國立交通大學

資訊科學與工程研究所

博士論文

CWT – 跨平台 Java 遊戲開發之 AWT/Swing 架構

CWT – An AWT/Swing Architecture for Cross-Platform Java Game Development

研 究 生：汪益賢

指導教授：吳毅成　教授

蔡文能　教授

中華民國九十八年七月

CWT－ 跨平台 Java 遊戲開發之 AWT/Swing 架構
CWT – An AWT/Swing Architecture for Cross-Platform Java Game Development

研 究 生：汪益賢　　Student：Yi-Hsien Wang

指導教授：吳毅成　　Advisor：I-Chen Wu

蔡文能　　Wen-Nung Tsai

國 立 交 通 大 學
資 訊 科 學 與 工 程 研 究 所

博 士 論 文

A Dissertation

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

July 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年七月

CWT：跨平台 Java 遊戲開發之 AWT/Swing 架構

研究生：汪益賢　　　　　　　　　　　　指導教授：吳毅成 博士

蔡文能 博士

國立交通大學資訊科學與工程研究所博士班

摘要

近年來，Java 平台在效能上的長足進步，使其達到開發遊戲之所需。然而，實際應用 Java 開發遊戲之後，我們發現 Java 標準內建的視窗工具組 AWT 及 Swing 在各種不同的執行環境組態中存在繪圖效能不一致的問題，這些執行環境的組合包含下列四項：一、Java 執行環境版本（JRE）；二、繪圖應用程式界面（API）；三、Java 執行期系統參數；四、常用的作業系統，如 Windows XP、Windows Vista、Fedora 及 Mac OS X。這種效能不一致的現象使得遊戲開發人員難以預測 Java 遊戲在使用者電腦的繪圖效能，也降低了 Java 倡言的「撰寫一次即可隨處運行（Write-Once-Run-Anywhere）」的跨平台優勢。

為了解決繪圖效能不一致的問題，使遊戲在不同平台上執行皆能達到高速且一致的繪圖效能，我們提出了一套 AWT/Swing 架構，稱為 CYC 視窗工具組（CYC Window Toolkit），簡稱 CWT，具有下述數項特性。首先，CWT 架構支援多種常用的高速繪圖函式庫，如 DirectX 及 OpenGL，為了相容於沒有硬體加速的環境，CWT 也使用 Java AWT 來繪圖。CWT 架構也維持 Java 的跨平台特性，支援各 Java 虛擬機器（Java VM）、.NET 公共語言執行環境（.NET CLR）以及各種不同的作業系統。再者，CWT 提供與 Java AWT/Swing 1.1 版相同 API 的元件，降低將既有 Java 遊戲移植到 CWT 的

難度。對於需要進一步調整遊戲效能的程式設計師來說，CWT 提供一對一對應的API，以便直接操作 CWT 內部的 DirectX 與 OpenGL 物件，以及遊戲相關的各種參數。此外，CWT 也可以應用於 3D 應用程式，這對於現在流行的 3D 遊戲設計十分重要。

我們依照 CWT 架構實作了三種 CWT 工具組，並開放於專屬網站[15]。這三種實作各自使用 AWT、DirectX 及 OpenGL 來繪製使用者介面，尤其是後二項實作，可以在支援的作業系統中，得到繪圖加速卡的硬體加速支援。為了測試並比較原本 Java AWT/Swing 與 CWT 的繪圖效能，我們設計了兩支測試程式，用來測試基本繪圖能力（Micro-benchmark）與綜合繪圖能力（Macro-benchmark）。測試程式執行於常用的JRE，如 MSVM 及 JRE 1.4 至 1.6 版，以及前面提到的四個作業系統中，其結果顯示：CWT 比 Java AWT/Swing 更能在這些不同的組態中達到更好且更一致的繪圖效能。由於 CWT 提供 Java AWT/Swing 1.1 版的介面，也可以在 Java 1.1 版的環境下運行，因此，CWT 的 API 數量及 Java 執行期系統參數都較 Java AWT/Swing 少，較少的測試組合有助於提升程式設計師的生產力。

我們將實作 CWT 以及測試效能的經驗，歸納出三點方向，使 Java 在未來成為更好的跨平台遊戲的開發平台。一、由於繪圖加速卡的快速演進，Java 應開放內部的DirectX 與 OpenGL 物件，讓遊戲程式設計師能夠直接存取新功能或調整繪圖行為。二、Java AWT/Swing 的繪圖管線應該與 JRE 分開發行，有助於更快速地升級、除錯該繪圖管線，並支援舊版本的 JRE。三、複用現有的 DirectX 與 OpenGL 綁定（Bindings）以降低開發成本、提高可維護性、簡化運用 Java AWT/Swing 於 Java 3D 及 JOGL 應用程式中的難度，並提高效能。

CWT – An AWT/Swing Architecture for Cross-platform Java Game Development

Student：Yi-Hsien Wang　　Advisor：Dr. I-Chen Wu
Dr. Wen-Nung Tsai

Institute of Computer Science and Engineering
National Chiao Tung University

Abstract

In recent years, the performance of Java platforms has been greatly improved, which makes Java satisfy the requirements for developing games. However, after practicing in real game development, we observe that a phenomenon of *performance inconsistency* exists in the graphics of Java AWT/Swing with different combinations of JREs, graphics APIs, system properties, and operating systems (OSs), including Windows XP, Windows Vista, Fedora and Mac OS X. This phenomenon makes it hard to predict the rendering performance of Java games and weakens the merits of the *Write-Once-Run-Anywhere* feature of Java.

In order to solve the above problems, we propose a portable AWT/Swing architecture, called CYC Window Toolkit (CWT), for developing cross-platform Java games with high and consistent rendering performance. CWT has the following features. First, the CWT architecture supports multiple graphics libraries such as AWT, DirectX and OpenGL, multiple virtual machines such as Java VM and .NET CLR, and multiple OSs. Next, CWT supports AWT/Swing 1.1 compatible widgets, so it can be easily applied to existing Java games. For programmers who want to fine tune their games, CWT supports one-to-one

mapping APIs to directly manipulate DirectX and OpenGL objects and other game-related properties. In addition, CWT supports interoperability with 3D applications, which is an important feature for 3D game design.

We implemented three versions of CWT – AWT, DirectX, and OpenGL, to take advantage of graphics hardware acceleration on all supported OSs. The implementations of CWT are available on our website [15]. Two testing programs, including micro-benchmark and macro-benchmark, are also designed to evaluate the rendering performance of the original Java AWT/Swing and CWT. The benchmarking results show that CWT achieves more consistent and higher rendering performance than Java AWT/Swing does in commonly used JRES, including MSVM and JREs 1.4 to 1.6, on the four OSs. Moreover, CWT needs fewer efforts to test the combinations of graphics APIs and system properties, which greatly improves programmers' productivity.

Based on the benchmarking results and our experience, we propose three approaches to make Java be a better platform for developing cross-platform games in the future. First, since the video hardware evolves quickly, Java should open direct access to the internal DirectX and OpenGL objects for game programmers who need to access up-to-date hardware features or change the rendering behaviors. Second, Java should decouple the rendering pipelines of Java AWT/Swing from the JREs for faster upgrading and supporting old JREs. Third, Java should reuse existing DirectX and OpenGL bindings for lower developing cost, better maintainability, easier interoperability among Java AWT/Swing, Java 3D, and JOGL applications.

致謝

歷經了漫長的時間，突破了一道又一道的難關，我的博士生涯在 2009 年 7 月結束了。博士學位對我來說，是一項對於過程的肯定。除了我以外，還有許還有許多推動我不斷前進的師長同學、親朋好友的參與，我們互相鼓勵、互相扶持、互相激盪、互相砥礪，論文的想法論點、表達呈現、設計架構、實作測試在不斷地粹煉之下，逐步邁向充實與完善。我在此衷心感謝所有人的陪伴與參與。

感謝指導老師吳毅成教授和蔡文能教授，在我攻讀碩士與博士期間，不僅鞭策鼓勵我的學業研究，也指導提點我的做人處事。老師們一路上對我的信任與鼓勵，讓我無懼一道道挑戰的辛酸苦辣，享受突破後的沁涼甘甜。老師們理論與實務並重的治學方法，對我影響深遠。在我未來的人生道路，我將秉持老師的期勉，腳踏實地前進。

感謝論文口試委員朱正忠教授、吳昇教授、留忠賢教授、莊榮宏教授、黃世昆教授與焦惠津教授（以上按姓氏筆劃排列），對論文的缺失與方向，提出極具價值的建議。委員們的寶貴意見，讓我在未來的研究工作上，能以充滿信心的態度大步邁進。

感謝徐健智學長，從我初到網際網路應用實驗室起，帶領我學習實驗室發展的 Java 遊戲平台，種下此研究的種子。在研究各種網際網路遊戲技術以及軟體工程的過程中，和學長的討論總在令人享受與鼓舞的氛圍下進行。學長展現的研究熱忱，一直是我的榜樣。

感謝姜智耀、鄭欽議、吳秉儒三位學弟分別在 CWT 元件庫、CWT-GL 實作、以及 CWT-GL 結合 JOGL 等子研究上的協助，不但加速研究的進程，也讓我看到本論文研究在實務應用方面的可行性。

感謝所有實驗室學長、同學與學弟妹們的陪伴學習，一同分享研究中與生活中的喜怒哀樂。大夥兒在實驗室終日的奮戰，培養出的革命情感，令我終身難忘。

感謝群想網路科技的同事們，在產學合作的過程中，提供線上遊戲平台，讓我的博士論文除了理論架構之外，也將實務應用納入考量，大大地提升論文的涵蓋面與實用性，以符合遊戲業界需求。

最大的感謝，留給我最摯愛的爺爺、奶奶、媽媽及太太。感謝爺爺的苦心栽培、奶奶的慈愛關懷、媽媽的支持鼓勵、太太的包容扶持，讓我得以無後顧之憂，勇往直前。我摯愛的家人總是在我達到里程碑時，比我還要歡欣鼓舞；在我遇到困難挫折時，當作我最溫暖的避風港。家人的愛與關懷，是無與倫比的力量，提供我前進的動力。沒有他們，這篇論文將無法完成。我謹將此論文獻給我最摯愛的家人。

汪益賢

2009 年 7 月 15 日

Contents

摘要 i
Abstract iii
致謝 v
Contents vii
List of Figures ix
List of Tables xi
Chapter 1 Introduction 1
1.1 Evolution of Java Graphics 5
1.2 Problems of Java Graphics 8
1.3 Goals 12
Chapter 2 Design of CWT 15
2.1 CWT Architecture 15
2.2 Core CWT 18
2.2.1 Component Hierarchy 19
2.2.2 Event Model 22
2.2.3 Painting Model 24
Chapter 3 Implementation of CWT 27
3.1 Implementation Approaches 27
3.2 CWT-AWT 28
3.3 CWT-DX 29
3.3.1 Images and Rectangles 29
3.3.2 Figures and Texts 30
3.3.3 Optimization of CWT-DX 31
3.4 CWT-GL 32
3.4.1 Introduction to JOGL 32
3.4.2 Figures 32
3.4.3 Images 33
3.4.4 Texts 34
3.4.5 Off-screen Buffers 36
3.4.6 Graphics States 37
3.4.7 Optimization of CWT-GL 40
3.5 Mixing CWT-GL with JOGL 41

3.6 Related Work 43
3.6.1 Agile2D 44
3.6.2 FengGUI 44
3.6.3 Minueto 45
Chapter 4 Experiments 47
4.1 Test Programs 47
4.2 System Configuration 51
4.3 Rendering Environments (REs) 52
4.4 Definitions of Metrics 54
4.5 Analysis of Micro-Benchmark Results 54
4.5.1 Image Tests 55
4.5.2 Text Tests 57
4.5.3 Figure Tests 61
4.6 Analysis of Macro-Benchmark Results 63
Chapter 5 Discussion 85
5.1 Supporting Graphics Systems on Multiple Platforms 85
5.1.1 Encapsulation and Extension 88
5.1.2 Decoupling 89
5.1.3 Reuse 92
5.2 Drawbacks of CWT 95
Chapter 6 Conclusions 97
References 103
Appendix A Results of Micro-Benchmark 109
Appendix B Results of Macro-Benchmark 131
Appendix C Porting Guide 135
Vita 141

List of Figures

Figure 1. Java usage trend diagram. 3
Figure 2. CWT architecture. 16
Figure 3. Relation among components, events and graphics 19
Figure 4. Component hierarchy of Java AWT/Swing. 20
Figure 5. Component hierarchy of CWT. 20
Figure 6. Event hierarchy of Java AWT 22
Figure 7. A typical event processing flow in CWT. 23
Figure 8. Three implementations of the Graphics interface in CWT. 24
Figure 9. Eliminating unnecessary getting and releasing DC. 30
Figure 10. Sequence diagram of STR-like design in CWT. 39
Figure 11. Eliminating unnecessary changes of OpenGL state. 40
Figure 12. The flow of mixing CWT-GL with JOGL. 42
Figure 13. Agile2D architecture. 44
Figure 14. FengGUI architecture. 45
Figure 15. Screenshots of the micro-benchmarks. 48
Figure 16. A screenshot of the Bomberman game. 48
Figure 17. Averaged rendered images per seconds among OSs. 56
Figure 18. Averaged rendered texts per seconds among OSs. 59
Figure 19. Averaged rendered texts per seconds with different font sizes. 60
Figure 20. Averaged rendered figures per seconds among OSs. 62
Figure 21. Frame rates and Anomaly among OSs using Java 1.0/1.1 graphics APIs. 64
Figure 22. Frame rates and Anomaly among OSs using JRE∈{1.4} and SystemProperty∈{None}. 65
Figure 23. Frame rates and Anomaly among OSs using JRE∈{1.5} and SystemProperty∈{None}. 66
Figure 24. Frame rates and Anomaly among OSs using JRE∈{1.6} and SystemProperty∈{None}. 67
Figure 25. Frame rates and Anomaly among OSs using JRE∈{1.4} and SystemProperty∈{Special}. 68
Figure 26. Frame rates and Anomaly among OSs using JRE∈{1.5} and SystemProperty∈{Special}. 69
Figure 27. Frame rates and Anomaly among OSs using JRE∈{1.6} and

SystemProperty∈{Special}. 70
Figure 28. Frame rates and Anomaly among OSs using JRE∈{1.5} and SystemProperty∈{OpenGL}. 71
Figure 29. Frame rates and Anomaly among OSs using JRE∈{1.6} and SystemProperty∈{OpenGL}. 72
Figure 30. Frame rates and Anomaly on choosing different graphics APIs using JRE∈{1.4}. 75
Figure 31. Frame rates and Anomaly on choosing different graphics APIs using JRE∈{1.5}. 76
Figure 32. Frame rates and Anomaly on choosing different graphics APIs using JRE∈{1.6}. 77
Figure 33. Frame rates and Anomaly in commonly used JREs using Java 1.0/1.1 graphics APIs. 79
Figure 34. Frame rates and Anomaly in commonly used JREs using graphics APIs introduced since J2SE 1.4. 80
Figure 35. Frame rates and Anomaly in commonly used JREs using graphics APIs introduced since Java 5.0. 81
Figure 36. Java AWT/Swing and DirectX in MSVM. 89
Figure 37. Hardware-accelerated rendering pipelines supported in specific JVM versions. ... 90
Figure 38. Relation between JOGL and JVM. 91
Figure 39. Two DirectX bindings and three OpenGL bindings by Sun Microsystems. 92
Figure 40. Individual buffers used by JOGL programs and AWT/Swing, which does not allow translucent widgets. 93
Figure 41. A shared buffer used by JOGL programs and CWT-GL, which allows translucent widgets. 93
Figure 42. Suggestions for future Java AWT/Swing. 94
Figure 43. RuneScape in 765×503-sized resolution. 100
Figure 44. RuneScape in 1024×768-sized resolution. 100
Figure 45. Comparison between raster graphics and vector graphics. 101

List of Tables

Table 1. Current state of JREs. 3
Table 2. Performance evolution of Java AWT/Swing. 6
Table 3. Percentages of OSs [62]. 11
Table 4. OpenGL states for CWT-GL. 43
Table 5. Graphics APIs Tested in the Benchmarks. 50
Table 6. System properties for SystemProperty∈{Special} [3][58]. 50
Table 7. System hardware, configuration and OSs. 51
Table 8. JRE versions in the benchmarks. 51
Table 9. Combinations of graphics APIs which deliver the highest frame rates. 74
Table 10. OS support of Sun's rendering pipelines. 87
Table 11. Rendered items per second of opaque image tests. 110
Table 12. Rendered items per second of transparent image tests. 111
Table 13. Rendered items per second of translucent image tests. 112
Table 14. Rendered items per second of runtime opaque image tests. 113
Table 15. Rendered items per second of runtime transparent image tests. 114
Table 16. Rendered items per second of runtime translucent image tests. 115
Table 17. Rendered items per second of simple text tests. Font size is 12. 116
Table 18. Rendered items per second of article tests. Font size is 12. 117
Table 19. Rendered items per second of texts with different font size from 10 to 64. 118
Table 20. Rendered items per second of line tests. 119
Table 21. Rendered items per second of polyline tests. 120
Table 22. Rendered items per second of polygon tests. 121
Table 23. Rendered items per second of polygon filling tests. 122
Table 24. Rendered items per second of rectangle tests. 123
Table 25. Rendered items per second of rectangle filling tests. 124
Table 26. Rendered items per second of round rectangle tests. 125
Table 27. Rendered items per second of round rectangle filling tests. 126
Table 28. Rendered items per second of arc tests. 127
Table 29. Rendered items per second of arc filling tests. 128
Table 30. Rendered items per second of oval tests. 129
Table 31. Rendered items per second of oval filling tests. 130
Table 32. Average frame rate (in FPS) of the Bomberman game. 131

Table 33. Average frame rate (in FPS) of the Bomberman game, where JRE∈{1.5}. 132
Table 34. Average frame rate (in FPS) of the Bomberman game, where JRE∈{1.6}. 133

Chapter 1 Introduction

Since released by Sun Microsystems Inc. (abbreviated as Sun) in 1995, Java [48] has become increasingly popular owing to its higher productivity and portability than C/C++. For productivity, a report [39] 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 [37] also presented a similar result which concluded that Java was 30~200% more productive than C++. For portability, unlike C/C++ programs, which have to be compiled to native executable code specifically for each platform, Java programs are compiled into a format called bytecode running atop *Java virtual machine* (JVM), which encapsulates platform-specific features and provides a common set of *application programming interfaces* (APIs) on all supported platforms. Therefore, such "*Write Once, Run Anywhere*" (WORA) feature makes Java more portable than C/C++.

Java has attracted much attention in game industry. Along with the growth of *World Wide Web* (WWW) in the late 1990s, many Java casual applet games, which can run in Web browsers, were deployed over the Internet, including *Yahoo! Games* [68], *ArcadePod.com* [16] and *CYC games* [59][13]. Other than the widespread applet games, several commercial stand-alone Java games were also developed, such as *You Don't Know Jack* [19], *Law & Order: Dead on the Money* [61] and *Tribal Trouble* [33]. Examples of commercial *massively multiplayer online* (MMO) Java games include *RuneScape* [17], *Puzzle Pirates*

[60] and *Wurm Online* [32]. 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* [27] was ported to mobile devices [14].

After these practices in game industry, however, Java has some criticisms. The most discussed topics include runtime speed, rendering performance, and deployment issues. First of all, early implementations of the JVM normally indeed delivered poorer performance. In general, the performance of programs running in JVM 1.0 is about 20 to 40 times slower than in C/C++ [23]. Fortunately, after several significant revisions in the JVM, the tweaked Java programs using J2SE 1.4 ran on the average only about 20-50% slower than the tweaked C/C++ programs [23]. Java SE 5.0 is typically only 1.1 times slower [5]. Sun's benchmarks also suggest that Java SE 6 is 20% to 25% faster than Java SE 5.0 [49]. Therefore, the runtime speed is no longer a serious problem for Java game development.

As for the *graphical user interface* (GUI) part of Java, early implementation of Java AWT/Swing components or widgets also performed slowly due to the lack of graphics hardware acceleration. Since most game programs, especially high profile games[1], have intensive GUI operations, such as animation or complex scenes, it is critical to reach high rendering performance. In order to deal with this issue, *Microsoft DirectX* [30] and *Open Graphics Library* (OpenGL) [35], which are two major graphics libraries used in game industry, were introduced in the implementations of Java AWT/Swing since J2SE 1.4 and Java SE 5.0, respectively. With the supports of graphics hardware acceleration, the rendering performance of Java is largely improved when compared with previous versions. However, current implementations still have some problems that limit Java game

[1] According to [[28]], high-profile games usually attempt to attract the highest attention from retailers and media. Such games normally require several millions of US 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.

development, such as inconsistent rendering performance in different Java versions and on different platforms. In this dissertation, we will mainly investigate these problems.

Table 1. Current state of JREs.

<table>
<tr><th>Java Version</th><th>Supported OSs</th><th>Released Time</th><th>JRE Size (MB)</th><th>Percentage of Web Browser Users (May 2009)</th></tr>
<tr><td>MSVM (Java 1.1.4)</td><td>Windows</td><td>1997/02</td><td>5.0</td><td>4.81%</td></tr>
<tr><td>J2SE 1.3</td><td rowspan="4">Windows
Mac OS
Linux
Solaris</td><td>2000/05</td><td>7.9</td><td>0.20%</td></tr>
<tr><td>J2SE 1.4</td><td>2002/02</td><td>15.2</td><td>5.11%</td></tr>
<tr><td>J2SE 5.0</td><td>2004/09</td><td>15.8</td><td>18.63%</td></tr>
<tr><td>Java SE 6</td><td>2006/11</td><td>15.5</td><td>71.02%</td></tr>
</table>

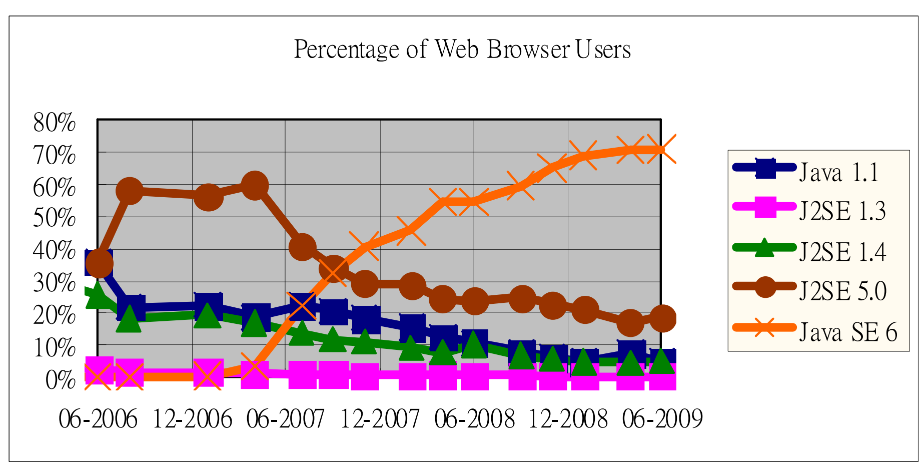


Figure 1. Java usage trend diagram.

As for the deployment issues, several problems have to be considered. First of all, few game consoles are shipped with JVMs. However, about 70% of the game market is console games, and *personal computer* (PC) games own the most of the rest. The lack of JVM supports greatly limits the deployment of Java games on the consoles.

The next deployment problem is that not all PCs have *Java runtime environment* (JRE) installed. According to Millward Brown's survey in December 2008 [1], only 81% Internet-enabled PCs have JRE installed. On some platforms such as Windows 98, ME, 2000, early release of XP, and Mac OS X, certain JREs are pre-installed along with the

operating systems (OSs). On other platforms, users need to install JREs by themselves before they can play Java games. However, the installation may not be allowed without administrator's privileges. The problem that required JRE versions are not installed has variant influence on Java program types. Stand-alone Java applications can be shipped with a JRE, so they can run on target systems without such problem. However, Java applets or web start applications need pre-installed JREs, which causes deployment issues.

Even if OSs have JRE installed, the installed JRE version may not sufficient to run Java games. In such case, upgrades are required. For example, in order to enable the OpenGL rendering pipeline on Windows, Java SE 5.0 is required. According to the statistical data in [12] during April to May in 2009, as shown in Table 1, the percentages of Web browser users of popular JRE 1.1, 1.4, 5.0, and 6 are 4.81%, 5.11%, 18.63%, and 71.02%, respectively. Figure 1 shows that at any given time since June 2006 when this research started, there are always at least three major JRE versions used by more than 5% Web browser users, including twelve-year-old MSVM. To support most installed JRE versions, Java programmers could be limited to old Java versions. However, many features, especially improvements of graphics performance, are only available in new JREs.

In view of these problems, research for Java Graphics that is reviewed in Subsection 1.1 has been done to make Java more suitable for game development. However, some problems which are identified in Subsection 1.2 still remain in the GUI part of Java, in terms of AWT and Swing, especially when programmers try to deploy cross-platform Java games with high and consistent rendering performance. By consistent rendering performance, we mean to deliver similar rendering performance on different OSs or different rendering environments when using the same 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. Subsection 0

briefly describes our goals for solving these problems, and also summarizes the organization of the rest of this dissertation.

1.1 Evolution of Java Graphics

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

For high rendering performance, game developers commonly employ Microsoft DirectX or OpenGL to access specialized hardware features, such as direct access to the video memory in graphics cards, and constructing 3D scenes.

Using Java AWT/Swing is the standard way to perform rendering operations in Java. However, Java AWT/Swing did not take full advantage of graphics cards, before J2SE 1.4. As shown in Table 2, for example, Java 1.0/1.1 partially uses *Windows graphics device interface* (GDI) on Microsoft Windows platforms to accelerate image operations. Since J2SE 1.2, buffered images have been introduced, which lets programmers directly access pixels of images. However, since J2SE 1.2, software rendering was employed, instead of the hardware acceleration, to guarantee rendering quality on all platforms. Consequently, the rendering performance degrades in these Java versions.

Since J2SE 1.3, *AWT Native Interface* has been introduced which allows programmers to render into Java AWT components through third parties' graphics libraries, such as DirectX and OpenGL. This way is an alternative to using *Java Native Interface* (JNI) to access native libraries. However, the main drawback of using this technology is the loss of platform independency, which is a great concern in developing cross-platform games.

Table 2. Performance evolution of Java AWT/Swing.

Java Version	Graphics-Related Enhancement	Release Time
1.0	AWT (OS rendered widgets)	May 1995
1.1	Hardware-accelerated rendering using GDI on Microsoft	Feb. 1997
1.2	Swing, Java2D (Software rendered widgets)	Dec. 1998
1.3	AWT Native Interface	May 2000
1.4	Hardware-accelerated rendering • DirectX on Microsoft Windows • Shared Memory Extension (SHM) on X Window systems • Quartz 2D on Apple Mac OS X	Feb. 2002
5.0	OpenGL pipeline on Windows, Linux and Solaris	Sep. 2004
6	Improved OpenGL rendering pipeline	Nov. 2006
6u10	Improved DirectX rendering pipeline	Nov. 2008

In order to enhance the rendering performance of pure Java programs, Sun has started to access graphics hardware features via DirectX since J2SE 1.4 [45] and OpenGL since J2SE 5.0 (or 1.5) [52]. After that, Sun keeps improving the DirectX-based Java 2D pipeline (abbreviated as DirectX pipeline) and the OpenGL-based Java 2D pipeline (abbreviated as OpenGL pipeline). For example, in first release of Java SE 6, the OpenGL pipeline has been redesigned to improve its usability. In Java SE 6 update 10, the DirectX pipeline has also been redesigned. Since J2SE 1.4, 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 [45]. Since then, the rendering performance of Java AWT/Swing has had a great boost. In particular, the use of OpenGL which is supported by multiple platforms is quite important to Java in which the cross-platform feature is critical.

However, as shown in Table 2, these hardware-accelerated rendering pipelines still have the following two limitations. First, the rendering pipelines are not ported back to old Java versions, since they are tightly bundled with specific Java versions. The rendering

pipelines are also not available on all platforms. For example, currently the OpenGL pipeline can only be enabled in JRE versions 1.5 and beyond on Windows and Linux, and JRE version 1.6 on Mac OS X 10.5.2. Second, the OpenGL pipeline is disabled by default, because it does not work well due to some hardware and driver issues [55]. 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 [55].

According to the analysis above, Sun' official implementations of Java AWT/Swing are still not good enough for developing cross-platform games with high rendering performance. Alternatively, several 3D graphical libraries were developed to build cross-platform Java games with high rendering performance, including *the OpenGL for Java* (GL4Java) [18], *Java binding for OpenGL* (JOGL) [50], *Lightweight Java Game Library* (LWJGL) [22] and *Java 3D* [46]. The first three libraries are OpenGL bindings, which provide low-level one-to-one mapped APIs to OpenGL. Using GL4Java, JOGL or LWJGL, Java programmers can access hardware features supported in OpenGL without writing 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* [61], *Jake2* [7] and *Wurm Online* [32], were created using these 3D graphical libraries, instead of 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 subsection, seven problems are still identified when Java AWT/Swing is employed to develop cross-platform games.

(1) Backward compatibility to old JREs without graphics acceleration

As described in Subsection 1.1, Java AWT/Swing and most 3D libraries require at least J2SE 1.4 to achieve high rendering performance. However, Table 1 and Figure 1 indicate that currently about 5% 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, the APIs of OpenGL and DirectX are different from that of Java AWT/Swing. Unlike Java AWT/Swing, both OpenGL and DirectX do not provide widget systems, which may decrease the productivity of Java game programmers. 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 [42]. 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 [54], J2SE 1.3 introduced AWT Native Interface that enables native code to draw directly on Java drawing surface [53], J2SE 1.4 introduced DirectX pipeline, while Java SE 5.0 introduced OpenGL pipeline. These significant changes results 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 shown in Section 4.5, the rendering performance of texts and figures drop seriously from Java 1.1 to J2SE 1.2, and from J2SE 1.3 to J2SE 1.4. Therefore, such a phenomenon may make programmers hard to tune up the performance for all of the JRE versions.

(4) Inconsistent rendering performance among different operating systems

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 GDI and DirectX on Microsoft Windows platforms, X Window System (X) [66] on Linux, and Quartz graphics layer (Quartz) [4] 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 [28]. 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 graphics 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 to old JREs. When programmers want to support legacy Java users, they may either only use old graphics API or write several versions of programs which access different graphics 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 graphics 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 [58]. 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 programmers. Consequently, it is hard for users to use proper settings that programmers want, or to set these system properties without administrator's rights or help, e.g., the system properties in the Java applets of the Web browsers [43]. Thus, this problem makes it hard for programmers to predict rendering performance on end users' systems.

(7) No direct access to internal DirectX and OpenGL functionalities

Since J2SE 1.4, Java starts to access hardware acceleration. For backward compatibility and portability on multiple platforms, the DirectX and OpenGL rendering pipelines are encapsulated in Java 2D API and are not accessible directly by programmers. However, in recent 15 years, since the graphics hardware evolves quickly, many new features are available as time goes by. Game industry typically tries to enhance the quality and performance of games by accessing these new features. Therefore, the approach of encapsulation by Sun may limit the Java game development for the following two reasons. First, Sun's implementations of rendering pipelines may not support game-related features, such as translucent widgets. Second, Sun's implementations rarely provide extensibility such as OpenGL's shaders for programmers to implement features not provided by fixed functionality.

Table 3. Percentages of OSs [62].

<table>
<tr><th colspan="2" rowspan="2">OS</th><th colspan="2">Percentage</th></tr>
<tr><th>May, 2006</th><th>May, 2009</th></tr>
<tr><td rowspan="2">Windows</td><td>2000/XP</td><td>84.9%</td><td>68.3%</td></tr>
<tr><td>Vista</td><td>0.0%</td><td>18.4%</td></tr>
<tr><td colspan="2">Mac OS</td><td>3.6%</td><td>6.1%</td></tr>
<tr><td colspan="2">Linux</td><td>3.4%</td><td>4.1%</td></tr>
</table>

1.3 Goals

In view of the seven problems listed in Subsection 1.2, our goal is to enhance the rendering performance of Java for cross-platform game development. According to our experience, we also propose some directions which may make Java more suitable for developing cross-platform games in the future. This dissertation consists of three major parts as follows.

(1) Evaluate the rendering performance of Java AWT/Swing with different combinations of JREs, graphics 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 3. The evaluation results indicate that the performance inconsistency of Java AWT/Swing exists among the four OSs, even if the same hardware configuration is used. In addition, the results also show that no specific graphics APIs and system properties are guaranteed to obtain high and consistent rendering performance in different JREs. The problems weaken the merits of Java's *Write-Once-Run-Anywhere* feature.

(2) Propose solutions to solve the above problems of Java AWT/Swing and compare the results with those of Java AWT/Swing. We propose a window toolkit called CYC Window Toolkit (CWT), a fast-rendering lightweight GUI toolkit which renders all its widgets via native graphics libraries. We implement CWT using DirectX, OpenGL, and Java AWT. The benchmarking results show that CWT achieves more consistent and higher rendering performance in commonly used JREs, including MSVM and JRE 1.4 to 1.6, on four OSs.

(3) Propose three suggestions to future development of Java AWT/Swing. First, the internal DirectX and OpenGL objects should be accessible for game programmers

who need to access up-to-date hardware features or change the rendering behaviors. Second, decouple the rendering pipelines of Java AWT/Swing from the JREs for faster upgrading and supporting old JREs. Third, the bindings of DirectX and OpenGL should be reused for lower developing costs, better maintainability, easier interoperability among Java AWT/Swing, Java 3D, and JOGL applications.

The rest of this dissertation is organized as follows. Chapter 2 presents the design of CWT. Chapter 3 introduces three implementations and optimization techniques of CWT. Chapter 4 describes the configurations of JREs and benchmark programs used in this dissertation. This chapter also analyzes the experimental results. Chapter 5 discusses the software development problems of Sun's Java AWT/Swing. Solutions for making Java a better game platform are also proposed. Finally, Chapter 6 concludes our work and suggests possible future extensions of CWT.

Chapter 2 Design of CWT

This section describes the design of CYC Window Toolkit (CWT). First, the architecture of CWT is given in Subsection 2.1. The CWT architecture encapsulates multiple graphics libraries and provides a Java AWT/Swing compatible API, which can be further divided into three major parts: component hierarchy, event model, and painting model. These parts are designed by mostly following the design of Java AWT/Swing.

Following the architecture, we have implemented three implementations of CWT, using three graphics libraries: DirectX, OpenGL, and AWT, respectively. Each implementation has its own main goal. The DirectX implementation enhances the rendering performance of MSVM, the OpenGL implementation improves the rendering performance on multiple platforms, and the AWT implementation acts as a backup while neither DirectX nor OpenGL is available. Finally, we summarize work related to CWT, including *Agile2D*, *FengGUI*, and *Minueto*.

2.1 CWT Architecture

CWT, as shown in Figure 2, is designed to provide high and consistent rendering performance for cross-platform Java game development, while keeping the same APIs of Java AWT/Swing and backward compatibility to Java 1.1. For the part of high rendering performance, CWT uses DirectX and OpenGL to render AWT/Swing widgets, so the graphics performance is improved through video hardware acceleration. As for users with limited video hardware where DirectX or OpenGL is not available, CWT uses Java AWT to render widgets. Next, for the parts of 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 AWT/Swing APIs.

In Figure 2, we define three wrapper implementations, including CWT-DX, CWT-GL, CWT-GL, and CWT-AWT, which are introduced as follows.

(1) For DirectX, CWT accesses DirectX 3.0, which is supported in Microsoft Java VM (MSVM) [31] via a wrapper identified as CWT-DX.

(2) For OpenGL, CWT accesses it via OpenGL bindings, identified as CWT-GL. There are several candidate libraries: GL4Java (supporting OpenGL 1.3), JOGL and LWJGL (both supporting OpenGL 2.0). All these libraries are generally available in various OSs, including Windows, Mac OS X, Linux, and Solaris. In this dissertation, we choose JOGL to implement CWT-GL due to its official supports from Sun.

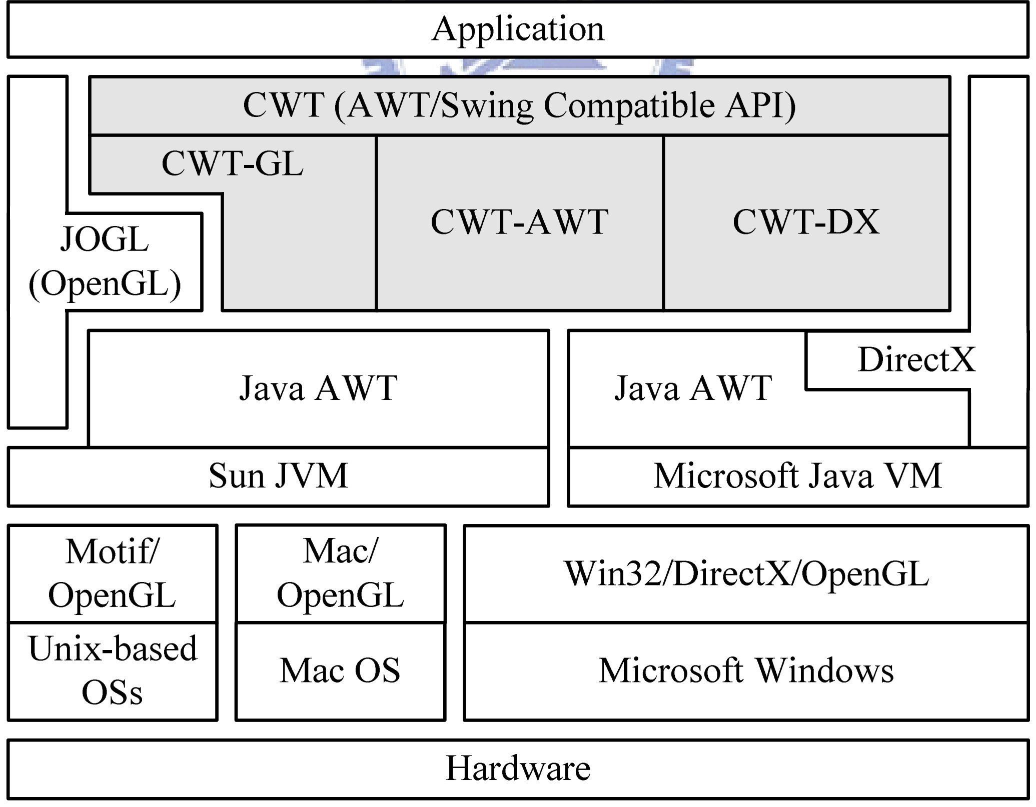


Figure 2. CWT architecture.

(3) When neither DirectX nor OpenGL is supported by the underlying OSs, CWT accesses Java AWT via a simple wrapper, identified as CWT-AWT.

Besides the full advantage of the hardware acceleration for most commonly used JREs even including 1.1, with this architecture, CWT also offers the following features.

(1) Allow programmers to access internal DirectX and OpenGL objects directly to manipulate more hardware features and to tune performance.

(2) Support mixing AWT/Swing widgets with DirectX and OpenGL.

(3) Provide more game-related features such as translucent widgets.

According the analysis above, CWT helps solve and improve the seven problems mentioned in Subsection 1.2, as follows.

(1) For the problem of backward compatibility to old JREs without graphics acceleration, the rendering performance of MSVM has been improved by CWT-DX [63]. Together with CWT-GL [64] for J2SE 1.4 and beyond on multiple OSs designed in this dissertation, CWT covers in total 99.80% of Web browser users in Table 1, where MSVM, J2SE 1.4, Java SE 5.0, and Java SE 6 are used by 4.81%, 5.11%, 18.63% and 71.02% 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, such as OpenGL, 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 graphics APIs, CWT provides one set of graphics API which is compatible to Java 1.1. This improves the compatibility issue and reduces test efforts. The details are described in Subsection 4.6.

(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 Subsection 4.6.

(7) For the problem of no direct access to internal DirectX and OpenGL functionalities, CWT allows programmers to access the internal DirectX and OpenGL objects directly so that the programmers can manipulate more hardware features.

In this dissertation, we have implemented three wrapper implementations: CWT-AWT, CWT-GL, and CWT-DX. All the implementations as well as demonstrations are available on our website [15]. We also put a porting guide in Appendix C.

2.2 Core CWT

Supporting a Java AWT/Swing compatible API, CWT consists of three major parts: component hierarchy, event model, and painting model. The relation among the three pars is shown in Figure 3. The component hierarchy models a hierarchical component structure similar to Java AWT/Swing 1.1. The event model specifies the event-handling process. The

painting model defines an abstract class called Graphics which allows Java programs to draw on components realized on various devices. In CWT, Graphics is implemented by various graphics libraries, including Java AWT, DirectX and OpenGL. Components are rendered via Graphics instances onto native screen resources.

Components and Events		
Graphics (Painting)		
CWT-GL	CWT-AWT	CWT-DX3
JOGL (OpenGL 2.1)	Java AWT (Motif/Mac/Win32)	Microsoft Java SDK (DirectX 3.0)

Figure 3. Relation among components, events and graphics

2.2.1 Component Hierarchy

Java AWT adopts the *Composite pattern* [11] that allows programmers to build a 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. Container is a special component which can contain other components, including Container itself, and can arrange and resize the components inside.

Java AWT uses peer architecture to maintain native look-and-feel of widgets on various OSs. To support the cross-platform feature, Java AWT provides a common set of GUI components on different OSs and peers implemented on each OS that connect the GUI components and underlying native GUI systems. For example, WButtonPeer is implemented on Windows platforms, which presents the Button component in Windows look-and-feel, while MButtonPeer provides Motif look-and-feel on Linux. As shown in

Figure 4, the components which have their own opaque native parts are called heavyweight components. For deciding which peers to be created in this design, an abstract class called `Toolkit` should be implemented on each OS. For example, `WToolkit` works on Windows platforms and creates peers in Windows look-and-feel.

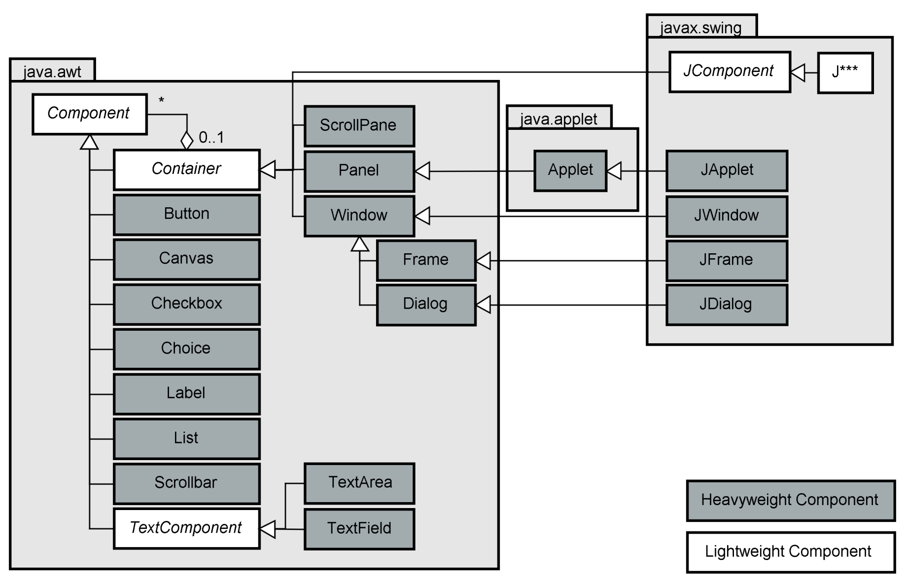


Figure 4. Component hierarchy of Java AWT/Swing.

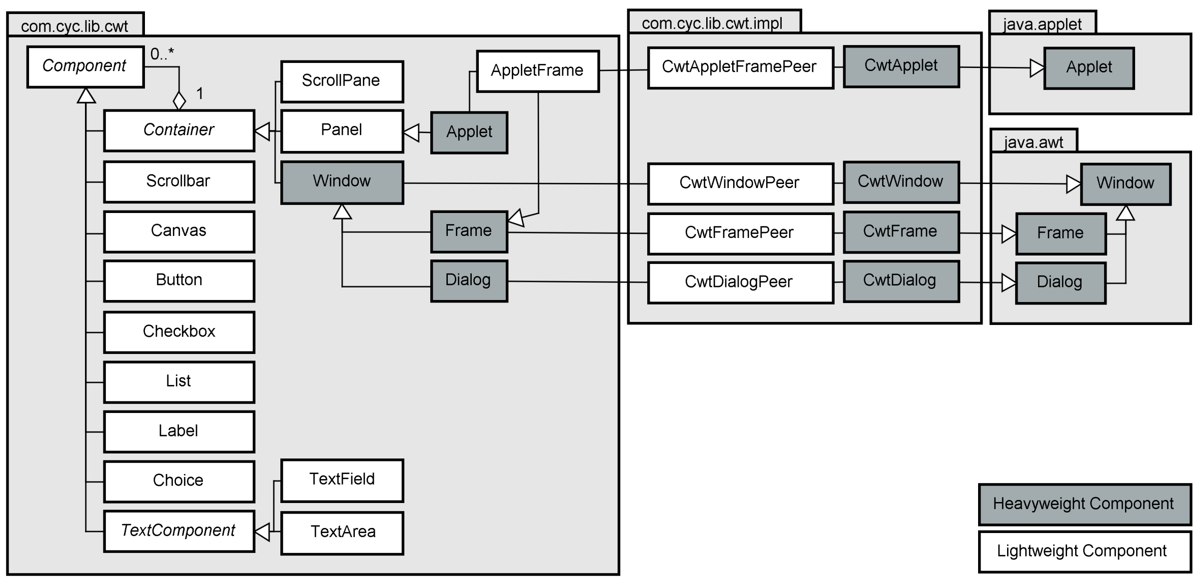


Figure 5. Component hierarchy of CWT.

In contrast, Java Swing provides lightweight components which are drawn on heavyweight containers, including re-implemented AWT heavyweight components. The lightweight components have no peers to connect to native components and are rendered by Java, not by OSs. The lightweight technology allow Java Swing provide more high-level components which are not natively available on all OSs, such as tree view, list box, and tabbed panes. Swing also supports pluggable look-and-feel so that programs may have Motif look-and-feel when running on Windows platforms. These features are important for game development, since game programs normally render GUI by themselves instead of using native components.

The component hierarchy of CWT is similar to that of Java AWT/Swing but different in the implementation of components. As shown in Figure 5, all heavyweight components of Java AWT, except for `Applet`, `Window`, `Frame` and `Dialog`, are redesigned as lightweight components in CWT. The peer architecture is still preserved in CWT so that we only need to modify much fewer parts of the entire AWT architecture. In this design, the rendering of the lightweight components is performed in corresponding peers. Furthermore, the lightweight components need at least one heavyweight component at the top level to draw on. Therefore, CWT internally wraps four corresponding AWT heavyweight containers – `Window`, `Frame`, `Dialog`, and `Applet`, for rendering all the lightweight components. When CWT programs initiate these containers, the wrapped AWT components are created to show other CWT lightweight components.

CWT `Applet` is designed in a different way from other three heavyweight components. Since `Applet` is a kind of `Panel` which requires to be contained in a `Window` root instance, a new container named `AppletFrame`, a kind of CWT `Frame`, is created for containing CWT `Applet` instances. As shown in Figure 5, the `AppletFrame` has its own peer implementation which wraps an AWT `Applet` instance as the canvas. In this design, the price to pay,

however, is the changes of <applet> tag in HTML files. Since the browsers can only accept the AWT `Applet` type and in CWT only the `CwtApplet` class inherits the AWT `Applet` class, programmers need to modify the applet tag so that the browsers launch `CwtApplet` and then `CwtApplet` launches target applets by reading the applet parameter "`cwtapplet`". This can be illustrated by the following code segments.

Original applet tag:

```
<applet code=YourApplet.class></applet>
```

Modified applet tag:

```
<applet code=com.cyc.lib.cwt.applet.CwtApplet.class>
    <param name="cwtapplet" value="YourApplet">
</applet>
```

2.2.2 Event Model

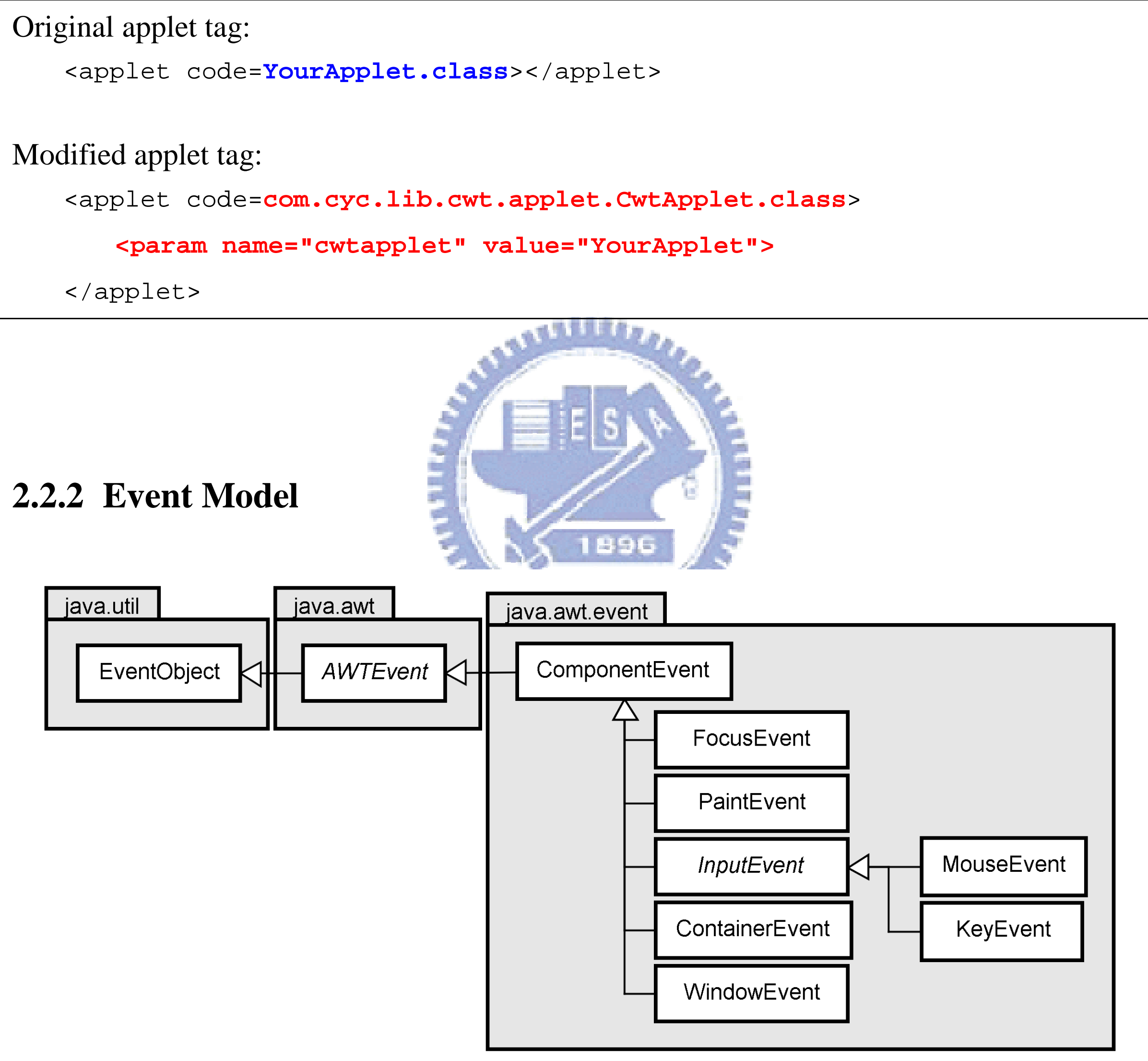


Figure 6. Event hierarchy of Java AWT

CWT follows the event hierarchy of Java AWT, as shown in Figure 6, and the *delegation-based event model* [47] used in Java 1.1 and beyond. However, the event processing flow is slightly different from Java AWT, as follows. In CWT, the four wrapped

AWT heavyweight components mentioned in the previous subsection get native events from OSs and dispatch them to the contained lightweight components. In order to dispatch events to the proper components, the four container peers, CwtAppletFramePeer, CwtWindowPeer, CwtFramePeer, and CwtDialogPeer, as shown in Figure 5, act as event listeners of the four heavyweight components. These listeners analyze the original AWT events, generate corresponding CWT events, and put the new events into a event queue. Then, an event dispatching thread dispatches the events to the proper CWT components. Figure 7 illustrates the event-processing flow in the following steps.

- Step 1: a mouse moving event occurs in CwtFrame, which is a wrapped AWT Frame. This event is handled by CwtFramePeer implementing the mouse motion listener.
- Step 2: the CwtFramePeer gets the mouse position from the event object and finds out which lightweight components in CWT Frame the mouse is on.
- Step 3: the CwtFramePeer generates a new CWT mouse event with translated position and puts it into EventQueue.
- Step 4: the thread EventDispatchThread retrieves the event and dispatches it to the component where the mouse is moving over.

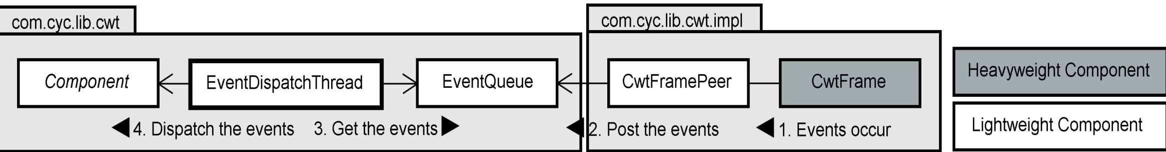


Figure 7. A typical event processing flow in CWT.

2.2.3 Painting Model

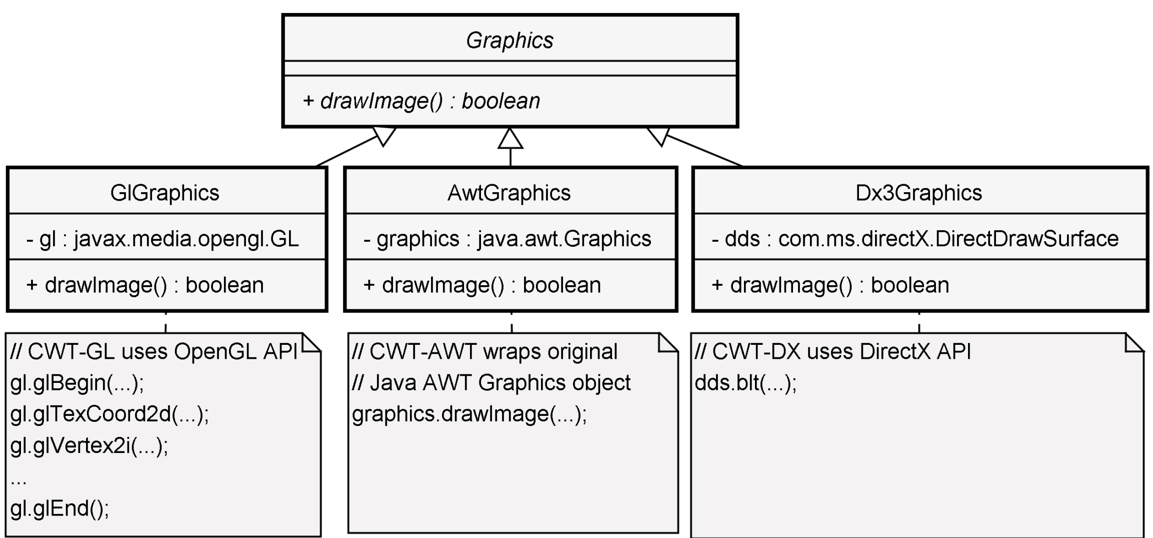


Figure 8. Three implementations of the Graphics interface in CWT.

The painting model is the key for rendering, and it is especially important for games. Since the rendering operations are eventually performed on native components on multiple platforms, this model defines a common set of rendering API for the various native GUI systems. In Java AWT, the abstraction of rendering is mainly in three classes: `Graphics`, `Image`, and `Font`.

The AWT `Graphics` class is an abstract class which defines basic rendering operations, such as rendering primitives, texts, and images. The `Image` class is also an abstract class which represents native image resources. The two abstract classes are realized by using various graphics libraries on different OSs, such as Windows GDI and DirectX on Microsoft platforms, X Window System on Linux, Quartz graphics layer on Mac OS X, and OpenGL on multiple OSs. On the other hand, the `Font` class adopts the peer architecture to connect the native font information needed for rendering texts.

Both Java AWT and Swing use a callback mechanism for painting [56]. 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. On different GUI systems, corresponding `Graphics` instances are created dynamically in runtime so that they can render on the native components. Since J2SE 1.2, a delegation design is introduced to `Graphics` instead of the inheritance design. This design is also apply to Image and Font classes which also

In CWT, the functions of the `Graphics` and `Image` classes are implemented by using different graphics libraries. As shown in Figure 8, several subclasses implement the abstract class Graphics by Java AWT Graphics, DirectX, and OpenGL, called CWT-AWT, CWT-DX, and CWT-GL, respectively. The `Font` class is designed in a different way. Since we directly access the font resource provided by Java AWT, there is no need to implement different font peers for each GUI system.

To support efficient lightweight painting and game-related features, CWT follows the design of Swing. Many Swing features are applied to the CWT components. The most important features are transparency, double buffering, and optimized repaint process [56].

1896

Chapter 3 Implementation of CWT

Based on the architecture of CWT described in Chapter 2, three graphics libraries are chosen to implement CWT, including Java AWT, DirectX 3.0 supported in MSVM, and JOGL. These implementations are identified as CWT-AWT, CWT-DX, and CWT-GL, respectively. CWT-AWT is a simple wrapper of Java AWT, while the rest two require much more efforts to implement and optimize. In this chapter, we will briefly introduce the most important techniques employed in the CWT implementations.

3.1 Implementation Approaches

There are two approaches to implement AWT compatible window toolkits: peer extension and reimplementation.

- **Peer extension approach.** Java uses it to support pluggable look and feel on different OSs by implementing the interfaces in package `java.awt.peer`. Though these interfaces were originally designed for cross-platform, we can replace the peer implementation by a different one, like Charva [38]. With this approach, we need to modify no Java code but just set the class name of the new toolkit in a system property called "`awt.toolkit`" before running the Java programs.
- **Reimplementation approach.** Using this approach, we have 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 "`com.cyc.lib.cwt`", and then recompile their programs.

This dissertation adopts the reimplementation approach, since it has the following three

advantages. First, better algorithms, such as the `RepaintManager` in Swing, or bug fixes, such as the focus issues in Java 1.1, can be applied to CWT to improve the performance outside the peer part. Second, CWT interface for game development may need to be extended in the future, but the peer extension approach can hardly achieve this goal. Third, CWT interface remains unchanged even when new Java version adds more methods. With peer extension approach, the added methods in new Java versions have to be implemented in CWT, or CWT could not be run in the new versions. For these reasons, our implementation is based on the reimplementation approach.

3.2 CWT-AWT

CWT-AWT, basically trivial to implement, is to keep the portability in different JVMs on different OSs. In CWT-AWT wrapper implementation, an instance of Java AWT Graphics is wrapped in CWT Graphics. Since both Graphics objects have the same interface, invoking methods of CWT Graphics will mostly invoke the corresponding methods of AWT Graphics. Consequently, the rendering performance of CWT-AWT is slightly worse than the original owing to extra wrapping overhead. According to the benchmarking results made by us in [63], the overhead is about 10.3%.

Although CWT-AWT is basically a simple wrapper implementation, some optimizations are still introduced in its implementation. For example, CWT-AWT will avoid unnecessary state changes since some changes incur overhead, including font, color and clipper. Changes of these states only have effects when rendering operations really occur. Therefore, CWT-AWT only applies changes right before the rendering operations.

3.3 CWT-DX

CWT-DX is implemented by using DirectX 3.0 provided by Microsoft Java SDK [31]. Using Microsoft Java SDK is a quick solution that allows us to access DirectX without building a bridge via JNI. Although only old DirectX functionalities are available, since CWT only supports 2D rendering now, DirectX 3.0 already performs with sufficiently high performance.

CWT-DX accesses Windows GDI and DirectDraw to perform basic rendering operations. The functionalities of the Graphics class can be divided into two categories: (1) figures and texts, and (2) images. Each category is realized by different graphics libraries, as described in the following subsections.

3.3.1 Images and Rectangles

The major speedup from accessing DirectDraw to draw images comes from hardware-accelerated memory copy inside video cards. The `drawImage` methods of `Graphics` are implemented by the *bit block transfer* (blit) operation in DirectDraw. Furthermore, DirectDraw also accelerates color filling in a given rectangular area. Therefore, both `fillRect` and `clearRect` methods of `Graphics` are implemented by using the `bltColorFill` method of DirectDraw.

However, using DirectDraw, programmers may encounter the surface lost problem. Surfaces represent the memory on the video cards which are used to store images. The contents of the surfaces could be freed 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 [54]. 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 introduced since J2SE 1.4 [54].

3.3.2 Figures and Texts

Since the DirectX does not provide text and figure rendering APIs, Windows GDI is used instead. When using Window GDI to draw figures and texts 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 [29]. Thus, it is important to minimize the locking times to reduce the performance overhead. In order to minimize the times, CWT does not release the obtained DC handle, until any of the image rendering or rectangle filling methods is invoked. This optimization can be illustrated in Figure 9.

Pseudo-code of Java AWT	Pseudo-code of optimized DirectX and GDI commands (selected=true, alive=true)
`g.drawImage(img1);`	`dds.blt(img1);`
`g.drawImage(img1);`	`dds.blt(img1);`
`g.setColor(c1);`	`hdc = dds.getDC();`
	`Gdi32.SelectColor(hdc, c1);`
	~~`dds.releaseDC(hdc);`~~
`if(selected)`	~~`hdc = dds.getDC();`~~
`  g.drawOval(...);`	`di32.Ellipse(hdc);`
	~~`dds.releaseDC(hdc);`~~
`if(alive) {`	
`    g.fillRect(...);`	~~`hdc = dds.getDC();`~~
	`Gdi32.Rectangle(hdc);`
	`Gdi32.releaseDC(hdc);`
`    g.drawImage(img2);`	`dds.blt(img2);`
`}`	
`g.setColor(c2);`	~~`hdc = dds.getDC();`~~
	~~`Gdi32.SelectColor(hdc, c2);`~~
	~~`dds.releaseDC(hdc);`~~
`g.fillRect(...);`	`dds.colorFill(c2);`

Figure 9. Eliminating unnecessary getting and releasing DC.

3.3.3 Optimization of CWT-DX

For high rendering performance, game programmers usually put pre-rendered images and textures into video memory of 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 off-screen images will stay at the system memory. Practically, programmers may need to put frequently used off-screen 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.

For most game applications, programmers want to control all the details for high rendering performance. In order to give programmers such flexibilities, CWT allows programmers to decide the memory location when creating images, and allow them to copy images between the video memory and system memory. CWT-DX 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-DX. Using `Toolkit`, programmers can get DirectDraw objects for advanced operations. For more details, please refer to Appendix C.

DirectDraw supports high performance for 2D image rendering, but it still relies on Windows GDI to render texts and figures. Game programmers usually use pre-rendered text and figures, saved as images, to solve this problem. Although DirectX does not support figures drawing, it can draw rectangles (including horizontal and vertical lines) by filling colors into them, as J2SE 1.4 does [54]. With hardware acceleration, these operations perform better than rendering non-rectangular shapes.

3.4 CWT-GL

In this subsection, we briefly introduce how CWT-GL implements the Graphics class using JOGL. We divide the functionalities of the Graphics class into five parts: figures, images, texts, off-screen buffers and graphics states, whose design issues and strategies are described in Subsections 3.4.2 to 3.4.6, respectively.

3.4.1 Introduction to JOGL

JOGL [50] is an open-sourced project initiated by the Game Technology Group at Sun Microsystems Inc. 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) [51] and will possibly be included in the Java SE core library in the future.

3.4.2 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 as small number of lines or triangles 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].

3.4.3 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 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 [2]. 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 [24] (or called texture packing [65]), which groups small images into a single power-of-two texture to utilize memory. This technique is commonly used in game applications [65]. Note that the problem of optimizing texture packing can be reduced to the two-dimensional Knapsack problem, which is known to be NP-hard [9].

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.

3.4.4 Texts

Since text drawing is not directly supported in OpenGL, two alternatives are used in OpenGL applications: image-based and geometry-based approaches [24]. 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 shape the texts are of, the more polygons and processing power are needed. For example, Asian languages, such as Chinese, typically require more polygons to emulate.

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* (LFU) 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 [24]. The text engine generates glyphs for each character and also caches them in display lists for later uses. 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. Like the way how we optimize circle drawing described in Subsection 3.4.2, 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 Subsection 4.5.2, we set one megabyte 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 cache texts are drawn in white color with black

background. Then, we enable the blending function to blend the designated color[2] before drawing the texts. The blending function is also specified so that the white color of the cached texts will be drawn by designated color and the black background will become transparent.[3] 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.

3.4.5 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) [36] and Framebuffer Object (FBO) [34]. Both have been implemented in CWT-GL, since both have advantages over each other, as described as follows.

- **Pbuffer.** The pbuffer technique [36] is an OpenGL extension. Pbuffers allow programmers to create hardware-accelerated off-screen buffers. This method is faster than old ways when doing off-screen rendering, such as `glReadPixels()`, `glDrawPixels()` and `glCopyTexSubImage2D()`, which involves copying pixels between *video memory*[4] and *system memory*[5] [34]. 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

[2] The designated color is specified by using the `glBlendColor()` function.
[3] 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}.
[4] Video memory refers to the memory on the video cards which hold data for display devices.
[5] System memory refers to the memory where a computer holds current programs and data which CPU works with.

OpenGL context switching, which takes extra time [34]. 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 [34]. Despite of these disadvantages, the pbuffer technique is supported by more graphics cards, since it was introduced earlier than FBO.

- **FBO.** The FBO [34] 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 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 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.

3.4.6 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 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 account.

First, most CWT components are of lightweight. 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) [8] 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 [6].

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 10 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.

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

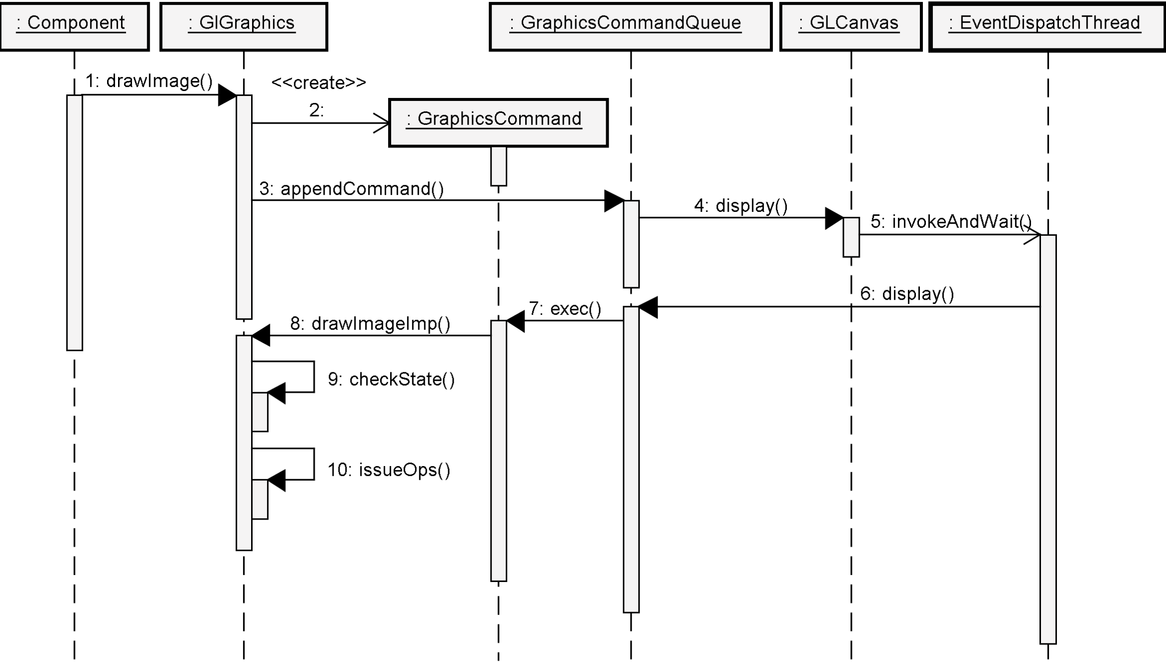


Figure 10. Sequence diagram of STR-like design in CWT.

3.4.7 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.

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 11. Eliminating unnecessary changes of OpenGL state.

In optimization method (2), we try to minimize the number of state changes in OpenGL. As described in Subsection 3.4.5, 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 11 shows three cases of unnecessary changes of OpenGL state (pseudo-code with strikeout) when drawing an image continuously: (a) glBegin()/glEnd(), (b) glBindTexture(), and (c) glEnable()/glDisable(). In order to achieve this, CWT-GL uses variables to indicate 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 required ones. Therefore, unnecessary state changes can be avoided.

3.5 Mixing CWT-GL with JOGL

CWT can be seamlessly mixed with JOGL without the problems of unexpected rendering performance and visual effects when mixing Java AWT/Swing components with these 3D libraries, mentioned in Subsection 1.2, since the CWT-GL implementation uses JOGL as its internal render. The problems of rendering performance and visual effects are solved by the shared buffer design between CWT-GL and JOGL programs, as discussed in Subsection 5.1.3.

In order to mix CWT-GL with JOGL, several issues have to be considered, including shared view buffer, rendering order, and maintaining OpenGL states. First of all, in order to seamlessly mix CWT-GL with JOGL, a shared view port is designed. CWT-GL initiates a

GLCanvas object provided by JOGL and renders everything on the canvas, which can be shared by other JOGL programs. To do so, CWT-GL let programmers access the GLCanvas object so that the programmers can render 3D content on the canvas.

Once CWT-GL and the JOGL programs render on the same canvas, the next issue is to organize the rendering order. Since 3D programs may use CWT to design interactive user interfaces and *head-up display* (HUD), such as menu and chatting box, CWT widgets should be rendered after the JOGL programs rendering scenes. Therefore, CWT defines the rendering order to that the JOGL programs render 3D scenes first and then CWT renders atop the scenes, as shown in Figure 12.

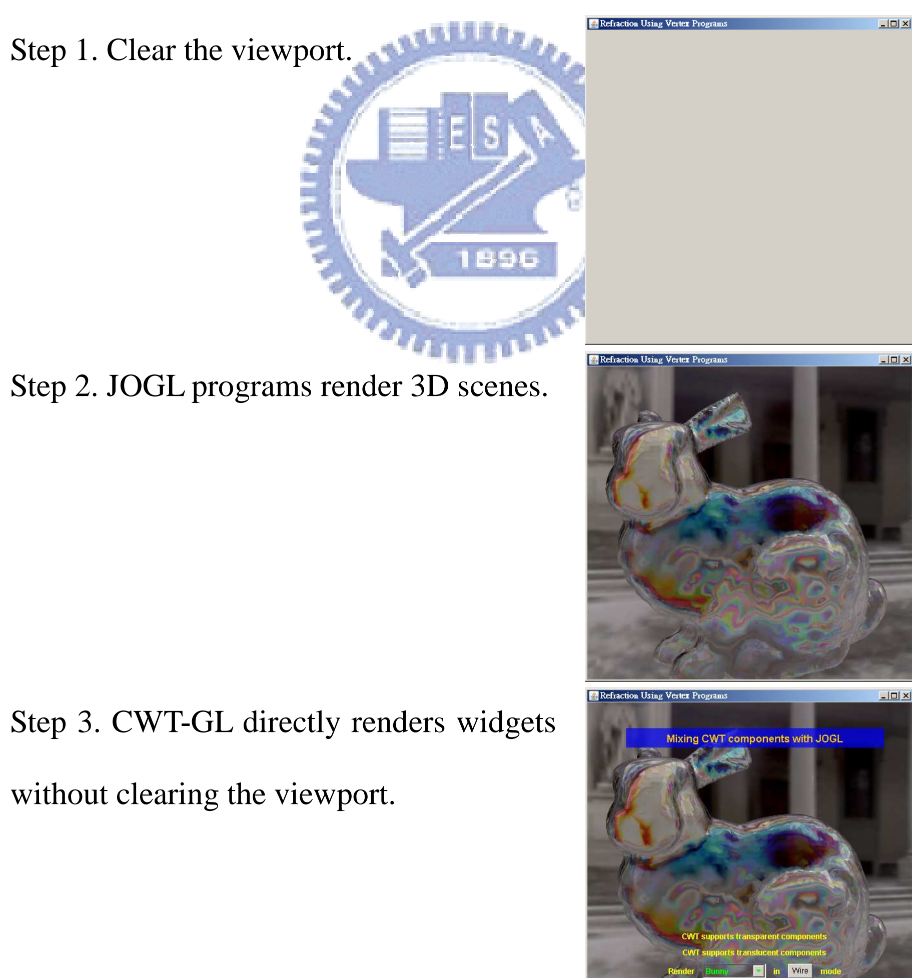


Figure 12. The flow of mixing CWT-GL with JOGL.

Table 4. OpenGL states for CWT-GL.

State	CWT-GL	Description
Projection	Orthographic	CWT-GL uses orthographic projection since CWT is a 2D library which only requires a 2D orthographic viewing region.
Viewport	(0, 0, width, height)	The upper left corner of the viewport is located at (0, 0) so that the coordination system is the same as Java AWT/Swing.
Camera	(0, 0, 1, 0, 0, 0, 0, 1, 0)	In CWT-GL, the camera is placed at location (0, 0, 1) and looks at location (0, 0, 0) with up direction (0, 1, 0).
Lighting	Disabled	CWT-GL does not need lighting.
Depth test	Disabled	Depth test has to be disabled so that widgets will be rendered atop 3D scenes.
Clipper	Enabled	CWT-GL uses clipping planes of OpenGL to implement the clipper of Java AWT/Swing.
Modelview and texture matrices	Identity matrix	CWT-GL uses identity matrix for modelview and texture matrices, which means that no transformation is required.

The final issue is to maintain OpenGL states before CWT-GL and the JOGL programs start to render. OpenGL is a state machine which controls the rendering behaviors of operations. Therefore, the OpenGL states have to be set correctly. The required OpenGL states of CWT-GL are listed in Table 4. Before CWT-GL starts to render, the OpenGL states will be stored and be changed as described in Table 4. After CWT-GL finishes the rendering, the OpenGL state will be restored so that CWT-GL will not affect the rendering behaviors of the JOGL programs.

3.6 Related Work

Some research, such as *Agile2D* [26], focuses on building OpenGL adapters to Java AWT/Swing, while others, such as *FengGUI* [40], and *Minueto* [10] try to create toolkits with different APIs. In this subsection, we review the work related to CWT.

3.6.1 Agile2D

Agile2D [26] implements an almost complete set of Java 2D functionalities based on GL4Java to replace the repaint manager of Swing, as shown in Figure 13. 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 [68], ArcadePod.com [16] and CYC games [13][59]. Second, Agile2D is based on GL4Java which only supports OpenGL version 1.4 and has no plan for evolution. Third, Agile2D does not support rendering of off-screen buffers. Finally, Agile2D only supports the first 256 characters in Unicode, e.g. ISO 8859-1. These issues can limit the applications of Agile2D.

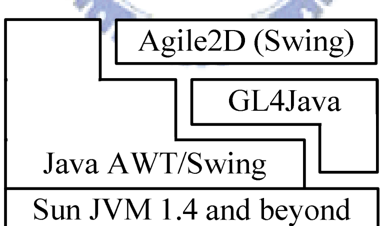


Figure 13. Agile2D architecture.

3.6.2 FengGUI

FengGUI [40] 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. As shown in Figure 14, FengGUI provides a new set of commonly used widgets and graphics API with easy-to-use design, which are

different from Java AWT/Swing components. In addition, FengGUI can also be combined with several 3D game engines, including jMonkey Engine [20], jPCT [21], and Xith3D [67]. Programmers can also directly access JOGL or LWJGL, since FengGUI does not encapsulate these two APIs. However, using FengGUI, programmers need to learn not only the new API, but JOGL or LWJGL, which may reduce the programmers' productivity. In addition, FengGUI supports only JRE 1.5 and beyond, which also limits possible Web users.

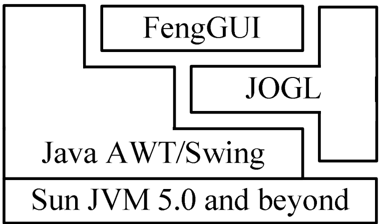


Figure 14. FengGUI architecture.

3.6.3 Minueto

Minueto [10] 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 dissertation.

Chapter 4 Experiments

In order to evaluate the consistency of rendering performance of a Java program running in possible combinations of toolkit, JREs, graphics APIs, system properties, and OSs, we implemented two testing programs as our benchmarks, available on the website [15]. 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 rendering environment (RE). In the remaining part of this section, we will briefly show our experiments.

4.1 Test Programs

We implemented two test programs, available on our website [15]. 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, described as follows.

- Image tests, as shown in Figure 15 (A), are 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.

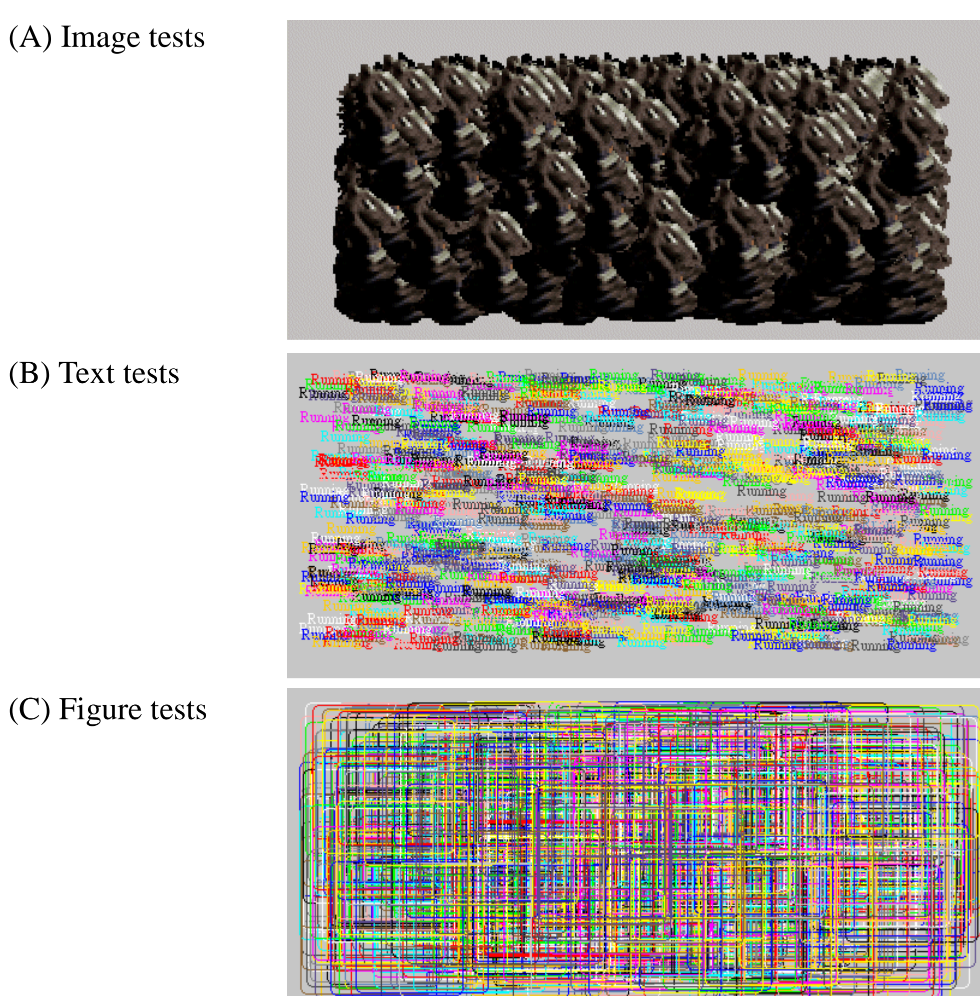


Figure 15. Screenshots of the micro-benchmarks.

Figure 16. A screenshot of the Bomberman game.

- Text tests, as shown in Figure 15 (B), have two subtests: simple texts (using the word “Running”) and articles (consisting of about 13,000 characters on the screen, including 1,562 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.
- Figure tests, as shown in Figure 15 (C), include 12 subtests which draw lines, polylines, polygons, rectangles, round 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 [59], as shown in Figure 16. 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 20000 frames.

Both 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 graphics APIs, we also measure the rendering performance of different combinations of graphics APIs, as shown in Table 5, four APIs for creating back buffers and three APIs for creating runtime images for dynamic processing. In order to simplify the names of the graphics APIs, we abbreviate these sets of APIs in the remaining dissertation. 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 [3][58], some system properties allow programmers to customize how Java 2D performs rendering operations. Therefore, we also specified these system properties when running our benchmarks and selected the most significant parts which influenced rendering performance most, as shown in Table 6.

Table 5. Graphics APIs Tested in the Benchmarks.

Usage	API	ID	JDK
Back Buffers	`Component.createImage(w, h)`	Img	1.0 ~
	`Component.createVolatileImage(w, h)`	Vlt	1.4 ~
	`GraphicsConfiguration.createCompatibleImage(w, h)`	Cpt	
	`GraphicsConfiguration.createCompatibleVolatileImage(w, h)`	CptVlt	
Runtime Images	`Toolkit.createImage(imageProducer)`	Img	1.0 ~
	`GraphicsConfiguration.createCompatibleImage(w, h, trans)`	Cpt	1.4 ~
	`GraphicsConfiguration.createCompatibleVolatileImage(w, h, trans)`	CptVlt	1.5 ~

Table 6. System properties for SystemProperty∈{Special} [3][58].

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 7. System hardware, configuration and OSs.

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

Table 8. JRE versions in the benchmarks.

Java Version	OS		
	Windows XP and Vista	Fedora Core	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

4.2 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 3.

In order to make fair comparison, we used two computers with roughly equivalent computing power to install the four OSs, as shown in Table 7. 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 128MB 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 8. 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 website [50]. The release build we used to run the benchmarks in this dissertation was JSR-231 1.1.1-rc8.

4.3 Rendering Environments (REs)

In this dissertation, we ran test programs in all the *rendering environments* (RE) with the combination of using different JREs, graphics 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, GraphicsAPI, SystemProperty, OS).

- *Toolkit*∈{AWT, CWT-DX, CWT-GL}. "AWT" represents the Java AWT graphics library, "CWT-DX" represents the DirectX implementation of CWT, and "CWT-GL" represents the OpenGL implementation of CWT in this dissertation.
- *JRE*∈{MSVM, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6}. "MSVM" denotes Microsoft Java VM [31], and "1.1" to "1.6" denote Sun JRE version 1.1 to 1.6, respectively.
- *GraphicsAPI*∈*BackBufferAPI*×*RuntimeImageAPI*, where *BackBufferAPI*={Img, Vlt, Cpt, CptVlt} and *RuntimeImageAPI*={Img, Cpt, CptVlt}, as shown in Table 5.

Therefore, there are in total 12 combinations tested in this dissertation.

- *SystemProperty*∈{None, OpenGL, 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 6, which follows the hints in [3][58].
- *OS*∈{XP, Vista, Fedora, MacOS}. "XP," "Vista," "Fedora" and "MacOS" respectively represent the following operating systems, Windows XP, Windows Vista, Fedora Core 6 and Mac OS X 10.4.11.

For example, RE(AWT, 1.6, Vlt+Cpt, OpenGL, 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; RE(CWT-GL, 1.4, None, Img+Img, MacOS) refers to the RE that uses CWT-GL, runs in JRE version 1.4, uses Java 1.0/1.1 graphics 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, RE(AWT, *, Vlt+Cpt, None, MacOS) means all the REs with AWT and with the combinations of GraphicsAPI∈{Vlt+Cpt} and SystemProperty∈{None} on Mac OS X.

4.4 Definitions of Metrics

To measure the rendering performance of the benchmarks, we use three metrics: (1) rendered items per second, (2) frame rate, and (3) Anomaly. Rendered items per second count the number of times an image, text or figure is rendered per second in the micro-benchmark. As for the macro-benchmark, frame rate is commonly employed to measure the rendering speed expressed by frames per second (FPS). For a RE r, FrameRate(r) denotes the frame rate in r. Anomaly for a set of REs, say R, is defined as follows.

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

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.

4.5 Analysis of Micro-Benchmark Results

The micro-benchmark program opens a 600x300-sized window and counts the number of times an image, text or figure is rendered per second. The micro-benchmark is consisting in total 21 subtests. Each subtest was performed in the combinations of using different JREs, graphics APIs, system properties, and OSs. In our analysis, we will focus on the performance inconsistency and eliminate unnecessary details of all the results. Therefore, all

the results shown here are the averaged results of the subtests. The whole results of the micro-benchmark are listed in Appendix A.

4.5.1 Image Tests

The rendering performance of images is quite important in 2D game development, since normally games are formed by images. As shown in Figure 17, CWT achieves high and consistent scores among the four OSs in the image tests due to the use of hardware acceleration. As for AWT, programmers need to use new graphics APIs and system properties, which also get benefits from hardware acceleration, to obtain better results. However, while achieving good results using some combinations of graphics APIs and system properties, the rendering performance is still inconsistent among the four OSs. Besides, using new graphics APIs and system properties loses the compatibility to old JREs, such as Java 1.1, which is still used by a portion of Web users [12]. This makes it hard to decide which combination of graphics APIs and system properties should be used in cross-platform game development.

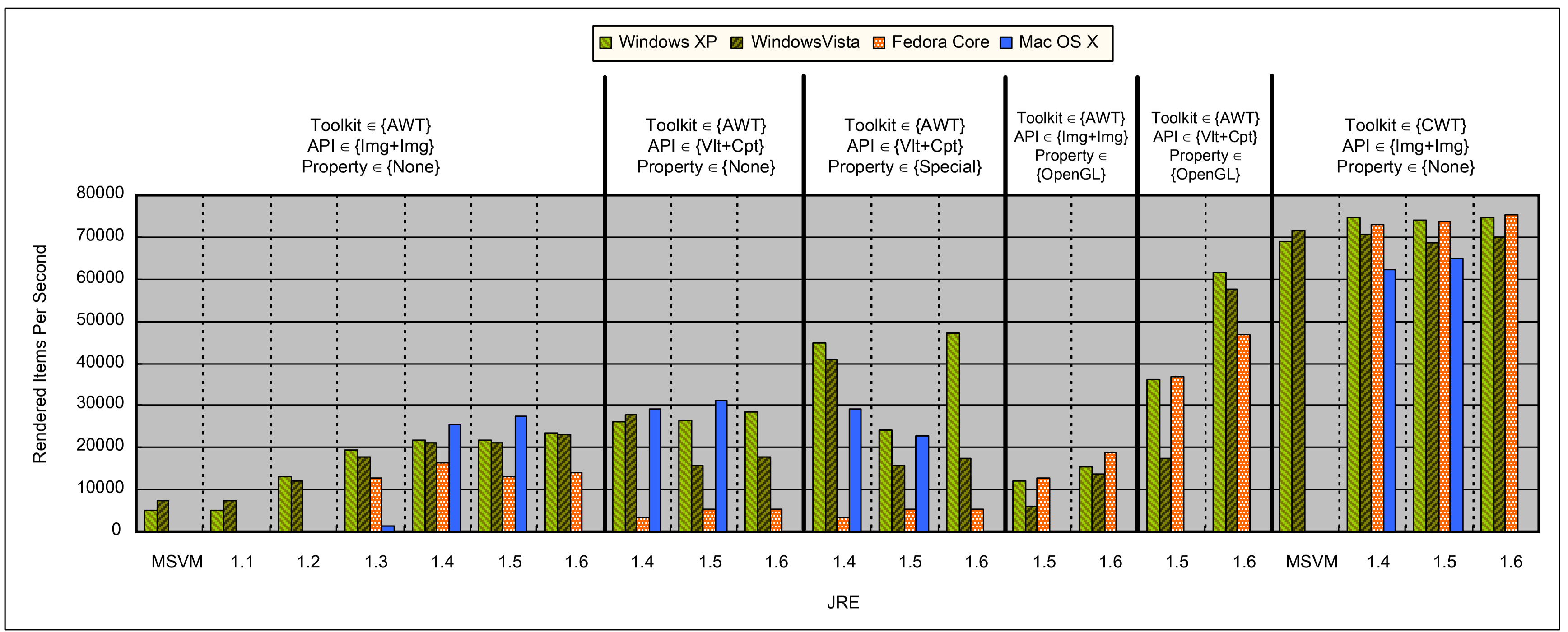


Figure 17. Averaged rendered images per seconds among OSs.

4.5.2 Text Tests

The rendering performance of texts is also important in many network games which support chatting systems. The tests are divided into two parts: fix-sized and various-sized texts. First, for the fix-sized text rendering, as shown in Figure 18, different from the results of the image test, CWT, especially CWT-GL, is not good at rendering texts, since texts are rendered by CPU, not by GPU. Therefore, the hardware acceleration of GPU is not quite helpful in text rendering. For example, when DirectDraw is used, CWT-DX put as many images and off-screen buffers as possible into video memory. However, this makes the rendering performance of texts worse, since CPU has to access the video memory through a relatively slow bus, such as PCI, AGP, and PCIe. When OpenGL is used, there is no direct supports of text rendering. Alternatively, CWT-GL renders the texts based on two approaches: (1) texture-based (firstly render the texts to images and then render the images) and (2) geometry-based approaches (use lines and polygons to form the texts). In this test, the texture-based approach is employed. Without direct hardware support, CWT performs worse than AWT does.

As for AWT, we can also find the situation that hardware-accelerated image rendering may make the rendering performance of texts worse. When the combinations of graphics APIs and system properties achieve higher rendering performance in image rendering, the performance of text rendering may become lower. Another fact that we want to point out is that the text rendering is still inconsistent not only among the four OSs, but also among JREs. Normally, we expect that the rendering performance would be better in newer JREs. However, on Windows XP and Vista, JRE 1.4 performs worse than JRE 1.2 and 1.3 do. Fortunately, the performance of text rendering is much better in JRE 1.5 and 1.6. However, the requirement of proper combinations of graphics APIs, system properties, and JREs may

still makes it hard to optimize the rendering performance of texts for unpredictable Web users' rendering environments.

Next, we analyze the rendering performance of texts with different font sizes on Windows XP. As mentioned in Subsection 3.4.4, CWT-GL implements texture-based and geometry-based text-rendering approaches. In the texture-based approach, the cache size of the texture maps has a great influence on the rendering performance. As shown in Figure 19, when using a 16-megabyte texture map to cache texts, the texture-based approach is faster than geometry-based approach in drawing texts. However, such cache size may not be practical, since some computers only have limited memory. When the cache size is limited to one megabyte, the rendering speed decreases dramatically when drawing texts with size larger than 24, since the number of different characters exceeds the capacity of the cache. In such case, the geometry-based approach delivers better rendering performance.

Next, we analyze the results of Java AWT. As shown in Figure 19, the rendering speed of texts varies much when different graphics APIs are used. When Java 1.0/1.1 graphics APIs are used (Img+Img), AWT often delivers much better rendering performance than the case of using new graphics APIs (Vlt+Cpt) and system properties (Special). This is because that the text rendering is performed by CPU, not by GPU. CPU can directly render the texts to the off-screen buffer located in the system memory when Java 1.0/1.1 graphics APIs are used. In contrast, CPU renders the texts to volatile images located in the video memory when new graphics APIs are used, so the rendering speed decreases dramatically. In order to overcome this problem, a text cache mechanism has been introduced to Java AWT. For example, Java AWT caches the frequently used characters with size 16 and below. Therefore, in such case, the rendering speed of the texts is greatly improved.

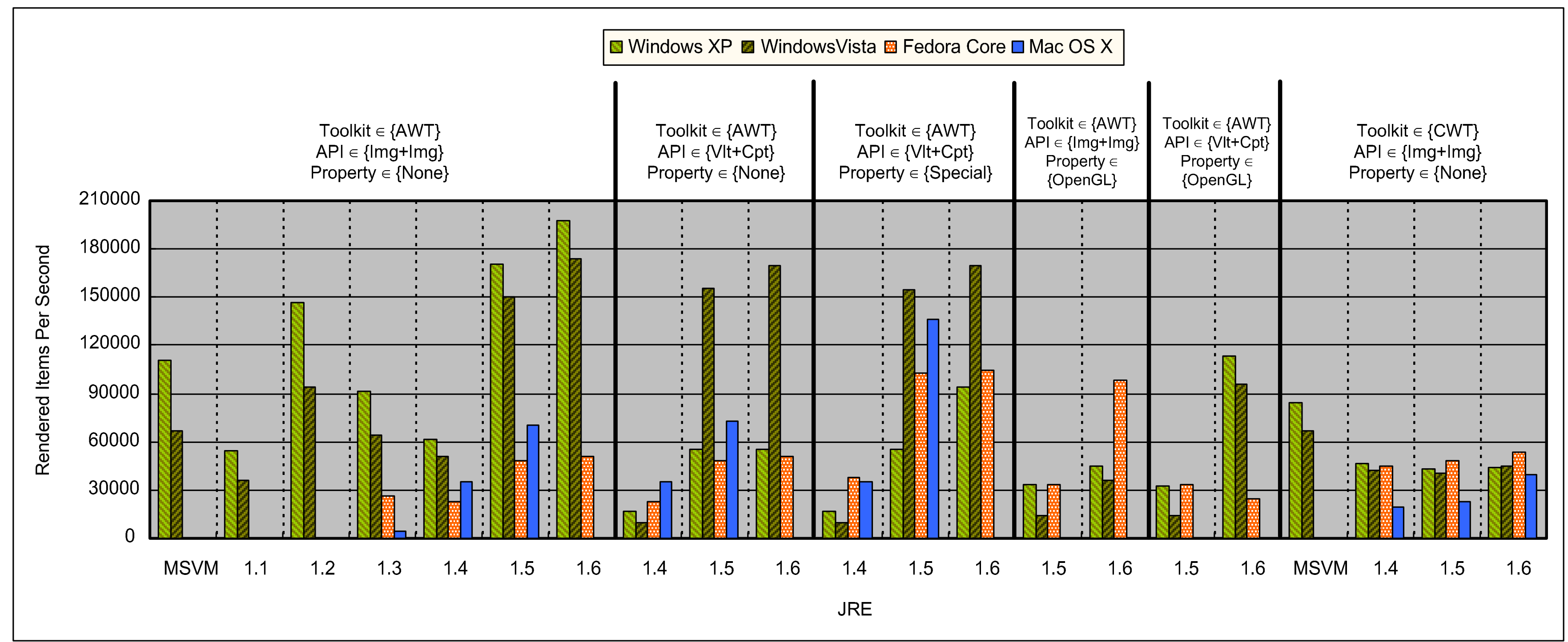


Figure 18. Averaged rendered texts per seconds among OSs.

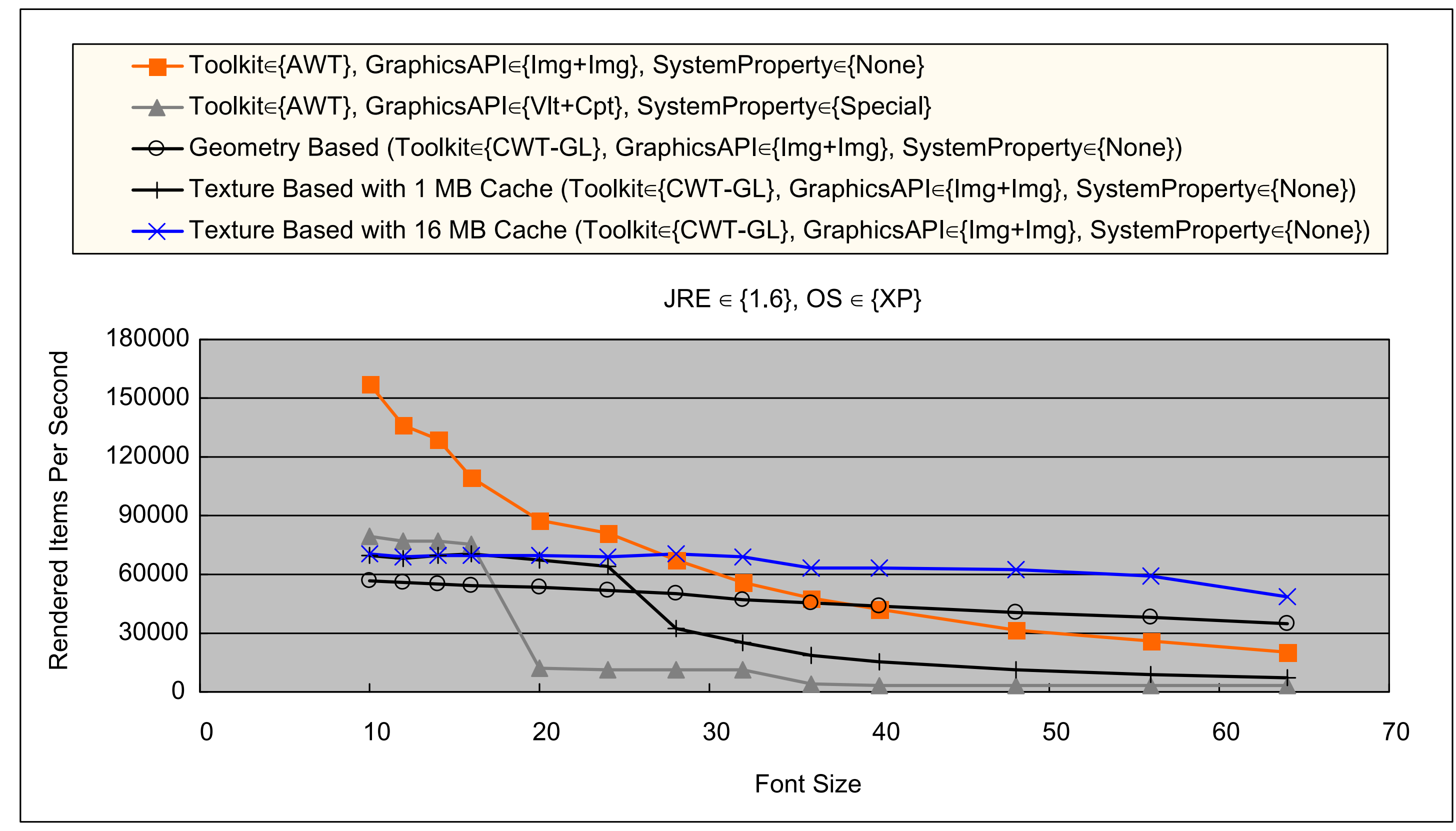


Figure 19. Averaged rendered texts per seconds with different font sizes.

4.5.3 Figure Tests

As shown in Figure 20, CWT-DX does not deliver good results in rendering figures, since these operations are also not hardware-accelerated in DirectDraw. Therefore, in order to render figures onto back buffers, CPU has to access the video memory. On the other hand, CWT-GL performs these operations quite well since the figures are rendered using lines and polygons which have great supports in OpenGL-accelerated graphics cards. Therefore, CWT-GL achieves high and consistent rendering performance on the four OSs.

AWT still delivers inconsistent performance when rendering figures. The impacts of using different graphics APIs and system properties are different from those of image and text tests. For example, in many cases, to achieve high and consistent rendering performance of figures among the four OSs, programmers should use Java 1.0/1.1 graphics APIs (Img+Img). However, the benchmarking program which uses new graphics APIs (Vlt+Cpt) and proper system properties delivers much better rendering performance in JRE 1.5 on Mac OS X. Therefore, it is still hard to find a combination of graphics APIs and system properties which achieves high and consistent rendering performance among the four OSs.

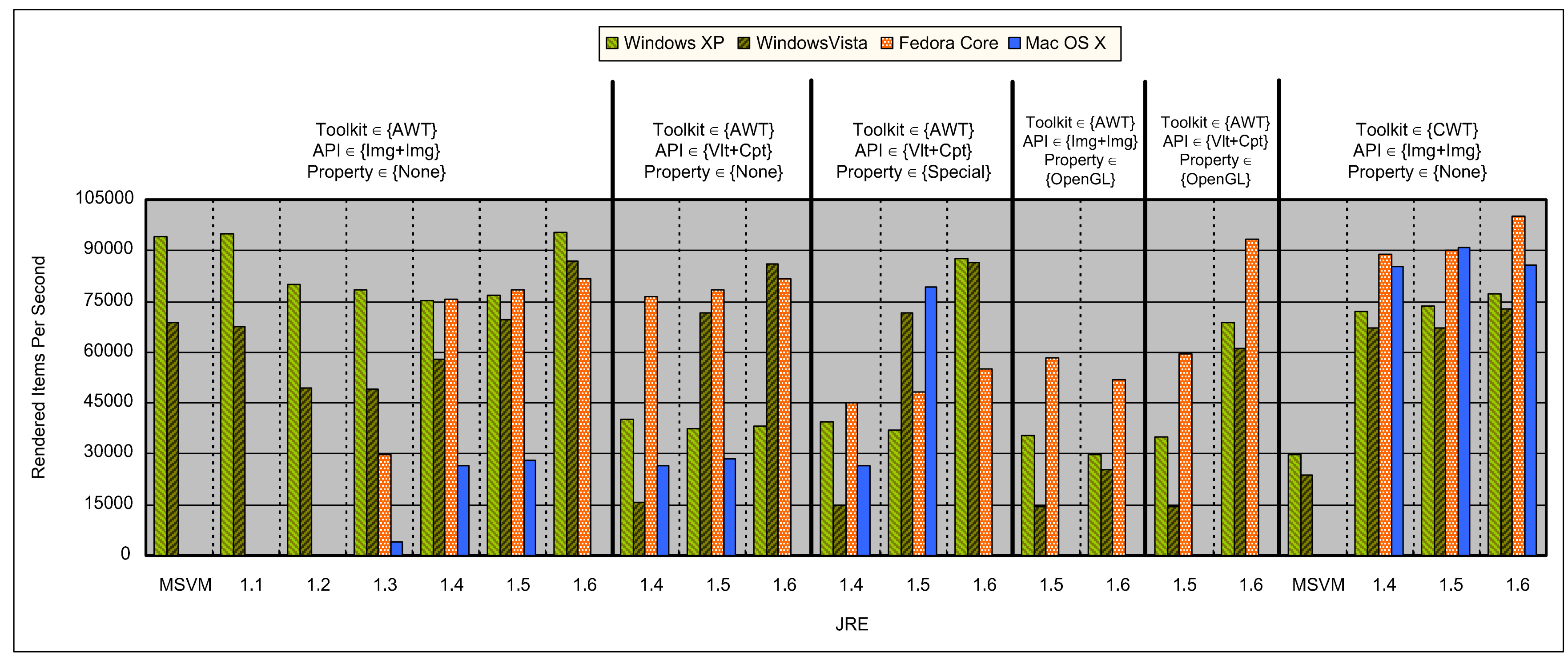


Figure 20. Averaged rendered figures per seconds among OSs.

4.6 Analysis of Macro-Benchmark Results

We analyze the results by three aspects of the results which can represent the performance inconsistency of Java AWT: (1) grouped by OSs, (2) grouped by the combinations of graphics APIs and system properties, and (3) grouped by JREs. The whole results are listed in Appendix B. We summarize the results of 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 graphics APIs are used, i.e. *GraphicsAPI*∈{Img+Img}. As shown in Figure 21, 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*∈{1.3, 1.4, 1.5, 1.6}, *Anomaly*(AWT, *jre*, Img+Img, 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 graphics APIs and system properties available since JRE 1.4, as shown in Figure 22 to Figure 27. In the combinations of *SystemProperty*∈{None, 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 28 and Figure 29, when OpenGL pipeline is enabled (*SystemProperty*∈{OpenGL}), using the combinations of *GraphicsAPI*∈{CptVlt+Cpt, CptVlt+CptVlt, Vlt+Cpt, Vlt+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. Therefore, when the OpenGL pipeline is excluded (that is, *SystemProperty*∈{None, Special}), the inconsistent performance among different OSs exists even when we use new graphics APIs and system properties to tune up the Bomberman game.

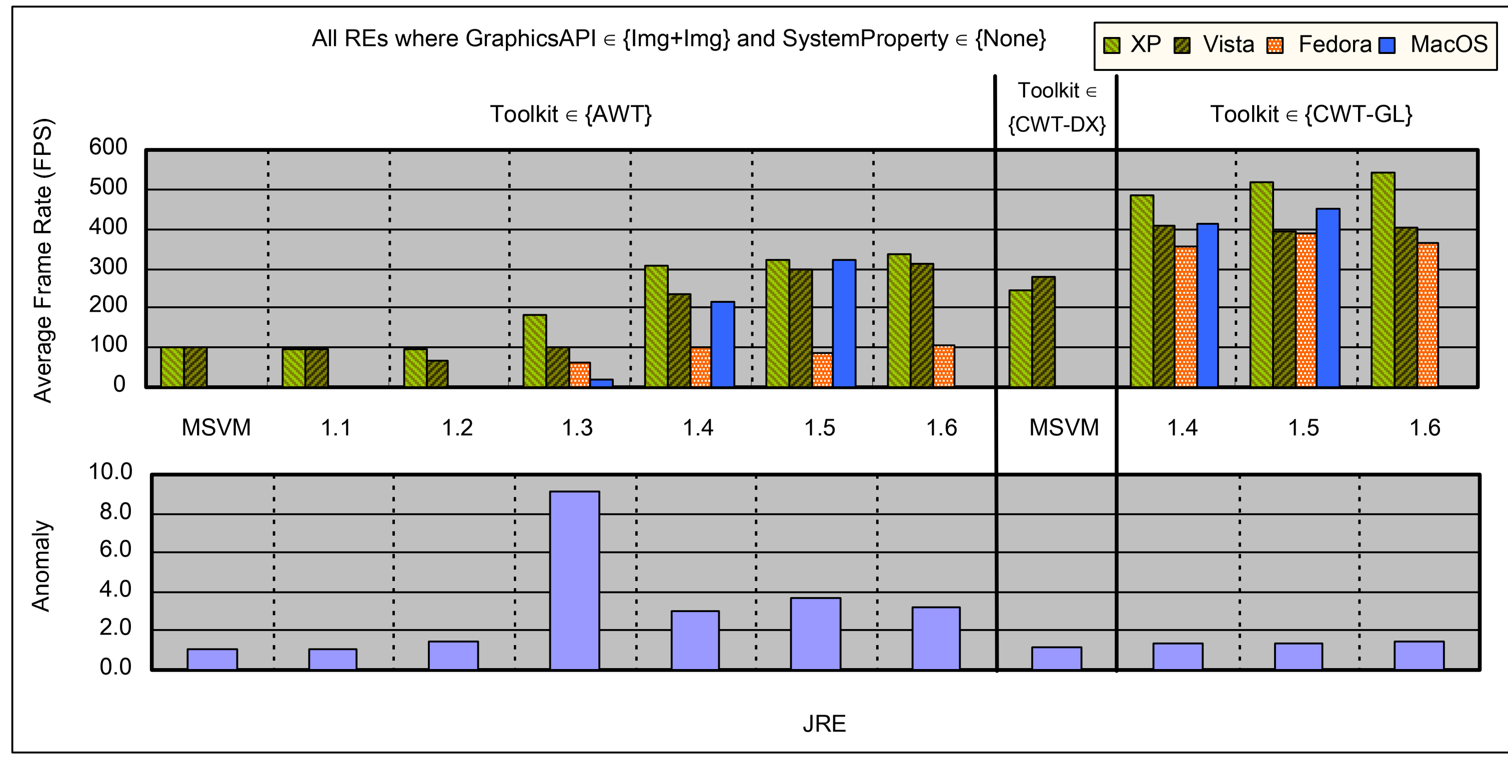


Figure 21. Frame rates and Anomaly among OSs using Java 1.0/1.1 graphics APIs.

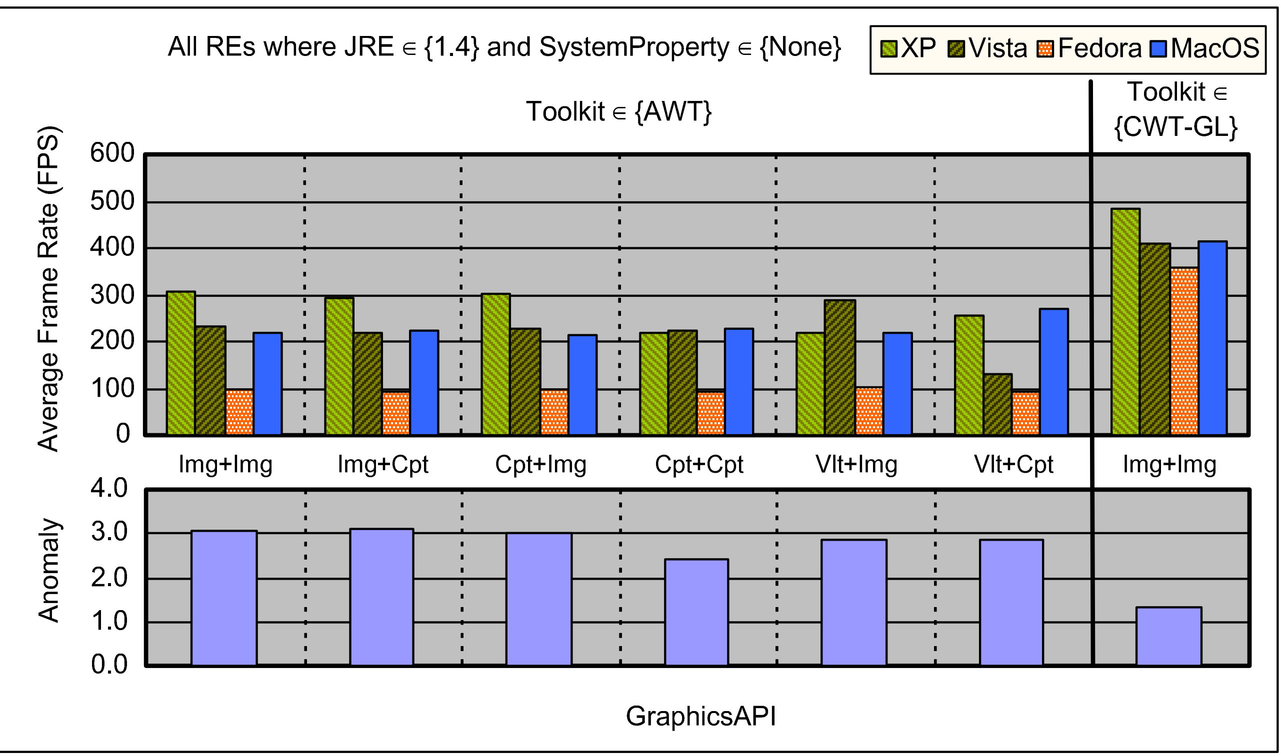


Figure 22. Frame rates and Anomaly among OSs using JRE∈ {1.4} and SystemProperty∈ {None}.

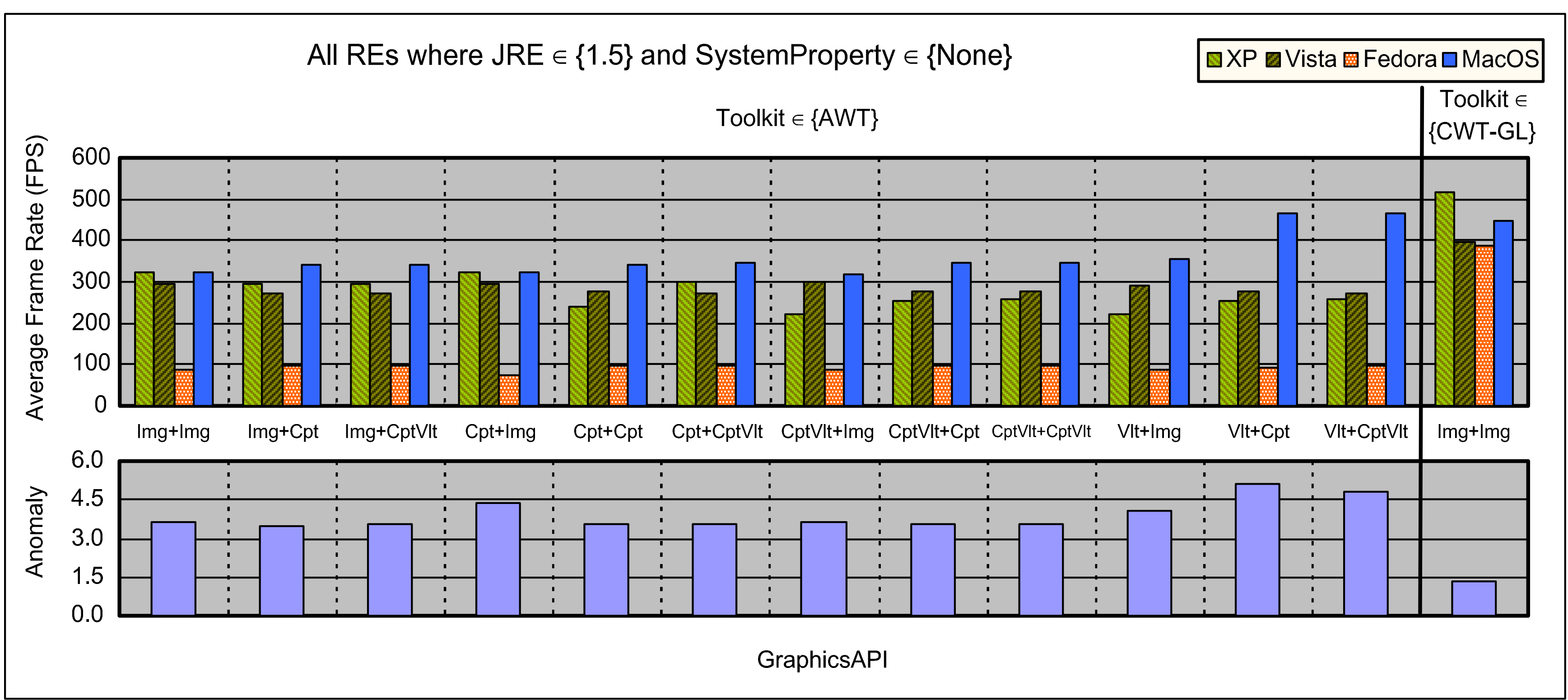


Figure 23. Frame rates and Anomaly among OSs using JRE∈{1.5} and SystemProperty∈{None}.

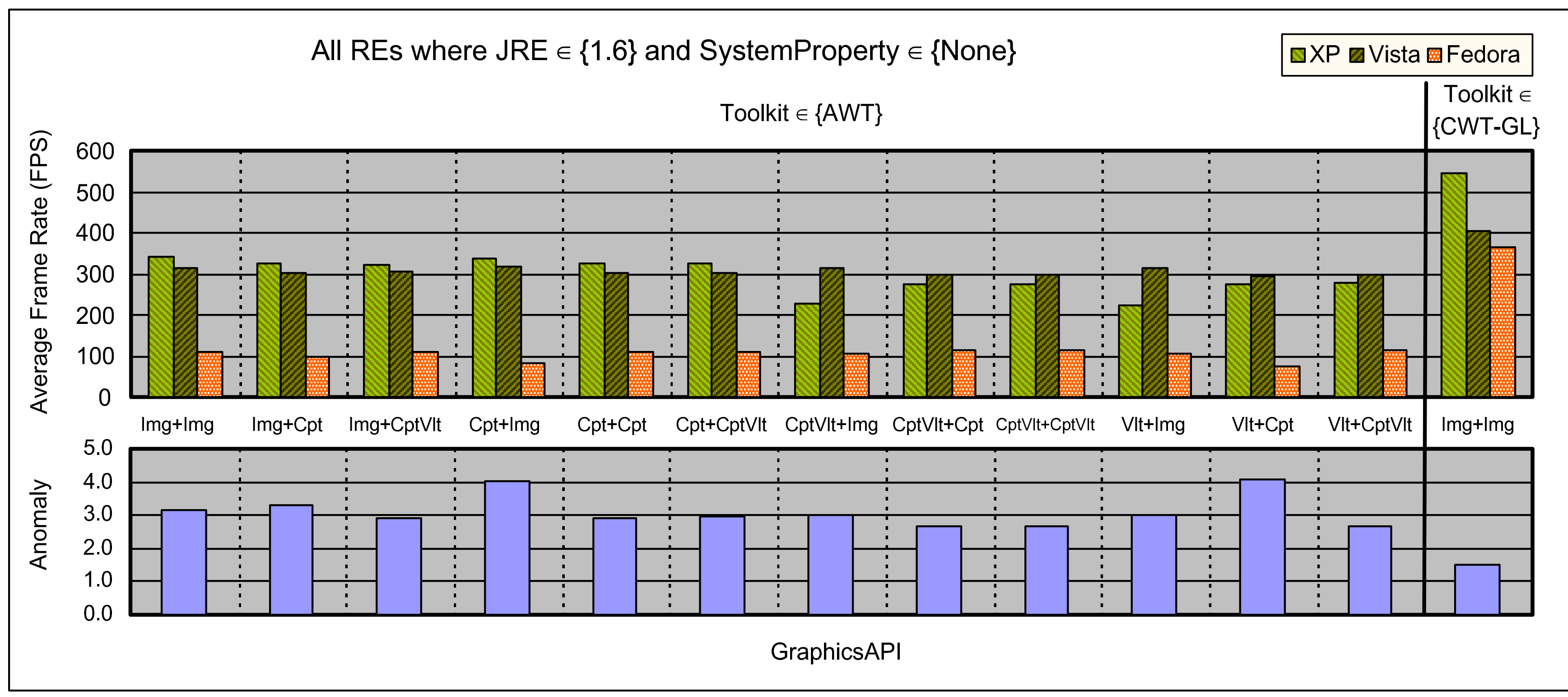


Figure 24. Frame rates and Anomaly among OSs using JRE∈ {1.6} and SystemProperty∈ {None}.

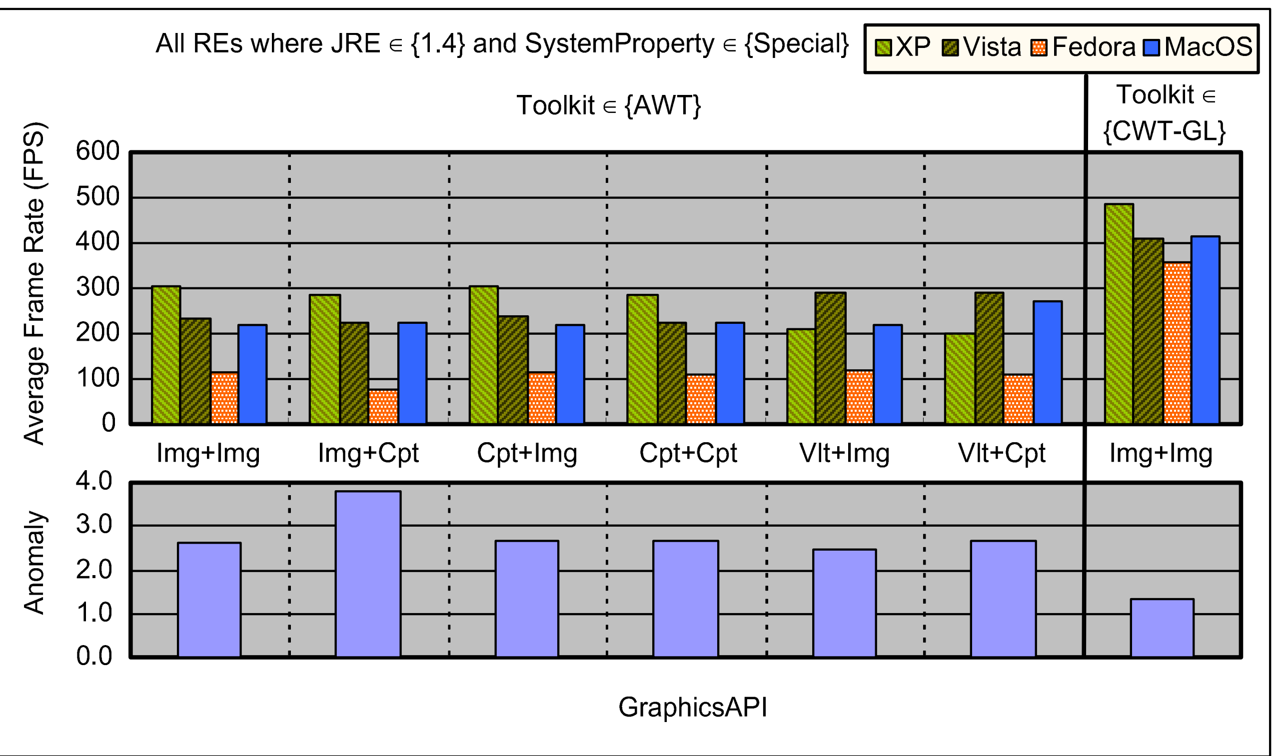


Figure 25. Frame rates and Anomaly among OSs using JRE∈ {1.4} and SystemProperty∈ {Special}.

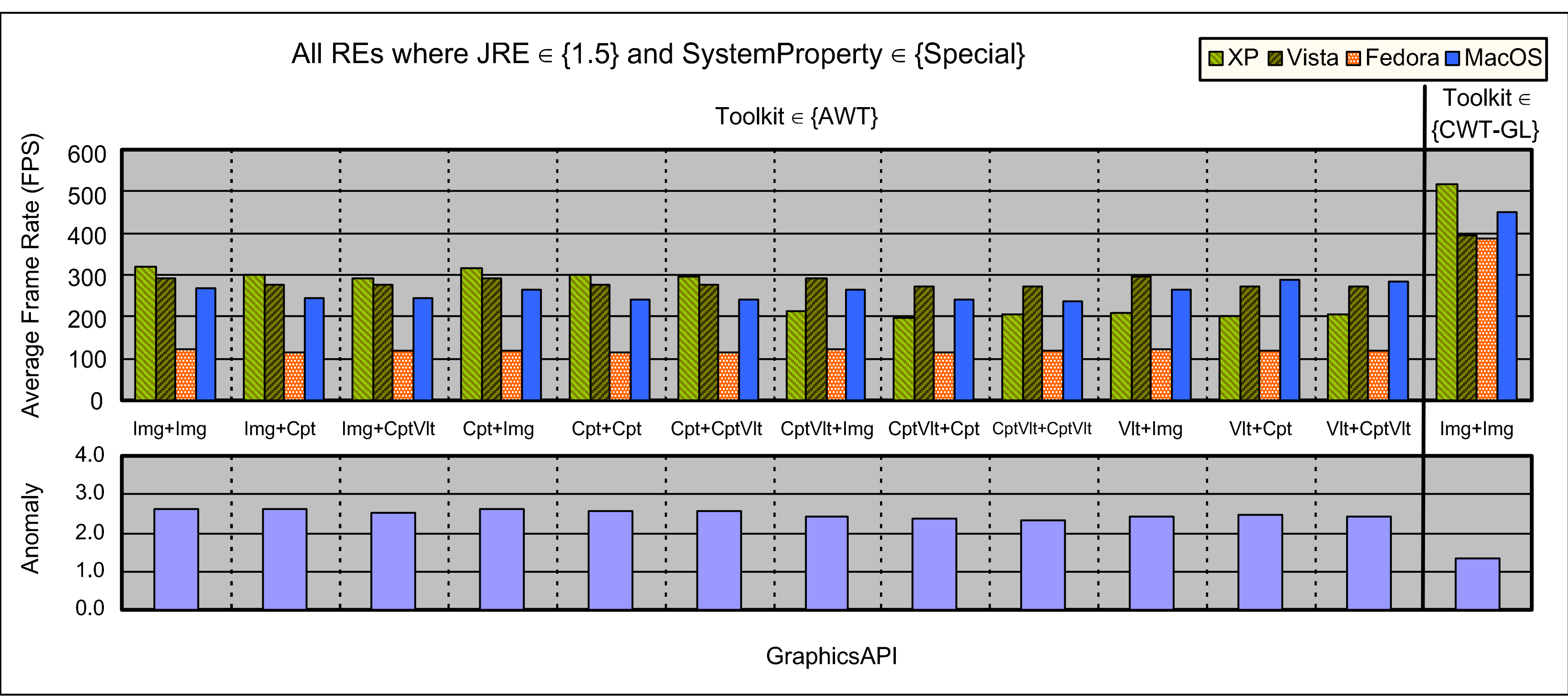


Figure 26. Frame rates and Anomaly among OSs using JRE∈ {1.5} and SystemProperty∈ {Special}.

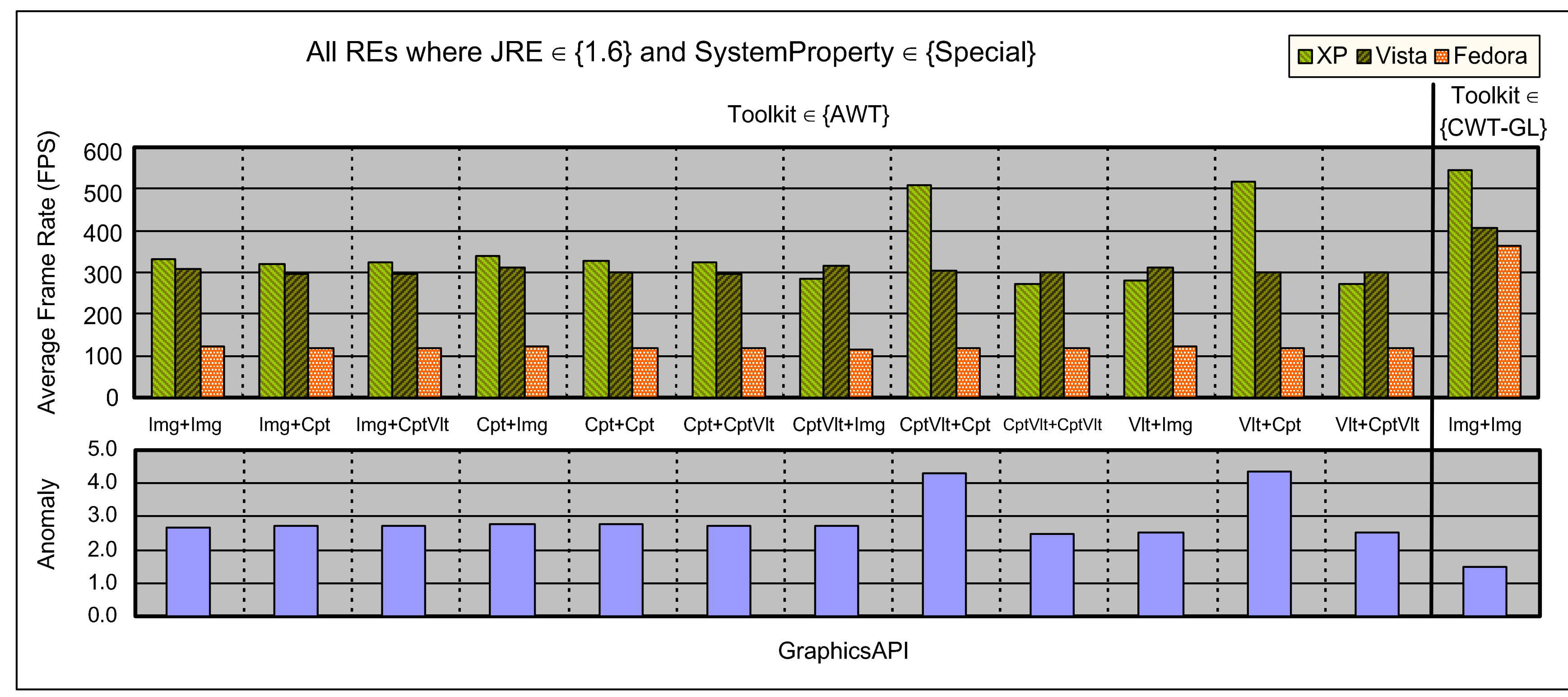


Figure 27. Frame rates and Anomaly among OSs using JRE∈{1.6} and SystemProperty∈{Special}.

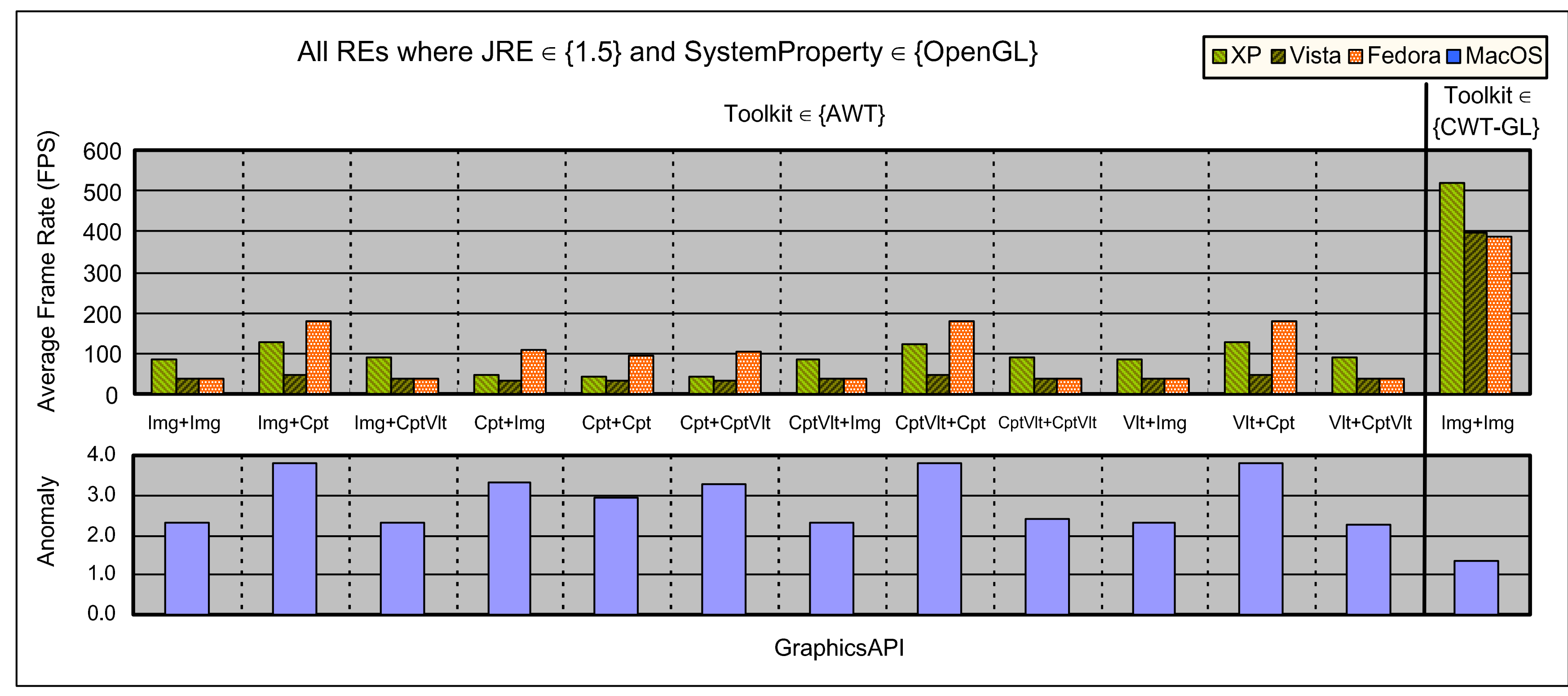


Figure 28. Frame rates and Anomaly among OSs using JRE∈{1.5} and SystemProperty∈{OpenGL}.

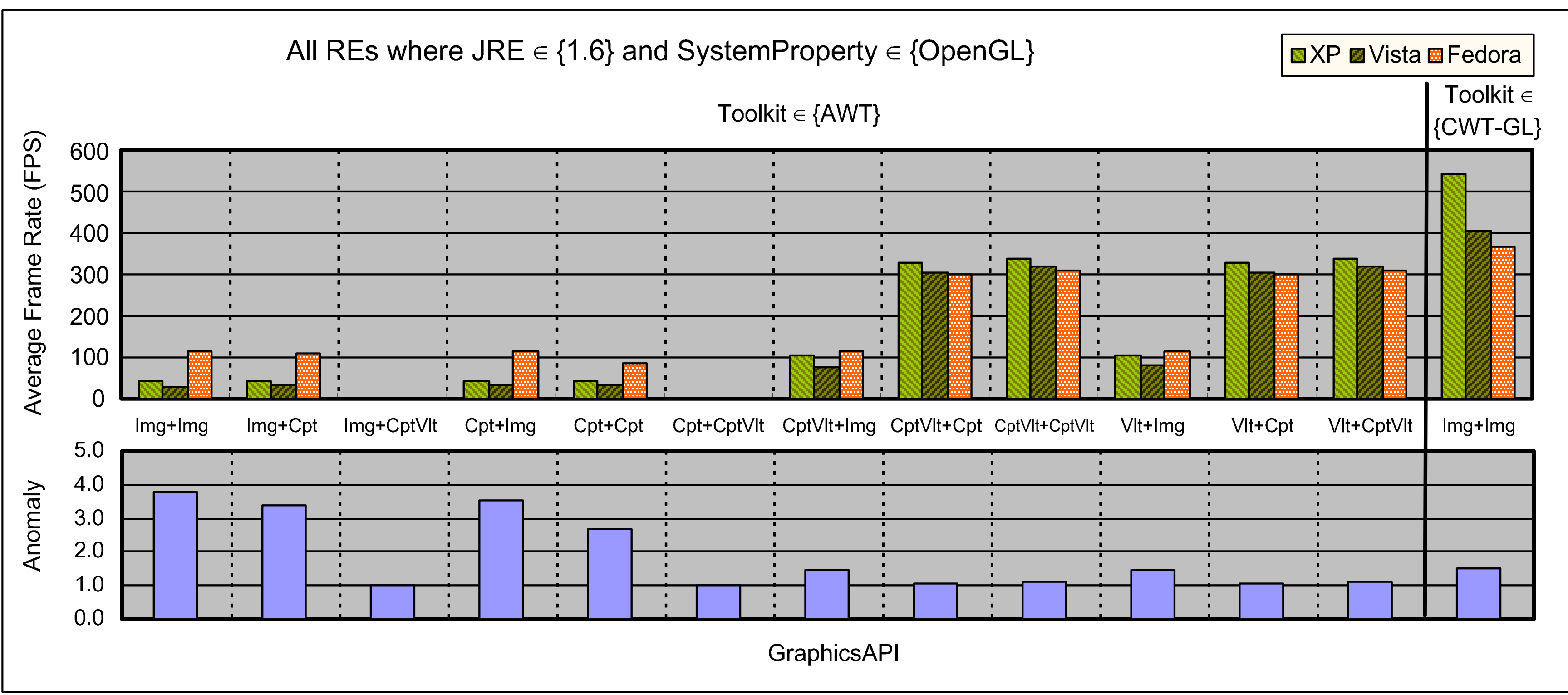


Figure 29. Frame rates and Anomaly among OSs using JRE∈{1.6} and SystemProperty∈{OpenGL}.

- **The rendering performance of using different combinations of graphics APIs and system properties is inconsistent.** Typically, using different graphics APIs results in different rendering performance. As mentioned in Subsections 4.1 and 4.3, Java AWT provides several types of graphics APIs and system properties for tuning up performance. However, we observe that the graphics 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 graphics APIs and system properties in different *REs* to achieve the best rendering performance.

  Table 9 summarizes the best combinations of graphics APIs and system properties for achieving the best frame rates in given *REs*, and Figure 30 to Figure 32 show the results of comparing different combinations of Graphics APIs and system properties. From these results, we do not find a combination of graphics APIs and system properties which 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 *GraphicsAPI*∈{Img+Img, Cpt+Img} and *SystemProperty*∈{None, Special} often achieves the best results. (b) However, using the combinations of *GraphicsAPI* ∈{CptVlt+Cpt, Vlt+CptVlt} and *SystemProperty*∈{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 *GraphicsAPI*∈{Vlt+Cpt} to achieve the best frame rates. (d) Moreover, when OpenGL pipeline enabled (*SystemProperty*∈{OpenGL}), the combinations for the best frame rates are also different from those above. The combinations of *GraphicsAPI*∈{Img+Cpt, CptVlt+Cpt, Vlt+Cpt} achieve the best frame rates in JRE 1.5, while the combinations of *GraphicsAPI*∈{CptVlt+CptVlt, Vlt+CptVlt} achieve the best frame rates in JRE 1.6. (e) Even worse, using wrong combinations of graphics APIs with the OpenGL pipeline

would cause serious consequences. The frame rates may become as low as only 2 FPS. Therefore, the inconsistent rendering performance of the combinations of graphics APIs and system properties makes it hard to decide which combinations should be used in developing cross-platform Java games.

Table 9. Combinations of graphics APIs which deliver the highest frame rates.

JRE	System Property	OS			
		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

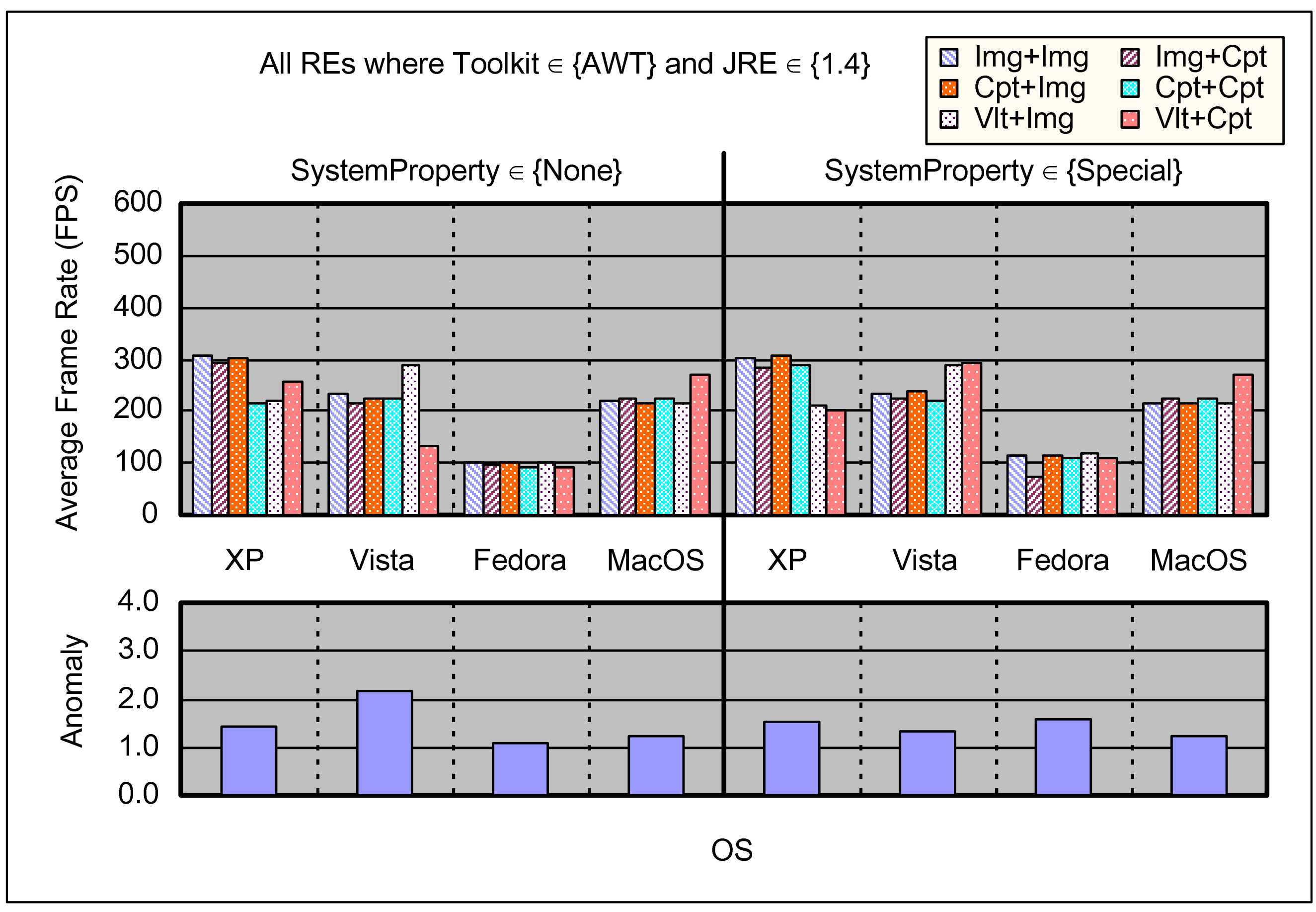


Figure 30. Frame rates and Anomaly on choosing different graphics APIs using JRE∈ {1.4}.

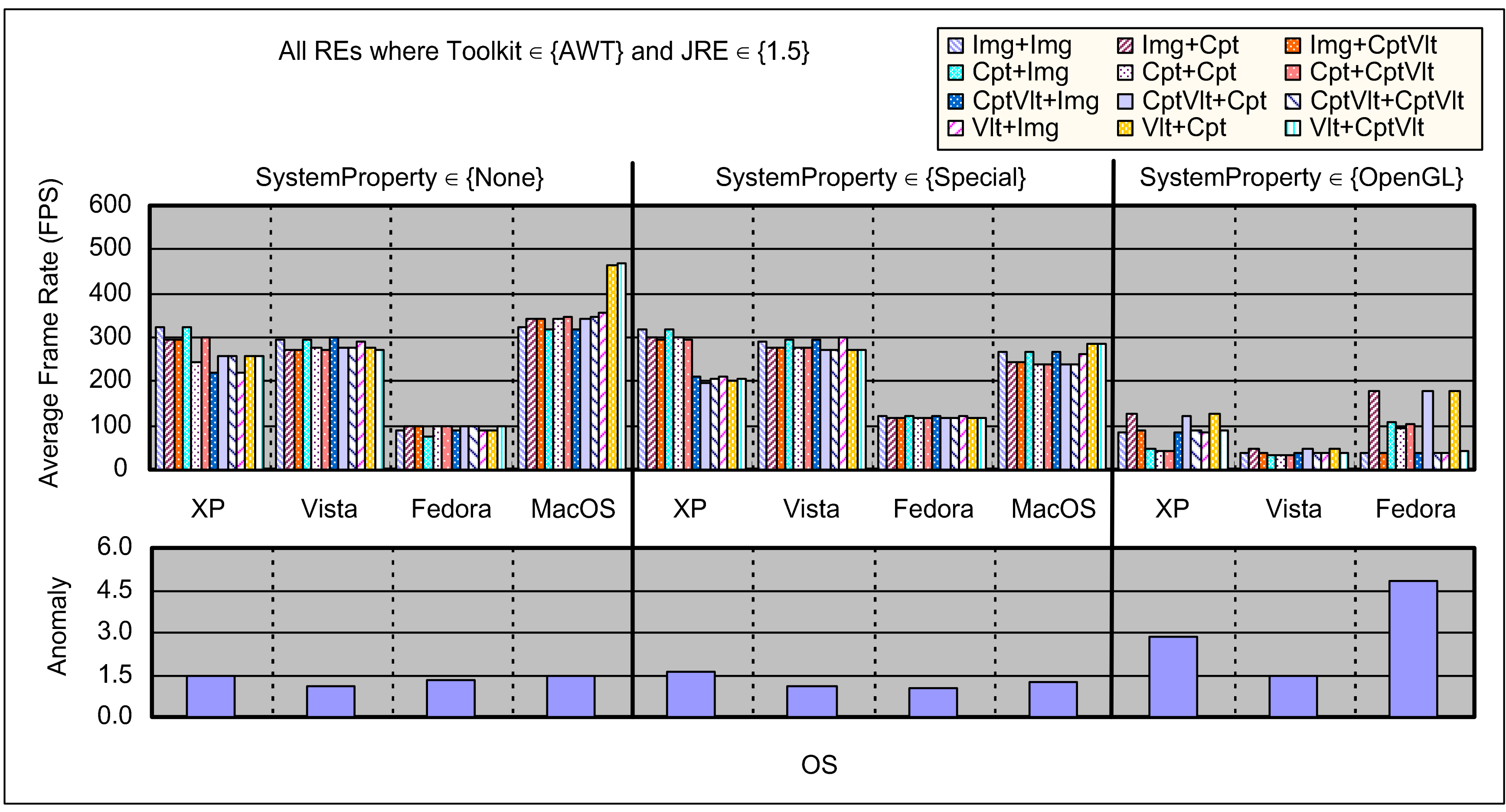


Figure 31. Frame rates and Anomaly on choosing different graphics APIs using JRE∈ {1.5}.

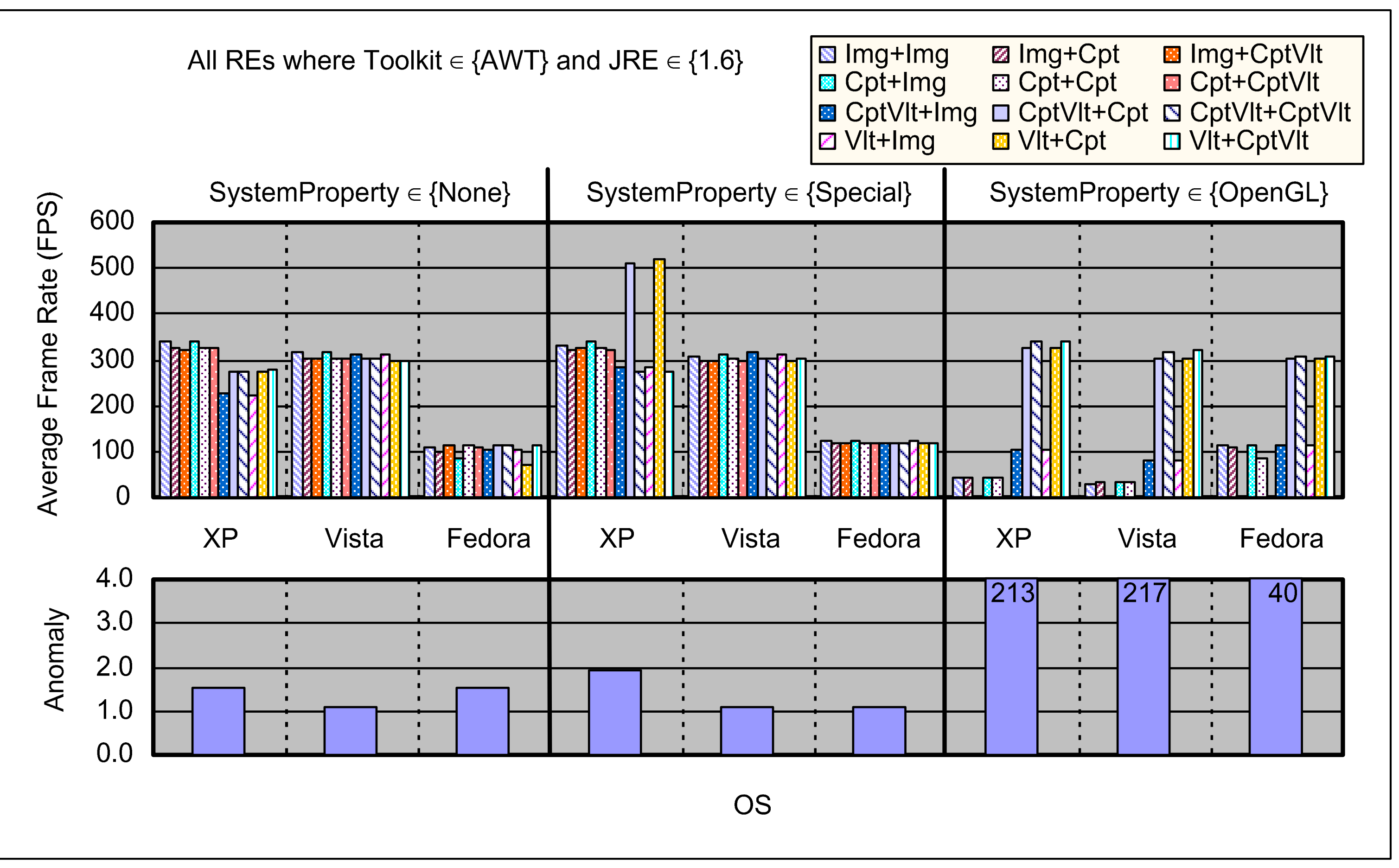


Figure 32. Frame rates and Anomaly on choosing different graphics APIs using JRE∈{1.6}.

- **The rendering performance of Java AWT is inconsistent among commonly used JREs.** As mentioned in Subsection 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 account.

  As shown in Figure 33, older JREs normally delivered worse frame rates, especially MSVM. In the case of using Java 1.0/1.1 graphics APIs, for all $os \in$ {XP, Vista, Fedora, MacOS}, *Anomaly*(AWT, *, Img+Img, None, *os*) ranges from 1.18 to 3.31.

  When new graphics APIs are used, as shown in Figure 34 and Figure 35, the performance inconsistency among JREs also exists. For example, programmers may tune their programs to achieve very high frame rates 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*(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.

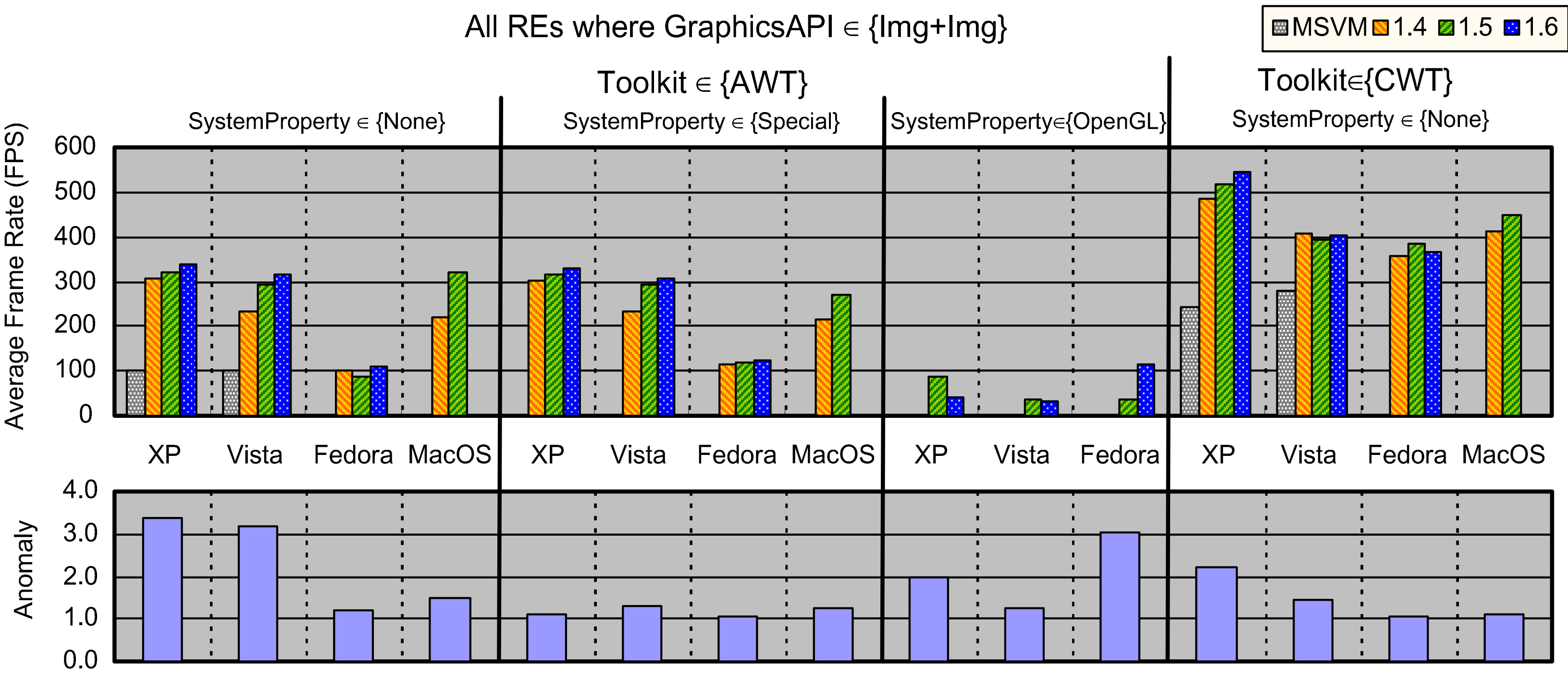


Figure 33. Frame rates and Anomaly in commonly used JREs using Java 1.0/1.1 graphics APIs.

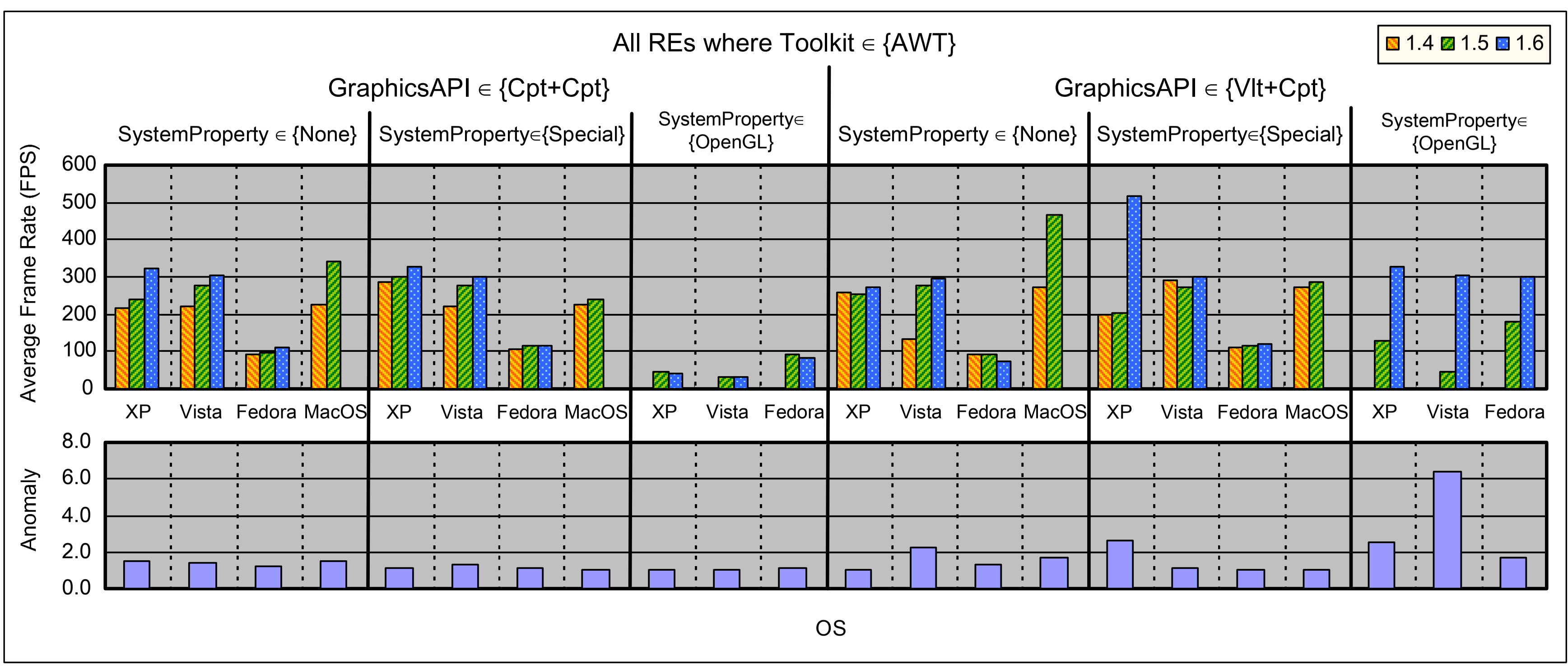


Figure 34. Frame rates and Anomaly in commonly used JREs using graphics APIs introduced since J2SE 1.4.

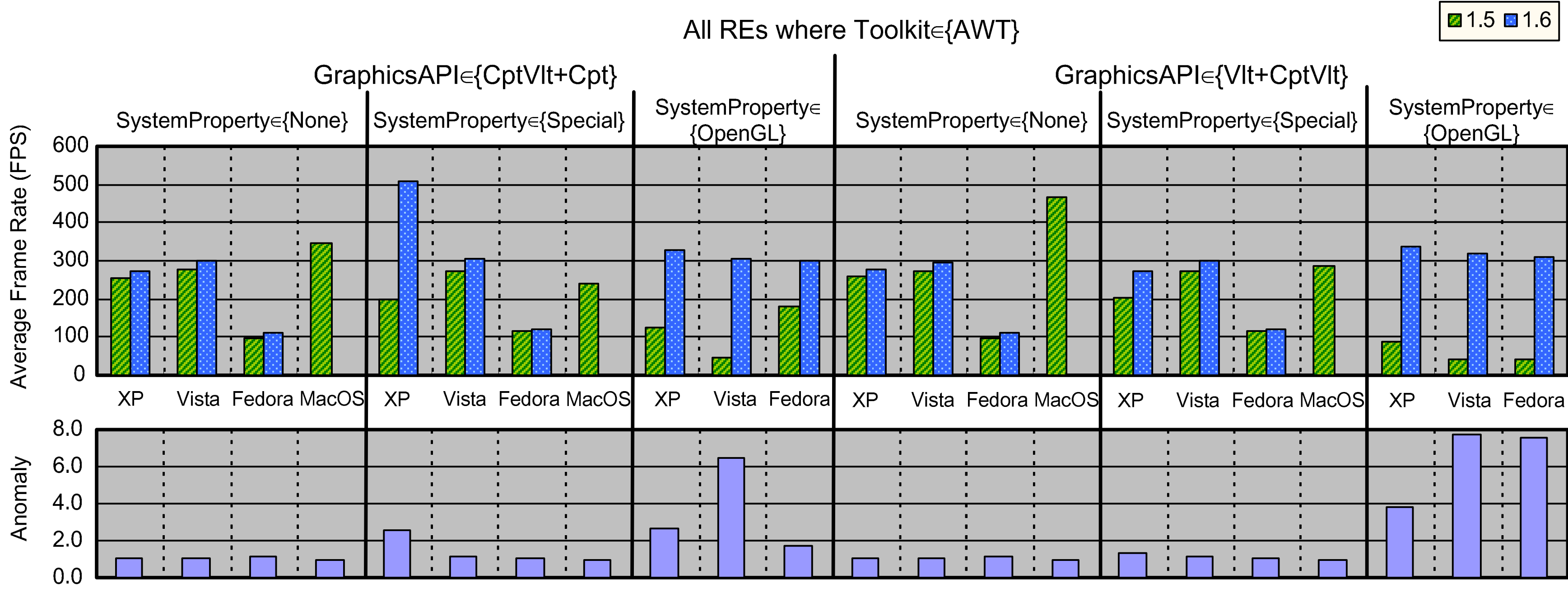


Figure 35. Frame rates and Anomaly in commonly used JREs using graphics APIs introduced since Java 5.0.

To sum up the results of Java AWT, we find it hard to optimize the rendering performance in the combinations of JREs, graphics 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 graphics APIs and system properties. However, according to our experimental results, we find no combinations which 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 REs. Thus, the 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.

Next, we summarize the results of CWT as follows.

- **CWT-GL achieves higher and more consistent rendering performance among the four OSs than Java AWT does.** In Figure 21 to Figure 29, 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*(CWT-GL, *jre*, Img+Img, 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 graphics APIs and system properties.** CWT-GL supports Java 1.0/1.1 graphics 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 graphics APIs are supported, CWT-GL still achieves almost the highest rendering performance when compared with Java AWT which has a number of graphics APIs and system properties to tune the rendering performance up.

- **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 33, CWT achieves the best frame rates in the set of *JRE*∈{MSVM, 1.4, 1.5, 1.6}. Meanwhile, CWT also delivers more consistent frame rates than Java AWT does. For example, for all *os*∈{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 REs than those in Java AWT. This is quite important especially when games run in users' computers with various REs. Furthermore, the graphics APIs and system properties in CWT-GL are simpler than Java AWT, which also helps reduce the testing efforts. Therefore, our experimental results suggest that CWT-GL is more suitable for cross-platform Java game development than Java AWT.

Chapter 5 Discussion

Although the *Write-Once-Run-Anywhere* (WORA) feature of Java 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 chapter, we further discuss the problems of current Java 2D rendering pipelines on the four OSs: Windows XP, Windows Vista, Mac OS X, and Fedora Core. Then, we give suggestion for making Java a better platform for developing cross-platform games. Finally, the limitations of CWT are presented.

5.1 Supporting Graphics Systems on Multiple Platforms

This subsection discusses the problems of different graphics systems on the four OSs and corresponding implementations of Java 2D rendering pipelines. These include Window graphics device interface (GDI) and DirectX on Microsoft Windows platforms, Desktop Window Manager (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 introduced some changes to the Java 2D rendering pipelines [44]. 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 average 20% more frames per second. However, the frame rates were still about two to three times slower than those on other three OSs, which is due to the lack of full hardware acceleration on the graphics system. This is an example that low rendering performance may occur when Java games run in the different rendering pipelines on different OSs.

On Mac OS X, Apple Inc. 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 [58]. Properly configuring the behaviors of Quartz may improve the rendering performance, since some rendering operations are anti-aliased [3].

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 28 shows that Sun's 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 using different graphics APIs. This shows inconsistent rendering performance of the OpenGL pipeline when Java games use improper graphics APIs. In contrast, CWT-GL achieves high and consistent rendering performance on all of the four OSs by direct access to OpenGL via JOGL.

Indeed, it is not an easy task to design a cross-platform graphics library with high and

consistent rendering performance on multiple platforms with various graphics systems mentioned above. For cross-platform part, Sun creates two cross-platform graphics libraries for Java: AWT and Swing, which encapsulate the differences and complexity of underlying graphics systems. Java AWT uses native widgets supported by OSs so that the look-and-feels of the OSs are kept. Java Swing adopts lightweight components which are rendered by Java so that the look-and-feel of Swing programs can be changed. Both AWT and Swing provide standard widgets on all supported OSs, which makes Java GUI highly portable.

For the rendering performance part, Sun currently has several DirectX and OpenGL pipelines to accelerate the rendering of Java AWT/Swing, as shown in Table 2 and Table 10. However, the rendering pipelines are tightly bound to specific Java versions and OSs, since they are not ported back to old Java versions and may not be supported on all OSs. This approach causes more serious the performance inconsistency of Java AWT/Swing among Java versions and OSs. Consequently, the rendering performance of Java GUI is not portable, which makes it hard to create cross-platform Java games that require high rendering performance.

Table 10. OS support of Sun's rendering pipelines.

JRE Version	Windows	Linux	Mac OS 10.4.x	Mac OS 10.5.2
1.4	DirectX	N/A	N/A	N/A
5.0	DirectX OpenGL	OpenGL	N/A	N/A
6	DirectX Improved OpenGL	Improved OpenGL	N/A	Improved OpenGL
6u10	Improved DirectX Improved OpenGL	Improved OpenGL	N/A	N/A

According to the analysis above, we propose that the design of Java AWT/Swing should follow the Open-Closed Principle (OCP) [25], which states that "software entities should be open for extension, but closed for modification." We analyze the adoption of OCP to Java AWT/Swing by three aspects in the following subsections: (1) encapsulation and extension, (2) decoupling, and (3) reuse.

5.1.1 Encapsulation and Extension

In order to provide platform independence, Java core libraries have encapsulated platform-dependent features. For example, Java AWT/Swing has been developed by hiding the differences among the graphics systems on Windows, Mac OS, Linux, and Solaris. Such encapsulation makes Java highly platform-independent and portable. The encapsulation also helps performance improvement without changes made on APIs. For example, Sun introduce DirectX and OpenGL pipelines to Java AWT/Swing which greatly improve its rendering performance. Since accessing DirectX and OpenGL is completely hided, Java AWT/Swing programs are benefited with few or no modifications.

However, the encapsulation may not meet future requirements. In game industry, since video cards evolve quickly, high-profile game producers have to keep moving on the trend and using the new features to create games with better visual quality and performance. Therefore, the capability of extension should be considered in the design of graphics API. For example, OpenGL specifies a way to extend its functions for vendors. The capability of extension is an important key to achieve OCP.

Currently, as stated in problem 7 in Subsection 1.2, Java AWT/Swing does not allow direct access to internal DirectX and OpenGL objects, which is not open for extension. Since DirectX and OpenGL evolve along with video cards, new versions are released almost every one to two years. Java AWT/Swing will not get benefits from the new features

unless the implementations of the rendering pipelines adopt them. Java game programmers also lose the ability to access the latest video card features, fine tune the rendering performance, and change the behaviors of the rendering pipelines, such as writing shader code in Java AWT/Swing and making translucent components.

As discussed above, we suggest that Java AWT/Swing should provide not only a platform-independent API but also a way to access internal DirectX and OpenGL objects for game programmers who need to access up-to-date video card features or change the rendering behaviors. Such approach maintains the platform independency for normal users while giving flexibility for advanced users. For example, as shown in Figure 36, MSVM provides both Java AWT/Swing and DirectX APIs, which gives programmers the ability to fine tune rendering performance.

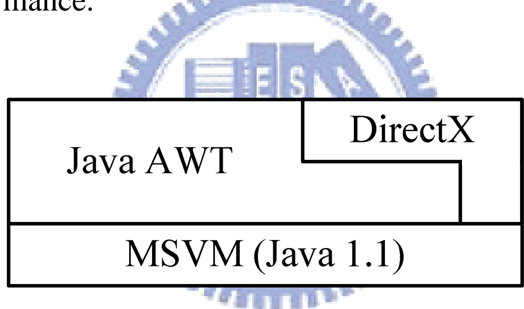


Figure 36. Java AWT/Swing and DirectX in MSVM.

5.1.2 Decoupling

The current approach by Sun to improve the rendering pipelines is to bundle the rendering pipelines with specific JVM versions and platforms, as shown in Table 10 and Figure 37. However, this approach also incurs the following three problems.

(1) Since the rendering pipelines are bundled with new JREs and are not ported back to old JREs, users need to upgrade to one of the new JREs.

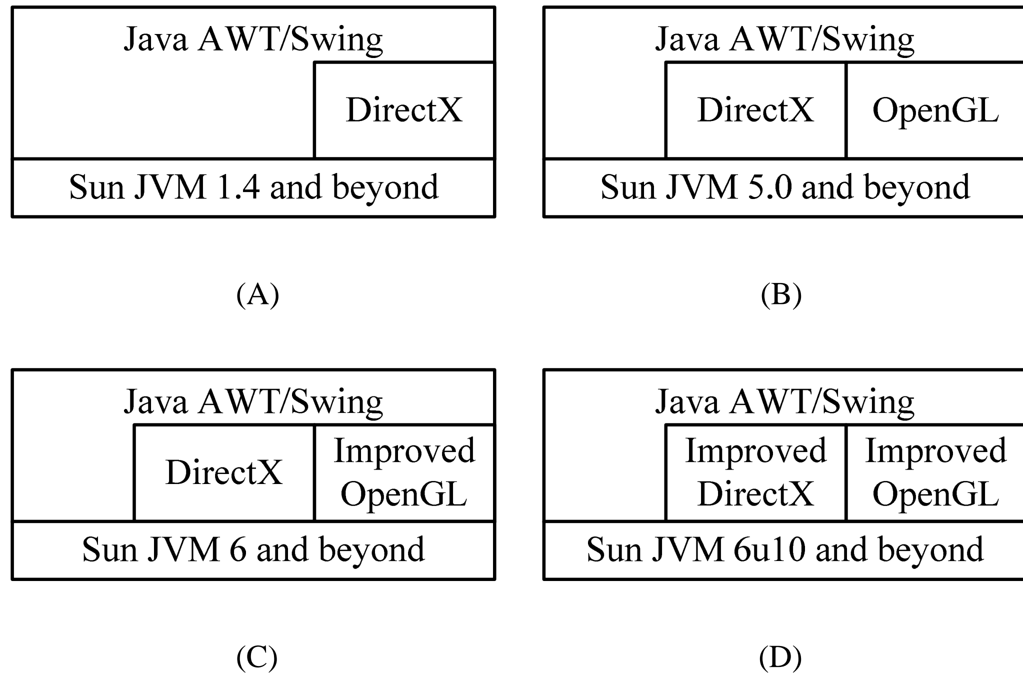


Figure 37. Hardware-accelerated rendering pipelines supported in specific JVM versions.

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

(3) Programmers must wait for newer JREs to improve the reliability, performance or more support of the rendering pipelines. For example, future JREs are required to solve the following two problems: that 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.

According to the discussion above, we suggest that the rendering pipelines 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 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. A good example is that JOGL is decoupled from JRE, as shown in Figure 38. With the design of decoupling, JOGL supports Sun JVM 1.4 and beyond. JOGL is also easier to upgrade, since its download size is only 1 MB, while JRE 1.4 and beyond require more than 15 MB for download.

To sum up, the benefits of decoupling Java AWT/Swing 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 can focus on other design issues.

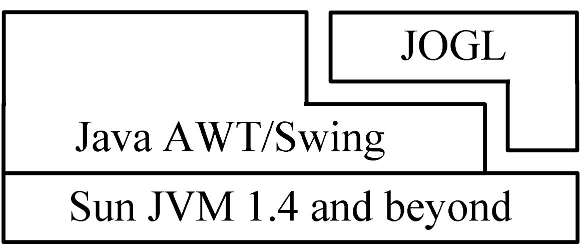


Figure 38. Relation between JOGL and JVM.

5.1.3 Reuse

Over the past years, Sun has designed at least three products which have similar functions of accessing DirectX or OpenGL, include Java AWT/Swing, Java 3D, and JOGL. Several DirectX and OpenGL bindings written by JNI have been created in these products. As a result, Sun created two DirectX bindings and three OpenGL bindings, as shown in Figure 39. These bindings incur several problems as follows.

(1) Since different teams created several bindings with similar functions, the developing time and cost are higher.

(2) Since these duplicated bindings have to be maintained, the overall maintainability is decreased.

(3) Since the products use different bindings, more efforts are required to make these products work together. For example, as shown in Figure 40, it is hard to mix Java AWT/Swing OpenGL pipeline (2D rendering) with JOGL (3D rendering) until Sun resolve the interoperability in Java SE 6. Before Java SE 6, mixing Java AWT/Swing and JOGL causes performance degrade and flickering due to the synchronization between the individual buffers of the two graphics systems. It is also hard to support translucent widgets.

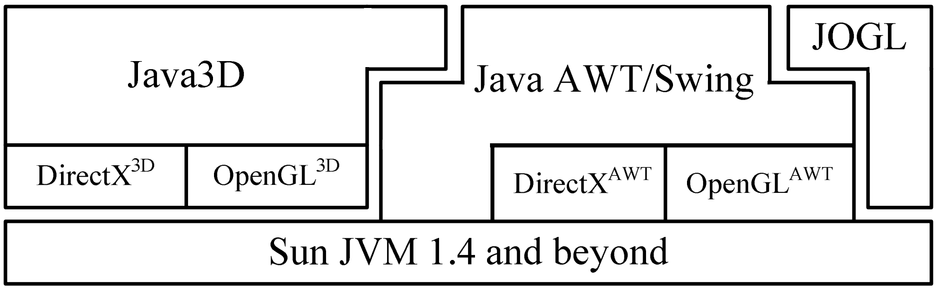


Figure 39. Two DirectX bindings and three OpenGL bindings by Sun Microsystems.

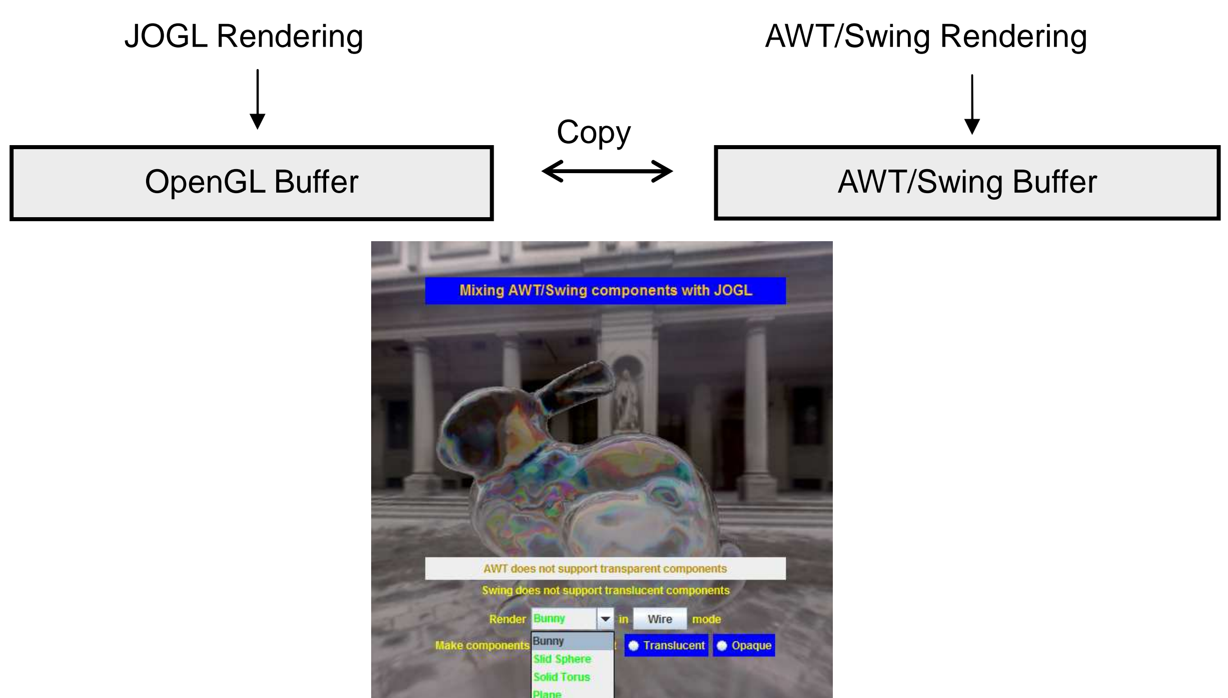


Figure 40. Individual buffers used by JOGL programs and AWT/Swing, which does not allow translucent widgets.

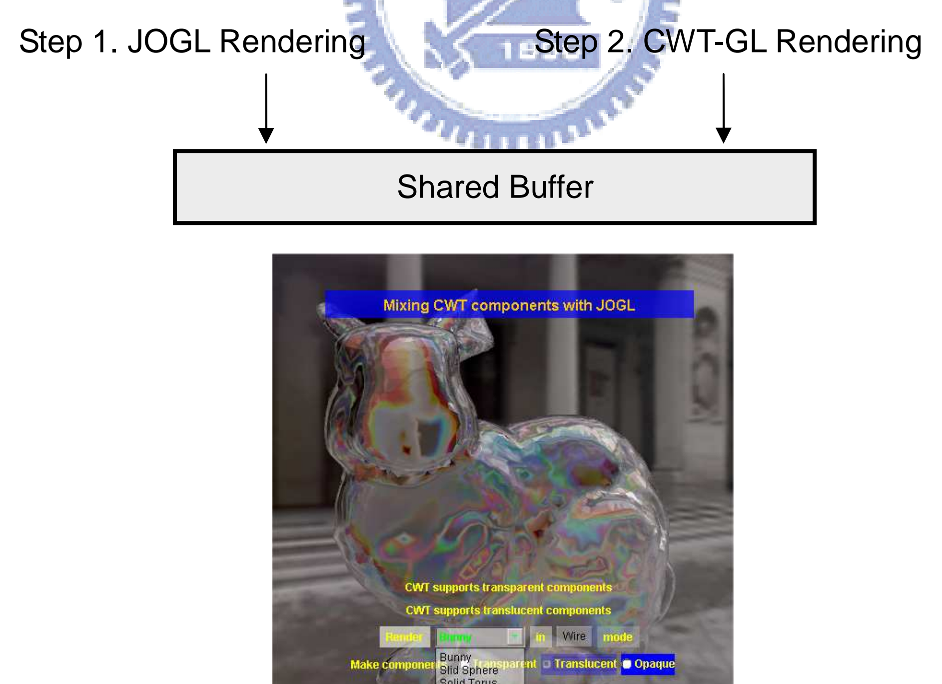


Figure 41. A shared buffer used by JOGL programs and CWT-GL, which allows translucent widgets.

According to the discussion above, we suggest that the bindings of DirectX and OpenGL should be reused. For example, as shown in Figure 41, CWT-GL uses JOGL to render everything, so the interoperability between CWT-GL and JOGL APIs are much easier and more efficient because of the shared buffer. Another good example is that Java 3D uses JOGL as its internal rendering pipeline since version 1.5.

Since JOGL is a well developed OpenGL bindings, it can be the default OpenGL bindings for Java 2D. Once Java AWT/Swing uses JOGL to implement its OpenGL rendering pipeline, not only interoperability but performance among Java AWT/Swing, Java 3D, and JOGL applications will be greatly improved.

To sum up, the following three suggestions can be considered in future Java AWT/Swing. First, the internal DirectX and OpenGL objects should be accessible for game programmers who need to access up-to-date hardware features or change the rendering behaviors. Second, the rendering pipelines of Java AWT/Swing should be decoupled from the JREs for higher and more consistent rendering performance, faster upgrades, and better supports of old JREs. Third, the bindings of DirectX and OpenGL should be reused for lower developing cost, better maintainability, easier interoperability among Java AWT/Swing, Java 3D, and JOGL applications. The three suggestions are illustrated in Figure 42. With applied OCP, these designs may make Java a better game platform in the future.

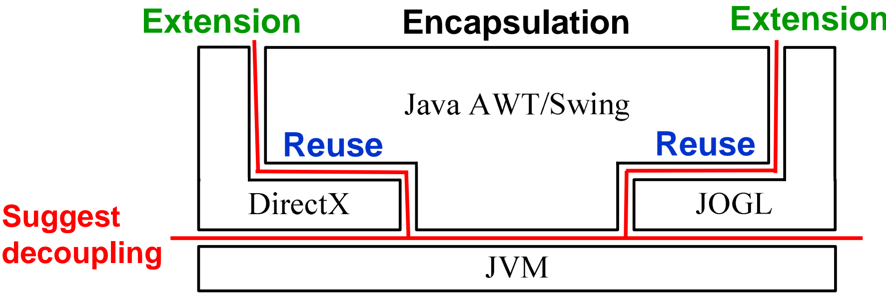


Figure 42. Suggestions for future Java AWT/Swing.

5.2 Drawbacks of CWT

This subsection lists the 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, CWT switches to use CWT-AWT implementation, which incurs 10.3% extra overhead [63]. 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.

E
S
A
1896

Chapter 6 Conclusions

In this dissertation, we design a portable AWT/Swing architecture, called CYC Window Toolkit (CWT), for high and consistent rendering performance for developing cross-platform Java games, especially for applet games written in Java 1.1.

The features of CWT can be summarized as follows. (1) Reach high and consistent performance when using DirectX and OpenGL to render widgets in MSVM and JRE 1.4 to 1.6, which are currently used by most Web browser users. We demonstrated the performance of CWT by applying it to a real applet game, the Bomberman game. (2) Support Java AWT/Swing compatible widgets. Hence, CWT can be easily applied to existing Java games. In addition, programmers who have been familiar to Java AWT/Swing API can adopt CWT without learning new APIs. (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 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 for advanced programmers.

This dissertation implements three versions of the CWT architecture and compares their rendering performance with that of Java AWT on four OSs, including Windows XP, Windows Vista, Fedora and Mac OS X. The results indicate that the approach employed by CWT generally reaches higher and more consistent rendering performance in MSVM and JRE 1.4 to 1.6 on the four OSs. Furthermore, it also helps reduce the efforts of tuning the rendering performance by choosing different graphics APIs and system properties.

The contributions of this dissertation are listed as follows:

- Evaluate the rendering performance of the original Java AWT with different combinations of JREs, graphics 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 graphics APIs and system properties for mostly used JREs on the four operating systems. This weakens the merit of Write-Once-Run-Anywhere of Java for game development.
- Implement three versions of CWT via DirectX, JOGL and AWT, which takes advantage of video hardware acceleration on multiple OSs. Compared to Java AWT, CWT-DX and CWT-GL achieves more consistent and higher rendering performance in MSVM and JRE 1.4 to 1.6 on the four tested OSs.
- The experimental results also reveal three suggestions for future Java. First, the internal DirectX and OpenGL objects should be accessible for game programmers who need to access up-to-date hardware features or change the rendering behaviors. Second, the rendering pipelines of Java AWT/Swing should be decoupled from the JREs for higher and more consistent rendering performance, faster upgrades, and better supports of old JREs. Third, the bindings of DirectX and OpenGL should be reused for lower developing cost, better maintainability, easier interoperability among Java AWT/Swing, Java 3D, and JOGL applications.

We have established a website [15] for releasing the latest implementations of CWT. The benchmark programs and results are also available on the website, as well as demonstrations and a porting guide.

Suggestions for possible future extensions of CWT include Support for Java AWT/Swing 1.2 APIs and beyond, and support for vector graphics. Since CWT is designed mainly for game development, these features may further improve the usability of CWT.

(1) Support for Java AWT/Swing 1.2 APIs and Beyond

Currently, the implementations of CWT only support Java AWT/Swing 1.1 compatible API. Although CWT already reaches high and consistent rendering performance, some Java game programmers may have been familiar to J2SE 1.2 and beyond, which introduces more advanced 2D graphics API supporting line styles, gradient- or texture-filled geometries, affine transform and irregular clipping areas [57]. Supporting these advanced 2D graphics features in CWT may help Java game programmers create more runtime visual effects instead of pre-rendered images.

(2) Support for Vector Graphics

With the rapid development of video cards, the resolutions of monitors have also been improved greatly. At the end of 1980s', the resolution may be limited to CGA (up to 640×200) and EGA (up to 640×350). The screen resolutions become bigger and bigger as time goes by, such as VGA (640×480), SVGA (800×600), XGA (1024×768), SXGA (1280×1024), UXGA (1600×1200), HD (1920×1080), and so on. In the future, there will be surely more new resolutions.

Since games may be played in various resolutions, game programmers need to choose one or more resolutions to support. For example, *RuneScape* [17] supports dynamic resolutions, such as 765×503 and 1024×768 in window mode shown in Figure 43 and Figure 44, respectively.

Figure 43. RuneScape in 765×503-sized resolution.

Figure 44. RuneScape in 1024×768-sized resolution.

However, supporting various resolutions may bring challenges to the design of user interfaces, since the user interfaces may have to be limited to certain smaller size in order to support most commonly used resolutions. With the enhancement of monitors, modern games may need to support several resolutions both in full screen mode and window mode so that the user interfaces can utilize the visible space of the monitors. Traditionally, games are designed in a fixed sized resolution, such as 800×600, typically in full screen mode. In full screen mode, game scenes are normally scaled to fit the maximum resolution of the monitors, such as 1280×1024. On the other hand, games in window mode, such as those

embedded inside Web browsers, would only be shown in the original sizes. However, when fixed-sized scenes designed for small resolutions, such as 800×600, are scaled to different resolutions, such as 1280×1024, the visual quality would be worse. For example, the apparent quality would loss, when a checkbox in fixed-size 80×25, as shown in Figure 45 (A) is scaled to size 240×75, as shown in Figure 45 (B).

Vector graphics is a good solution to the scaling problem, since vector graphics can be scalable to any size without loss of detail. For example, Figure 45 (C) presents that the checkbox is scaled to size 240×75 with smooth outlines. In vector graphics, the rendered sizes of the graphics primitives, including figures, texts, and images, are decided mathematically in runtime. In fact, vector graphics is common in 3D games that allow to zoom in/out and to rotate the scenes without loss of details.

Rendering widgets by graphics primitives, CWT can be extended to support vector graphics while remaining Java AWT/Swing API, which may greatly help the design of user interfaces of games for different resolutions.

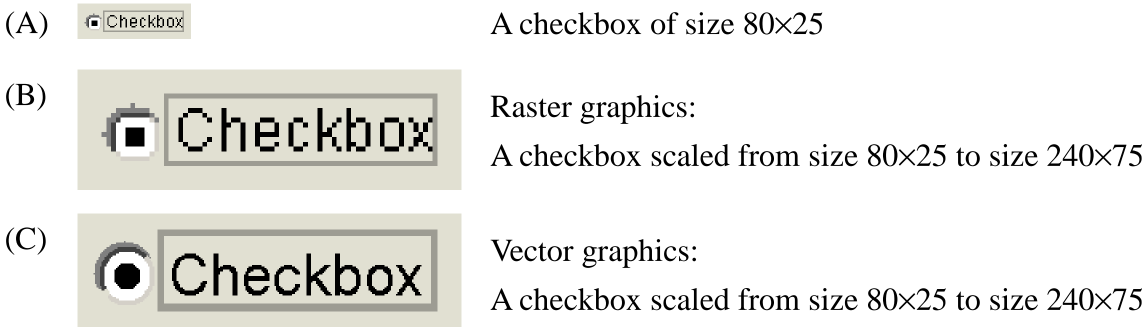


Figure 45. Comparison between raster graphics and vector graphics.

References

[1] Adobe Systems Inc. *Flash Content Reaches 99.0% of Internet Viewers*. Millward Brown survey, conducted December 2008.

[2] Advanced Micro Devices Inc. Radeon X1600 Series – GPU Specifications. Advanced Micro Devices Inc.
http://ati.amd.com/products/RadeonX1600/specs.html (last access: July 2009)

[3] Apple Inc. *Java System Property Reference for Mac OS X*. Apple Inc.

[4] Apple Inc. Graphics & Imaging Overview. Apple Inc.
http://developer.apple.com/graphicsimaging/overview.html (last access: July 2009)

[5] T. Bruckschlegel. *Microbenchmarking C++, C#, and Java*. Dr. Dobb's Journal, 2005.

[6] A. L. Burrows and D. England. Java 3D, 3D Graphical Environments and Behaviour. *Software Practice and Experience*, Vol. 32, Issue 4, April 2002; 359–376.

[7] Bytonic Software. Jake2. Bytonic Software.
http://www.bytonic.de/html/jake2.html (last access: July 2009)

[8] C. Campbell. *STR-Crazy: Improving the OpenGL-based Java 2D Pipeline*. Sun Microsystems Inc.
http://weblogs.java.net/blog/campbell/archive/2005/03/strcrazy_improv_1.html
(last access: July 2009)

[9] P. C. Chu and J. E. Beasley. A Genetic Algorithm for the Multidimensional Knapsack Problem. *Journal of Heuristics*, Vol. 4, Number 1, 1998; 63-86.

[10] A. Denault and J. Kienzle, Avoid Common Pitfalls When Programming 2D Graphics in Java: Lessons Learnt from Implementing the Minueto Toolkit. *ACM Crossroads*, Volume 13 Issue 3, March 2007.

[11] E. Gamma, R. Helm, R. Johnson and J. Vissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[12] A. Gray. GC Usage Statistics.
http://www.andrew-gray.com/dist/stats.shtml (last access: July 2009)

[13] C.-C. Hsu and I.-C. Wu. An Event-driven Framework for Inter-user Communication Applications. *Information and Software Technology*, Vol.48, July 2006; 471-483.

[14] IN-FUSIO. Age of Empires II Mobile. IN-FUSIO. http://www.in-fusio.com/ (last access: July 2009)

[15] Internet Application Technology Lab. CWT - CYC Window Toolkit. National Chiao-Tung University, Taiwan. http://java.csie.nctu.edu.tw/cwt/ (last access: July 2009)

[16] IonChron Inc. Java Games. IonChron Inc. http://www.arcadepod.com/java/ (last access: July 2009)

[17] Jagex Ltd. RuneScape. Jagex Ltd. http://www.runescape.com/ (last access: July 2009)

[18] Jausoft. GL4Java: OpenGL for Java. Jausoft. http://gl4java.sourceforge.net/ (last access: July 2009)

[19] Jellyvision Inc. You Don't Know Jack. Jellyvision Inc. http://www.jellyvision.com/ (last access: July 2009)

[20] jMonkeyEngine.com. jMonkey Engine. jMonkeyEngine.com. http://www.jmonkeyengine.com/ (last access: July 2009)

[21] jPct.net. jPCT. jPct.net. http://www.jpct.net/ (last access: July 2009)

[22] lwjgl.org. LWJGL, Lightweight Java Game Library. lwjgl.org. http://lwjgl.org/ (last access: July 2009)

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

[24] T. McReynolds and D. Blythe. *Advanced Graphics Programming Using OpenGL*. Morgan Kaufmann, February 2005.

[25] B. Meyer. *Object-Oriented Software Construction (2nd Edition)*. Prentice Hall, March 2000.

[26] J. Meyer, B. Bederson, and J.-D. Fekete. Agile2D OpenGL Renderer. Human-Computer Interaction Lab, University of Maryland, USA. http://www.cs.umd.edu/hcil/agile2d/ (last access: July 2009)

[27] Microsoft Corp. Age of Empires. Micorsoft Corp.

http://www.microsoft.com/games/empires/ (last access: July 2009)

[28] Microsoft Corp. *Desktop Window Manager*. Microsoft Corp. http://msdn2.microsoft.com/En-US/library/aa969540.aspx (last access: July 2009)

[29] Microsoft Corp. DirectX: Platform SDK. Microsoft Corp.

[30] Microsoft Corp. Microsoft DirectX. Microsoft Corp. http://msdn.microsoft.com/en-us/directx/ (last access: July 2009)

[31] Microsoft Corp. Microsoft Java Virtual Machine Support. Microsoft Corp. http://www.microsoft.com/mscorp/java/ (last access: July 2009)

[32] Mojang Specifications. Wurm Online. Mojang Specifications. http://www.wurmonline.com/ (last access: July 2009)

[33] Oddlabs ApS. Tribal Trouble. Oddlabs ApS. http://tribaltrouble.com/ (last access: July 2009)

[34] OpenGL Architecture Review Board. *GL_EXT_framebuffer_object in OpenGL Extension Registry*. The OpenGL Architecture Review Board. http://www.opengl.org/registry/specs/EXT/framebuffer_object.txt (last access: July 2009)

[35] OpenGL Architecture Review Board. OpenGL: Open Graphics Library. OpenGL Architecture Review Board. http://www.opengl.org/ (last access: July 2009)

[36] OpenGL Architecture Review Board. *WGL_ARB_pbuffer in OpenGL Extension Registry.* The OpenGL Architecture Review Board. http://www.opengl.org/registry/specs/ARB/wgl_pbuffer.txt (last access: July 2009)

[37] G. Phipps. Comparing Observed Bug and Productivity Rates for Java and C++. *Software Practice and Experience*, Vol. 29, Issue 4, April 1999; 345–358.

[38] R. Pitman. Charva: A Java Windowing Toolkit for Text Terminals. http://www.pitman.co.za/projects/charva/index.html (last access: July 2009)

[39] E. Quinn and C. Christiansen. *Java Pays – Positively.* IDC Bulletin #W16212, 1998.

[40] J. Schaback. FengGUI, Java GUIs with OpenGL. http://www.fenggui.org/ (last access: July 2009)

[41] D. Shreiner, M. Woo, J. Neider and T. Davis. *OpenGL Programming Guide: The*

Official Guide to Learning OpenGL, Version 1.4, Fourth Edition. Addison-Wesley Professional, November 2003.

[42] Sun Microsystems Inc. *Bug ID: 5037133 Mixed mode rendering and 3D effects using Java2D and JOGL together*. Sun Microsystems Inc. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=5037133 (last access: July 2009)

[43] Sun Microsystems Inc. *Bug ID: 6260751 Applets Can't Set sun.java2d.noddraw=true*. Sun Microsystems Inc. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6260751 (last access: July 2009)

[44] Sun Microsystems Inc. *Bug ID: 6343853 Rendering Issues on Vista Caused by Use of GDI and DDraw on Onscreen Surfaces*. Sun Microsystems Inc. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6343853 (last access: July 2009)

[45] Sun Microsystems Inc. *High Performance Graphics – Graphics Performance Improvements in the Java 2 SDK, version 1.4*. Sun Microsystems Inc., 2001.

[46] Sun Microsystems Inc. Java 3D. Sun Microsystems Inc. https://java3d.dev.java.net/ (last access: July 2009)

[47] Sun Microsystems Inc. *Java AWT: Delegation Event Model.* Sun Microsystems Inc., 1997.

[48] Sun Microsystems Inc. Java Home Page. Sun Microsystems Inc. http://java.sun.com/ (last access: July 2009)

[49] Sun Microsystems Inc. *Java SE 6 Performance White Paper*. Sun Microsystems Inc. 2006.

[50] Sun Microsystems Inc. JOGL, Java bindings for OpenGL API. Sun Microsystems Inc. https://jogl.dev.java.net/ (last access: July 2009)

[51] Sun Microsystems Inc. *JSR 231: Java Binding for the OpenGL API.* Sun Microsystems Inc. http://jcp.org/en/jsr/detail?id=231 (last access: July 2009)

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

[53] Sun Microsystems Inc. *The AWT Native Interface*. Sun Microsystems Inc., 1999.

[54] Sun Microsystems Inc. *The VolatileImage APIs User Guide*. Sun Microsystems Inc., 2001.

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

[56] Sun Microsystems Inc. *Painting in AWT and Swing*. Sun Microsystems Inc., 2003.

[57] Sun Microsystems Inc. *Programmer's Guide to the Java 2D API – Enhanced Graphics and Imaging for Java*. Sun Microsystems Inc., 2001.

[58] Sun Microsystems Inc. *System Properties for Java 2D Technology*. Sun Microsystems Inc., 2004.

[59] ThinkNewIdea Internet Technology Corp. CYC Games. ThinkNewIdea Internet Technology Corp. http://cycgame.com/ (last access: July 2009)

[60] Three Rings Design, Inc. Puzzle Pirates. Three Rings Design, Inc. http://www.puzzlepirates.com/ (last access: July 2009)

[61] Vivendi Universal Games. Law & Order: Dead on the Money. Legacy Interactive, Inc. http://www.legacyinteractive.com (last access: July 2009)

[62] W3Schools. OS Platform Statistics. Refsnes Data Inc. http://www.w3schools.com/browsers/browsers_os.asp (last access: July 2009)

[63] Y.-H. Wang, I.-C. Wu, and J.-Y. Jiang. A Portable AWT/Swing Architecture for Java Game Development. *Software Practice and Experience*, Vol. 37, Issue 7, June 2007; 727-745.

[64] Y.-H. Wang and I.-C. Wu. Achieving High and Consistent Rendering Performance of Java AWT/Swing on Multiple Platforms. *Software Practice and Experience*, Vol. 39, Issue 7, March 2009; 701-736.

[65] A. Wong and A. Kennings. Adaptive multiple texture approach to texture packing for 3D video games. *Proceedings of the 2007 conference on Future Play*, Toronto, Canada; 189-196.

[66] X.Org Foundation. X.Org Project. X.Org Foundation. http://www.x.org/wiki/ (last access: July 2009)

[67] Xith3D Community. The Xith3D Project. Xith3D Community. http://www.xith.org/ (last access: July 2009)

[68] Yahoo Inc. Yahoo! Games. Yahoo Inc.
http://games.yahoo.com/ (last access: July 2009)

Appendix A Results of Micro-Benchmark

This appendix presents the detail results of the micro-benchmark program. The micro-benchmark includes 21 tests, divided into image, text and figure tests, as follows.

- Image tests, as shown in Figure 15 (A), are 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.
- Text tests, as shown in Figure 15 (B), have two subtests: simple texts (using the word "Running") and articles (consisting of about 13,000 characters on the screen, including 1,562 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.
- Figure tests, as shown in Figure 15 (C), include 12 subtests which draw lines, polylines, polygons, rectangles, round rectangles, arcs, ovals, solid polygons, solid rectangles, solid round rectangles, pies, and solid ovals. The metric for these tests is "rendered items per second."

Table 11. Rendered items per second of opaque image tests.

Toolkit	JRE	System Property	Graphics API	OS			
				Windows XP	Windows Vista	Fedora Core 6	Mac OS X
AWT	MSVM	None	Img+Img	11221	63104	N/A	N/A
	1.1	None	Img+Img	11197	60436	N/A	N/A
	1.2	None	Img+Img	39873	60048	N/A	N/A
	1.3	None	Img+Img	40409	60877	31792	2097
	1.4	None	Img+Img	42492	14776	35232	66341
			Vlt+Cpt	58662	14737	2617	66371
		Special	Vlt+Cpt	58854	36180	2615	66577
	1.5	None	Img+Img	42856	36478	35178	68965
			Vlt+Cpt	59900	40849	4602	70422
		Special	Vlt+Cpt	58571	63264	4560	71089
		OpenGL	Img+Img	27619	63317	34554	N/A
			Vlt+Cpt	27512	41447	36818	N/A
	1.6	None	Img+Img	43617	42360	40106	N/A
			Vlt+Cpt	61099	42277	4598	N/A
		Special	Vlt+Cpt	58486	17533	4596	N/A
		OpenGL	Img+Img	25274	17247	37276	N/A
			Vlt+Cpt	52398	43440	47393	N/A
CWT-DX	MSVM	None	Img+Img	63104	43503	N/A	N/A
CWT-GL	1.4	None	Img+Img	60436	43414	58846	54644
	1.5	None	Img+Img	60048	22023	59405	56775
	1.6	None	Img+Img	60877	50150	60096	N/A

Table 12. Rendered items per second of transparent image tests.

Toolkit	JRE	System Property	Graphics API	OS			
				Windows XP	Windows Vista	Fedora Core 6	Mac OS X
AWT	MSVM	None	Img+Img	1888	71428	N/A	N/A
	1.1	None	Img+Img	1884	80819	N/A	N/A
	1.2	None	Img+Img	23112	80429	N/A	N/A
	1.3	None	Img+Img	22869	80171	7407	1212
	1.4	None	Img+Img	23636	3901	17994	25222
			Vlt+Cpt	70226	3869	3945	25286
		Special	Vlt+Cpt	70194	20966	3948	25265
	1.5	None	Img+Img	23752	20847	17998	25786
			Vlt+Cpt	70258	22492	6683	25969
		Special	Vlt+Cpt	70126	75680	6552	33875
		OpenGL	Img+Img	41841	75718	34498	N/A
			Vlt+Cpt	41494	22458	36567	N/A
	1.6	None	Img+Img	23205	23044	19414	N/A
			Vlt+Cpt	70291	23091	6683	N/A
		Special	Vlt+Cpt	44483	17223	6693	N/A
		OpenGL	Img+Img	16757	17027	22377	N/A
			Vlt+Cpt	68306	22596	47096	N/A
CWT-DX	MSVM	None	Img+Img	71428	22586	N/A	N/A
CWT-GL	1.4	None	Img+Img	80819	22589	79703	65588
	1.5	None	Img+Img	80429	15027	80260	69380
	1.6	None	Img+Img	80171	63775	81967	N/A

Table 13. Rendered items per second of translucent image tests.

Toolkit	JRE	System Property	Graphics API	OS			
				Windows XP	Windows Vista	Fedora Core 6	Mac OS X
AWT	MSVM	None	Img+Img	1887	72011	N/A	N/A
	1.1	None	Img+Img	1884	81303	N/A	N/A
	1.2	None	Img+Img	3255	80906	N/A	N/A
	1.3	None	Img+Img	3612	81922	1009	1133
	1.4	None	Img+Img	6145	3905	3283	3457
			Vlt+Cpt	6037	3872	3285	18820
		Special	Vlt+Cpt	37983	3183	3277	18801
	1.5	None	Img+Img	6122	3568	4668	3505
			Vlt+Cpt	5968	5985	4705	20029
		Special	Vlt+Cpt	213	6034	4616	7628
		OpenGL	Img+Img	764	27119	1858	N/A
			Vlt+Cpt	38639	5906	36710	N/A
	1.6	None	Img+Img	8504	6121	4743	N/A
			Vlt+Cpt	8378	6142	4559	N/A
		Special	Vlt+Cpt	45099	580	4707	N/A
		OpenGL	Img+Img	7475	17037	4651	N/A
			Vlt+Cpt	69188	8494	46699	N/A
CWT-DX	MSVM	None	Img+Img	72011	8521	N/A	N/A
CWT-GL	1.4	None	Img+Img	81303	8531	79746	65645
	1.5	None	Img+Img	80906	7032	80775	68807
	1.6	None	Img+Img	81922	63965	82462	N/A

Table 14. Rendered items per second of runtime opaque image tests.

Toolkit	JRE	System Property	Graphics API	OS			
				Windows XP	Windows Vista	Fedora Core 6	Mac OS X
AWT	MSVM	None	Img+Img	11221	62998	N/A	N/A
	1.1	None	Img+Img	11190	61349	N/A	N/A
	1.2	None	Img+Img	4245	60728	N/A	N/A
	1.3	None	Img+Img	40551	61984	31745	1284
	1.4	None	Img+Img	42613	14781	34884	49983
			Vlt+Cpt	5940	14342	2617	23577
		Special	Vlt+Cpt	27427	4129	2616	23614
	1.5	None	Img+Img	42698	36416	8569	58456
			Vlt+Cpt	6040	41061	4613	25737
		Special	Vlt+Cpt	5888	6073	4535	5609
		OpenGL	Img+Img	887	23651	1209	N/A
			Vlt+Cpt	27654	41841	36647	N/A
	1.6	None	Img+Img	43910	6211	9245	N/A
			Vlt+Cpt	10067	6217	4605	N/A
		Special	Vlt+Cpt	45112	635	4607	N/A
		OpenGL	Img+Img	25523	17413	37378	N/A
			Vlt+Cpt	41550	43289	46511	N/A
CWT-DX	MSVM	None	Img+Img	62998	10356	N/A	N/A
CWT-GL	1.4	None	Img+Img	61349	10350	59405	55146
	1.5	None	Img+Img	60728	22185	59976	57229
	1.6	None	Img+Img	61984	39504	61099	N/A

Table 15. Rendered items per second of runtime transparent image tests.

Toolkit	JRE	System Property	Graphics API	OS			
				Windows XP	Windows Vista	Fedora Core 6	Mac OS X
AWT	MSVM	None	Img+Img	1887	71942	N/A	N/A
	1.1	None	Img+Img	1883	81833	N/A	N/A
	1.2	None	Img+Img	4967	81212	N/A	N/A
	1.3	None	Img+Img	5545	81300	2481	1204
	1.4	None	Img+Img	9900	3910	4045	4063
			Vlt+Cpt	9680	3758	3952	22064
		Special	Vlt+Cpt	37935	4809	3949	22048
	1.5	None	Img+Img	9967	5548	6676	4132
			Vlt+Cpt	9790	9579	6691	23888
		Special	Vlt+Cpt	8812	9587	6596	10221
		OpenGL	Img+Img	783	27911	1913	N/A
			Vlt+Cpt	42016	9600	36746	N/A
	1.6	None	Img+Img	11917	9895	6659	N/A
			Vlt+Cpt	11709	9897	6682	N/A
		Special	Vlt+Cpt	45167	588	6701	N/A
		OpenGL	Img+Img	9951	17449	6661	N/A
			Vlt+Cpt	69412	11901	46801	N/A
CWT-DX	MSVM	None	Img+Img	71942	11944	N/A	N/A
CWT-GL	1.4	None	Img+Img	81833	11960	80428	65876
	1.5	None	Img+Img	81212	9294	81212	69188
	1.6	None	Img+Img	81300	64294	83379	N/A

Table 16. Rendered items per second of runtime translucent image tests.

Toolkit	JRE	System Property	Graphics API	OS			
				Windows XP	Windows Vista	Fedora Core 6	Mac OS X
AWT	MSVM	None	Img+Img	1887	71496	N/A	N/A
	1.1	None	Img+Img	1884	81398	N/A	N/A
	1.2	None	Img+Img	3258	80648	N/A	N/A
	1.3	None	Img+Img	3614	82147	1010	1127
	1.4	None	Img+Img	6137	3906	3282	3481
			Vlt+Cpt	6020	3742	3285	18884
		Special	Vlt+Cpt	37558	3205	3277	18841
	1.5	None	Img+Img	6129	3581	4703	3477
			Vlt+Cpt	6080	5924	4746	20106
		Special	Vlt+Cpt	213	6026	4555	7660
		OpenGL	Img+Img	765	27382	1859	N/A
			Vlt+Cpt	38790	5918	36674	N/A
	1.6	None	Img+Img	8521	6123	4746	N/A
			Vlt+Cpt	8370	6129	4749	N/A
		Special	Vlt+Cpt	45126	578	4753	N/A
		OpenGL	Img+Img	7481	17467	4656	N/A
			Vlt+Cpt	69124	8485	46948	N/A
CWT-DX	MSVM	None	Img+Img	71496	8533	N/A	N/A
CWT-GL	1.4	None	Img+Img	81398	8528	80000	65847
	1.5	None	Img+Img	80648	7032	81037	69220
	1.6	None	Img+Img	82147	64349	83102	N/A

Table 17. Rendered items per second of simple text tests. Font size is 12.

Toolkit	JRE	System Property	Graphics API	OS			
				Windows XP	Windows Vista	Fedora Core 6	Mac OS X
AWT	MSVM	None	Img+Img	165016	120579	N/A	N/A
	1.1	None	Img+Img	80257	34948	N/A	N/A
	1.2	None	Img+Img	163398	32960	N/A	N/A
	1.3	None	Img+Img	171624	34207	51741	7359
	1.4	None	Img+Img	111111	87006	35569	47110
			Vlt+Cpt	24590	50950	35646	47423
		Special	Vlt+Cpt	24581	101350	63883	47619
	1.5	None	Img+Img	197642	104311	52337	115384
			Vlt+Cpt	56305	95724	51705	120096
		Special	Vlt+Cpt	56732	15002	111607	157067
		OpenGL	Img+Img	41005	14846	38118	N/A
			Vlt+Cpt	39450	173612	38206	N/A
	1.6	None	Img+Img	238474	180075	54327	N/A
			Vlt+Cpt	57230	179212	54844	N/A
		Special	Vlt+Cpt	125226	15851	114155	N/A
		OpenGL	Img+Img	46596	15695	106457	N/A
			Vlt+Cpt	177095	204360	20223	N/A
CWT-DX	MSVM	None	Img+Img	120579	198680	N/A	N/A
CWT-GL	1.4	None	Img+Img	34948	199734	34152	25146
	1.5	None	Img+Img	32960	37518	37027	29538
	1.6	None	Img+Img	34207	148957	41911	N/A

Table 18. Rendered items per second of article tests. Font size is 12.

Toolkit	JRE	System Property	Graphics API	OS			
				Windows XP	Windows Vista	Fedora Core 6	Mac OS X
AWT	MSVM	None	Img+Img	56454	48575	N/A	N/A
	1.1	None	Img+Img	29245	57318	N/A	N/A
	1.2	None	Img+Img	129650	53918	N/A	N/A
	1.3	None	Img+Img	10502	54465	1445	1010
	1.4	None	Img+Img	11217	45689	10042	22407
			Vlt+Cpt	8189	20553	9787	22268
		Special	Vlt+Cpt	8056	86505	11229	22344
	1.5	None	Img+Img	143967	23786	45017	25108
			Vlt+Cpt	53956	5965	44708	25286
		Special	Vlt+Cpt	54248	3940	93168	115207
		OpenGL	Img+Img	25050	4291	28312	N/A
			Vlt+Cpt	25154	126262	28126	N/A
	1.6	None	Img+Img	156914	130208	46904	N/A
			Vlt+Cpt	54190	130321	46860	N/A
		Special	Vlt+Cpt	62190	12723	95421	N/A
		OpenGL	Img+Img	42625	12814	89766	N/A
			Vlt+Cpt	49520	144370	28121	N/A
CWT-DX	MSVM	None	Img+Img	48575	139664	N/A	N/A
CWT-GL	1.4	None	Img+Img	57318	139794	55248	13362
	1.5	None	Img+Img	53918	34523	60193	15891
	1.6	None	Img+Img	54465	42637	65731	N/A

Table 19. Rendered items per second of texts with different font size from 10 to 64.

Font Size	Test 1	Test 2	Test 3	Test 4	Test 5
10	157053	79183	56871	69932	70763
12	136551	77337	55953	68269	68646
14	129146	77078	55078	69406	69697
16	109803	75165	54489	70244	69699
20	87800	12370	53453	67169	69388
24	80805	11491	51910	64102	69100
28	67481	11226	50120	32237	70424
32	55730	11029	47142	25077	68852
36	48225	4085	45587	18892	63261
40	41862	3316	43767	15587	62942
48	31992	3112	40407	11410	62331
56	25685	2981	37741	8814	58823
64	20141	2858	35013	6929	48732

Test 1: RE=(AWT, 1.6, Img+Img, None, XP)
Test 2: RE=(AWT, 1.6, Vlt+Cpt, Special, XP)
Test 3: Geometry-based text engine, RE=(CWT-GL, 1.6, Img+Img, None, XP)
Test 4: Texture-based text engine with 1 MB cache, RE=(CWT-GL, 1.6, Img+Img, None, XP)
Test 5: Texture-based text engine with 16 MB cache, RE=(CWT-GL, 1.6, Img+Img, None, XP)

Table 20. Rendered items per second of line tests.

Toolkit	JRE	System Property	Graphics API	OS			
				Windows XP	Windows Vista	Fedora Core 6	Mac OS X
AWT	MSVM	None	Img+Img	301204	111779	N/A	N/A
	1.1	None	Img+Img	304878	247671	N/A	N/A
	1.2	None	Img+Img	309919	248034	N/A	N/A
	1.3	None	Img+Img	323278	242729	94876	6834
	1.4	None	Img+Img	276765	176886	138002	37155
			Vlt+Cpt	37229	174217	144092	37304
		Special	Vlt+Cpt	35842	144230	131118	37248
	1.5	None	Img+Img	297667	145632	147786	38431
			Vlt+Cpt	67144	184549	147347	39011
		Special	Vlt+Cpt	65876	37602	138248	266431
		OpenGL	Img+Img	65274	35202	218659	N/A
			Vlt+Cpt	63831	265962	220913	N/A
	1.6	None	Img+Img	340912	269301	158060	N/A
			Vlt+Cpt	67876	269785	158062	N/A
		Special	Vlt+Cpt	226244	20804	145630	N/A
		OpenGL	Img+Img	50033	21058	132275	N/A
			Vlt+Cpt	231843	306170	382733	N/A
CWT-DX	MSVM	None	Img+Img	111779	300059	N/A	N/A
CWT-GL	1.4	None	Img+Img	247671	300639	311868	255974
	1.5	None	Img+Img	248034	39830	333362	312508
	1.6	None	Img+Img	242729	196592	350470	N/A

Table 21. Rendered items per second of polyline tests.

Toolkit	JRE	System Property	Graphics API	OS			
				Windows XP	Windows Vista	Fedora Core 6	Mac OS X
AWT	MSVM	None	Img+Img	119047	2959	N/A	N/A
	1.1	None	Img+Img	119141	53096	N/A	N/A
	1.2	None	Img+Img	72185	53634	N/A	N/A
	1.3	None	Img+Img	70224	54367	32418	1120
	1.4	None	Img+Img	62188	66430	52466	4547
			Vlt+Cpt	63722	67872	52947	4573
		Special	Vlt+Cpt	61932	57229	48669	4569
	1.5	None	Img+Img	62111	55493	53801	4614
			Vlt+Cpt	63884	55370	53457	4647
		Special	Vlt+Cpt	62292	2503	54386	83240
		OpenGL	Img+Img	56053	2494	79323	N/A
			Vlt+Cpt	54744	56433	83939	N/A
	1.6	None	Img+Img	68369	58009	55269	N/A
			Vlt+Cpt	65819	58189	55782	N/A
		Special	Vlt+Cpt	100671	19548	57803	N/A
		OpenGL	Img+Img	31439	19711	55782	N/A
			Vlt+Cpt	80906	62866	105708	N/A
CWT-DX	MSVM	None	Img+Img	2959	62735	N/A	N/A
CWT-GL	1.4	None	Img+Img	53096	62814	56882	43949
	1.5	None	Img+Img	53634	26723	57384	50539
	1.6	None	Img+Img	54367	74924	59101	N/A

Table 22. Rendered items per second of polygon tests.

Toolkit	JRE	System Property	Graphics API	OS			
				Windows XP	Windows Vista	Fedora Core 6	Mac OS X
AWT	MSVM	None	Img+Img	107449	3580	N/A	N/A
	1.1	None	Img+Img	106391	80953	N/A	N/A
	1.2	None	Img+Img	66904	78411	N/A	N/A
	1.3	None	Img+Img	65104	84127	30018	954
	1.4	None	Img+Img	57339	61702	49228	4045
			Vlt+Cpt	64184	62814	49635	4072
		Special	Vlt+Cpt	61906	53956	45620	4074
	1.5	None	Img+Img	54789	52101	50403	4117
			Vlt+Cpt	63667	51546	50522	4112
		Special	Vlt+Cpt	62162	2454	52594	78492
		OpenGL	Img+Img	55907	2443	75037	N/A
			Vlt+Cpt	54944	52337	79239	N/A
	1.6	None	Img+Img	63371	53859	52210	N/A
			Vlt+Cpt	66577	53859	52174	N/A
		Special	Vlt+Cpt	101146	19467	54486	N/A
		OpenGL	Img+Img	30278	19700	52558	N/A
			Vlt+Cpt	90854	59737	100133	N/A
CWT-DX	MSVM	None	Img+Img	3580	59085	N/A	N/A
CWT-GL	1.4	None	Img+Img	80953	59062	89928	64654
	1.5	None	Img+Img	78411	25835	90800	79744
	1.6	None	Img+Img	84127	84459	94876	N/A

Table 23. Rendered items per second of polygon filling tests.

Toolkit	JRE	System Property	Graphics API	OS			
				Windows XP	Windows Vista	Fedora Core 6	Mac OS X
AWT	MSVM	None	Img+Img	35688	3564	N/A	N/A
	1.1	None	Img+Img	36328	17166	N/A	N/A
	1.2	None	Img+Img	7405	17142	N/A	N/A
	1.3	None	Img+Img	8194	17231	2992	758
	1.4	None	Img+Img	13088	14841	10398	7747
			Vlt+Cpt	25117	14443	10402	7685
		Special	Vlt+Cpt	24867	7093	7691	7671
	1.5	None	Img+Img	13564	7857	10455	7930
			Vlt+Cpt	25012	12614	10416	8007
		Special	Vlt+Cpt	24792	1076	9592	13707
		OpenGL	Img+Img	8085	1082	2812	N/A
			Vlt+Cpt	8045	12962	2819	N/A
	1.6	None	Img+Img	15211	13304	10534	N/A
			Vlt+Cpt	25497	13346	10485	N/A
		Special	Vlt+Cpt	9792	6284	9540	N/A
		OpenGL	Img+Img	12345	6396	9557	N/A
			Vlt+Cpt	9498	15086	5341	N/A
CWT-DX	MSVM	None	Img+Img	3564	15224	N/A	N/A
CWT-GL	1.4	None	Img+Img	17166	15191	17287	12543
	1.5	None	Img+Img	17142	11206	17327	13101
	1.6	None	Img+Img	17231	7841	17467	N/A

Table 24. Rendered items per second of rectangle tests.

Toolkit	JRE	System Property	Graphics API	OS			
				Windows XP	Windows Vista	Fedora Core 6	Mac OS X
AWT	MSVM	None	Img+Img	158899	20164	N/A	N/A
	1.1	None	Img+Img	163043	88599	N/A	N/A
	1.2	None	Img+Img	272734	97546	N/A	N/A
	1.3	None	Img+Img	266532	101388	84364	11031
	1.4	None	Img+Img	252525	114068	124378	25167
			Vlt+Cpt	45276	109249	125425	25453
		Special	Vlt+Cpt	43808	132391	125736	25523
	1.5	None	Img+Img	268337	136861	131810	26122
			Vlt+Cpt	23995	171037	130776	26305
		Special	Vlt+Cpt	23763	43290	125733	243902
		OpenGL	Img+Img	62849	40160	113472	N/A
			Vlt+Cpt	62061	235124	114155	N/A
	1.6	None	Img+Img	297619	245101	136363	N/A
			Vlt+Cpt	23763	243906	137362	N/A
		Special	Vlt+Cpt	320512	20430	132042	N/A
		OpenGL	Img+Img	48843	20636	119712	N/A
			Vlt+Cpt	180073	264550	247536	N/A
CWT-DX	MSVM	None	Img+Img	20164	263162	N/A	N/A
CWT-GL	1.4	None	Img+Img	88599	263157	142585	116550
	1.5	None	Img+Img	97546	38859	151209	136115
	1.6	None	Img+Img	101388	157563	159235	N/A

Table 25. Rendered items per second of rectangle filling tests.

Toolkit	JRE	System Property	Graphics API	OS: Windows XP	OS: Windows Vista	OS: Fedora Core 6	OS: Mac OS X
AWT	MSVM	None	Img+Img	116551	24473	N/A	N/A
	1.1	None	Img+Img	116822	92266	N/A	N/A
	1.2	None	Img+Img	72046	92764	N/A	N/A
	1.3	None	Img+Img	73064	93691	10825	12319
	1.4	None	Img+Img	54014	58116	61728	127877
			Vlt+Cpt	77002	56796	62266	127442
		Special	Vlt+Cpt	77041	54864	34956	128205
	1.5	None	Img+Img	54545	55268	63559	143266
			Vlt+Cpt	24374	49180	63345	147928
		Special	Vlt+Cpt	24378	84745	46196	58433
		OpenGL	Img+Img	58640	84889	86206	N/A
			Vlt+Cpt	58846	50200	92137	N/A
	1.6	None	Img+Img	57983	51387	64766	N/A
			Vlt+Cpt	24378	51404	64766	N/A
		Special	Vlt+Cpt	105708	19825	46845	N/A
		OpenGL	Img+Img	29862	19955	45358	N/A
			Vlt+Cpt	87977	54288	109897	N/A
CWT-DX	MSVM	None	Img+Img	24473	54248	N/A	N/A
CWT-GL	1.4	None	Img+Img	92266	54268	100603	75757
	1.5	None	Img+Img	92764	25037	102529	80000
	1.6	None	Img+Img	93691	82417	104166	N/A

Table 26. Rendered items per second of round rectangle tests.

Toolkit	JRE	System Property	Graphics API	OS			
				Windows XP	Windows Vista	Fedora Core 6	Mac OS X
AWT	MSVM	None	Img+Img	57981	56284	N/A	N/A
	1.1	None	Img+Img	58049	33245	N/A	N/A
	1.2	None	Img+Img	55844	33714	N/A	N/A
	1.3	None	Img+Img	51318	36638	39011	2258
	1.4	None	Img+Img	53975	92764	90634	19022
			Vlt+Cpt	31165	91968	91463	19013
		Special	Vlt+Cpt	30574	46210	43029	19044
	1.5	None	Img+Img	47953	54864	92193	19391
			Vlt+Cpt	35071	47953	92592	19515
		Special	Vlt+Cpt	34738	2874	39113	50985
		OpenGL	Img+Img	29982	2858	38749	N/A
			Vlt+Cpt	29411	44078	38480	N/A
	1.6	None	Img+Img	82965	44860	94279	N/A
			Vlt+Cpt	35646	44994	94696	N/A
		Special	Vlt+Cpt	54387	13954	60681	N/A
		OpenGL	Img+Img	34152	14058	58892	N/A
			Vlt+Cpt	36945	75872	42722	N/A
CWT-DX	MSVM	None	Img+Img	56284	75872	N/A	N/A
CWT-GL	1.4	None	Img+Img	33245	75833	46699	72709
	1.5	None	Img+Img	33714	28619	45058	65673
	1.6	None	Img+Img	36638	34028	56053	N/A

Table 27. Rendered items per second of round rectangle filling tests.

Toolkit	JRE	System Property	Graphics API	OS			
				Windows XP	Windows Vista	Fedora Core 6	Mac OS X
AWT	MSVM	None	Img+Img	75187	10534	N/A	N/A
	1.1	None	Img+Img	77399	29509	N/A	N/A
	1.2	None	Img+Img	18265	30072	N/A	N/A
	1.3	None	Img+Img	11963	32495	7040	2069
	1.4	None	Img+Img	25653	43152	46554	32651
			Vlt+Cpt	40927	42277	46816	32658
		Special	Vlt+Cpt	40021	16929	17568	32873
	1.5	None	Img+Img	25693	11831	48200	33594
			Vlt+Cpt	41061	24220	47938	33890
		Special	Vlt+Cpt	40529	1687	21431	28527
		OpenGL	Img+Img	16960	1690	10025	N/A
			Vlt+Cpt	16950	23900	10023	N/A
	1.6	None	Img+Img	31853	24887	48732	N/A
			Vlt+Cpt	41806	25008	48574	N/A
		Special	Vlt+Cpt	21222	10367	19454	N/A
		OpenGL	Img+Img	20740	10585	18912	N/A
			Vlt+Cpt	15316	30637	18522	N/A
CWT-DX	MSVM	None	Img+Img	10534	30537	N/A	N/A
CWT-GL	1.4	None	Img+Img	29509	30624	44299	39968
	1.5	None	Img+Img	30072	18268	43053	39557
	1.6	None	Img+Img	32495	14345	53342	N/A

Table 28. Rendered items per second of arc tests.

Toolkit	JRE	System Property	Graphics API	OS			
				Windows XP	Windows Vista	Fedora Core 6	Mac OS X
AWT	MSVM	None	Img+Img	47006	47709	N/A	N/A
	1.1	None	Img+Img	46201	65905	N/A	N/A
	1.2	None	Img+Img	30593	66401	N/A	N/A
	1.3	None	Img+Img	27052	75766	21270	2390
	1.4	None	Img+Img	37220	54825	119426	14820
			Vlt+Cpt	27512	53513	120096	14829
		Special	Vlt+Cpt	27036	27943	30832	14810
	1.5	None	Img+Img	33783	26009	124792	15058
			Vlt+Cpt	30959	34193	124069	15146
		Special	Vlt+Cpt	30681	2888	29347	40053
		OpenGL	Img+Img	24267	2871	28571	N/A
			Vlt+Cpt	23745	31558	28489	N/A
	1.6	None	Img+Img	48496	32334	129645	N/A
			Vlt+Cpt	31465	32580	130434	N/A
		Special	Vlt+Cpt	40816	12509	39861	N/A
		OpenGL	Img+Img	26389	12649	38719	N/A
			Vlt+Cpt	26164	45689	32587	N/A
CWT-DX	MSVM	None	Img+Img	47709	45899	N/A	N/A
CWT-GL	1.4	None	Img+Img	65905	45871	71564	114416
	1.5	None	Img+Img	66401	23012	66341	105857
	1.6	None	Img+Img	75766	22528	86206	N/A

Table 29. Rendered items per second of arc filling tests.

Toolkit	JRE	System Property	Graphics API	OS			
				Windows XP	Windows Vista	Fedora Core 6	Mac OS X
AWT	MSVM	None	Img+Img	43377	12486	N/A	N/A
	1.1	None	Img+Img	43314	49011	N/A	N/A
	1.2	None	Img+Img	15723	56681	N/A	N/A
	1.3	None	Img+Img	12239	63085	8001	2036
	1.4	None	Img+Img	18934	32815	40672	16781
			Vlt+Cpt	28414	32064	40794	16759
		Special	Vlt+Cpt	28232	14625	14869	16799
	1.5	None	Img+Img	19098	11980	41367	16998
			Vlt+Cpt	28653	18270	41152	17109
		Special	Vlt+Cpt	28354	2041	18272	27377
		OpenGL	Img+Img	12909	2041	10360	N/A
			Vlt+Cpt	12892	18116	10365	N/A
	1.6	None	Img+Img	32523	18808	41829	N/A
			Vlt+Cpt	28746	19003	41666	N/A
		Special	Vlt+Cpt	17476	8743	21761	N/A
		OpenGL	Img+Img	21312	8867	21011	N/A
			Vlt+Cpt	14460	31289	17504	N/A
CWT-DX	MSVM	None	Img+Img	12486	31551	N/A	N/A
CWT-GL	1.4	None	Img+Img	49011	31605	69637	69092
	1.5	None	Img+Img	56681	18603	64906	67688
	1.6	None	Img+Img	63085	13201	83472	N/A

Table 30. Rendered items per second of oval tests.

Toolkit	JRE	System Property	Graphics API	OS			
				Windows XP	Windows Vista	Fedora Core 6	Mac OS X
AWT	MSVM	None	Img+Img	52065	51089	N/A	N/A
	1.1	None	Img+Img	53040	61602	N/A	N/A
	1.2	None	Img+Img	25201	63317	N/A	N/A
	1.3	None	Img+Img	21766	72538	17562	2789
	1.4	None	Img+Img	32916	70224	131810	11713
			Vlt+Cpt	29964	69897	132978	11694
		Special	Vlt+Cpt	29428	23629	27168	11730
	1.5	None	Img+Img	29019	21425	138248	11853
			Vlt+Cpt	33474	30407	138121	11900
		Special	Vlt+Cpt	33119	2886	25884	34309
		OpenGL	Img+Img	21713	2877	24638	N/A
			Vlt+Cpt	21331	27347	24537	N/A
	1.6	None	Img+Img	67114	28005	144369	N/A
			Vlt+Cpt	33821	28248	145348	N/A
		Special	Vlt+Cpt	36629	12028	52065	N/A
		OpenGL	Img+Img	30870	12160	50437	N/A
			Vlt+Cpt	36864	61702	40718	N/A
CWT-DX	MSVM	None	Img+Img	51089	62421	N/A	N/A
CWT-GL	1.4	None	Img+Img	61602	62499	58456	96215
	1.5	None	Img+Img	63317	26543	55066	82964
	1.6	None	Img+Img	72538	32757	69864	N/A

Table 31. Rendered items per second of oval filling tests.

Toolkit	JRE	System Property	Graphics API	OS			
				Windows XP	Windows Vista	Fedora Core 6	Mac OS X
AWT	MSVM	None	Img+Img	13665	11586	N/A	N/A
	1.1	None	Img+Img	13663	47232	N/A	N/A
	1.2	None	Img+Img	13718	46684	N/A	N/A
	1.3	None	Img+Img	10206	53511	6813	2526
	1.4	None	Img+Img	17299	37453	40420	15898
			Vlt+Cpt	11181	37119	40639	15939
		Special	Vlt+Cpt	11136	12810	14116	15974
	1.5	None	Img+Img	17211	10123	40181	16196
			Vlt+Cpt	11907	16663	40584	16253
		Special	Vlt+Cpt	11844	1866	17251	26014
		OpenGL	Img+Img	12395	1870	10378	N/A
			Vlt+Cpt	12340	16368	10385	N/A
	1.6	None	Img+Img	35859	17087	41597	N/A
			Vlt+Cpt	11928	17325	41106	N/A
		Special	Vlt+Cpt	16662	8511	21616	N/A
		OpenGL	Img+Img	22391	8606	20821	N/A
			Vlt+Cpt	16720	33244	18698	N/A
CWT-DX	MSVM	None	Img+Img	11586	34514	N/A	N/A
CWT-GL	1.4	None	Img+Img	47232	34522	57033	60777
	1.5	None	Img+Img	46684	19612	53918	57780
	1.6	None	Img+Img	53511	15255	68057	N/A

Appendix B Results of Macro-Benchmark

The macro-benchmark program is to simulate a Bomberman game. 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 20000 frames.

Table 32. Average frame rate (in FPS) of the Bomberman game.
The numbers with "*" mean that the screen was not rendered correctly.

Toolkit	JRE	System Property	Graphics API	OS			
				Windows XP	Windows Vista	Fedora Core 6	Mac OS X 10.4.11
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
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

Table 33. Average frame rate (in FPS) of the Bomberman game, where JRE∈{1.5}. The numbers with "*" mean that the screen was not rendered correctly.

Toolkit	JRE	System Property	Graphics API	OS			
				Windows XP	Windows Vista	Fedora Core 6	Mac OS X 10.4.11
AWT	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
			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
			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

Table 34. Average frame rate (in FPS) of the Bomberman game, where JRE∈{1.6}. The numbers with "*" mean that the screen was not rendered correctly.

Toolkit	JRE	System Property	Graphics API	OS			
				Windows XP	Windows Vista	Fedora Core 6	Mac OS X 10.4.11
AWT	1.6	None	Img+Img	340	315	108	N/A
			Img+Cpt	325	301	98	N/A
			Img+CptVlt	323	304	111	N/A
			Cpt+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
			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

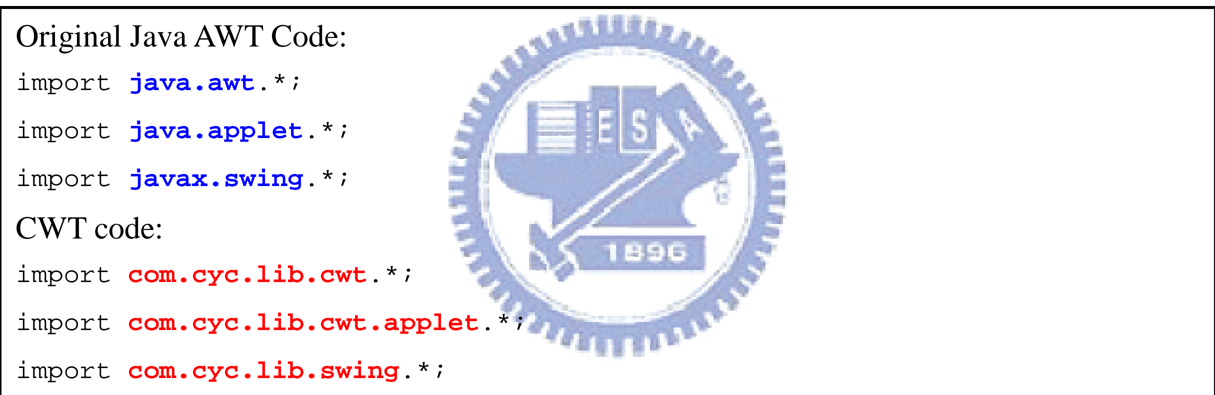
E S A
1896

Appendix C　Porting Guide

Before port Java programs to CWT, check that the Java GUI code only access 1.1 functionalities.

C.1 Import Statements

Simply change the `import` statements that use Java AWT (`java.awt`) to use CWT (`com.cyc.lib.cwt`). Then, recompile the programs.



Original Java AWT Code:

```
import java.awt.*;
import java.applet.*;
import javax.swing.*;
```

CWT code:

```
import com.cyc.lib.cwt.*;
import com.cyc.lib.cwt.applet.*;
import com.cyc.lib.swing.*;
```

C.2 Double buffering

CWT internally implements double-buffering technology. Therefore, for better rendering performance, code for double rendering can be removed.

Original Java AWT Code:

```
public void paint(Graphics g) {
    Dimension d = getSize();
    if((offImage == null) ||
       (d.width != offImage.getWidth(this) ||
```

```
        (d.height != offImage.getHeight(this))) {
        //Create offscreen buffer
        offImage = createImage(d.width, d.height);
        offGraphics = offImage.getGraphics();
    }
    //Perform rendering using offGraphics
    //Copy offscreen buffer to screen
    g.drawImage(offImage, 0, 0, this);
}
```

CWT Code:

```
public void paint(Graphics g) {
    //Perform rendering using g
}
```

C.3 Active Rendering

In each normal repaint, CWT copies altered areas of the off-screen buffer to screen. In the case of active rendering, however, CWT will not copy the altered content to screen. Therefore, in this case, a new method `flip()` introduced in the `Graphics` class should be called to trigger the repaint procedure on all the components atop the altered area, and copy the area to screen.

Original Java AWT Code:

```
Graphics g = component.getGraphics();
//perform rendering
```

CWT Code:

```
Graphics g = component.getGraphics();
//perform rendering
g.flip();
```

C.4 <applet> Tag in Html

CWT wraps Java Applet. Therefore, when using Java Applet, the html code which launches your programs should be modified first so that `CwtApplet` can launch your Applet programs.

Original Java AWT Code:

```
<applet code=YourApplet.class>
</applet>
```

CWT code:

```
<applet code=com.cyc.lib.cwt.applet.CwtApplet.class>
    <param name="cwtapplet" value="YourApplet">
</applet>
```

C.5 JavaScript

Since Applet now is wrapped in CWT, new methods are provided to obtain the Applet instance for accessing JavaScript. The Applet instance can be reached by calling getAppletContext().getNativeApplet().

Original Java AWT Code:

```
import java.applet.Applet;
import netscape.javascript.JSObject;
public class YourApplet extends Applet
{
    void foo()
    {
        JSObject jso =JSObject.getWindow(this);
        //Use jso to call JavaScript
    }
}
```

CWT Code:

```
import com.cyc.lib.cwt.applet.Applet;
import netscape.javascript.JSObject;
public class YourApplet extends Applet {
    void foo() {
        JSObject jso =JSObject.getWindow(getAppletContext().
                                          getNativeApplet());
        //Use jso to call JavaScript
    }
}
```

C.6 CWT Implementations

Programmers can specify the system property "cwt. toolkit" to assign one of three implementations to run.

- **CWT-AWT**
  -Dcwt.toolkit=com.cyc.lib.cwt.impl.awt.AwtToolkit
- **CWT-GL**
  -Dcwt.toolkit=com.cyc.lib.cwt.impl.fbo.GlToolkit
- **CWT-DX**
  -Dcwt.toolkit=com.cyc.lib.cwt.impl.dx3.Dx3Toolkit

Programmers can also specify the applet parameter "cwt. toolkit" as follows.

- **CWT-AWT**
  <param name="cwt.toolkit" value="com.cyc.lib.cwt.impl.awt.AwtToolkit">
- **CWT-GL**
  <param name="cwt.toolkit" value="com.cyc.lib.cwt.impl.fbo.GlToolkit">
- **CWT-DX**
  <param name="cwt.toolkit" value="com.cyc.lib.cwt.impl.dx3.Dx3Toolkit">

C.7 More Useful APIs

CWT adds some methods to AWT API for giving programmers more abilities to manipulate game objects.

- `com.cyc.lib.cwt.Component`

    `void setOpaque(boolean b)`

    If true the components background will be filled with the background color. Otherwise, the background is transparent, and whatever is underneath will show through. The default value is false. This attribute will affect the background of most components: Canvas, Checkbox, Choice, Label, List, Scrollbar, ScrollPane, Panel, TextArea and TextField.

    `boolean isOpaque()`

    Check if the component background is transparent or not.

    `void setOpacity(float opacity)`

    Make the components be drawn in translucent mode and whatever is underneath will show through. Opacity value is between 0.0 (totally transparent) to 1.0 (totally opaque). The default value is 1.0.

    `float getOpacity()`

    Get opacity value of this component. The value is between 0.0 (totally transparent) to 1.0 (totally opaque).

- `com.cyc.lib.cwt.Window`

    `void addGLEventListener(Object listener)`

    Add a `javax.media.opengl.GLEventListener` to CWT-GL internal `GLCanvas`. If multiple listeners are added to the `GLCanvas`, they are notified of events in an arbitrary order

- `com.cyc.lib.cwt.impl.dx3.Dx3Toolkit`

    `static com.ms.directX.DirectDraw getDirectDraw()`

    Access the DirectDraw object 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.

- `com.cyc.lib.cwt.impl.dx3.Dx3Image`

    `com.ms.directX.DirectDrawSurface getDirectDrawSurface()`

    Get DirectDraw surfaces of CWT images to handle the pixels of the images.

- `com.cyc.lib.cwt.impl.fbo.GlImage`

    `void setPriority(float priority)`

    Set priority of images to minimize texture memory thrashing. The value is between 0.0 and 1.0.

- `com.cyc.lib.cwt.impl.fbo.GlGraphics`

    `static void setGlyphFontSizeThreshold(int fontSize)`

    Set the threshold of font size for enabling geometry-based rendering. If rendered font size is bigger than or equals to the threshold, the text engine uses geometry-based rendering. Otherwise, use texture-based rendering. Default value is 28.

    `static void setTextCacheSize(int bytes)`

    Specify the maximum memory size for text cache. Default value is one megabyte.

Vita

Yi-Hsien Wang was born in Yunlin, Taiwan in 1977. He received the B.S., M.S. and Ph.D. degree in Computer Science and Information Engineering from National Chiao Tung University, Hsinchu, Taiwan, in 2000, 2002 and 2009, respectively. He was also a visiting scholar at Department of Electrical Engineering, University of Washington, Seattle, from 2007 to 2008. He worked for ThinkNewIdea Inc. from 2000 to 2009. His research interests include Internet gaming technology and software engineering.