SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2009; **39**:701–736
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.920

Achieving high and consistent rendering performance of Java AWT/Swing on multiple platforms

Yi-Hsien Wang and I-Chen Wu*,†

Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan

SUMMARY

Wang *et al.* (*Softw. Pract. Exper.* 2007; 37(7):727–745) observed a phenomenon of performance inconsistency in the graphics of Java Abstract Window Toolkit (AWT)/Swing among different Java runtime environments (JREs) on Windows XP. This phenomenon makes it difficult to predict the performance of Java game applications. Therefore, they proposed a portable AWT/Swing architecture, called CYC Window Toolkit (CWT), to provide programmers with high and consistent rendering performance for Java game development among different JREs. They implemented a DirectX version to demonstrate the feasibility of the architecture. This paper extends the above research to other environments in two aspects. First, we evaluate the rendering performance of the original Java AWT with different combinations of JREs, image application programming interfaces, system properties and operating systems (OSs), including Windows XP, Windows Vista, Fedora and Mac OS X. 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, and compare the rendering performance of CWT with Java AWT/Swing. The results show that CWT-GL achieves more consistent and higher rendering performance in JREs 1.4 to 1.6 on the four OSs. The results also hint at two approaches: (a) decouple the rendering pipelines of Java AWT/Swing from the JREs for faster upgrading and supporting old JREs and (b) use other graphics libraries, such as CWT, instead of Java AWT/Swing to develop cross-platform Java games with higher and more consistent rendering performance. Copyright © 2009 John Wiley & Sons, Ltd.

Received 15 September 2007; Revised 9 February 2009; Accepted 10 February 2009

KEY WORDS: CYC Window Toolkit; DirectX; OpenGL; Windows; Linux; Mac OS X

*Correspondence to: I-Chen Wu, Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan.
†E-mail: icwu@csie.nctu.edu.tw

Contract/grant sponsor: National Science Council; contract/grant number: NSC 95-2622-E-009-008-CC3
Contract/grant sponsor: ThinkNewIdea, Inc.

1. INTRODUCTION

Since it was released in 1995, Java [1] has attracted much attention in the game industry. Along with the growth of *World Wide Web* in the late 1990s, many Java casual applet games were deployed over the Internet, including *Yahoo! Games* [2], *ArcadePod.com* [3] and *CYC games* [4,5]. It is because these games can be easily distributed over the Internet and played on multiple operating systems (OSs). Other than the applet games, several commercial stand-alone Java games were also developed, such as *You Don't Know Jack* [6], *Law & Order*: *Dead on the Money* [7] and *Tribal Trouble* [8]. Examples of commercial *massively multiplayer online* Java games include *RuneScape* [9], *Puzzle Pirates* [10] and *Wurm Online* [11]. Java games also appeared on mobile devices and soon became the mainstream language for game development on these devices. For example, *Age of Empires II* [12] was ported to mobile devices [13].

Although PCs have great support of Java, many of the current Java games on PCs are still *low-profile games*‡. 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 hesitate to use it for *high-profile games*‡.

In view of these problems, research for Java Graphics that is reviewed in Section 1.1 has been done to make Java more suitable for game development. However, some problems that are identified in Section 1.2 still remain in the GUI part of Java, in terms of Abstract Window Toolkit (AWT) and Swing, 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 (REs) when using the same hardware or equivalent-power hardware. The consistency of rendering performance is quite important, since programmers would expect Java programs to run with similar performance on multiple OSs. Section 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 section reviews the graphics part of Java, which is of great concern to game developers today and is the focus of this paper.

For high rendering performance, game developers commonly employ DirectDraw and Direct3D of *Microsoft DirectX* (or DirectGraphics in DirectX 7.0 and beyond) [15] or *Open Graphics Library* (OpenGL) [16] to access specialized hardware features, such as direct access to the video memory in graphics cards and constructing 3D scenes.

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. Therefore, the rendering performance of AWT/Swing programs (before J2SE 1.4) is not as good as programs accelerated by the graphics hardware.

‡According to [14], high-profile games usually attempt to attract the highest attention from retailers and media. Such games normally require several millions of U.S. dollars to advance in technologies, such as graphics. On the other hand, low-profile games target at niche groups of players and try to lower down developing costs.

Realizing this fact, Sun has started to access graphics hardware features via DirectX since J2SE 1.4 [17] and OpenGL since J2SE 5.0 (or 1.5) [18]. Full-screen mode has been supported and new types of images, such as *volatile images* and *managed* (or *compatible*) *images*, have been designed to take advantage of graphics hardware [17]. Since then, the rendering performance of Java AWT/Swing has had a great boost. In particular, the use of OpenGL that is supported by multiple platforms is quite important to Java in which the cross-platform feature is critical.

However, currently the *OpenGL-based Java 2D pipeline* (abbreviated as OpenGL pipeline) can only be enabled in Java runtime environments (JREs) 1.5 and beyond on Windows and Linux, and JRE 1.6 on Mac OS X 10.5.2. It is disabled by default, because it does not work well in some cases [19]. Although Java SE 6 (or 1.6) introduces a newly designed OpenGL pipeline that gives much better stability and performance than that in J2SE 5.0, the pipeline is again disabled by default for robustness issues [19].

In addition to the 2D graphical libraries, several 3D graphical libraries were also developed to build cross-platform Java games with high rendering performance, including *the OpenGL for Java* (GL4Java) [20], *Java binding for OpenGL* (JOGL) [21], *Lightweight Java Game Library* (LWJGL) [22] and *Java 3D* [23]. The first three libraries are OpenGL bindings, which provide low-level one-to-one mapped application programming interfaces (APIs) to OpenGL. Using GL4Java, JOGL or LWJGL, Java programmers can access hardware features supported in OpenGL without writing Java Native Interface (JNI) wrappers. On the other hand, Java 3D provides high-level APIs, which use OpenGL and DirectX internally, for creating, rendering and manipulating 3D scene graphs.

Using these libraries not only improves greatly the rendering performance on supported platforms, but also helps to build modern 3D games with realistic scenes. As a result, several cross-platform 3D Java games, including *Law & Order*: *Dead on the Money* [7], *Jake2* [24] and *Wurm Online* [11], were created using these 3D graphical libraries, instead of using AWT/Swing, to achieve high rendering performance.

1.2. Problems of Java Graphics

Although the graphics part of Java evolves as described in the previous section, some problems are still identified when Java AWT/Swing is employed to develop cross-platform games. This section lists six problems: The first three were identified by Wang *et al.* [25], while the last three are identified in this paper.

(1) *Backward compatibility with the old JREs without graphics acceleration.*
As described in Section 1.1, Java AWT/Swing and most 3D libraries require at least J2SE 1.4 to achieve high rendering performance. However, according to the statistical data in [26] during April and May in 2008, the percentages of Web browser users of popular JREs, as shown in Table I, indicate that about 11% of Web browser users still used JREs below 1.4, where game applications cannot obtain the benefit of hardware acceleration mentioned above. Thus, this problem is significant when game programmers particularly for applets need to take the legacy Java users into consideration.

(2) *Unexpected rendering performance and visual effects when mixing Java AWT/Swing components with these 3D libraries, supported in DirectX and OpenGL.*
Directly accessing the 3D graphics libraries instead of Java AWT/Swing usually achieves good rendering performance. However, these libraries do not provide widget systems.

Table I. Current state of JREs.

Java version	Supported OSs	Released time	JRE download size on Windows (MB)	Percentage of Web browser users (April to May 2008)	Available CWT implementation
MSVM (Java 1.1.4)	Windows	02/1997	5	10.75	DirectX (in [25])
J2SE 1.3	Windows,	05/2000	7.9	0.48	AWT (in [25])
J2SE 1.4	Mac OS,	02/2002	15.2	10.27	
J2SE 5.0	Linux,	09/2004	15.8	23.77	OpenGL (in this paper)
Java SE 6	Solaris	11/2006	13.2	54.73	

Consequently, when mixing Java AWT/Swing components with these 3D libraries, the performance may still be limited to that of the widget systems, or even worse [27]. In addition, the AWT/Swing components typically control their repainting timing and process, which may cause some unexpected visual effects, such as flickering and tearing. Thus, game programmers typically build their own widget systems for their games. However, this reduces the productivity of programming.

(3) *Inconsistent rendering performance among different JREs*
The problem of inconsistent rendering performance among different JREs occurs since significant changes are made in the graphics part of newer JREs. For example, J2SE 1.2 introduced software rendering for outputting equal rendering quality on different platforms [28], J2SE 1.3 introduced *AWT Native Interface* that enables native code to draw directly on Java drawing surface [29], J2SE 1.4 introduced DirectX pipeline, while Java SE 5.0 introduced OpenGL pipeline. These significant changes result in two phenomena. First, these changes are tightly bound to the versions of the JREs and are rarely ported back to old JREs. Second, not all of the changes improve rendering performance. As observed in [25], the rendering performance of texts and figures drops seriously from Java 1.1 to J2SE 1.2 and from J2SE 1.3 to J2SE 1.4. Therefore, such a phenomenon may make it difficult for programmers to tune up the performance for all of the JRE versions.

(4) *Inconsistent rendering performance among different OSs.*
The rendering performance of Java AWT/Swing is inconsistent among different OSs, even when the same hardware configuration and JRE are used. 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) [30] on Linux and *Quartz graphics layer* (Quartz) [31] 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 [32]. 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.

(5) *Inconsistent rendering performance on choosing different image APIs.*
In order to let programmers access hardware features, J2SE 1.4 introduced volatile images and managed images (or so-called compatible images). Later, J2SE 5.0 introduced translucent-supported volatile images. Using these new APIs properly may improve the overall rendering performance but lose the backward compatibility with the old JREs. When programmers want to support legacy Java users, they may either only use old image API or write several versions of programs that access different image APIs in different JREs. However, the problem of inconsistent rendering performance still occurs in either way.

(6) *Inconsistent rendering performance on setting different system properties.*
Besides the choices of image APIs described in the fifth problem, system properties also need to be set carefully for better rendering performance. For example, the system property '`sun.java2d.opengl`' needs to be specified to enable the OpenGL pipeline [33,34]. However, these system properties need to be set before the startup of Java AWT/Swing, which means that programmers cannot dynamically change the settings during runtime. It is even worse that some of these need to be specified by users, not just by programmers. Consequently, it is difficult for users to use the proper settings that programmers want, or to set these system properties without the administrator's rights or help, e.g., the system properties in the Java applets of the Web browsers [35]. Thus, this problem makes it difficult for programmers to predict rendering performance on end users' systems.

1.3. Goals of this paper

In view of the first three problems listed in Section 1.2, Wang *et al.* [25] developed a toolkit, named *CYC Window Toolkit* (CWT), and implemented a DirectX version on Microsoft Windows XP to deal with the problems. In this paper, in order to solve all the six problems, we extend their work to various JREs on more OSs in the following two aspects:

(1) Evaluate the rendering performance of the original Java AWT with different combinations of JREs, image APIs, system properties and OSs, including Windows XP, Windows Vista, Fedora and Mac OS X. These OSs are selected according to population percentages shown in Table II. 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. In addition, the results also show that no specific image APIs and system properties are guaranteed to obtain high and consistent rendering performance in different JREs. These problems weaken the merits of the *Write-Once-Run-Anywhere* (WORA) feature of Java.
(2) Propose solutions to solve the above problems of Java AWT and compare the results with those of Java AWT/Swing. We implement an OpenGL version of CWT [25] via JOGL, named CWT-GL. The results show that CWT-GL achieves more consistent and higher rendering

Table II. Percentages of OSs in April 2008 [36].

OS		Percentage
Windows	XP	73.3
	Vista	8.8
Mac OS		4.6
Linux		3.7

performance in JRE 1.4 to 1.6 on the four OSs. The results also hint two approaches. First, decouple the rendering pipelines of Java AWT/Swing from the JREs for faster upgrading and supporting old JREs. Second, allow programmers to use other graphics libraries, such as CWT, instead of Java AWT/Swing to develop cross-platform Java games with higher and more consistent rendering performance.

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 the 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. For this implementation, CWT accesses 3D graphics hardware acceleration via OpenGL. Several OpenGL bindings are available, including GL4Java [20], JOGL [21] and LWJGL [22]. Among these bindings, JOGL is chosen due to the official support from Sun Microsystems. Based on JOGL, CWT-GL not only renders graphical primitives through 3D hardware acceleration, but also optimizes the rendering performance by several techniques. Therefore, CWT-GL achieves high and consistent rendering performance when compared with Java AWT/Swing. Finally, we summarize work related to CWT, including Agile2D, FengGUI, and Minueto.

2.1. Introduction to JOGL

JOGL is an open-sourced project initiated by the Game Technology Group at Sun Microsystems [21]. JOGL is a Java binding for OpenGL and provides access to the latest OpenGL API, including writing shader code. 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. JOGL is a development version of the JSR-231 (Java binding for the OpenGL API) [37] and will possibly be included in the Java SE core library in the future.

2.2. Introduction to CWT

CWT was proposed by Wang *et al.* [25] to provide high and similar rendering performance for Java game development on different platforms. Their approach 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 backward compatibility, CWT provides AWT/Swing compatible APIs for Java 1.1 and beyond. In other words, CWT has been designed to adapt the DirectX and OpenGL APIs into the graphics APIs of AWT/Swing of Java 1.1.

The architecture of CWT is shown in Figure 1. Supporting Java AWT/Swing compatible APIs, CWT defines component hierarchy, event model and painting model [25]. 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` that is required to be implemented in the wrapper implementations of CWT: CWT-DX, CWT-GL and

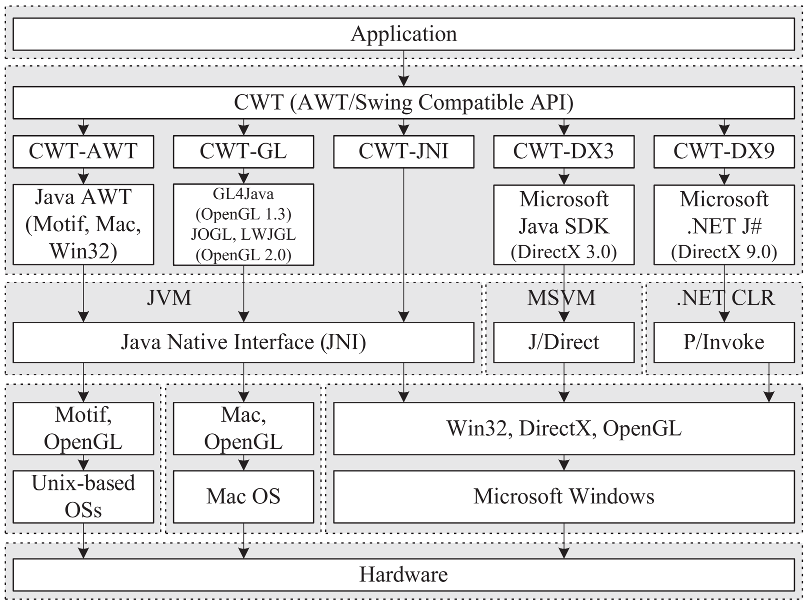

Application
CWT (AWT/Swing Compatible API)
CWT-AWT
CWT-GL
CWT-JNI
CWT-DX3
CWT-DX9
Java AWT (Motif, Mac, Win32)
GL4Java (OpenGL 1.3) JOGL, LWJGL (OpenGL 2.0)
Microsoft Java SDK (DirectX 3.0)
Microsoft .NET J# (DirectX 9.0)
JVM
MSVM
.NET CLR
Java Native Interface (JNI)
J/Direct
P/Invoke
Motif, OpenGL
Mac,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

Figure 1. Architecture of CWT.

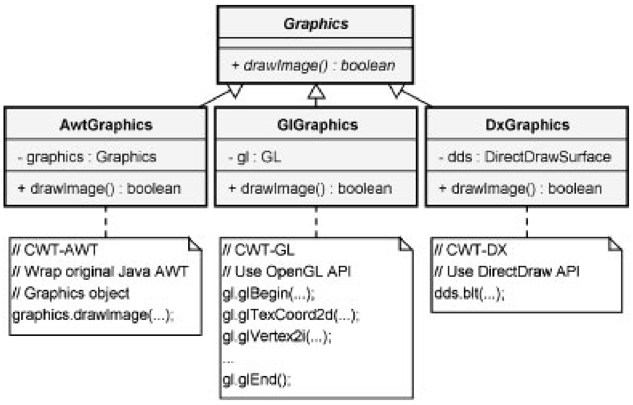
CWT-AWT, respectively, described as follows:

- For DirectX, CWT accesses DirectX 3.0, which is supported in Microsoft Java Virtual Machine (MSVM) [38] via a wrapper identified as CWT-DX3, and DirectX 9.0, which is supported in .NET J# [39] via a wrapper identified as CWT-DX9. The two together are called CWT-DX.
- For OpenGL, CWT accesses it via a wrapper, identified as CWT-GL. There are several candidate libraries: GL4Java (supporting OpenGL 1.3), JOGL and LWJGL (supporting OpenGL 2.0). All these libraries are available in various OSs, including Microsoft Windows, Mac OS X, Linux and Solaris. In this paper, we use JOGL.
- When neither DirectX nor OpenGL is supported by the underlying OSs, CWT accesses Java AWT via a simple wrapper, identified as CWT-AWT.

All of the wrapper implementations share the component model and event model, but realize the `Graphics` class using different graphics libraries. As shown in Figure 2, CWT-AWT, CWT-GL and CWT-DX, respectively, use Java AWT, OpenGL API and DirectX API to implement the operations defined in the `Graphics` class. The greater details of these models are described in [25].

With this architecture, CWT offers the following features:

(1) Take full advantage of the hardware acceleration for most commonly used JREs even including 1.1, since CWT is implemented using DirectX and OpenGL.
(2) Allow to access the DirectX and OpenGL objects directly to manipulate more hardware features and to tune performance.



```
// CWT-AWT
// Wrap original Java AWT
// Graphics object
graphics.drawImage(...);
```

```
// CWT-GL
// Use OpenGL API
gl.glBegin(...);
gl.glTexCoord2d(...);
gl.glVertex2i(...);
...
gl.glEnd();
```

```
// CWT-DX
// Use DirectDraw API
dds.blt(...);
```

Figure 2. Three implementations of the Graphics interface in CWT.

(3) Adapt DirectX and OpenGL APIs for AWT/Swing interfaces.
(4) Allow to use AWT/Swing widgets while accessing DirectX and OpenGL.

This architecture also helps to solve and improve the six problems mentioned in Section 1.2, as follows:

(1) For the problem of backward compatibility with the old JREs without graphics acceleration, Wang *et al.* [25] have improved the rendering performance by implementing CWT-DX3 for MSVM. Together with CWT-GL for J2SE 1.4 and beyond on multiple OSs designed in this paper, CWT covers in total 99.52% of Web browser users in Table I, whereas MSVM, J2SE 1.4, Java SE 5.0 and Java SE 6 are used by 10.75, 10.27, 23.77 and 54.73% Web browser users, respectively.
(2) For the problem caused by mixing Java AWT/Swing components with the 3D libraries, CWT supports AWT/Swing compatible widgets rendered by the 3D libraries, including DirectX and JOGL. Since 3D scenes and the widgets are rendered by the same libraries, this problem no longer exists.
(3) For the problem of inconsistent rendering performance among different JREs, CWT is independent of JREs so that CWT can be applied to almost all the JREs, even including JDK 1.1 (backwards) and future JREs (forwards). Therefore, rendering performance among different JREs becomes more consistent.
(4) For the problem of inconsistent rendering performance among different OSs, CWT directly uses hardware acceleration supported on the OSs to avoid the problem of inconsistent rendering performance caused by the different rendering pipelines on different OSs.
(5) For the problem of inconsistent rendering performance on choosing different image APIs, CWT provides one set of graphics API that is compatible with Java 1.1, and internally uses hardware acceleration features. This improves the compatibility issue and reduces test effort. The details are described in Section 4.1.
(6) For the problem of inconsistent rendering performance when setting different system properties, users do not need to set system properties before the startup of the CWT programs, since CWT lets users (including programmers) configure the rendering behaviors during runtime. The details are described in Section 4.1.

CWT is available on our web site [40]. Since CWT provides an AWT/Swing-like API, the way to use CWT instead of Java AWT is replacing the '`import`' statements of code from '`java.awt.*`' to '`com.cyc.lib.cwt.*`'. More details of the porting guide are also available on the web site [40].

2.3. Design of CWT-GL

In this section, we briefly introduce how CWT-GL implements the `Graphics` class using JOGL. We divide the functionalities of the `Graphics` class into five parts: figures, images, texts, off-screen buffers and graphics states, whose design issues and strategies are described in Sections 2.3.1–2.3.5, 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. Most importantly, we use the number of lines or triangles as small as possible to achieve high rendering performance for game development. For example, CWT-GL uses just enough one-pixel lines to approximate a round circle [41].

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 [42]. The values of the bounds may vary, depending on users' systems and graphics cards. Currently, CWT-GL does not support images larger than the bounds of the underlying system.

Second, some old graphics cards only support power-of-two-sized texture [41]. Therefore, if the image is not power-of-two in dimension and the graphics card does not support non-power-of-two image, 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. For example, a 65×33-pixel^2 image has to be padded to a 128×64-pixel^2 size before it can be used as a texture map. This problem can be solved by introducing *texture mosaicing* [43] (or called *texture packing* [44]), which groups small images into a single power-of-two texture to utilize memory. This technique is commonly used in the game applications [44]. Note that the problem of optimizing texture packing can be reduced to the 2D Knapsack problem, which is known to be NP-hard [45].

Finally, the size of texture memory is also limited [41]. Thus, OpenGL as well as CWT-GL needs to manage the texture memory by moving textures in and out according to the priority of the textures. The method `glPrioritizeTextures()` can set the priority to minimize texture memory thrashing. Therefore, CWT-GL adds a corresponding attribute (named `priority`) in the `Image` class for programmers to manage the texture 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 [43]. The image-based approach draws texts by rendering images on which texts are pre-rendered or dynamically rendered during runtime. This approach is further divided into two methods, bitmaps and texture maps. The former is simpler and more efficient in memory utilization. However, the latter is normally faster than the former since the latter is directly supported by hardware acceleration.

Although the image-based approach is easy to implement, it has two drawbacks. First, a pre-rendered text has fixed resolution, so the quality of scaled texts would not be as good as that of the originals. Second, when the font size is large, the images consume more memory and rendering time. For example, a 32×32-sized character costs three times more memory than a 16×16-sized character does.

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 the shape of the texts, the more the polygons and processing power needed. For example, Asian languages, such as Chinese, typically require more polygons to emulate.

According to the analysis above, CWT-GL implements both approaches described as follows:

- In the image-based approach, the text engine first renders the texts into texture maps in a character-by-character basis, and then uses the texture maps to display the texts. The texts will be cached in the texture maps for later uses. We use the *Least Frequently Used* algorithm to maintain the character cache. The number of texture bindings can be reduced by putting a number of the characters in one texture map instead of generating each individual character in its own texture map, since drawing a string typically involves drawing a series of characters.
- In the geometry-based approach, we follow the common method described in [43]. The text engine generates glyphs for each character and also caches them in display lists for later use. Since most glyphs contain curves, such as quadratic parametric curves and Bezier curves, the text engine needs to use cubic interpolation to draw the curves. Similar to the way we optimize circle drawing described in Section 2.3.1, we only interpolate each curve by a limited number of steps according to the distance between two ends of the curves.

Since both approaches have *cons and pros*, CWT-GL lets programmers configure the rendering behaviors of the text engine during runtime, such as the size of texture cache and the threshold of font size for enabling geometry-based rendering. According to our experimental results in [40], we set 1 MB as the default size of the texture cache and 32 as the default threshold of the font size to be a balanced point between the rendering speed and memory consumption. When font size is larger than the threshold, CWT-GL uses the geometry-based approach to draw texts; otherwise, CWT-GL uses the texture-based approach.

We adopt two methods to reduce the memory used by the texture map for the text cache as follows: First, in order to reduce the number of cached texts, the color information of the texts is removed, i.e. we let characters with different colors share the same cache space in the texture map.

To do this, the cached texts are drawn in white color with a black background. Then, we enable the blending function to blend the designated color[§] before drawing the texts. The blending function is also specified so that the white color of the cached texts will be drawn by the designated color and the black background will become transparent[¶]. Second, since the color information is not needed in the cached texts, we use a one-byte-per-pixel grayscale texture map, which can still be accelerated by hardware.

2.3.4. Off-screen buffers

Rendering to off-screen buffers is a common operation in Java AWT. It is useful for performing double buffering, dynamically creating images during runtime (runtime images) for special effects. Although there are several ways to do so in OpenGL, only few are hardware-accelerated and fast enough for game development. Currently, two techniques for fast off-screen rendering are *pixel buffer* (pbuffer) [46] and *Framebuffer Object* (FBO) [47]. Both have been implemented in CWT-GL, since both have advantages over each other, as described as follows:

- *Pbuffer.* The pbuffer technique [46] is an OpenGL extension. Pbuffers allow programmers to create hardware-accelerated off-screen buffers. This method is faster than the old ways when doing off-screen rendering, such as `glReadPixels()`, `glDrawPixels()` and `glCopyTexSubImage2D()`, which involves copying pixels between *video memory*[‖] and *system memory*[**] [41]. However, each pbuffer is associated with one distinct OpenGL context, which incurs overhead in both time and space as described in the following. Switching to another pbuffer causes OpenGL context switching, which takes extra time [41]. Moreover, since pbuffers cannot share space, each pbuffer must contain its own data for some extra buffers, such as depth buffer, stencil buffers, accumulation buffers [41]. Despite of these disadvantages, the pbuffer technique is supported by more graphics cards, since it was introduced earlier than the FBO.
- *FBO.* The FBO [47] extension is a good alternative to pbuffer, since it makes off-screen rendering more efficient and easier to use. Unlike pbuffer, binding a different FBO does not require context switching, because different FBOs are allowed to share one OpenGL context, such as depth buffer, stencil buffers and accumulation buffers. The design of sharing context also helps to reduce memory consumption. Moreover, FBO is easier to set up than pbuffer. As a result, FBO now becomes the better choice of off-screen rendering in OpenGL. The only problem of FBO, however, is that it is a new extension and supported by fewer graphics cards than pbuffer.

According to the analysis above, the order of techniques for off-screen rendering in CWT-GL is (1) FBO and (2) pbuffer. For backward compatible consideration to make CWT-GL work on

[§]The designated color is specified by using the `glBlendColor()` function.

[¶]The blending function is configured as `glBlendFunc(GL_CONSTANT_COLOR, GL_ONE_MINUS_SRC_COLOR)`. The color C rendered on target will be $C_{src} \times C_{blend} + C_{dst} \times (1.0 - C_{src})$, where C denotes each individual red, green and blue color from 0.0 to 1.0. Therefore, the white color ($C_{src} = 1.0$) part of the cached texts will be drawn with C_{blend}, while the black background ($C_{src} = 0.0$) will be drawn with C_{dst}.

[‖]Video memory refers to the memory on the video cards that holds data for display devices.

[**]System memory refers to the memory where a computer holds current programs and data that CPU works with.

systems with old graphics cards where neither FBO nor pbuffer is available, CWT-GL creates AWT off-screen buffers for rendering, and then transfers the buffers into textures when rendering is finished.

2.3.5. Graphics states

Graphics states control rendering behaviors, including origins, clipping areas, colors and fonts. For example, first set the foreground or background color into the graphics state for subsequent drawing of such lines or circles. Java AWT encapsulates the graphics states into the `Graphics` objects, while OpenGL stores them in the OpenGL contexts. In AWT/Swing applications, a `Window` object may contain a number of `Component` objects, and each `Component` object maintains its own graphics states in its `Graphics` object. However, in multithread environments, concurrently drawing `Component` objects in Java must carefully make graphics states of OpenGL contexts consistent. Therefore, in order to design a mechanism to let CWT run correctly in multithread environments, we need to take the following two points into consideration.

First, most CWT components are of lightweight [25]. These lightweight components are finally painted on the four CWT heavyweight components, including Window, Frame, Dialog and Applet. Heavyweight components independently keep graphics states such as painting colors, while lightweight components in a single heavyweight component share the same graphics states. Thus, a single heavyweight component must execute correctly the interleaved rendering operations from these lightweight components with different graphics states, and set the proper graphics states before drawing its lightweight components. This implies that we need to serialize the rendering operations and execute them by a single thread.

Second, OpenGL is mainly designed for single-threaded usage. As suggested by the *single-threaded rendering* (STR) [48] introduced in Java SE 6, using a single thread to issue OpenGL commands is more efficient and reliable than using the multithreaded way, which is common in AWT/Swing applications. It would be better to avoid the multithreading approach, since it introduces more unexpected rendering performance issues in Java programs as indicated in [49].

According to the two points, we adopt the design of STR into CWT. We use the Java AWT *event dispatching thread* (EDT) as the single command processing thread, since one of the responsibilities of the EDT is to repaint the components. An example of this design is depicted in Figure 3 and the steps involved are described as follows:

- *Step 1*: a rendering request `drawImage()` is issued to a `GlGraphics` object.
- *Step 2*: the request is translated into an internal command with the required state.
- *Step 3*: the command is put into a command queue.
- *Steps 4 and 5*: the command queue invokes the method `display()` of an internal `GLCanvas` object in order to activate the EDT to execute this rendering request.
- *Step 6*: the activated EDT invokes the method `display(GLAutoDrawable drawable)` implemented by the command queue.
- *Steps 7 and 8*: the command is retrieved and executed by the EDT by calling the `drawImageImp()` method in the `GlGraphics` object.
- *Step 9*: before the command is executed, the `GlGraphics` object changes the states of the OpenGL context to match the required state of the command.
- *Step 10*: the `GlGraphics` object issues the OpenGL commands.

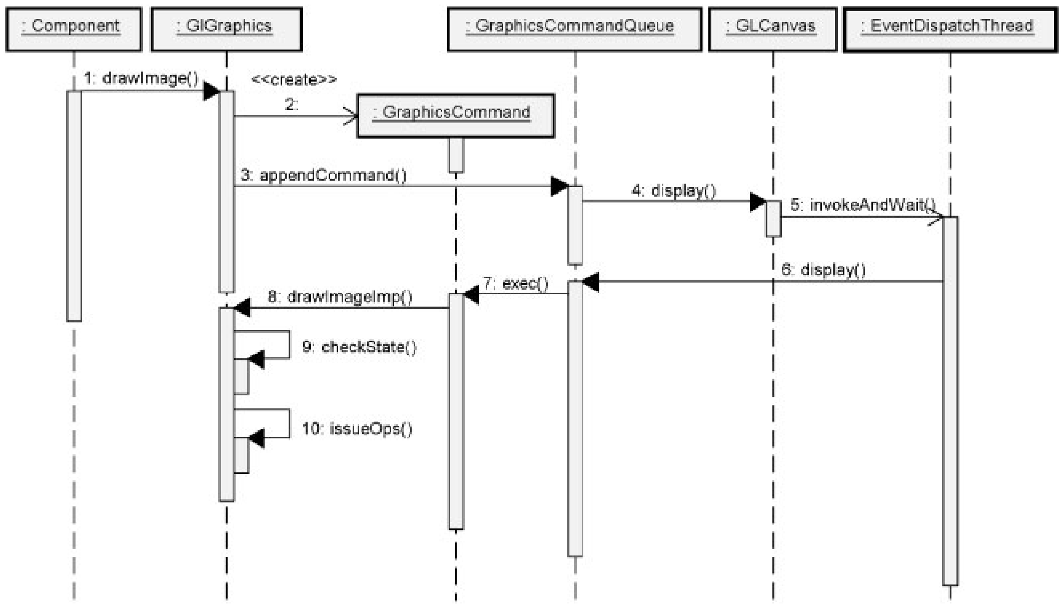


Figure 3. Sequence diagram of STR-like design in CWT.

Therefore, all the OpenGL operations are issued by the same thread, and the required states of the graphics commands are ensured.

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. These optimizations effectively improve the rendering performance of CWT-GL implementation.

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, while these tests are required for drawing transparent images, translucent images and texts. Turning off unnecessary checking can greatly improve the performance.

In optimization method (2), we try to minimize the number of state changes in OpenGL. As described in Section 2.3.4, OpenGL context switching takes extra overhead in time [41], for example, binding textures and invoking `glBegin()/glEnd()`. Others such as changing color and setting clipping area do not affect speed much. However, since JOGL invokes corresponding OpenGL API via JNI, it is still a good idea to reduce the number of JNI calls. Therefore, CWT-GL tries to minimize the number of method calls that change OpenGL states. For example, Figure 4 shows three cases of unnecessary changes of OpenGL state (pseudo-code with strikeout) when drawing an image continuously: (a) `glBegin()/glEnd()`, (b) `glBindTexture()` and

Pseudo-code of Java AWT	Pseudo-code of optimized OpenGL commands
g.drawImage(img1);	gl.glBindTexture(img1);
	gl.glEnable(GL.GL_TEXTURE_2D);
	gl.glBegin(GL.GL_QUARDS);
	gl.glTexCoord2d(...);
	gl.glVertex2i(...);
	...
	~~gl.glEnd();~~
	~~gl.glDisable(GL.GL_TEXTURE_2D);~~
g.drawImage(img1);	~~gl.glBindTexture(img1);~~
	~~gl.glEnable(GL.GL_TEXTURE_2D);~~
	~~gl.glBegin(GL.GL_QUARDS);~~
	gl.glTexCoord2d(...);
	gl.glVertex2i(...);
	...
	gl.glEnd();
	gl.glDisable(GL.GL_TEXTURE_2D);

Figure 4. Eliminating unnecessary changes of OpenGL state.

(c) `glEnable()/glDisable()`. In order to achieve this, CWT-GL uses variables to indicate the current states of bound textures, color, clipping area and type of `glBegin()`. Before issuing the rendering operations to OpenGL, CWT-GL changes OpenGL states only when the states are different from the required ones. Therefore, unnecessary state changes can be avoided.

2.5. Related work

Some research, such as Agile2D [50], focuses on building OpenGL adapters to Java AWT/Swing, while others, such as FengGUI [51] and Minueto [52], try to create toolkits with different APIs. In this section, we review the work related to CWT.

2.5.1. Agile2D

Agile2D [50] implements an almost complete set of Java 2D functionalities based on GL4Java to replace the repaint manager of Swing. Therefore, it improves the rendering part of Swing without the need of re-implementing Swing components. 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* [2], *ArcadePod.com* [3] and *CYC games* [4,5]. Second, Agile2D is based on GL4Java that 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.

2.5.2. FengGUI

FengGUI [51] 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. In addition, FengGUI can also be combined with several 3D game engines, including jMonkey Engine [53], jPCT [54] and Xith3D [55]. 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 also the JOGL or LWJGL, which may reduce the programmers' productivity. In addition, FengGUI supports only JRE 1.5 and beyond, which also limits the possible Web users.

2.5.3. Minueto

Minueto [52] is a Java 2D game framework based on Java AWT/Swing. The author designed it especially for undergraduate students in order to ease the work of Java game programming, including graphics, input and sound. Therefore, the API of Minueto is different from Java AWT/Swing, which requires extra efforts to port existing Java games to Minueto. For high rendering performance, Minueto provides an expansion module called MinuetoGL using JOGL. Unfortunately, although testing the rendering performance of their engine on Windows XP, Linux and Mac OS, the author did not address how different settings can affect the Java rendering performance, which is one of the objectives in this paper.

3. EXPERIMENTS

In order to evaluate the consistency of rendering performance of a Java program running in possible combinations of toolkit, JREs, image APIs, system properties and OSs, we implemented two testing programs as our benchmarks, available on the web site [40]. One benchmark tests the performance of rendering primitives, while the other focuses on the performance of the Bomberman game, which is measured by two metrics: *frame rate* and *Anomaly*. All the benchmarks were performed on two computers with roughly equivalent computing power. Since there are numerous combinations of the five factors, we introduce a five-tuple identifier to represent each combination, called *RE*. In the remaining part of this section, we will briefly present our experiments.

3.1. Test programs

We implemented two test programs, available on our web site [40]. One is a *micro-benchmark* and the other is a *macro-benchmark*. The micro-benchmark program opens a 600×300-sized window and counts the number of times an image, text or figure is rendered within a given time. The image test is further divided into six subtests, including opaque images, transparent images, translucent images, runtime opaque images, runtime transparent images and runtime translucent images. Each subtest renders as many corresponding 110×110-sized images as possible in a given time. The text test has two subtests: simple texts (using the word 'Running') and articles (consisting of about 13 000 characters on the screen, including 1562 different characters in Chinese, English and other languages). The font size in both tests is 12. In addition, in order to decide the performance of our text engine, the rendering speeds of texts with different font sizes, from 10 to 64, are also measured. The figure test includes 12 subtests that draw lines, polylines, polygons, rectangles, round

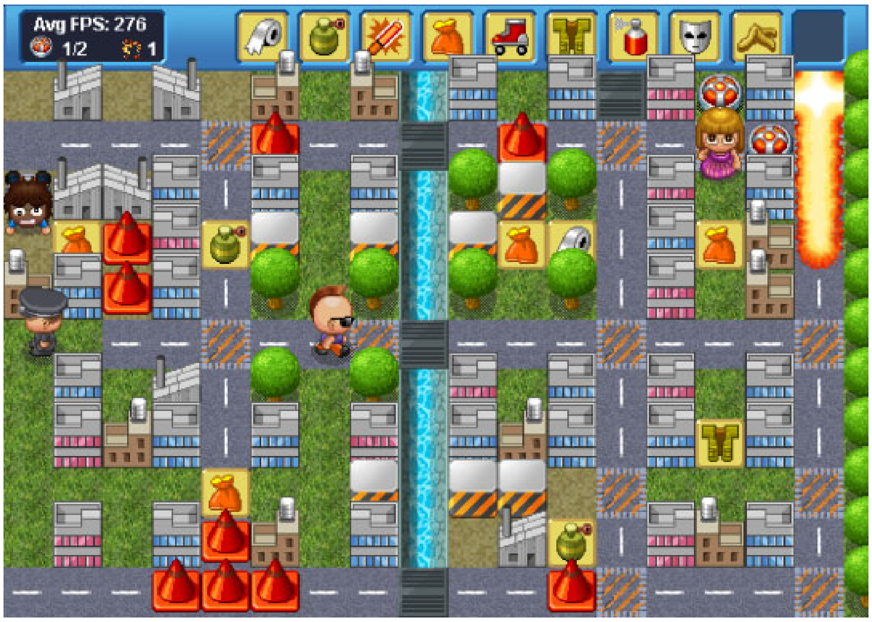


Figure 5. A screenshot of the Bomberman game.

rectangles, arcs, ovals, solid polygons, solid rectangles, solid round rectangles, pies and solid ovals. The metric for these tests is ‘rendered items per second’.

The macro-benchmark program is to simulate a Bomberman game, an applet game developed by [4], as shown in Figure 5. The panel size of the game is 560×395. On average, the game draws 196 opaque images, 122 transparent images and 14 text characters in each frame. Among the transparent images, about 58 are runtime images, which are dynamically created during runtime. We measured the average frame rate of the Bomberman game in rendering 20 000 frames.

Since we want to shorten and simplify the analysis part of this paper, we only focus on the macro-benchmark. The results of the micro-benchmark are reported on our web site [40], instead.

Both the benchmarks use double buffering to avoid flickering. The programs first rendered items into a back buffer and then copied the back buffer to the front buffer, which was shown on the screen.

Since game programmers normally try to optimize the frame rates of their games by using different combinations of image APIs, we also measure the rendering performance of different combinations of image APIs, as shown in Table III, four APIs for creating back buffers and three APIs for creating runtime images for dynamic processing. In order to simplify the names of the image APIs, we abbreviate these sets of APIs in the remaining paper. The APIs for back buffers are identified by Img (Java 1.0/1.1 Image), Cpt (Compatible Image), Vlt (Volatile Image) and CptVlt (Compatible Volatile Image), while the APIs for runtime images are identified by Img, Cpt and CptVlt. Since it is possible to test all the cases of choosing APIs for back buffers and for runtime images, there are in total 12 test cases of choosing these APIs.

According to JDK documents [33,34], some system properties allow programmers to customize how Java 2D performs rendering operations. Therefore, we also specified these system properties

Table III. Image APIs tested in the benchmarks.

Usage	API	ID	JDK
Back buffers	`Component.createImage(width, height)`	Img	1.0∼
	`Component.createVolatileImage(width, height)`	Vlt	1.4∼
	`GraphicsConfiguration.createCompatibleImage(width, height)`	Cpt	
	`GraphicsConfiguration.createCompatibleVolatileImage(width, height)`	CptVlt	
Runtime images	`Toolkit.createImage(imageProducer)`	Img	1.0∼
	`GraphicsConfiguration.createCompatibleImage(width, height, transparency)`	Cpt	1.4∼
	`GraphicsConfiguration.createCompatibleVolatileImage(width, height, transparency)`	CptVlt	1.5∼

Table IV. System properties for SystemProperty ∈ Special [33,34].

OS	System properties
Windows XP & Vista	• `sun.java2d.translaccel=true` and `sun.java2d.ddforcevram=true` Specify if translucent images should be hardware-accelerated when DirectX pipeline is in use.
Fedora	• `sun.java2d.pmoffscreen=true` or `false` Specify whether Java 2D stores images in pixmaps when DGA is not available.
Mac OS X	• `apple.awt.graphics.EnableQ2DX=true` in J2SE 1.4 Use hardware acceleration to speed up rendering of images, lines, rectangles and characters. • `apple.awt.graphics.UseQuartz=false` in J2SE 5.0 and beyond Use Sun's 2D renderer instead of Apple's 2D renderer.

Table V. System hardware, configuration and OSs.

Computer	Hardware	OS	Graphics Card Driver
1	• AMD X2 3800+2.0 GHz • 1 GB DDR 400 • ATI Radeon X1650 with 256 MB GDDR2 AGP	Windows XP Professional SP2 Windows Vista Business Fedora Core 6	ATI Catalyst 8.4
2	• Intel Core 2 Due 2.0 GHz • 1 GB DDR2 667 • ATI Mobility Radeon X1600 with 128 MB GDDR3 PCIe	Mac OS X 10.4.11	Bundled driver

Table VI. JRE versions in the benchmarks.

	OS		
	Windows XP and Vista	Fedora	Mac OS X
MSVM (Java 1.1.4)	5.0.0.3810	N/A	N/A
Sun Java 1.1	1.1.8_10	N/A	N/A
Sun J2SE 1.2	1.2.2_17	N/A	N/A
Sun J2SE 1.3	1.3.1_20	1.3.1_20	1.3.1_16
Sun J2SE 1.4	1.4.2_17	1.4.2_17	1.4.2_16
Sun J2SE 5.0	1.5.0_15	1.5.0_15	1.5.0_13
Sun Java SE 6	1.6.0_05	1.6.0_05	N/A

when running our benchmarks and selected the most significant parts that influenced the rendering performance of the benchmarks most, as shown in Table IV.

3.2. System configuration

We performed our benchmarks on four OSs, including Windows XP Professional SP2, Windows Vista Business, Fedora Core 6 and Mac OS X 10.4.11, which were chosen according to the population percentages shown in Table II.

In order to make a fair comparison, we used two computers with roughly equivalent computing power to install the four OSs, as shown in Table V. For the hardware part, Computer 1 is a desktop PC with AMD X2 3800+2.0 GHz, 1 GB DDR 400 RAM and ATI Radeon X1650 with 256 MB GDDR2 via AGP bus, while Computer 2 is an iMac with Intel Core 2 Due 2.0 GHz, 1 GB DDR2 667 RAM and ATI Mobility Radeon X1600 with 128 MB GDDR3 via PCIe bus. As for the OSs, Computer 1 has Windows XP Professional, Windows Vista Business and Fedora Core 6 installed with ATI Catalyst 8.4. Computer 2 has Mac OS X 10.4.11 installed with bundled graphics card driver. Both computers worked in true color mode and disabled font anti-aliasing.

We installed most of the popular JREs on these OSs. The versions of the JREs are given in Table VI. However, we did not perform the benchmarks in JRE 1.1 and 1.2 on Fedora and Mac OS X, since we could not successfully configure these old versions.

Since JOGL is still under development, there are several release builds available on the web site [21]. The release build we used to run the benchmarks in this paper was JSR-231 1.1.1-rc8.

3.3. Rendering environments (REs)

In this paper, we ran test programs in all the *REs* with the combination of using different JREs, image APIs, system properties and OSs. In order to easily point out which *RE* we are referring to, we identify each *RE* by an identifier, a tuple of five attributes (*Toolkit*, *JRE*, *ImageAPI*, *SystemProperty*, *OS*).

- $\mathit{Toolkit} \in \{\mathrm{AWT}, \mathrm{CWT-DX}, \mathrm{CWT-GL}\}$. 'AWT' represents the Java AWT graphics library, 'CWT-DX' represents the DirectX implementation of CWT in [25] and 'CWT-GL' represents the OpenGL implementation of CWT in this paper.
- $\mathit{JRE} \in \{\mathrm{MSVM}, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6\}$. 'MSVM' denotes Microsoft Java VM [38] and '1.1' to '1.6' denote Sun JRE version 1.1 to 1.6, respectively.
- $\mathit{ImageAPI} \in \{\mathit{BackBufferAPI} + \mathit{RuntimeImageAPI}\}$, where $\mathit{BackBufferAPI} = \{\mathrm{Img}, \mathrm{Vlt}, \mathrm{Cpt}, \mathrm{CptVlt}\}$ and $\mathit{RuntimeImageAPI} = \{\mathrm{Img}, \mathrm{Cpt}, \mathrm{CptVlt}\}$, as shown in Table III. Therefore, there are in total 12 combinations tested in this paper.
- $\mathit{SystemProperty} \in \{\mathrm{None}, \mathrm{OpenGL}, \mathrm{Special}\}$. 'None' represents the case that no system properties are specified for JREs, 'OpenGL' denotes the case that OpenGL pipeline is enabled (by setting the system property `sun.java2d.opengl` to `true`) and 'Special' refers to the system properties specified for JREs according to Table IV, which follows the hints in [33,34].
- $\mathit{OS} \in \{\mathrm{XP}, \mathrm{Vista}, \mathrm{Fedora}, \mathrm{MacOS}\}$. 'XP', 'Vista', 'Fedora' and 'MacOS', respectively, represent the following OSs, Windows XP, Windows Vista, Fedora Core 6 and Mac OS X 10.4.11.

For example, $\mathit{RE}(\mathrm{AWT}, 1.6, \mathrm{Vlt}+\mathrm{Cpt}, \mathrm{OpenGL}, \mathrm{XP})$ refers to the *RE* that uses AWT, runs in JRE version 6, chooses the method `createVolatileImage()` in `Component` class for back buffers and the method `createCompatibleImage()` in `GraphicsConfiguration` class for runtime images, enables OpenGL pipeline and runs on Windows XP; $\mathit{RE}(\mathrm{CWT-GL}, 1.4, \mathrm{None}, \mathrm{Img}+\mathrm{Img}, \mathrm{MacOS})$ refers to the *RE* that uses CWT-GL, runs in JRE version 1.4, uses Java 1.0/1.1 image APIs for both back buffers and runtime images and runs on Mac OS X 10.4.11.

For simplicity, we introduce the wildcard character '*' to indicate a group of *REs* for all cases in the attribute. For example, $\mathit{RE}(\mathrm{AWT}, *, \mathrm{Vlt}+\mathrm{Cpt}, \mathrm{None}, \mathrm{MacOS})$ means all the *REs* with AWT and with the combinations of $\mathit{ImageAPI} \in \{\mathrm{Vlt}+\mathrm{Cpt}\}$ and $\mathit{SystemProperty} \in \{\mathrm{None}\}$ on Mac OS X.

3.4. Definitions of metrics

To measure the rendering performance of the macro-benchmark, 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 an *RE* r, *FrameRate*(r) denotes the frame rate in r. *Anomaly* for a set of *RE*s, say R, is defined as follows:

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

The metrics *Anomaly* is defined specifically for the worst case, which could happen out of programmers' expectation. Since programmers may believe that the rendering performance of Java is also similar when porting to other *REs*, they may optimize their games by only testing in some limited *REs*. However, the users who use other *REs* may experience much worse rendering performance. Therefore, we use the above definition for *Anomaly*, instead of some other metrics such as standard deviation.

4. ANALYSIS

This paper focuses on analyzing the results of the macro-benchmark, as shown in Table AI in Appendix A. As mentioned in Section 3.1, results of the micro-benchmark are listed on our web site [40] in order to shorten the length of this paper.

In order to simplify our analysis in advance, we select different aspects of the results that can represent the performance inconsistency of Java AWT. The three chosen aspects are (1) grouped by OSs, (2) grouped by the combinations of image APIs and system properties and (3) grouped by JREs.

In the following sections, we first summarize the results in the three aspects mentioned above. Next, we discuss the results and our suggestions for cross-platform Java game development.

4.1. Summary of macro-benchmark results

We summarize the results of Java AWT as follows.

- *The rendering performance of Java AWT is inconsistent among the four OSs.* First of all, we compare the rendering performance of a Java 1.1 program among the four OSs. In this case, Java 1.0/1.1 image APIs are used, i.e. $ImageAPI \in \{\text{Img}+\text{Img}\}$. As shown in Figure 6, Fedora normally delivers much slower frame rates than those on other OSs, which is the main source of performance inconsistency among the four OSs. For all $jre \in \{1.3, 1.4, 1.5, 1.6\}$, $Anomaly(\text{AWT}, jre, \text{Img}+\text{Img}, \text{None}, *)$ ranges from 3.06 to 9.10. This means that the rendering performance of the Java 1.1 program would be quite different on the four OSs, especially on Fedora.

  This phenomenon also exists when we use new image APIs and system properties available since JRE 1.4, as shown in Figures 7 and 8. In the combinations of $SystemProperty \in \{\text{None}, \text{Special}\}$, the frame rates on Fedora are still slower than those on other OSs by a factor of 2.33 to 5.10.

  In Figure 9, when OpenGL pipeline is enabled ($SystemProperty \in \{\text{OpenGL}\}$), using the combinations of $ImageAPI \in \{\text{CptVlt}+\text{Cpt}, \text{CptVlt}+\text{CptVlt}, \text{Vlt}+\text{Cpt}, \text{Vlt}+\text{CptVlt}\}$ in JRE 1.6, achieves high and consistent frame rates on all OSs. However, the OpenGL pipeline is not reliable enough since it renders incomplete screens in some of these *RE*s during our test. Therefore, if the OpenGL pipeline of Java AWT is not considered (that is, $SystemProperty \in \{\text{None}, \text{Special}\}$), the inconsistent performance among different OSs exists even when we use new image APIs and system properties to tune up the Bomberman game.
- *The rendering performance of using different combinations of image APIs and system properties is inconsistent.* Typically, using different image APIs results in different rendering

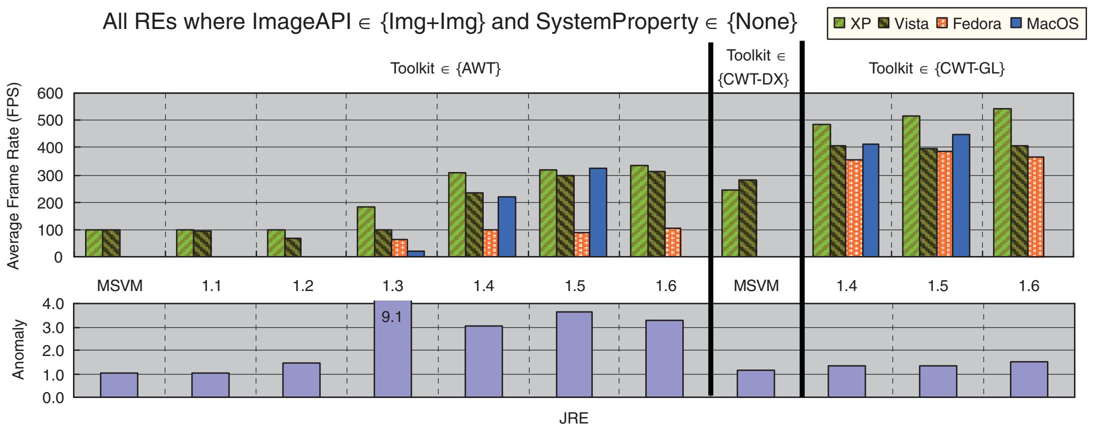


Figure 6. Frame rates and Anomaly among OSs using Java 1.0/1.1 image APIs.

performance. As mentioned in Sections 3.1 and 3.3, Java AWT provides several types of image APIs and system properties for tuning up the performance. However, we observe that the image APIs and system properties have irregular impacts on the rendering performance in different *REs*. In other words, programmers may need to use different combinations of image APIs and system properties in different *REs* to achieve the best rendering performance.

Table VII summarizes the best combinations of image APIs and system properties for achieving the best frame rates in given *REs* and Figure 10 shows the results of comparing different combinations of Image APIs and system properties. From these results, we do not find a combination of image APIs and system properties that can deliver *high* and *consistent* frame rates in all *REs*, as illustrated in the following examples. (a) On Windows XP, Vista and Fedora, using the combinations of $ImageAPI \in \{\text{Img+Img, Cpt+Img}\}$ and $SystemProperty \in \{\text{None, Special}\}$ often achieves the best results. (b) However, using the combinations of $ImageAPI \in \{\text{CptVlt+Cpt, Vlt+CptVlt}\}$ and $SystemProperty \in \{\text{Special}\}$ instead in JRE 1.6 on Windows XP delivers 1.57 times faster frame rate. (c) As for Mac OS X, programmers should use the combinations of $ImageAPI \in \{\text{Vlt+Cpt}\}$ to achieve the best frame rates. (d) Moreover, when OpenGL pipeline enabled ($SystemProperty \in \{\text{OpenGL}\}$), the combinations for the best frame rates are also different from those above. The combinations of $ImageAPI \in \{\text{Img+Cpt, CptVlt+Cpt, Vlt+Cpt}\}$ achieve the best frame rates in JRE 1.5, while the combinations of $ImageAPI \in \{\text{CptVlt+CptVlt, Vlt+CptVlt}\}$ achieve the best frame rates in JRE 1.6. (e) Even worse, using wrong combinations of image APIs with the OpenGL pipeline would cause serious consequences. The frame rates may become as low as only two FPS.

Therefore, the inconsistent rendering performance of the combinations of image APIs and system properties makes it difficult to decide which combinations should be used in developing cross-platform Java games.

- *The rendering performance of Java AWT is inconsistent among commonly used JREs.* As mentioned in Section 1.1, Java 2D rendering pipelines evolve over JREs. Therefore, it is normal that the rendering performance is inconsistent among different JREs. However, since the old JREs are still used by a portion of users, programmers should take the inconsistency of rendering performance among JREs into consideration.

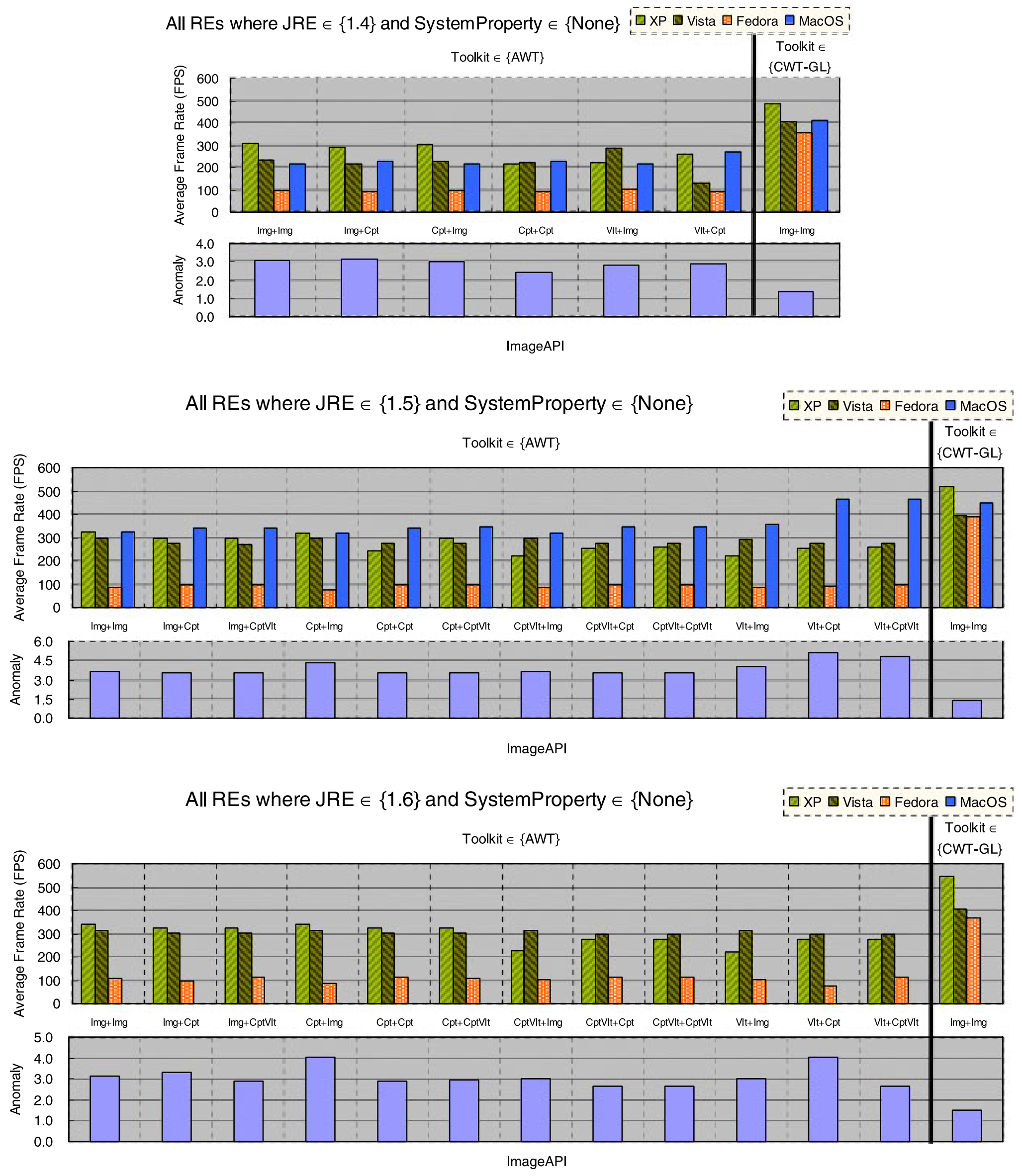


Figure 7. Frame rates and Anomaly among OSs using SystemProperty ∈ {None}.

As shown in Figure 11, older JREs normally delivered worse frame rates, especially MSVM. In the case of using Java 1.0/1.1 image APIs, for all $os \in \{\text{XP}, \text{Vista}, \text{Fedora}, \text{MacOS}\}$, $\textit{Anomaly}(\text{AWT}, *, \text{Img+Img}, \text{None}, os)$ ranges from 1.18 to 3.31.

When new image APIs are used, the performance inconsistency among JREs also exists. For example, programmers may tune their programs to achieve very high frame rates

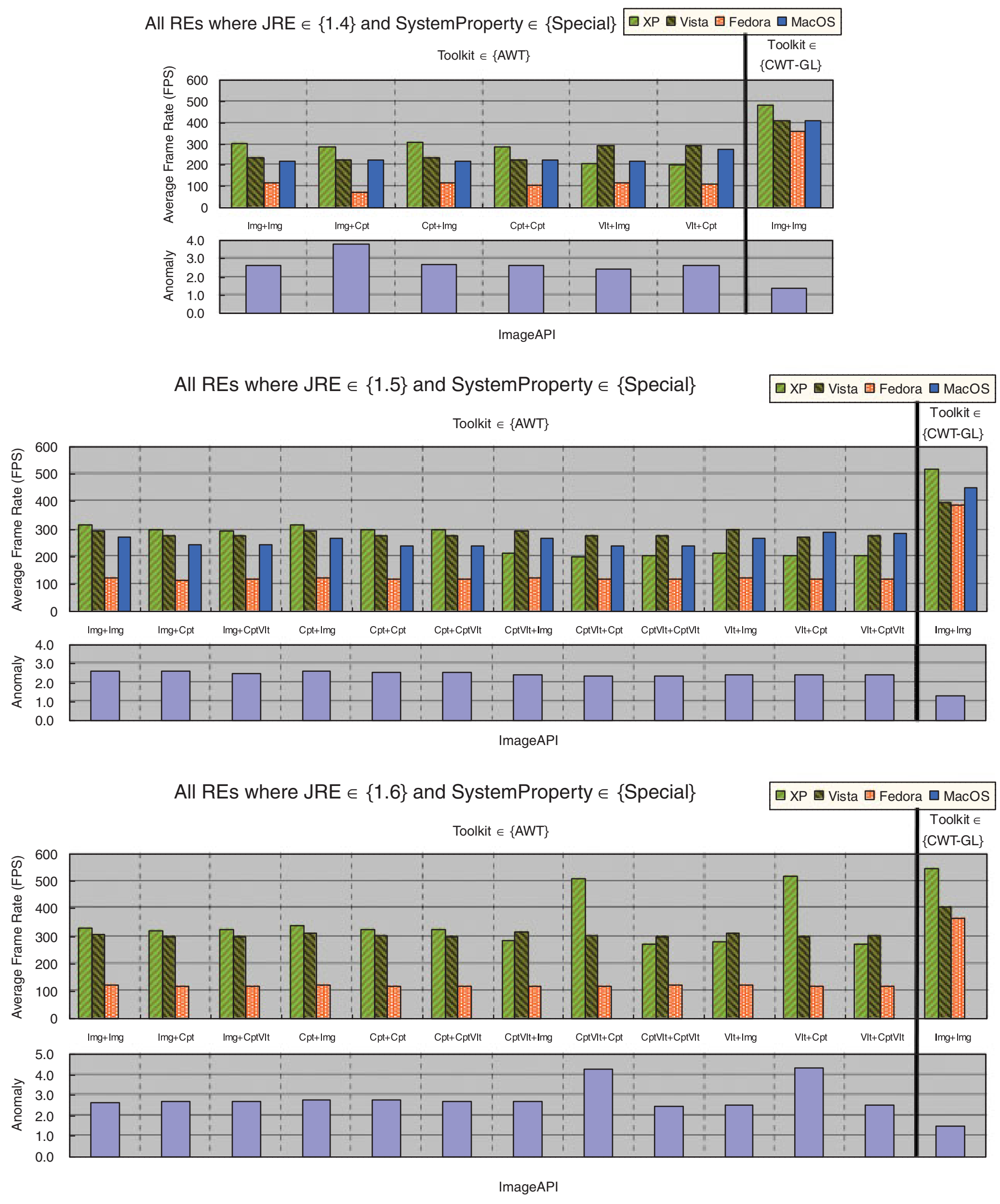


Figure 8. Frame rates and Anomaly among OSs using SystemProperty ∈ {Special}.

in certain JREs, such as *RE*(AWT, 1.5, Vlt+Cpt, None, MacOS) and *RE*(AWT, 1.6, Vlt+Cpt, Special, XP), but the same programs would not perform as well in other JREs. For example, in our benchmark, *Anomaly*(AWT, $*$, Vlt+Cpt, None, MacOS) and *Anomaly*

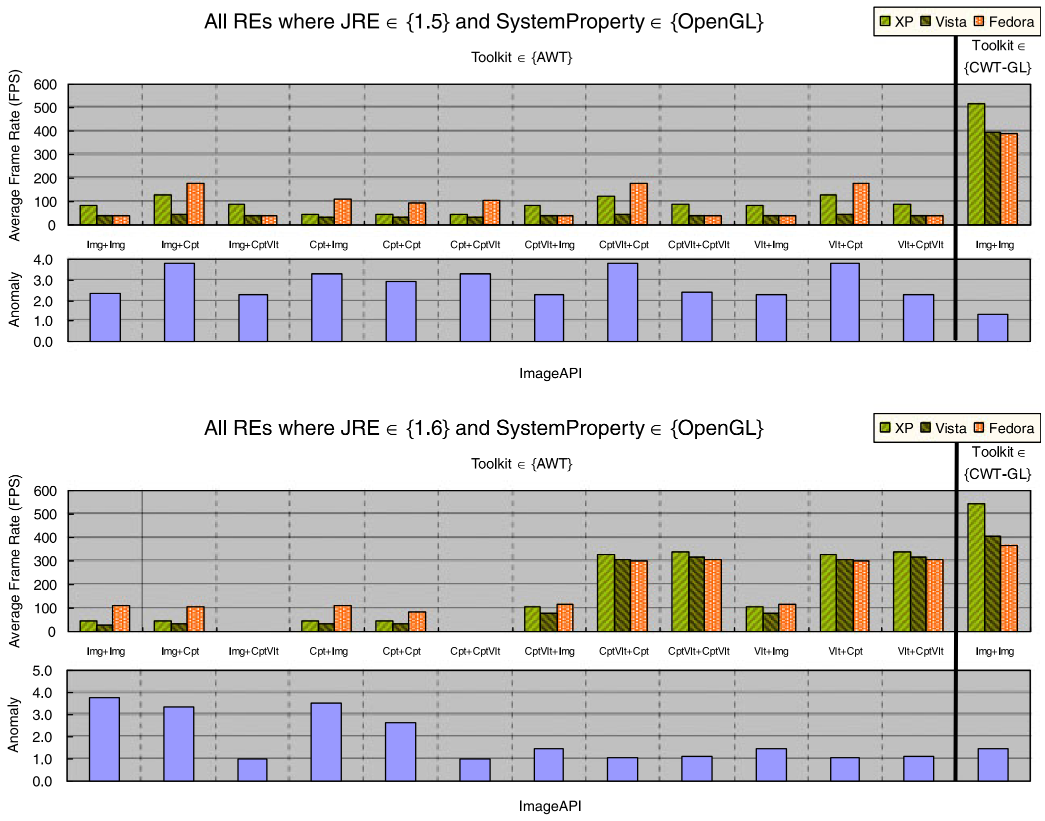


Figure 9. Frame rates and Anomaly among OSs using SystemProperty ∈ {OpenGL}.

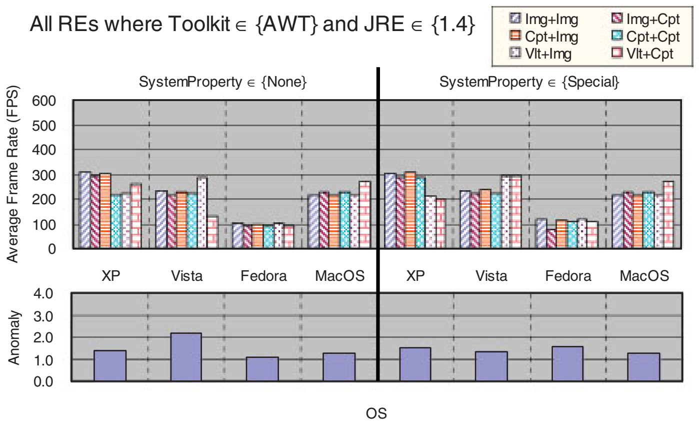
(AWT, ∗, Vlt+Cpt, Special, XP) are 1.71 and 2.60, respectively. Therefore, since old JREs are still used, programmers need to deal with the performance inconsistency among JREs.

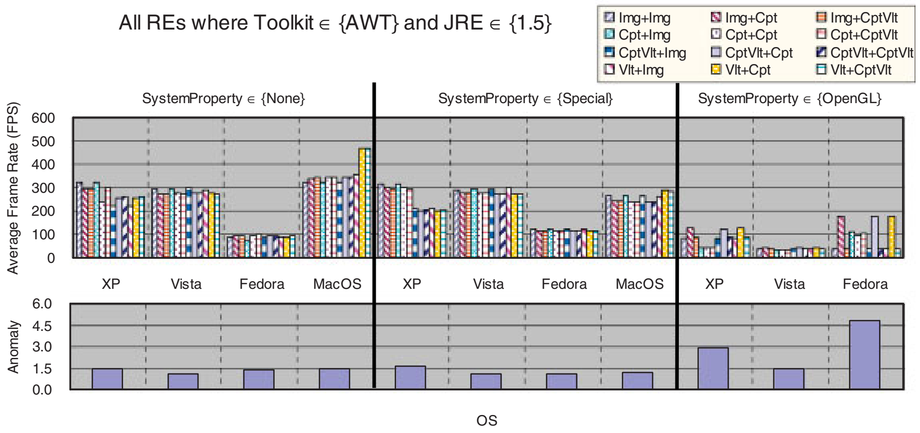
To sum up the results of Java AWT, we find it difficult to optimize the rendering performance in the combinations of JREs, image APIs, system properties and OSs for cross-platform Java games. In order to solve the problem of performance inconsistency among different *REs*, we try to use a number of different combinations of image APIs and system properties. However, according to our experimental results, we find no combinations that can achieve high, consistent and reliable rendering performance among all *REs*. When optimizing the rendering performance for one *RE*, we observe that the same program would perform differently in other *RE*s. Thus, efforts for performance testing are required for programmers to develop cross-platform Java games requiring consistently high rendering performance.

It is even worse that some of the parameters, such as JRE versions, system properties and OSs, 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.

Table VII. Combinations of image APIs, which deliver the highest frame rates in given REs.

		OS			
JRE	System property	Windows XP	Windows Vista	Fedora	Mac OS X
1.4	None	Img+Img, Cpt+Img	Vlt+Img	Img+Img, Cpt+Img, Vlt+Img	Vlt+Cpt
	Special	Img+Img, Cpt+Img	Vlt+Img, Vlt+Cpt	Img+Img, Cpt+Img, Vlt+Img	Vlt+Cpt
1.5	None	Img+Img, Cpt+Img	Img+Img, Cpt+Img CptVlt+Img	Img+Cpt, Img+CptVlt, Cpt+Cpt, Cpt+CptVlt, CptVlt+Cpt, CptVlt+CptVlt	Vlt+Cpt, Vlt+CptVlt
	Special	Img+Img, Cpt+Img	Img+Img, Cpt+Img, CptVlt+Img, Vlt+Img	Img+Img, Cpt+Img, CptVlt+Img, Vlt+Img	Vlt+Cpt, Vlt+CptVlt
	OpenGL	Img+Cpt, CptVlt+Cpt, Vlt+Cpt	Img+Cpt, CptVlt+Cpt, Vlt+Cpt	Img+Cpt, CptVlt+Cpt, Vlt+Cpt	N/A
1.6	None	Img+Img, Cpt+Img	Img+Img, Cpt+Img, CptVlt+Img, Vlt+Img	Img+Img, Img+CptVlt, Cpt+Cpt, Cpt+CptVlt, CptVlt+Cpt, CptVlt+CptVlt, Vlt+CptVlt	N/A
	Special	CptVlt+Cpt, Vlt+CptVlt	Img+Img, Cpt+Img, CptVlt+Img, Vlt+Img	Img+Img, Cpt+Img, CptVlt+CptVlt, Vlt+Img	N/A
	OpenGL	CptVlt+CptVlt, Vlt+CptVlt	CptVlt+CptVlt, Vlt+CptVlt	CptVlt+CptVlt, Vlt+CptVlt	N/A





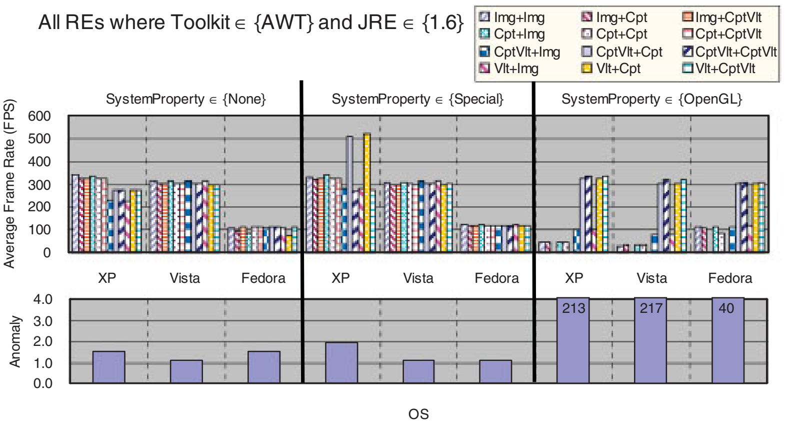


Figure 10. Frame rates and Anomaly on choosing different image APIs.

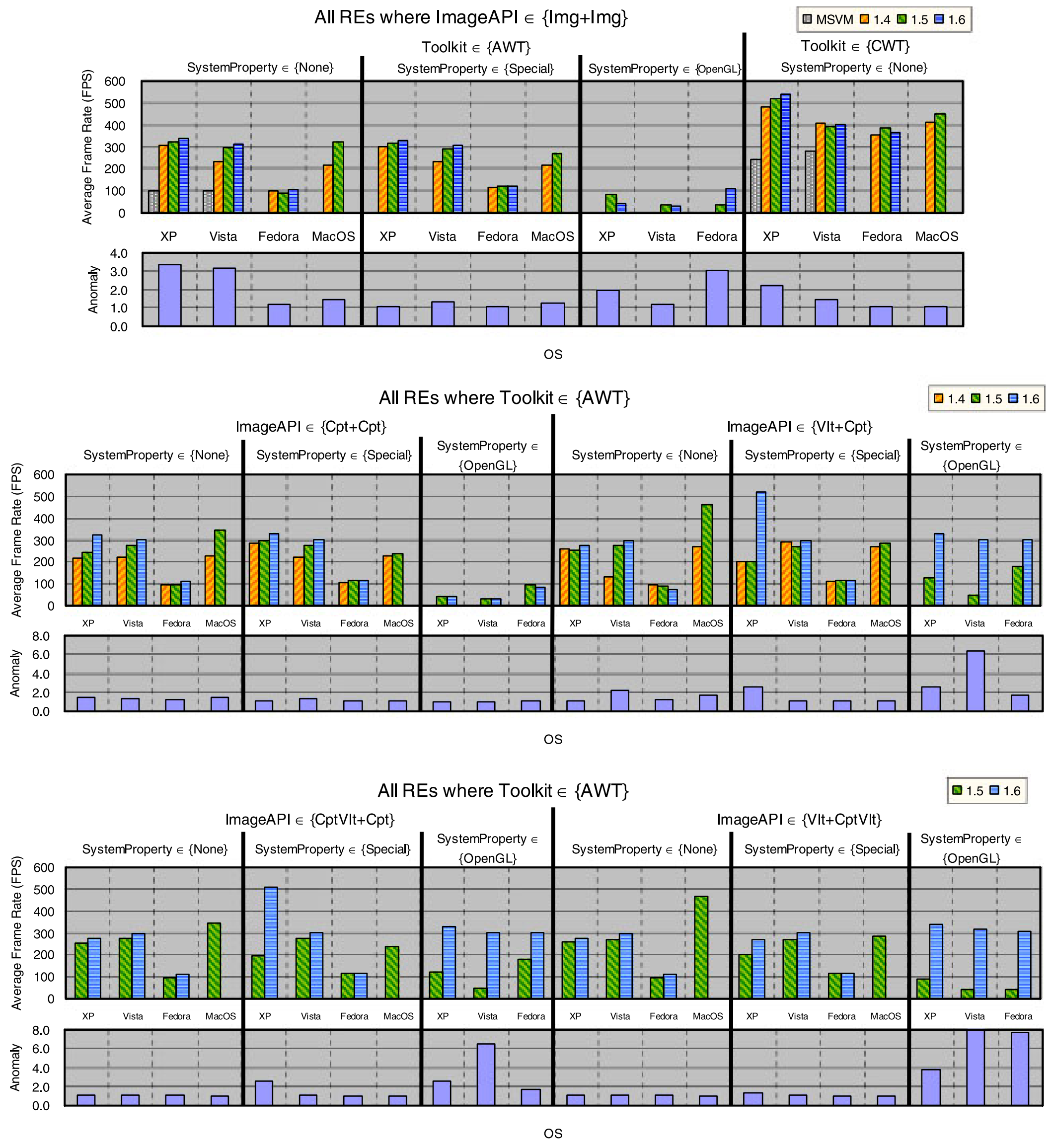


Figure 11. Frame rates and Anomaly in commonly used JREs.

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 Figures 6–9, CWT-GL often delivers the highest frame rates, and also more consistent rendering performance than Java AWT. For example, for all $jre \in \{1.4, 1.5, 1.6\}$, $Anomaly(\mathrm{CWT-GL}, jre, \mathrm{Img+Img}, \mathrm{None}, *)$ ranges from 1.34 to 1.49, while *Anomaly*(AWT, *jre*, Img + Img, None, $*$) ranges from 3.06 to 3.64. Therefore, CWT-GL performs more consistently than AWT does among the four OSs.

- *CWT-GL needs fewer efforts to test the combinations of image APIs and system properties.* CWT-GL supports Java 1.0/1.1 image APIs and requires no system properties before the startup of programs. Therefore, the efforts to optimize the rendering performance in all of the *RE*s can be greatly reduced. In fact, although only old image APIs are supported, CWT-GL still achieves almost the highest rendering performance when compared with the Java AWT, which has a number of image APIs and system properties to tune up the rendering performance.
- *CWT delivers higher and more consistent rendering performance in the set of JRE*{MSVM, 1.4, 1.5, 1.6}, which covers most Web users. It is important that games deliver high and consistent rendering performance in commonly used JREs. As shown in Figure 11, CWT achieves the best frame rates in the set of $JRE \in \{MSVM, 1.4, 1.5, 1.6\}$. Meanwhile, CWT also delivers more consistent frame rates than Java AWT does. For example, for all $os \in$ {XP, Vista, Fedora, MacOS}, *Anomaly*(CWT, $*$, Img + Img, None, *os*) ranges from 1.08 to 2.22, while *Anomaly*(AWT, $*$, Img + Img, None, *os*) ranges from 1.18 to 3.31.

Generally speaking, the rendering performance of CWT-GL is higher and more consistent on supported *RE*s than those in Java AWT. This is quite important especially when games run in users' computers with various *RE*s. Furthermore, the image APIs and system properties in CWT-GL are simpler than Java AWT, which also helps to 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 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 section, we further discuss the problems of current Java 2D rendering pipelines including implementations on the four OSs. Then, we give suggestion for developing Java 2D rendering pipelines. Finally, the limitations of CWT are presented.

4.2.1. Problems of current Java 2D rendering pipelines

This section discusses the problems of different graphics systems on the four OSs and corresponding implementations of Java 2D rendering pipelines. These include Window GDI and DirectX on Microsoft Windows platforms, *DWM* on Microsoft Windows Vista, *X Window System* (X) on Fedora, *Quartz graphics layer* (Quartz) on Mac OS X and OpenGL on all of the four OSs.

On Microsoft Windows platforms, the main rendering pipelines of Java AWT/Swing rely on GDI and DirectX. GDI was used in Java AWT since Java 1.0/1.1, while DirectX was introduced since J2SE 1.4 to greatly improve the rendering performance of Java AWT/Swing. In addition, Windows Vista has a new graphics system called DWM, which runs on top of Direct3D instead of GDI. In order to make Java programs run well with DWM, Sun Microsystems introduced some changes to the Java 2D rendering pipelines [56]. Consequently, these changes altered the rendering performance on Windows Vista.

On Fedora, Java AWT/Swing is built on X, which is designed according to the client–server model so as to operate over network. When the X server and X clients are located in the same machine, *shared memory extension* (SHM) is introduced into X to allow them to jointly access shared memory rapidly. According to the macro-benchmark results, when using SHM, the Bomberman game delivered on an average 20% more frames per second. However, the frame rate was still much slower than that on the other three OSs, which is due to the lack of full hardware acceleration on the graphics system. This is an example of low rendering performance when Java games run in the different rendering pipelines on different OSs.

On Mac OS X, Apple has its own peer implementation of Java AWT atop Quartz. Therefore, the performance factors are different from those on Windows platforms and Fedora, especially for the system properties [34]. Properly configuring the behaviors of Quartz may improve the rendering performance, since some rendering operations are anti-aliased [34].

For cross-platform Java games, OpenGL is available on all of the four OSs. According to our benchmarking results, the OpenGL pipeline of Java AWT/Swing has shown its potential on cross-platform Java game development. For example, Figure 9 shows that the OpenGL pipeline in some cases delivered equivalent frame rates on Windows platforms and Fedora. However, the OpenGL pipeline may deliver very poor frame rates in other cases when different image APIs are used. This shows inconsistent rendering performance of the OpenGL pipeline when Java games use improper image APIs. In contrast, CWT-GL achieves high and consistent rendering performance on all of the four 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 [18]. 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 the 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 X 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.

4.2.2. *Decoupling of Java rendering pipelines from JREs*

According to the discussion in the previous section, we suggest that the rendering pipeline of Java AWT/Swing should be decoupled from JREs, since both Java AWT and Swing provide the capability of extension, for example, implement new peers of AWT to replace those in the old JREs, such as CWT, or replace the repaint manager of Swing, such as Agile2D. Once the rendering pipelines are decoupled, they are independent of the JRE versions and can be applied to the older JREs. To sum up, the benefits of decoupling the graphics part from the JRE are listed as follows.

(1) The rendering pipelines can be applied to the older JREs.
(2) The rendering pipelines are easier to upgrade due to smaller size when compared with the whole JRE.

(3) New features and bug fixes of the rendering pipelines can be released faster (without waiting for new JRE releases).
(4) The responsibility of performance is shifted to supporting graphics libraries such as CWT.
(5) JRE developers such as Sun Microsystems can focus on other design issues.

4.2.3. Drawbacks of CWT

Although CWT seems ideal as described in the above subsections, this subsection lists the potential drawbacks of CWT as follows:

- CWT is not designed for general-purpose applications. For example, CWT-GL needs modern video cards with 3D hardware acceleration for delivering better results.
- When no hardware acceleration is available, the CWT switches to use CWT-AWT implementation, which incurs extra overhead (about 10.3% [25]). For example, when neither FBO nor pbuffer is available, CWT-GL will use Java AWT internally to perform off-screen rendering.
- Using CWT will not benefit from any additional features supported by new Java versions, since currently we implement only Java AWT/Swing 1.1 compatible API.

In order to access the hardware acceleration via JNI, applets using CWT-GL need to be signed and acquire permissions from users when executed in Web browsers.

5. CONCLUSIONS

This paper designs an OpenGL version (CWT-GL) of the CWT architecture defined in [25] using JOGL and compares its rendering performance with that of Java AWT on four OSs, including Windows XP, Windows Vista, Fedora and Mac OS X. The results indicate that CWT-GL generally reaches higher and more consistent rendering performance in J2SE 1.4 to 6 on the four OSs. Furthermore, it also helps to reduce the efforts of tuning the rendering performance by choosing different image APIs and system properties.

The contributions of this paper that differ from [25] are as follows:

- Evaluate the rendering performance of the original Java AWT with different combinations of JREs, image APIs, system properties and OSs. 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. This concludes that programmers can hardly optimize the rendering performance of Java AWT using different combinations of image APIs and system properties for mostly used JREs on the four OSs. This weakens the merit of WORA of Java for game development.
- Implement an OpenGL version of CWT [25] via JOGL, which takes advantage of 3D hardware acceleration on multiple OSs. Compared with the Java AWT, CWT-GL achieves more consistent and higher rendering performance in JRE 1.4 to 1.6 on the four tested OSs.
- The experimental results also reveal two points: (1) the rendering pipelines of Java AWT/Swing should be decoupled from the JREs for higher and more consistent rendering performance, faster upgrades and better support of the old JREs, and (2) programmers can use other graphics libraries, such as CWT, instead of Java AWT/Swing to develop cross-platform Java games with higher and more consistent rendering performance.

We have established a web site [40] for releasing the latest implementations of CWT. The benchmark programs and results are also available on the web site, as well as the demonstrations and a porting guide.

Future extension includes more optimizations on CWT-GL and the cooperation with JOGL to apply our work to 3D game development in Java. We will also introduce more game-related features into CWT, such as animations and visual effects, to further improve the usability of CWT.

APPENDIX A

The results of macro-benchmark is given in Table AI.

Table AI. Average frame rate (in FPS) of the Bomberman game.

				OS			
Toolkit	JRE	System property	Image API	Windows XP	Windows Vista	Fedora Core 6	Mac OS X 10.4.11
AWT	MSVM	None	Img+Img	101	99	N/A	N/A
	1.1	None	Img+Img	98	94	N/A	N/A
	1.2	None	Img+Img	98	67	N/A	N/A
	1.3	None	Img+Img	182	99	62	20
	1.4	None	Img+Img	306	232	100	218
			Img+Cpt	293	217	94	225
			Cpt+Img	301	227	100	216
			Cpt+Cpt	217	222	94	227
			Vlt+Img	220	288	101	217
			Vlt+Cpt	258	132	94	271
		Special	Img+Img	304	234	116	217
			Img+Cpt	285	224	75	225
			Cpt+Img	307	237	115	217
			Cpt+Cpt	286	222	108	226
			Vlt+Img	210	290	118	217
			Vlt+Cpt	200	292	110	272
	1.5	None	Img+Img	322	295	88	322
			Img+Cpt	295	274	97	341
			Img+CptVlt	296	272	96	343
			Cpt+Img	321	295	74	321
			Cpt+Cpt	242	277	97	343
			Cpt+CptVlt	299	273	98	345
			CptVlt+Img	220	299	88	320
			CptVlt+Cpt	256	275	97	344
			CptVlt+CptVlt	259	278	97	345
			Vlt+Img	222	291	88	356
			Vlt+Cpt	256	275	91	464
			Vlt+CptVlt	259	273	97	467
		Special	Img+Img	318	291	121	269
			Img+Cpt	299	277	115	244
			Img+CptVlt	294	277	117	245
			Cpt+Img	317	294	120	265

Table AI. *Continued.*

Toolkit	JRE	System property	Image API	OS			
				Windows XP	Windows Vista	Fedora Core 6	Mac OS X 10.4.11
			Cpt+Cpt	299	277	116	240
			Cpt+CptVlt	297	277	116	241
			CptVlt+Img	212	294	121	265
			CptVlt+Cpt	199	274	116	240
			CptVlt+CptVlt	205	273	117	237
			Vlt+Img	211	298	122	264
			Vlt+Cpt	202	271	117	287
			Vlt+CptVlt	204	273	117	285
		OpenGL	Img+Img	∗86	37	37	N/A
			Img+Cpt	127	47	178	N/A
			Img+CptVlt	90	39	40	N/A
			Cpt+Img	45	∗33	110	N/A
			Cpt+Cpt	44	32	94	N/A
			Cpt+CptVlt	44	32	105	N/A
AWT			CptVlt+Img	85	38	37	N/A
			CptVlt+Cpt	124	47	178	N/A
			CptVlt+CptVlt	90	39	37	N/A
			Vlt+Img	∗85	38	37	N/A
			Vlt+Cpt	127	47	178	N/A
			Vlt+CptVlt	89	39	40	N/A
	1.6	None	Img+Img	340	315	108	N/A
			Img+Cpt	325	301	98	N/A
			Img+CptVlt	323	304	111	N/A
			pt+Img	339	316	84	N/A
			Cpt+Cpt	324	303	111	N/A
			Cpt+CptVlt	326	301	110	N/A
			CptVlt+Img	227	312	104	N/A
			CptVlt+Cpt	274	300	112	N/A
			CptVlt+CptVlt	275	300	112	N/A
			Vlt+Img	224	312	104	N/A
			Vlt+Cpt	274	296	73	N/A
			Vlt+CptVlt	277	297	112	N/A
		Special	Img+Img	331	307	124	N/A
			Img+Cpt	319	296	118	N/A
			Img+CptVlt	324	296	119	N/A
			Cpt+Img	340	311	122	N/A
			Cpt+Cpt	327	301	117	N/A
			Cpt+CptVlt	323	296	118	N/A
			CptVlt+Img	283	316	116	N/A
			CptVlt+Cpt	510	303	119	N/A
			CptVlt+CptVlt	272	300	120	N/A
			Vlt+Img	281	313	124	N/A
			Vlt+Cpt	519	300	119	N/A
			Vlt+CptVlt	272	301	119	N/A
		OpenGL	Img+Img	43	∗30	113	N/A
			Img+Cpt	43	32	108	N/A
			Img+CptVlt	2	2	2	N/A

Table AI. *Continued.*

Toolkit	JRE	System property	Image API	OS: Windows XP	Windows Vista	Fedora Core 6	Mac OS X 10.4.11
			Cpt+Img	43	32	113	N/A
			Cpt+Cpt	43	*32	85	N/A
			Cpt+CptVlt	2	2	2	N/A
			CptVlt+Img	104	78	115	N/A
			CptVlt+Cpt	327	304	302	N/A
			CptVlt+CptVlt	338	319	308	N/A
			Vlt+Img	104	79	115	N/A
			Vlt+Cpt	327	303	*302	N/A
			Vlt+CptVlt	338	319	308	N/A
CWT-DX	MSVM	None	Img+Img	245	280	N/A	N/A
CWT-GL	1.4	None	Img+Img	484	408	357	412
	1.5	None	Img+Img	518	395	387	449
	1.6	None	Img+Img	544	405	365	N/A

The numbers with '*' mean that the screen was not rendered correctly.

ACKNOWLEDGEMENTS

The authors would like to thank the National Science Council of the Republic of China and ThinkNewIdea Inc. 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. Sun Microsystems Inc. Java Home Page. Sun Microsystems Inc. Available at: http://java.sun.com/ [8 March 2008].
2. Yahoo Inc. Yahoo! Games. Yahoo Inc. Available at: http://games.yahoo.com/ [8 March 2008].
3. IonChron Inc. Java Games. IonChron Inc. Available at: http://www.arcadepod.com/java/ [8 March 2008].
4. ThinkNewIdea Internet Technology Corp. CYC Games. ThinkNewIdea Internet Technology Corp. Available at: http://cycgame.com/ [8 March 2008].
5. Hsu C-C, Wu I-C. An event-driven framework for inter-user communication applications. *Information and Software Technology* 2006; **48**:471–483.
6. Jellyvision Inc. You don't kow Jack. Jellyvision Inc. Available at: http://www.jellyvision.com/ [8 March 2008].
7. Vivendi Games. Law & order: Dead on the money. Legacy Interactive, Inc. Available at: http://www.legacyinteractive.com [8 March 2008].
8. Oddlabs ApS. Tribal trouble. Oddlabs ApS. Available at: http://tribaltrouble.com/ [8 March 2008].
9. Jagex Ltd. RuneScape. Jagex Ltd. Available at: http://www.runescape.com/ [8 March 2008].
10. Three Rings Design, Inc. Puzzle pirates. Three Rings Design, Inc. Available at: http://www.puzzlepirates.com/ [8 March 2008].
11. Mojang Specifications. Wurm online. Mojang Specifications. Available at: http://www.wurmonline.com/ [8 March 2008].
12. Microsoft Corp. Age of empires. Micorsoft Corp. Available at: http://www.microsoft.com/games/empires/ [8 March 2008].
13. IN-FUSIO. Age of Empires II mobile. IN-FUSIO. Available at: http://www.in-fusio.com/ [8 March 2008].
14. Marner J. Evaluating Java for game development. Department of Computer Science, University of Copenhagen, Denmark, 2002; 8–9.
15. Microsoft Corp. Microsoft DirectX. Microsoft Corp. Available at: http://www.microsoft.com/windows/directx/ [8 March 2008].
16. The OpenGL Architecture Review Board. OpenGL: Open Graphics Library. The OpenGL Architecture Review Board. Available at: http://www.opengl.org/ [8 March 2008].

17. Sun Microsystems Inc. *High Performance Graphics—Graphics Performance Improvements in the Java 2 SDK*, version 1.4. Sun Microsystems Inc., 2001.
18. Sun Microsystems Inc. *New Java 2D Features in J2SE 5.0*. Sun Microsystems Inc., 2004.
19. Sun Microsystems Inc. *Update*: *Desktop Java Features in Java SE 6*. Sun Microsystems Inc., 2005.
20. Jausoft. GL4Java: OpenGL for Java. Jausoft. Available at: http://gl4java.sourceforge.net/ [8 March 2008].
21. Sun Microsystems Inc. JOGL, Java bindings for OpenGL API. Sun Microsystems Inc. Available at: https://jogl.dev.java.net/ [8 March 2008].
22. lwjgl.org. LWJGL, Lightweight Java Game Library. Available at: lwjgl.org. http://lwjgl.org/ [8 March 2008].
23. Sun Microsystems Inc. Java 3D. Sun Microsystems Inc. Available at: http://java.sun.com/products/java-media/3D/ [8 March 2008].
24. Bytonic Software. Jake2. Bytonic Software. Available at: http://www.bytonic.de/html/jake2.html [8 March 2008].
25. Wang Y-H, Wu I-C, Jiang J-Y. A portable AWT/Swing architecture for Java Game Development. *Software Practice and Experience* 2007; **37**(7):727–745.
26. Gray A. GC Usage statistics. Available at: http://www.andrew-gray.com/dist/stats.shtml [8 March 2008].
27. Sun Microsystems Inc. *Bug ID*: *5037133 Mixed mode rendering and 3D effects using Java2D and JOGL together*. Sun Microsystems Inc. Available at: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=5037133 [8 March 2008].
28. Sun Microsystems Inc. *The VolatileImage APIs User Guide*. Sun Microsystems Inc., 2001; 5–9.
29. Sun Microsystems Inc. *The AWT Native Interface*. Sun Microsystems Inc., 1999.
30. X.Org Foundation. X.Org Project. X.Org Foundation. Available at: http://www.x.org/wiki/ [8 March 2008].
31. Apple Inc. Graphics & imaging overview. Apple Inc. http://developer.apple.com/graphicsimaging/overview.html [8 March 2008].
32. Microsoft Corp. Desktop Window Manager. Microsoft Corp. Available at: http://msdn2.microsoft.com/En-US/library/aa969540.aspx [8 March 2008].
33. Sun Microsystems Inc. *System Properties for Java 2D Technology*. Sun Microsystems Inc., 2004.
34. Apple Inc. *Introduction to Java Property*, *VM Option*, *and Info.plist Key Reference for Mac OS X*. Apple Inc. Available at: http://developer.apple.com/documentation/Java/Conceptual/JavaPropVMInfoRef/index.html [8 March 2008].
35. Sun Microsystems Inc. Bug ID: 6260751 applets can't set sun.java2d.noddraw=true. Sun Microsystems Inc. Available at: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6260751 [8 March 2008].
36. W3Schools. OS platform statistics. Refsnes Data Inc. Available at: http://www.w3schools.com/browsers/browsers_os.asp [8 March 2008].
37. Sun Microsystems Inc. JSR 231: Java binding for the OpenGL API. Sun Microsystems Inc. Available at: http://jcp.org/en/jsr/detail?id=231 [8 March 2008].
38. Microsoft Corp. Microsoft Java virtual machine support. Microsoft Corp. Available at: http://www.microsoft.com/mscorp/java/ [8 March 2008].
39. Microsoft Corp. Microsoft Visual J#. Microsoft Corp. Available at: http://msdn.microsoft.com/vjsharp/ [8 March 2008].
40. Internet Application Technology Lab. CWT—CYC Window Toolkit. National Chiao-Tung University, Taiwan. Available at: http://java.csie.nctu.edu.tw/cwt [8 March 2008].
41. OpenGL Architecture Review Board, Shreiner D, Woo M, Neider J, Davis T. *OpenGL Programming Guide*: *The Official Guide to Learning OpenGL*, version 1.4 (4th edn). Addison-Wesley: Reading, MA, 2003; 37–41, 319–324, 369–372, 408–410, 454–456.
42. Advanced Micro Devices Inc. Radeon X1600 Series—GPU specifications. Advanced Micro Devices Inc. Available at: http://ati.amd.com/products/RadeonX1600/specs.html [8 March 2008].
43. McReynolds T, Blythe D. *Advanced Graphics Programming Using OpenGL*. Morgan Kaufmann: Los Altos, CA, 2005; 274–276, 520–525.
44. Wong A, Kennings A. Adaptive multiple texture approach to texture packing for 3D video games. *Proceedings of the 2007 Conference on Future Play*, Toronto, Canada, 2007; 189–196.
45. Chu PC, Beasley JE. A genetic algorithm for the multidimensional Knapsack problem. *Journal of Heuristics* 1998; **4**(1):63–86.
46. OpenGL Architecture Review Board. WGL_ARB_pbuffer in OpenGL Extension Registry. The OpenGL Architecture Review Board. Available at: http://www.opengl.org/registry/specs/ARB/wgl_pbuffer.txt [8 March 2008].
47. OpenGL Architecture Review Board. GL_EXT_framebuffer_object in OpenGL Extension Registry. The OpenGL Architecture Review Board. Available at: http://www.opengl.org/registry/specs/EXT/framebuffer_object.txt [8 March 2008].
48. Campbell C. STR-Crazy: Improving the OpenGL-based Java 2D Pipeline. Sun Microsystems Inc. Available at: http://weblogs.java.net/blog/campbell/archive/2005/03/strcrazy_improv_1.html [8 March 2008].
49. Burrows AL, England D. Java 3D, 3D graphical environments and behaviour. *Software Practice and Experience* 2002; **32**(4):359–376.
50. Meyer J, Bederson B, Fekete J-D. Agile2D OpenGL renderer. Human–computer Interaction Lab, University of Maryland, U.S.A. Available at: http://www.cs.umd.edu/hcil/agile2d/ [8 March 2008].
51. Schaback J. FengGUI, Java GUIs with OpenGL. Available at: http://www.fenggui.org/ [8 March 2008].

52. Denault A, Kienzle J. Avoid common pitfalls when programming 2D graphics in Java: Lessons learnt from implementing the Minueto toolkit. *ACM Crossroads* 2007; **13**(3).
53. jMonkeyEngine.com. jMonkey Engine. jMonkeyEngine.com. Available at: http://www.jmonkeyengine.com/ [8 March 2008].
54. jPCT. Available at: http://www.jpct.net/ [8 March 2008].
55. Xith3D Community. The Xith3D Project. Xith3D Community. Available at: http://www.xith.org/ [8 March 2008].
56. Sun Microsystems Inc. Bug ID: 6343853 rendering issues on Vista caused by use of GDI and DDraw on onscreen surfaces. Sun Microsystems Inc. Available at: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6343853 [8 March 2008].