SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2007; **37**:727–745
Published online 2 November 2006 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.786

A portable AWT/Swing architecture for Java game development

Yi-Hsien Wang, I-Chen Wu*,† and Jyh-Yaw Jiang

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

SUMMARY

Recently, the performance of Java platforms has been greatly improved to satisfy the requirements for game development. However, the rendering performance of Java 1.1, which is still used by about one-third of current Web browser users, is not sufficient for high-profile games. Therefore, practically, Java game developers, especially those who use applets, have to take this into consideration in most environments. In order to solve the above problems, this paper proposes a portable window toolkit architecture called the CYC Window Toolkit (CWT) with the ability to: (1) reach high rendering performance particularly in Java 1.1 applications and applets when using DirectX to render widgets in CWT; (2) support AWT/Swing compatible widgets, so hence the CWT can be easily applied to existing Java games; (3) define a general architecture that supports multiple graphics libraries such as AWT, DirectX and OpenGL, multiple virtual machines such as Java VM and .NET CLR, and multiple operating systems (OSs) such as Microsoft Windows, Mac OS and UNIX-based OSs; (4) provide programmers with one-to-one mapping APIs to directly manipulate DirectX objects and other game-related properties. The CWT has also been applied to an online Java game system to demonstrate the proposed architecture. Copyright © 2006 John Wiley & Sons, Ltd.

Received 2 November 2005; Revised 17 April 2006; Accepted 31 July 2006

KEY WORDS: Java AWT; .NET; CWT; CYC Window Toolkit; DirectX; OpenGL

INTRODUCTION

Since modern computer game development is becoming progressively more complex, many commercial games are made by large teams, typically ranging from 20 to 50 members [1].

*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 of the Republic of China, ThinkNewIdea, Inc.; contract/grant number: NSC 94-2622-E-009-011-CC3

The productivity of programming teams plays a very important role of reducing the time to market, and this has influenced the mainstream languages of the game industry.

Before 1990, most games had been developed mainly in assembly languages to boost performance [2]. In the 1990s, the C language with relatively high productivity became popular in the game industry, since hardware and compiler technologies continuously improved the performance of the C language. For example, in 1993, ID Software Inc. [3] developed the game *Doom* in which most code was written in C. Furthermore, when compared with C, the C++ language increases the productivity with little loss of performance owing to the features of object orientation, such as maintainability and reusability.

Since Sun Microsystems (abbreviated as Sun) released the Java language [4] in 1995, Java has become increasingly popular owing to its higher productivity and portability than C/C++. A report [5] from IDC in 1998 showed that writing the code in pure Java instead of C++ increased the overall productivity by a factor of 30% and the coding phase alone by 65%. Since that study was made using *Java Development Kit* (*JDK*) 1.0.2, these figures could be greater today owing to the improved capabilities of the modern Java 2 Platforms Standard Edition (J2SE). Phipps [6] also presented a similar result, and concluded that Java was 30–200% more productive than C++. With the growth of the Internet, the Java applet together with its high portability—write once, run anywhere—has become popular on the Internet. These advantages have caused Java to become a candidate for one of the mainstream languages for game development. For example, Yahoo! Games [7], ArcadePod.com [8], CYC games [9–11] and RuneScape [12] use Java.

However, most of the above Java games are *low profile games*[‡], because Java programs normally have lower performance, especially in *graphic user interface* (*GUI*) components, also called *widgets*, which has made game developers hesitant to use it for *high-profile games*. In general, the performance of programs written in Java 1.0 is about 20 to 40 times slower than in C/C++ [13]. After several significant revisions in the *Java Virtual Machine* (*JVM*), the tweaked Java programs using J2SE 1.4 ran on average only about 20–50% slower than the tweaked C/C++ programs [13]. Unfortunately, the GUI part of early Java programs still performed slowly. Since most game programs, especially high-profile games, have intensive GUI operations, such as animation or complex scenes, it is critical to reach high rendering performance.

For high rendering performance, DirectDraw and Direct3D of Microsoft DirectX (or DirectGraphics in DirectX 7.0 and beyond) [14] or OpenGL (Open Graphics Library) [15] are commonly used to access specialized hardware features, such as direct access to the *video memory* in graphics cards, constructing 3D scenes and writing shader code. Thus, an important topic for high-profile game programs in Java is to incorporate DirectX or OpenGL into Java GUI design.

The earliest GUI mechanism in Java is *Abstract Window Toolkit* (*AWT*) [16]. In order to keep the cross-platform capability of Java, the AWT was designed to provide a common set of widgets which use the peer architecture [17] to maintain the look-and-feel in native operating systems (OSs), such as Win32 windows components [18] in Microsoft Windows and Motif [19] in UNIX-based OSs. Such widgets are called *heavyweight components* [20]. However, the GUI designed on one platform may look different on other platforms, and AWT widgets cannot satisfy the requirements of

[‡]Low profile and high profile games are defined in [13].

DOI: 10.1002/spe

applications that require more complex widgets, such as trees and tables, and advanced graphics functionalities, such as pattern filling and color management.

In order to solve the above problems of AWT, Sun announced *Java Foundation Classes* (*JFC*) [21] in 1998, which includes Swing and Java 2D. Swing provides more complex widgets based on *lightweight* support of Java AWT 1.1 that does not rely on the native widgets of the underlying OSs. Java 2D enables advanced 2D graphics functionalities, imaging, text and printing.

Although the functionalities of the JFC are generally good for applications requiring more complex widgets or advanced graphics, its implementation suffers from poor rendering performance when compared with heavyweight components. In order to improve rendering performance, Sun started to access specialized hardware features via DirectX in J2SE 1.4 [22] and OpenGL in J2SE 5.0 (or 1.5) [23]. However, the OpenGL-based Java 2D pipeline (abbreviated OpenGL pipeline) of J2SE 5.0 brings more bugs and is not usable in real applications. In order to resolve this problem, Java SE 6.0 introduces a newly designed OpenGL pipeline that provides much better stability and performance than that in J2SE 5.0 [24].

Although rendering performance in J2SE 1.4 and beyond has been greatly improved, many hardware acceleration features, disabled by default, need to be configured carefully. Unless users explicitly configure proper properties in the Java control panel, the applet games will not fully benefit from hardware acceleration [25]. Thus, it is inconvenient for applet game users, especially for those without permission granted, to modify the system properties. Although signed Java applets can modify the system properties, the modifications also influence other Java programs running on the JRE.

Considering past work also related to the rendering performance issue, IBM's Standard Widget Toolkit (SWT) [26] is an alternative to AWT/Swing which produces a more native-like appearance and a higher performance than the early Java versions, since all its widgets are designed as heavyweight ones. However, SWT does not completely take advantage of specialized hardware features for game applications, so it may perform inefficiently for high-profile games. In contrast to the native widget approach, other solutions incorporate DirectX or OpenGL into Java GUI design. Bellotti *et al.* [27] designed a set of APIs named *DirectJ*, which can directly access DirectX via the *Java Native Interface* (*JNI*), and thus achieved significant graphics performance improvement in J2SE. The *OpenGL for Java* (GL4Java) [28] wraps OpenGL APIs via JNI. However, the GL4Java has been outdated since the last release in 2001. *Java binding for OpenGL API* (*JOGL*) [29] is also a one-to-one mapping to the OpenGL APIs. The *Lightweight Java Game Library* (*LWJGL*) [30] is another library specifically tuned for writing games. Both JOGL and LWJGL are still in active development now. Java 3D [31] is a high-level API, which has OpenGL and DirectX implementations, for creating, rendering and manipulating 3D scene graphs. Using these libraries can greatly improve the rendering performance of Java. However, these solutions still suffer from the two problems discussed as follows.

First, although both DirectX and OpenGL have good rendering performance, they do not provide their own widget systems. Consequently, when mixing other widget systems, such as Win32 window components, with DirectX or OpenGL, the performance is still limited to that of the widget systems (or is even worse). In addition, the native widgets typically control their repainting timing and process, which may cause some undesired effects, such as flicking.

Second, many of the above solutions are supported on the Java 2 platforms only. Unfortunately, *Microsoft Java Virtual Machine* (*MSVM*) [32], which is in Java 1.1.4, is currently used by about 31.0% of browser users (calculated from the statistical data in [33] during the latest two months),

DOI: 10.1002/spe

while the percentages of JRE 1.4.x and JRE 1.5.0 are 24.5% and 41.8%, respectively. Thus, for game programmers, it is still important that applet games can be played in most environments.

Other reasons for using Java 1.1 are that:

- many applet games based on Java 1.1 were developed in the past when Microsoft Internet Explorer only supported Java 1.1.4 (for example, CYC games [9–11] still keep using the past MSVM for compatibility of their code);
- the J# language in .NET supports Java 1.1 only;
- most PDAs or mobile phones using Personal Java support Java 1.1 only.

In this paper, our approach is to enhance the graphics performance of Java 1.1 by using DirectX or OpenGL to render all widgets, figures, images and texts. We propose a window toolkit called the *CYC Window Toolkit* (*CWT*), a fast-rendering lightweight GUI toolkit that renders all its widgets via native graphics libraries. The CWT can automatically switch the hardware acceleration features in runtime, so users do not need to manually set the properties. In addition, the CWT also has the following features:

- it supports AWT/Swing compatible widgets, and, hence, the CWT can be easily applied to existing Java games;
- it defines a general architecture that supports multiple graphics libraries (Java AWT, DirectX and OpenGL), multiple virtual machines (JVM and .NET CLR) and multiple platforms (Windows, Mac OS, UNIX-based OSs, etc.);
- it provides programmers with extra APIs to directly manipulate DirectX objects and other game-related properties, and, therefore, advanced programmers can control abilities of the DirectX without waiting for functionalities supported by Sun.

The rest of this paper is organized as follows. First, the architecture of the CWT is defined. Then, our implementation strategy and current status of the CWT are given, followed by experimental results and performance analysis, including a real application of the CWT. Finally, conclusions of our work are presented.

THE ARCHITECTURE OF THE CWT

Figure 1 shows the architecture of the CWT, which is designed to use various graphics libraries, such as Java AWT, DirectX and OpenGL, to build AWT/Swing compatible widgets. For Java AWT, the CWT accesses it via a simple wrapper, identified as CWT-AWT. This is used when neither DirectX nor OpenGL is supported by the underlying OSs.

For DirectX, the CWT accesses DirectX 3.0, which is supported in MSVM via a wrapper identified as CWT-DX3, and DirectX 9.0, which is supported in .NET J# [34] via a wrapper identified as CWT-DX9. Both together are also called CWT-DX.

For OpenGL, the 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, Linux and Solaris. However, these libraries have problems that the development of the GL4Java is stopped, and Both JOGL and LWJGL only work in J2SE 1.4 and beyond. However, the CWT is also required to support

DOI: 10.1002/spe

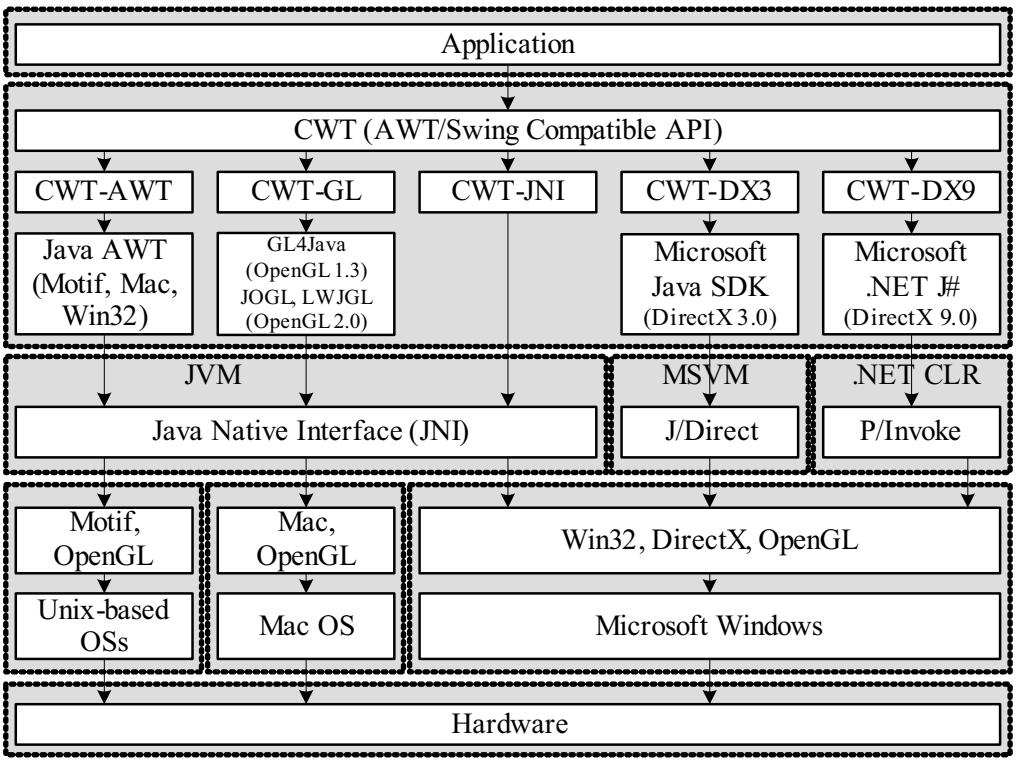

Figure 1. The architecture of the CWT.

old Java VMs as mentioned above. In order to resolve these problems, we can choose to use JNI to access OpenGL via a wrapper, identified as CWT-JNI.

Two methods for graphics library selection are provided. First, the CWT can detect the available graphics libraries and choose one in runtime automatically. Second, programmers can also designate an implementation to use. If designated libraries are unavailable, the CWT uses CWT-AWT instead.

In the following subsections, we will introduce the following four design topics: the component design; the event model; the painting model; and design issues for game applications.

Component design

For components, the hierarchy of the CWT is similar to that of Java AWT. The Composite pattern [35] is the key design of Java AWT that allows programmers to build the complex GUI hierarchy by recursively composing objects in a tree-like manner. Two abstract classes—`Component and Container`—are the key classes of the entire hierarchy. Component is the root class of all the widgets.

DOI: 10.1002/spe

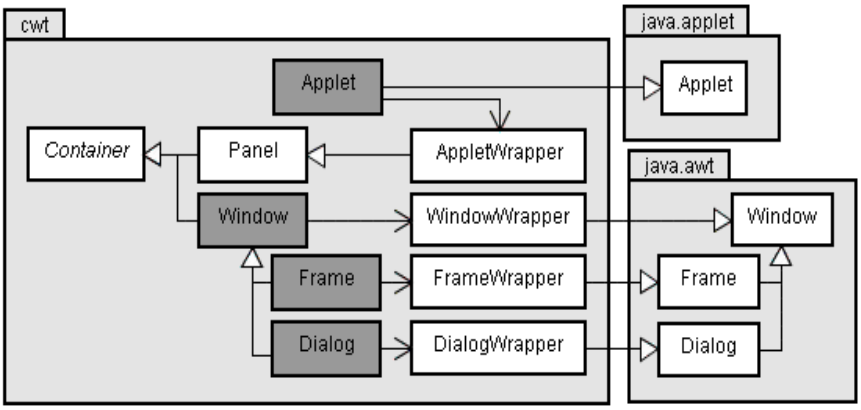

Figure 2. Four top containers of the CWT.

Container is a special component which can contain other components, including Container itself, and can arrange and resize the components inside.

All heavyweight components of Java AWT except for the four containers, `Window`, `Frame`, `Dialog` and `Applet`, as shown in Figure 2, are redesigned as lightweight components in the CWT. The lightweight components need the canvas of a heavyweight one at the top level to draw. The first three containers wrap the corresponding AWT components and use the wrapped components for painting. When clients initiate these containers, the wrapped AWT components are created and shown on the screen. However, the CWT `Applet` is designed in a different way from the above three. The CWT `Applet` directly inherits `Applet` of Java AWT, since the browsers can only accept `Applet` of Java AWT.

The wrapped heavyweight AWT containers act not only as canvases but also as bridges to pass native events, such as window, mouse and keyboard events, from the AWT heavyweight containers to the CWT components, as explained in the following subsection.

Event model

The CWT follows a *delegation-based event model* [36] used in Java 1.1 and beyond. However, the event processing flow is slightly different from AWT. The four heavyweight containers mentioned in the previous subsection can get native events from OS and process them immediately. The lightweight components rely on the four heavyweight ones to get native events. In order to dispatch events to the proper components, the CWT has an `EventManager` class that analyzes the original events, regenerates new events, puts the new events into the internal event queue and then dispatches the events to the proper components. Figure 3 illustrates an event processing example in the following steps.

1. A mouse moving event occurs in a wrapped `Frame`. Then `EventManager` implementing the mouse motion listener receives this event.

DOI: 10.1002/spe

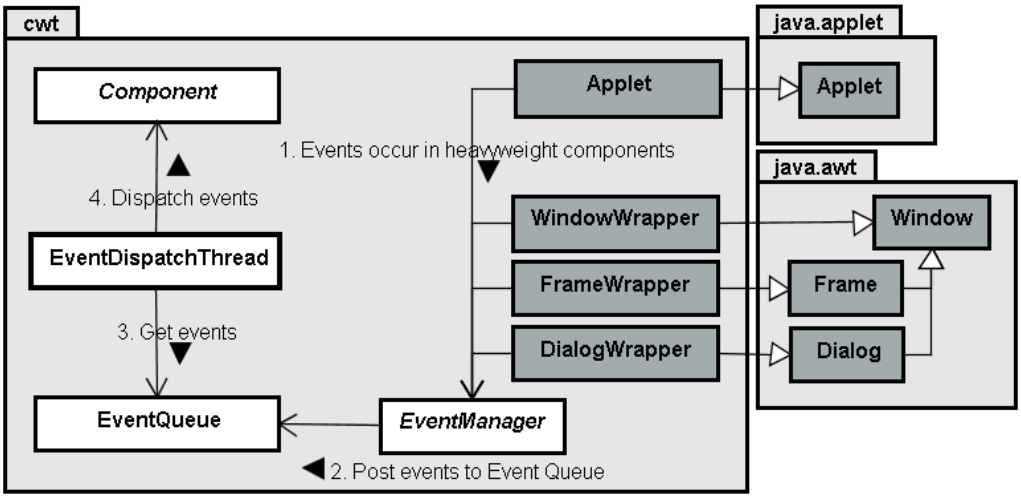

Figure 3. Event flow in the CWT.

2. The `EventManager` gets the mouse position from the event object and finds out which lightweight components in `Frame` the mouse is on.
3. The `EventManager` generates a new CWT mouse event with translated position and puts it into `EventQueue`.
4. The thread `EventDispatchThread` retrieves the event and dispatches it to the component where the mouse is.

Painting model

The painting model is the key for rendering, and it is especially important for games. The CWT follows the design of both Java AWT and Swing, which uses a callback mechanism for painting [20]. Two callback methods to be overridden in `Component` are `paint` and `update`. Programs place the rendering code in the two methods and use the `Graphics` parameter object for drawing on the component. The `Graphics` has several states to be configured: color, font, translation and clip rectangle.

In order to improve the rendering speed, it is very important to re-implement `Graphics` to use DirectX or OpenGL APIs. For example, in CWT-DX3, the `drawImage` methods of `Graphics` are implemented by the `blt` method of DirectX. Both the `fillRect` and `clearRect` methods of `Graphics` are implemented by the `bltColorFill` method of DirectX. Since the DirectX does not provide text and figure rendering APIs, the Windows *graphics device interface* (*GDI*) is used instead.

Like Swing, the widgets of the CWT except for the four containers are all lightweight. To support efficient lightweight painting and game-related features, the CWT follows the design of Swing.

DOI: 10.1002/spe

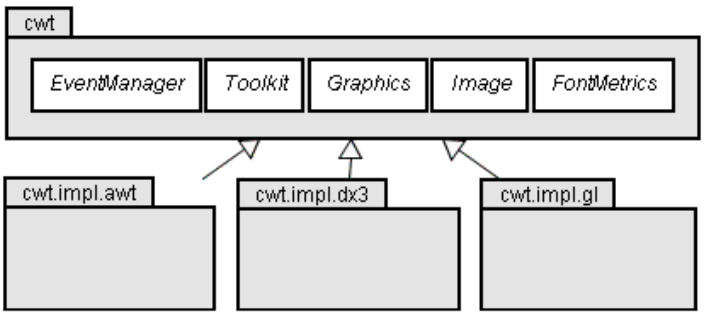

Figure 4. Five abstract classes that enable multiple implementations.

Many Swing features are applied to the CWT components. The most important features are transparency, double buffering and the optimized repaint process [20].

The painting part of the CWT has five abstract classes, `EventManager`, `Toolkit`, `Graphics`, `Image` and `FontMetrics`, as shown in Figure 4, in order to support implementations for different graphics libraries. Any implementation (for painting) has to inherit and implement the five classes. Note that, in contrast, these different implementations share the same code of the component and event parts in the CWT. The EventManager, which is an additional class to the Java AWT, transfers events from the Java AWT containers to the CWT, as illustrated in Figure 3. Other classes perform the same tasks as those in the Java AWT.

Design issues for game applications

For high rendering performance, game programmers usually put pre-rendered images and textures into the video memory of the graphics cards, so that the graphics cards can directly access them. However, since the size of the video memory is limited, programs may not be able to put all the images into the video memory. Therefore, some of the offscreen images will stay at the system memory. Practically, programmers may need to put frequently used offscreen images into the video memory and move those used infrequently to the system memory. Carefully managing the video memory resource can improve the overall rendering performance.

There exist different approaches to the storing and accelerating of offscreen images among different Java versions [37]. In Java 1.0 and 1.1, Windows GDI functionalities are used to accelerate offscreen images. In J2SE 1.2 and beyond, the Windows GDI functionalities are discarded for outputting equal rendering quality on different platforms. However, without hardware acceleration, this results in lower rendering performance than Java 1.0 and 1.1 in some aspects such as text and figure rendering. Since J2SE 1.4, new images, also called *volatile images* (whose class name is `VolatileImage`), are designed to store image data into video memory, if possible. For example, on Microsoft Windows platforms, the volatile images are implemented by DirectX or OpenGL. Furthermore, a new type of image called a *managed image* is also introduced from J2SE 1.4. The managed images are stored in the

DOI: 10.1002/spe

system memory and can be automatically cached in the video memory for performance optimization according to access types and frequency [37].

For most game applications, programmers want to control all the details for high rendering performance. In order to give programmers such flexibilities, the CWT allows programmers to decide the memory location when creating images, and allows them to copy images between the video memory and system memory. The CWT-DX3 also supports direct access to DirectX 3.0 API of MSVM. For example, programmers can get DirectDraw surfaces by calling the `getDDSurface` method of `Image` of CWT-DX3. Using `Toolkit`, programmers can get DirectDraw objects for advanced operations.

However, using DirectDraw, programmers may encounter the surface lost problem, which occurs when users change the screen resolution or simply switch to another window. If surfaces are lost, the image data are gone and must be restored or re-rendered [37]. The CWT tries to rebuild the lost surfaces if the images are loaded directly from image files. Otherwise, if images are created and maintained by programmers, CWT sets a flag 'contentLost' in the image objects, as the design of volatile images since J2SE 1.4 [37].

DirectX and OpenGL both support high performance for 2D image rendering, but text and figure rendering may still rely on Windows GDI. Game programmers usually use pre-rendered text and figures, saved as images, to solve this problem. Although DirectX does not support figure drawing, it can draw rectangles (including horizontal and vertical lines) by filling colors into them, as J2SE 1.4 does [37]. With hardware acceleration, these operations perform better than rendering non-rectangular shapes.

When using Window GDI to draw texts or figures in a DirectDraw surface, programs need to get a GDI-compatible device context (DC) handle for the surface. This will lock the surface and thus incur extra overhead [38]. Thus, it is important to minimize the locking times to reduce the performance overhead. In order to minimize the times, the CWT does not release the obtained DC handle, until any of the image-blitting or rectangle-filling methods is invoked.

THE IMPLEMENTATION OF THE CWT

This section introduces our implementation approach and the current state of the implementation for the CWT. There are two approaches to implement AWT compatible window toolkits: peer extension and re-implementation.

First, for the peer extension approach, Java uses it to support pluggable look-and-feel on different OSs by implementing the interfaces in package `java.awt.peer`. Although these interfaces were originally designed for cross-platform operation, we can replace the peer implementation by a different one, such as Charva [39]. With this approach, we need to modify no Java programs but just set the new toolkit in system properties.

Second, the re-implementation approach is to rewrite the entire set of the `java.awt` package. With this approach, programmers have to change all the `import` statements for '`java.awt`' and '`java.applet`' to '`cwt`', and then recompile their programs.

The re-implementation approach has the following three advantages. First, better algorithms or mechanisms, such as the `RepaintManager` in Swing, can be applied to the CWT to improve the performance outside the peer part. Second, the CWT interface for game development may need to

DOI: 10.1002/spe

Table I. Added methods of the CWT (compared with AWT 1.1).

Classes	Added methods
Applet	`cwt.Image getCwtImage()` `cwt.AppletWrapper getAppletWrapper()`
Toolkit	`cwt.Image createImage(... , boolean bVram)` `com.ms.directX.DirectDraw getDirectDraw()`

Component	`cwt.Image createImage(... , boolean bVram)`
	`void setDoubleBufferedEnabled(boolean)`
	`void setOpaque(boolean)`

Graphics	`void flip()`

Image	`boolean contentsLost()`
	`com.ms.directX.DirectDrawSurface getDDSurface()`
	`void transferToVideoMemory()`
	`void transferToSystemMemory()`

be extended in the future, but the peer extension approach can hardly achieve this goal. Third, the CWT interface remains unchanged even when a new Java version adds more methods. With the peer extension approach, the added methods in new Java versions have to be implemented in the CWT, or the CWT could not be run in the new versions. For these reasons, our implementation is based on the re-implementation approach.

The CWT architecturally supports different wrapper implementations, as shown in Figure 1. Currently, we have implemented two, CWT-AWT and CWT-DX3. CWT-AWT, which is basically trivial to implement, helps us to keep the portability in different JVMs in different OSs, but the rendering performance is slightly worse than the original owing to extra wrapping overhead.

CWT-DX3 is implemented by using DirectX 3.0 provided by Microsoft Java SDK [32]. Using Microsoft Java SDK is a quick solution that allows us to access DirectX without building a bridge via JNI. Although it is the drawback that only old DirectX functionalities are available, since the CWT only supports 2D rendering now, DirectX 3.0 already performs with sufficiently high performance, as shown in the 'Experiments' section later.

In the future, we will implement CWT-DX9 and CWT-GL. For CWT-DX9, we will simply port the CWT to J#, which can access the latest DirectX 9.0. For CWT-GL, we will implement our own OpenGL bindings via JNI (CWT-JNI). The OpenGL version can allow the CWT to be used in various OSs with hardware acceleration, and is no longer limited to Microsoft Windows.

API Comparison to AWT 1.1

Table I summarizes the APIs that are incompatible with Java AWT 1.1 or are extensions from Java AWT 1.1. These APIs are as follows.

- **Incompatible Applet methods.** In the `cwt.Applet` class, which inherits `java.awt.Applet`, a method `getAppletWrapper` is added for obtaining the wrapped CWT component. Programmers should use the wrapped component for some operations,

DOI: 10.1002/spe

such as the `instanceof cwt.Component` test, since the `cwt.Applet` class is not a subclass of the `cwt.Component` (described in the 'Component design' subsection). Moreover, the method signature of the `getImage` method is changed to '`cwt.Image getCwtImage()`'. Programmers should use this method instead of the original '`java.awt.Image getImage()`' method, since the `cwt.Image` class is not a subclass of the `java.awt.Image` class.

- **DirectX object related methods.** In the `Toolkit` class, programmers can access the DirectDraw object by calling the `getDirectDraw` method to get full control of rendering. The DirectDraw object can be used to create offscreen images, query the capabilities of the graphics card and perform other DirectDraw specific operations, supported by Microsoft Java SDK [32]. Furthermore, programmers can get DirectDraw surfaces of the CWT images by calling the `getDDSurface` method of the `Image` class to handle the pixels of the images.
- **Double buffering related methods.** The `Component` class adds the `setDouble BufferedEnabled` method to support double buffering by turning on or off the rendering strategy. The `Graphics` class can control the surface flipping by calling the `flip` method that switches between the front and back buffers.
- **Transparent component related methods.** The `Component` class adds the transparency support via the `setOpaque` methods. The CWT supports the rendering ordering and elimination process for the transparency of the components.
- **Offscreen image related methods.** In order to achieve the best performance, the `createImage` methods of the `Component` and `Toolkit` classes include a parameter to indicate the memory location either in video memory or in system memory. The image can be transferred between video memory and system memory by calling the `transferToVideoMemory` and `transferToSystemMemory` methods of the `Image` class. To deal with the surface lost issues of DirectDraw, the `Image` class adds a method, called `contentsLost`.
- **OpenGL related methods.** The APIs for OpenGL are omitted since we have not implemented CWT-GL.

EXPERIMENTS

In this section, we describe the design of a set of test programs as our benchmark to analyze the rendering performance in different rendering environments, defined in the 'Rendering environments' subsection below. All the test programs were run on a personal computer with the following specification: AMD K7 Barton 2500+CPU, 1 GB DDR400 RAM and an nVidia Geforce 4600 graphics card with 128 MB VRAM. The driver version of the graphics card was ForceWare 66.93. The machine ran on Microsoft Windows XP Professional Edition with service pack 2. The display resolution was 1280×1024 true color mode and font anti-aliasing was disabled.

Test programs

The test programs open windows with a size of 600×300 and render moving images, text and figures 20 000 times in each different rendering environment as follows.

DOI: 10.1002/spe

Figure 5. Transparent and opaque images for testing.

1. Images. These use a (110×110)-pixel^2 transparent image and a (110×110)-pixel^2 opaque image, as shown in Figure 5. The transparent image is mirrored in runtime for the mirrored image tests.
2. Text. This uses the simple string 'Running', with a font size of 12 pixels. Then, four randomly generated Chinese characters are used each time to simulate a chat system and to test the efficiency of the text cache mechanism of J2SE 1.4 and beyond.
3. Figures. These fill rectangles with (110×110)-pixel^2 area and circles with 110-pixel diameter.

In brief, for each rendering environment, we use seven test programs respectively rendering the following items: (1) transparent images; (2) opaque images; (3) mirrored transparent images; (4) simple strings; (5) random strings; (6) rectangles; and (7) circles.

Rendering environments

In our experiments, we considered several different rendering environments for our test programs, as shown in Table II. In order to determine the gap of the rendering performance between Java and C, we implemented a set of test programs in C directly accessing DirectDraw in DirectX 7.0. The C programs were built with the optimization of maximal speed (by setting the compiling option /O2) in Microsoft Visual Studio 6.0 with service pack 6 installed. Such a rendering environment was identified as C-DX.

For the test programs in Java, we considered the following four kinds of Java rendering environments.

1. The original Java AWT using the normal image objects. In this case, we used different JVMs, Sun JVM 1.1 to 6.0 and MSVM, whose rendering environments were identified as AWT-J1 to AWT-J6 and AWT-JM, respectively.
2. The original Java AWT using managed images supported by J2SE 1.4 to 6.0. The tests ran with default settings, i.e. no system properties were set, which were identified as AWT-J4M to AWT-J6M. On the other hand, when translucency acceleration was specifically enabled (by setting both `sun.java2d.translaccel` and `sun.java2d.ddforcevram` system properties to `true`) [40], we identified them as AWT-J4MS to AWT-J6MS.
3. The CWT with the AWT implementation as described in the 'Architecture of the CWT' section. In this case, we used different JVMs, such as in (1), whose rendering environments were identified as CWT-J1 to CWT-J6 and CWT-JM, respectively. Similarly to (2), the versions using managed images were identified as CWT-J4M to CWT-J6M and CWT-J4MS to CWT-J6MS.

DOI: 10.1002/spe

Table II. Rendering times (in milliseconds) for each rendering environment.

Language	Library	Virtual machine		ID	Transparent images	Opaque images	Mirrored images	Simple strings	Random strings	Rectangles	Circles
Java	AWT	MSVM 5.0.0.3810		AWT-JM	5156	1343	5171	187	312	125	656
		JVM 1.1.8_10		AWT-J1	5140	1313	5125	328	1437	125	625
		JVM 1.2.2_17		AWT-J2	1062	1078	4469	359	359	515	1828
		JVM 1.3.1_17		AWT-J3	1109	1156	3890	375	11 594	531	2125
		JVM 1.4.2_11	Image	AWT-J4	1375	1359	2656	765	7641	922	1797
			Managed	AWT-J4M	265	266	2515	1078	7828	125	1031
			image	AWT-J4MS	266	266	328	1093	8141	125	1000
		JVM 1.5.0_6	Image	AWT-J5	1390	1375	2656	671	703	937	1812
			Managed	AWT-J5M	266	265	2516	968	1015	125	1031
			image	AWT-J5MS	266	265	265	969	1015	125	1046
		JVM 1.6.0_b73	Image	AWT-J6	1390	1453	2422	656	687	906	1156
			Managed	AWT-J6M	250	250	2281	953	1000	125	1000
			image	AWT-J6MS	265	328	266	110	1625	235	1547
	CWT-AWT	MSVM 5.0.0.3810		CWT-JM	5421	1390	5407	203	406	141	703
		JVM 1.1.8_10		CWT-J1	5500	1406	5500	390	1562	141	641
		JVM 1.2.2_17		CWT-J2	1312	1328	4719	593	640	750	2093
		JVM 1.3.1_17		CWT-J3	1375	1422	4140	609	11 782	765	2406
		JVM 1.4.2_11	Image	CWT-J4	1657	1656	2953	1046	8609	1218	2109
			Managed	CWT-J4M	281	281	2734	1109	8578	140	1062
			image	CWT-J4MS	281	281	359	1110	8157	140	1062
		JVM 1.5.0_6	Image	CWT-J5	1656	1656	2938	937	984	1219	2109
			Managed	CWT-J5M	281	281	2735	1000	1062	140	1046
			image	CWT-J5MS	281	281	281	1000	1063	140	1047
		JVM1.6.0_b73	Image	CWT-J6	1656	1719	2687	891	937	1171	1422
			Managed	CWT-J6M	281	266	2500	969	1016	140	1031
			image	CWT-J6MS	312	359	312	140	1671	266	1562
	CWT-DX	MSVM 5.0.0.3810		CWT-DX3	**266**	**265**	**265**	**203**	**484**	**156**	**1109**
C	DirectX7			C-DX	234	234	234	187	359	109	937

4. The CWT with the DirectX implementation as described in the 'Architecture of the CWT' section. This item was identified as CWT-DX3.

For simplicity, we use the wild card '*' to indicate a group of rendering environments. For example, AWT-* (or AWT-J*) includes the first and second kinds of rendering environments; AWT-J*–AWT-J*M* includes the first kind of rendering environment only (excluding AWT-J*M and AWT-J*MS); AWT-J*M* includes the second kind of rendering environment.

Analysis

The experimental results are shown in Table II. We analyze these results by considering the following three comparisons: (1) CWT-DX3 versus C-DX; (2) CWT-DX3 versus AWT-*; and (3) CWT-J* versus AWT-*.

CWT-DX3 versus C-DX

In general, the performance of CWT-DX3 is, on average, about 19.8% (or 64.9 ms) slower than those in C-DX. The overhead is generally acceptable. Thus, Java game programmers can rely on the CWT

DOI: 10.1002/spe

to reach high performance (close to that in C/C++) without writing native code to get the benefit of native graphics libraries.

*CWT-DX3 versus AWT-**

In this comparison, we discuss the rendering of images, text and figures separately.

For the transparent and opaque images rendering tests, CWT-DX3 generally performs much better than most of AWT-*. The performance of CWT-DX3 is much faster than that of AWT-J*–AWT-J*M* by a factor ranging from 4.0 to 19.4 in the transparent and opaque image tests. This is because CWT-DX3 runs with full DirectX acceleration, while AWT-J*–AWT-J*M* do not. On the other hand, AWT-J*M* runs −23.8% to 6.4% faster than CWT-DX3 when the managed images are used. The reason why AWT-J*M* and CWT-DX3 run roughly equally is that they all use DirectX.

The results of the mirrored transparent image tests show that AWT-J*–AWT-J*M* runs much more slowly than CWT-DX3 by a factor ranging from 9.1 to 19.5, since they do not use DirectX for runtime created images, such as mirrored and rotated images. AWT-J*MS runs nearly as fast as CWT-DX3 does owing to the new API enabling translucency acceleration for volatile images [23]. Nevertheless, without proper system properties, AWT-J*M still performs much more slowly than the CWT-DX3 by a factor ranging from 8.6 to 9.5.

In Table II, several inconsistent results are found: AWT-J4, AWT-J5 and AWT-J6 are 26.1% to 34.8% slower than AWT-J2 in transparent and opaque image tests. In this paper, this phenomenon is said to be *performance inconsistent* when a new version performs much more slowly than an old version. Such a phenomenon may make programmers confused and therefore it is hard to tune up the performance.

In the text rendering tests, we find more serious performance-inconsistent results. Old Java versions, such as AWT-JM, AWT-J1, AWT-J2 and AWT-J3, may produce better results than those of newer versions AWT-J4* to AWT-J6*, excluding AWT-J6MS which produces the best result. For the simple string tests, the reason for this is that in order to have a similar quality on all platforms, the J2SE renders the text by itself [37], not by the underlying OSs, which introduces large overheads. AWT-J*M* excluding AWT-J6MS is very slow, because the CPU has to access volatile images stored in the video memory, which limits the performance of text drawing. CWT-DX3, AWT-JM, AWT-J1 and AWT-J6MS are the fastest versions. The first three simply use Windows GDI. For AWT-J6MS, Sun has optimized its text cache mechanism and stored the cached glyphs in video memory [41], so it generates a good result. AWT-J6M still does not benefit from this mechanism, since the new text cache needs the translucent acceleration support.

The random string tests can reflect some real applications, such as chat systems. AWT-J3 and AWT-J4* introduce huge overhead upon caching the text images when first rendering. The caching mechanism makes them perform much more slowly than CWT-DX3 by a factor ranging from 15.8 to 24.0. Since the cache mechanism of J2SE 5.0 and Java SE 6.0 has been improved [23,41], the rendering time of AWT-J5* and AWT-J6* is reduced greatly to only 1.4 to 3.4 times slower than that of CWT-DX3.

The rectangle filling tests show that AWT-JM, AWT-J1, AWT-J*M* and CWT-DX3 all use the special optimization mentioned in the ‘Design issues for game applications’ subsection to perform this task. Owing to the non-hardware-accelerated rendering pipeline, AWT-J2 to AWT-J6 fills rectangles more slowly than CWT-DX3 by a factor ranging from 3.3 to 6.0.

DOI: 10.1002/spe

In the circle filling tests, the performance of CWT-DX3 is slower than those of AWT-JM and AWT-J1 by a factor of about 1.8, is roughly equal to those of AWT-J6 and AWT-J*M* and is faster than those of AWT-J2 to AWT-J5 by a factor ranging from 1.6 to 1.9. The performance-inconsistency still exists in J2SE. These tests show that DirectX does not accelerate figure rendering, so Windows GDI is used to perform this task. However, mixing DirectX with Windows GDI does not perform as well as Windows GDI only, which is used in Java 1.1.

The analysis of the comparison can be summarized as follows.

- When compared with AWT-JM and AWT-J1, CWT-DX3 is much faster for rendering images, but slower for rendering rectangles and circles. In text rendering tests, CWT-DX3 is faster than AWT-J1, but slightly slower than AWT-JM.
- When compared with AWT-J2 to AWT-J6, CWT-DX3 renders almost all the test items much faster.
- When compared with AWT-J*M*, the performance of CWT-DX3 is roughly equal for rendering transparent images opaque images and circles, much faster for strings and slightly slower for rendering rectangles. If proper system properties are configured, AWT-J*MS and CWT-DX3 run roughly equally when rendering mirrored transparent images; otherwise, AWT-J*M still run much more slowly than CWT-DX3.

In addition, we also observe a serious problem, performance inconsistency, among AWT-*, especially for strings and figures.

CWT-J versus AWT-**

Since the CWT wraps the four native containers to get events, it does introduce extra overheads. Comparing CWT-J* with the corresponding AWT-J*, we find that the average overhead is 10.3% (or 154.9 ms).

Real application

We have applied the CWT to one Internet applet game, Mahjong, developed in Java 1.1 by [9–11], and tested the rendering performance, as shown in Figure 6. The window size of the game is 760 × 580. The tested game panel consists of about 100 transparent images (half of the images are rotated and scaled in runtime), about 80 characters, 25 rounded corner rectangles and 10 rectangles. Each of the tests is performed by rendering the whole game panel 10 000 times.

The results for the benchmark are listed in Table III. CWT-DX3 improves the rendering performance by a factor of 7.58 when compared with AWT-JM, and by a factor ranging from 3.15 to 5.91 when compared with the newer JVMs, AWT-J2 to AWT-J6. CWT-DX3 also runs 17%, 6% and 10% faster than AWT-J4MS, AWT-J5MS and AWT-J6MS, respectively. However, if the system properties are not set properly, the CWT runs faster than AWT-J*M by a factor ranging from 2.68 to 3.13.

We analyzed the performance in terms of frame rates, since the frame rate is an important criterion for games. The minimal acceptable frame rate for smooth animation is about 30 frames per second (FPS). For some highly interactive games such as first person shooting games, the acceptable frame rate is around 60 FPS. In Table III, the frame rate of AWT-JM is 24.66 FPS, below the acceptable requirement. In slower computers, the frame rate could be worse. The frame rate of CWT-DX3 is 186.86 FPS. This indicates that CWT-DX3 can maintain acceptable frame rates, even for some slower computers for the above game application.

DOI: 10.1002/spe

Figure 6. A game screenshot using the CWT.

Discussion

In this subsection, we discuss the issues of performance, deployment and the limitation of the CWT.

From the performance analysis in the previous subsection, CWT-DX3 reaches good rendering performance and is generally better than AWT-J*. Thus, with the help of the CWT, applet game users who use MSVM can obtain a rendering performance close to that of using J2SE 1.4 and beyond. Since MSVM, J2SE 1.4 and J2SE 5.0 cover nearly 97% users [33], this means that applet games with the CWT can achieve good rendering performance in most applet environments.

A serious problem for Java is performance inconsistency. For example, although Swing components (AWT-J2) were developed for portability and more GUI functionalities, the performance of text and figure rendering is worse than that with AWT-J1. Since J2SE 1.4, Sun has tried to use volatile images, but has made the performance of text rendering even worse. Even though the same Java VM version

DOI: 10.1002/spe

Table III. Benchmark on Mahjong.

ID	Milliseconds	Factor	Frames per second
AWT-JM	405 547	7.58	24.66
AWT-J1	316 328	5.91	31.61
AWT-J2	168 391	3.15	59.39
AWT-J3	170 156	3.18	58.77
AWT-J4	223 375	4.17	44.77
AWT-J4M	167 266	3.13	59.79
AWT-J4MS	62 594	1.17	159.76
AWT-J5	197 234	3.69	50.70
AWT-J5M	145 187	2.71	68.88
AWT-J5MS	56 968	1.06	175.54
AWT-J6	193 140	3.61	51.78
AWT-J6M	143 672	2.68	69.60
AWT-J6MS	58 797	1.10	170.08
CWT-DX3	53 515	1.00	186.86

is used, the performance may be different when different APIs are used and different system properties are configured. Most importantly, the performance inconsistency makes it difficult for programmers to predict the overall rendering performance.

For Java AWT/Swing, Sun's approach is to strenuously enhance the performance of Java AWT/Swing while still making them general for all programmers. For example, in J2SE 1.4 and beyond, managed images can be copied automatically into the video memory for performance optimization according to access types and frequency [37]. However, for most game applications, programmers need to control all the details for high performance, such as complete DirectX or OpenGL functionalities, memory location of the images and even the timing of garbage collection. Therefore, it would be a better approach to create API mappings such as GL4Java, JOGL and LWJGL, and allow them to work with AWT/Swing with high performance, as the CWT does.

Deployment is another issue in applet game development. For Web applets, programmers have few choices of setting environment. The JRE of applets may be neither the expected versions nor the proper configuration. In the case of using unexpected JRE versions, if applet game programmers want their games to be playable in most environments, the best JRE version at which programmers should target would be Java 1.1 owing to compatibility. However, the rendering performance is limited in Java 1.1. In contrast, the CWT can still perform well in this version. In the case of using improper configuration, although the performance of AWT-J*MS is close to that of CWT-DX3, AWT-J*MS needs proper system properties to enable some of its hardware acceleration features. Unfortunately, the settings are disabled by default. Therefore, users need to set them manually. However, in some computer systems, for example those supported at Internet coffee shops, users are not granted sufficient system permissions to modify the settings.

Although the CWT reaches good rendering performance in MSVM, it still has some drawbacks. First, the CWT is not designed for general-purpose applications. Second, when no hardware acceleration is available, the CWT switches to CWT-AWT implementation, which incurs extra

DOI: 10.1002/spe

overhead (about 10.3%). Third, using the CWT cannot benefit from any additional features supported by new Java versions but not implemented by the CWT. Finally, in order to access the hardware acceleration via JNI, applets using CWT need to be signed and acquire permissions from users when executed in Web browsers.

CONCLUSIONS

This paper designs a portable AWT/Swing architecture, called the CYC Window Toolkit (CWT), for high rendering performance for Java games development, especially for applet games written in Java 1.1.

The features of the CWT can be summarized as follows.

(1) High performance is reached in Java 1.1 applications and applets when using DirectX to render widgets. The CWT performs well in MSVM, which is currently used by about one-third of Web users. We have demonstrated the performance of the CWT by applying it to a real applet game, Mahjong [9]. In the benchmark we measured, the performance of the game running in MSVM was improved by a factor of 7.58, and was about 10% faster than the latest Java SE 6.0 (build 73) with full hardware acceleration enabled.
(2) AWT/Swing compatible widgets are supported. Hence, the CWT can be easily applied to existing Java games.
(3) A general architecture that supports multiple graphics libraries such as AWT, DirectX and OpenGL, multiple virtual machines such as Java VM and .NET CLR, and multiple OSs such as Microsoft Windows, Mac OS and UNIX-based OSs is defined.
(4) Programmers are provided with one-to-one mapping APIs to directly manipulate DirectX objects and other game-related properties for advanced programmers.

The contributions of this paper include the following five points: (1) a general and portable widget architecture for Java game development has been defined; (2) CWT-AWT and CWT-DX3 have been implemented; (3) the high performance of the proposed design has been demonstrated experimentally; (4) the performance inconsistency in Java versions has been pointed out; (5) the usability of the CWT by applying it to a real applet game, Mahjong, has been demonstrated.

Future extension includes OpenGL implementation and DirectX 9.0 implementation. In the former, since OpenGL are available on many platforms, the OpenGL implementation is better than DirectX implementation for cross-platform consideration. In the latter, since the .NET J# can access DirectX 9.0 via built-in APIs and Microsoft provides tools to convert Java 1.1 code into J# code, using .NET J# is another alternative so as to access the latest DirectX.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous referees for their valuable comments, which greatly improved this paper. They would also like to thank the National Science Council of the Republic of China, and ThinkNewIdea, Inc. for jointly financially supporting this research under contract number NSC 94-2622-E-009-011-CC3. The authors would also like to thank ThinkNewIdea, Inc. for providing the data required for this research.

DOI: 10.1002/spe

REFERENCES

1. Rollings A, Adams E. *Andrew Rollings and Ernest Adams on Game Design*. New Riders: Berkeley, CA, 2003.
2. Sweeney T. Tim Sweeney of Epic Games: A Critical Look at Programming Languages, Fargo Kosak D (ed.), GameSpy Industries, 2000. http://archive.gamespy.com/legacy/articles/devweek_c.shtm [12 September 2006].
3. ID Software Inc. Doom. http://www.idsoftware.com/ [12 September 2006].
4. Sun Microsystems Inc. Java Home Page. http://java.sun.com/ [12 September 2006].
5. Quinn E, Christiansen C. Java pays—positively. *IDC Bulletin #W16212*, 1998.
6. Phipps G. Comparing observed bug and productivity rates for Java and C++. *Software: Practice and Experience* 1999; **29**(4):345–358.
7. Yahoo Inc. Yahoo! Games. http://games.yahoo.com/ [12 September 2006].
8. IonChron Inc. Java Games. http://www.arcadepod.com/java/ [12 September 2006].
9. ThinkNewIdea Internet Technology Corp. CYC Games. http://cycgame.com/ [12 September 2006].
10. Hsu C-C, Wu I-C. An event-driven framework for inter-user communication applications. *Information and Software Technology* 2006; **48**:471–483.
11. Wu I-C, Hsu C-C. The Model and Systems for Play-on-table Games. *IEICE Transactions on Information and Systems* 2004; **E87-D**(11):2503–2508.
12. Jagex Limited. RuneScape. http://www.runescape.com/ [12 September 2006].
13. Marner J. *Evaluating Java for Game Development*. Department of Computer Science, University of Copenhagen: Denmark, 2002.
14. Microsoft Corp. Microsoft DirectX. http://www.microsoft.com/windows/directx/ [12 September 2006].
15. The OpenGL Architecture Review Board. OpenGL: Open Graphics Library. http://www.opengl.org/ [12 September 2006].
16. Sun Microsystems Inc. *The AWT in 1.0 and 1.1*. Sun Microsystems Inc., 1997.
17. Sun Microsystems Inc. *Abstract Window Toolkit (AWT)*. Sun Microsystems Inc., 1995.
18. Petzold C. *Programming Windows* (5th edn). Microsoft Press: Redwood, CA, 1998.
19. The Open Group. Motif GUI Toolkit. http://www.opengroup.org/motif/ [12 September 2006].
20. Sun Microsystems Inc. *Painting in AWT and Swing*. Sun Microsystems Inc., 2003.
21. Sun Microsystems Inc. *Java Foundation Classes: Now and the Future*. Sun Microsystems Inc., 1997.
22. Sun Microsystems Inc. *High Performance Graphics—Graphics Performance Improvements in the Java 2 SDK, version 1.4*. Sun Microsystems Inc., 2001.
23. Sun Microsystems Inc. *New Java 2D Features in J2SE 5.0*. Sun Microsystems Inc., 2004.
24. Sun Microsystems Inc. *Single-Threaded Rendering for OpenGL Pipelines*. Sun Microsystems Inc., 2005.
25. Sun Microsystems Inc. Bug ID: 6260751 Applets Can't Set sun.java2d.noddraw=true. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6260751 [12 September 2006].
26. IBM Corp. SWT: The Standard Widget Toolkit. http://www.eclipse.org/swt/ [12 September 2006].
27. Bellotti F, Berta R, De Gloria A, Poggi A. DirectJ: Java APIs for optimized 2D graphics. *Software—Practice and Experience* 2001; **31**(3):259–275.
28. Jausoft. GL4Java: OpenGL for Java. http://gl4java.sourceforge.net/ [12 September 2006].
29. Sun Microsystems Inc. JOGL, Java bindings for OpenGL API. https://jogl.dev.java.net/ [12 September 2006].
30. lwjgl.org. LWJGL, Lightweight Java Game Library. http://lwjgl.org/ [12 September 2006].
31. Sun Microsystems Inc. Java 3D. http://java.sun.com/products/java-media/3D/ [12 September 2006].
32. Microsoft Corp. Microsoft Java Virtual Machine support. http://www.microsoft.com/mscorp/java/ [12 September 2006].
33. Gray A. GC usage statistics. http://www.andrew-gray.com/dist/stats.shtml [12 September 2006].
34. Microsoft Corp. Microsoft Visual J#. http://msdn.microsoft.com/vjsharp/ [12 September 2006].
35. Gamma E, Helm R, Johnson R, Vissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley: Reading, MA, 1995.
36. Sun Microsystems Inc. *Java AWT: Delegation Event Model*. Sun Microsystems Inc., 1997.
37. Sun Microsystems Inc. *The VolatileImage API User Guide*. Sun Microsystems Inc., 2001.
38. Microsoft Corp. *DirectX: Platform SDK*. Microsoft Corp., 2000.
39. Pitman R. Charva: A Java windowing toolkit for text terminals. http://www.pitman.co.za/projects/charva/index.html [12 September 2006].
40. Sun Microsystems Inc. *System Properties for Java 2D Technology*. Sun Microsystems Inc., 2004.
41. Sun Microsystems Inc. Bug ID: 5104393 Provide d3d Pipeline to Improve Runtime Performance of Swing/2D. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=5104393 [12 September 2006].

DOI: 10.1002/spe