國 立 交 通 大 學

資 訊 工 程 學 系

碩 士 論 文

CWT — 可支援不同繪圖函式庫的 AWT 實作

CWT — The AWT API over Different Graphics Libraries

研 究 生：姜智耀　　Student：Jyh-Yaw Jiang

指導教授：吳毅成　　Advisor：I-Chen Wu

中華民國九十四年六月

CWT — 可支援不同繪圖函式庫的 AWT 實作
CWT — The AWT API over Different Graphics Libraries

研 究 生：姜智耀　　Student：Jyh-Yaw Jiang

指導教授：吳毅成　　Advisor：I-Chen Wu

國 立 交 通 大 學
資 訊 工 程 系
碩 士 論 文

A Thesis
Submitted to Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in

Computer Science and Information Engineering

June 2005

Hsinchu, Taiwan, Republic of China

中華民國九十四年六月

CWT — 可支援不同繪圖函式庫的 AWT 實作

研究生：姜智耀　　指導教授：吳毅成

國立交通大學 資訊工程學系 碩士班

摘要

抽象視窗工具組(Abstract Window Toolkit，以下簡稱 AWT)是一套在 Java 中專門負責開發圖形使用者介面(GUI)的 API。基本上，AWT 可以分為兩個部分，使用者介面工具組(UI Toolkit)與繪圖工具組(Graphics Toolkit)。繪圖工具組這個部分，包含基本的影像處理與繪圖功能，也是影響 Java 繪圖效能最關鍵的部分。與繪圖加速卡有著密切關係的繪圖函式庫目前有兩大主流，分別為業界開放標準的 OpenGL，以及微軟的 DirectX。Java 的繪圖效能不彰，主要來自虛擬機器(Virtual Machine)本身的瓶頸。如何突破這個瓶頸，獲得硬體加速的直接支援，繪圖函式庫的支援將會是關鍵性的因素。

為了能提升 Java 的繪圖效能，並使許多現有的 Java 程式可以因此而受惠。本論文提出了一套開放式的架構，稱為 CYC Window Toolkit(簡稱 CWT)，以 AWT 為共同的 API，並針對繪圖工具組的這個部分作改善，使其可以採用不同繪圖函式庫來實作。並達成以下三項特性：1.提升繪圖效能(Performance Improvement)、2.向下相容性(Backward Compatible)、3.可移植性(Portability)。

CWT — The AWT API over Different Graphics Libraries

Student：Jyh-Yaw Jiang　　Advisor：I-Chen Wu

Department of Computer Science and Information Engineering
National Chiao Tung University

Abstract

Abstract Window Toolkit (AWT) is one of API in Java, which providing programmers to build Graphical User Interface (GUI). Basically, AWT has two parts, UI Toolkit and Graphics Toolkit. Graphics Toolkit includes basic functionalities for drawing and image processing; it's also the key part to affect Java rendering performance. Graphics library is close coupled with graphics cards. In the present, this graphics library has two main streams, OpenGL and DirectX. OpenGL is open standard in industry. DirectX is proposed by Microsoft. Java's bad rendering performance is due to the bottleneck of Virtual Machine. How to break through this bottleneck to get hardware-accelerated rendering support, graphics library will be the key role。

In order to improve Java rendering performance and let most existed Java program benefit from this. This thesis proposes an open architecture, called CYC Window Toolkit (CWT), which lets AWT API be the common interface and makes the Graphics Toolkit of AWT supported different implementations over different graphics libraries。And finally have this three features：1. Performance Improvement, 2. Backward Compatible, 3. Portability.

誌謝

首先，我要感謝我的指導教授 — 吳毅成教授，因為他的細心指導，不辭辛勞地給予指正，這篇論文才得以順利完成。以及我的碩士論文口試委員們 — 莊榮宏教授、郭耀煌教授、許舜欽教授，感謝他們細心地審查我的論文，並且提供許多寶貴的意見。

另外我還要特別感謝汪益賢學長，協助並指導本論文的系統架構設計，如果沒有學長的指導，本論文一定無法完成。每當我遇到實作瓶頸時，汪益賢學長總是能耐心地協助我釐清問題的癥結，並且與我一起討論出解決的方法，因此我才能夠順利地完成論文。

目錄

表目錄

圖目錄

第一章、緒論

本章的一開始在 1.1 節將對本論文的研究背景做個簡單的介紹，討論 Java 這個程式語言應用在遊戲軟體開發上的可行性，1.2 節中會介紹 Java 這個程式語言在生產力(Productivity)與效能(Performance)上的優劣，1.3 節則進一步介紹各種有助於提升 Java 繪圖效能(Rendering Performance)的各種繪圖函式庫，在 1.4 節中會介紹本論文希望解決的問題與設計目標，最後在 1.5 節中會介紹本論文的內容大綱。

1.1 背景介紹

數位內容產業，隨著政府積極推動「新世紀兩兆雙星產業發展計畫」而開始被重視，將有助於改善國內長久以來軟硬體發展不均衡的現象，並可藉此提升我國整體產業的全球競爭力。

數位內容（Digital Content）係指「將圖片、文字、影像、語音等運用資訊科技加以數位化並整合運用之產品或服務」。包含「數位遊戲」、「電腦動畫」、「數位學習」、「數位影音應用」、「行動應用服務」、「網路服務」、「內容軟體」、「數位出版典藏」、「數位藝術」等九大領域。[34]

「線上遊戲」(Online Game)，因為「天堂」[37]這款遊戲的成功，近年來在國內蓬勃發展。在眾多數位內容相關產業中，可以算是知名度最高，也是最廣為一般大眾所熟悉的一項新興產業。基本上這亦屬於「數位遊戲」這個領域的一部分。

「數位遊戲」基本上包含以下四類：

1. 電腦遊戲 (PC Game)

2. 遊樂器遊戲 (Console Game)
   例如 PS2[16]、Xbox[13]、GameCube[14]。

3. 可攜式遊戲 (Portable Game)
   例如 PSP[16]、NDS[14]、GameBoy[14]、PDA。

4. 大型機台遊戲 (Arcade Game)

電腦遊戲，因為國內獨特的文化背景，所以在國內遊戲產業中有著舉足輕重的地位。目前在國內最熱門的線上遊戲也僅侷限於電腦遊戲，接下來將進一步介紹這一個部分。

1.1.1 電腦遊戲

電腦遊戲除了一般的「單機遊戲」之外，還有需要連上網路才能玩的「線上遊戲」，而「線上遊戲」又可以再細分為兩種：[33]

1. 撮合線上遊戲 (Match Making Game)
   遊戲伺服器主要扮演撮合玩家對戰的角色，提供數人或數十人同時進行互動。例如「戲谷麻將館」[32]、「CYC 遊戲大聯盟」[35]。這類遊戲多半是訴求輕鬆休閒的輕量級遊戲，所以也被稱為「休閒遊戲」(Casual Game)。

2. 多人線上遊戲 (Massive Multi-player Online Game，MMOG)
   遊戲伺服器可以提供成千上萬的玩家同時進行互動，遊戲有故事及時空背景，可以讓玩家進行角色扮演。例如「天堂」、「仙境傳說」[36]。

隨著個人電腦的普及，電腦遊戲開始在市場上佔有一席之地。而隨著網際網路的發達，線上遊戲更是突然竄紅，幾乎成為廠商開發遊戲時不可或缺的選擇。

然而根據資策會的概略統計，市面上有 75%的線上遊戲來自韓國，包含「仙境傳說」、「天堂」這兩款超人氣的線上遊戲。歐、美、日佔 10%，國內的自製遊戲只佔了 15%。面對國外廠商的大舉入侵，國內廠商亦面臨了極大的威脅，如何提升遊戲軟體開發的競爭力，將成為國內遊戲產業非常重要的一項課題。[33]

1.1.2 遊戲軟體開發

遊戲軟體目前在開發時大多用 C/C++來撰寫，甚少採用 Java。儘管 Java 本身有物件導向語言設計上的優勢，能有效縮短軟體的開發時程，減少後續的維護成本，提高軟體開發人員的生產力。但是因為 Java 本身在繪圖效能上的瓶頸，導致 Java 應用在遊戲軟體開發上的可行性並不高，通常只能侷限於開發輕量級的「休閒遊戲」(Casual Game)。

回顧過去的歷史，當大部分的軟體開發人員都還在用組合語言的時候，C 語言的出現對他們來說還是不夠快。但是直到 1993 年 ID Software 推出了毀滅戰士(Doom) 之後，這款幾乎完全用 C 語言開發的遊戲扭轉了所有人對 C 語言的看法。而現在已經沒有軟體開發人員會再完全地用組合語言來開發遊戲了。[5]

對於遊戲軟體開發來說，生產力(Productivity)和效能(Performance)一樣重要。如圖 1-1 所示，從組合語言到 C/C++，效能是可以被接受的，而且因為 C/C++其高階語言的優勢，其生產力可以大幅度地提升。所以與組合語言相比，C/C++可以在更短的時間內開發更複雜的遊戲軟體。

然而與 C/C++相比，Java 其實有著更多高階語言上的優勢，因此，接下來將進一步介紹 Java 在生產力與效能上的優劣，以探討其應用在遊戲軟體開發上的可行性。至於.Net 平台[3]雖然也是可以探

討的一個項目，但是由於這是另一個不小的主題，所以並不是本論文所討論的重點。然而因為.Net 中的 J#幾乎可以被視為和 Java 是一樣的程式語言，所 Java 的部分若是能獲得解決，.Net 的 J#其實也能因此而受益。[29]

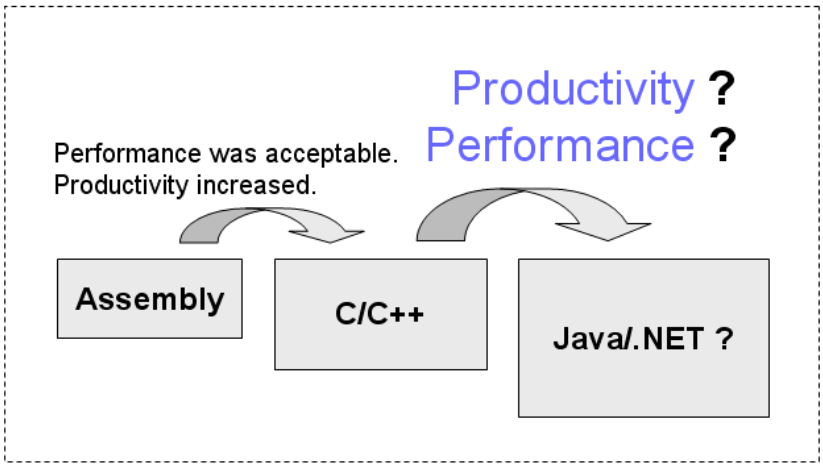


圖 1-1 : Java 或.Net 應用在遊戲軟體開發上的機會。

1.2 Java

Java 的出現，最早是為了解決消費性電子產品需求而設計的一套全新語言。在 1995 年，由昇陽(Sun Microsystems，以下簡稱 SUN)正式將它應用於全球資訊網(World Wide Web)。Java 在設計上最主要的一項考量就是希望能獨立於所有平台之上，達到跨平台的目標。然而 Java 在物件導向語言設計上的優勢，亦讓 Java 成為非常熱門的新一代程式語言。

1.2.1 生產力(Productivity)

如何有效縮短軟體的開發時程，減少後續的維護成本，提高軟體開發人員的生產力，這一直是軟體開發上一項非常重要的課題。Java 在這方面的優勢，早已經被證實。根據國際研究機構 IDC 在 1998 年 5 月的一項研究報告中指出，與 C/C++相比，完全採用 Java 來開發

的軟體專案，的確可以縮短軟體的開發時程，減少後續的維護成本，提高軟體開發的生產力。表 1-1 為該研究報告所提出的數據資料：

	時間/成本 節省	生產力提升
開發階段	40%	67%
維護階段	30%	42%
整體	25%	

表 1-1：與 C/C++相比，完全採用 Java 來開發的軟體專案。

資料來源： "Java Technology Pays Positively ", IDC, 1998 年五月[5][15][17]

1.2.2 效能(Performance)

關於 Java 的效能部分，基本上可以分為執行效能和繪圖效能這兩個部分來探討。這一節將只討論前者，後者由於是本論文討論的重點，所以則留到下一節作深入的介紹。一般人普遍的印象總認為 Java 的執行效能並不好，但是隨著 SUN 不斷地釋放出更新版本的 Java，這個問題逐漸地被改善，甚至有已經機會跟 C/C++匹敵了。以下將引用一份技術報告[5]的資料來說明(表 1-2)，Java 的執行效能的確已經有所改善。尤其是在對程式碼的部分作某種程度的最佳化之後，改善的效果將更為明顯。

JDK/JRE 版本	比 C++ 慢的程度係數 (Factors slower than C++)
1.0	20 – 40
1.1	10 – 20
1.2	1 – 15
1.3	0.7 – 4
1.4	0.5 – 3

表 1-2：與 C/C++相比，Java 的執行效能。

事實上，Java 的執行效能可以有如此著顯著的改善，Sun 早期提出的即時編譯技術(Just-In-Time Compilation，簡稱 JIT)[22]和後來的熱點編譯技術(HotSpot compilation，簡稱 HotSpot)[24]，功不可沒。

此外，對於繪圖效能的部分，SUN 也終於在最新版本的 Java 中提出了硬體支援的解決方案，這部分和繪圖函式庫息息相關，也是本論文的重點議題，因此將在下一節中作專門而深入的介紹。表 1-3 整理出 Java 不斷演進的過程。[5][19][22][23][24][20]

JDK/JRE 版本	描述	使用者介面 (User Interface)	正式推出時間 (Release Time)
1.0	簡單的直譯方式 (A simple interpreter)	提出 AWT 1.0	1996
1.1	提出 **JIT** 編譯技術	提出 AWT 1.1 和 Swing	1997
1.2	轉換為 Java 2 平台以及成熟的 **JIT** 編譯技術	提出 Java 2D	1998
1.3	提出 **HotSpot** 編譯技術		2000
1.4	成熟的 **HotSpot** 編譯技術	支援 **DirectX** 的硬體加速繪圖	2002
5.0 (1.5)		支援 **OpenGL** 的硬體加速繪圖	2004

表 1-3 : Java 的演進過程。

1.3 Java 的繪圖函式庫

電腦硬體不斷推陳出新，而繪圖加速卡在最近幾年更是進步神速，也讓電腦遊戲軟體在開發時充滿了更多的可能性，視覺上的效果將可以更加地多采多姿。因此，與繪圖加速卡關係密切的繪圖函式庫，在遊戲軟體的開發上將扮演舉足輕重的角色。

繪圖函式庫目前有兩大主流，OpenGL[28]是業界的開放標準，其跨平台的特性使其可被使用在不同的作業系統上；而 DirectX[10]雖然只能用在微軟視窗作業系統(Microsoft Windows)上面，但是微軟視窗作業系統的龐大市佔率，已為其奠定了良好的使用基礎。

Java 的繪圖效能不彰，主要來自虛擬機器(Virtual Machine，以下簡稱 VM)本身的瓶頸[18]。如何突破這個瓶頸，獲得硬體加速的直接支援，繪圖函式庫將會是關鍵性的因素。而在 Java 中負責繪圖部分的抽象視窗工具組(Abstract Window Toolkit，以下簡稱 AWT)[26]亦將扮演一個非常重要的角色。

因此，接下來的幾個章節將進一步討論 Java 可以使用的各種繪圖函式庫。首先在 1.3.1 節將先介紹 Java 的基本繪圖函式庫，並介紹其發展過程。而在 1.3.2 節將介紹非 Java 基本套件的其他繪圖函式庫，這些繪圖函式庫因為可以獲得硬體的支援，所以對於繪圖效能的提升有一定的幫助。最後在 1.3.3 節將介紹微軟視窗平台上特別提供給 Java 用的繪圖函式庫，雖然這個繪圖函式庫僅能在微軟的視窗平台上使用，但是因為可以獲得 DirectX 的硬體支援，所以仍然是一項值得考慮的解決方案。

1.3.1 Java 的基本繪圖函式庫

抽象視窗工具組(Abstract Window Toolkit，以下簡稱 AWT)是一

套在 Java 中專門負責開發圖形使用者介面(Graphical User Interface)的 API。基本上，AWT 可以分為兩個部分，一個部分是使用者介面工具組(UI Toolkit)，而另一個部分是繪圖工具組(Graphics Toolkit)。

使用者介面工具組這個部分，包含視窗元件的建立，以及事件的處理機制。AWT 的重型元件(Heavy-weight Component)，是由不同平台的視窗系統所生成，以提供一致的呈現。之後推出的 Swing[6]，基本上仍然是建構在 AWT 之上，除了提供更多的元件之外，最重要的是全部改為自行繪製的輕型元件(Light-weight Component)，因此將十分仰賴 AWT 在繪圖工具組這個部分的繪圖效能。

繪圖工具組這個部分，包含基本的影像處理與繪圖功能，屬於 Java 的基本繪圖函式庫，也是影響 Java 繪圖效能最關鍵的部分。JDK 1.2 推出的 Java 2D[9]，在與 AWT 相容的前提下，完全接管了這個部分，並提供更強大的影像處理與繪圖功能，如圖 1-2。然而為了提供 Java 2D 的眾多新功能，JDK 1.2 不得不採用軟體實作。因此繪圖效能不但沒有改善，反而更糟。

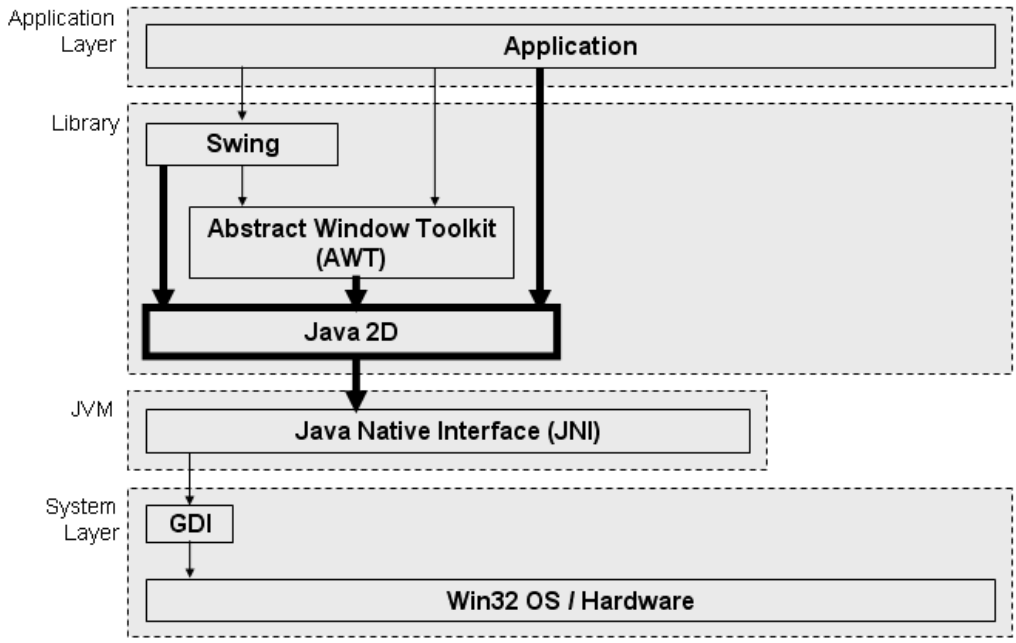


圖 1-2 : JDK 1.2 版之後推出的 Java 2D。

終於到了 JDK 1.4，繪圖效能才又有了突破性的改善，在 Win32 平台上採用 DirectX 來獲得硬體支援[24][18]。然而這樣的改善結果卻依然成效不彰，除了程式必須改寫之外(必須使用 VolatileImage[27] 才行)，對於透明圖和不透明圖在繪圖效能上的改善程度也不盡相同。而 JDK 1.5 雖然又增加了可以採用 OpenGL 來獲得硬體支援[20]，但是目前為止其硬體的相容性很差，時常無法啟用，很多時候就算啟用了卻也沒什麼效果可言。圖 1-3 和圖 1-4 分別為 Java 2D 支援 DirectX 和 OpenGL 的圖示說明。

很明顯地，Java 的基本套件在繪圖效能上的改善結果始終不是很理想，這也是 Java 長久以來一直為人所詬病之處。然而 Java 並不是沒有機會的，下一節將介紹其他非 Java 基本套件的繪圖函式庫，看看這些額外的套件如何改善 Java 的繪圖效能。

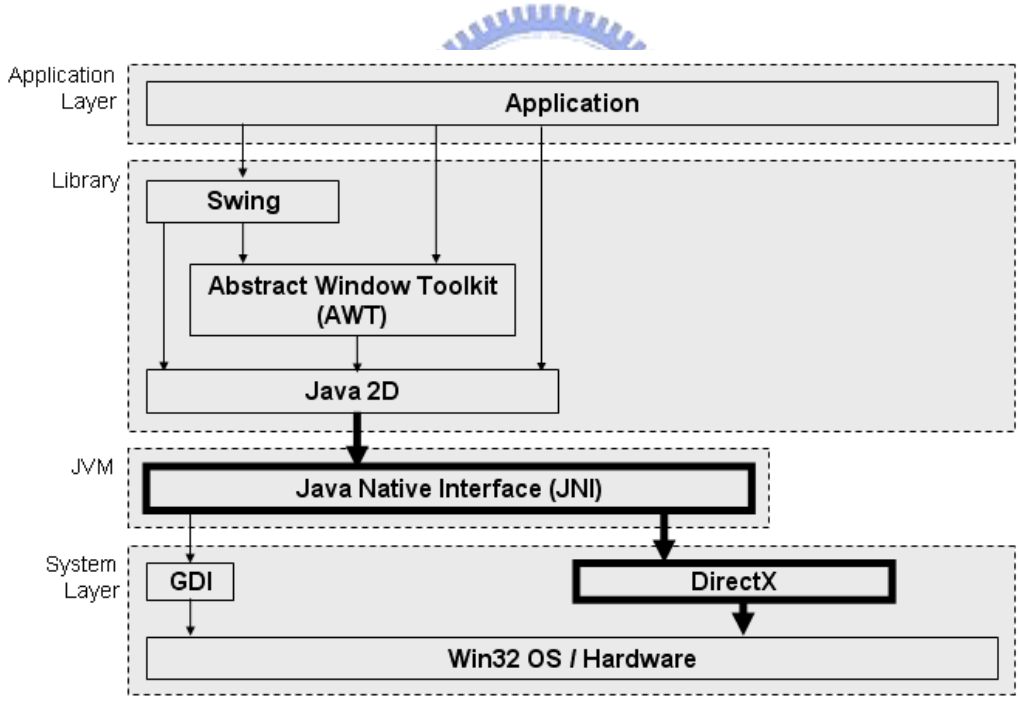


圖 1-3 : JDK1.4 版之後，Java 2D 採用 DirectX 支援。

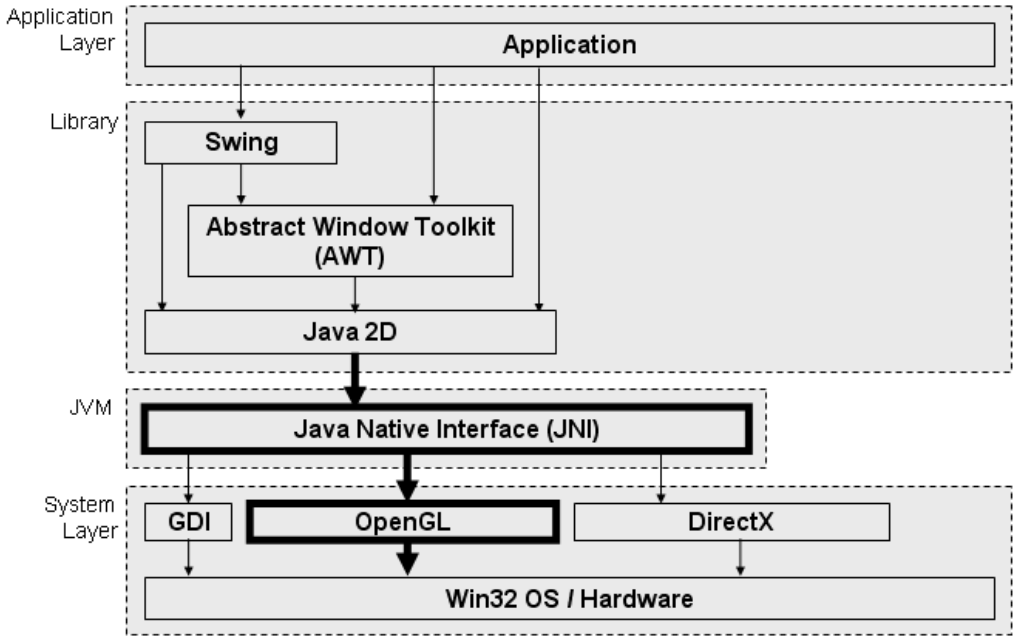


圖 1-4：JDK1.5 版之後，Java 2D 採用 OpenGL 支援。

1.3.2 Java 的其他繪圖函式庫

除了 Java 內建的基本套件之外，如圖 1-5 所示，其實透過 Java 的原生介面(Java Native Interface，以下簡稱 JNI)[19]，也是有機會可以穿過 VM 直接使用作業系統提供的各種函式庫，這當然也包含了一般的繪圖函式庫，像是 DirectX 和 OpenGL。雖然這的確有機會可以提升 Java 的繪圖效能，但是 採用此方式唯一的麻煩是必須自行建立一對一的原生實作，好讓 Java 程式可以使用作業系統提供的各種函式庫。此外也將會因此而喪失 Java 跨平台的優勢，不然就是必須盡可能地在各種平台上完成不同的原生實作來彌補這個缺點。

在 JDK 1.2 之後，進入了 Java 2 的時代。這時由於「AWT Native Interface」[26]使用文件的公佈，使得這類解決方案又多了一項實作方式，各種作業系統提供的繪圖函式庫可以在不影響效能的情況下，透過 Java Native Interface(JNI)直接在 AWT 元件上進行繪圖的動作。

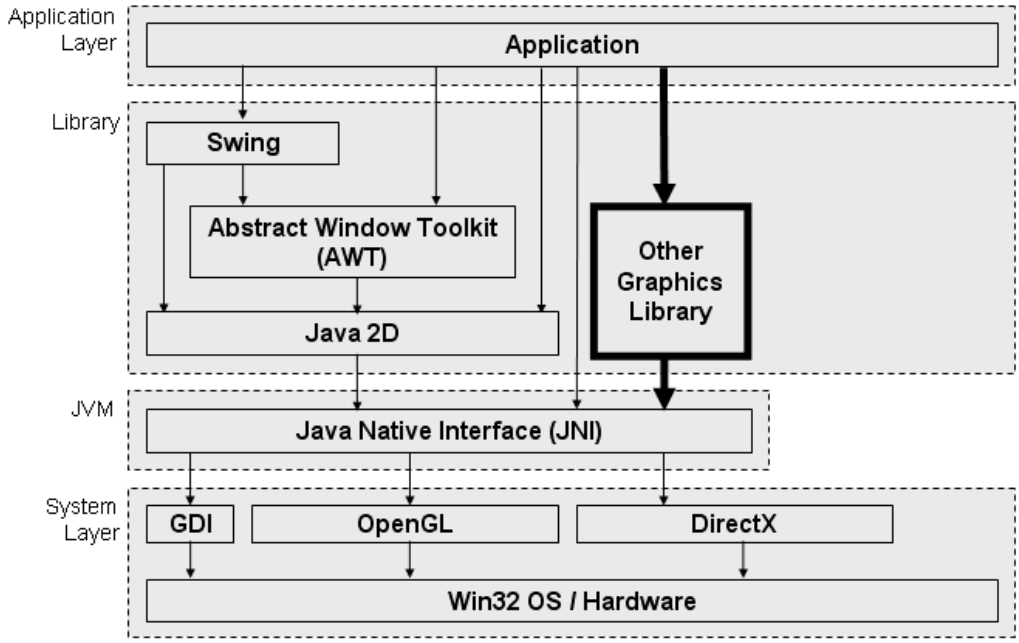


圖 1-5：採用 JNI 的其他 Java 繪圖函式庫。

想要透過 JNI 自行建立完整的一對一的原生實作，其實並不是件簡單的事。然而值得慶幸的是，目前這類型的解決方案已經有許多現成的套件可以使用了，想提升繪圖效能還是有機會的，如表 1-4 所列。

簡稱	函式庫名稱	原生支援	贊助單位
SWT	Standard Widget Toolkit [4]	GDI	IBM
JOGL	Java bindings for OpenGL API [21]	OpenGL	SUN
LWJGL	Light-Weight Java Gaming Library [2]	OpenGL	Source Forge

表 1-4 : Java 的其他繪圖函式庫。

1.3.3 微軟視窗平台上的繪圖函式庫

除了 Sun 推出的 Java 虛擬機器之外，另外特別值得一提的就是微軟的 Java 虛擬機器(Microsoft Java Virtual Machine，以下簡稱 MSVM)。雖然 MSVM 僅支援 JDK 1.1，但是因為其內建於微軟的瀏覽器 Internet Explorer(IE)，所以也讓它成為一般使用者最容易接觸到的 Java 虛擬機器。

此外，微軟為了讓 MSVM 可以跟作業系統有更緊密的結合，以便開發更多更有價值的應用程式，特別推出了一整套的解決方案，統稱為 Microsoft SDK for Java[11]。如圖 1-6 所示，其中的 Microsoft Language Extension 更提供了可以直接使用 DirectX 的 API，這一點對於 Java 在繪圖效能的改善上有著莫大的助益。雖然這將會破壞 Java 跨平台的特性，但是因為微軟視窗作業系統擁有龐大的市佔率，亦使 MSVM 在繪圖效能改善上的可能性有著不可或缺的影響力。

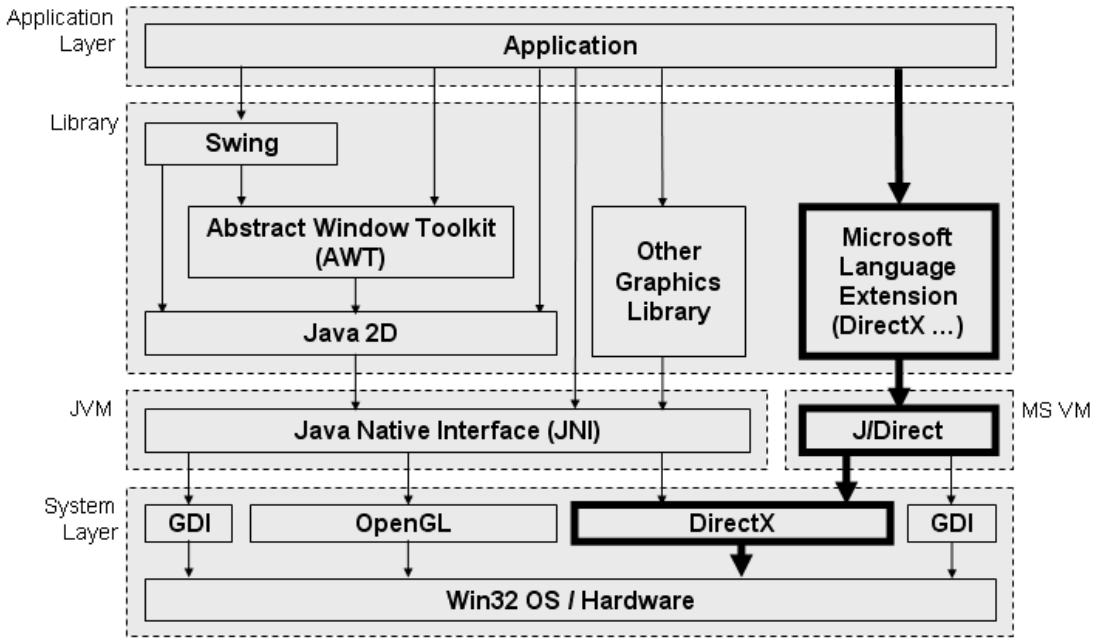


圖 1-6 : MSVM 上可以使用 DirectX 的方式。

1.4 設計目標

想讓 Java 的繪圖效能可以獲得硬體加速的支援，目前其實存在著許多不同的解決方案[1][30]。在 Java 2 平台上有兩套繪圖函式庫可以支援 OpenGL，分別為 SUN 的 JOGL(Java bindings for OpenGL API)，以及 Source Forge 的 LWJGL(Light-Weight Java Gaming Library)。此外，在 MSVM 上面也有一套 Microsoft Language Extension 可以支援使用 DirectX。雖然這些繪圖函式庫各有優缺點，但是也證明了 Java 的繪圖效能是其實是有機會可以改善的。

為了能提升 Java 的繪圖效能，並使許多現有的 Java 程式可以因此而受惠。本論文提出了一套開放式的架構，如圖 1-7 所示，稱為 CYC Window Toolkit (簡稱 CWT)，以 AWT 為共同的 API，並針對繪圖工具組這個部分作改善，使其可以採用不同繪圖函式庫來實作。並達成以下三項特性：1.提升繪圖效能(Performance Improvement)、2.向下相容性(Backward Compatible)、3.可移植性(Portability)。

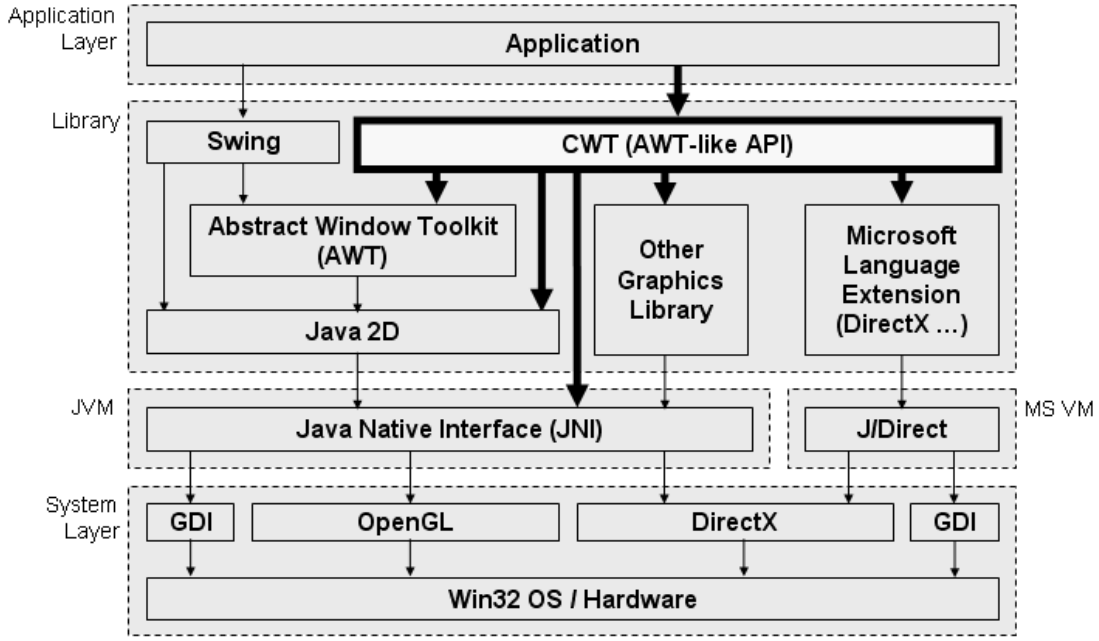


圖 1-7 : CWT (CYC Window Toolkit)。

1.5 本文大綱

本論文的第一章介紹了 Java 在繪圖效能改善上的相關背景說明，並提出本論文的解決方案以及預期達成的目標。第二章將針對本論文提出的系統設計架構做詳細的說明。第三章將討論實作的內容，以及實際的效能評測，並針對實際案例的應用結果做說明。第四章將針對本論文提出的解決方案作一個總結，並進一步討論未來的發展方向。

第二章、系統設計

由於本系統與抽象視窗工具組(Abstract Window Toolkit，以下簡稱 AWT)息息相關，因此將先介紹 AWT 的內部架構，再說明本論文提出的 CYC Window Toolkit(以下簡稱 CWT)的設計架構。

2.1 AWT

AWT 可以分為兩個部分，使用者介面工具組(UI Toolkit)與繪圖工具組(Graphics Toolkit)。

然而若以類別封裝的角度來看，AWT 可以分為三大部分，元件模式(Component Model)、事件模式(Event Model)、繪圖模式(Painting Model)。之後幾節的介紹也將從這個角度來一一詳細說明。從圖 2-1 可以看出，這三個部分在主要功能上的分類，其中與繪圖工具組(Graphics Toolkit)相關的繪圖模式(Painting Model)部份將是影響繪圖效能的關鍵，也會是 CWT 與 AWT 最大的不同之處。

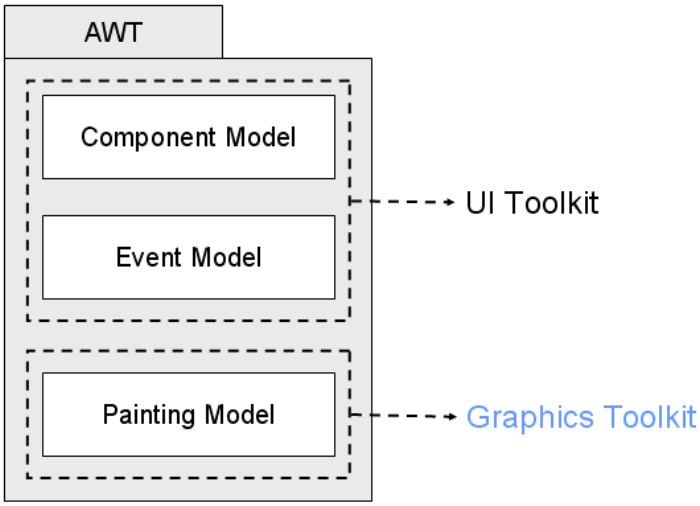


圖 2-1 : AWT 的基本組成架構。

2.1.1 元件模式(Component Model)

如圖 2-2 所示，AWT 的所有元件都是繼承 Component 這個類別，包含 Container 也是。Container 是可以用來裝元件的容器類別，如圖所示的視窗(由 Frame 這個類別產生)就是屬於 Container 的一種，裡面裝的三個按鈕(由 Button 這個類別產生)則是屬於 Component 的一種。

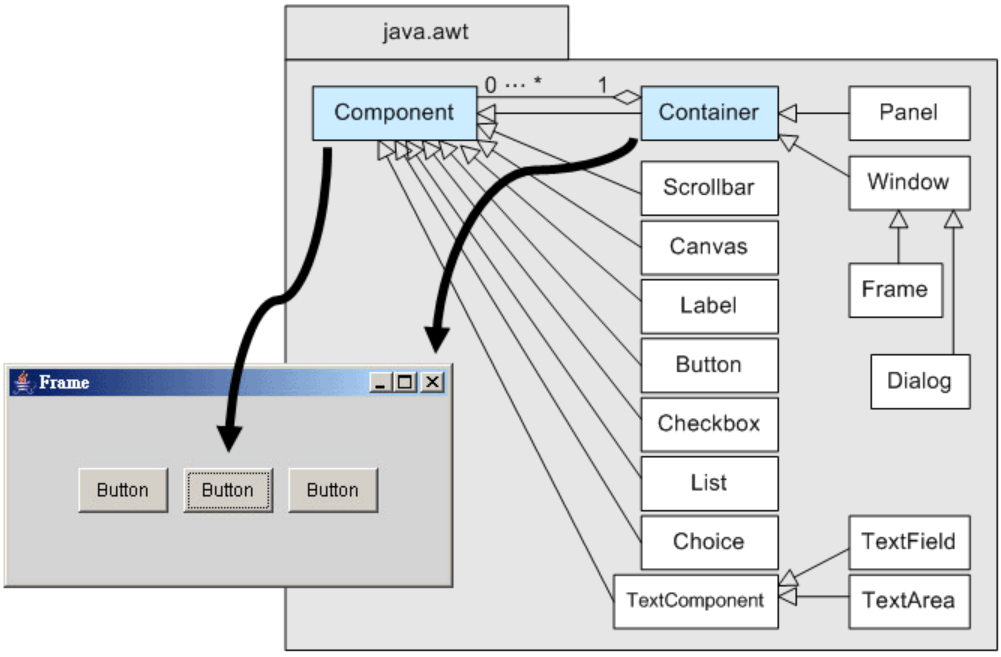


圖 2-2 : AWT 元件的階層式架構。

除了 Component 和 Container 這兩個基礎類別之外，其他的元件在 AWT 中都是屬於重型元件(Heavy-weight Component)，是由不同平台的視窗系統所生成，以提供與平台一致的呈現，所以又稱為原生元件(Native Component)。

除了重型元件(Heavy-weight Component)之外，AWT 也提供了輕型元件(Light-weight Component)，Component 和 Container 即是屬於這一類的元件。輕型元件指的就是該元件與系統平台無關，其呈現方式必須自行繪製，因此這部份將十分仰賴 AWT 的繪圖工具組，

並且將大幅考驗 Java 的繪圖效能。Swing 的絕大部分元件就是屬於這樣的輕型元件。雖然輕型元件是自行繪製的，但是也總要有地方可以被畫，所以重型元件就必須扮演這樣的角色。至少要有一個重型元件提供畫布，讓所有的輕型元件可以畫在上面。

2.1.2 事件模式(Event Model)

AWT 在 1.1 版之後提出了委派型事件模式(Delegation Event Model)的事件處理機制。如圖 2-3 所示，傾聽者(Listener)會向事件來源(Event Source)註冊，事件來源在有事件產生的時候就會將事件傳遞給所有註冊的傾聽者，傾聽者在收到事件後則會做出必要的反應。

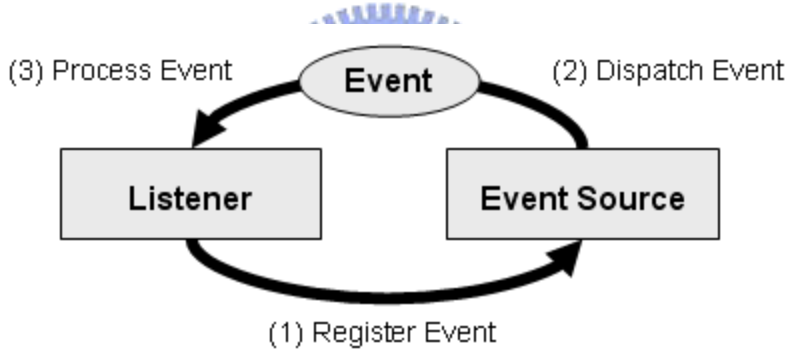


圖 2-3： 委派型事件模式(Delegation Event Model)。

如圖 2-4 所示，在 AWT 中有著各式各樣的事件，基本上都被歸類為 AWTEvent，其中與元件直接相關的 ComponentEvent 還可以再往下細分許多種。各種鍵盤和滑鼠的輸入事件，畫面更新的重繪事件與視窗顯示狀態變更的事件等等，這些都是提供 AWT 各種互動反應不可或缺的重要角色。

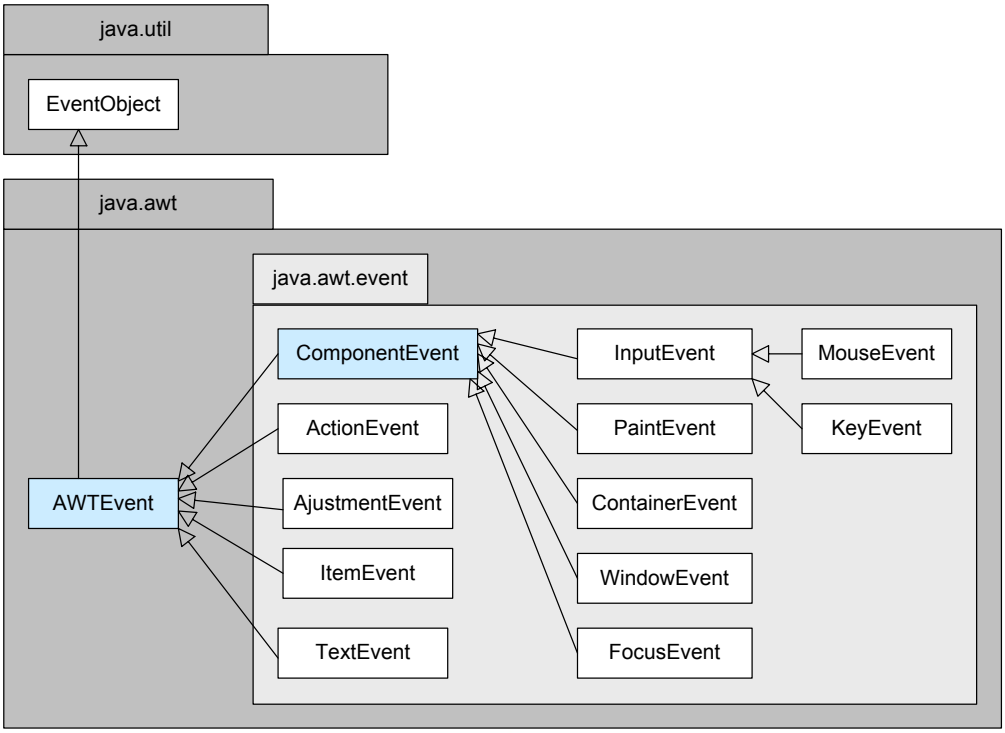


圖 2-4：AWT 的事件。

2.1.3 繪圖模式(Painting Model)

這個部分最重要的就是 Graphics 這個類別，Graphics 在 AWT 中扮演了畫布的角色，並提供了繪製影像、文字與圖形的基本功能。此外 Image 和 FontMetrics 則是分別和影像與文字的處理有關，與影像處理有關的部分主要都在 java.awt.image 這個套件中。至於如何取得 Image 物件，一般來說就要靠 Toolkit 這個核心類別了。因此，這幾個類別在 AWT 的繪圖工具組中都將扮演很重要的角色，包含 Graphics、Image、FontMetrics 和 Toolkit。

此外，AWT 中重繪的方式也與事件處理機制密不可分。基本上分為兩大類：系統驅動的重繪程序(System-triggered painting) [25] 和應用程式驅動的重繪程序(App-triggered painting) [25]。圖 2-5 以這兩種重繪程序為例，說明了 AWT 事件處理機制的運作方式。

系統驅動的重繪程序(System-triggered painting)與系統的重型

元件有密切的關係，當這類元件需要被重繪時，將由視窗平台上的事件分派執行緒(Native Event Dispatch Thread)直接呼叫該元件的paint()函式立刻進行重繪，例如元件被臨時擋住之後的顯示等等。

應用程式驅動的重繪程序(App-triggered painting)一般來說則是與輕型元件比較有關，需要重繪時就呼叫 repaint()這個函式來丟出重繪事件(Paint Event)。然後將由 AWT 內部的事件處理機制來來處理，事件佇列(Event Queue)負責存放這些事件，事件分派執行緒(Event Dispatch Thread)負責分派這些事件給相關的元件進行處理，除了重繪事件之外，其他的事件也是同樣的處理方式，例如鍵盤或滑鼠的事件等等。

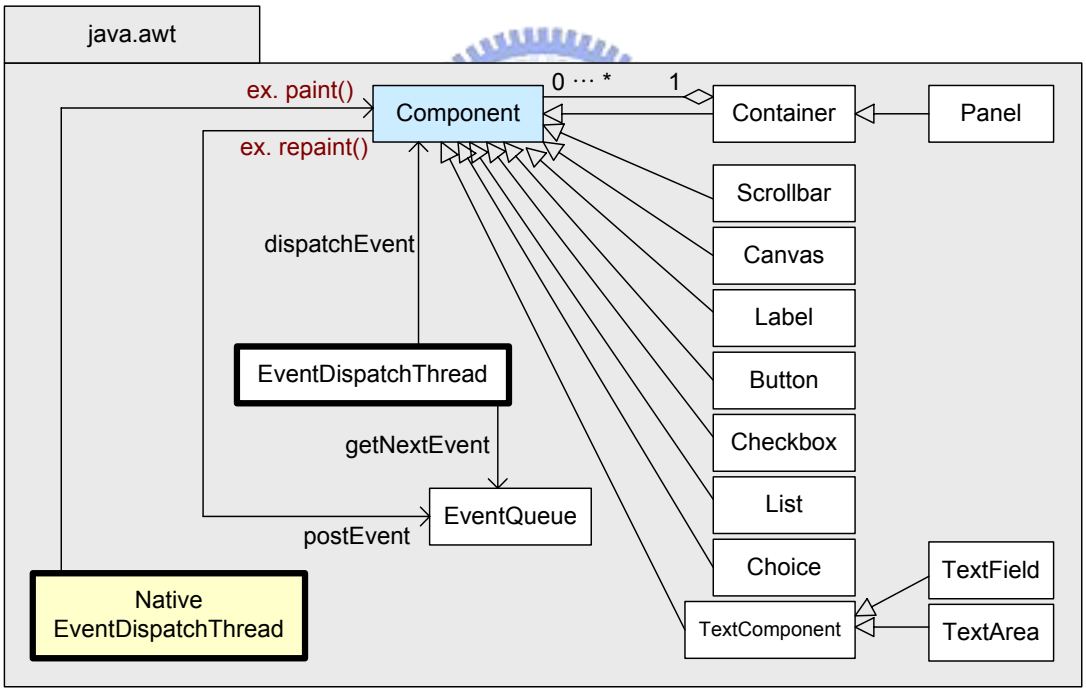


圖 2-5 : AWT 中的事件分配。

2.2 CWT

CWT 的主要目標就是提出了一套開放式的架構，以 AWT 為共同的 API，並針對繪圖工具組的這個部分作改善，使其可以採用不同繪圖函式庫來實作。

如圖 2-6 所示為 CWT 的巨觀架構，基本上與 AWT 一樣可以分為三種模式來看，以下簡單說明這三種模式與 AWT 最主要的不同之處。接下來幾個章節將分別進一步詳細說明這三種模式。

1. 元件模式(Component Model) :
   主要包含系統原生容器(Native Containers)與輕型元件(Light-weight Components)。

2. 事件模式(Event Model) :
   透過 EventManager 這個類別將系統原生容器(Native Containers)產生的事件傳遞給下面的輕型元件(Light-weight Components)。

3. 繪圖模式(Painting Model):
   與繪圖工具組密切相關的幾個類別將被抽象化以便支援不同的繪圖實作。包含 Toolkit、Graphics、Image 和 FontMetrics。

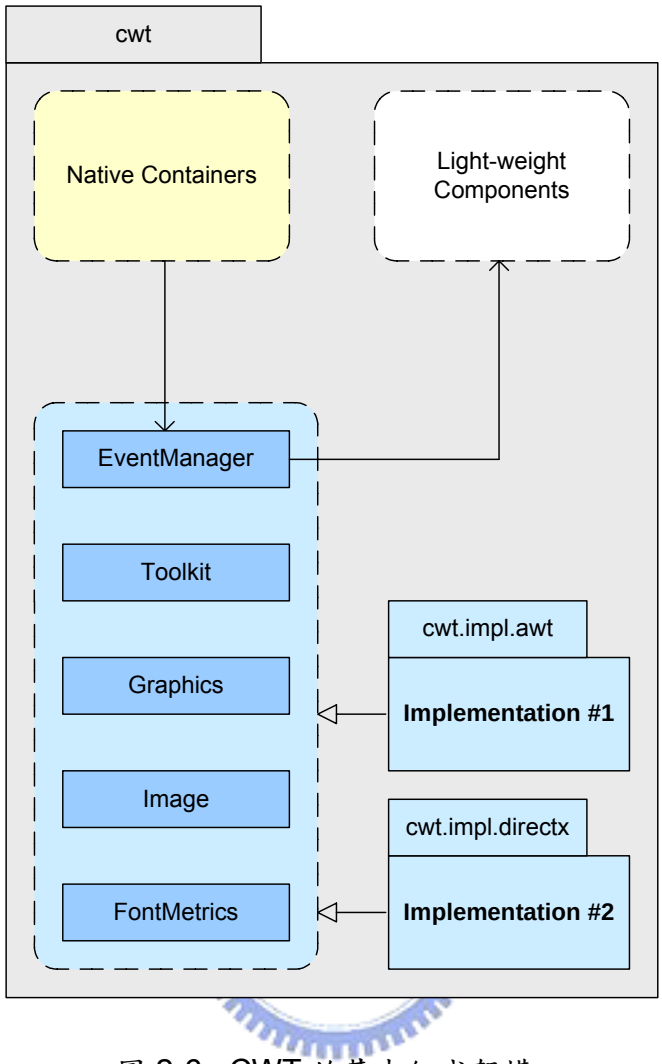


圖 2-6 : CWT 的基本組成架構。

2.2.1 元件模式(Component Model)

由系統原生容器(Native Containers)與輕型元件(Light-weight Components)所組成，這部分將和 Swing 一樣讓所有元件都是自行繪製，因此將會十分仰賴 CWT 的繪圖效能，然而這部份將是 CWT 重點改良的部分，如此以來所有元件也將能因此而受惠。

如圖 2-7 所示，在 CWT 中的系統原生容器(Native Containers)主要包含 Applet、Window、Frame 和 Dialog 這四個類別，其餘都是自行繪製的輕型元件(Light-weight Components)。系統原生容器(Native Containers)除了擔任畫布的角色，讓所有輕型元件

(Light-weight Components)可以畫在上面之外，也負責扮演事件來源(Event Source)的角色，這部份將在下一章節所討論的事件模式(Event Model)中詳細說明。

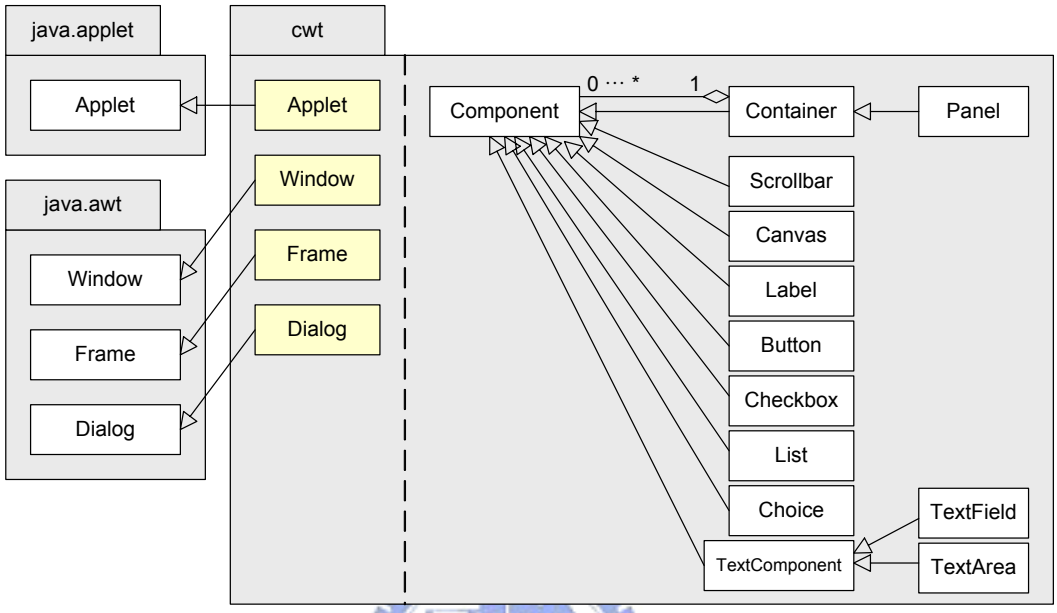


圖 2-7：CWT 元件的階層式架構

2.2.2 事件模式(Event Model)

CWT 訴求的是繪圖效能的改善，因此關於事件處理機制的部份將完全承襲 AWT。然而因為在 CWT 中，大部分元件都是屬於自行繪製的輕型元件(Light-weight Components)，所以必須仰賴系統原生容器(Native Containers)提供各種事件。為了將這些系統原生事件轉交給 CWT 內部的事件機制來處理，EventManager 類別就扮演了這樣一個中介的角色。

如圖 2-8 所示，EventManager 負責接收系統原生容器(Native Containers)的事件，並丟給 CWT 內部的事件佇列(Event Queue)存放，等待 CWT 內部的事件分派執行緒(Event Dispatch Thread)來分派給相關的輕型元件(Light-weight Components)進行處理。

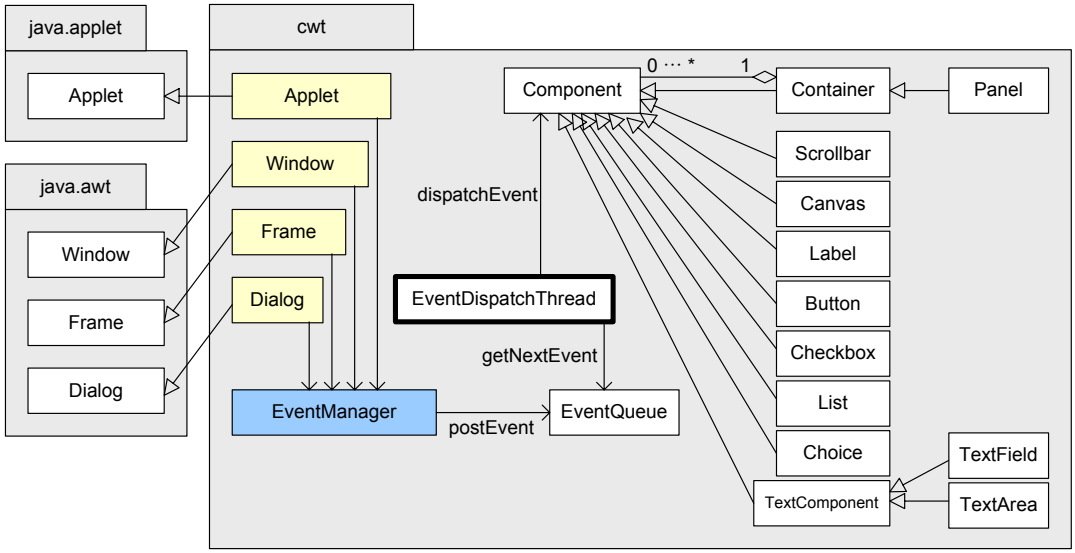


圖 2-8 : CWT 中的事件分派機制。

2.2.3 繪圖模式(Painting Model)

這個部分是 CWT 最重要的部分，之前提過在 AWT 的繪圖工具組中扮演非常重要角色的幾個類別，Graphics、Image、FontMetrics 和 Toolkit，這些都是影響繪圖效能的關鍵。因此，為了讓這部分能夠支援不同繪圖函式庫的實作，這幾個類別都已被抽象化，好讓各種繼承的類別，可以使用不同的繪圖函式庫完成不同的繪圖實作。

如圖 2-9 所示，不同的繪圖函式庫將可以提供不同的繪圖實作，如何提升繪圖效能，就看這些繪圖函式庫的實作方式。重點是 CWT 提供了這樣一個開放式的架構，可以進行擴充與最佳化。此外 CWT 在繪圖效能所作的任何改善，也將會使自行繪製的輕型元件 (Light-weight Components)間接受惠，發揮最大的好處。

此外要特別說明的是，EventManager 除了接收系統原生容器 (Native Containers)提供的各種事件之外，也將在收到系統原生容器 (Native Containers)秀出的事件後，決定那一個繪圖實作被採用。決定的方式可以依實作的不同而調整。例如可以優先採用特別的繪圖實作，但是當執行環境不支援時則改為最基本的 AWT 繪圖實作。

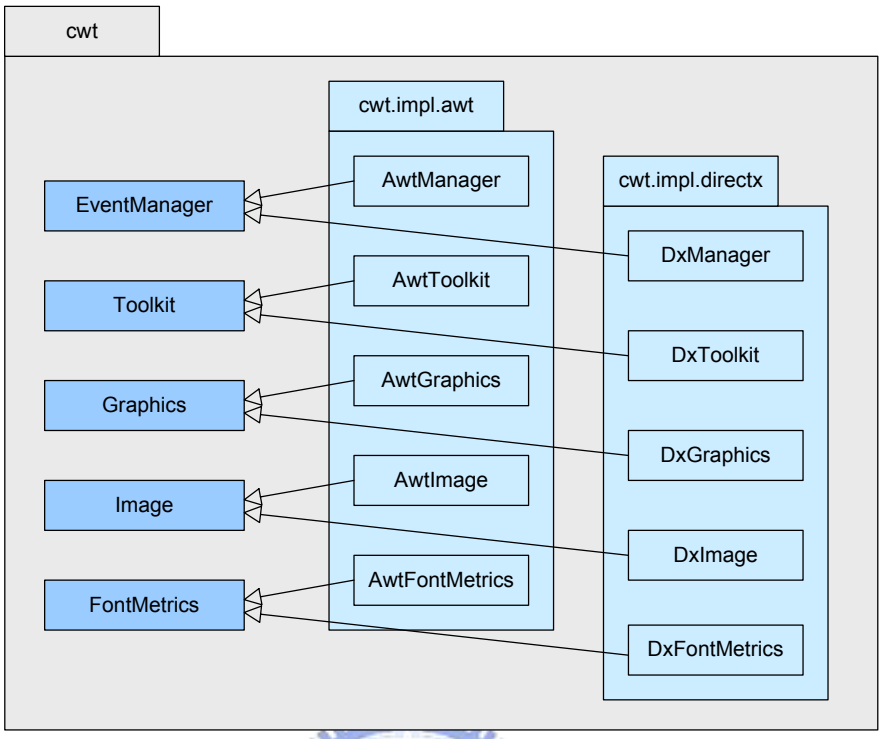


圖 2-9 : CWT 中不同的繪圖實作方式。

第三章、系統實作

介紹完本論文提出的 CWT 的設計架構之後，接下來將說明實作的內容與結果，以及實際的效能評測，最後會有一個實際案例的應用。

3.1 實作內容

CWT 是以 AWT 為共同的 API，並針對繪圖工具組的這個部分作改善，使其可以採用不同繪圖函式庫來實作。因此 AWT 中的使用者介面工具組(UI Toolkit)與繪圖工具組(Graphics Toolkit)這兩個部分是 CWT 必須完成的，接下來的兩個章節將分別詳細說明實作的內容。

3.1.1 使用者介面工具組(UI Toolkit)

使用者介面工具組(UI Toolkit)這個部分主要完成了元件模式(Component Model)和事件模式(Event Model)這兩項。

元件模式(Component Model)的部分，基本上和 AWT 差不多，主要完成的是各種元件架構。如圖 2-8 所示，系統原生容器(Native Containers)和輕型元件(Light-weight Components)是 CWT 最重要的兩大類元件。如圖 3-1 所示，這些元件都是 CWT 中自行繪製的輕型元件(Light-weight Components)的一組實作範例。

事件模式(Event Model)的部分，基本上也和 AWT 的完全一樣，如圖 2-4，AWT 有的事件， CWT 也通通都有。此外與 AWT 相同的事件分派機制(Delegation Evenet Model)，如 2.1.2 節所介紹的部分。以及最主要的，透過 Event manager 類別將事件從系統原生容器(Native Containers)傳遞給輕型元件(Light-weight Components)的部分，也都已經可以正常運作了。事件模式(Event Model)部分的完

成，等於是打通了 CWT 的任督二脈，使 CWT 的元件除了顯示之外，也具備可以操作的互動功能，這也是使用者介面工具組(UI Toolkit)這個部分最重要的功用。

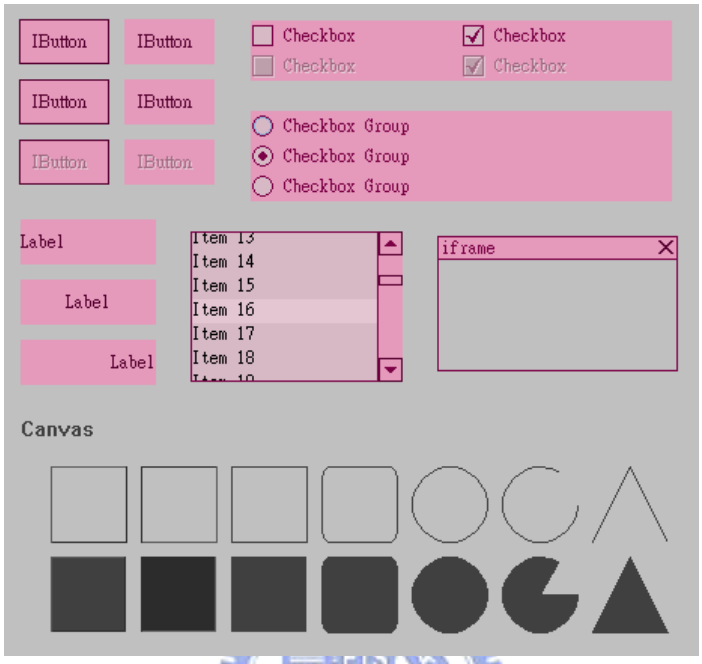


圖 3-1：CWT 元件的實作範例。

3.1.2 繪圖工具組(Graphics Toolkit)

繪圖工具組(Graphics Toolkit)所完成的就是繪圖模式(Paint Model)的部分，這部份將提供基本的繪圖功能，也是影響繪圖效能最關鍵的部分。為了驗證 CWT 的設計架構是可以支援不同繪圖函式庫的，本論文在 CWT 的繪圖工具組(Graphics Toolkits)這個部分已完成以下兩種繪圖函式庫的實作，其架構如圖 3-2 所示。

1. AWT

   AWT 是 Java 虛擬機器(Java Virtual Machine，以下簡稱 JVM)的基本繪圖函式庫，所以這部份也是 CWT 最基本的實作，以確保與現有 Java 程式的向下相容性。

2. Microsoft Language Extension (DirectX)

這部份是針對微軟的 Java 虛擬機器(Microsoft Java Virtual Machine，以下簡稱 MSVM)所完成的進階實作，著重於透過獲得 DirectX 的支援以提供更好的繪圖效能。

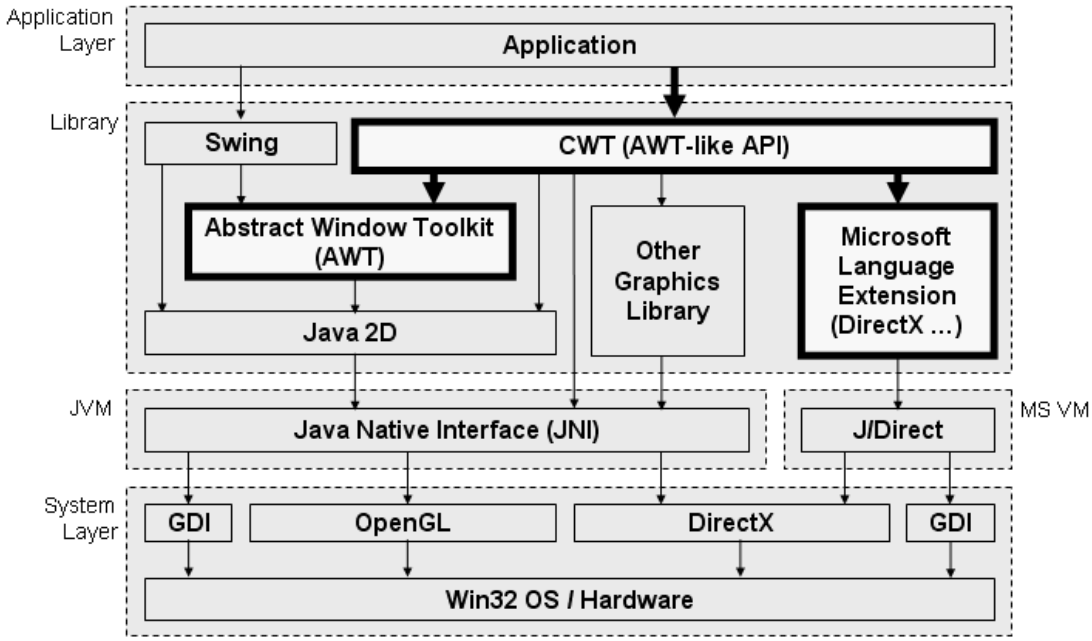


圖 3-2 : CWT 完成的兩種繪圖函式庫實作。

此外要特別說明，DirectX 僅提供影像處理與矩形區域填滿的繪圖功能，並且重點改善了這部分的繪圖效能。因此其餘的繪圖功能，像是文字與圖形等等，將仰賴 GDI[12]來提供。所以如圖 3-3 所示，DirectX 中的 DirectDraw 和 GDI 的架構圖。

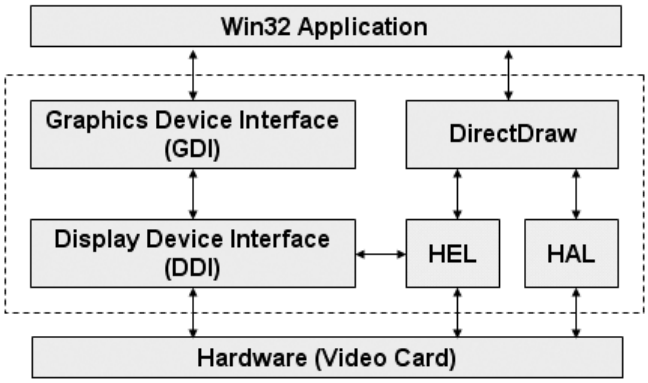


圖 3-3 : DirectDraw 與 GDI 的架構圖。[10]

3.2 效能評測

CWT 除了提供可以支援不同繪圖函式庫的架構之外，最主要的目標就是希望透過繪圖函式庫的支援提升 Java 的繪圖效能。接下來將針對 CWT 所完成的兩種繪圖函式庫的實作，作實際的效能評測，以驗證繪圖效能是否有所改善。

測試的平台包含 JVM 1.1 版到 1.5 版和 MSVM。其中 JVM 1.4 版與 JVM 1.5 版因為有 DirectX 的硬體加速支援，所以會是測試的重點。而 MSVM 的 DirectX 支援也是 CWT 實作的測試重點。雖然 JVM 1.5 版有 OpenGL 的支援，但是因為仍然有許多硬體相容性的問題，即使是目前最新釋放出的版本都尚未完全解決這個問題，因此不列入本次測試的範圍。

繪圖函式庫的測試部份將與 Java 原本的 AWT 作比較，而 CWT 已完成的兩種繪圖函式庫實作分別為：

1. AWT，以下簡稱 CWT-AWT。

2. Microsoft Language Extension(DirectX)，以下簡稱 CWT-DirectX。

在接下來的幾個章節中將會各有不同的測試項目，但是其測試的案例都將分為三大類來比較：

1. CWT-AWT vs. AWT
   如表 3-1。比較重點為 CWT-AWT 的向下相容性，其效能不應該與 AWT 差太多。

2. CWT-DirectX vs. AWT
   如表 3-2。比較重點為 DirectX 在影像(Image)繪圖效能上的優勢。幕後影像(Off-screen Image)統一使用 Image 物件。

3. CWT-DirectX (Image) vs. AWT (VolatileImage)

如表 3-3。在 JVM1.4 版與 1.5 版中只有 VolatileImage 支援 DirectX 加速。而且 VolatileImage 只被用來當做幕後影像 (Off-screen Image)。因此特別抽出來個別比較。

簡稱	繪圖函式庫	虛擬機器
AWT on MSVM	AWT	MSVM 5.0.3810
CWT-AWT on MSVM	CWT-AWT	
AWT on JVM 1.1	AWT	JVM 1.1.8_10
CWT-AWT on JVM 1.1	CWT-AWT	
AWT on JVM 1.2	AWT	JVM 1.2.2_17
CWT-AWT on JVM 1.2	CWT-AWT	
AWT on JVM 1.3	AWT	JVM 1.3.1_15
CWT-AWT on JVM 1.3	CWT-AWT	
AWT on JVM 1.4	AWT	JVM 1.4.2_08
CWT-AWT on JVM 1.4	CWT-AWT	
AWT on JVM 1.5	AWT	JVM 1.5.0_03
CWT-AWT on JVM 1.5	CWT-AWT	

表 3-1 : CWT-AWT vs. AWT。

簡稱	繪圖函式庫	虛擬機器
AWT on MSVM	AWT	MSVM 5.0.3810
AWT on JVM 1.1	AWT	JVM 1.1.8_10
AWT on JVM 1.2	AWT	JVM 1.2.2_17
AWT on JVM 1.3	AWT	JVM 1.3.1_15
AWT on JVM 1.4	AWT	JVM 1.4.2_08
AWT on JVM 1.5	AWT	JVM 1.5.0_03
CWT-DirectX on MSVM	CWT-DirectX	MSVM 5.0.3810

表 3-2 : CWT-DirectX vs. AWT。

簡稱	繪圖函式庫	幕後影像	虛擬機器
(DX) AWT on JVM 1.4	AWT	VolatileImage	JVM 1.4.2_08
(DX) AWT on JVM 1.5	AWT	VolatileImage	JVM 1.5.0_03
CWT-DirectX on MSVM	CWT-DirectX	Image	MSVM 5.0.3810

表 3-3 : CWT-DirectX (Image) vs. AWT (VolatileImage)。

CPU	Pentium 4 3.0G
RAM	1 GB
顯示卡	nVidia Geforce 6600GT 128 MB VRAM (support DirectX 9.0c and OpenGL 1.5)
作業系統	Windows XP SP1

表 3-4： 測試環境。

表 3-4 為測試環境。測試的方式是跑一支動畫程式，最後測量所花費的時間。測試程式的程式碼如附錄一。這支動畫程式會同時移動兩個繪製對象，畫面的更新次數為 100 次，每次更新都會將這兩個繪製對象重複畫 100 次，以測試出繪圖效能上的差異。測試用的繪製對象分為以下兩類：

1. 透明圖與不透明圖

   圖片的解析度為 110x110，單位為像素(pixel)，如圖 3-4 和圖 3-5 所示。測試結果將在 3.2.1 節說明。

2. 文字

   如圖 3-6 所示。測試結果將在 3.2.2 節說明。

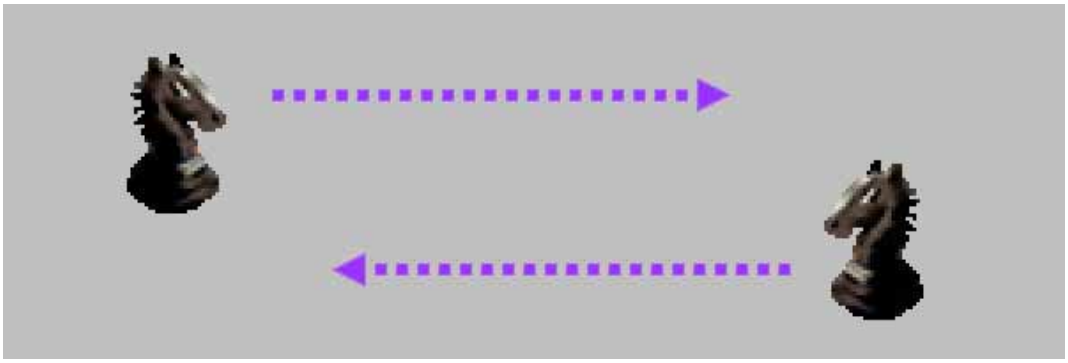


圖 3-4：測試用的透明圖片與動畫呈現方式。

圖 3-5：測試用的不透明圖片與動畫呈現方式。

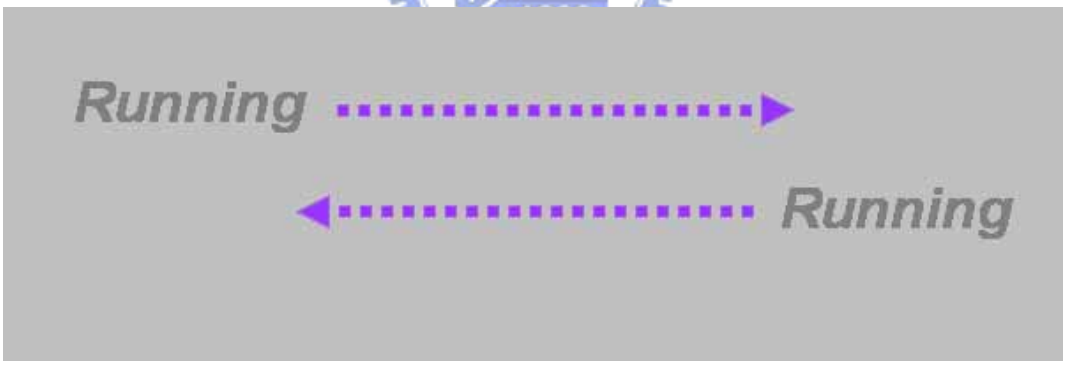


圖 3-6：測試用的文字與動畫呈現方式。

3.2.1 透明圖與不透明圖的效能評測

這部份是關於透明圖與不透明圖在繪圖效能上的測試。JVM1.1 版和 MSVM 一樣，在全彩 32 位元的模式下，不透明圖的繪圖效能雖然很差，但是透明圖的繪圖效能更嚴重，根本跑不動，因此這部份沒有測試數據。然而在高彩 16 位元的模式下卻反而有不錯的表現，甚至比新版本的 JVM 還好一點。所以這部分的測試結果都將包含這兩種色彩模式：全彩 32 位元與高彩 16 位元。

以下分三大類來比較：

1. CWT-AWT vs. AWT

   測試結果如圖 3-7 與圖 3-10 所示。雖然 CWT 因為必須提供支援不同繪圖函式庫架構的關係，可能會產生多餘的執行成本。然而從測試結果可以看出，CWT-AWT 這部分在各種平台上與原來 AWT 的比較結果相差不大，因此其向下相容性(Backward Compatible)是可以被接受的。

2. CWT-DirectX vs. AWT

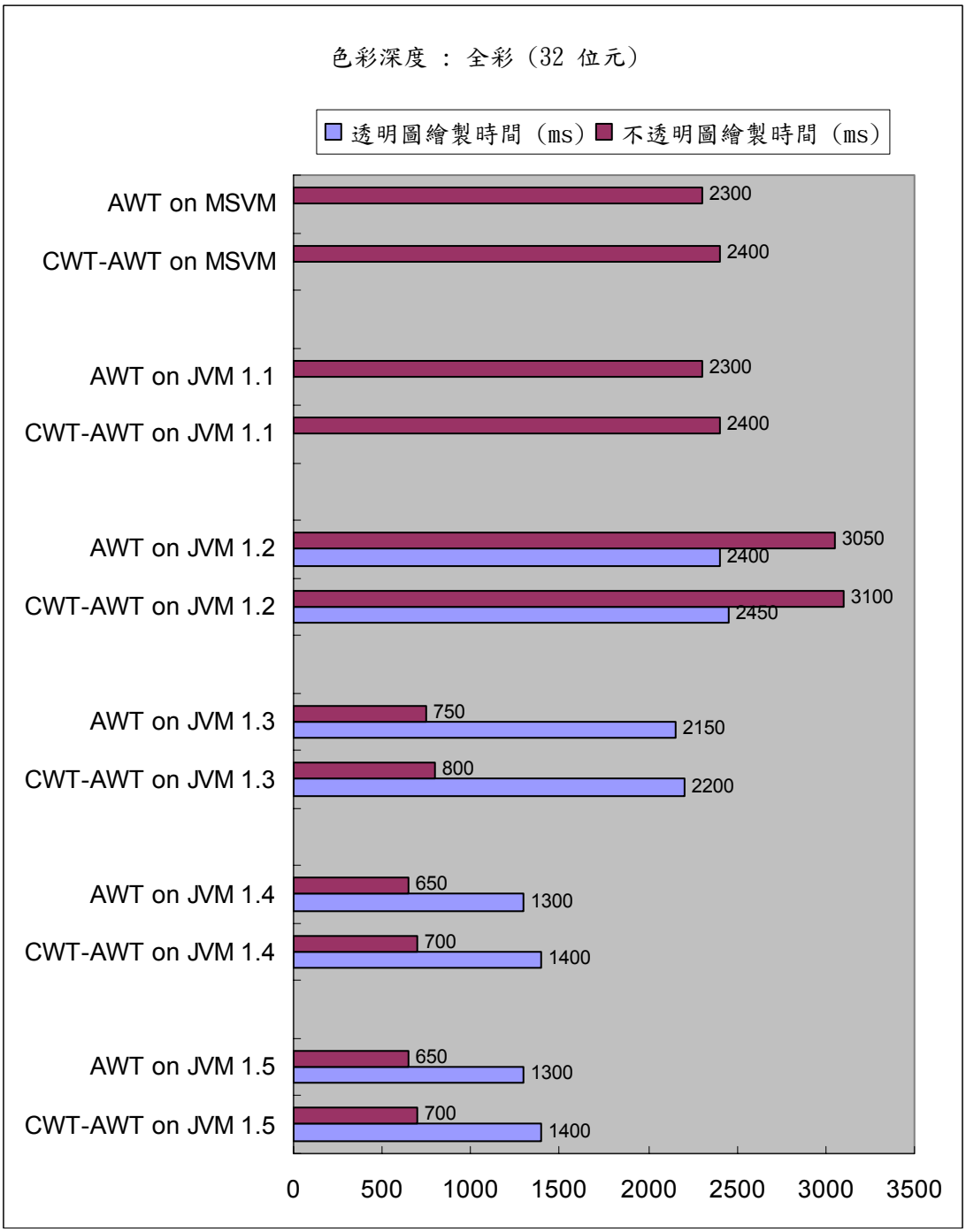
   測試結果如圖 3-8 與圖 3-11 所示。整體來說，CWT-DirectX 對於透明圖和不透明圖，無論是在全彩 32 位元的模式或是高彩 16 位元的模式下，都很穩定地有一致性的改善。

3. CWT-DirectX (Image) vs. AWT (VolatileImage)

   測試結果如圖 3-9 與圖 3-12 所示。JVM 1.4 版以後開始提供 DirectX 的硬體支援，不過這必須透過使用 VolatileImage 類別才行，AWT 原來的 Image 類別並不支援。從測試結果可以看出，這部份的效能改善在 1.4 版只對不透明圖有效， 而且還小贏 CWT-DirectX 一點點，這應該和記憶體有關，CWT-DirectX 並未直接使用顯示卡記憶體而採用系統記憶體，因為 MSVM 本身的限制。而到了 JVM 1.5 版，不管是透明圖或是不透明圖，卻反而

沒有任何改善。但是整體來看，CWT-DirectX 對於透明圖和不透明圖，無論是在全彩 32 位元的模式或是高彩 16 位元的模式下，都很穩定地有一致性的改善。

圖 3-7：透明圖與不透明圖 (全彩)，CWT-AWT vs. AWT。



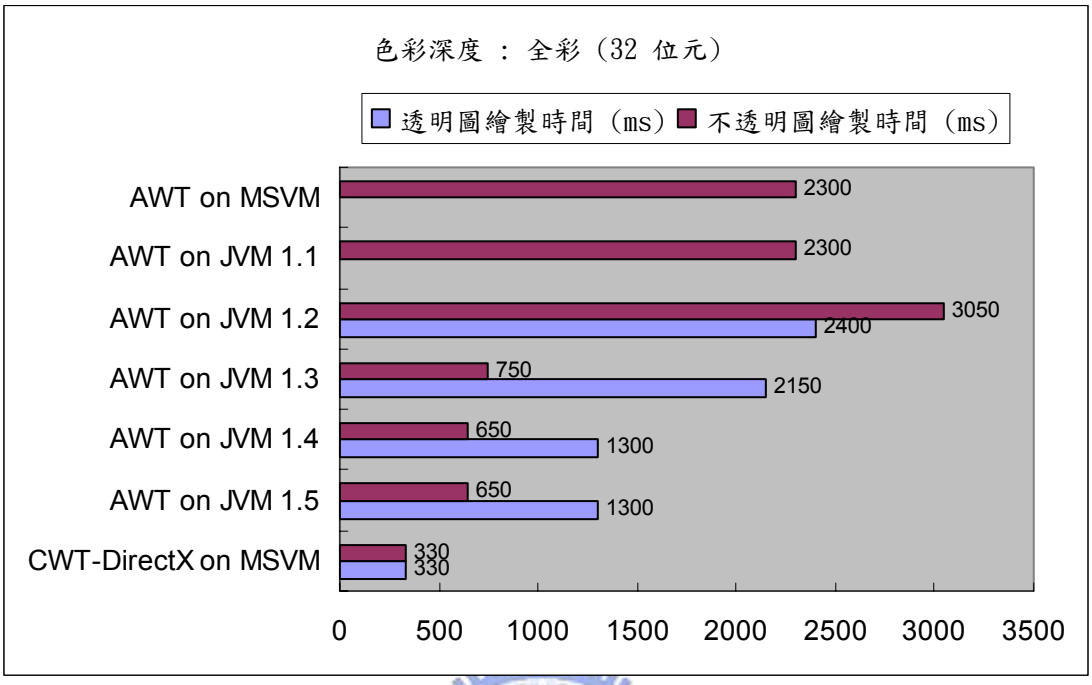


圖 3-8：透明圖與不透明圖 (全彩)，CWT-DirectX vs. AWT。

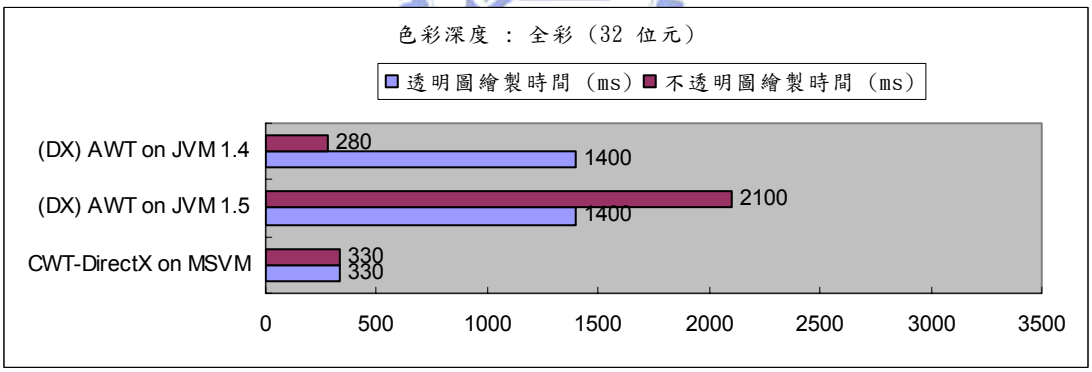


圖 3-9：透明圖與不透明圖 (全彩)，
CWT-DirectX (Image) vs. AWT (VolatileImage)。

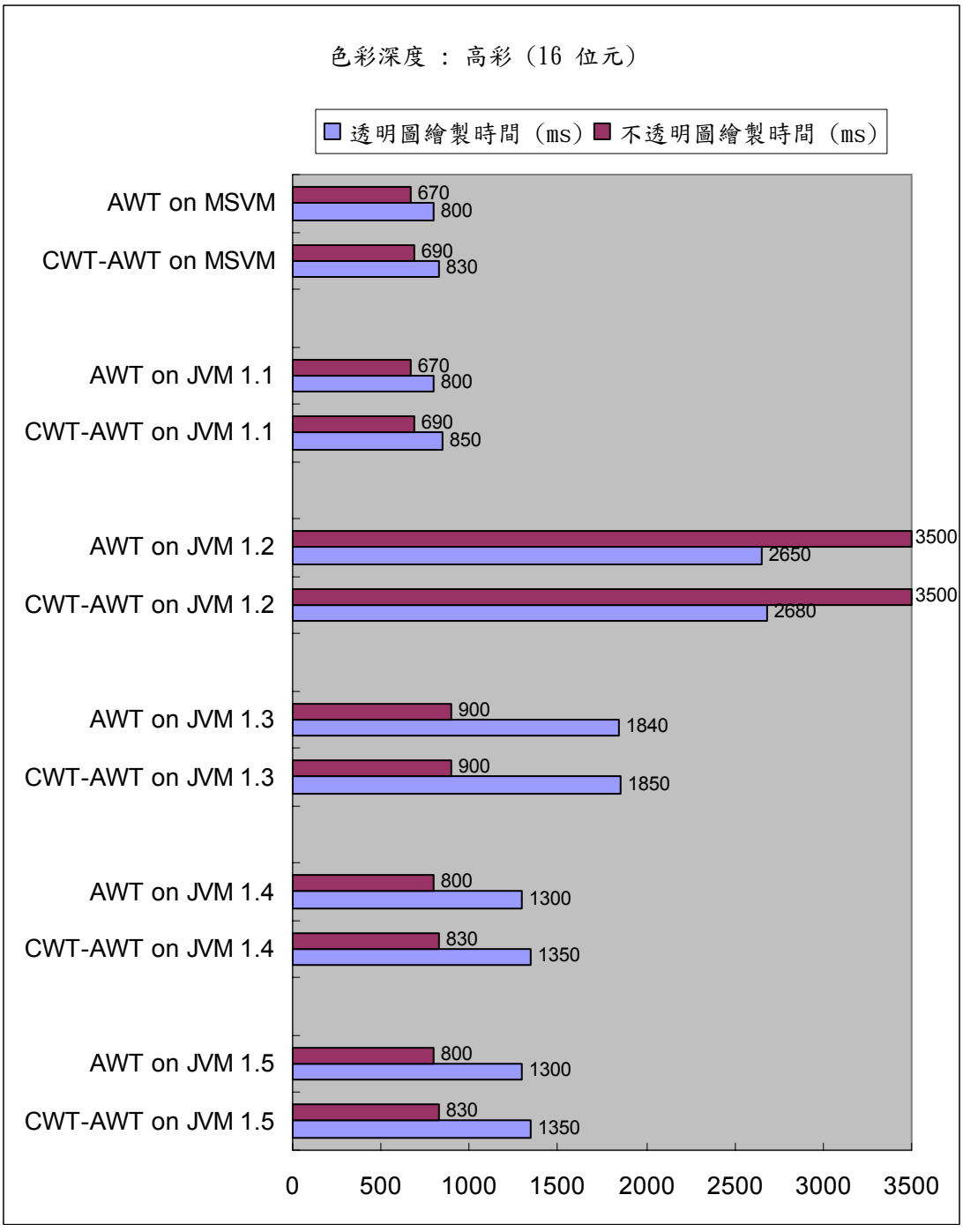


圖 3-10 : 透明圖與不透明圖 (高彩)，CWT-AWT vs. AWT。

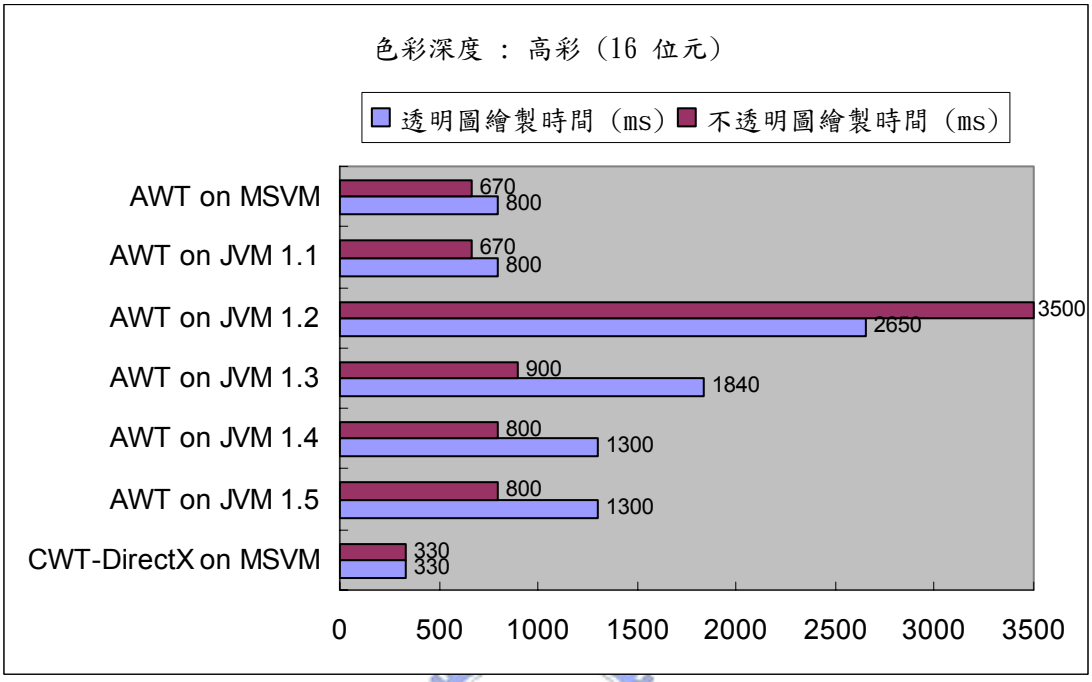


圖 3-11：透明圖與不透明圖 (高彩)，CWT-DirectX vs. AWT。

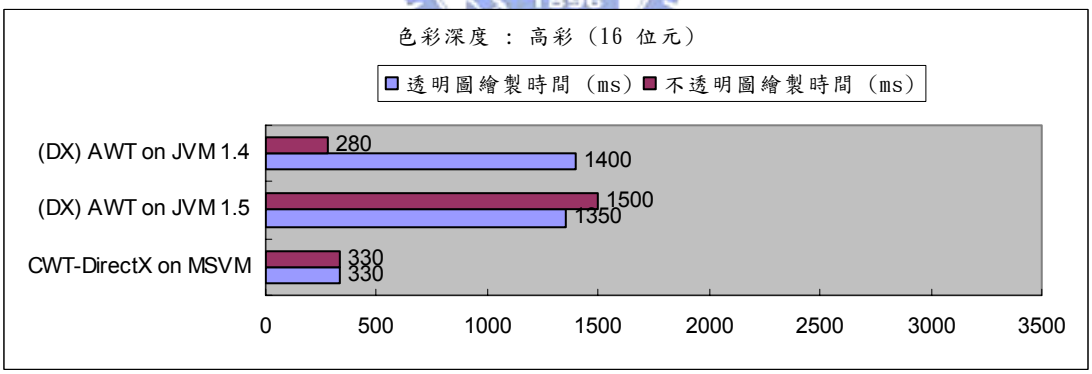


圖 3-12：透明圖與不透明圖 (高彩)，
CWT-DirectX (Image) vs. AWT (VolatileImage)。

3.2.2 文字的效能評測

這部分的測試重點是文字繪製的部分，以下分三大類來比較：

1. CWT-AWT vs. AWT

   測試結果如圖 3-13 所示。這部分與之前在圖片的測試上相同，從測試結果可以看出，CWT-AWT 這部分在各種平台上與原來 AWT 的比較結果相差不大，因此其向下相容性(Backward Compatible)是可以被接受的。

2. CWT-DirectX vs. AWT

   測試結果如圖 3-14 所示。由於 DirectX 無法支援而採用 GDI 的關係，一般來說結果很明顯地比較差，這一點和預期的相同。經過進一步分析後發現，每次要使用 GDI 來畫字，必須針對顯示記憶體做 LOCK 和 UNLOCK 的動作，這部份所需的時間成本非常高。因此，在某些最佳化的案例下，例如連續畫 N 個字時，是可以將 LOCK 和 UNLOCK 的動作從 N 次減少為一次的，測試結果發現，這的確可以大幅度地改善效能。此外，由於 DirectX 的優勢是對於圖片的處理，因此這部分若是改以預先繪製好的圖片來取代的話，基本上也是可以獲得解決的。另外，在畫字遠比畫圖多的情況下，改使用 CWT-AWT 也是 CWT 目前可以提供的解決方式之一。

3. CWT-DirectX (Image) vs. AWT (VolatileImage)

   測試結果如圖 3-15 所示。JVM 1.4 版以後開始提供 DirectX 的硬體支援，不過這必須透過使用 VolatileImage 類別才行，AWT 原來的 Image 類別並不支援。從測試結果可以看出，JVM 的 DirectX 支援結果也很差。雖然 CWT-DirectX 在一般的情況下也是一樣差，但是跟前一項比較相同的是，在最佳化的案例下，其效能是可以有大幅度改善的。

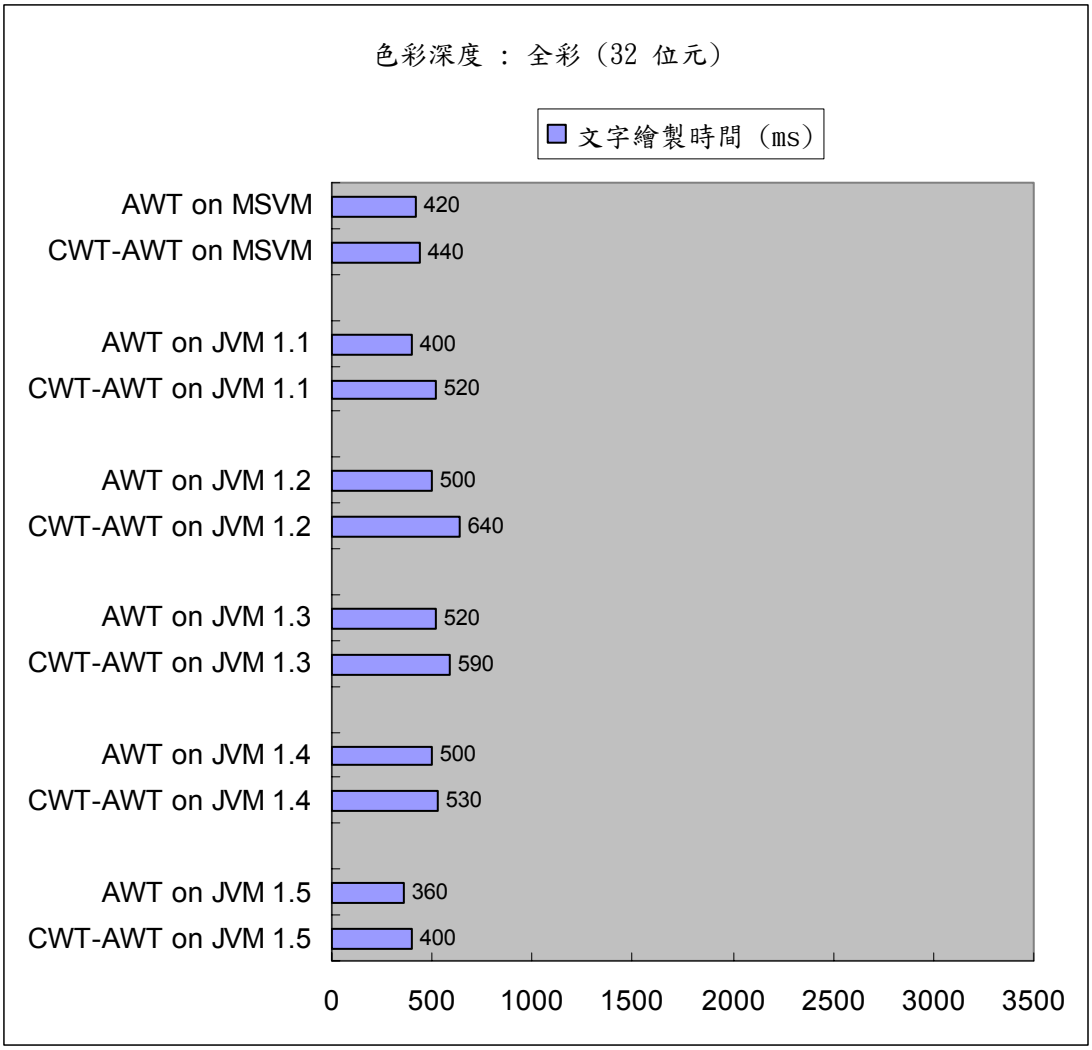


圖 3-13 : 文字，CWT-AWT vs. AWT。

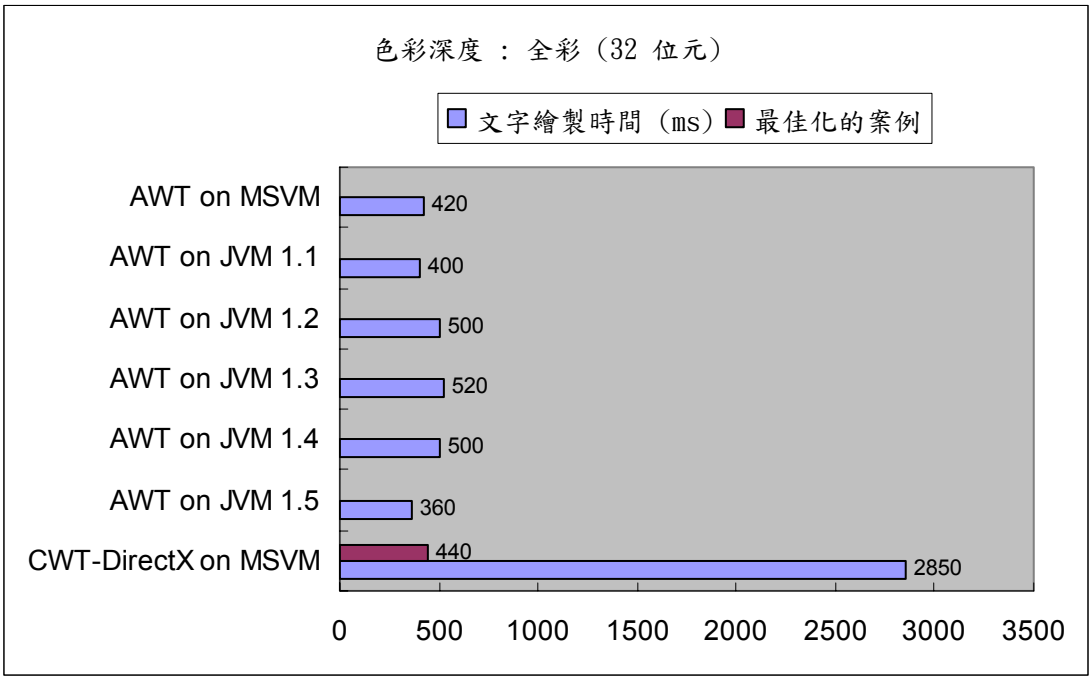


圖 3-14： 文字，CWT-DirectX vs. AWT。

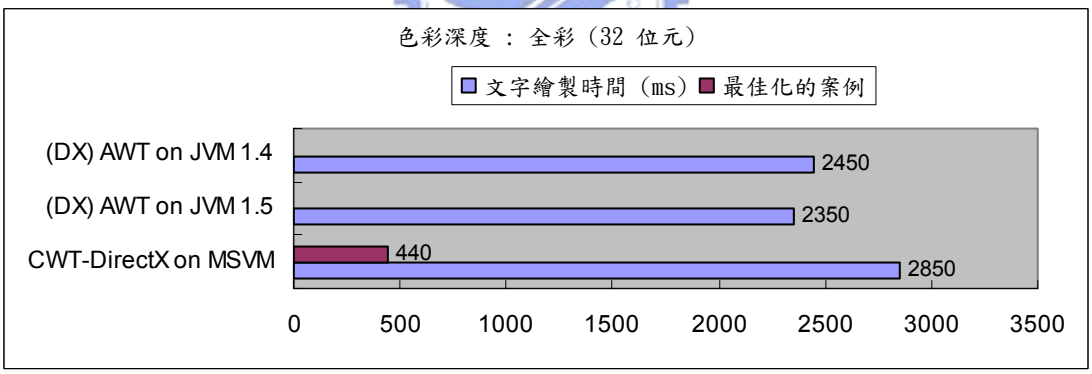


圖 3-15： 文字，CWT-DirectX (Image) vs. AWT (VolatileImage)。

3.3 實例探討 ：CYC Game

CYC[31]是本實驗室所研發的一套網路遊戲開發平台，提供了完整的開發元件，讓遊戲內容設計者可以在 Client 端發展遊戲。然而受限於 Java 本身繪圖效能的限制，CYC 目前只有一些傳統的桌上型遊戲(例如麻將、棋類遊戲和牌類遊戲等等)。如圖 3-16 和圖 3-17 所示，分為遊戲大廳畫面與遊戲畫面。

就使用者介面與畫面的呈現來看，CYC Game 目前有以下兩大問題：

1. 互動功能多，使用者介面較為複雜
   遊戲軟體為了提供更多的互動功能，因此操作介面會越來越複雜。所以一般來說，使用者介面的部分在遊戲軟體開發上的要求也特別高。而 AWT 因為其完整的元件架構與事件處理機制，所以在這部分提供了一個很好的解決方案。

2. 受限於繪圖效能，遊戲畫面以靜態為主
   然而因為 AWT 在繪圖效能上的不足，使得遊戲軟體在開發上有所限制，畫面大多以靜態畫面為主，儘量避免使用動畫，以兼顧遊戲執行時的流暢度。

基本上，本論文提出的 CWT 提供了與 AWT 完全相同的功能，並改善了繪圖效能這部分的缺失。CWT 所完成的兩個部分，使用者介面工具組(UI Toolkit)與繪圖工具組(Graphics Toolkit)，剛好分別可以解決以上兩項問題。以下分別探討 CWT 目前使用在 CYC Game 上面的應用情形：

1. CWT 的向下相容性，大大降低了轉換所需的成本。由於 CWT 提供了與 AWT 相同的介面，所以只需要將程式中 import 部分從原本的 java.awt.*改為 com.cyc.lib.cwt.*，CWT 即可順利地應用在

CYC Game 上面。這部分經過實際的測試，功能上沒有太大的問題，CYC Game 在使用 CWT 之後仍然可以正常運作。如圖 3-16 和圖 3-17 所示。

2. CWT 的使用者介面工具組(UI Toolkit)提供了與 AWT 完全相同的功能，所以完全可以應付 CYC Game 在使用者介面上的需求。CYC Game 不需要因此而改變使用者介面的設計方式。

3. CWT 的繪圖工具組(Graphics Toolkit)除了與 AWT 相容之外，並提供了更好的繪圖效能。這部份在前一節的效能評測上已經得到驗證。雖然目前以靜態遊戲畫面為主的 CYC Game，並無法很明顯地感受到這項好處。但是這也為 CYC Game 提供了更多的發展空間，未來在開發新遊戲時，將不用受到繪圖效能的限制，可以大量使用影像圖片動畫等等的遊戲效果，完全發揮 CWT 在這方面的優勢。

圖 3-16：使用 CWT 繪製的 CYC 遊戲大廳。

圖 3-17：使用 CWT 繪製的 CYC 遊戲。

第四章、結論與未來展望

為了能提升 Java 的繪圖效能，並使許多現有的 Java 程式可以因此而受惠。本論文提出了一套開放式的架構，以 AWT 為共同的 API，並針對繪圖工具組的這個部分作改善，使其可以採用不同繪圖函式庫來實作。最後本論文完成了這套系統，並達成了以下三項特性：

1. 提升繪圖效能(Performance Improvement):
   根據 Microsoft Language Extension 這部分所完成的實作結果顯示，本論文提出的這套系統成功地獲得了 DirectX 的支援，在影像的繪圖效能上有非常顯著而且穩定的改善。不管是透明圖或是不透明圖，其繪圖效能的改善程度都是一致性地領先。這證明了本系統在影像繪圖效能改善上的優勢，的確已經達到預期的目標。此外，預先處理好的影像(Image)可以用來取代即時繪製的圖形與文字，這也是一般遊戲軟體普遍採用的方式。如此將可大幅度地應用 DirectX 在影像處理上的優勢，有效提升影像的繪圖效能。

2. 向下相容性(Backward Compatible):
   除了可以提升繪圖效能的實作之外，本論文提出的系統也完成了另一項實作，AWT。這部分在實例探討中，已經成功地被應用在現有的 CYC Game 上面。向下相容的優勢已經被證明是可行的。

3. 可移植性(Portability):
   本論文提出的系統除了完成兩項繪圖函式庫的實作，以印證前述兩項特點之外。也證明了這套開放式的架構，的確可以支援更多不同繪圖函式庫的實作。因此，透過支援新的繪圖函式庫實作，來作為移植到其他平台的方式，理論上來說是可行的，例如.Net Framework。這部分雖然並未實際完成實作，但是這可以是本系統未來的發展方向之一。

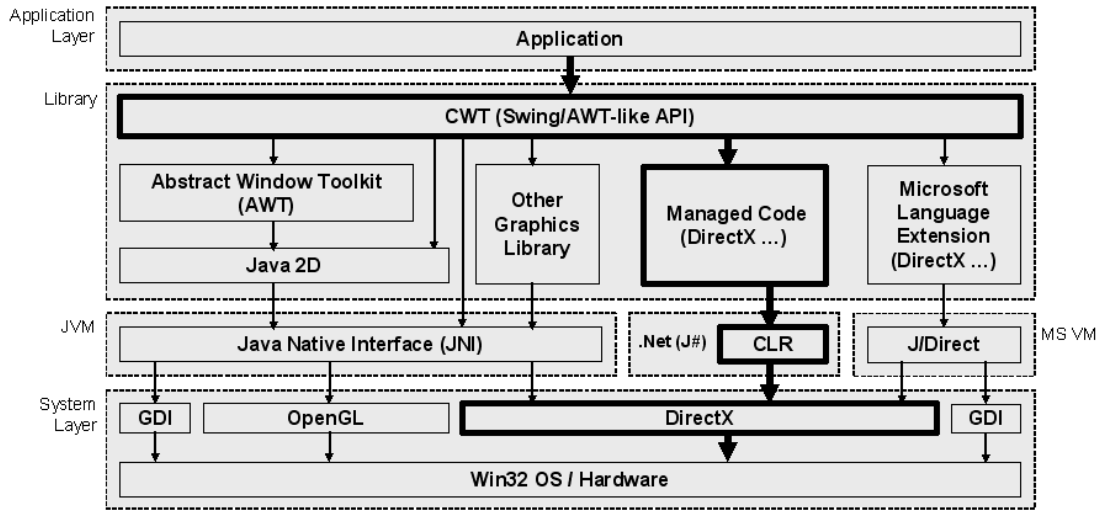


圖 4-1 : CWT 的未來展望。

雖然 CWT 在提供可以支援不同繪圖函式庫的架構已經完成，但是要真正發揮 CWT 的好處，重點在於實作採用的繪圖函式庫。因此未來的發展方向將著重於後續的應用，以及彌補 CWT 的不足之處。

1. Swing

   雖然 CWT 僅支援 AWT 介面，但是由於 Swing 是根據 AWT 為基礎所做的延伸，因此理論上來說，CWT 想要延伸支援 Swing 也是有機會的，如此一來 CWT 將可支援更多現有的 Java 程式。因此這部份也是一項很重要的未來發展方向，如圖 4-1 所示。

2. .Net Framework [3]

   首先，透過支援新的繪圖函式庫實作，來作為移植到.Net 平台的方式。理論上來說是可行的，如此將大幅度地提升 CWT 的價值，提供現有 Java 程式移植到.Net 平台的最佳解決方案。因此這部份將會是未來最優先考慮的發展方向，如圖 4-1 所示。

3. 其他繪圖函式庫

由於 CWT 可以支援不同的繪圖函式庫實作，因此其他許多繪圖函式庫也是可以考慮新增的。例如 SWT、JOGL 與 LWJGL 等等。

參考文獻

[1] Bernd Kreimeier, "Dirty Java - Using the Java Native Interface within Games." Game Developer Magazine. 1999, available from http://www.gamasutra.com/features/19990611/java_01.htm.

[2] Caspian Rychlik-Prince, Brian Matzon, Elias Naur, Erik Duijs, Ioannis Tsakpinis, Mark Bernard, "LWJGL, Lightweight Java Game Library", available from http://www.lwjgl.org/.

[3] David S. Platt, "Introducing Microsoft .NET", Third Edition, Microsoft, 2003.

[4] IBM, "SWT Standard Widget Toolkit - Development Resources" , available from http://dev.eclipse.org/viewcvs/index.cgi/%7Echeckout%7E/platform-swt-home/dev.html.

[5] Jacob Marner, "Evaluating Java for Game Development", Department of Computer Science University of Copenhagen, Denmark, 2002.

[6] James Elliott, Robert Eckstein (Editor), Marc Loy, David Wood, Brian Cole, "Java Swing", Second Edition, O'Reilly, 2002.

[7] John Sharp, Andy Longshaw, Roxburgh Pet, Peter Roxburgh, "Microsoft Visual J# .NET", Microsoft, 2002.

[8] John Zukowski, "Java AWT Reference", O'Reilly, 1997.

[9] Jonathan Knudsen, "Java 2D Graphics", O'Reilly 1999.

[10] Microsoft Corporation, "DirectX 7.0 SDK", 2000, available from http://www.microsoft.com/.

[11] Microsoft Corporation, "Microsoft SDK for Java 4.0", 1999, available from http://www.microsoft.com/.

[12] Microsoft Corporation, "Windows GDI SDK", 2000, available from http://www.microsoft.com/.

[13] Microsoft Corporation, "Xbox", available from http://www.microsoft.com/taiwan/xbox/.

[14] Nintendo, "Nintendo", available from http://www.nintendo.com/home.

[15] Robert Wells, "Java offers increased productivity." 1999, available from http://www.wellscs.com/robert/java/productivity.htm.

[16] Sony Computer Entertainment, "Play Station", available from http://www.sceh.com.tw/.

[17] Sun Microsystems, "A Jolt of Efficiency", 1998, available from http://java.sun.com/features/1998/07/efficiency.html.

[18] Sun Microsystems, "Graphics Performance Improvements", available from http://java.sun.com/products/java-media/2D/perf_graphics.html.

[19]Sun Microsystems, "JDK 1.1.8 Documentation", 1998, available from http://java.sun.com/products/archive/jdk/1.1/index.html.

[20]Sun Microsystems, "JDK 5.0 Documentation", 2004, available from http://java.sun.com/j2se/1.5.0/docs/index.html.

[21]Sun Microsystems, "JOGL, Java bindings for OpenGL API", available from http://jogl.dev.java.net.

[22]Sun Microsystems, "Java 2 SDK, Standard Edition Documentation Version 1.2.2_006", 1999, available from http://java.sun.com/products/archive/j2se/1.2.2_017/index.html.

[23]Sun Microsystems, "Java 2 SDK, Standard Edition Documentation Version 1.3.1", 2001, available from http://java.sun.com/j2se/1.3/docs/index.html.

[24]Sun Microsystems, "Java 2 SDK, Standard Edition Documentation Version 1.4.2", 2003, available from http://java.sun.com/j2se/1.4.2/docs/index.html.

[25]Sun Microsystems, "Painting in AWT and Swing", available from http://java.sun.com/products/jfc/tsc/articles/painting/index.html.

[26]Sun Microsystems, "The AWT Native Interface", available from http://java.sun.com/j2se/1.5.0/docs/guide/awt/1.3/AWT_Native_Interface.html.

[27]Sun Microsystems, "VolatileImage API User's Guide ", 2001, ftp://ftp.java.sun.com/docs/j2se1.4/VolatileImage.pdf.

[28] The OpenGL Architecture Review Board (ARB), "OpenGL", available from http://www.opengl.org/.

[29] 朱俊欣, "The Study of Portability of Java AWT to .NET Windows Forms", 交通大學資訊工程系, 碩士論文, 2004.

[30] 林秉宏, "A General Graphic Platform for Java Online Games", 交通大學資訊工程系, 碩士論文, 2002.

[31] 徐健智, "A General Development Platform for Play-on-table Game Over Internet", 交通大學資訊工程系, 碩士論文, 1999.

[32] 第三波資訊, "戲谷麻將館", available from http://www.mjonline.com.tw/.

[33] 傅鏡暉, "線上遊戲產業之道：數位內容、營運經驗", 上奇科技, 2004.

[34] 經濟部數位內容產業推動辦公室, "數位內容產業白皮書", 經濟部工業局 2004.

[35] 群想網路科技, "CYC 遊戲大聯盟", available from http://cycgame.com.

[36] 遊戲新幹線, "仙境傳說", available from http://ro.gameflier.com/default.asp.

[37] 遊戲橘子, "天堂", available from http://service.gamania.com/lineage/index.asp.

附錄一、DemoAnimator.java

```
/*   2005.06   */

package com;

import java.awt.*;
import java.awt.event.*;

public class DemoAnimator extends Canvas implements Runnable, MouseListener
{
    public final static String DEMO_TITLE[] = {"Transparent Image","Opaque Image",
                                               "Text","Rect","Circle",
                                               "Transp. Image and Text",
                                               "Transp. Image and Rect",
                                               "Transp. Image and Circle",
                                               "Transp. Image , Text and Rect",
                                               "Transp. Image , Text and Circle"};

    private final static int DEMO_IMG_TRANSP = 0;
    private final static int DEMO_IMG_OPAQUE = 1;
    private final static int DEMO_TEXT = 2;
    private final static int DEMO_RECT = 3;
    private final static int DEMO_CIRCLE = 4;
    private final static int DEMO_IMG_TRANSP_AND_TEXT = 5;
    private final static int DEMO_IMG_TRANSP_AND_RECT = 6;
    private final static int DEMO_IMG_TRANSP_AND_CIRCLE = 7;
    private final static int DEMO_IMG_TRANSP_AND_TEXT_AND_RECT = 8;
    private final static int DEMO_IMG_TRANSP_AND_TEXT_AND_CIRCLE = 9;

    private int demoType = 0;

    private int drawCount = 1;

    private boolean bDoubleBuffering = true;

    //================================================================

    private final static Color fgColor = Color.gray;
    private final static Color bgColor = Color.lightGray;
    private final static Font textFont = new Font("Arial", Font.ITALIC | Font.BOLD, 32);

    private static int frameCount = 100;
    private Image runImage[] = new Image[2];
    private byte[] sync= new byte[0];

    private int left_x;
    private int left_y = 100;

    private int right_x;
    private int right_y = 160;

    public DemoAnimator()
    {
        this(DEMO_IMG_TRANSP, 100);
    }

    public DemoAnimator(int demoType, int drawCount)
    {
        this.demoType = demoType;
        this.drawCount = drawCount;

        this.setSize(600, 300);
        this.setBackground(bgColor);

        this.addMouseListener(this);

        switch(demoType)
        {
            case DEMO_IMG_TRANSP_AND_TEXT:
            case DEMO_IMG_TRANSP_AND_RECT:
            case DEMO_IMG_TRANSP_AND_CIRCLE:
```

```
            case DEMO_IMG_TRANSP_AND_TEXT_AND_RECT:
            case DEMO_IMG_TRANSP_AND_TEXT_AND_CIRCLE:
            case DEMO_IMG_TRANSP:
                runImage[0] = Toolkit.getDefaultToolkit().getImage("run_4x.gif");
                runImage[1] = MirrorFilter.getImage(runImage[0], MirrorFilter.LEFT_RIGHT);
                break;
            case DEMO_IMG_OPAQUE:
                runImage[0] = Toolkit.getDefaultToolkit().getImage("run_4x_opaque.gif");
                runImage[1] = MirrorFilter.getImage(runImage[0], MirrorFilter.LEFT_RIGHT);
                break;
            default:;
        }

        if((demoType == DEMO_IMG_TRANSP_AND_TEXT)
            || (demoType == DEMO_IMG_TRANSP_AND_RECT)
            || (demoType == DEMO_IMG_TRANSP_AND_CIRCLE)
            || (demoType == DEMO_IMG_TRANSP_AND_TEXT_AND_RECT)
            || (demoType == DEMO_IMG_TRANSP_AND_TEXT_AND_CIRCLE)
            || (demoType == DEMO_IMG_TRANSP) || (demoType == DEMO_IMG_OPAQUE))
        {
            MediaTracker mt = new MediaTracker(this);
            for(int i = 0; i < runImage.length; i++)
            {
                mt.addImage(runImage[i], i);
             }
            try
            {
                mt.waitForAll();
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
        }

        refresh();
    }

    //========================================================================

    private Dimension offDimension = new Dimension(0, 0);
    private Image offImage;
    private Graphics offGraphics;

    public void paint(Graphics g)
    {
        synchronized(sync)
        {

            Dimension d = getSize();
            if(bDoubleBuffering)
            {
                if((offGraphics == null) || (d.width != offDimension.width) || (d.height != offDimension.height))
                {
                    offDimension = d;
                    offImage = createImage(d.width, d.height);
                    offGraphics = offImage.getGraphics();
                }
            }
            else
            {
                offGraphics = g;
            }

            offGraphics.setColor(this.getBackground());
            offGraphics.fillRect(0, 0, d.width, d.height);


            // render
            render(offGraphics);


            if(bDoubleBuffering)
            {
                g.drawImage(offImage, 0, 0, this);
            }
```

```
            timeStep();

            sync.notifyAll();
        }
    }

    public void update(Graphics g)
    {
        paint(g);
    }

    public void render(Graphics g)
    {
        for(int i = 0; i < drawCount; i++)
        {
            switch(demoType)
            {
                case DEMO_IMG_TRANSP:
                case DEMO_IMG_OPAQUE:
                    g.drawImage(runImage[0], left_x, left_y, this);
                    break;
                case DEMO_TEXT:
                    g.setColor(fgColor);
                    g.setFont(textFont);
                    g.drawString("Running", left_x, left_y);
                    break;
                case DEMO_RECT:
                    g.setColor(fgColor);
                    g.fillRect(left_x, left_y, 110, 110);
                    break;
                case DEMO_CIRCLE:
                    g.setColor(fgColor);
                    g.fillOval(left_x, left_y, 110, 110);
                    break;
                case DEMO_IMG_TRANSP_AND_TEXT:
                    g.setColor(fgColor);
                    g.setFont(textFont);
                    g.drawString("Running", left_x, left_y);
                    g.drawImage(runImage[0], left_x, left_y, this);
                    break;
                case DEMO_IMG_TRANSP_AND_RECT:
                    g.setColor(fgColor);
                    g.fillRect(left_x, left_y, 110, 110);
                    g.drawImage(runImage[0], left_x, left_y, this);
                    break;
                case DEMO_IMG_TRANSP_AND_CIRCLE:
                    g.setColor(fgColor);
                    g.fillOval(left_x, left_y, 110, 110);
                    g.drawImage(runImage[0], left_x, left_y, this);
                    break;
                case DEMO_IMG_TRANSP_AND_TEXT_AND_RECT:
                    g.setColor(fgColor);
                    g.setFont(textFont);
                    g.drawString("Running", left_x, left_y);
                    g.fillRect(left_x, left_y, 110, 110);
                    g.drawImage(runImage[0], left_x, left_y, this);
                    break;
                case DEMO_IMG_TRANSP_AND_TEXT_AND_CIRCLE:
                    g.setColor(fgColor);
                    g.setFont(textFont);
                    g.drawString("Running", left_x, left_y);
                    g.fillOval(left_x, left_y, 110, 110);
                    g.drawImage(runImage[0], left_x, left_y, this);
                    break;
                default:;
            }
        }

        for(int i = 0; i < drawCount; i++)
        {
            switch(demoType)
            {
                case DEMO_IMG_TRANSP:
                case DEMO_IMG_OPAQUE:
                    g.drawImage(runImage[1], right_x, right_y, this);
                    break;
                case DEMO_TEXT:
```

```
                    g.setColor(fgColor);
                    g.setFont(textFont);
                    g.drawString("Running", right_x, right_y);
                    break;
                case DEMO_RECT:
                    g.setColor(fgColor);
                    g.fillRect(right_x, right_y, 110, 110);
                    break;
                case DEMO_CIRCLE:
                    g.setColor(fgColor);
                    g.fillOval(right_x, right_y, 110, 110);
                    break;
                case DEMO_IMG_TRANSP_AND_TEXT:
                    g.setColor(fgColor);
                    g.setFont(textFont);
                    g.drawString("Running", right_x, right_y);
                    g.drawImage(runImage[1], right_x, right_y, this);
                    break;
                case DEMO_IMG_TRANSP_AND_RECT:
                    g.setColor(fgColor);
                    g.fillRect(right_x, right_y, 110, 110);
                    g.drawImage(runImage[1], right_x, right_y, this);
                    break;
                case DEMO_IMG_TRANSP_AND_CIRCLE:
                    g.setColor(fgColor);
                    g.fillOval(right_x, right_y, 110, 110);
                    g.drawImage(runImage[1], right_x, right_y, this);
                    break;
                case DEMO_IMG_TRANSP_AND_TEXT_AND_RECT:
                    g.setColor(fgColor);
                    g.setFont(textFont);
                    g.drawString("Running", right_x, right_y);
                    g.fillRect(right_x, right_y, 110, 110);
                    g.drawImage(runImage[1], right_x, right_y, this);
                    break;
                case DEMO_IMG_TRANSP_AND_TEXT_AND_CIRCLE:
                    g.setColor(fgColor);
                    g.setFont(textFont);
                    g.drawString("Running", right_x, right_y);
                    g.fillOval(right_x, right_y, 110, 110);
                    g.drawImage(runImage[1], right_x, right_y, this);
                    break;
                default:;
            }
        }
    }

    //=========================================================================

    private void startAnimation()
    {
        initStep();

        new Thread(this).start();
    }

    //=========================================================================

    private boolean running = false;

    private int count = 0;

    private long usedTime;
    private float fps;

    long beginTime;

    public void refresh()
    {
        left_x = 40+count*4;
        right_x = 40+(frameCount-count)*4;
    }

    public void initStep()
    {
        count = 0;
        refresh();
```

```
        running = true;

        beginTime = System.currentTimeMillis();
    }

    public void timeStep()
    {
        if(running)
        {
            count++;
            refresh();
            if(count >= frameCount)
            {
                usedTime = System.currentTimeMillis() - beginTime;
                fps = (float)(((float)usedTime)/1000.0);
                fps = (float)((float)count / (fps+0.0001));
                System.out.println(usedTime+"(FPS:"+fps+")");

                running = false;
            }
        }
    }

    public void run()
    {
        while(running)
        {
            synchronized(sync)
            {
                this.repaint();
                try
                {
                    sync.wait();
                }
                catch(Exception e)
                {
                }
            }
        }
    }

    //=========================================================================

    public void close()
    {
    }

    public void mouseClicked(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}

    public void mouseReleased(MouseEvent e)
    {
        this.startAnimation();
    }

    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}

    public static void main(String[] args)
    {
        int demoType = Integer.parseInt(args[0]);
        int drawCount = Integer.parseInt(args[1]);

        Frame f = new Frame("Demo ["+DemoAnimator.DEMO_TITLE[demoType]+"]: draw "+args[1]+" times.");
        f.addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
               System.exit(0);
            }
        });
        f.setSize(600, 300);
        f.add(new DemoAnimator(demoType, drawCount));
        f.show();
    }
}
```