中華民國八十六年全國計算機會議

以 Java 爲組織三維圖學及網路程式庫
JavaGL and JavaNL:
A Multi-participant 3D Graphics Application Interface in Java

陳炳宇, 楊宗哲, 歐陽明
Bing-Yu Chen, Tzong-Jer Yang, and Ming Ouhyoung

台灣大學資訊工程系 通訊與多媒體實驗室
Communications and Multimedia Laboratory,
Department of Computer Science and Information Engineering,
National Taiwan University, Taipei, Taiwan, R.O.C.

摘要

本論文提出兩個 Java 函式庫, 分別是圖形函式庫 -- JavaGL, 及網路函式庫 – JavaNL. 由於這兩個函式庫全以 Java 寫成, 因此很適合於網際網路、全球資訊網等應用. JavaGL 提供了適合在網路上執行的 3D 圖形函式庫, 而 JavaNL 則可協助程式發展者更容易開發多人共同使用的應用程式. 本論文除了提出發展過程中所獲得的經驗外, 亦提出JavaGL 及 JavaNL 的效能評量.

Abstract

This paper presents implementation issues on both a 3D graphics library, JavaGL[1], and a network library, JavaNL[2]. These two libraries are both written in pure Java, so they are suitable for use in the Internet or WWW, as well as embedded systems. JavaGL provides programmers an application interface to develop 3D graphics applications over network; JavaNL helps programmers to develop multi-participant applications easier. Performance evaluations are also addressed.

1. Introduction

As the Internet and World Wide Web (WWW) become so popular, many Internet-based products, including Network Computers [1] and WebTVs, have been developed. Furthermore, in the visible future, 3D web navigation (e.g., VRML) will be a reasonable demand for web browsers. However the Internet itself is a heterogeneous network environment, if we want to deliver WWW contents with 3D information , we need a 3D graphics capability in each different platform. Therefore, we decide to develop a 3D graphics library that is platform independent. Java is chosen as our programming language for its hardware-neutral feature, and its wide availability on many hardware platforms, including embedded systems.

We also notice that a multi-participant interactive environment would be a potential demand for Internet applications, hence we also developed a Java network library, called JavaNL, to help programmers to develop 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, including performance evaluations, in section 4. Discussions and conclusions are addressed in section 5.

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.

Functions in OpenGL can be divided into 3 categories: OpenGL Utility Library (GLU), OpenGL (GL), and OpenGL Extensions 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).

GL implements 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.

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. For that reason, we also include GLAUX in our JavaGL package.

---

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

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

This work is 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.

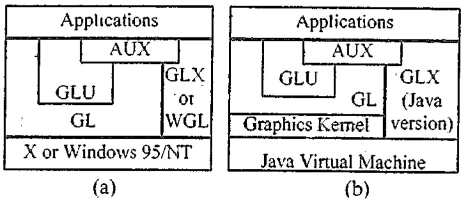


Figure 1. (a)The hierarchy of OpenGL modules. (b)The hierarchy of JavaGL modules.

The implementation of JavaGL follows 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.

The graphics kernel shown in Figure 1(b) is the most important part in JavaGL, and is illustrated in the following section.

2.1 Implementation of JavaGL graphics kernel

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

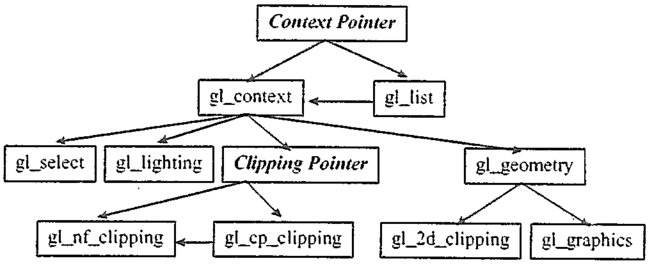


Figure 2. The hierarchy of JavaGL's graphics kernel.

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.

2.2 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 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 are 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's 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's 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 can be divided into several smaller ones, where each routine differs from one another very much, it's worth to divide this routine into several smaller pieces. The purpose is to reduce unnecessary network transmissions for unused code segments.

   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, and the corresponding code segment transmitted would be redundant. 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 reason is again 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.

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 one another simultaneously. JavaNL, a multi-participant network library, is developed to fit the purpose.

JavaNL adopts some 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

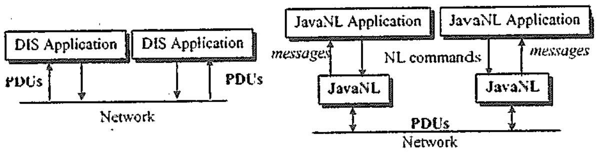
DIS is a set of IEEE standards including IEEE Std 1278.1-1995 [9][10] for application protocols and IEEE Std 1278.2-1995 [11] for communication services and profiles. IEEE P1278.3 is for exercise management and feedback, and has not yet been standardized.

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 PDUs (Protocol Data Units) 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 [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 3(b).



(a) (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.

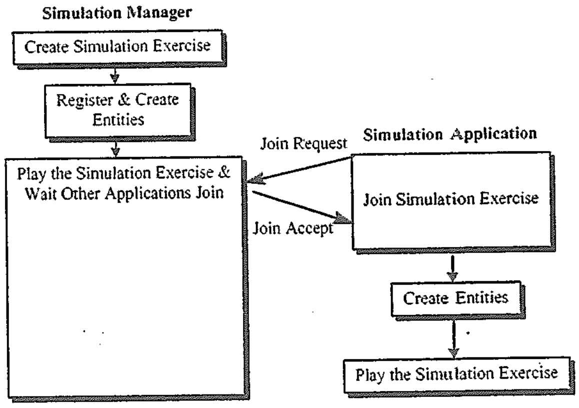


Figure 4. The control flow of PDUs in JavaNL.

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.

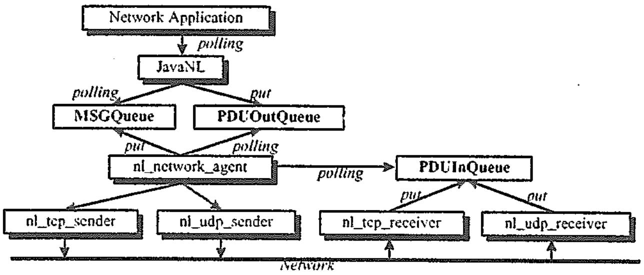


Figure 5. PDU sending and receiving in JavaNL.

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.

4. Results

Currently, we have implemented over 160 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 and selection. Functions not supported so far are mainly for anti-aliasing and texture mapping. Figure 6 shows a simple Java applet that draws a rectangle using JavaGL.

We also provide 16 examples on our JavaGL web page. These examples are selected from the OpenGL Programming Guide [5], and can be executed directly in Internet browsers supporting Java.

```
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

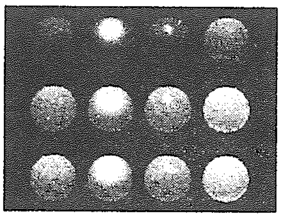


Figure 7. Twelve spheres rendered with JavaGL. Each sphere contains 256 polygons. This program is an example in OpenGL Programming Guide [5] (code from Listing 6-3, pp. 183-184, Plate 16).

To evaluate JavaGL's performance, we used a test program that renders 12 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 test program using C program language with standard OpenGL APIs. The C version was then compiled with Mesa 3-D graphics library [13], a software-based OpenGL like graphics library, and the rendering time was measured. We also compiled the C version with hardware accelerated OpenGL on both platforms. The results are listed in Table 1 and Table 2.

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 [14].

Table 2 lists the result measured on a Pentium-200 PC. The test program using SUN JDK 1.0.2 [15] is 4 times slower then the one using Symantec Café 1.51 [16] with the support of Just-In-Time (JIT) compiler [17].

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 (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.

We also applied JavaGL to render complex models. Figure 8 shows our department building containing **5273** triangles, and the rendering time is **6150 ms** 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.

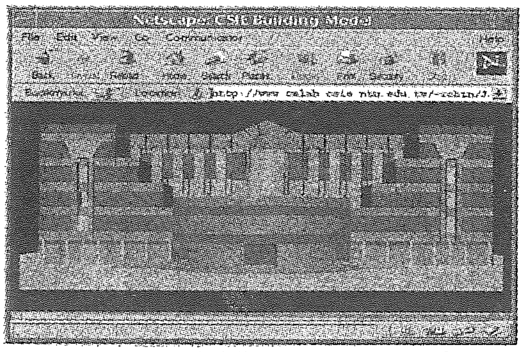
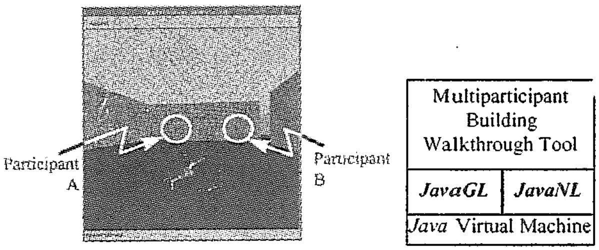


Figure 8. Our department building rendered with JavaGL on Netscape Navigator 4.0pr2. This model contains 5273 triangles and takes **6150 ms** on a PC with Intel Pentium-200 CPU and 64 MB memory.

To demonstrate the usage of 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 9(a). The system hierarchy is shown in Figure 9(b).



(a) (b)

Figure 9. (a) 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. (b) The system hierarchy of a multi-participant building walkthrough application using JavaGL and JavaNL.

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 3. We also measured the round-trip time of JavaGL PDU, Java UDP, and C UDP, and the results are 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.

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. Discussions and Conclusions

Since we upload JavaGL to our web server, there have been over **2000** 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.

SUN also revealed its Java3D [19] this year. 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.

There is another Java 3D graphics library J3DI (Java 3D Interactive) that is developed by Maciej Gorny. J3DI provides similar functionality supported in JavaGL, and is available on the Internet [20].

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'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 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.

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

7. 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.

[19] "*Java3D API*," Sun Microsystems, Inc., Http://java.sun.com/products/java-media/3D/.

[20] Maciej Gorny, "*Java 3D Interactive – 3D Graphics Class Library for Java Home Site*," Http://www.rowan.edu/~gorny/.