodes, instead of using an arbitrary geometric mesh node to satisfy their desire. Therefore, to display an arbitrary geometric mesh data efficiently is more important.

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

For example, if there is an arbitrary geometric mesh as Figure 5, we could use only one node instead of using such a huge tree.

```
Group {
  children [
    Shape { # shape with first material
    }
    Shape { # shape with second material
    }
    Transform {
      children [
        Shape { # shape in other group
        }
        Shape { # shape in other group with other material
        }
      ]
    }
    Transform {
      children [
        Shape { # shape in other group
        }
        Shape { # shape in other group with other material
        }
      ]
    }

  ]
}
```

Figure 5 An arbitrary geometric mesh with different materials in VRML format. Since there are several materials, there will be several Shape nodes to make the tree huge.

4. Results

4.1 JavaGL - A 3D Graphics Library in Java

Currently, we have implemented over than **220** 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, texture-mapping, mip-mapping, evaluators, NURBS, and stippled geometry, etc. Functions not supported so far are mainly for anti-aliasing.

To test the capabilities of JavaGL, we have provided **26** examples on our JavaGL web page[1]. These examples are selected from the *OpenGL Programming Guide* and can be executed directly on Java-enabled Internet browsers. To evaluate the performance of JavaGL, we used a test program that renders **12** spheres with different materials, where each sphere contains **256** polygons, as shown in Figure 6. The performance of the test program was measured on both a SUN Ultra-10 workstation and an Intel PentiumIII-1G PC. The results are listed in Table 1. This test program is also an example in the *OpenGL Programming Guide*.

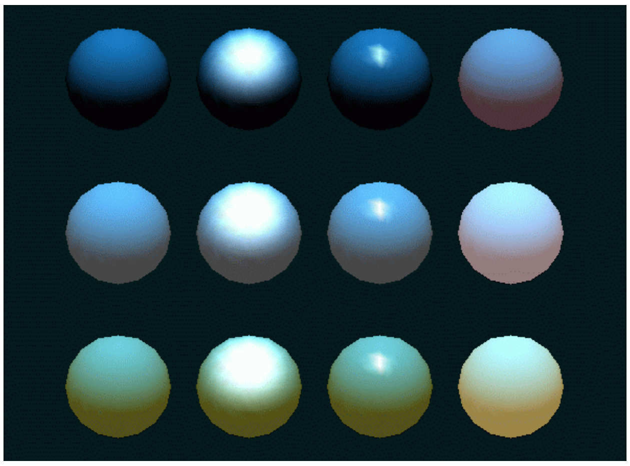


Figure 6 Twelve spheres are rendered to measure performance. Each sphere contains 256 polygons and has different material. This program is an example in *OpenGL Programming Guide* (code from Listing 6-3, pp. 183-184, Plate 16). This figure is rendered with JavaGL.

Environment	Rendering Time (ms)	Platform
Sun JDK 1.3	**219**	Intel PentiumIII-1GHz, 512MB memory, Microsoft Windows NT 4.0
	1061	Sun Ultra-10 360MHz, 256MB memory, Sun Solaris 7

Table 1 The performance comparison of JavaGL on a PC and a workstation. The test result is shown in Figure 6.

Environment	Rendering Time (ms)	Platform
Sun JDK 1.3	**219**	Intel PentiumIII-1GHz, 512MB memory, Microsoft Windows NT 4.0
Sun JDK 1.2.2 Sun HotSpot 1.0.1	**16,700**	Intel Pentium-200Hz, 64MB memory, Microsoft Windows 95
Symantec Café 1.5.1 Symantec JIT 2.0b3	**4,070**	

Table 2 A performance comparison on two different PCs. The two PCs are the most hi-end ones today and three years ago. The test program is using the same program as shown in Figure 6.

To do the migration testing, we also use the same testing program as more than two years ago. Table 2 lists the results measured on two different PCs. One is Intel PentiumIII-1G PC, and the other is Intel Pentium-200 PC, both of them are the most hi-end PCs today and three years ago. Besides the migrations of the hardware and the Java compiler (also including the just-in-

[1] Http://nis-lab.is.s.u-tokyo.ac.jp/~robin/JavaGL

time compiler), JavaGL has also been improved several times. As the result, the newest version of JavaGL using SUN JDK 1.3 is much faster than the old version of JavaGL on an old machine even using JIT (just-in-time) compiler.

Figure 7 shows a simple Java applet that draws a rectangle using JavaGL, which is similar to the simple example in the *OpenGL Programming Guide* (Listing 1-2, pp. 13, Figure 1-1), which is an official programming guide of OpenGL.

```
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) {
    myGL.glXSwapBuffers(g, this);
  }

  public void start() {
    myGL.glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    myGL.glClear(GL.GL_COLOR_BUFFER_BIT);
    myGL.glColor3f(1.0f, 1.0f, 1.0f);
    myGL.glMatrixMOde(GL.GL_PROJECTION);
    myGL.glLoadIdentity();
    myGL.glOrtho(-1.0f, 1.0f, -1.0f, 1.0f, -1.0f, 1.0f);
    myGL.glBegin(GL.GL_POLYGON);
      myGL.glVertex2f(-0.5f, -0.5f);
      myGL.glVertex2f(-0.5f, 0.5f);
      myGL.glVertex2f(0.5f, 0.5f);
      myGL.glVertex2f(0.5f, -0.5f);
    myGL.glEnd();
    myGL.glFlush();
  }
}
```

Figure 7 A simple example of JavaGL to show a white rectangle.

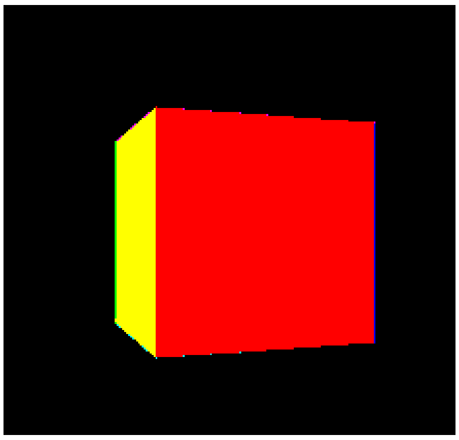


Figure 8 A simple example to render a simple cube with different colors for comparing the performance of JavaGL and Java3D. This model is an example (HelloUnverse) in the package of Java3D. This figure is rendered with JavaGL.

For comparing with Java3D, we write a program to draw a cube with different colors as the HelloUniverse example in the package of Java3D, as shown in Figure 8, and test on the same machine with Intel PentiumIII-1G CPU and a high-end display card. The result shows both of them are real-time.

Compare with the previous version of JavaGL, the byte-code size has been decreased 26.49%, and the performance has been only decreased 8.86%.

4.2 VRML Browser Applet by Using JavaGL

Now, we have implemented over than **70%** VRML nodes in our VRML browser applet. Besides the nodes, route and event transmission mechanisms have also been implemented.

To evaluate the performance of this VRML browser applet, we use a test model that renders a table with different colors for the legs and top, which is selected from the *Virtual Reality Modeling Language (VRML 97)*, the specification of VRML. This model contains **204** polygons, as shown in Figure 9. The performance of the test file was also measured on both a SUN Ultra-10 workstation and an Intel PentiumIII-1G PC. The results are listed in Table 3.

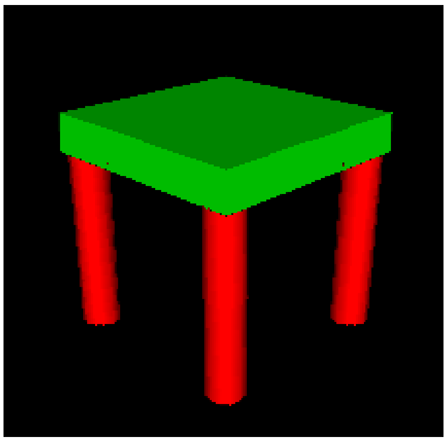


Figure 9 A simple table is rendered to measure performance. It contains 204 polygons and has two different colors for the legs and top. This model is an example in *Virtual Reality Modeling Language (VRML97)* (code pp. 211-213, Figure D.3). This figure is rendered with JavaGL.

Environment	Rendering Time (ms)	Platform
Sun JDK 1.3	**15**	Intel PentiumIII-1GHz, 512MB memory, Microsoft Windows NT 4.0
	23	Sun Ultra-10 360MHz, 256MB memory, Sun Solaris 7

Table 3 The performance comparison of VRML browser applet on a PC and workstation.

To test the performance enhancement of arbitrary geometric mesh data, we use a large arbitrary geometric mesh data with

5,273 polygons and **12** kinds of materials as shown in Figure 10, and the results are shown in Table 4. The testing platform is also Intel PentiumIII-1GHz, 512MB memory, Microsoft Windows NT 4.0, Sun JDK 1.3.

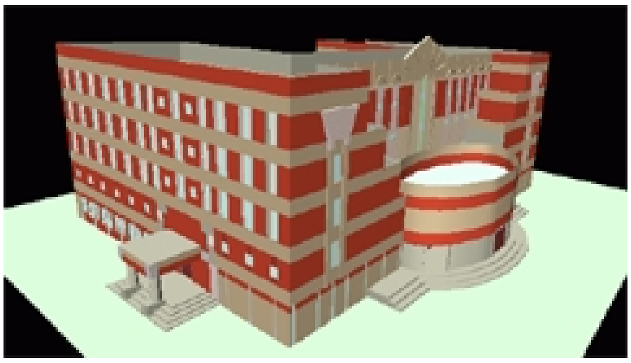


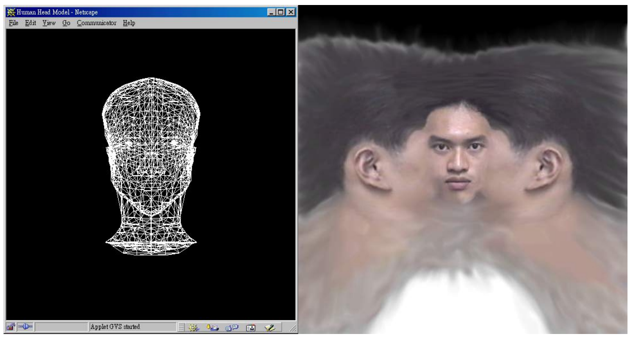
Figure 10 This arbitrary geometric mesh model contains 5,273 polygons with 12 kinds of materials in VRML format.

Rendering Methods	Rendering Time (ms)
Show as original VRML format	**447**
After performance enhancement	**375**

Table 4 The performance enhancement comparison.

4.3 3D Human Head Texture Mapping

To test the performance of texture mapping, we also develop a 3D human head texture mapping and build it with JavaGL. To do this, we use a 3D human head model with **2,185** polygons, and a **512×512** human face image as shown in Figure 11.


Human Head Model - Netscape
File Edit View Go Communicator Help
Applet GVS started

(a) (b)

Figure 11 The data for the 3D human head texture-mapping (a) the 3D human head model, and (b) the human face image.

Environment	Rendering Time (ms)	Platform
Sun JDK 1.3	**187**	Intel PentiumIII-1GHz, 512MB memory, Microsoft Windows NT 4.0
	434	Sun Ultra-10 360MHz, 256MB memory, Sun Solaris 7

Table 5 The performance testing of 3D human head texture mapping on a PC and a workstation.

Since the entire source codes are written with pure Java, and so is JavaGL, the program could be run on all kinds of Java-enabled platform directly from our web page. The performance testing is as Table 5. The result is shown in Figure 12.

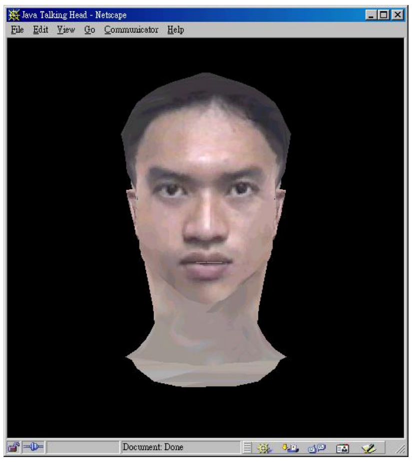

Java Talking Head - Netscape
File Edit View Go Communicator Help
Document: Done

Figure 12 The result of 3D human head texture mapping.

5. Conclusions and Future Work

Since we upload JavaGL to our web server, there are many people around the world have 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.

Sun Microsystems, Inc. has combined its Java2D into Java2, and released the newest version of Java3D in the summer of 1999. Because Java2D is part of Java core packages, it can benefit from hardware acceleration, though this will need many efforts on porting it to each platform. By using Java2D or Java3D to be the base of JavaGL may be a solution of the performance problem.

From our comparison, although Java3D uses OpenGL or DirectX as its graphics engine, the performance of JavaGL is not very worse for the simple model. Hence, JavaGL seems more suitable for small program or model on Web3D platform, since the user side does not need to install the run-time library before using the program.

As our experiences of developing JavaGL and its other applications on the Internet, the run time performance is so far not the main problem for the Java applications, since the machine's performance is getting better and better day by day. But the code size problem is still the large problem, no mater for byte code size or the model data size, since the network bandwidth is still very narrow.

At this moment, JavaGL is being applied to develop a VRML browser applet and other Java-based project in our laboratory. The goal of the Java-based projects 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 the Internet.

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

Since the network bandwidth is the most important problem for the 3D graphics on the Internet, after minimizing the code size of JavaGL and our VRML browser applet, we will try to minimize the size of the model or use the progressive refinement technologies for such issue.

We have developed JavaGL for almost four years, to make it to be an open-source project is also our future work. Java and all Java based mark is registered trademarks of Sun Microsystems, Inc., so we will change the name of JavaGL in the near future.

6. Acknowledgements

We would like to appreciate to Prof. Ming Ouhyoung, who supported the development of JavaGL since 1996, and always gives us many suggestions for developing it. We would also need to appreciate Mr. Hideki Mori, who made our testing models. The Graphics Group of Communication and Multimedia Lab., National Taiwan University creates the model of 3D human head texture mapping, thanks for all the members of that group.

7. References

[1] "The Source for Java™ Technology," Sun Microsystems, Inc., 2000. http://java.sun.com.

[2] "OpenGL – High Performance 2D/3D Graphics," OpenGL, Org., 2000. http://www.opengl.org.

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

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

[5] Bing-Yu Chen, Tzong-Jer Yang, and Ming Ouhyoung, "JavaGL - a 3D Graphics Library in Java for Internet Browsers," in IEEE Trans. on Consumer Electronics, p.271 – p.278, Vol. 43, No. 3, 1997.

[6] Brian Guenter, Cindy Grimm, Daniel Wood, Henrique Malvar, and Fredrick Pighin, "Making Faces," in Computer Graphics (SIGGRAPH 98 Proceedings), pp. 55-66, 1998.

[7] Hugues Hoppe, "Efficient Implementation of Progressive Meshes," in Computer & Graphics, Vol. 22, No. 1, pp. 27-36, 1998.

[8] Hugues Hoppe, "Progressive Meshes," in Computer Graphics (SIGGRAPH 96 Proceedings), pp. 99-108, 1996.

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

[10] JavaSoft, "The Java 3D API Specification," Sun Microsystems, Inc., 2000.

[11] Mark Segal, and Kurt Akeley, "The OpenGL Graphics Systems: A Specification (Version 1.1)," Silicon Graphics, Inc., 1996.