Evaluating Java AWT for Cross-Platform Java Game Development

Yi-Hsien Wang and I-Chen Wu
Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan
{wangys,icwu}@csie.nctu.edu.tw

Abstract*-This paper extends the research of CYC Window Toolkit (CWT) [11] to other environments in two aspects. First, we evaluate the rendering performance of Java AWT in most commonly used JREs on four operating systems (OSs), including Windows XP, Windows Vista, Fedora Core and Mac OS. The evaluation results indicate that the performance inconsistency of Java AWT also exists among the four OSs, even if the same hardware configuration is used. Second, we design an OpenGL version of CWT, named CWT-GL, to take advantage of modern 3D graphics cards. The results show that CWT-GL achieves more consistent and higher rendering performance in JRE 1.4 to 1.6 on the four OSs.*

Keywords: CYC Window Toolkit, OpenGL, Windows, Linux, Mac OS

1. Introduction

Since released in 1995, Java has attracted much attention in game industry. Along with the growth of *World Wide Web* (WWW) in the late 1990s, many Java casual applet games were deployed over the Internet, including *Yahoo! Games* and *CYC Games*. It is because these games can be easily distributed over the Internet and played on multiple operating systems (OSs). Other than applet games, several commercial stand-alone Java games were also developed, such as *You Don't Know Jack* and *Tribal Trouble*. Examples of commercial *massively multiplayer online* (MMO) Java games include *RuneScape* and *Wurm Online*.

Although PCs have great support of Java, many Java games on PCs are still *low-profile games*[1]. The most discussed reasons include the runtime speed and the rendering performance of Java, because early implementations of the JVM and *graphic user interface* (GUI) components, also called *widgets*, normally delivered poorer performance, which made game developers hesitant to use it for *high-profile games*[1].

[1] Low-profile and high-profile games are defined in [4].

In view of these problems, research for Java graphics that is reviewed in Subsection 1.1 has been done to make Java more suitable for game development. However, some problems which are identified in Subsection 1.2 still remain in the GUI part of Java, especially when programmers try to deploy cross-platform Java games with consistent rendering performance. By consistent rendering performance, we mean to deliver similar rendering performance on different OSs or different rendering environments when using the same or equivalent-power hardware. Subsection 1.3 briefly describes our goals for solving these problems, and also summarizes the organization of the rest of this paper.

1.1. Evolution of Java Graphics

This subsection reviews the graphics part of Java, which is of great concern to game developers today and is the focus of this paper.

The standard way to perform 2D graphics in Java is the use of Java AWT/Swing. However, Java AWT/Swing did not take full advantage of graphics cards, before J2SE 1.4. J2SE 1.4 introduces the *DirectX-based Java2D pipeline* (abbreviated as DirectX pipeline) on Microsoft Windows platforms, while J2SE 5.0 introduces the *OpenGL-based Java 2D pipeline* (abbreviated as OpenGL pipeline) on Windows platforms and Linux. Since then, the rendering performance of Java AWT/Swing has had a great boost. In particular, the use of OpenGL which is supported by multiple platforms is quite important to Java in which the cross-platform feature is critical.

1.2. Problems of Java Graphics

Although the rendering performance of Java AWT/Swing has been improved since J2SE 1.4, some problems are still identified in cross-platform game development. This subsection lists three problems as follows.

(1) Inconsistent rendering performance among different JREs.

This problem occurs since significant changes are made in the graphics part of newer JREs, as described in Subsection 1.1. These changes are

Table 1. Current state of JREs.

Java Version	Percentage of Web Browser Users	Available CWT Implementation
MSVM	10.75%	DirectX (in [11])
J2SE 1.4	10.27%	OpenGL (in this paper)
J2SE 5.0	23.77%	
Java SE 6	54.73%	

tightly bound to the versions of the JREs and are rarely ported back to old JREs. However, according to the statistical data in [2] during April and May in 2008, the percentages of Web browser users of popular JREs, as shown in Table 1, indicate that about 10.75% of Web browser users still used MSVM, and the percentage for J2SE 1.4 Web browser users was 10.27%. Using MSVM on Windows platforms or J2SE 1.4 on non-Windows platforms, such as Linux, game applications cannot obtain the benefit of hardware acceleration.

(2) Inconsistent rendering performance among different operating systems.

The problem is caused by different implementations of graphics systems as follows. Java 2D rendering pipelines are built on different graphics systems on different OSs, such as Window *graphics device interface* (GDI) and DirectX on Microsoft Windows platforms, *X Window System* (X) on Linux, and *Quartz graphics layer* (Quartz) on Mac OS X. In addition, Windows Vista has a new graphics system called *Desktop Window Manager* (DWM), which runs on top of Direct3D and through which GDI rendering is redirected. Other than the above graphics systems, OpenGL is supported on all of the four OSs. Since the JREs involve these different graphics systems on different OSs, the optimization of Java games for one OS may not be applicable to other OSs. Therefore, more efforts are required for testing and optimizing the games on all targeted OSs.

(3) OpenGL pipeline is not stable enough.

Although the OpenGL pipeline is supported on multiple OSs, it is disabled by default owing to the stability issues [8]. During our test, the OpenGL pipeline sometimes delivered incorrect screen, such as blank or incomplete screen. Besides, Java game programmers cannot fully depends on the OpenGL pipeline, since it can only be enabled in JREs 1.5 and beyond on Windows and Linux, and JRE 1.6 on Mac OS 10.5.2.

1.3. Goals of This Paper

In view of the first problems listed in Subsection 1.2, Wang, Wu and Jiang [11] developed a toolkit, named *CYC Window Toolkit* (CWT), and implemented a DirectX version on Microsoft Windows XP to improve the rendering performance of MSVM. In this paper, we extend their work to various JREs on more OSs in the following two aspects.

First, evaluate the rendering performance of Java AWT in different JREs on four OSs, including Windows XP, Windows Vista, Fedora Core and Mac OS. The evaluation results indicate that the performance inconsistency of Java AWT/Swing also exists among the four OSs, even if the same hardware configuration is used. This problem weakens the merits of the *Write-Once-Run-Anywhere* (WORA) feature of Java.

Second, propose solutions to solve the above problems of Java AWT and compare the results with those of Java AWT. That is, we design an OpenGL version of CWT [11] via OpenGL, named CWT-GL. The results show that CWT-GL achieves more consistent and higher rendering performance in JRE 1.4 to 1.6 on the four OSs.

The rest of this paper is organized as follows. Section 2 presents the design of CWT-GL. Section 3 describes the configurations of JREs and benchmark programs used in this paper. Section 4 analyzes the experimental results. Finally, Section 5 presents conclusions and future work.

2. Design of CWT-GL

This Section describes the design of CWT-GL. Subsection 2.1 introduces JOGL and Subsection 2.2 reviews the architecture of CWT. Subsection 2.3 presents the design of the CWT-GL based on JOGL and Subsection 2.4 discusses the issues related to performance optimization. Subsection 2.5 summarizes the work related to CWT.

2.1. Introduction to JOGL

JOGL (Java binding for the OpenGL API) [9] is an open-sourced project initiated by Sun Microsystems. JOGL is a Java binding for OpenGL and provides access to the latest OpenGL API. JOGL abstracts the OpenGL functionality from platform-specific libraries, such as `wgl`, `glx` and `agl`, to create a platform-independent OpenGL API. The abstraction greatly improves the portability of JOGL on different OSs. Since JOGL is a development version of the JSR-231 and will possibly be included in the Java SE core library in the future, we implemented CWT-GL using JOGL.

2.2. Architecture of CWT

CWT [11] was proposed to provide high and similar rendering performance for Java game development on different platforms. It aimed to enhance the graphics performance by directly using DirectX and OpenGL to render all widgets, figures, images and texts. For ease of use and

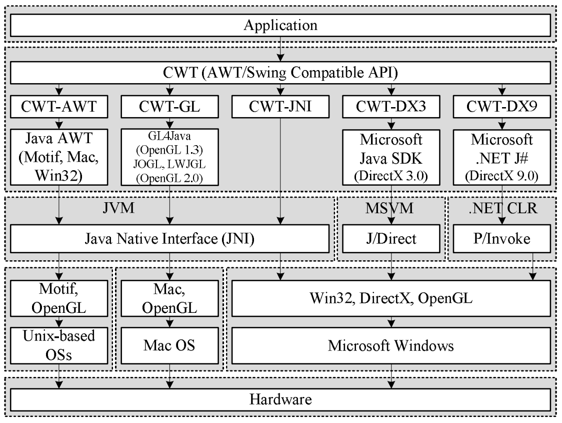
Application
CWT (AWT/Swing Compatible API)
CWT-AWT
CWT-GL
CWT-JNI
CWT-DX3
CWT-DX9
Java AWT (Motif, Mac, Win32)
GL4Java (OpenGL 1.3) JOGL, LWJGL (OpenGL 2.0)
Microsoft Java SDK (DirectX 3.0)
Microsoft .NET J# (DirectX 9.0)
JVM
MSVM
.NET CLR
Java Native Interface (JNI)
J/Direct
P/Invoke
Motif, OpenGL
Mac, OpenGL
Win32, DirectX, OpenGL
Unix-based OSs
Mac OS
Microsoft Windows
Hardware

Figure 1. Architecture of CWT.

backward compatibility, CWT provides AWT/Swing compatible APIs for Java 1.1 and beyond.

The architecture of CWT is shown in Figure 1. Supporting Java AWT/Swing compatible APIs, CWT defines component hierarchy, event model, and painting model. The component hierarchy models a component hierarchy similar to Java AWT/Swing 1.1. The event model specifies the event-handling process. The painting model defines an abstract class called `Graphics` which is required to be implemented in the wrapper implementations of CWT: CWT-DX, CWT-GL, and CWT-AWT. All of the wrapper implementations share the component model and event model, but realize the `Graphics` class using different graphics libraries, such as DirectX, OpenGL and Java AWT. The greater details of these models are described in [11].

2.3. Design of CWT-GL

In this subsection, we briefly introduce how CWT-GL implements the Graphics class using JOGL. We divide the functionalities of the Graphics class into three parts: figures, images, and texts, whose design issues and strategies are described in Subsections 2.3.1 to 2.3.3, respectively.

2.3.1. Figures

In Java 1.1, the Graphics class allows programs to draw several kinds of figures, including lines, rectangles, ovals, round rectangles, polylines and polygons. These figures are mainly of two types – outline and solid figures. In OpenGL, outline figures can be assembled by lines, while solid figures can be filled by triangles. Therefore, we use lines and solid triangles for these figure-drawing and figure-filling methods, respectively.

2.3.2. Images

In CWT-GL, images are loaded onto so-called texture maps to fill rectangles. In practice, there are several limitations when we use texture mapping for the simulation of drawing AWT images. These limitations and the corresponding solutions are described as follows.

First, the size of each texture has a maximum bound. For example, the limitation on texture size of ATI X1600 series, which are used as our test beds, is up to (4096×4096)-pixel^2. Currently, CWT-GL does not support images larger than the bounds of the underlying system.

Second, most graphics cards only support power-of-two-sized texture. When dealing with non-power-of-two images in the graphics cards, JOGL pads the image by creating a power-of-two texture image and then draws the original image onto the new one. However, the price to pay is more memory consumed. This problem can be solved by introducing texture packing [1], which groups small images into a single power-of-two texture to utilize memory.

2.3.3. Texts

Since text drawing is not directly supported in OpenGL, two alternatives are used in OpenGL applications: image-based and geometry-based approaches [10]. The image-based approach draws texts by rendering images on which texts are pre-rendered or dynamically rendered during runtime. On the other hand, the geometry-based approach represents texts in a series of lines, curves and polygons. Since the texts are presented in 3D models, scaling the texts will not cause the effect of artifact. However, the more complex shape the texts are of, the more polygons and processing power are needed. For example, Asian languages, such as Chinese, typically require more polygons to emulate.

CWT-GL implements both approaches. Since both approaches have cons and pros, CWT-GL lets programmers configure the behaviors of the text engine during runtime, such as the threshold of font size for enabling geometry-based rendering.

2.4. Optimization of CWT-GL

In order to achieve fast rendering for game development, CWT-GL introduces two optimization methods, (1) disabling unnecessary checking and testing, and (2) minimizing the number of state changes in OpenGL.

In optimization method (1), we disable some unnecessary checking and testing of OpenGL before performing certain rendering operations. For example, alpha testing and blending mode are unnecessary when the programs draw opaque images and figures.

In optimization method (2), we try to minimize the number of state changes in OpenGL, which take extra overhead in time [10]. For example, binding textures and invoking glBegin()/glEnd(). In order to achieve this, CWT-GL uses variables to indicate current states of bound textures and type of glBegin(). Before issuing the rendering operations to OpenGL, CWT-GL changes OpenGL states only when the states are different from required ones. Therefore, unnecessary state changes can be avoided.

2.5. Related Work

Agile2D [6] implements an almost complete set of Java 2D functionalities based on GL4Java to replace the repaint manager of Swing. The authors of Agile2D also showed the improvement in rendering performance to Sun's Java 2D implementation. However, there exist some problems in Agile2D. First, Agile2D supports only J2SE 1.4 and beyond, and it does not support the acceleration of Java AWT 1.1, which is still used by many applet games, such as *Yahoo! Games* and *CYC Games*. Second, Agile2D is based on GL4Java which only supports OpenGL version 1.4 and has no plan for evolution. Finally, Agile2D only supports the first 256 characters in Unicode, e.g. ISO 8859-1. These issues can limit the applications of Agile2D.

FengGUI [5] is a Java graphics toolkit based on JOGL and LWJGL. This toolkit specially focuses on the rendering performance for multimedia and game applications, and has been used in several commercial projects. FengGUI provides a new set of commonly used widgets and graphics API with easy-to-use design. Programmers can also directly access JOGL or LWJGL, since FengGUI does not encapsulate these two APIs. However, using FengGUI, programmers need to learn not only the new API, but JOGL or LWJGL, which may reduce the programmers' productivity. In addition, FengGUI supports only JRE 1.5 and beyond, which also limits possible Web browser users.

3. Experiments

We implemented a benchmark program to simulate a Bomberman game, as shown in Figure 2. We measured the average frame rate of the Bomberman game in rendering 20000 frames. For backward compatible to Java 1.1, we only used Java 1.1 API to implement the program.

We performed our benchmark on four OSs. In order to make fair comparison, Windows XP Professional SP2, Windows Vista Business and Fedora Core 6 were installed in a PC specified in Table 2, while Mac OS 10.4.11 was installed in an iMac with computing power roughly equivalent to

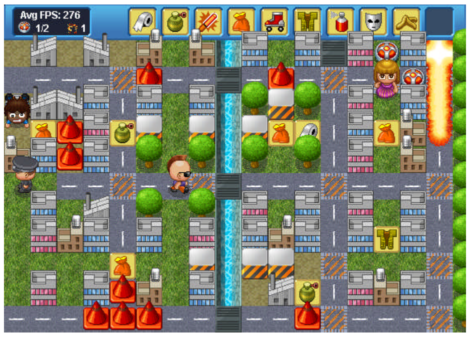

Figure 2. A screenshot of the benchmark.

Table 2. System hardware.

Computer	Hardware
1	● AMD X2 3800+ 2.0GHz ● ATI Radeon X1650 with 256 MB GDDR2 AGP (1280×1024@60Hz). Driver version: Catalyst 8.4
2	● Intel Core 2 Due 2.0GHz ● ATI Mobility Radeon X1600 with 128MB GDDR3 PCIe (1440×900@60Hz)

the PC. Both computers worked in true color mode and disabled font anti-aliasing. We installed most of the popular JREs on these OSs, including MSVM (version 5.0.0.3810), Sun JRE versions 1.3.1_20, 1.4.2_16, 1.5.0_13, and 1.6.0_05. Note that Mac OS 10.4 does not support Java SE 6.

We ran test programs in all the *rendering environments* (RE) with the combination of using different JREs, system properties, and OSs. We identify each RE by an identifier, a tuple of four attributes (Toolkit, JRE, SystemProperty, OS).

- *Toolkit*∈{AWT, CWT-DX, CWT-GL}. AWT represents the Java AWT graphics library, CWT-DX represents the DirectX implementation of CWT in [11], and CWT-GL represents the OpenGL implementation of CWT in this paper.
- *JRE*∈{M, 4, 5, 6}. "M" represents MSVM, and "4" to "6" denote JRE version 1.4 to 6, respectively.
- *SystemProperty*∈{N, O}. "N" represents the case that no system properties are specified for JREs, and "O" denotes the case that OpenGL pipeline is enabled (by setting the system property `sun.java2d.opengl` to `true`).
- *OS*∈{XP, VS, FC, MC}. "XP," "VS," "FC" and "MC" respectively represent the following operating systems, Windows XP, Windows Vista, Fedora Core and Mac OS.

For example, (AWT, 6, O, XP) refers to the RE that uses AWT, runs in JRE version 6, enables OpenGL pipeline, and runs on Windows XP. For simplicity, the wildcard character "*" indicates a group of REs for all cases in the attribute. For example, (AWT, *, O, *) means all the REs with AWT and with OpenGL pipeline enabled on the four OSs.

To measure the rendering performance, we use two metrics: *Frame Rate* and *Anomaly*. Frame rate is commonly employed to measure the rendering speed expressed by *frames per second* (FPS) for the macro-benchmark. For a RE *r*, *FrameRate*(*r*) denotes the frame rate in *r*. *Anomaly* for a set of REs, say *R*, is defined as follows.

$$Anomaly(R) = \frac{\max_{r \in R}(FrameRate(r))}{\min_{r \in R}(FrameRate(r))}$$

Anomaly is employed to measure the inconsistency of the rendering speed in a given set of REs.

4. Analysis

In this section, we first summarize the results of Java AWT and CWT-GL in Subsection 4.1. Next, we discuss the results and our suggestions for cross-platform Java game development in Subsection 4.2.

4.1. Summary of Results

First, we summarize the results of Java AWT as follows. The rendering performance of Java AWT is inconsistent among the four OSs. Figure 3 shows the results of comparing the rendering performance among the four OSs, given RE (AWT, {M, 4, 5, 6}, N, *). As shown in the figure, Fedora Core 6 often delivers much slower frame rates than those on other OSs, which is the main source of performance inconsistency among the four OSs. For all j∈{4, 5, 6}, and p∈{N}, *Anomaly*(AWT, j, p, *) ranges from 3.06 to 3.64. This means that the rendering performance of the same Java program would be quite different on the four OSs, especially on Fedora Core.

This phenomenon also exists when we enable the OpenGL pipeline in RE (AWT, 6, O, *), as shown in Figure 3. The OpenGL pipeline delivered much worse frame rates. Furthermore, the OpenGL pipeline was not stable enough since it sometimes rendered incomplete screens in these REs during our tests.

To sum up the results of Java AWT, programmers may find it hard to optimize the rendering performance on the four OSs for cross-platform Java games. From the aspect of OS, Java games running on Fedora Core would be two to three times slower than those running on other OSs. Thus, the efforts for performance testing are required for programmers to develop cross-platform Java games requiring consistently high rendering performance.

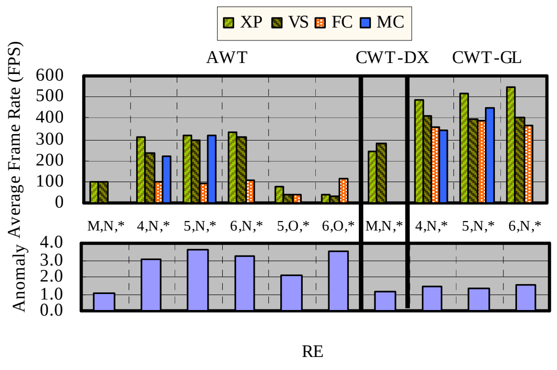

Figure 3. Frame rates and Anomaly among different OSs in the REs, (AWT, {M,4,5,6}, *), (CWT-DX, M, *) and (CWT-GL, {4,5,6}, *).

It is even worse that some of the parameters, such as JRE versions and system properties, are controlled by users, not by the programmers, especially for Java applet games, where the programmers have fewer choices. Therefore, Java AWT/Swing programmers need to pay more attention to the issue when consistently high rendering performance is required for cross-platform Java games.

Next, we summarize the results of the CWT as follows. CWT-GL achieves higher and more consistent rendering performance among the four OSs than Java AWT does. In Figure 3, RE (CWT-GL, {4, 5, 6}, N, *) often delivers the highest frame rates, and also more consistent rendering performance than Java AWT. For example, for all j∈{4, 5, 6}, *Anomaly*(CWT-GL, j, N, *) ranges from 1.34 to 1.49, while *Anomaly*(AWT, j, N, *) ranges from 3.06 to 3.64. Therefore, CWT-GL performs more consistently than AWT does among the four OSs.

Generally speaking, the rendering performance of CWT-GL is higher and more consistent on supported REs than those in Java AWT. This is quite important especially when games run in users' computers with various REs. Furthermore, the system properties in CWT-GL are simpler than Java AWT, which also helps reduce the testing efforts. Therefore, our experimental results suggest that CWT-GL is more suitable for cross-platform Java game development than Java AWT.

4.2. Discussion

Although the WORA (Write-Once-Run-Anywhere) feature is very attractive to Java game developers and Java has been greatly improved on performance in terms of JVM and graphics, the inconsistency of rendering performance weakens the merit of WORA for game development, especially for cross-platform games running in various REs. In this subsection, we further discuss the problems of current Java 2D rendering pipelines.

As described in Subsection 1.2, since the JREs involve different graphics systems on different OSs, it is not surprised that Java AWT delivers inconsistent rendering performance among the OSs. For example, the rendering performance on Fedora Core is relatively low when compared with other OSs. Fortunately, the OpenGL pipeline shows its potential on cross-platform Java game development. However, the OpenGL pipeline is still not good and stable enough. In contrast, CWT-GL achieves high and consistent rendering performance on all of the tested OSs by direct access to OpenGL via JOGL.

The current approach by Sun to solving the problem is to bundle the OpenGL pipeline in JRE, since J2SE 5.0 [7]. However, this approach also incurs the following three problems.

(1) Since the new OpenGL pipeline is bundled with new JREs and is not ported back to old JREs, users need to upgrade to one of the new JREs.

(2) In order to obtain new features or fix bugs in the OpenGL pipeline, programmers and users have to upgrade their entire JREs, not only the part of the OpenGL pipeline.

(3) Programmers must wait for newer JREs to improve the reliability, performance or more support of the OpenGL pipeline. For example, future JREs are required, since the OpenGL pipeline is currently not reliable enough, and that Mac OS 10.4 and below do not support the OpenGL pipeline.

These problems weaken the motivation of using Java AWT/Swing to develop cross-platform Java games with high and consistent rendering performance.

5. Conclusions

This paper identifies the inconsistent rendering performance problem on four popular OSs, including Microsoft Windows XP and Vista, Fedora Core, and Mac OS. The results indicate that it is hard to predict the rendering performance of Java AWT on the four OSs.

To solve this problem, we have designed an OpenGL version (CWT-GL) of the CWT architecture defined in [11] using JOGL. The results indicate that CWT-GL generally reaches higher and more consistent rendering performance in J2SE 1.4 to 6 on the four OSs.

The contributions of this paper are listed as follows. (1) Identify the problem of inconsistent rendering performance of Java AWT/Swing on four OSs. (2) Implement CWT-GL so that CWT can be applied to more OSs other than Microsoft Windows. (3) Demonstrate the high and consistent rendering performance of CWT-GL experimentally on four popular OSs.

Future extension includes more optimizations on CWT-GL and applying our work to 3D game development in Java. We will also implement new Java AWT/Swing APIs introduced after version 1.1 to improve the usability of CWT.

Acknowledgements

The authors would like to thank the National Science Council of the Republic of China for joint financial support of this research under contract number NSC 95-2622-E-009-008-CC3. Next, the authors would like to thank ThinkNewIdea, Inc. for providing the data required for this research. Finally, the authors would also like to thank Chin-Yi Cheng who contributed to the OpenGL implementation of CWT.

References

[1] Alexander Wong and Andrew Kennings, “Adaptive multiple texture approach to texture packing for 3D video games,” *Proceedings of the 2007 conference on Future Play*, Toronto, Canada, pp.189-196.

[2] Andrew Gray. GC Usage Statistics. http://www.andrew-gray.com/dist/stats.shtml

[3] Internet Application Technology Lab. CYC Window Toolkit. National Chiao-Tung University, Taiwan. http://java.csie.nctu.edu.tw/cwt

[4] Jacob Marner, *Evaluating Java for Game Development*, Dept. of Computer Science, Univ. of Copenhagen, Denmark, 2002.

[5] Johannes Schaback, FengGUI, Java GUIs with OpenGL. http://www.fenggui.org/

[6] Jon Meyer, Ben Bederson, and Jean-Daniel Fekete, Agile2D. Human-Computer Interaction Lab, University of Maryland, USA. http://www.cs.umd.edu/hcil/agile2d/

[7] Sun Microsystems Inc., *New Java 2D Features in J2SE 5.0*. Sun Microsystems Inc., 2004.

[8] Sun Microsystems Inc., *Update: Desktop Java Features in Java SE 6*. Sun Microsystems Inc., 2005.

[9] Sun Microsystems Inc., *JSR 231: Java Binding for the OpenGL API*. Sun Microsystems Inc. http://jcp.org/en/jsr/detail?id=231

[10] Tom McReynolds and David Blythe, *Advanced Graphics Programming Using OpenGL*. Morgan Kaufmann, February 2005.

[11] Yi-Hsien Wang, I-Chen Wu and Jyh-Yaw Jiang. “A portable AWT/Swing architecture for Java game development,” *Software Practice and Experience*, vol. 37, issue 7, pp.727-745, June 2007.