逢　甲　大　學

資訊工程學系專題報告

虛擬寵物與市場設計實作

學生：　陳奇進 (四乙)
　　林琨傑 (四乙)
　　游博欽 (四乙)

指導教授：　林國貴

中華民國九十一年十二月

目錄

圖表目錄

摘要

關於寵物養成方面的遊戲大多為日本遊戲，歐美這方面的遊戲大都較重真實模擬如模擬城市一類.姑且不論風格怎樣，這方面的遊戲總是有一個很大的通病，也就是不耐玩，遊戲內容不過就是一直讓玩家一直在處理那些遊戲系統定好的那幾種問題而已，如模擬城市裡會發生火災.天災等…，遊戲缺乏變化，玩久了自然令人覺得枯燥乏味，這是這一類遊戲的最大通病，不過就遊戲來說，這點也是最致命的一項缺點。

反觀現今的 PC 遊戲市場，暢銷的遊戲約有一大半以上為網路連線遊戲，但是就遊戲性而言，現今的歐美網路連線遊戲不過就是兩方在網路上打打殺殺，你一刀我一槍的，再不然就是即時制戰略遊戲那種一直生產資源，等資源一夠就出動兵器打戰等等一類的遊戲，就遊戲本身而言，算是相當沒有娛樂性和教育意義的一種遊戲，而耐玩度更可以算是各種類型遊戲裡最差的，有的只是聲光效果而已，然而為何這麼一種遊戲會風靡全世界，原因就是多了一樣最重要的關鍵——網路。

現今的科技發達帶給了我們相當大的方便，其中最明顯的莫過於網路，網路的發達縮短了人和人的距離，就算相隔十萬八千里，也只要網路一開即可互相對談，不可否認地，網路通訊的發達的確帶給了世界相當大的影響。那麼網路對遊戲又有什麼影響呢，答案相當明顯，網路的發達給了遊戲的玩家間互相交流的機會，也就是所謂的”互動”，不管是誰，不管做任何事，沒有人會喜歡獨自一個人，只有自己一個人玩的遊戲難免讓人覺得寂寞，如果有幾個可以一起分享樂趣的人的話…..，而網路實現了這件事。所以，我們選擇以虛擬寵物結合網路及交易市場當作我們的專題目標。

第一章 導論

1.1 動機

近年來全世界遊戲產值年年升高，去年總產值已超過電影工業，可見未來娛樂產業中遊戲的市場潛力無窮。例如自 1997 年以來，韓國的遊戲正在逐漸成為國家的支柱產業之一，包括電腦遊戲、網路遊戲、街機、電子遊戲、手機遊戲在內的遊戲製作和代理發行行業得以長足發展，並且一直保持著旺盛的增長勢頭。整體業界已經形成依託於以 200 名專業遊戲選手為核心的數量龐大的遊戲玩家和 3 億美元（約合 24 億元人民幣）出口規模的消費群體，由 1496 家遊戲製作及經營業、46882 家包括網吧和其他娛樂場所在內的遊戲提供業，8 個包括韓國政府部門下屬的韓國尖端遊戲產業協會 KESA、韓國遊戲支援中心 KGPC 在內的遊戲協會、在 288 家有 IT 相關學位的大學中由政府指定贊助的 10 家遊戲大學及研究院、六家包括 On Game Net、GameTV 在內的有線電視和衛星廣播專業遊戲頻道等組成的龐大的遊戲產業群體。

回到主題， 虛擬寵物遊戲要如何利用網路功能來增加遊戲的娛樂性，日本遊戲而言，養成遊戲幾乎沒有一個對應網路功能的，因為遊戲充其量也只是養寵物而已，支援網路頂多比比數據看誰養的好或來個寵物對戰而已，一點吸引人的地方都沒有，而如果像有些網站直接做個虛擬實境讓玩家在網路上養，又顯得沒有任何吸引人之處，畢竟沒有人有那種時間天天跑網站看寵物活的怎樣，;而就歐美方面模擬經營遊戲方面而言，這種遊戲能支援網路的方式最多是成立一個網路模擬的社區或都市，每個玩家管理自己的事務，而又能保持玩家間的互動，可是問題跟之前一樣，誰有那麼多時間一整天都在網路上看有沒有事情發生，所以這個方法也不實際。

因此，吸引我們製作有關遊戲的專題，並且試著運用了市場最新趨勢的網路和 3D 技術。

1.2 目的

藉由專題的實作，學習團隊分工及軟體開發的流程。並將過去所學有關的技術實地運用，以期假以時日能夠運用在商場上。

1.3 報告總覽

第一章，介紹我們做的遊戲是什麼，製作的動機、目的。

第二章，描述專題整體架構，說明遊戲中使用到的技術背景知識。

第三章，詳細報告了虛擬寵物的實作以及 3D 程式的寫作。

第四章，交易市場的實作說明，網路及音樂的實作。

第五章，製作專題的感想及未來的展望。

第二章 虛擬寵物遊戲介紹

2.1 發展平台

發展平台：Microsoft Windows

發展工具：Visual C++ 6.0、DirectX 8.1 SDK

輔助工具：Photoshop

2.2 遊戲設計製作分工

寵物系統	寵物養成及 3D 環境建構	陳奇進
交易市場	提供物品交易的環境實作	林琨傑
網路	網路交談功能	游博欽
音效	背景音樂	游博欽

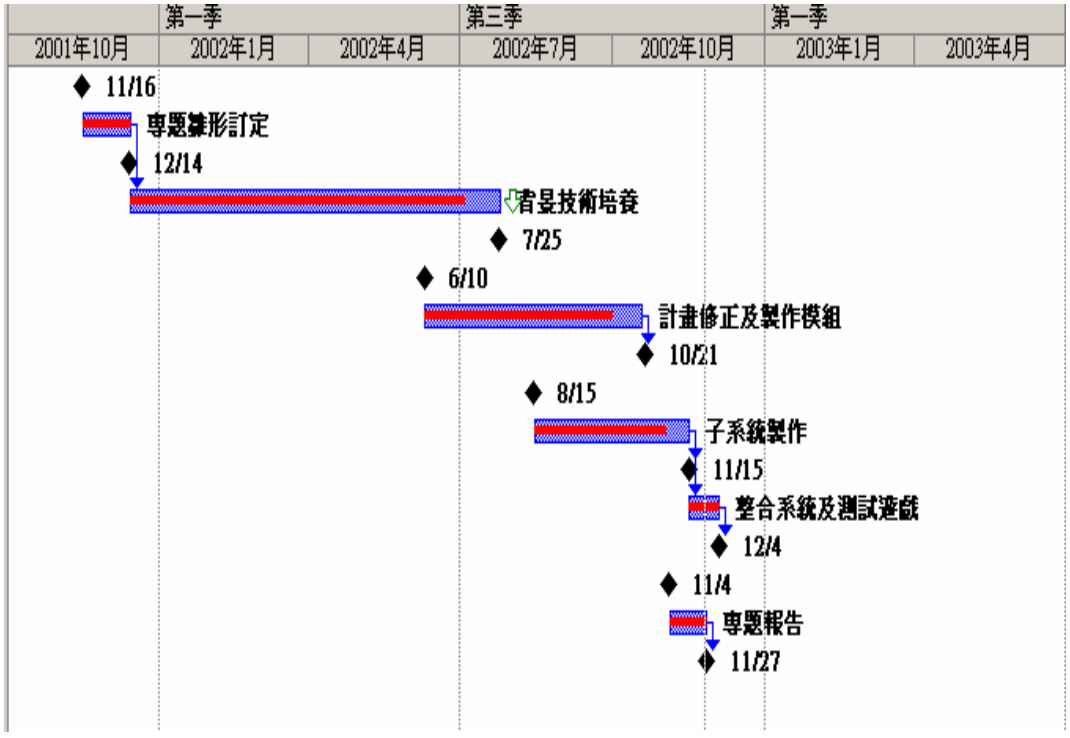


圖 2.1　專題時程圖

2.3 技術背景

DirectX 是一種介於電腦硬體跟 Windows 程式之間協商溝通的軟體，由於電腦週邊硬體的種類、製造商很多，所以微軟就開發了這套 DirectX 協商溝通軟體，讓硬體廠商所製造出來的電腦硬體能夠直接支援 DirectX 所提供的功能，而軟體設計廠商也可以不用管硬體廠商的硬體規格，直接使用 DirectX 所提供的一切功能設計多媒體軟體。所以，目前許多的電腦遊戲都是用 DirectX 所開發，包括即將發行的新型遊戲機 Xbox 也是使用 DirectX 來開發遊戲。

DirectX 是當紅的 2D、3D application 開發 SDK，特別是應用在多媒體，graphics、games 上。目前許多的 pc game 甚至 arcade game 都是用 DirectX 所開發，將來還會更普及。DirectX 的前身 WinG，也就是 Windows Game SDK，事實上根本是為遊戲製作而量身打造的，到了 DirectX 之後，由於對硬體的支援更普遍，更密切， 已經不只是遊戲專用的 SDK。

使用 DirectX 最大的好處是你可以很方便取得硬體性能資訊，將硬體的性能發揮得淋漓盡致，包括 2D、3D、音效、搖桿、鍵盤等輸入設備、網路等，這一點是其他的 3D SDK 如 Glide 和 OpenGL 所做不到的。

在以前，光是要支援一個遊戲的音效，就是一件很麻煩的事，你得考慮到 Ad Lib， SoundBlast，音源器，MIDI，VOC，後來的又出現 wave table，Windows wav 等格式和新產品，將來還會有更複雜的東西出現，光是搞這些東西就會耗掉程式設計師許多時間，相對的，純粹花在遊戲設計的時間就少了很多，效率就不好。

其實不只是音效，顯示卡的性能也越來越好，越來越複雜，不論 2D 或 3D 加速卡，如果你是一個有經驗的程式設計師或寫過顯示卡的驅動程式，你就會了解到要寫一個可以適用於所有的系統的 game 是一件

非常不容易的事，如今可憐的程式設計師發現他們又要面對網路，特別是 internet。

所以 DirectX 的最大好處不是在於 game 本身，而是對硬體的充分支援，和其便利性。

DirectX **包含幾個部分：**

DirectDraw: 最主要的功能是處理顯示記憶體和系統記憶體， 比如你可以規劃系統記憶體來儲存影像，再將這些影像以硬體所支援的功能輸出到顯示記憶體，於是就可以在螢幕上看到影像。由於 DirectDraw 可以偵測並且充分使用顯示卡的性能，因此可以將顯示速度提昇到最佳的狀況，這一點是 win32 SDK 所做不到的，因為 Windows 是多工作業系統，基本上並不允許任何應用程式隨意存取系統或顯示記憶體，所有的 window 影像都必須由作業系統來儲存和安排。

你可以使用 DirectDraw 設定你的程式是普通的 windows window 或者使用整個螢幕而捨棄 window，成為一個 full-screen application， 在這種狀況下你可以任意設定解析度。

除此之外,DirectDraw 還提供一些基本的 2d 功能,比方設定 color key，使圖形的背景顏色成為透明.這一點在 game 中應用很廣泛。Clipper 則可以使我們的角色在 window 中自由運動，不必擔心會跑到 window 外，或是角色超出顯示記憶體的定址(addressing)範圍。其他還有一些比較冷僻，對 game 沒有什麼用處，但是硬體還是支援的部分，像是 overlay。

Direct3D Immediate Mode：使用 Direct3D IM 並不是一件愉快的事,首先,你必須提供 3d model 轉換成 world， world 轉成 view， view 轉成投影平面的三組矩陣，然後你還得設定 light， render，texture，z-buffer 等等的 state， 換句話說，你必須先對線性代數和 3d 電腦圖學下一番功夫,瞭若指掌，不然你不知道這些到底是什麼

東西，也無從使用 IM。若是依我之見，除非你要開發一些 Direct3D Retained Mode 所缺乏的功能，或是你要使用某種 RM 所不支援的 3d 檔，否則沒有必要使用 IM。但是要偵測 3D 加速卡，讓 3D 應用程式能有最佳表現，還是要用到一小部分的 D3D IM。

Direct3D Retained Mode：架構在 IM 之上的一個介面。可以這樣比喻，IM 好像是程式語言，RM 是由程式語言所寫的函式庫。RM 有豐富的介面，拿來開發一套標準的 3D game 是綽綽有餘，起碼寫一個像古墓奇兵(tomb raider)的 game 一定沒問題。透過 Direct3D RM 你可以很簡單地載入 3D model， 使 model 運動，或是由 user 來控制，也可以設定 model 的貼圖，材質，animation，功能相當的多，很多人以為 D3D RM 所寫出來的程式跑不快，事實不然，畢竟 D3D RM 就是由 D3D IM 所寫成的。

DirectPlay：用來開發網路遊戲的介面，幾乎所有較普遍的通訊協定都支援，現在又有 Lobby 介面讓 user 可以在 internet 上很輕鬆地和玩家或玩家之間彼此溝通，不必花費很多時間寫 client-sever 介面程式.比如你可以透過這個介面來找對手，和對方連線來玩 game。

DirectSound：從字義上我們就了解這是一個處理音效的介面，它可以偵測玩家系統上的音效卡，從而決定所要輸出的音效種類(midi，voice)和音質。

DirectInput：處理輸入設備，包括搖桿，滑鼠和鍵盤等。

構件物件模型(COM)

大多數的 DirectX API 的物件及介面都是基於構件物件模型。構件物件模型是建立在一個使用面向物件的系統之上的，這就是構件物件模型程式的核心模式。這也是一個可以創建多個介面的介面。它是一個作業系統級的物件模型。

許多 DirectX API 都是以構件物件模型爲標準而創建的。你可以這樣理解：一個物件就象一個黑盒子記錄了硬體及應用程式通過一個介面所進行的所需的通訊。命令送到或接收自一個通過構件物件模型介面的物件，這就叫做巨集。例如：IDirectDraw4::GetDisplayMode 巨集是通過 IDirectDraw4 介面從 DirectDraw 物件來獲得當前顯示適配器的顯示模式。

在運行時一個物件可以與另一個物件相互捆綁，也可以使用其他物件所提供的介面。如果你知道物件是構件物件模型物件，並知道這個物件所支援的介面，你的應用程式(或另一個物件) 便可決定第一物件所能提供的服務。所有的構件物件模型的物件都能繼承一個巨集，查詢介面巨集，能讓你決定物件提供的介面並爲這些介面創建指標。

IUnknown 介面

所有的構件物件模型(以下簡稱 COM)介面都是從一個叫 IUnknown 的介面派生出來的。這個介面使 DirectX 能一直控制物件並有能力操作更複雜的介面。 IUnknown 有以下三個方法：

1. AddRef，能使一個物件的關聯值在當一個介面或另一個應用程式將自己連接到這個物件時加一。
   當物件被創建，它的關聯值被設置爲 1。每次應用程式獲得物件的介面或調用 AddRef 巨集時，這個物件的關聯值會增加一。可以使用 Release 巨集來使物件的關聯值減一。
2. QueryInterface，獲得物件所支援的所需特殊介面指標的特性。它決定物件是否支援一個特殊的 COM 介面。如果是，系統會增加物件的關聯值，並且應用程式可立即使用這個介面。
3. Release，能使物件的關聯值減少一。當值爲零時，物件就被解除。當關聯值爲零，物件將自行解除。使用 AddRef 宏使關聯值增加一.
   這部分 IUnknown 巨集介面繼承於物件。
   程式一定要用這個巨集來釋放 IUnknown::AddRef,

IUnknown::QueryInterface, 所做的調用，或所創建的 DirectDrawCreate。

AddRef 和 Release 巨集控制物件的關聯值。例如：如果你創建一個 DirectDrawSurface 物件，則關聯值設置爲一，當每次一個過程對於這個物件返回一個介面的指標時，該過程必須調用 AddRef 通過該指標來增加關聯值。你必須爲每個 AddRef 調用設置一個相對應的 Release。在該指標被銷毀前，你必須通過該指標來調用 Release。在關聯值變爲 0 時，這個物件變被銷毀並且所有的介面都失效。

DirectX 的 COM 介面

DirectX 程式師手冊中的 COM 介面是建立在一個很基本的 COM 程式階段。每個物件的介面都表示爲一個設備，就象 IDirectDraw4，IDirectSound 及 IDirectPlay，都直接來自於 IUnknown COM 介面。創建這些基本的物件是靠運用專門的動態銜接庫(DLL)，除此之外還用 CoCreateInstance 功能來創建 COM 物件。

一般來說，DirectX 物件模式爲每個設備提供一個主物件。其他提供服務的物件是從這個主物件派生出來的。例如： DirectDraw 物件就是顯示適配器。你能使用它創建 DirectDrawSurface 物件來表示顯存或 DirectDrawPalette 物件來表示硬體調色板。相同的 DirectSound 物件表示音效卡，所創建的 DirectSoundBuffer 物件表示卡上的音源。

除此之外，主設備物件能表示並決定硬體設備的性能，象螢幕尺寸及發色數，及音效卡是否有波表合成器。

C++與 COM 介面

對 C++程式師，一個 COM 介面就象一個抽象類。就是，它定義設置一個信號燈及語法但不啓用，並沒有實在的資料與介面。在一個 C++

抽象類，所有的巨集都定義爲抽象虛擬，也就是說他們並無實在的代碼.

C++抽象虛擬函數及 COM 介面都使用一個叫做 VTABLE 的設備。一個 VTABLE 包含一個介面所包含的所有函數的位址。如果你需要一個程式或物件使用這些函數，你可以使用 QueryInterface 巨集來檢驗物件上的介面並獲得該介面的指標。在使用 QueryInterface 後，程式或物件事實上就是獲得物件 VTABLE 的指標，這個巨集可以調用物件所包含的介面的巨集。這個機制使另一個或其他物件所使用的私人資料與子進程的調用互不相干。

COM 物件與 C++物件的另一個相似之處是宏的第一爭用是介面或類的命名，叫做 C++爭用。因爲 COM 物件和 C++物件在二進位上完全相容，編譯 COM 介面與 C++抽象類採用的是相同的語法。降低了代碼複雜度。例如：C++爭用就象一個已知的參數而不是編碼，並由 C++間接操作 VTABLE。

2.4 系統架構

整個專題的架構如下：

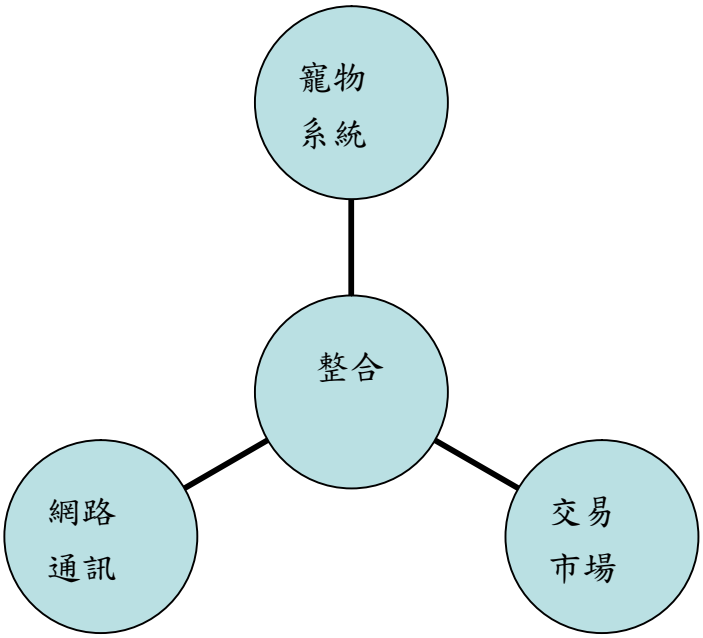


圖 2.2　整體架構圖

寵物系統：虛擬寵物養成環境及行為模式，玩家飼養寵物的介面
交易市場：提供玩家購買在飼養寵物時所需的生活用品
網路通訊：讓玩家在交易市場交易時能即時溝通

第三章 虛擬寵物實作

3.1 Direct3D 繪圖引擎

在寵物系統的實作部分，全部都是採用即時 3D 運算，因此在繪圖核心部分都是使用 Direct3D 實作。

繪製一個 3D 場景的工作可分成二個階段。第一階段叫做「轉換和打光」(通常縮寫成 T&L)。在這個階段裏，物件的每個頂點都會從原本的抽象浮點座標空間被轉換成像素為主的螢幕空間，轉換時也曾把繪製場景用的虛擬攝影機的特性納入考量。除此之外，還能在頂點上套用不同型態的光源效果。(還有如裁切和檢視埠縮放等重要工作也都是在第一階段發生)。第二個階段，稱為掃描成像 (rasterization)，基本上是將這些轉換和打光過的頂點組織成點、線和三角形。掃描器會在 DirectDraw 繪圖頁(稱為繪製目標)上繪出最終的形狀，其過程中應用了貼圖對映，以及在連接的頂點間內插各種的性質(如色彩)。掃描成像的同時也利用深度緩衝區來決定哪個最終像素是使用者看得見的，哪一個則是被其他基本形狀的像素所蓋住。圖 3.2 說明了這二階段如何配合來達到繪製一個 3D 物件的作業。

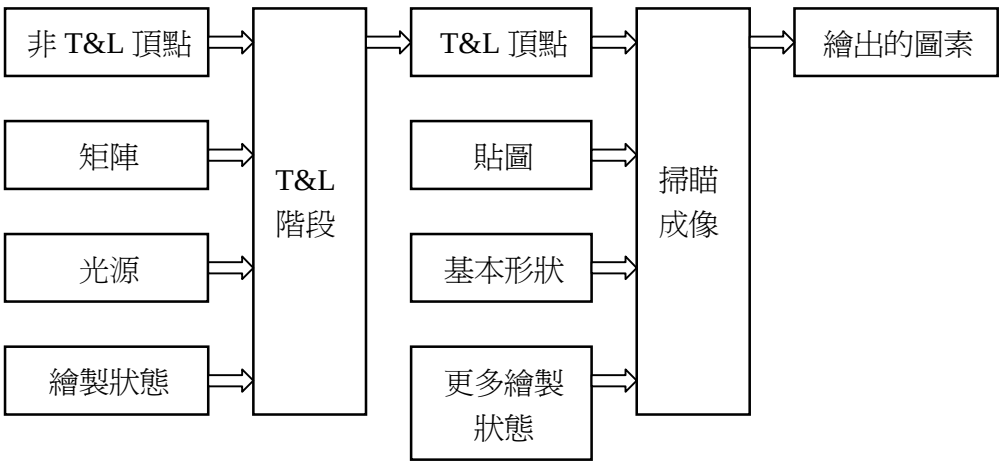
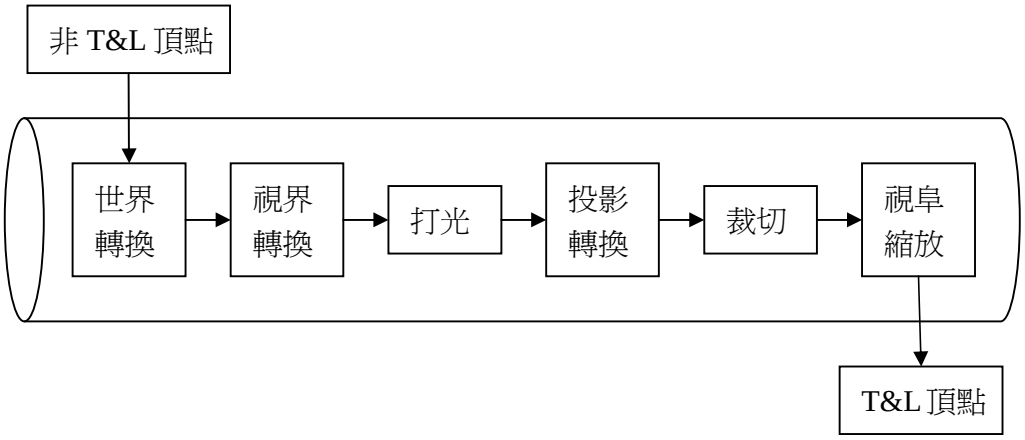


圖 3.1　繪製一個 3D 物件的兩個階段

Direct3D 可以執行 T&L 階段和掃描成像階段作業。我們可以依照程式需求，設定 Direct3D 來分別執行這些步驟或在一次函式呼叫中全部完成，如果願意的話，也可以用自己設計的轉換和打光，並且將處理過的頂點直接傳給 Direct3D 掃描器，跳過 Direct3D 的 T&L 階段。接著討論 Direct3D 頂點和轉換以及打光的過程。

T&L 處理管線的概要

轉換和打光的過程通常可視為一種處理管線。在此過程中，尚未轉換和打光的頂點會從一端進入，中間會執行一些連續的作業，最後，轉換過和打光過的頂點會從另一端出來。您的應用程式可以指定某些想要用到的矩陣、檢視埠，和任何的光源，來設定 T&L 處理管線。程式會將頂點送入管線中，加以轉換、打光、裁切，將其投射到螢幕空間，並且依檢視埠設定值加以縮放。離開管線的頂點會被當作「已處



理」，而且可以再傳給掃描成像器。T&L 處理管線的示意圖如圖 3.3。

圖 3.2　T&L 管線

我們可以調整 T&L 處理管線，讓它省略一些或全部的步驟。或套用我們自己設計的轉換和打光演算法，並且關掉部分的處理動作並將

已經在 Dtrect3D 中轉換和打光過的頂點傳送給 Direct3D。然而，在某些情況下，最好還是使用 Direct3D 的完整 T&L 處理管線。因為這個程式已經最佳化，能夠運用所有最新的 CPU 延伸功能。同時，T&L 可以在某些 3D 圖形卡的特定硬體上快速執行。下面說明當一個頂點通過管線的各種階段時會發生什麼事。

世界轉換

在 T&L 過程中，各種繪製出的物件的所有座標都需要轉換成一個常用的座標系統，稱為世界空間(world space)。但是對我們的程式來說，用它自己的局部座標系統(稱為模型空間：model space，或本體空間：local space)來表示每個物件的座標會比較方便。定義在模型空間和世界空間轉換方式的矩陣稱為世界轉換矩陣(world transformation matrix)。在某些狀況下，使用模型空間比較容易。例如，要移動一個物件，只需重新定義模型空間和世界空間的轉換，會比手動改變世界空間中物件的所有座標來得容易且快速。模型空間也允許「實例化」(instancing)，這意謂著您可以使用某個模型座標到世界座標轉換，在其中畫一個物件，例如球體，然後在其他的地方再畫一次，採用另一種模型-世界轉換方式。模型空間也允許更多的自然局部轉換。比方說，不論球體在世界空間的哪裏，當起點落在球體的中心時，要讓球體繞著它的中心轉是最容易的。

T&L 處理管線的第一階段會用到我們指定的世界轉換矩陣，將物件頂點中的位置座標從局部空間轉成世界空間。世界轉換矩陣可以運用任何的旋轉組合方式(物件繞著 x 軸、y 軸或 z 軸)、平移(物件沿著 x 軸、y 軸或 z 軸)以及縮放(放大或縮小物件)。圖 3.4 說明了世界座標系統和一個模型的局部座標系統問的關係。

世界座標系統中最重要的特性是它提供了一個座標空間讓所有的 3D 物件分享，而不是為每個 3D 物件取得唯一的座標系統。一旦在世界座標中指定了頂點，Direct3D 就不需要記住任何的局部座標，或處理

任何的模型座標到世界座標轉換。從局部座標系統到世界座標系統間的轉換就類似於將各種以磅、公斤和噸計算的物體變成以公克計算，也就是替世界的所有物件提供了一個共同因數(denominator)。如果不需要用到個別的模型空間，而希望直接在世界座標中設定某個物件的頂點的話，則可以把世界轉換矩陣設成單位矩陣 (Identity matrix)。這表示該物件的模型空間等於共同的世界空間。

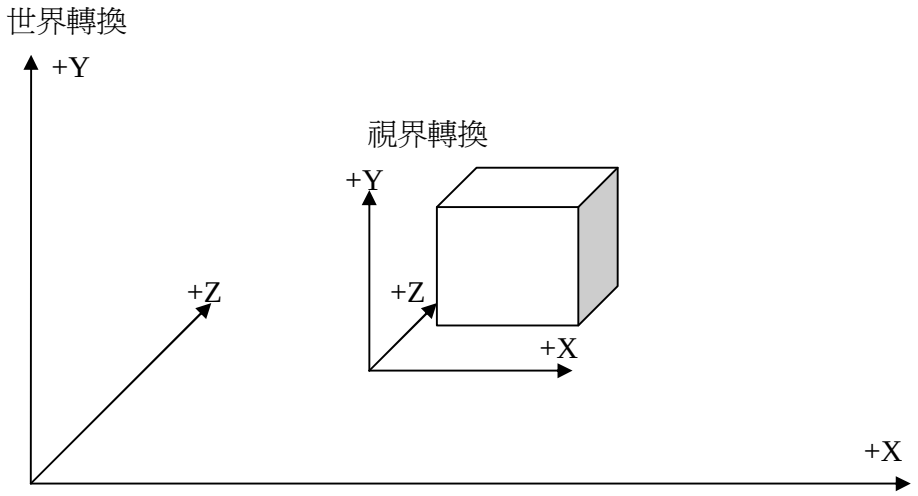


圖 3.3　世界轉換（世界和本體座標系統）

視界轉換

T&L 處理管線的第二階段會將頂點從世界空間轉換到攝影機空間(camera space)，攝影機空間中的虛擬攝影機會在原點，並直接指向正 Z 軸。在這個階段中光源(在世界空間中已經設定)也會被轉換到攝影機空間。

打光

到此為止，任何現存光源的效果都已被計算出來，並且應用到頂點上。打光的程式會查看每個頂點的位置及所屬法向量(normal vector，也就是方向為遠離包含頂點的多邊形的向量)、色彩以及目前

的材質。程式會依據所有的因素和光源的性質，去計算每種光在頂點上的效果，並且將頂點的最終色彩存回頂點的結構。之後 Direct3D 就不需要再考慮光源和材質了。

投影轉換和平頂角錐檢視

處理管線的下一階段會依據場景中的物件和指定檢視埠問的距離來縮放他們，縮放也稱為投影轉換(projection transformation)，也就是在繪製的場景中製造出深度的呈現效果，這會讓遠方的物件看起來較離檢視埠近的物件來得小。一旦發生了投影轉換，頂點就會被視為存在於投影空間。要了解投影轉換是如何運作，可以想像一個視覺上的平頂角錐(viewing frustum)。所謂平頂角錐是一種金字塔的幾何形狀但是沒有尖頂。在電腦圖學中，平頂角錐檢視的形成原理是去想像放置一個平頂角錐，讓它的尖頂落在攝影機的位置，而攝影角度朝下指向金字塔的中央，金字塔的四個「牆」會沿著螢幕四邊投射出去，並且在近端和遠端的裁切平面(clipping plane)裁切金字塔的前後方。最後的平頂角錐檢視代表了在繪製場景中的可見到的攝影機空間體積。雖然可以用很多種的投影方式(都會影響 3D 模型從攝影機空間投射到螢幕的作法)，不過我們使用的是最普遍的一種：透視投影(perspective projection)，就是讓離攝影機較遠的物件變得比離攝影機較近的物件還小。另一種不會縮放場景中的物件的轉換方式，稱為正交投影(orthogonal projection)。雖然正交投影在某些應用上很有用，但對於大多數的第一人稱遊戲而言，還是會採用依距離遠近來作縮放物件的標準作法。

要執行透視投影，投影轉換會將平頂角錐檢視轉成一個立方體(cuboids)，它是一種邊長並不完全相同的立方形狀。因為平頂角錐檢視的近端會比遠端小，較近的物件看起來會比遠端的物件大，在場景中產生透視的效果。

圖 3.4 說明了平頂角錐檢視的組成部分

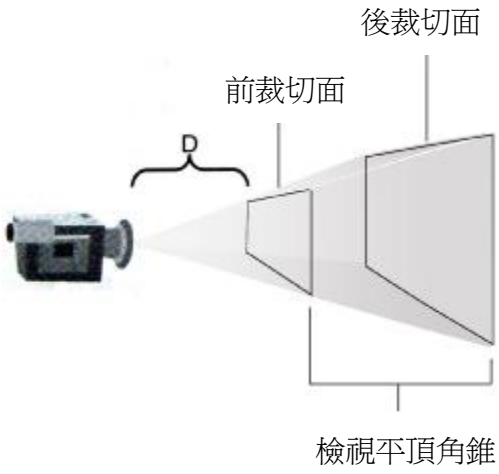


圖 3.4　檢視平頂角錐

在圖 3.5 中，您可以看到前後的裁切平面。裁切平面的作用是用來定義當繪製場景時觀察者可見到的內容。前裁切平面定義了某個物件若要被包含在繪製的場景中時的最近距離，後裁切平面則定義了能被包含的最遠距離。凡是落在平頂角錐以外的物件都不會成像。近端和遠端的裁切平面的重要性在於他們設定了 z 軸上的最小和最大值。假如沒有遠端的裁切平面，繪圖器就不會知道 z 軸的最大值是多少。記住，您所設定的遠端裁切平面的範圍會影響到場景速度和視覺品質。如果您設得太接近，會碰到「突然出現」(popping)，這意謂著物件會突然跳入檢視中，而不是逐漸向外移動，而變得越來越小且不明顯。假如您將平面設得太遠，會讓裁切的效果不彰，因為會有更多物件會被繪製而造成成像時間變長。

平頂角錐檢視可用視野(field of view)來說明，視野包括由攝影機伸出的平面所構成的角度以及從視點到前後裁切平面的距離。這種距離可由前後平面的 Z 座標所定義。D 變數則是從攝影機(視界轉換所定義的空間原點)到前裁切平面間的距離。

圖 3.6 說明了投影轉換將平頂角錐檢視轉成一個新的座標空間的作法。因為我們用的是透視投影，平頂角錐會變成立方體。一旦投影

完成，x 軸的左平面極值會是-1，而右平面極值則是 1，y 軸的底平面極值會是-1，而頂端平面極值則是 1，z 軸的前平面極值會是 0，而後平面極值則是 1。

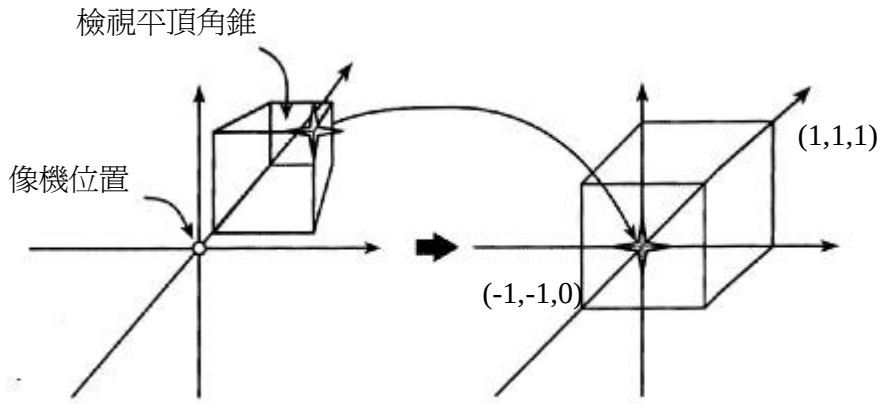


圖 3.5　透視投影

裁切（Clipping）

裁切是確保那些完全不在平頂角錐檢視內的物件不會成像，而那些與平頂角錐檢視相交的物件則不會有任何一個像素被繪到檢視埠設定的矩形之外。

檢視埠縮放

T&L 處理管線的最後一個步驟是調整頂點來配合檢視埠。檢視埠讓我們設定如何將繪製圖樣對應到繪圖目標平面上。我們可以同時指定平移和縮放作業。一般來說都會填滿整個繪圖目標，因此不會設定，而是縮放頂點座標讓 X 座標的 1 值對應到左邊線，X 的 1 值對應到右邊線，的-值對應到底線，而 Y 的 1 值對應到頂端線。

以下為建立一個 Direct3D 程式的基本步驟：

Step 1: **建立一個視窗**

製作視窗程式的第一個步驟當然就是先產生一個視窗。而視窗程式都是由 WinMain 函數開始執行。以下為 WinMain 函數及視窗初始化：

```
INT WINAPI WinMain( HINSTANCE hInst,
                    HINSTANCE,
                    LPSTR, INT )
{
   //登記 window class.
   WNDCLASSEX wc = {    sizeof(WNDCLASSEX),
                        CS_CLASSDC,
                        MsgProc,
                        0L,
                        0L,
                        GetModuleHandle(NULL),
                        NULL, NULL, NULL, NULL,
                        "D3D 應用程式",
                        NULL };
   RegisterClassEx( &wc );

   //建立應用程式視窗
   HWND hWnd = CreateWindow( "D3D 應用程式",
                             "D3D 應用程式",
                             WS_OVERLAPPEDWINDOW,
                             100, 100, 300, 300,
                             GetDesktopWindow(),
                             NULL,
                             wc.hInstance,
                             NULL );
}
```

Step 2：**初始化** D3D

視窗準備好之後，接著要初始化繪圖用的 Direct3D 物件。首先立 Direct3D 物件

```
g_pD3D = Direct3DCreate8( D3D_SDK_VERSION );
```

接著取得目前的顯示模式

```
D3DDISPLAYMODE d3ddm;
g_pD3D->GetAdapterDisplayMode(
        D3DADAPTER_DEFAULT, &d3ddm );
```

然後設定參數

```
D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory( &d3dpp, sizeof(d3dpp) );
d3dpp.Windowed   = TRUE;
d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
d3dpp.BackBufferFormat = d3ddm.Format;
```

最後建立 Direct3D 裝置

```
g_pD3D->CreateDevice(
        D3DADAPTER_DEFAULT,
        D3DDEVTYPE_HAL,
        hWnd,
        D3DCREATE_SOFTWARE_VERTEXPROCESSING,
        &d3dpp,
        &g_pd3dDevice )
```

以上程式建立一個使用預設顯示卡的 Direct3D 裝置

Step 3: **處理系統訊息**

在建立好視窗及初始化 Direct3D 物件之後，就可以開始準備繪圖了。在大部分的情形之下，視窗程式利用訊息迴圈處理系統訊息，然後利用訊息佇列中沒有訊息的時候繪圖。

```
// 訊息處理迴圈
MSG msg;
while( GetMessage( &msg, NULL, 0, 0 ) )
{
    TranslateMessage( &msg );
    DispatchMessage( &msg );
}
```

迴圈每執行一次，DispatchMessage 就會呼叫 MsgProc 處理佇列中的訊息。當 WM_PAINT 在佇列中時，程式呼叫 Render 繪圖。

```
LRESULT WINAPI MsgProc( HWND hWnd, UINT msg, WPARAM
 wParam, LPARAM lParam )
{
    switch( msg )
    {
        case WM_DESTROY:
        PostQuitMessage( 0 );
        return 0;

        case WM_PAINT:
            Render();
            ValidateRect( hWnd, NULL );
            return 0;
    }
    return DefWindowProc( hWnd, msg, wParam, lParam
 );
}
```

Step 4：**定義頂點類型**

定義要繪製的 3D 物件的頂點類型。利用自訂的頂點結構及 flexible vector format (FVF)。

```
struct CUSTOMVERTEX
{
    FLOAT x, y, z;
    DWORD color;        // 頂點色彩
};
```

接下來定義 FVF 告訴頂點緩衝區我們所採用的自訂頂點類型。

```
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_DIF
FUSE)
```

Flexible Vertex Format 旗標有好幾種類型，在此我們只使用代表位置的 X、Y、Z 及色彩資訊。

Step 5：**設定頂點緩衝區**

頂點格式設定好之後，就可以開始建構 3D 物件了。以下程式建立一個 3D 空間中的三角形。

```
CUSTOMVERTEX g_Vertices[] =
{
  { 10.0f,  0.0f, 0.5f, 0xffff0000, }, // x,y,z,色彩
  {  0.0f, 10.0f, 0.5f, 0xff00ff00, },
  {-10.0f,  0.0f, 0.5f, 0xff00ffff, },
};
```

接著呼叫 IDirect3DDevice8::CreateVertexBuffer 建立頂點緩衝區。

```
if( FAILED( g_pd3dDevice->CreateVertexBuffer(
            3*sizeof(CUSTOMVERTEX),
            0,
            D3DFVF_CUSTOMVERTEX,
            D3DPOOL_DEFAULT, &g_pVB)))
   return E_FAIL;
```

CreateVertexBuffer 的前 2 個參數告訴 Direct3D 要求的緩衝區大小及用法。下 2 個參數詳細描述了頂點向量的格式及新緩衝區在記憶體的區域。向量格式就是之前所定義的 D3DFVF_CUSTOMVERTEX。而 D3DPOOL_DEFAULT 旗標告訴 Direct3D 將頂點緩衝區建立在最適合的地方。而最後一個參數為緩衝區的記憶體位址。

建立完緩衝區之後，接著將三角型的頂點資料填入緩衝區。

```
VOID* pVertices;
if( FAILED( g_pVB->Lock( 0,
                         sizeof(g_Vertices),
                         (BYTE**)&pVertices,
                         0 ) ) )
   return E_FAIL;
memcpy(  pVertices,
         g_Vertices,
         sizeof(g_Vertices) );
g_pVB->Unlock();
```

在填入緩衝區之前需先呼叫 IDirect3DVertexBuffer8::Lock 鎖定之後才可以填入資料，填完之後呼叫 IDirect3DVertexBuffer8::Unlock 解除鎖定。

Step 6: **繪製**

將頂點資料填入緩衝區之後，便可以開始繪圖。繪圖之前需先清除 back buffer。

```
g_pd3dDevice->Clear(    0,
                        NULL,
                        D3DCLEAR_TARGET,
                        D3DCOLOR_XRGB(0,0,255),
                        1.0f, 0L );
```

接著所有繪圖的程式段都必須在 IDirect3DDevice8::BeginScene 和 IDirect3DDevice8::EndScene 之間，這二個函數告訴系統我們開始繪圖與完成繪圖。

```
g_pd3dDevice->BeginScene();
```

繪製頂點緩衝區中的資料需要幾個步驟，首先要設定繪製的串流資料來源，以及要繪製的頂點緩衝區。

```
g_pd3dDevice->SetStreamSource(
            0, g_pVB, sizeof(CUSTOMVERTEX) );
```

SetStreamSource 的第一個參數告訴 Direct3D 裝置資料流的來源。第二個參數指定連結到資料流的頂點緩衝區。第三個參數是大小。

下個步驟呼叫 IDirect3DDevice8::SetVertexShader 告訴 Direct3D 所使用的 vertex shader 自訂 vertex shader 是屬於高階技巧，然而在大部分情形之下 vertex shade 只要包含 FVF 就行了。讓 Direct3D 知道所要處理的適合種型態的頂點，以下程式設定 VertexShader 為之前設定的 FVF。

```
g_pd3dDevice->SetVertexShader(
                D3DFVF_CUSTOMVERTEX );
```

SetVertexShader 的唯一一個參數就是所要處理的 vertex shader。這個參數的值可以由 IDirect3DDevice8::CreateVertexShader 產生，或者是一個 FVF。

接下來，以下程式使用 IDirect3DDevice8::DrawPrimitive 繪製頂點緩衝區中的頂點資料。

```
g_pd3dDevice->DrawPrimitive(
                D3DPT_TRIANGLELIST,
                0,
                1 );
```

DrawPrimitive 的第一個參數告訴 Direct3D 繪製何種基本型態。可以是點串列、線串列、線條塊、三角形串列、三角形條及三角形扇。第二個參數指定繪製的第一個頂點索引，第三個是欲繪製頂點的個數。

最後一個步驟就是呼叫 EndScene。

```
g_pd3dDevice->EndScene();
```

Step 7: **結束**

最後當程式結束的時候，必須將物件釋放掉，所以的 COM 物件都必須呼叫 IUnknown::Release 方法來告知系統，物件才會真正從記憶體中移除。

```
VOID Cleanup()
{
    if( g_pd3dDevice != NULL)
        g_pd3dDevice->Release();
    if( g_pD3D != NULL)
        g_pD3D->Release();
}
```

3.2 程式架構

虛擬寵物的實作，主要架構如圖 3.5 所示：

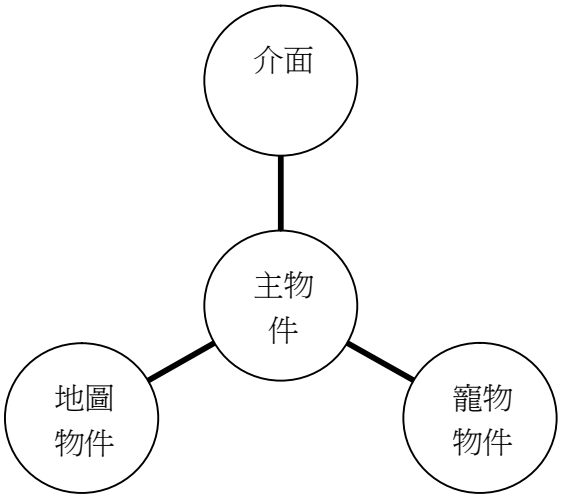


圖 3.5　虛擬寵物核心架構

主物件

類別名稱：CMyD3DApplication

說明：視窗程式框架，負責視窗的建立、訊息處理、3D 環境建立、3D 物件初始化、輸出入控制、繪圖引擎及其他物件的初始化。

成員變數：

型態	變數名稱	說明
LPDIRECT3D8	m_pD3D	Direct3D 物件
LPDIRECT3DDEVICE8	m_pd3dDevice	Direct3D 裝置
BOOL	m_bLoadingApp	程式是否正在讀取資料
ID3DXFont*	m_pD3DXFont	負責輸出文字

UserInput	m_UserInput	儲存 INPUT 資料
MainMap*	m_pMainMap	地圖物件，整個 3D 世界都是由此物件負責
Monster*	m_pPoring[]	寵物物件，負責所有寵物的圖形、行爲、繪製、狀態等等
Int	m_nRenderIndex[]	深度緩衝區
Interface*	m_pMainIF	介面物件，負責介面的初始化及繪圖
FLOAT	m_fLeft	世界左邊界
FLOAT	m_fRight	世界右邊界
FLOAT	m_fTop	世界上邊界
FLOAT	m_fDown	世界下邊界
FLOAT	m_fTheta	目前觀測角度
FLOAT	m_fMoveSpeed	移動速度
FLOAT	m_fRotSpeed	旋轉速度
D3DXVECTOR3	m_vecCurLoc	目前位置
D3DXVECTOR3	m_vecCurLookAt	目前觀測位置
D3DXVECTOR3	m_vecUp	目前位置的法向量

成員函式：

InitDeviceObjects	初始化場景物件
RestoreDeviceObjects	恢復場景物件
InvalidateDeviceObjects	釋放裝置物件
DeleteDeviceObjects	刪除裝置物件
Render	繪圖
FrameMove	處理人物移動
FinalCleanup	應用程式結束時執行，最後清除物件
ConfirmDevice	確認顯示裝置
RenderText	輸出文字

UpdateInput	更新輸入的資料
ReadSettings	讀取在 Windows 中登錄的設定
WriteSettings	登錄設定

地圖物件

類別名稱：MainMap

說明：3D 世界的建構，Vertex、貼圖

成員變數：

LPDIRECT3DVERTEXBUFFER8	m_pVB;	頂點緩衝區物件
LPDIRECT3DTEXTURE8	m_pTexture[2];	貼圖物件
LPDIRECT3DTEXTURE8	m_pRailingTexture;	貼圖物件
LPDIRECT3DTEXTURE8	m_pTreeTexture;	貼圖物件
CD3DMesh*	m_pSkyBox;	3D 模型物件
CD3DMesh*	m_pWall;	3D 模型物件
D3DXVECTOR3	TreeLoc[40];	場景中物件的位置
FLOAT	m_fLeft;	左邊界
FLOAT	m_fRight;	右邊界

成員函式：

Render	繪圖
InitObject	初始化場景中的所有 3D 模型的頂點、貼圖資料等
RestoreDeviceObjects	恢復裝置物件
InvalidateDeviceObjects	釋放裝置物件

寵物物件

類別名稱：Monster

說明：負責所有寵物的圖形、行為、繪製、狀態等等

成員變數：

LPDIRECT3DVERTEXBUFFER8	m_pVB;	頂點緩衝區物件
LPDIRECT3DTEXTURE8	m_pTexture[8];	貼圖物件
LPDIRECT3DTEXTURE8	m_pMindTexture[3];	貼圖物件
//物件屬性		
FLOAT	m_fHeight;	寵物的高度
FLOAT	m_fWidth;	寵物的寬度
FLOAT	m_fTheta;	面向的角度
FLOAT	m_fDistance;	與玩家的距離
D3DCOLOR	m_Color;	基底色彩
D3DXVECTOR3	m_vecLoc;	寵物位置
//寵物狀態		
STATE	MonState;	寵物狀態，包含體重、年齡、飢餓度、心情指數
int	m_nMindMode;	寵物目前的心情表情
int	m_nMindCtr;	顯示表情的計數
//frame 屬性		
FLOAT	m_fRenderTheta;	繪圖參考用角度
int	m_iCtr;	寵物移動計數
int	m_iLR;	繪圖用變數
int	m_iFB;	繪圖用變數
int	m_iWaitTime;	寵物動作變數
//行爲　移動		
FLOAT	m_fMoveX;	移動目標的 X 值
FLOAT	m_fMoveZ;	移動目標的 Z 值
FLOAT	m_fLength;	移動距離
bool	m_bStop;	寵物是否移動中

成員函式：

//繪圖	
Render	繪圖
InitObject	初始化寵物模型的頂點、貼圖、位置、色彩等資訊
InvalidateDeviceObjects	釋放裝置物件
//寵物行爲	
RandomMove	隨機移動
Behavior	與玩家的互動處理
//存取函式	
getLoc	取得寵物位置
getTheta	取得寵物面向角度
getDistance	取得與玩家距離
getState	取得寵物狀態

3.3 場景繪製核心

第一人稱視角的移動

遊戲中玩家的視角是採用第一人稱視角，當玩家轉彎時，就要利用對 Y 軸旋轉的旋轉矩陣作視界轉換。而當玩家移動時，就要利用對軸、Z 軸位移的位移矩陣作視界轉換，才能改變玩家所看到的畫面。

下面的旋轉矩陣將原來的點(x、y、z)繞 Y 軸旋轉 θ 角：

$$[x'y'z'1]=[x\ y\ z\ 1]\begin{bmatrix}\cos\theta & 0 & -\sin\theta & 0\\ 0 & 1 & 0 & 0\\ \sin\theta & & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

下面的位移矩陣將點(x、y、z)移動到新的點(x', y', z')，T 是位移量：

$$[x'y'z'1]=[x\ y\ z\ 1]\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ T_x & T_y & T_z & 1\end{bmatrix}$$

然後依據使用者的輸入改變視界矩陣：

```
D3DXMATRIX matView;
if( m_UserInput.bRotateLeft
    && !m_UserInput.bRotateRight )        //往左轉
{
   m_fTheta += m_fRotSpeed;               //旋轉角度
   //建立旋轉矩陣
   D3DXMatrixRotationY( &matView,
               D3DXToRadian(-m_fRotSpeed));

   //作用旋轉矩陣
```

```
   D3DXVec3TransformNormal(
            &m_vecCurLookAt,
            &(m_vecCurLookAt-m_vecCurLoc),
            &matView );
   m_vecCurLookAt += m_vecCurLoc;
}
else if( m_UserInput.bRotateRight
         && !m_UserInput.bRotateLeft )   //往右轉
{
   m_fTheta -= m_fRotSpeed;              //旋轉角度
   //建立旋轉矩陣
   D3DXMatrixRotationY( &matView,
                D3DXToRadian(m_fRotSpeed) );
   //作用旋轉矩陣
   D3DXVec3TransformNormal(
            &m_vecCurLookAt,
            &(m_vecCurLookAt-m_vecCurLoc),
            &matView );
   m_vecCurLookAt += m_vecCurLoc;
}
if( m_UserInput.bRotateUp
      && !m_UserInput.bRotateDown )      //往前移動
{
   //位移
   D3DXVECTOR3 offset = D3DXVECTOR3(
      m_fMoveSpeed*cosf(D3DXToRadian(m_fTheta)),
      0.0f,
      m_fMoveSpeed*sinf(D3DXToRadian(m_fTheta)) );
   m_vecCurLoc   += offset;
      m_vecCurLookAt   += offset;
}
else if( m_UserInput.bRotateDown
            && !m_UserInput.bRotateUp )  //往後移動
```

```
{
   D3DXVECTOR3 offset = D3DXVECTOR3(
      m_fMoveSpeed*cosf(D3DXToRadian(m_fTheta)),
      0.0f,
      m_fMoveSpeed*sinf(D3DXToRadian(m_fTheta)) );
   m_vecCurLoc    -= offset;
   m_vecCurLookAt -= offset;
}
//產生視界轉換矩陣
D3DXMatrixLookAtLH( &matView, &m_vecCurLoc,
                    &m_vecCurLookAt, &m_vecUp );
//設定觀測矩陣
m_pd3dDevice->SetTransform( D3DTS_VIEW, &matView );
```

3D 場景建構

在場景的建立方面，為了讓繪製的場景更真實，我們在 3D 物件上使用貼圖來模擬真實視界所看到的影像及環境。

要使用貼圖首先必須指定貼圖座標。所謂貼圖就是一個 2 維的色彩值陣列。陣列中的每個元素都有一個唯一的貼圖位址，基本上就是行和列的位址，分別定義成 u（列）和 v（行）。這個位址稱為貼圖座標，是以貼圖本身的座標空間來表示的。在貼圖空間中的位址則是相對於貼圖原點。Direct3D 要求貼圖中的所有貼圖圖素都用統一的位址範圍，才能把貼圖圖素對應到物件上。為了達成這個目的，Direct3D 使用正規化的位址體系。

```
struct ThreeDVERTEX
{
   D3DXVECTOR3   position;      //頂點座標
   D3DCOLOR      color;         //色彩
   FLOAT         tu,tv;         //貼圖座標
};
```

```
#define D3DFVF_ThreeDVERTEX (D3DFVF_XYZ|D3DFVF_DIF
FUSE|D3DFVF_TEX1)

ThreeDVERTEX* pVertices;
//需先鎖定才能填入模型資料
if( FAILED( hr = m_pVB->Lock( 0, 0,
                    (BYTE**)&pVertices, 0 ) ) )
      return DXTRACE_ERR_NOMSGBOX( "Lock", hr );

// 建立一個地板模型
FLOAT range = m_fRight - m_fLeft;     //地板範圍
for( int i=0; i<4; i++ )
{
   //一個四方形的地板
   pVertices[i].position  = D3DXVECTOR3(
           (float)pow(-1,i/2)*-range*5.0f,
           0.0f,
           (float)pow(-1,i+1)*range*5.0f );
   pVertices[i].color     = 0xffffffff;
   pVertices[i].tu        = (i/2)*range*2.0f;
   pVertices[i].tv        = ((i+1)%2)*range*2.0f;
}
// 建立樹木模型
for( i=4; i<8; i++ )
{
   pVertices[i].position  = D3DXVECTOR3(
                0.0f,
                (i%2)*3.02f*2.0f,
                (float)pow(-1,i/2)*1.33f*2.0f );
   pVertices[i].color     = 0xffffffff;
   pVertices[i].tu        = ((i-4)/2)*1.0f;
   pVertices[i].tv        = ((i+1)%2)*1.0f;
```

```
}
//讀入草地貼圖檔案
D3DXCreateTextureFromFile(   pd3dDevice,
                             "草皮.jpg",
                             &m_pTexture[0] );
//讀入樹木貼圖檔案
D3DXCreateTextureFromFile(   pd3dDevice,
                             "tree02S.tga",
                             &m_pTreeTexture
//設定世界轉換
D3DXMatrixIdentity( &matWorld );
pd3dDevice->SetTransform( D3DTS_WORLD,
                          &matWorld );
//指定貼圖
pd3dDevice->SetTexture( 0, m_pTexture[0] );
//繪製地板模型
pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP,
                           0, 2 );
//設定 ALPHA 混合透明
pd3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE
                          , TRUE );
//設定 ALPHA 型態
pd3dDevice->SetRenderState( D3DRS_SRCBLEND
                          , D3DBLEND_SRCALPHA );
pd3dDevice->SetRenderState( D3DRS_DESTBLEND
                          , D3DBLEND_INVSRCALPHA );
//設定 ALPHA TEST
pd3dDevice->SetRenderState( D3DRS_ALPHATESTENABLE
                          , TRUE );
//設定 ALPHA 值
pd3dDevice->SetRenderState(  D3DRS_ALPHAREF
                          , 0x08 );
pd3dDevice->SetRenderState( D3DRS_ALPHAFUNC
```

```
                              , D3DCMP_GREATEREQUAL );
//繪製樹木
pd3dDevice->SetTexture( 0, m_pTreeTexture );
for( int j=0 ; j<40 ; j++ )
{
   //讓樹木面向畫面繪製
   D3DXMatrixRotationY(   &matRotWorld,
                          -D3DXToRadian( theta ) );
   //根據隨機值改變樹木的位置
   D3DXMatrixTranslation( &matWorld,
                          TreeLoc[j].x,
                          0.0f,
                          TreeLoc[j].z );
   D3DXMatrixMultiply( &matWorld,
                       &matRotWorld,
                       &matWorld );
   //設定世界矩陣
   pd3dDevice->SetTransform( D3DTS_WORLD,
                             &matWorld );
   //繪製
   pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP,
                              8, 2 );
   D3DXMatrixIdentity( &matWorld );
   pd3dDevice->SetTransform( D3DTS_WORLD,
                             &matWorld );
}
```

3.4 寵物行為

在一般情形下，寵物會隨機移動，下面的函數執行此工作。

```
void Monster::RandomMove( D3DXVECTOR3 loc )
{
   if( m_bStop )
   {
      if( m_iWaitTime >= 500 )
      {
         m_bStop = false;
         if( m_fMoveX == 0.0f && m_fMoveZ == 0.0f )
         {
            m_fMoveX = ((rand()%800)-400)/100.0f;
            m_fMoveZ = ((rand()%800)-400)/100.0f;
         }
         m_fTheta = D3DXToDegree( atan2f( m_fMoveX,
 m_fMoveZ ) );
         m_fLength = (FLOAT)sqrt( m_fMoveX*m_fMoveX
 + m_fMoveZ*m_fMoveZ );

         m_fMoveX = m_fMoveX/(m_fLength/0.01f);
         m_fMoveZ = m_fMoveZ/(m_fLength/0.01f);
         m_iWaitTime = 0;
      }
      else
         m_iWaitTime ++;
   }
   else
   {
      m_vecLoc.x += m_fMoveX;
      m_vecLoc.z += m_fMoveZ;
```

```
      m_fLength -= 0.01f;
      if( m_fLength<=0 )
      {
         m_fMoveX = 0.0f;
         m_fMoveZ = 0.0f;
         m_bStop = true;
      }
      m_fDistance = D3DXVec3Length( &(m_vecLoc - lo
c) );
   }
}
```

當玩家對寵物作某些動作時，寵物會做出反應。

```
void Monster::Behavior( int BehMode, D3DXVECTOR3 lo
c )
{
   switch( BehMode )
   {
      //摸，寵物心情會變好，也會轉過來面向玩家
      case BehMode_TOUCH :
         if( MonState.mind < 100 )
            MonState.mind += 5;
         if( MonState.mind > 100 )
            MonState.mind = 100;
         m_nMindMode = MindMode_HEART;
         m_nMindCtr = 100;
         m_bStop = true;
         m_iWaitTime = 0;
         m_fTheta = m_fRenderTheta+180;
         break;
      //餵食，增加飽食度，心情也會變好
      case BehMode_FEED :
```

```
            m_fMoveX = loc.x - m_vecLoc.x;
            m_fMoveZ = loc.z - m_vecLoc.z;
            m_bStop = true;
            MonState.hungry += 2;
            m_iWaitTime = 500;
            m_nMindCtr = 100;
            m_nMindMode = MindMode_NODE;
            break;
         //打，心情會變差
         case BehMode_HIT :
            m_nMindMode = MindMode_ARGRY;
            if( MonState.mind > 0 )
               MonState.mind -= 5;
            if( MonState.mind < 0 )
               MonState.mind = 0;
            m_nMindCtr = 100;
            m_bStop = true;
            m_iWaitTime = 0;
            m_fTheta = m_fRenderTheta+180;
            break;
      }
   }
```

第四章 交易市場與網路

4.1 市場 CG 的顯示

BMP

Windows 所處理的 BMP 可分為下面 3 種：

- DDB(Device Dependent Bitmap;與設備有關的 BMP 格式)
  由 HMITMAP 行處理常式來處理，使用 CreateBitmap 等產生
- DIB(Device Inpendent Bitmap;與設備無關的 BMP 格式)規定記憶體的格式，沒有產生的 API
- DIBSection(兼具以上兩者特長的 BMP 格式)使用 CreateDIBSection 產生

為什麼會有 3 種不同的 BMP 格式？原因應該是在早期 Windows 系統上無法有效處理與設備無關規定記憶體上規格的 BMP 格式(即 DIB)，DDB 雖然跟設備有關但處理速度快，而 DIB 因為與設備無關，所以能直接從程式端操作記憶體，即使有百般的不情願，當時的電腦配備規格仍然必須同時具備這兩種不同的格式，第三種 DIBSection 為後來才新增的格式。以現在的電腦配備規格(顯示卡)來看，他對 DIB 的 DDB 繪製速度幾乎一樣，因此如果需要直接操作記憶體影像並顯示在畫面上，直接用 DIBSection 也沒問題。如果不顯示到畫面上的話，使用 DIBSection 會消耗珍貴的 Windows 系統資源，DIB 既然有能力進行跟 DIBSection 影像同樣的處理，除非有其他特殊理由，否則應該使用 DIB 會比較好。那麼，「什麼時候才要用 DDB，答案定必須有 HBITMAP 型處理常式，且不曾操作到記憶體影像;換句話說，如果程式有需要的時候才用 DDB。這是「只能這麼做」的特殊情形，不必煩惱要用哪一種 BMP 格式，應該很容易分辨(也許還有其他 DDB、DIB 曾經存裡的原因，不過這都已經是「過去式」。

DIB 的架構

在此先把 DIE 建立和操作整理成類別，以便後面使用。DIB 不是用 Windows 的 API 產生，而是要根據 Windows 系統的「規則」才能產生。先介紹一下 DIB 的架構。

DIB 是圖形影像 ，具有尺寸大小 、色彩等的資訊 。這些資訊都記錄在「BITMAPINFO 結構體」上。處理 DIB 的 API(SetDIBitsToDevice 等)則使用 BITMAPINFO 的指標(pointer)做為參數。

BITMAPINFO 結構體的格式如下

```
typedef struct tagBITMAPINFO
{
   BITMAPINFOHEADER    brniHeader;
   RGBQUAD             bmiColors[l];
}BITHAPINFO;
```

DIB 的大小、色彩則記錄在 BITMAPINFOHEADER 結構體裡。BITMAPINFOHEADER 結構的格式如下：

```
typedef struct tagBITMAPINFOHEADER
{
   DWORD bisize;
   LONG  biwidth;
   LONG  biHeight;
   WORD  biBitCount;
   DWORD bicompression;
   DWORD biSizelmage;
   LONG  biXPelsPerMeter;
   LONG  biYPelsPerMeter;
   DWORD biCIrUsed;
   DWORD biCIrUsed;
```

```
   DWORD biCIrImportant;
}
BITHAPINFOHEADER, *PBITMAPINFOHEADER.;
```

DTB 的可顯示色彩包括有「2 色(l bpp)」、「16 色(4 bpp)」、「256 色(8 bpp)」、「65,536 色(16 bpp)」和「1,677 萬色(24/32 bpp)」，到「256 色」為止的 DIB 足「調色盤」上的系統色彩。 在這些色彩顯示模式常中，l bpp 和 4 bpp 比較接近用 1 byte 處埋多個像素的感覺，所以操作比較複雜。其實這兩種顯示模式所能表現的色彩並不多，各位可以暫時把它放在一旁。

BITMAPINFOHEADER 之成員變數的設定值即如表所示。

成員變數	設定值
biSize	BITMAPINFOHEADER 結構的大小(sizeof(BITMAPINFOHEADER).
biWidith	DIB 寬
biHeight	DIB 高
biplanes	永遠設為 1
biBitCount	每一像素的位元數(bit per pixel)
biCompression	設定 Bl_RGB
biSizeImaqe	整個圖形的位元組數
biXPelsPerMeter	設為 0
biYPelsPerMeter	設為 0
biCirUsed	設為 0
biClriImportant	設為 0
biXPelsPerMeter · biYPelsPerMeter	是設定解析度(即每公尺所含像素數)，但程式範例並不會用到，故設為「0」

如果有先寫好標頭檔(header)，後面只要再保留圖形要用的記憶體空間即可，不過這裡也有一定的規則。這條規則就是"每條掃描線要包括 4byte”，所以要做一個照條件進位的換算動作。DIB 的掃描線

(scan line)是指「1 條橫線」。照條件進位的計算如下：求出 1 條掃描線的必要 byte 數

```
inline unsigned CDib::ScanBytes(int pixWidth,int pixDepth)
{
        return (unsigned)(((long)pixWidth * pixDepth + 31)/ 32) * 4;
}
```

pixDepth 則是位元數，計算單位是 1 個位元。計算式是先加 31 後再除以 32，所得商數即為 byte 數，最後再乘以 4 就可以得知該圖形換算後應該是幾個 4byte。將前面所得數值乘上圖形高度，即可得到必要 byte 數

```
bitsAlloc = ScanBytes(Width, detth) * height;
```

保留這個數量的記憶體空間之後，DIB 的產生過程也到此結束。這裡使 GlobalAlloc 來保留足夠的記憶體空間，如想改用 new、HeapAlloc 也沒關係，當然也可以自己另寫一個記憶體空間保留函式。只要能「保留記憶體空間」的動作就行。

讀入 BMP 檔-CDib::LoadBMP

DTB 類別的程式碼中還有另一個「讀入 BMP 檔」的函式，這也是一個很重要的函式。BMP 檔的作用不只是要簡化結構而已，BMP 的格式還要能把 DIB 直接轉成檔案，資料排列等都要跟 DIB 一模一樣。

BMP 檔的資料結構

BITMAPFILEHEAD	表示這是 BMP 檔的標頭
BITMAPINFO BITMAPINFOHEADER RGBQUAD	DIB 寬、高等相關資訊(如有必要時，則爲調色盤的資訊)
位元影像-	

接在 BITMAPFILEHEADER 的後面是 BITMAPINFO，裡面的資訊則跟產生 DIB 時所使用的資訊一樣。後面的位元影像也是跟 DIB 同樣的陣列。

地圖的顯示

CG 的顯示與圖形合成

重疊處理是要在記憶體上完成。如果等到顯示時才重疊在一起，會變成兩段式先「繪製背景」、再「繪製重疊的 CG，，可能會只顯示背景而已。

在合成背景和人物時，記憶體上必須保留「背景」、「人物」和「顯示圖形」這三個 CG 影像(假設現在想把另一個人物跟背景重疊峙，只要記憶體一直保留著背景 CG，就不需要重新讀取 CG 到記憶體上)。當 CG 資料可以「用完就丟」，當然就不需要保留了。

製作合成用的類別

合成的第一步驟是先把背景複製到顯示用影像 (記憶體)，按著冉跟人物合成。這兩個動作需要有「複製」和「合成」的函式。

有時直接把 CDIb 類別拿來用也不一定有好處。CDib 類別不會決定色彩，因此如不讓它能決定色彩的話，就必須設計一個”不受色彩影響”的函式關於地圖的顯示，必須先有一個座標係，才能顯示地圖或人物還要按照 8bpp、16bpp、24bpp 的分類等級一一設計其專用類別，實在是工程浩大，而實際又不是每個都非要不可，所以我們選擇繼承 CDib 再設計一個 24bpi 專用的類別。合成用的類別 lmagc.Cpp 則繼承 CDIb，只修改必要的部分。

```
//合成用類別 Image.h
class CDC;
class Cimage:
```

```
{
public:
   CImage ( ) : CDib( )  {}
Cimage ( int  width ,  int  height);
BOOL Create (int width, int height);
void Copy(const CImage *image, const CRect &rect);
void Copy(const CImage *image);
void MixImage (const CImage *image, const CRect &re
ct,
COLORREF trans=RGB(0, 255, 0));
} ;
//成員函式
inline void Cimage::Copy(const CImage *image)
{
Copy (image, CRect(0, 0, image->Width( ), image->He
ight( )))
}
-------------------------------------------------
//合成用類別 Image.cpp
//建構式
CImage::CImage( int  width  ,  int  height)
{
Create(width, height);
}

//產生 DTB
BOOL  CImage:: Create (int width, int height)
{
return CDib;:Create(width, height, 24);
}
//複製區域
voidCImage::Copy(const CImage *image, const CRect &
rect)
```

```
int        len = rect.Width ( ) * 3;
for (int y=rect .top; y<rect .bottom; y++)
{
memcpy(GetBits (rect.left, y), image->GetBits(rect.
left, y), len);
}
//複製(有考慮到透明色部分)
voidCIage::MixImage(const  CImage*image , const  CR
ect  &rect ,
COLORREF trans_color)
const unsigned char trans b = GetBValue(trans colo
r);
const unsigned char trans_g = GetGValue(trans_colo
r);
const unsigned char trans_r = GetRValue(trans_colo
r);
for(int y=reck .top; y<rect .bottom; y++) {
byte_t *p = (byte_t *)GetBits(rect. left, y);
const byte_t *q = (byte_t *)image->GetBits(rect. le
ft, y);
for (int x=rect.left; x<rect.right; x++) {
const byte_t  b = *q++;
const byte_t  g = *q++;
const byte t  r = *q++;
if(b!=trans_b||g!=trans_g||r!=trans_r)
{
p[0] = b;
p[1] =g
p[2] = r;
}
p + =3;
}
}
```

```
}
```

複製用的函式

雖然講是講「圖形」，不過其實是記憶體上的資料，因此複製的動作可使用「memCpy」。

```
void  CImage::Copy(const  CImage *image,const  CRec
t &rect)
{
int  len = rect.Width( ) * 3;
for( int y = rect.top;y<rect.bottom;y++)
{
memcpy(GetBits(rect.left,y),image->GetBits(rect.le
ft,y),len);
}
)
```

寬度的地方乘以 3」是因為「1 像素=:3byte」的關係。不斷複製必須要的高度，即可複製圖形。

合成

合成的過程中必須決定出一個規則，否則程式無法判斷哪些部分要合成、哪些部分又不需要合成，當然就不會進行複製。這個判斷的方法必須要訂清楚。說穿了，就足捉供「遮罩圖形」的方法或設定「透明色彩」的方法。

如欲使用遮罩圖形，則須另行製作遮罩用的圖形，所以 CG 製作比較麻煩。我們使用透明色彩來進行合成。至於透明色彩的選擇方式，最常使用的是 RGB(0，255.0)，的亮綠色。當然也可以自訂其他色彩，不過最好是“非常極端、CG 不會用到的色彩。

```
const byte_t  b = *q++;   藍
```

```
const byte_t  g = *q++;   綠
const byte_t  r = *q++;    紅
//是否爲透明色彩?

if(b!trans_b||g!=traans_g||r!=trrans_r)
{
   p[0]=b;          //藍色部分的複製
   p[1]=g;          //綠色部分的複製
   p[2]=r;          //紅色部分的複製
    p+=3;
}
```

合成處理是先在迴圈中逐一判斷各個像素足 「透明色彩」或「非透明色彩」，若為「非透明色彩」則進行複製。

如果不控制 DIB 的色彩數量，這個處理動作會越變越複雜，因此這裡設限為 24bpp。若為 16bpp 或 32bpp，則須另外設計其他適合的常式。

座標系的轉換

如果使用的地圖是垂直向下看的方格地圖，畫面顯示的座標係級同地圖的座標系(只需放大或縮小)，不必花太多時間，但如是 45 度全視角的畫面，就需要轉換座標值實作時，先作出平面的地圖再配合轉換座標即可作成 45 度的地圖必須先有一一個座標系，才能顯示地圖或人物。

如果使用的地圖足垂百向下看的方格地圖，畫面顯示的座標系即同地圖的座標丟(只須放大或縮小)，不必太花時間，但如果是 45 度全視角的畫面，就需要轉換蝴值。

不先處理好座標轉換的問題，甚至會無法做移動測試。45 度全視角的座標系即如圖所示。

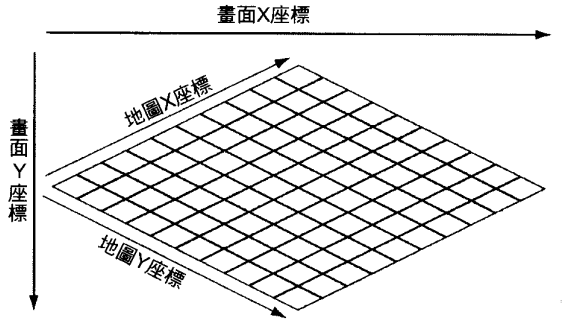


圖 4.1　45 度全視角的座標系

從地圖座標轉換成畫面座標

我們先想想從地圖座標轉換成畫面座標時的轉換式。轉換成畫面座標時，就已經從從「點」變成「方格」，所以要求出框住菱形外圍的矩形的「左上角」座標。

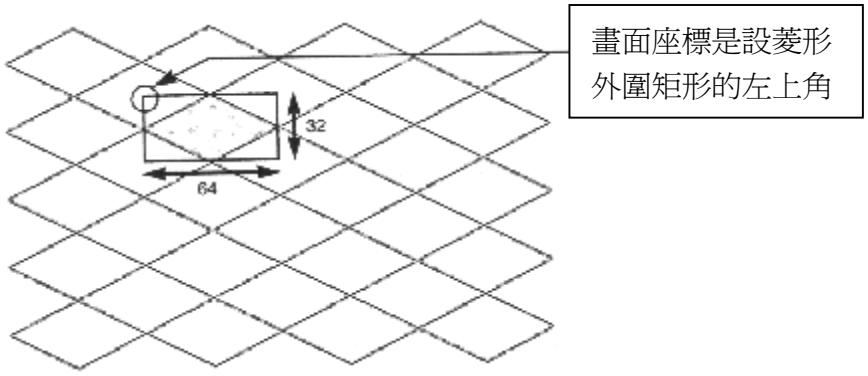


圖 4.2　畫面座標的位置

框住菱形外圍的矩形大小是「64X32」，而在 640X480 的畫面(程式區域)上則可顯示 10X10 的地圖。因為方格間有互相重疊，所以畫面座標的最小單位是這個尺寸大小的 1/2(長 X 寬=16X32)。

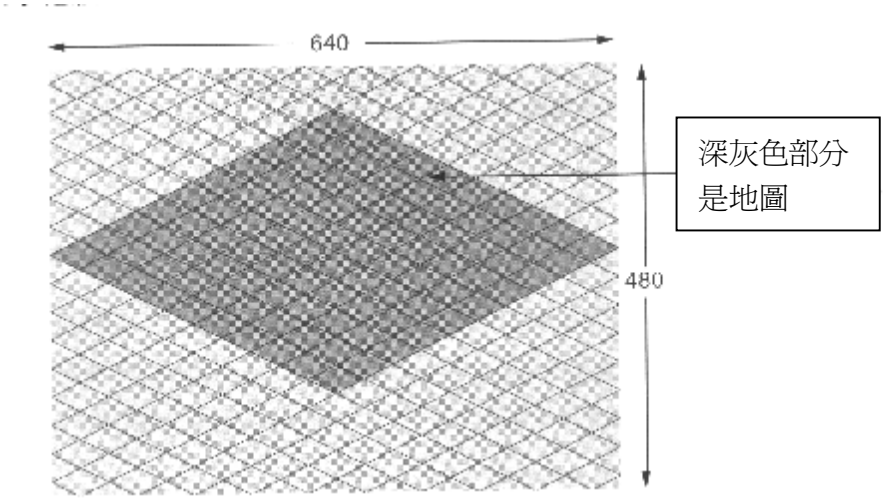


圖 4.3 矩形大小與介面的關係

這時候，可以先用一個比較容易算的數值試算看看。最容易算的數值當然是「0」。地圖座標為「0，0」時，則畫面座標為「x=0」，Y 座標則偏移 6 個方格(最小單位 12:，故可得出下列公式：

```
MAPGRID_WVIDTH  =64
MAPGRID_HEIGHT=32
```

```
view_x=0
view_y=12*(MAPGRID_HEIGHT/2)
```

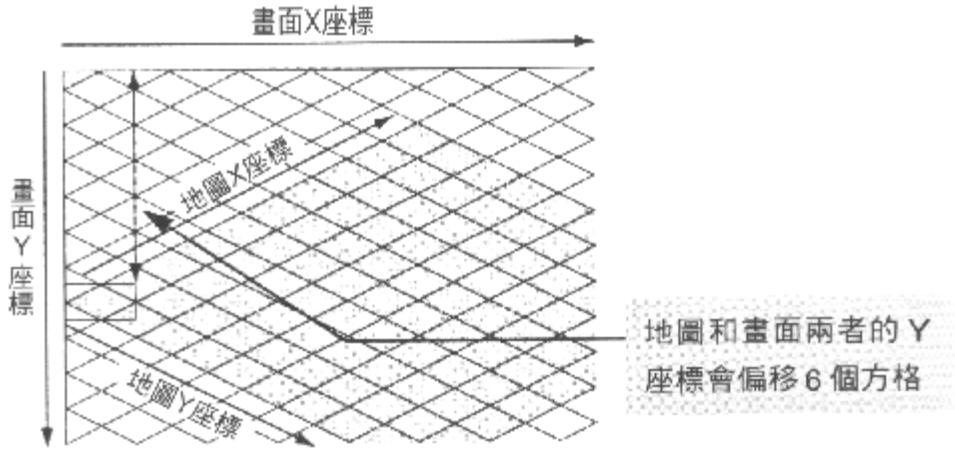


圖 4.4 地圖座標原點

地圖 X 座標(map_X)每增加 1，畫面 Y 座標(view_Y)就會相對減少 1。也就是說

```
view_x=map_x*(MAPGRID_WIDTH/2)
view_y=(12-map_x)*(MAPGRID_HEIGHT/2)
```

而地圖 Y 座標(Map_Y)每增加 1，畫面 X 座標(view_X)也同樣增加 1

```
view_x=(map_x+map_y)*(MAPGRID_WIDTH/2)
view_y=(12-map_x+map_y)*(MAPGRID_HEIGHT/2)
```

地圖座標和畫面座標兩者的轉換公式即如上所示。

從畫面座標轉換成地圖座標

反過來的話，要如何從畫面座標求出地圖座標?

公式轉換的計算單位是像素，不過像素座標都是整數，如果求出的結果不是整數時還是會強迫改成整數。因此計算結果可能會有加減 1 像素的誤差(這是無條件進位或無條件捨去時所產生的誤差，所以或許可以視為一定會有誤差)。

利用漸層色地圖

先在記憶體上做個「漸層色地圖」搭配使用也是一種方法。

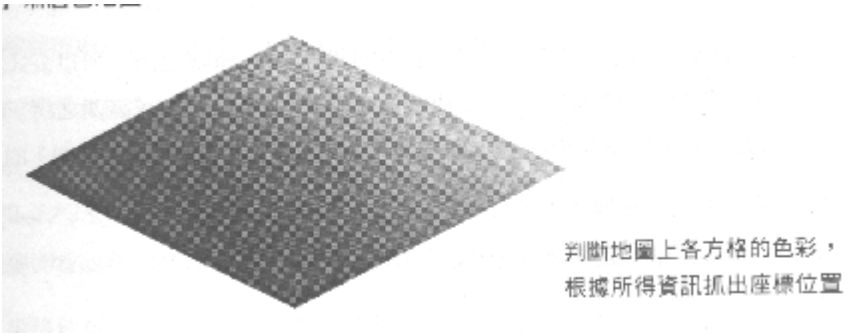


圖 4.5 漸層色地圖

這種方法的優點是能適用於各種不同形狀的地固。受限於印刷問題，雖然上圖使用黑白兩色，實際上 X 座標用紅色、Y 座標用藍色會更容易判斷座標位置。

因為這裡的目的只是為了「用滑鼠指出」，加不需要太嚴格要求，所以先用計算式就好了。

座標轉換的計算式

首先，方格的長寬比例是「2:1」，所以畫面 Y 座標要放大 2 倍。然後地圖座標也要從方格單位改成像素單位，即地圖座標乘以 64 倍，0~63 為「0」、64~127 為「11…．以此類惟(因為畫面座標是以像素為計算單位) 試求畫而座標「0，0」時的地圖座標。地圖 Y 座標跟畫面座標總共偏移了 6.5 個方格，所以要把偏移的部分補正回來。

```
view_y=MAPGRID_HEIGHT*13/2;
```

因為這是整數運算，所以要寫成 13/2 而非 6.5.接著，再把 Y 座標放大 2 倍

```
View_ y* = 2;
```

最後再套用地圖座標轉換成畫面座標峙的模式，即可得到下列公式

```
map_x = (view_x-view_y)/MAPGRID_WIDTH
maP_y = (view_x+view_y)/MAPGRID_WIDTH
```

乍看之下，這個公式似乎符合基本的數學邏輯，但是裡面隱藏著 bug。

地圖座標轉換成畫面座標時的計算都在地圖座標的範闊之內，所以公式運算不會有問題，但從畫面座標轉換成地圖座標時卻需要考慮到不在範圍之內的部分「不在範圍之內」是指在畫面上(視窗內)、但未顯示菱形方格(地圖)的部分，不在範圍之內的座標值可能會讓「view_x-view_y」或「view_x+view_y」變成負數。這裡本來就是整數運算，故計算結果如圖所示，0 的長度會明顯比其他數字長。

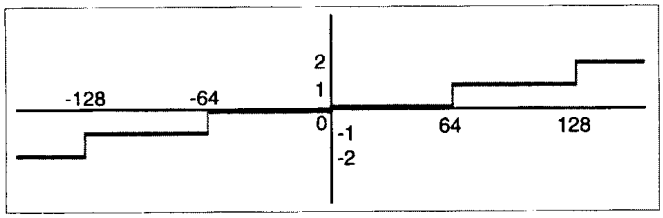


圖 4.6 長條圖

如此一來，當結果出現負數時就會偏移 1 個方格，也沒辦法直接兩數相除所以參考以下解決方法。

解法一 :實數運算。如果問題是出在整數運算，改用實數運算就可以解決問題。

實數運算(浮點小數運算)可以使用「floor」函式。利用 flool 函式就能得到預期中的數值(「floor(x)」會傳回表示”x 以下的最大整數”的浮點小數值)。

應用 floor 的公式即為

```
map_x=int(floor(double(view_x-view_y)/MAPGRID 一 WID
```

```
TH
map_y=int(floor(double(view_x+view_y)/MAPGRID一WID
TH
```

解法(2):應用移位 (shift) 。即使直接做整數運算，有時不用除算改用移位(shift)反而能得到預期中的數值。

應用移位的公式即為:

```
map_X = (view_x-view_y)>>6
map_y = (VieW_x+view_y)>>6
```

移位分成「算術移位(Shiftarithmeti。)」和「邏輯移位(Shiftlogical)」兩種。

利用算術移位可以得到想要的結果，但邏輯移位會讓負數變成很大的正數。其實就算變成再大的正數，都還是「不在範圍之內」，所以不成問題。 不過，移位只能利用在 2 的乘幕計算(2、4、8、16·……等)而已，用途比較有限。

解法 3:加上 offset。因為計算中出現負數才會發生問題，那把計算範圍限制為正數就好了。

```
map_x=(view_x-view_y+MAPGRID_WIDTH*10)/MAPGRID_WID
TH-10
map_y=(view_x-view_y+MAPGRID_WIDTH*10)/MAPGRID_WID
TH-10
```

如果偏移的方格數目高達 10 個，確實就不在範圍之內。只要事先加上適當數值，等到除算後再減去該數值即可。(若不在範圍之內實不需要正確數值，則 offset 可設為「1」。Offsrt 值為 1 時，-1 的範圍會變大，但-1 已經不在範圍之內，所以沒關係。

解法(4):分類處理。利用 if 敘述分成負數、正數兩種不同情形該如何進行計算，即可得到正確的計算結果。

不過，最近的 CPU 不太擅長處理條件分支，而且一進入條件分支之後來源程式碼的「流向」就會被分開，降低整個程式的可閱讀性，所以很多時候不一定非用分類處理的做法不可。

4.2 DirectPlay

主要提供網路連結的環境，並且允許軟體開發者提供多玩家參與的功能，而無需擔心傳輸媒介帶來的複雜問題。DirectPlay 統一處理各種傳輸媒介，無論在 TCP/IP 網路、IPX 網路或數據機。在 DirectPlay 下層的服務供應器(Server Provider)會協助使用這些不同的溝通媒介，並用特定的傳輸方式處理所有通訊相關事宜。

Direct Play 架構

微軟建立一套通訊傳輸使用的裝置驅動程式介面，實際上在 WOSA(The Windows Open System Architecture)中把這些裝置驅動程式稱為服務供應器(Service Provider)。它有幾項優點：

- 為軟體開發者提供一致的介面
- 隱藏不同的通訊協定之間的不同
- 允許 Third-party 廠商撰寫他們自己的服務供應器

遊戲或應用軟體會與管理多個服務供應器的 DirectPlay 溝通。DirectPlay 本身附有四種服務供應器：Network IPX / SPX、Network TCP / IP、Modem 以及 Serial Connection。網路服務供應器以 WINSOCK 為其基礎，而數據機服務供應器則以 Windows 的 TAPI(Telephony Application Programming Interface)為基礎。

<table>
<tr><td colspan="4">Game</td></tr>
<tr><td colspan="4">DircetPlay</td></tr>
<tr><td rowspan="2">Service Provider</td><td>Modem Service Provider</td><td>Network Service Provider</td><td rowspan="2">Service Provider</td></tr>
<tr><td>TAPI</td><td>WINSOCK</td></tr>
</table>

圖 4.7 DirectPlay 架構圖

如何用 DirectPlay 進行遊戲

DirectPlay 活動的組合稱為 session。依所使用服務供應器的不同，你可以在同一個通訊媒介上同時執行不同遊戲的許多 session。DirectPlay 提供列舉 session、建立 session、以及既存 session 的連接方法。Session 也可以用密碼加以保護並對使用者加以驗證。

一旦 session 被建立，玩家就可以互相連接了。DirectPlay 提供建立、刪除、列舉、以及描述玩家的物件方法，也可以對玩家分組以便於管理並提升通訊效率。

通訊

點對點架構

當使用網路服務供應器時，一個玩家先建立一個遊戲 session。他同時變成 session 的主電腦(session host)。主電腦回應那些來自其他 DirectPlay 物件的 session 資訊要求，每一個新加入的 DirectPlay 物件都會收到一份加入時其他 DirectPlay 物件的列表，這份列表在整個 session 建立的過程中都會被維護。因為每一個 DirectPlay 物件都知道其他物件，他們就可以直接互相傳遞訊息而不需要經過主電腦。所以這個 session 是點對點的。

主從式架構

這種狀態市由使用網際網路遊戲伺服器或撥接服務的服務供應器所產生的。這種情況下，所有訊息都必須經過主電腦。因為他更有效率地使用通訊頻寬，這種架構可以容納更多玩家。

Lobby 架構

DirectPlayLobby 物件為玩家和觀眾提供虛擬的會面處來模擬真實的大廳功能。玩家可以搜尋其他條件接近的玩家、與他們對談、設定遊戲。大廳會接手通知參加者，並同時啟動各人電腦中的應用程式。或其他廠商可以為特定遊戲或有相同興趣的參與者提供自訂的大廳，甚至可以經由收費方式達到商業目的。大廳伺服器還可以提供遊戲分數、個人偏好設定、遊戲紀錄的功能。

關於 GUID

『通用唯一識別碼』(globally unique identifiers，簡稱 GUID)被廣泛地和 COM 聯合使用以辨視物件與介面，並且在需要唯一識別碼的地方突然出現。GUID 是一個 32 位數的 16 進位序列，而且一定是獨一無二的。這有一個有效的 GUID 例子：

6B29FC40-CA47-1067-B31D-00DD010662DA

DirectPlay 使用這些序列，在通訊網路上確認 DirectPlay 物件。服務供應器、DirectPlay 的 session、DirectPlay 資料區塊，以及 DirectPlay 應用軟體都是以 GUID 來辨識，一旦要使用 DirectPlay，除了使用以外，還要建立。

要使用 DirectPlay 建立連線遊戲環境，有幾個基本的步驟，底下一一說明：

Setp 1.**建立一個** DirectPlay **物件，列舉出可使用的網路連線型態**

```
// 建立 IDirectPlay8Peer 物件
// 傳回指向 IDirectPlay8Peer 界面的指標給 hr
hr = CoCreateInstance( CLSID_DirectPlay8Peer,NULL,C
LSCTX_INPROC_SERVER,IID_IDirectPlay8Peer,(LPVOID*)
 &g_pDP);
// 初始 DirectPlay
```

```
hr = g_pDP->Initialize(NULL, DirectPlayMessageHandl
er, 0);
// DirectPlayMessageHandler 是處理回呼訊息的函式
//像 Windows 傳送 message 在溝通是靠 window procedure 在處
理一樣
```

要列舉出電腦有提供的連線服務要使用：IDirectPlay8Peer::EnumServiceProviders

這函式使用到一些以提供資料型態的 buffer 來存連線服務資訊

```
DPN_SERVICE_PROVIDER_INFO*  pdnSPInfo      = NULL;
DPN_SERVICE_PROVIDER_INFO*  pdnSPInfoEnum   = NULL;
DWORD             dwItems        = 0;
DWORD             dwSize         = 0; //初始 buffer 爲
0
DWORD              i;
// 決定實際 buffer size
hr = g_pDP->EnumServiceProviders(NULL, NULL, NULL,
&dwSize, &dwItems, 0);
pdnSPInfo = (DPN_SERVICE_PROVIDER_INFO*) new BYTE[d
wSize];
// 儲存連線服務的資訊到 buffer
hr = g_pDP->EnumServiceProviders(NULL, NULL, pdnSPI
nfo, &dwSize, &dwItems, 0)
// 列出提供的連線服務
    pdnSPInfoEnum = pdnSPInfo;
for (i = 0; i < dwItems; i++)
{  printf("Found Service Provider:  %S\n", pdnSPInf
oEnum->pwszName);
    pdnSPInfoEnum++;
}
```

Step 2.建立 DirectPlay address object，提供 Host session

```
IDirectPlay8Address*  g_pDeviceAddress = NULL;
// 給 IDirectPlay8Address Device 配 Address
hr = CoCreateInstance( CLSID_DirectPlay8Address, NU
LL,
CLSCTX_INPROC_SERVER, IID_IDirectPlay8Address,

(LPVOID*) &g_pDeviceAddress );
// 給 object address 確認有哪些提供的服務
hr = g_pDeviceAddress->SetSP(&CLSID_DP8SP_TCPIP );
HRESULT SetSP(
const GUID *const pguidSP à指向提供服務的 GUID
);
```

接下來要描述自己作為 Host 有哪些東西，使用函式：DPN_APPLICATION_DESC，可能會包含以下幾種資訊

- The application's GUID 好像身分 ID，每個 application 都有不同 ID
- 允許連線的玩家數目
- 連線名稱

```
DPN_APPLICATION_DESC    dpAppDesc;
// 描述 application
ZeroMemory(&dpAppDesc, sizeof(DPN_APPLICATION_DES
C));
dpAppDesc.dwSize = sizeof(DPN_APPLICATION_DESC);
dpAppDesc.guidApplication = g_guidApp;
// 接下來實際建立 Host
hr = g_pDP->Host( &dpAppDesc,
                  &g_pDeviceAddress, 1,
                  NULL, NULL,
```

```
                NULL,  // 有關玩家的資訊
                0);    // dwFlags 對話盒用來了解更多玩家
資訊

Step 3.列出已經存在的 Host Session

//首先描述你想要的需求服務
ZeroMemory(&dpAppDesc, sizeof(DPN_APPLICATION_DES
C));
dpAppDesc.dwSize = sizeof(DPN_APPLICATION_DESC);
dpAppDesc.guidApplication = g_guidApp;
//接下來將之前你建立的 address object 以及 Desc 當參數傳給
// EnumHost
//傳回 DPN_OK 表示尋找過程完成，或傳回失敗訊息
hr = g_pDP->EnumHosts( &dpAppDesc,  // pApplication
Desc
g_pHostAddress,   // Host Address
g_pDeviceAddress, // Device Address
NULL, 0,   // pvUserEnumData, size
4,      // dwEnumCount 設定多久時間傳回 Host 列表
0,     // dwRetryInterval 設定重傳的時間 milliseconds
0,    // dwTimeOut 我方接收到最後一個 Hostenum 後，多久回
覆
NULL,    // pvUserContext
NULL,    // pAsyncHandle
DPNENUMHOSTS_SYNC);  // dwFlags 設定一些額外要求
//當上面請求成功後，你會收到 DPN_MSGID_ENUM_HOSTS_RESPON
SE
//裡面有 session 的資訊，而 message handler 依 msg 作相對應
的處理
HRESULT WINAPI DirectPlayMessageHandler( PVOID pvUs
erContext, DWORD dwMessageId, PVOID pMsgBuffer )
{
```

```
  switch( dwMessageId )
    { case DPN_MSGID_ENUM_HOSTS_RESPONSE:
        {  //
      }
```

//全部動作完成後，你就可以則想要的 session。

Step 4.**連結** session

在連線之前有必要條件要先準備好：

- DPN_APPLICATION_DESC 裡面要包含自己的 GUID 以方便加入 session
- host address 這是在之前列舉 hosts 應該得知的
- 自己的 object address

使用函式 IDirectPlay8Peer::Connect，Host 會接收 DPN_MSGID_INDICATE_CONNECT 訊息，裡面有你的資訊。當 Host 接受連線成功後，DirectPlay message handler 接收 DPN_MSGID_CONNECT_COMPLETE 的訊息，你也會接收到 DPN_MSGID_CREATE_PLAYER 的通知，表示獲准加入。

```
DPN_APPLICATION_DESC       dpnAppDesc;
IDirectPlay8Address*       pHostAddress = NULL;
ZeroMemory(&dpnAppDesc, sizeof(DPN_APPLICATION_DES
C));
dpnAppDesc.dwSize = sizeof(DPN_APPLICATION_DESC);
dpnAppDesc.guidApplication = g_guidApp;
hr = g_pDP->Connect(&dpnAppDesc,   // Application
描述
pHostAddress,    // Host Address
g_pDeviceAddress,
NULL, NULL, NULL, 0, NULL, NULL, NULL, DPNCONNECT_S
```

```
YNC );
 if( FAILED( hr))
 {  //連線失敗所作的處理
 }
```

Step 5.**傳送** message **給其他玩家，及接收** message

使用函式 IDirectPlay8Peer::SendTo 可以傳送 message 給 session 中的每一位玩家或著指定單獨一位。

```
DPN_BUFFER_DESC dpnBuffer;
WCHAR           wszData[256]; //建一個存 data 的 buffer
dpnBuffer.pBufferData = (BYTE*) wszData; //轉換資料型態
//計算資料長度
dpnBuffer.dwBufferSize = 2 * (wcslen(wszData) + 1);
hr =
g_pDP->SendTo( DPNID_ALL_PLAYERS_GROUP, //設爲傳給每個 player
&dpnBuffer,  // 指向 BufferDesc 結構體的指標
1,          // 指向 BufferDesc 結構體的數目
0,  // milliseconds，超過這個時間未送出，msg 從 send queue 刪除
NULL,       // pvAsyncContext
NULL,       // pvAsyncHandle
DPNSEND_SYNC |DPNSEND_NOLOOPBACK ); //設定 msg 傳送的方式
```

接收 Message：

當別人送 msg 給你時你的 DirectPlay message handler 接收 DPN_MSGID_RECEIVE message 裡面有資料 buffer 供讀取訊息。

```
//接收訊息的 message handler:
HRESULT WINAPI DirectPlayMessageHandler(void *pvContext, DWORD dwMsgId, void *pvMessage)
```

```
{
switch( dwMsgId )
{
case DPN_MSGID_RECEIVE:
{  PDPNMSG_RECEIVE    pMsg;
pMsg = (PDPNMSG_RECEIVE) pMsgBuffer;
//可以輸出一些訊息
break;
   }
}
return(DPN_OK);
}
```

Step 6.Host Migration

這是在說明當連線時，整個 session 必須有一個 host，而當初建立這個 session 的 host 要離開時,是否連線就中斷呢,還是將擔任 host 的職權移交給其他 player，這裡就是要教我們如何轉移 host 權限。

要使用這項功能，要在當初建立 session 時的 host，他的 DPN_APPLICATION_DESC 中的 dwFlag 要設定 DPNSESSION_MIGRATE_HOST。

```
DPN_APPLICATION_DESC   dpnAppDesc;
DpnAppDesc.dwFlags =  DPNSESSION_MIGRATE_HOST;
```

而 Host 另外可以使用函式 IDirectPlay8Peer::TerminateSession 來強制中斷連線。當 Host 離開時，在 session 中未離開的玩家會接收 DPN_MSGID_HOST_MIGRATE 訊息，裡面有整個 session 的資訊。

```
HRESULT WINAPI
DirectPlayMessageHandler( void *pvContext,
                          DWORD dwMsgId,
                          void *pMsgBuffer)
{
   switch( dwMsgId )
   {
      case DPN_MSGID_HOST_MIGRATE:
      {
      PDPNMSG_HOST_MIGRATE   pHostMigrateMsg;
      pHostMigrateMsg =
               (PDPNMSG_HOST_MIGRATE)pMsgBuffer;
      // 判斷是否獲得成爲新的 Host
      if( pHostMigrateMsg->dpnidNewHost
                    == g_dpnidLocalPlayer)
            //條件成立，你就是新的 Host;
      else
            //否則
            //新 Host 是 pHostMigrateMsg->dpnidNewHost
      break;
      }
   }
}
```

以上的介紹加以整合使用就可以建立一個基本的連線對話環境。整個流程如下：

```
//使用 COM 物件
CoInitializeEx(NULL, COINIT_MULTITHREADED);
InitDirectPlay();      // 初始化 DirectPlay object
CreateDeviceAddress(); // 建立自己的 object adderss
HostSession();         // 選擇成爲 Host
EnumDirectPlayHosts(); // 選擇成爲 Connect
```

```
ConnectToSession();
SendDirectPlayMessage();  // 開始傳送訊息交談

// 離開 session 後要釋放資源
CleanupDirectPlay();    //釋放使用的 DirectPlay 物件資源
CoUninitialize();       // 釋放 COM
```

實際連線模式執行過程

step 1.輸入玩家名稱，列舉出可使用的連線服務

step 2.選擇建立來當 Host，搜尋來找已存在的 session

step 3.選擇建立後，成為 Host，輸入要建立的連線名稱

step 4.完成對話框

step 5.在 step2 過程中選擇搜尋的情況，請輸入 Host 的 address

step 6.找到名為 TestGame 的 session 目前有 1 位玩家在，執行加入

圖 4.8 交談情況

4.3 遊戲音樂

大部份撥放遊戲背景音樂有三種方式

- WAVE 格式
- CD 音軌
- MIDI

我們在遊戲中使用 MIDI，MIDI 是 Musical Instrument Digital Interface 的簡稱。MIDI 不必像 Wave 檔那樣做聲音抽樣，所以檔案不大但可以長時間演奏。MIDI 是一種只紀錄演奏演奏相關資訊的檔案(像樂譜)，音樂本身並沒有存放在檔案中，所以再撥放時需要【相當於樂器的裝置】。

在 CPU 不像現在一樣強大時，需要 MIDI 音源器才能撥放 MIDI，現在的電腦都支援軟體 MIDI，直接使用 CPU 模擬出 MIDI 的聲音。

MIDI 缺點在使用軟體音源時，CPU 負荷高，且無法撥放音源能力以上的聲音。超過音源的情況，一是無法放入【人聲】，二是同時最大發音數不夠的情況下，可能有些音色聲音會不見，也就是，製作好的音樂無法在使用者電腦上 100%呈現。

遊戲中撥放背景音樂方式就是循環撥放，直到遊戲結束。

我們是使用 Windows API 的 MCI(Multimedia Control Interface)來演奏 MIDI Data。MCI 是 Windows 用來操作多媒體的工具，還可以操控掃描器、CD、撥放動畫裝置、WAVE、VCR……。

MCI 的控制方法有兩種，Commend String 或 Command Message。

- Commend String 是把 MCI 指令寫成字串再丟出去。
- Command Message 以訊息和參數來組成必要的指令，控制方式較靈活。

我們使用 mciSendCommand 函數把指令送到 MCI，函數根據欲傳送的指令選用不同的旗標種類和參數類型。

```
MCIERROR mciSemdCommand(
MCIDEVICIED devicdID,   //指定開啓裝置時所得到的裝置 ID
UNIT uMessage,          //指定傳送的訊息
DWORD dwCommand,        //指定跟隨指令的旗標
DWORD dwParan           //指定指令的參數);
```

把 Command Message 傳送給 MCI，成功傳回 0，失敗傳回錯誤碼。

利用 MCI 演奏 MIDI 有一定的流程：

開啟 MCI 裝置 **à** MIDI 的撥放 / 停止 / 暫停 / 繼續 / **à** 關閉 MCI

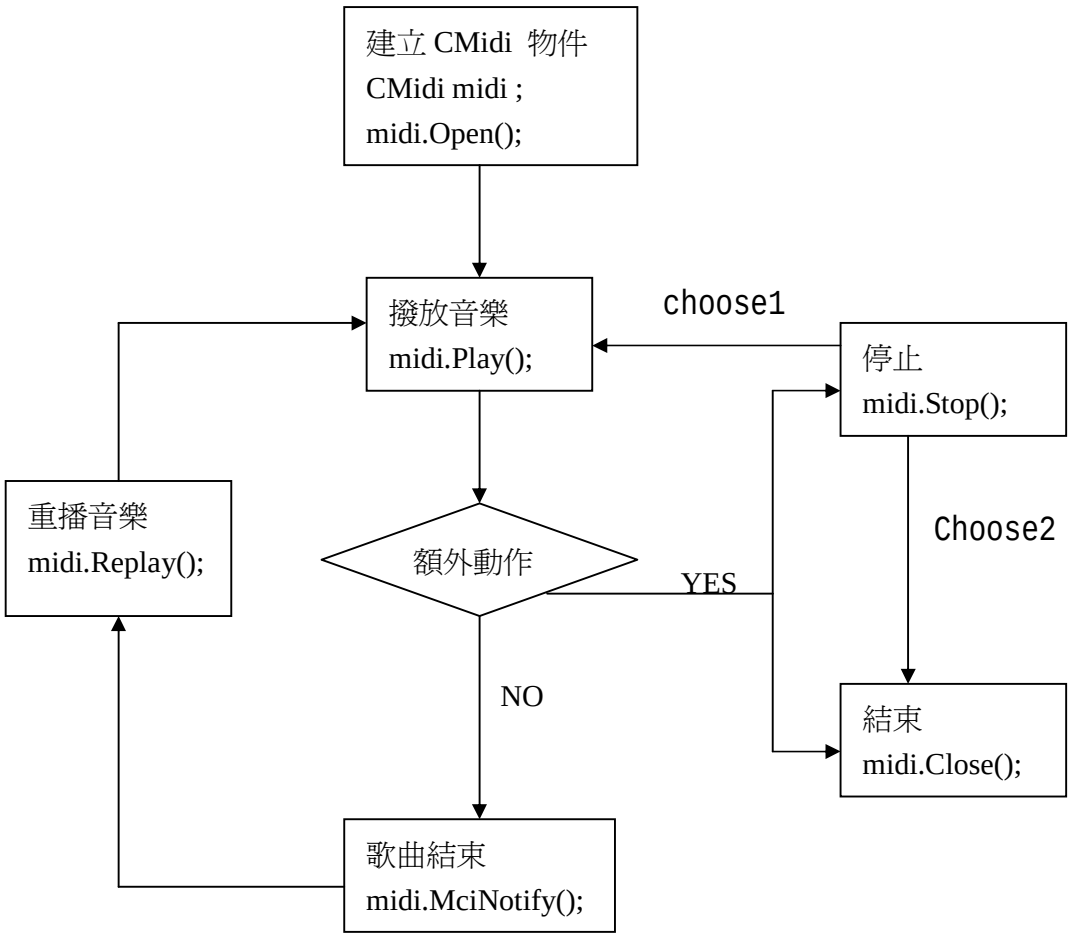


圖 4.9 Midi 動作流程圖

我們把使用到的相關函式包裝成一個類別好方便使用。類別定義如下：

```
class CMidi {
public:
CMidi(): Wnd(0), Id(0) {}
~CMidi() { Close(); }

BOOL  Open(CWnd *wnd);          //開啓裝置
BOOL  Close();                  //關閉裝置
BOOL  Play(const char *name);   //指定撥放的檔案
BOOL  Replay();                 //重覆撥放
BOOL  Stop();                   //停止
BOOL  MciNotify(DWORD id);      //音樂撥放完畢時，接受
                                //呼叫的函式
protected:
CWnd   *Wnd;
DWORD Id;                       //MCI 裝置 Id
} ;
```

Member Function	Member Variables
Open()	CWnd *Wnd
Close()	DWORD Id
Play()	
Replay()	
Stop()	
MciNotify()	

圖 4.10 Cmidi 類別圖

了解界面後，接下來看看實作部分。

```
// 開啓 Midi
```

```
BOOL  CMidi::Open(CWnd *wnd)
{  Wnd = wnd;
   return TRUE;
}
// 關閉
BOOL  CMidi::Close()
{  return TRUE;
}
// 撥放
BOOL  CMidi::Play(const char *name)
{  DWORD err;
   MCI_OPEN_PARMSopen;  // MCI_OPEN_PARMS 結構體物件
   // MIDI 裝置型態是 sequencer
   open.lpstrDeviceType = "sequencer";  open.lpstr
ElementName = "itdontrg.mid";   // 初始檔案名稱
if ((err = mciSendCommand(0, MCI_OPEN, MCI_OPEN_TYP
E|MCI_OPEN_ELEMENT|MCI_WAIT, (DWORD)&open)) != 0)
   {  char  errstr[256];
      mciGetErrorString(err, errstr, sizeof(errst
r));
      TRACE1("%s\n", errstr);
      Wnd->MessageBox(errstr);
      return FALSE;
   }     // 當無法取得裝置時顯示錯誤訊息對話框
   Id = open.wDeviceID;  // 取得裝置 Id
MCI_PLAY_PARMSplay;

// 結束撥放時，MM_MCINOTIFY 訊息會被送到
//MCI_PLAY_PARMS 結構體的 dwCallback 成員所指定的視窗
   play.dwCallback = (DWORD)Wnd->GetSafeHwnd();
if (mciSendCommand(Id, MCI_PLAY, MCI_NOTIFY, (DWORD)
&play))
{     mciSendCommand(Id, MCI_CLOSE, MCI_WAIT, 0);
```

```
      Id = 0;
      return FALSE;
   }
return TRUE;
}
// 從頭播放讀進的 Midi 檔案
BOOL  CMidi::Replay()
{  MCI_PLAY_PARMSplay;

//爲達到重覆撥放，程式必須能夠接受 MM_MCINOTIFY 訊息
//函式的呼叫方式就是就是傳遞 MCI_PLAY 給裝置叫他開始撥放
// MCI_SEEK_TO_START 演奏開始位置移動到 Data 最前面
// MCI_WAIT 結束撥放後傳回控制
// 下一次撥放結束時 MCI_NOTIFY 會通知程式

   play.dwCallback = (DWORD)Wnd->GetSafeHwnd();
if (mciSendCommand(Id, MCI_SEEK,MCI_SEEK_TO_START|
MCI_WAIT, 0)
    || mciSendCommand(Id, MCI_PLAY, MCI_NOTIFY, (DW
ORD)&play)) {
mciSendCommand(Id, MCI_CLOSE, MCI_WAIT, 0);
   Id = 0;
}
   return TRUE;
}

// 播放完畢
BOOL  CMidi::Stop()
{  UINT  id = Id;
   if (Id == 0)   return FALSE;
   Id = 0;
   DWORD err;
  if ((err = mciSendCommand(id, MCI_STOP, MCI_WAIT,
```

```
 0)) != 0
    || (err = mciSendCommand(id, MCI_CLOSE, MCI_WAI
T, 0)) != 0) { return FALSE; }
return TRUE;
}
// 播放完畢時，接受視窗呼叫的函式
BOOL  CMidi::MciNotify(DWORD id)
{  if (Id != id)
      return FALSE;
   return TRUE;
}
```

至於音樂的來源，我們是抓別人的 midi sample，然後取樣，再以音樂編輯軟體 remix 後輸出新的 midi。

軟體名稱：cakewalk SONAR

介面如下：

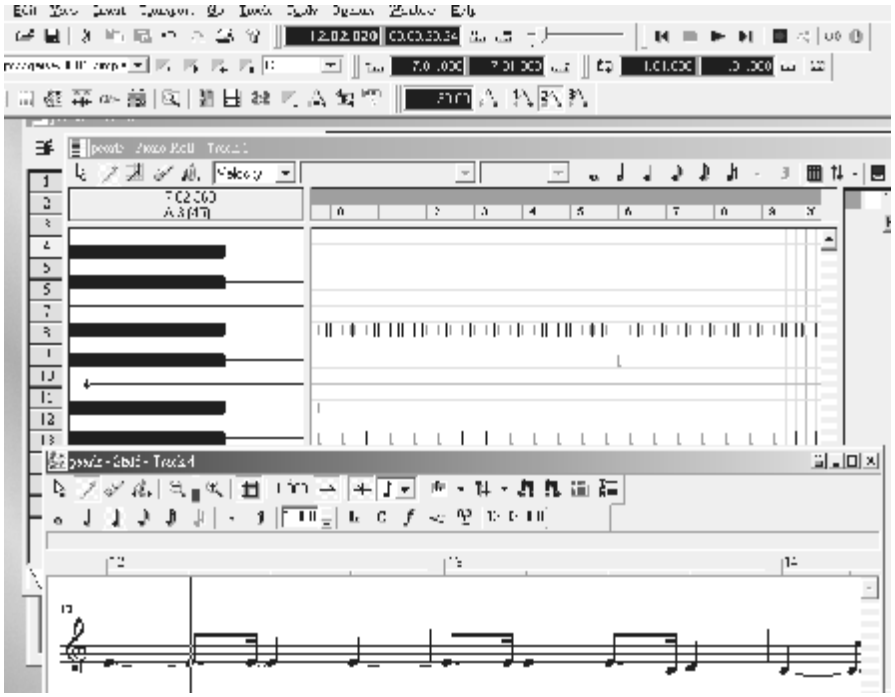


圖 4.11 SONAR 編輯音樂

第五章 心得與討論

5.1 所遇困難及問題

由於近幾年 3D 技術發展得非常快速，我們在作專題時又是盡量使用最新的技術，因此幾乎所有的技術資料都是由網路上取得，所以遇到困難大部分也都得自己想。而 3D 技術大部分都是抽象的概念，所以剛開始學蠻辛苦的。

開發工具是 VC++及 DX8.1SDK，之前對視窗程式設計沒碰過，所以要花一段時間去了解其架構，慢慢練習使用，而 MFC 龐大的工具庫，使用的複雜度及熟練度也是一條長的學習曲線，我只挑我會用到的部分，邊學邊使用。

而 DirectX 也是第一次碰，我是做訊息傳遞的，DirectPlay 有他自己的一套通訊連線方式，所以要看技術手冊來學習使用。英文不太好，有時看不懂想表達的意思，只好從程式碼下手，試著從中了解它要做什麼。

5.2 改進與未來展望

3D 環境部分

在即時 3D 繪圖方面，我們都只運用了最基本的效果，許多進階的效果沒有時間去研究是很可惜的，這些進階 D 效果都能讓虛擬 3D 環境更像真實世界，然而專題製作時間有限，人力也不足的情況下，只能做到這樣的地步。現在新技術不斷的被開發出來，因此若能持續加入新技術的效果，這樣的產品才有競爭力。

另外，由於缺乏建構 3D 模型的相關知識，因此我們使用到的 3D 模型都是最簡單的，這也是很可惜的地方。現今的 3D 遊戲必備的條件

之一就是華麗的 3D 模型，然而製作複雜的 3D 模型不是一朝一夕可以學成的，遊戲業界一定是團隊分工，有專門的人才負責的。

寵物養成部分

在我們的專題當中，寵物的行為模式並不多，因為時間有限，我們無法想出更多有趣的點子，這在商場上是很失敗的產品。因此如果能有好的企畫，我們的專題將更加具有競爭力。

網路方面

這一次交談的環境，還是傳統式的文字對談，而 DX8.1 支援可以聲音來對話，或許未來可以朝這方向發展，試試看效率通話的品質穩定度。

5.3 結論與感想

背景知識及技術的不純熟，所以準備了其他的資訊最後都沒用上，如之前打算使用資料庫，所以就花時間去準備，後來我發現技術不行，所以就先擱著，做別的。

整合的工作，每個人先分開做自己的 subsystem，最後在整合在一起，這方面也是需要花時間的。

參考資料

[1] M.U. Bradley Bargen & Peter Donnelly 著
『Inside DirectX』, p283-291
編譯 唐正一 , 校閱 廖荷婷 『深入剖析 DirectX』松崗 1999

[2] Hiroyuki TANAKA 著, 『Visual C++ 50 個精選專題實作』
p140-182, 博碩文化, Nov.2000

[3] Peter J. Kovach 著
『完全剖析 Direct3D』, p130-135
編譯 廖荷婷 孫浩淳, 校閱 官欣怡, Microsoft 2001

[4] 坂本千尋 著
『專業級遊戲程式設計』, p1-203，
編譯 李予青, 柏碩文化 2001

[5] 李建漢著
『DirectX 實技』, p1-301, 松崗 2000