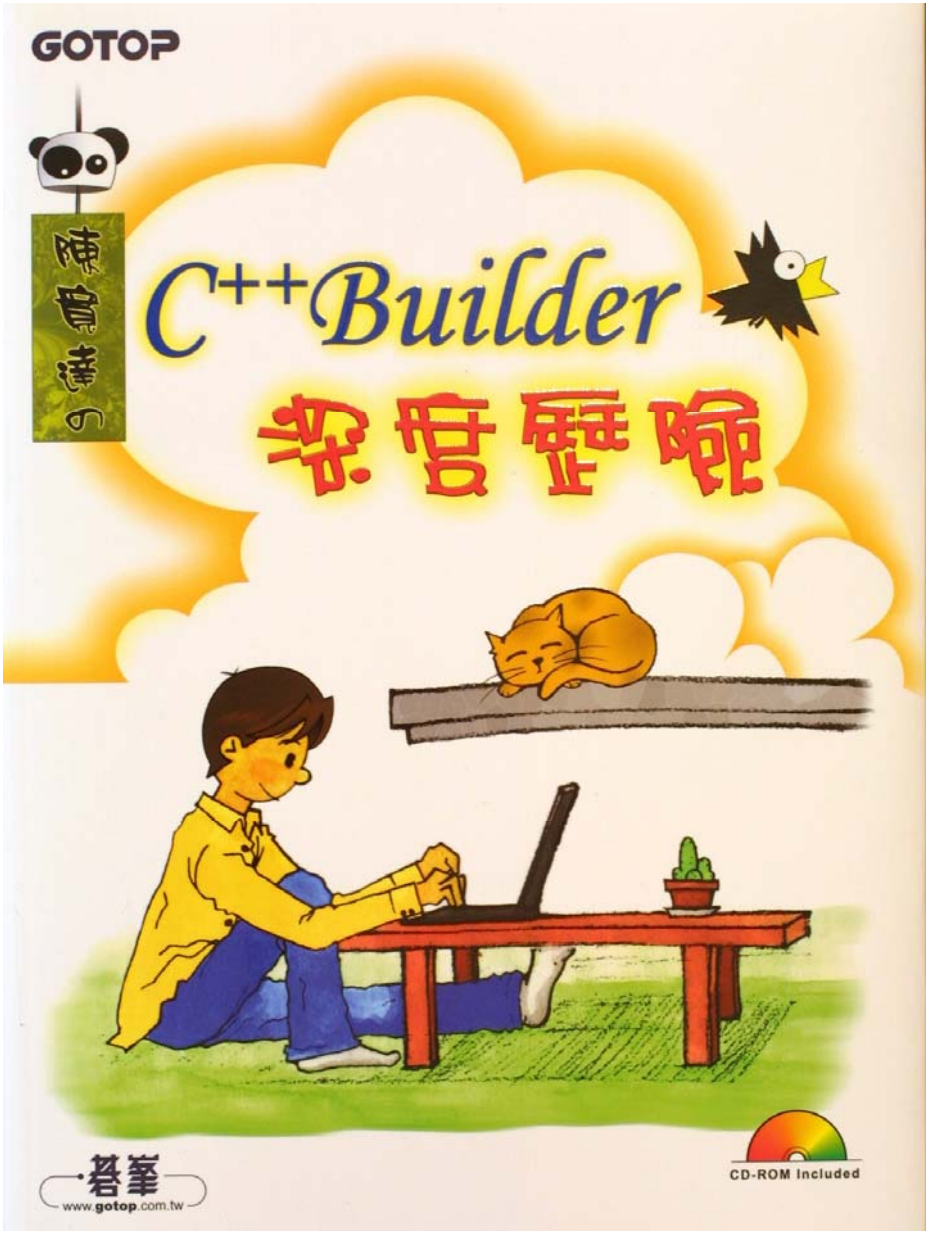

陳昇瑋

（原名：陳寬達）

creative commons

COMMONS DEED

姓名標示-非商業性-相同方式分享 2.5

您可自由：

- 重製、散布、展示及演出本著作
- 創作衍生著作

惟需遵照下列條件：

姓名標示. 您必須按照作者或授權人所指定的方式，保留其姓名標示。

非商業性. 您不得爲商業目的而使用本著作。

相同方式分享. 若您改變、轉變或改作本著作，僅在遵守與本著作相同的授權條款下，您始得散布由本著作而生的衍生著作。

- 爲再使用或散布本著作，您必須向他人清楚說明本著作所適用的授權條款。
- 如果您取得著作權人之許可，這些條件中任一項都能被免除。

您合理使用的權利及其他的權利，不因上述內容而受影響。

這是一份讓一般人易於了解的法律條款（完整的授權條款）摘要。

免責聲明

第九章

坦克大決戰

懷念古早時代的坦克大決戰遊戲嗎？
這紅白機時代經典遊戲的魅力，似乎至今未減。
既是人人愛玩的遊戲，又沒有太多的聲光特效，
玩得累了，手癢了，自己寫一套吧。

研一的暑假，可能也是學生生涯的最後一個暑假，帶著有假可玩直須玩的心理，期末考才剛結束就夥同幾位好友到南台灣渡假遊玩。

停留墾丁的那幾天，當然不例外地來到凱撒大飯店地下一樓的星際碼頭玩玩虛擬實境遊戲及「360 度腳踏車」。沒想到，就在星際碼頭入口處的電玩遊樂場，赫然發現某部電玩主機提供的是 BATTLE CITY 遊戲，也就是「坦克大決戰」。如同幾十年沒見的老朋友，一看到它，我就愣愣地定在主機旁，兩眼直盯遊戲畫面，直到朋友們合力把我架走爲止…

對於即將實作的第二個遊戲，原本心中是以「炸彈超人」爲底的，因爲它曾在我心中佔有極重要的地位，這留待後話。不料徵詢女友大人的意見時，她卻提出「坦克大決戰」的點子，沒想到她也在弟弟任天堂主機的 42 合 1 卡帶中玩過這遊戲，且還記憶頗深呢。再加上前陣子的一面之緣，充分感受到這紅白機時代經典遊戲的魅力，題目就這樣定下來了。

可是，這麼久以前的遊戲，細節規則忘得差不多，遊戲圖片也沒著落，得再想法找來玩玩，順便抓取圖形才行。於是我立刻到網路上下載任天堂模擬器，心想著只要再找到遊戲的 ROM 檔案即可。萬萬沒想到，我老早忘了它的英文名字叫「BATTLE CITY」，用「Tank」、「Tank War」、「坦克」、「坦克大決戰」等等字串爲關鍵字找了老半天找不著，氣死我了。最後，花了一天一夜的時間，好不容易在一個 100 合 1 的 ROM 中找到它，好辛苦哪。

怕有讀者未曾玩過此遊戲或是跟我一樣，年紀一大就把古早的遊戲忘光光了，在這先大略介紹一下好了。

任天堂版坦克大決戰

下圖是任天堂版坦克大決戰執行畫面，很令人懷念吧。後來我才知道，原來住我樓上房

間的同學就是個現成的坦克大決戰高手，可以一隻玩到最後一關（三十五關），接著繼續進攻第二輪。一看到我抓下來的執行畫面， 就立即嚷著「我知道，這是第三關，這一關要怎麼怎麼打就可以輕易過關...」，真是敗給他了。

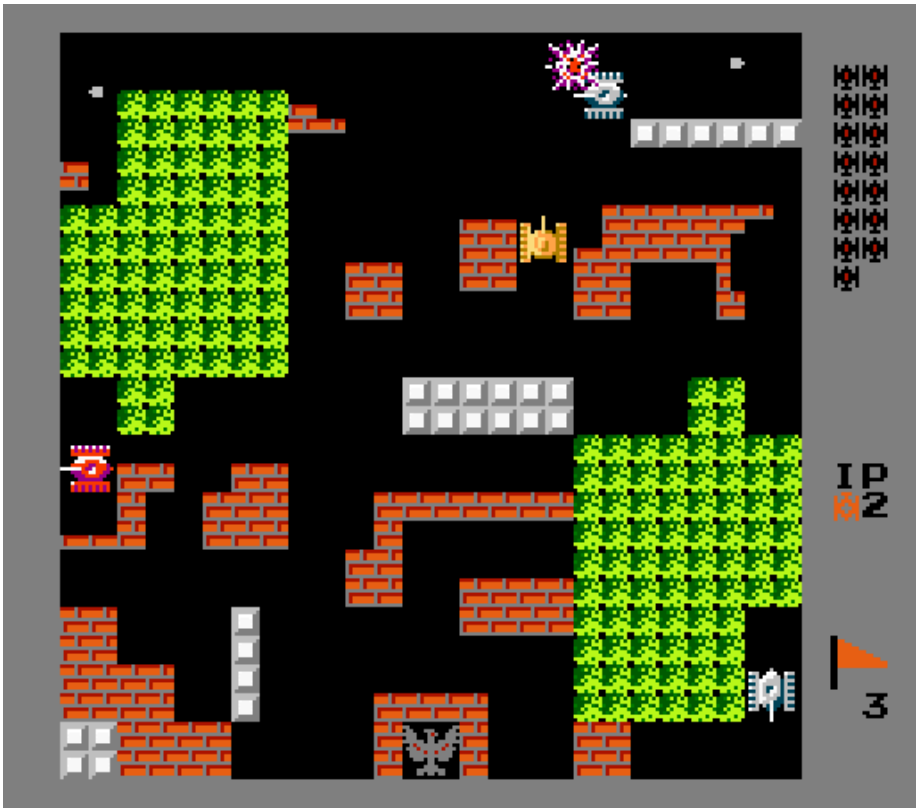

圖 9-1 / 坦克大決戰執行畫面，很懷念嗎？

圖中，黃色坦克爲主角，其餘皆是敵方坦克，會依序從畫面的左上角，正上方，右上角三個出生點出現，畫面上最多同時有四輛敵方坦克。每一關卡的目標只有一個，**「全力保護我方軍旗，並殲滅所有敵方坦克」**。每一關卡有一定的敵方坦克數量，似乎皆爲二十隻，全部打完就過關。不論我方坦克剩餘隻數，只要軍旗被打到，就立即 GAME OVER（我同學又說話了，因爲旗幟爲老鷹形狀，所以若軍旗被打壞可稱爲「烤小鳥」），好殘酷。

坦克大決戰的圖形很簡單，地形總共才五種：

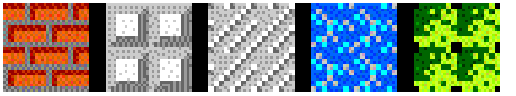

圖 9-2 / 坦克大決戰的地形

- 磚牆
  最常見的地形，不能通過，可被子彈擊毀。依擊中位置不同，一發子彈可打壞八分之一或四分之一。
- 鐵牆
  坦克無法通過，也不能被子彈擊毀，除非吃到三顆星星以上。
- 鋼板
  可在上面走動，但會打滑，每走一步會滑動好幾步。
- 海洋
  坦克無法通過，但子彈可通過。
- 樹林
  可以通過，但會遮住坦克，成爲絕佳的隱蔽場所。

坦克大決戰的好玩之處就是能夠善用這五種地形，設計成有趣且富挑戰性的關卡，地形的排列方式通常決定此關卡的難易度及平衡度，可說是一門大學問呢。

敵方坦克會不斷在畫面上走來走去，同時胡亂射擊，請小心，務必在槍林彈雨下殲滅它們。敵方坦克有四種：

圖 9-3 / 四種敵方坦克

- 第一種
  最常見，一切屬性普通。
- 第二種
  六個輪子，跑得很快，一不小心就會被它溜到軍旗旁偷打。
- 第三種
  第一種敵方坦克的改良版，只有炮管及尾端有些許不同，子彈飛行速度極快。
- 第四種
  體積最大，裝甲最厚，要擊中四次才會翹辮子。

摧毀四種坦克的得分依序是一百、兩百、三百及四百分，但對於以手榴彈（可摧毀所有敵方坦克的寶物）炸掉的坦克，則不計分。

每一關最多會出現四次寶物，每當轟掉閃著紅光的坦克後，寶物就會立即出現，寶物位置及種類依亂數決定。寶物有下列六種：

圖 9-4 / 六種寶物

- 弓箭

  將我方軍旗暫時以鐵牆圍住，沒有被攻破之虞。

- 時鐘

  看樣子大概也猜得到，可暫停敵方坦克的行動一段時間。

- 手榴彈

  將畫面上所有敵方坦克摧毀。

- 鋼盔

  我方坦克暫時無敵。

- 坦克

  生命數加一，但無論生命數多少，只要軍旗被攻破，遊戲即結束。

- 星星

  這是最有用的寶物，若沒有此寶物，要順利玩上十關我想也很難。吃到第一顆星星後，子彈速度會加速，如此才能與敵方的高級坦克匹敵；吃到第二顆星星後，可以連發兩發子彈；吃到第三顆星星後，可以無堅不摧，轟破鐵牆。這時我同學又說了「傳說中，吃到三十顆星星後，坦克就可以自由地行走海洋」。不過傳說歸傳說，沒試過誰也不曉得，依我的功力，往往還沒吃到三顆星星就掛點了，你可以幫我驗證這個「傳說」嗎？：)

寶物出現後，在畫面上閃呀閃的，若不去吃，待下一個閃著紅光的坦克出現後，寶物就會消失，所以吃寶物的動作要快，迅速確實才行。

設計自己的坦克大決戰

詳細介紹任天堂版本的坦克大決戰後，希望能讓你徹底瞭解這個經典級遊戲。接下來，讓我們也來寫一套坦克大決戰，希望能實作出其大部分的功能，甚至加入新的改良及功能。

對於前一章的「足球番」遊戲還滿意嗎？不知道你覺得太簡單，還是太難了？撰寫此章的同時，我已將程式全部撰寫完成了，完成後才發現，糟糕，不曉得是上一章的「足球番」太簡單還是這一章的坦克大決戰太難，「足球番」三支程式的程式碼林林總總加起來約兩千行，而這回坦克大決戰卻是四千行，正好是兩倍，哇 ~~ 好多啊，讀者你要有心理準備哦。

選擇坦克大決戰，也因爲它的幾個特性恰巧符合本章的教學目的：

1. 畫面處理 GDI「勉強夠用」，不需用到 DirectDraw，背景不用捲動。
2. 除遊戲主程式外，必須另有地圖／關卡編輯器的搭配才算完整，另外還配有圖庫編輯器，這兩支工具可推廣使用於許多益智、角色扮演甚至動作遊戲上頭。
3. 圖形使用不多，但是遊戲本身耐玩，不是須靠畫面才能吸引人的遊戲類型。
4. 可擴充的地方極多，例如可爲它加上網路連線功能，讓我可跟住台北的大哥合作破敵等等。

使用 GDI 來撰寫倉庫番可說是遊刃有餘，但今天對於坦克大決戰這樣的即時動作遊戲，在圖形不多且地圖不大的情況下，我只能說，勉強夠用。在 Pentium II 450、Windows NT 4.0 下十分順暢，在 IBM ThinkPad 570、Pentium II 300、Windows 98 下執行速度也還不錯，但這大概就是極限了。要不是有畫面上最多四輛敵方坦克的限制，光是重繪所有坦克及漫天飛舞的子彈我想就可能讓遊戲一頓一頓的，更別提即時處理使用者的輸入了。

先來訂立我們這套坦克大決戰的功能及特色：

1. 遊戲規則與坦克大決戰大致相同，主角必須保護軍旗，並將所有敵方坦克摧毀即可過

關。

2. 可視範圍即是地圖大小，因此不必支援地圖捲動。
3. 支援由多張地形圖片拼湊爲一個圖片群組，編輯地圖時可直接對整個或部分圖片群組進行操縱。
4. 角色大小沒有限制，亦即，哪天心血來潮，弄了一部半個畫面大的巨型坦克想加入遊戲成爲敵方坦克也不是問題。
5. 角色的移動以「點」爲單位，可以平滑移動，因此碰撞處理十分麻煩。這也是此程式的程式碼爲「足球番」程式碼兩倍行數的最大因素。
6. 所有圖片，包括角色圖形皆採外掛方式，可在不修改程式碼的情形下更動圖形。
7. 採用關卡制度，可讓使用者自行編輯關卡及遊戲。
8. 多層貼圖，因此可造出高度及層次感，也可設計出較真實的地形。

與正宗的「坦克大決戰」相較，最大的改進是**支援圖片群組**，**任意尺寸的角色**以及**多層貼圖**。其實若只想單純地實作「任天堂版本坦克大決戰」，這三項功能根本不必要，只要依賴「足球番」那種超陽春型圖庫編輯器及地圖編輯器，即可達成坦克大決戰所需的效果。

但是程式寫著寫著，我還是決定多留點空間給這支程式，使得它的發揮空間極大。而這些空間就留給讀者盡情發揮囉。

系統規劃

全套遊戲除了主程式外，另設計兩支工具程式－圖庫編輯器及地圖編輯器，理由是減少程式設計的複雜度及不必要的 overhead。

程式中，我大量使用類別及物件，亦即，遊戲畫面中所有看得到的地圖、圖格、坦克、子彈、寶物等等，通通都是物件。在物件導向程式設計中，好好地規劃類別以及類別間

的從屬、階層關係是最重要的，我先以一張圖說明此程式所設計的各個類別：

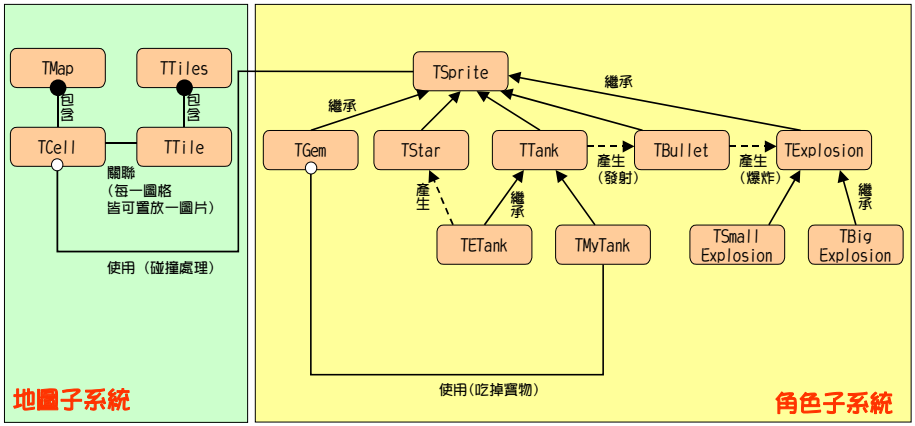

圖 9-5 / 重要類別及其關係大局觀

從圖 9-5 中，你可以得知整套程式的類別架構及規劃方式。所有類別可分爲兩個子系統：

- 地圖子系統

  包括圖庫及地圖兩部分。圖庫部分處理程式所用到的所有圖形，圖片由圖庫編輯器匯入，儲存在圖庫檔案中，並在遊戲執行時將圖片提供給地圖部分繪製。地圖部分則處理各個關卡地圖，配合地圖編輯器設計地圖，並能與圖庫合作無間來繪製遊戲畫面。

- 角色子系統

  遊戲畫面中，除了地形部分，其它不論會動不會動的物體皆由角色子系統來控制處理，如坦克、子彈、爆炸、寶物等等，最重要的是物體碰撞處理，物體與地形的碰撞處理程式碼將兩個子系統連結起來。

  若角色沒有經由碰撞處理函式根據所在的地形做出不同的回應，則地圖是地圖，角色是角色，你會看到坦克，子彈在畫面上亂跑，不論所在地形是山是海，兩個子系統將分別獨立行事，很奇怪的景象。

接著，我們再來詳細分派每個類別的工作，將任務切割清楚再予分派，是成功合作的大前提。隨著程式越趨複雜，程式碼越趨龐大時，你可以看到我反而會花更多篇幅講解系統規劃，而不是一味地講解程式碼。因爲只要擁有大局觀及清楚的程式架構，寫出好程

式只是遲早的事，端視實作能力而定；但若老在技術及程式碼上斤斤計較，分析及規劃能力窒礙不進，整體的程式設計能力是很難升級的。

分爲兩個子系統來介紹遊戲的類別規劃。

地圖子系統

圖庫處理

TTile 類別

- 管理一張圖片（待會常數定義就會看到，圖片大小定爲 32 x 32）。
- 記錄圖片屬性，如坦克能否通過、子彈能否通過、子彈能否破壞等等。
- 支援圖片群組。每個圖片群組爲最大 5 x 5 張圖片的集合，例如你可以設計大小爲 128 x 96 的城堡圖案，使用 4 x 3 大小的圖片群組，使用 12 個 *TTile* 物件，也就是 12 張圖片 。

TTiles 類別

- 擔任圖庫總管，管理（包括產生及摧毀、新增、刪除）旗下所有 *TTile* 物件。
- 負責儲存／載入整個圖庫。
- 提供圖片給地圖類別使用，才能順利地繪製遊戲畫面。

地圖處理

TCell 類別

- 管理地圖上一個圖格（與圖片大小一致，爲 32 x 32）。
- 記錄此圖格所使用的圖片編號，即 *TTile* 物件在圖庫中的編號。

- 管理圖格的破碎情形，將圖格區分爲 4 x 4 個小圖格，其中每個小圖格皆呈存在或不存在的狀態，繪製時只繪出存在狀態的小圖格，如此便可達到圖格破碎的效果（用於被子彈擊毀的磚牆）。
- 計算此圖格所佔用的矩型區域（考慮破碎狀態）。

TMap 類別

- 管理一道關卡的所有地圖，每個關卡擁有四張地圖（分別爲地形、地形物、物品及高地形物四層）。
- 負責此四層地圖的載入／儲存，還有使用最頻繁的繪製圖層工作。
- 繪製圖層時，會將此圖層所有圖格以對應的圖片繪製出來，其中地形層採直接繪製，而其它圖層使用透明貼圖。

角色子系統

TSprite 類別

角色子系統完全由 *TSprite* 類別來擔綱，所有角色子系統中的其它類別皆是 *TSprite* 的子類別，由此可見它的重要性。*TSprite* 類別的任務爲：

- 管理角色圖形。角色面對每個方向時各自有不同的代表圖形，而在同一個方向時也有多張動畫可以輪替使用，以達成走動、游動等效果。
- 記錄角色狀態，如座標、方向、移動速度、是否可見、是否在空中、佔用的矩型區域等等。
- 記錄角色屬性，如圖形是否具方向性、是否與地形物碰撞、是否與坦克碰撞、走動時是否自動切齊地形物等等。
- 更新角色動作。每進行一步動作，就要更新動畫、移動座標、進行碰撞處理。
- 碰撞處理，分別檢查與邊界、地形及其它角色的碰撞情況，再根據屬性及狀態來決

定碰撞後的反應。

- 繪製角色。將正確的角色圖形繪製在畫面上的對應位置。

工作很多吧。它實在太重要了，此類別要是沒有好好設計，保證遊戲繼續往下寫，需要進行碰撞處理，如子彈打到坦克、坦克吃到寶物等部分時，會後悔地叫苦連天。爲什麼我這麼確信呢？因爲…這是…經驗談。:P

坦克類別

所有的坦克皆是 *TTank* 類別的後代類別，*TTank* 類別的任務如下：

- 記錄坦克的狀態，如生命力、目前子彈數目、子彈爆炸威力、是否無敵模式等等。
- 處理與坦克碰撞相關的碰撞處理。
- 發射子彈，將子彈依坦克方向發射出去。

TTank 類別有兩個後代，分別是我方坦克專用的 *TMyTank* 及敵方坦克使用的 *TETank*：

- 我方坦克 *TMyTank*
  - 進行與寶物物件的碰撞處理，若碰到了即吃掉寶物。
- 敵方坦克 *TETank*
  - 負責在出現前發出金光閃閃的特殊效果。
  - 除非被幹掉，否則從不停止走動。
  - 以亂數自由走動、轉向、發射子彈，若連續碰撞超過某個數目，就一定轉向。

遊戲中會出現五種敵方坦克，在此我們爲五種坦克分別設計新的類別，而使用*Prototype*樣式[1]：先產生*TETank*類別物件，再根據五種敵方坦克的特性，分別設定它們的移動速度、子彈速度、爆炸威力、移動方式等等屬性。

---

[1] 請參考附錄C「參考書目」所列的Design Patterns一書。

咦？任天堂版的坦克大決戰不是只有四種敵方坦克嗎？嘻，因爲我偷偷多加了一種可「在天上飛」的超強敵方「坦克」，我同學又在身邊唸了：「破壞遊戲平衡度的傢伙…！」，呵呵，反正程式是我寫的，不甩他。這五個類別只要負責設定好它們各自對應的屬性即可。

金光閃閃的 TStar

敵方坦克出生前，同一地點會出現一顆星星閃呀閃的，接著敵方坦克才出現。這顆星星是由 *TStar* 類別負責繪製，*TStar* 類別任務最簡單了，只有一項：

- 原地不動，將星光閃動的幾張動畫秀完就行了。

TStar 類別只是被動地由製造效果的 *TETank* 使用，因此主控權完全操在 *TETank* 手上，待會我們就可看到 *TETank* 如何地控制 *TStar* 來達成金光閃閃的特殊效果。

子彈類別 TBullet

TTank 類別不是會發射子彈嗎？此子彈爲 *TBullet* 類別，子彈雖小，但 *TBullet* 類別可不簡單，它必須負責：

- 記錄發射子彈本身的坦克。
- 依坦克的方向及設定好的速度移動。
- 最重要的是碰撞處理：
  - 有些地形，雖然一般角色會撞上，但子彈不會撞到（如海洋），須特別處理。
  - 撞上邊界時，引發**小爆炸**。
  - 撞上地形物時，引發**小爆炸**，並損毀地形物（若該地形物可被損毀）。
  - 撞上坦克時，判斷是不是我方打到敵方或是敵方打到我方（若敵方打到敵方，則當

做沒事，子彈將穿越坦克而過），然後引發**大爆炸**。接著減少坦克生命力，若坦克掛了，則將坦克摧毀。

- 撞上別的子彈時，判斷是不是我方打到敵方或是敵方打到我方，若是，則將兩發子彈**摧毀**。

- 管理及繪製爆炸效果物件（*TExplosion*）。

爆炸效果類別

Oh，沒想到小小一顆子彈，任務這麼繁重。子彈爆炸時，它會產生 *TExplosion* 物件來製造爆炸效果，*TBullet* 及 *TExplosion* 物件的關係就如同 *TETank* 與 *TStar* 物件的關係一般，*TExplosion* 只負責繪製效果，它的生滅以及所有活動皆被產生它的 *TBullet* 物件控制。不過，*TExplosion* 類別還多了一項任務：

- 原地不動，將爆炸效果的幾張動畫秀完就行了。

- 若爆炸導致遊戲結束，則在爆炸結束後通報遊戲主迴圈：此局必須結束了。

至於 *TExplosion* 的兩個子類別：*TSmallExplosion* 及 *TBigExplosion*，分別代表小爆炸及大爆炸，沒有動作上的不同，只有圖形的不同而已。

TGem 寶物類別

最後，剩下 *TGem* 寶物類別，它的任務爲：

- 隨意找個地方擺，接著原地不動，等著我方坦克來吃。

就這樣，哇啊，簡單吧。角色子系統的類別也介紹完畢。

兩個子系統中所有類別的任務皆分派後，請再回頭看看圖 9-5，對於整個遊戲的架構，你是否已瞭然於胸了呢？那麼，理論上，目標已在眼前，接下來的路途，唯「實作」二字

而已。

地圖子系統

不用我說，你一定也知道，地圖子系統一定比角色子系統好寫多了。因爲柿子總先挑軟的吃，所以我才選擇先撰寫地圖子系統，對不對？

呵，才不是呢，我們不是要分別撰寫圖庫編輯器、地圖編輯器及遊戲主程式三支程式嗎？地圖子系統於三個程式中都會用到，而角色子系統只在遊戲主程式用得上而已。再加上地圖子系統裡所有程式碼並不依賴任何角色，但角色必須依賴地圖來進行碰撞處理，因此無論如何我們也得先從地圖子系統著手。

首先定義地圖子系統所需的常數（定義於 util.h）：

```
#define TILE_NUM_X      13                  // 畫面橫軸格數
#define TILE_NUM_Y      13                  // 畫面縱軸格數

#define TILE_WIDTH      32                  // 圖片寬度點數
#define TILE_HEIGHT     32                  // 圖片高度點數

#define SM_TILE_NUM_X   4
#define SM_TILE_NUM_Y   4

#define SM_TILE_WIDTH   (TILE_WIDTH / SM_TILE_NUM_X)   // 小碎片寬度點數
#define SM_TILE_HEIGHT  (TILE_HEIGHT / SM_TILE_NUM_X)  // 小碎片高度點數

#define WORLD_WIDTH     (TILE_WIDTH * TILE_NUM_X)      // 畫面寬度
#define WORLD_HEIGHT    (TILE_HEIGHT * TILE_NUM_Y)     // 畫面高度

const char* SIG_MYFILE  = "Xshadow_Stock";   // 圖庫及地圖檔案的檔頭標籤

const char* FN_TILE_ARCHIVE  = "TILES.TIA";  // 圖庫檔案

const char* FN_MAP_PREFIX    = "MAP";        // 關卡圖檔檔名 (MAP??.DAT)
const char* FN_MAP_EXT       = ".DAT";       // 關卡圖檔副檔名

const int LAYER_TERR         = 0;            // 地形層
const int LAYER_TERRITEM     = 1;            // 地形物層
```

```
const int LAYER_ITEM         = 2;            // 物品層
const int LAYER_HITERRITEM   = 3;            // 高地形物層

#define LAYER_MAX        3                   // 最多到高地形物層
```

圖片高及寬度爲 32 x 32，是十分常見且有效率的大小設定，因爲我們的 CPU 通用暫存器寬度也是 32 bit，在進行記憶體區塊搬移時，不會有不符合 DWORD alignment 的情況發生。

畫面橫軸及縱軸格數，13 x 13，是根據任天堂版坦克大決戰而訂，這樣一來，連地圖都可以照抄，享受一下不動大腦的悠閒舒適。:p

SIG_MYFILE 爲檔頭標籤，在讀取圖庫及地圖檔案時，先確認檔案開頭有沒有這個字串，以確定讀取的是我們自己的檔案，不會有誤讀的情況發生。

與原版坦克大決戰相比，多個圖層是一大改良，我定義了四個圖層：

- 地形層
  鋪在最底端，如沙地、水泥地、海洋等等，採不透明貼圖。因此不論如何，我們的畫面上一定有「地板」，不會讓玩者看到黑黑的圖格，與任天堂版坦克大決戰不同，請參看圖 9-1。
- 地形物層
  擺設磚牆、鐵牆的圖層，採透明貼圖，因此若磚牆有半塊打破了，可以看到下面的地形層。爲求效率起見，這是唯一進行碰撞判斷的圖層。因此，假設有一塊屬性設定爲不能通過的海洋圖片，若將它擺在地形層，坦克依然可以通過，因爲地形層並不進行碰撞處理；但若擺在地形物層，因爲碰撞處理的緣故，坦克就不能通過，這樣的設定使得關卡設計的自由度大增。
- 物品層
  意義與地形物層一樣，也採透明貼圖，只差在它並不進行碰撞處理。
- 高地形物層
  意義與物品層一樣，唯一的差別是繪製的順序。此圖層繪製順序在角色之後，所以看起來會在角色上方。例如擺上一些花棚，坦克可從其下通過。

四個圖層以及角色的貼圖順序爲：地形層、地形物層、物品層、地上的角色、高地形物層、天上的角色。

四個圖層各有其功能及特色，妥善地安排圖層，既可使程式好寫不少，又可造出更好看的佈景。

不過圖層數目一多，缺點也跟著衍生，每多一層圖層，貼圖部分就多了一份工作，它必須一一檢查 *TILE_NUM_X* 乘上 *TILE_NUM_Y* 個圖格的圖片編號，若該圖格有圖片就把圖片貼上，因此若分爲太多層，每層又擺一大堆東西時，很容易就大幅拉下遊戲的執行速度，因此必須謹慎考量。

圖庫處理

TTile 圖片類別

圖庫處理部分只有兩個類別：*TTile* 及 *TTiles*。而 *TTile* 圖片物件由 *TTiles* 圖庫物件管理，因此第一個就從管理單一圖片的 *TTile* 類別下手（定義於 TileUnit.h）：

```
#0001  // 圖片屬性: 可以通過, 圖片可被打破, 子彈可以通過, 此圖片是軍旗
#0002  enum TTileAttrElement {taCanPass, taCanBreak, taBulletCanPass,
#0003    taFlag};
#0004  typedef Set<TTileAttrElement, taCanPass, taFlag> TTileAttr;
#0005
#0006  class TTile { // 單一圖片
#0007  private:
#0008    Graphics::TBitmap* FBits; // 存放圖片的 bitmap
#0009    TTileAttr FAttr; // 屬性
#0010    bool FDisposed; // 是否已棄置不用
#0011
#0012    bool FFirstTile; // 是不是圖片群組裡的第一張圖片
#0013    Byte FXNum, FYNum; // 圖片群組的橫向及縱向圖片數目
#0014  protected:
#0015  public:
#0016    TTile();
#0017    ~TTile();
```

```
#0018
#0019   void init();
#0020
#0021   // assignment constructor
#0022   TTile& operator=(const TTile& t);
#0023   // copy constructor
#0024   TTile(const TTile& t);
#0025
#0026   // 載入及儲存圖片
#0027   void LoadFromStream(TStream* Stream);
#0028   void SaveToStream(TStream* Stream);
#0029
#0030   // 提供給外界存取的屬性
#0031   __property Graphics::TBitmap* Bitmap = {read = FBits};
#0032   __property TTileAttr Attr = {read = FAttr, write = FAttr};
#0033   __property bool Disposed = {read = FDisposed, write = FDisposed};
#0034
#0035   __property bool FirstTile =
#0036     {read = FFirstTile, write = FFirstTile};
#0037   __property Byte XNum = {read = FXNum,  write = FXNum};
#0038   __property Byte YNum = {read = FYNum,  write = FYNum};
#0039 };
```

與前一章足球番程式管理方式不同的是，不再很愚蠢地將圖庫中所有圖片置於一個 BMP 圖檔中，現在每張圖片分別管理，每張都是獨立的 bitmap，存放於 *TBitmap* 物件 *FBits*。

無法任意刪除的圖片

FDisposed 布林變數指的是此圖片是否已廢棄不用。這樣做的理由是，爲求方便，我直接使用陣列索引來做爲圖片編號，這很好。不過當刪除一張圖片時，原來陣列索引大於它的圖片的陣列索引就會減一，那麼原本編輯好的地圖（每個圖格皆儲存對應的圖片編號）就會混亂，應該秀出 4 號的圖格結果秀出 5 號，應該秀出 100 號的圖格結果秀出 101 號…結果每當圖片刪除時，所有的地圖檔都必須隨之修改，這是無法接受的情形。因此，若 *FDisposed* 爲 *true*，表示此圖片事實上已刪除，只是我們不將它從陣列中拿走，免得影響其它圖片的編號。

那你可能會抗議，若刪除的圖片都不拿掉，豈不白白浪費記憶體及磁碟空間，配置不必

要的 *TBitmap* 來存放不必要的圖形？唔，只要在載入及儲存圖片的函式中動點手腳，就可讓已棄置的圖片不再浪費資源來存放 bitmap 及其它資料。至於 *TTile* 物件則沒有辦法不建構，合理的解決方案是在遊戲設計時期先不去管它，等到要將遊戲移交給別人使用前，再撰寫一支工具程式來移除已棄置的圖片，並同時修正所有的地圖檔。

避免浪費記憶體及磁碟空間的手腳是這樣做的：

```
#0001  void TTile::LoadFromStream(TStream* Stream)
#0002  {
#0003    TReader* reader = new TReader(Stream, 2048);
#0004    try {
#0005      FDisposed = reader->ReadBoolean();
#0006      reader->FlushBuffer();
#0007      // 是否已棄置不用 ?? 是的話就不再讀取其它屬性
#0008      if (FDisposed) return;
#0009
#0010      FBits->LoadFromStream(Stream); // 圖形
#0011      reader->FlushBuffer();
#0012
#0013      // 屬性
#0014      for (TTileAttrElement x = taCanPass; x <= taFlag;
#0015        x = (TTileAttrElement)(x + 1)) {
#0016        bool b = reader->ReadBoolean();
#0017        if (b) FAttr = FAttr << x;
#0018      }
#0019
#0020      FFirstTile = reader->ReadBoolean();
#0021      if (FFirstTile) { // 若是群組頭頭, 則讀取群組長寬
#0022        FXNum = reader->ReadInteger();
#0023        FYNum = reader->ReadInteger();
#0024      }
#0025    } __finally {
#0026      delete reader;
#0027    }
#0028  }
#0029
#0030  void TTile::SaveToStream(TStream* Stream)
#0031  {
#0032    TWriter* writer = new TWriter(Stream, 2048);
#0033    try {
#0034      writer->WriteBoolean(FDisposed);
#0035      writer->FlushBuffer();
#0036      // 是否已棄置不用 ?? 是的話就不再寫入其它屬性
#0037      if (FDisposed) return;
```

```
#0038
#0039     FBits->SaveToStream(Stream); // 圖形
#0040
#0041     writer->FlushBuffer();
#0042
#0043     // 屬性
#0044     for (TTileAttrElement x = taCanPass; x <= taFlag;
#0045       x = (TTileAttrElement)(x + 1))
#0046       writer->WriteBoolean(FAttr.Contains(x));
#0047
#0048     writer->WriteBoolean(FFirstTile);
#0049     if (FFirstTile) { // 若是群組頭頭, 則寫入群組長寬
#0050       writer->WriteInteger(FXNum);
#0051       writer->WriteInteger(FYNum);
#0052     }
#0053   } __finally {
#0054     delete writer;
#0055   }
#0056 }
```

0010 及 0039 列分別讀出及寫入圖片影像，只要一個呼叫就了，這正是物件永續（object persistence）機制最快樂的應用。

圖片群組支援

下圖分別從兩個地圖編輯器執行畫面取得，左圖的圖片一團混亂，硬生生地將河流、小橋、流水，拆成好多圖片，散落在圖片堆內，讓人邊找邊拼，設計地圖時同時玩拼圖遊戲。右圖是支援圖片群組的圖片選取視窗，可以一次將整個單位的圖片框選，扔到地圖上；也可以只選取任何一部分，完全隨心所欲。

圖 9-6 / 有無支援圖片群組的比較

光看這兩個地圖編輯器的畫面，即刻的反應是：「天啊，不支援圖片群組的地圖編輯器對於關卡設計者真是太殘忍了」。於是，稟著仁民愛物的精神， *TTile* 類別也不落人後，提供圖片群組的支援。

若 *TTile* 物件本身爲「群組頭頭」，也就是圖片群組的最左上角那一張，則只有它知道圖片群組的寬與高資訊，所以圖片群組的大小由它來維護、管理，「群組頭頭」以外的圖片並不曉得自己究竟屬於哪個群組，又此群組的大小爲何。圖片群組對遊戲主程式沒有任何影響，它只是讓地圖編輯器使用起來方便且人性化多了。如果你曾讓不支援群組的地圖編輯器虐待過，那麼你一定會喜歡此設計，雖然圖片群組的支援用不著多少程式碼。

另外還有一點的好處是，這兒設計的圖片群組的唯一功能只有輔助地圖設計，而不會帶來其它困擾。後頭實作地圖編輯器時你將會看到，可以直接將整個群組以一張大圖的方式來張貼，也可以將它拆開張貼，如同古早的地圖編輯器那樣。

TTiles 圖庫類別

下面是 *TTiles* 類別的定義，它擔任圖庫總管的角色，管理（包括產生及摧毀、新增、刪除）旗下所有 *TTile* 物件，並負責儲存／載入整個圖庫。

```
#0001  class TTiles { // 圖庫
#0002  // Singleton Pattern
#0003  private:
#0004    static TTiles* FInstance;
#0005  public:
#0006    static TTiles& Instance();
#0007  private:
#0008    typedef std::vector<TTile*> TTileArray;
#0009    TTileArray* FTiles; // 所擁有的圖片陣列
#0010
#0011    int GetTileNum(); // 圖片數目
#0012
#0013    TTile& GetTile(int No); // 利用索引取得圖片
#0014  protected:
#0015    TTiles();
#0016    ~TTiles();
#0017  public:
#0018    int AddTile(TTile* NewTile); // 加入新的圖片
#0019    void FreeTiles(); // 釋放所有圖片
#0020
#0021    // 載入及儲存圖庫
#0022    void LoadFromFile(AnsiString FileName);
#0023    void SaveToFile(AnsiString FileName);
#0024
#0025    __property int TileNum = {read = GetTileNum};
#0026    __property TTile Tile[int No] = {read = GetTile};
#0027  };
```

0009 列宣告存放所有 *TTile* 物件的 *FTiles* 動態陣列，在此我以 C++ Standard Library 提供的 *vector* 來存放不定數目的 *TTile* 物件。

TTiles 類別沒幹啥大事，反正需要圖片時找它要就對了。因爲整個遊戲中，只用到一個圖庫，多了也沒用，所以我實作了 Singleton 樣式，強制整個系統最多只能有一個 *TTiles* 物件，且可於任何地方存取。

地圖處理

圖庫及圖片準備好後，接著才能撰寫地圖部分，因爲地圖必須依賴圖庫才能操作及顯示。地圖處理的最小單位爲圖格，用一個 *TCell* 物件來表示。除了存放此圖格所使用的圖片編號，另外還負責管理圖格的破碎情形，*TCell* 類別宣告如下（定義於 MapUnit.h）：

```
#0001  class TCell {
#0002  private:
#0003    int FTileNo; // 圖格所放置的圖片編號
#0004    int FLayer, FX, FY; // 圖層, 座標
#0005
#0006    // 圖格破碎表格
#0007    // SM_TILE_NUM_X x SM_TILE_NUM_Y 個小圖格, "破" 或 "沒破"
#0008    typedef bool TBreakMap[SM_TILE_NUM_X][SM_TILE_NUM_Y];
#0009    typedef TBreakMap* PBreakMap;
#0010
#0011    PBreakMap FBreakPtr; // 圖格破碎表格
#0012    TRect FRect; // 圖格所佔區域, 會隨圖格破碎而變更
#0013
#0014    // 用於破碎圖格的貼圖動作
#0015    Graphics::TBitmap* FCellBitmap, *FSMTileBitmap;
#0016
#0017    TTileAttr GetTileAttr();
#0018    void SetTileAttr(const TTileAttr Value);
#0019
#0020    bool GetCanPass(); // 這個圖格能否通過 ?
#0021    void SetTileNo(int Value);
#0022
#0023    // 重新計算圖格所佔區域
#0024    void CalcRect();
#0025    // 建立圖格圖形 (可能是破碎的)
#0026    void BuildCellBitmap();
#0027
#0028    void BreakMapChanged();
#0029
#0030    bool IsBroken(int x, int y);
#0031
#0032    Graphics::TBitmap* GetTileBitmap(); // 取得圖片 bitmap
#0033
#0034    __property Graphics::TBitmap* TileBitmap = {read = GetTileBitmap};
#0035  protected:
#0036  public:
#0037    TCell();
#0038    ~TCell();
#0039
#0040    void init(int Layer, int x, int y);
#0041
#0042    // assignment constructor
#0043    TCell& operator=(const TCell& c);
#0044
#0045    // 載入及儲存
#0046    void LoadFromStream(TStream* Stream);
```

```
#0047    void SaveToStream(TStream* Stream);
#0048
#0049    // 圖格破碎處理函式
#0050    void AllocBreakMap(); // 建立圖格破碎表格
#0051    void DisposeBreakMap(); // 釋放圖格破碎表格
#0052    void BreakBy(TRect ARect); // 依矩形區域設定圖格破碎表格
#0053
#0054    void Draw(TCanvas* Canvas, bool bTransparent);
#0055
#0056    __property int TileNo = {read = FTileNo, write = SetTileNo};
#0057    __property TTileAttr TileAttr =
#0058      {read = GetTileAttr, write = SetTileAttr};
#0059
#0060    __property TRect Rect = {read = FRect};
#0061
#0062    __property bool CanPass = {read = GetCanPass};
#0063  };
```

0003 列爲最重要的，記錄圖格所使用的圖片編號。除此之外，0004 列還記錄此圖格位於哪個圖層的哪個位置上，位置資訊於計算圖格佔用的矩形區域時派上用場，而圖層編號也是繪製圖格時所需的重要資訊。

0008 列爲其圖格破碎表格的宣告，此表格是四乘四的布林陣列，分別記錄每個小圖格「破」或「沒破」，以達成磚牆被擊毀的碎裂效果。

破碎圖格處理

圖格最吃重的任務大概就屬破碎圖格的處理了。0011 列將 *FBreakPtr* 宣告爲指向圖格破碎表格的指標，而不直接宣告指向圖格破碎表格是有其用意的。因爲可能呈破碎情況的圖格算是少數，只有位於地形物層（此層才有碰撞處理）且屬性帶有 *taCanBreak* 的圖片才可能破碎，更何況有時磚牆還沒破幾個，就過關了（若是我的話，大概是磚牆還沒破幾個，就被敵人幹掉了），所以以指標來指向破碎表格。正常圖格的 *FBreakPtr* 指標爲 *NULL*，只有在圖格被打破時，才動態地建立破碎表格來使用，如此便可以節省不少記憶體空間的使用。

設定圖格破碎表格內容，也就是圖格破碎情形的函式有二，一是呼叫 *DisposeBreakMap* 函式，釋放目前的圖格破碎表格，代表此圖格完全沒有破碎情形；二是呼叫 *BreakBy* 函式，指定一個 *TRect* 矩形區域，將圖格與此矩形區域交集的所有小圖格設定爲破碎（不存在）：

```
#0001  // 依矩形區域設定圖格破碎表格
#0002  void TCell::BreakBy(TRect ARect)
#0003  {
#0004    // 若此時還沒有破碎表格, 就建一個
#0005    if (!FBreakPtr) AllocBreakMap();
#0006
#0007    TRect R, R1;
#0008
#0009    // 與 ARect 取交集, 有交集的小格則設爲破掉
#0010    for (int y = 0; y < SM_TILE_NUM_Y; y++)
#0011      for (int x = 0; x < SM_TILE_NUM_X; x++) {
#0012        // 計算小格的矩形區域
#0013        R1 = SM_TILE_RECT;
#0014        OffsetRect(&R1, SM_TILEWIDTH[x], SM_TILEHEIGHT[y]);
#0015        OffsetRect(&R1, TILEWIDTH[FX], TILEHEIGHT[FY]);
#0016
#0017        // 若有交集, 則讓它破掉
#0018        if (IntersectRect(&R, &ARect, &R1))
#0019          (*FBreakPtr)[x][y] = true;
#0020      }
#0021
#0022    BreakMapChanged();
#0023  }
```

因爲 *n* * *TILE_WIDTH* 及 *n* * *TILE_HEIGHT* 兩個乘法運算使用頻率極高，因此 Util 單元特別提供 *TILEWIDTH* 及 *TILEHEIGHT* 兩個一維陣列作爲查表用途。

設定破碎表格的函式是，針對四乘四個小圖格一一測試，呼叫 *IntersectRect* API 函式，傳入兩個矩形區域，它會傳回一布林值代表這兩矩形是否重疊，一旦有交集，就將此小圖格設定爲破碎，繪製圖格時就不會畫出來了。

BreakMapChanged 函式用來重新計算圖格所佔用的矩形區域。若 *FBreakPtr* 爲 *NULL*，表示此圖格沒有任何破碎，就按照正常程序計算圖格佔有的矩形區域；但若 *FBreakPtr* 不爲 *NULL*，表示此圖格已有破碎發生，此時必須再分別從四個方向去檢查破碎表格，最

上方、最下方、最左方及最右方的未破碎小圖格，才能得到圖格的真正矩形區域。此矩形區域供碰撞處理函式使用，但你也許已經發現，以矩形來測試碰撞，寫出來的遊戲一定會有不合理的狀況發生，例如，若某圖格除了最左邊一排及最下邊一排的小圖格外，通通都被打破了，但它佔用的矩形區域還是跟原本一樣大，所以從右上方來的坦克還是無法進入它的空缺處。這個缺點只有更詳細的碰撞處理方式才能解決，目前我們暫且先將此問題擱下。

儲存圖格時，也會將目前的破碎表格一併寫入資料流中，因此，只要地圖編輯器支援，設計關卡時就可設定圖格的破碎情況，可以藉此設計一道殘破不堪的廢墟關卡，或是拼湊出字形、圖案等等，都是不錯的應用。

圖格的繪製

圖格的繪製工作由 *Draw* 函式負責。繪製過程中，有兩個要素需要考量：是否需要採用透明貼圖，以及破碎圖格的外觀。

是否需要採用透明貼圖呢？這個問題十分好作答，只要是地形層，就採不透明貼圖；只要不是地形層，就採透明貼圖。

至於破碎圖格的外觀，的確是比較棘手的問題，因爲圖格一旦呈破碎狀態，我們就無法直接使用圖庫所提供的圖片來繪製－因爲圖格的外觀改變了嘛。那麼，可否在建立圖庫時，就爲每一種破碎情況準備一張圖片呢？這方法不可行。因爲每個圖格有 16 個小圖格，在每個小圖格都可能破／沒破的情形下，可能有 2^16 = 65536 種破碎情形，數目太大了，建立這種多個圖片只會造成資源浪費。於是，我們只剩下一種方法：在圖格破碎情形改變時，立即爲此圖格準備一張呈現其破碎情形的圖片。首先，在 *TCell* 類別裡加入 *TBitmap* 物件 *FCellBitmap*，包含呈現圖格外觀的影像。然後撰寫 *BuildCellBitmap* 函式：

```
#0001  // 繪製破碎的小圖格
#0002  void TCell::BuildCellBitmap()
#0003  {
#0004    if (!FBreakPtr) { // 未破碎, 直接取用圖庫圖片
```

```
#0005     FCellBitmap->Assign(GetTileBitmap());
#0006     return;
#0007   }
#0008
#0009   // 先把 FTileBitmap 設成全部透明
#0010   FCellBitmap->Canvas->Brush->Color = TRANSPARENT_COLOR;
#0011   FCellBitmap->Canvas->Brush->Style = bsSolid;
#0012   FCellBitmap->Canvas->FillRect(TILE_RECT);
#0013
#0014   // 若爲破碎圖格, 則一一將仍存在的小圖格貼上
#0015   for (int n = 0; n <= 3; n++)
#0016     for (int m = 0; m <= 3; m++) {
#0017
#0018       // 若此小圖格破掉了就不用畫
#0019       if (IsBroken(m, n)) continue;
#0020
#0021       // 先複製到另一個 bitmap
#0022       FSMTileBitmap->Canvas->CopyRect(
#0023         SM_TILE_RECT,
#0024         GetTileBitmap()->Canvas,
#0025         Classes::Rect(SM_TILEWIDTH[m], SM_TILEHEIGHT[n],
#0026           SM_TILEWIDTH[m + 1], SM_TILEHEIGHT[n + 1]));
#0027
#0028       // 再貼到畫布上, 以達成透明貼圖效果
#0029       FCellBitmap->Canvas->Draw(SM_TILEWIDTH[m], SM_TILEHEIGHT[n],
#0030         FSMTileBitmap);
#0031     }
#0032 }
```

若圖格沒有破碎，*FCellBitmap* 的影像只要直接由圖庫取得，呼叫 *TBitmap::Assign* 函式複製過來即可。若圖格是破碎的，沒有更聰明的方法，必須依序一個個檢查小圖格的破碎與否，將還存在的小圖格畫到 *FCellBitmap* 上頭。這也是爲什麼，每一次破碎表格更動了，就必須呼叫 *BreakMapChanged* 函式的原因：*FCellBitmap* 必須重新繪製。

```
#0001  void TCell::BreakMapChanged()
#0002  {
#0003    CalcRect(); // 重新計算所佔用區域
#0004    BuildCellBitmap(); // 建立圖格圖形 (可能是破碎的)
#0005  }
```

麻煩的 *FCellBitmap* 準備好之後，任何時候需要繪製圖格時，只消呼叫 *TCell::Draw* 函式，將 *FCellBitmap* 貼到畫布上即可。必須注意的是，在非地形層的其它圖層中，編號 0 號的圖片表示此圖格是空的，沒有放置圖片，所以不畫。

```
#0001  // 將指定的地圖層畫在 Canvas 上
#0002  void TCell::Draw(TCanvas* Canvas)
#0003  {
#0004    if (FLayer != LAYER_TERR) { // 非地形層
#0005      if (FTileNo == 0) return; // 沒有設定物品
#0006
#0007      Canvas->Draw(TILEWIDTH[FX], TILEHEIGHT[FY], FCellBitmap);
#0008    } else {
#0009       // 地形層不需要透明貼圖
#0010       BitBlt(Canvas->Handle, TILEWIDTH[FX], TILEHEIGHT[FY],
#0011         TILE_WIDTH, TILE_HEIGHT, FCellBitmap->Canvas->Handle,
#0012         0, 0, SRCCOPY);
#0013    }
#0014  }
```

地圖總管 TMap

接下來是地圖總管－*TMap* 類別，類別宣告如下（定義於 MapUnit.h）：

```
#0001  class TMap {
#0002  private:
#0003    static TMap* FInstance;
#0004  public:
#0005    static TMap& Instance();
#0006  private:
#0007    typedef TCell TMapArray[TILE_NUM_X][TILE_NUM_Y];
#0008
#0009    TMapArray FMaps[LAYER_MAX + 1]; // (0 ~ LAYER_MAX) 層地圖
#0010
#0011    int FLevelNo; // 目前載入的關卡編號
#0012
#0013    int FRole_X, FRole_Y; // 角色的起始位置
#0014
#0015    void SetLevelNo(int Value);
#0016
#0017    void SetRole_X(int Value);
#0018    void SetRole_Y(int Value);
#0019  protected:
#0020    TMap();
#0021    virtual ~TMap();
#0022
#0023    AnsiString GetFileName(); // 根據關卡編號, 傳回對應的檔名
#0024  public:
#0025    void LoadFromFile();
```

```
#0026    void SaveToFile();
#0027
#0028    // 將某地圖層畫在 Canvas 上
#0029    void Draw(TCanvas* Canvas, int Layer);
#0030
#0031    TCell& GetCell(int Layer, int x, int y);
#0032
#0033    // 重設整張地圖, 或只重設某一層
#0034    void ResetAllLayers();
#0035    void ResetLayer(int Layer);
#0036
#0037    __property int LevelNo = {read = FLevelNo, write = SetLevelNo};
#0038
#0039    // 取得初始的角色位置
#0040    __property int Role_X = {read = FRole_X, write = SetRole_X};
#0041    __property int Role_Y = {read = FRole_Y, write = SetRole_Y};
#0042  };
```

0007 列宣告存放每層地圖的 *TCell* 二維陣列型態 *TMapArray*，它包含 *TILE_NUM_X* * *TILE_NUM_Y* 個圖格。而 0009 列宣告 *FMaps* 一維陣列，它包含 *LAYER_MAX* + 1 個 *TMapArray* 陣列，也就是每一關卡所需的所有地圖層。除了地圖層、關卡編號，0013 列還記錄著此道關卡中主角的初始位置。這裡應該還要加入其它關卡資訊，例如每道關卡的敵方坦克種類及出現順序等等。不過目前沒有這樣做，敵方坦克的出現時機及種類由亂數決定。

TMap 類別最常用的函式爲 *GetCell* 屬性，它需要三個參數，分別傳入圖層及座標，來取得指定圖層指定座標上的 *TCell* 圖格物件參考。

除了寫入地圖檔及讀出地圖檔兩個函式外，最重要的是繪製地圖層的 *Draw* 函式。此函式會將 *Layer* 參數所指定的圖層畫在 *Canvas* 上頭：

```
#0001  // 將指定的地圖層畫在 Canvas 上
#0002  void TMap::Draw(TCanvas* Canvas, int Layer)
#0003  {
#0004    for (int y = 0; y < TILE_NUM_Y; y++) // 對於每一圖格
#0005      for (int x = 0; x < TILE_NUM_X; x++)
#0006        FMaps[Layer][x][y].Draw(Canvas);
#0007  }
```

因爲所有的透明貼圖、破碎圖格的判斷、處理已包含於 *TCell* 類別，所以 *TMap::Draw* 函

式只需進入兩層迴圈，針對畫面上所有的圖格，呼叫 *TCell::Draw* 函式即可，把繪製整張地圖的複製處理分散各處，自個擊破了。

這就是地圖的繪製函式，在遊戲中，每次更新畫面時，會呼叫此函式四次，分別傳入地形、地形物、物品及高地形物等四個圖層編號。而在遊戲進行中，至少每秒鐘會有十二次以上的重繪動作，才不致讓使用者覺得畫面延滯，正因爲它是影響遊戲進行效率的重大關鍵，所以此段程式碼必須越精簡越好。

圖庫編輯器

目前爲止，我們已將地圖子系統中的四個類別撰寫完成，算是好的開始。不過，由於角色子系統的類別既多且複雜（複雜度主要來自碰撞處理），繼續撰寫角色子系統之前，讓我們先將比較簡單的圖庫編輯器及地圖編輯器完成，程式寫得筋疲力盡後，這兩支程式應該可帶來較爲「具體」的成就感。:p

圖庫編輯器的功能純粹爲管理圖片及圖片群組，只要提供新增／修改／刪除圖片及圖片群組的功能即足夠，絕對是三支程式中最簡單的，好，那我們就先從它下手。

在程式的撰寫步驟上，我還是延續先介面後程式的習慣，先在 C++Builder 整合環境中將使用者介面全部完成，再開始撰寫第一行程式碼。事實上，我本身平日開發大小程式時，也盡量依照這個準則。將使用者介面清楚制定下來後，才可嚴謹地定義出各使用者介面元件之間的互動關係及訊息傳遞流程，最後才在裡頭填寫程式碼，只要介面規劃沒有問題，骨架都椿好了磚頭要擺錯地方也很難。下圖是圖庫編輯器的設計時期畫面：

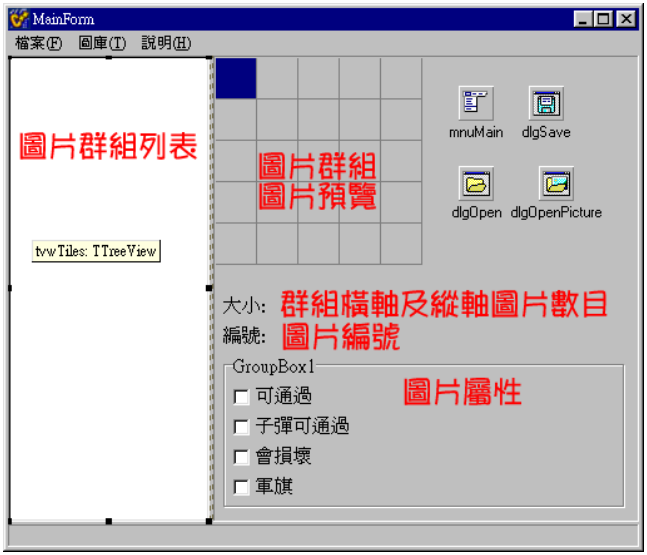

圖 9-7 / 圖庫編輯器主視窗的設計時期畫面

與陽春型圖庫編輯器最大的不同處是，以往以圖片爲操作單位，而現在以圖片群組爲操作單位，但仍可以個別設定每張圖片的屬性。

視窗左邊是一個樹狀檢視元件，使用這個元件原本的用意是，可以做到階層式的圖片群組分類，例如可將旱地、沙地、草地等歸一類，海洋、河流、湖泊等圖片群組歸一類。但你將可在執行畫面圖 9-8 看到，所有圖片群組描述筆直地排成一行，並沒有所謂階層觀念，這是怎麼回事咧？沒事，只是我懶得撰寫新增、刪除群組類別及樹狀檢視元件的節點拖曳搬移等功能，這種「一塊蛋糕」[2]等級的功能，相信你也可以在吃掉一塊蛋糕內的時間內寫出；再說，反正目前圖片少得可憐，也沒有分類的必要，就偷懶沒做這部分了。

[2] Err, I mean the phrase “a piece of cake” !!

圖 9-8 / 圖庫編輯器執行畫面

圖 9-8 中可以清楚地看到，「草怪」這個圖片群組包含 3 x 3 共九張圖片，但在右上角的 *TDrawGrid* 元件中，一次只能選擇一張圖片，再由右下角的四個 *TCheckBox* 元件來設定圖片屬性。並不是每個圖片群組都必須包含多張圖片，事實上，「草怪」是我爲了示範圖片群組功能才特別找來加入的，其它的群組，如草地、磚牆、海洋等，都只包含一張圖片而已。

新增及移除圖片群組

【圖庫】功能表下有【新增】、【移除】兩個功能，分別新增及移除一個圖片群組。新增圖片群組時，由使用者指定一個包含整個圖片群組圖形的 BMP 檔案，由程式計算此圖片群組的橫軸及縱軸圖片數目，接著一一將此群組的圖片切割取出，加入圖庫。

移除圖片群組時，先由使用者在樹狀檢視元件中選定一個群組描述，程式會一一將此群組所有圖片的 *FDisposed* 布林變數設爲 *true*，表示這些圖片已成孤兒，沒人要了，再將該群組描述移除，連身分都毀掉，徹底地讓世人遺忘它們。

新增及移除圖片群組的動作分別由 *mnuAddClick* 及 *mnuRemoveClick* 兩個事件處理函式來負責，程式碼如下：

```
#0001  void __fastcall TMainForm::mnuAddClick(TObject *Sender)
#0002  {
#0003    if (dlgOpenPicture->Execute()) {
#0004
#0005      // 產生及載入欲加入圖庫的 bitmap
#0006      Graphics::TBitmap* Bits = new Graphics::TBitmap;
#0007
#0008      try {
#0009        Bits->LoadFromFile(dlgOpenPicture->FileName);
#0010
#0011        // 若 bitmap 比單張圖片的尺寸還小, 無法處理
#0012        if (Bits->Width < TILE_WIDTH || Bits->Height < TILE_HEIGHT)
#0013          throw Exception("Bitmap is too small");
#0014
#0015        // 最大是 5 x 5 的圖片群組
#0016        // 圖形切割後的橫軸及縱軸圖片數目
#0017        int XNum = MIN(5, Bits->Width / TILE_WIDTH);
#0018        int YNum = MIN(5, Bits->Height / TILE_HEIGHT);
#0019
#0020        int FirstNo;
#0021        // 依序切割出 XNum * YNum 個圖片
#0022        for (int y = 0; y < YNum; y++)
#0023          for (int x = 0; x < XNum; x++) {
#0024            TTile* Tile = new TTile;  // 產生圖片物件
#0025
#0026            // 將對應的圖形複製到圖片的 bitmap 上
#0027            Tile->Bitmap->Canvas->CopyRect(TILE_RECT, Bits->Canvas,
#0028              Rect(x * TILE_WIDTH, y * TILE_HEIGHT,
#0029              (x + 1) * TILE_WIDTH, (y + 1) * TILE_HEIGHT));
#0030
#0031            Tile->FirstTile = (x == 0 && y == 0); // 是不是群組頭頭
#0032            Tile->XNum = XNum;  // 是群組頭頭的話, 負責記錄
#0033            Tile->YNum = YNum;  // 圖片群組的橫軸及縱軸圖片數目
#0034
#0035            // 將產生的新圖片加入圖庫中
#0036            if (Tile->FirstTile)
#0037              // 取得此群組的頭頭編號
#0038              FirstNo = TTiles::Instance().AddTile(Tile);
#0039            else
#0040              TTiles::Instance().AddTile(Tile);
#0041          }
#0042
#0043        // 將群組描述加入樹狀檢視元件
```

```
#0044       AddTreeNode(dlgOpenPicture->FileName, FirstNo);
#0045
#0046       UpdateControlStatus();
#0047     } __finally {
#0048       delete Bits;  // 原始影像沒有用了, 釋放掉
#0049     }
#0050   }
#0051 }
#0052
#0053 void __fastcall TMainForm::mnuRemoveClick(TObject *Sender)
#0054 {
#0055   if (!tvwTiles->Selected) return; // 一定要選定某個群組才行
#0056
#0057   // 取得目前圖片群組首張圖片編號
#0058   int No = (int)tvwTiles->Selected->Data;
#0059
#0060   TTiles& Tiles = TTiles::Instance();
#0061   // 將整個圖片群組的圖片都設爲"棄置"
#0062   for (int i = 0; i < Tiles.Tile[No].XNum *Tiles.Tile[No].YNum; i++)
#0063     Tiles.Tile[No + i].Disposed = true;
#0064
#0065   tvwTiles->Selected->Delete(); // 將圖片群組描述砍掉
#0066   UpdateControlStatus();
#0067 }
#0068
#0069 void __fastcall TMainForm::AddTreeNode(AnsiString FileName, int
#0070   FirstNo)
#0071 {
#0072   // 在樹狀檢視元件中加入此圖片群組的節點（描述）
#0073   // 圖片群組描述預設值爲加入的 BMP 圖形檔檔名
#0074   TTreeNode* node = tvwTiles->Items->Add(NULL,
#0075     ExtractFileNameNoExt(FileName));
#0076   node->Data = (void*)FirstNo;  // 記錄此群組對應的第一張圖片編號
#0077
#0078   // 若未選擇任何群組, 則幫他選擇第一個節點
#0079   if (tvwTiles->Selected == NULL)
#0080     tvwTiles->Selected = tvwTiles->Items->GetFirstNode();
#0081 }
```

0075 列的 *ExtractFileNameNoExt* 函式由 xFiles 單元提供，傳入一個檔案名稱，它會傳回除去副檔名後的結果，我用它來作爲群組描述的預設名稱。例如若匯入「綠油油的草地.BMP」，則此圖片群組的描述就會被設定爲「綠油油的草地」。此後你可以隨時經由樹狀檢視元件的修改節點文字功能來改變群組描述。

圖片與圖片群組的關聯

由於樹狀檢視元件的每個節點（*TTreeNode* 物件）擁有一個可供使用者自行應用的 *Data* 屬性，因此拿它來儲存此節點所對應的圖片群組再理想也不過了。雖然 *Data* 屬性的資料型態爲 *Pointer*，但在 Win32 下，指標爲 4 bytes，而整數也爲 4 bytes，同樣是那 32 bits，誰也管不著你怎麼用它。所以我直接拿它來儲存所對應圖片群組的首張圖片編號，只不過在指定及讀取時必須進行轉型，如 0058 及 0076 列。

儲存群組頭頭的圖片編號就夠了嗎？是的，因爲循著此編號找到首張圖片後，即可取得此圖片群組的橫軸及縱軸圖片數目，那程式就有足夠的資訊來顯示或使用此群組了。如 *mnuRemoveClick* 函式中 0057 ~ 0063 列的動作，由首張圖片取得圖片群組資訊後，就可一一將此群組裡所有圖片設爲「棄置」，讓整個圖片群組從此消失。

圖片群組描述的永續性

不過呢，也許你已發現，那些顯示在 *tvwTiles* 樹狀檢視元件的圖片群組描述似乎沒有記錄下來，*TTile* 及 *TTiles* 類別似乎也找不著與群組描述相關的欄位及程式碼，那麼這些群組描述是如何保存下來並維持資料永續性呢？答案在這：

```
#0001  void __fastcall TMainForm::mnuOpenClick(TObject *Sender)
#0002  {
#0003    if (dlgOpen->Execute()) // 開啓舊檔對話盒
#0004      try {
#0005        mnuNew->OnClick(NULL); // 先釋放圖庫內容
#0006
#0007        // 載入圖庫
#0008        TTiles::Instance().LoadFromFile(dlgOpen->FileName);
#0009        // 讀取圖片群組描述
#0010        ReadComponentResFile(
#0011          ChangeFileExt(dlgOpen->FileName, ".TVW"), tvwTiles);
#0012
#0013        FFileName = dlgOpen->FileName; // 讀取地形圖庫成功
#0014
#0015       // 若未選擇任何群組, 則幫他選擇第一個節點
```

```
#0016       if (!tvwTiles->Selected)
#0017         tvwTiles->Selected = tvwTiles->Items->GetFirstNode();
#0018     } __finally {
#0019       UpdateControlStatus();
#0020     }
#0021  }
#0022
#0023  void __fastcall TMainForm::mnuSaveClick(TObject *Sender)
#0024  {
#0025    // 若是"另存新檔" 或還未指定檔名, 就先問使用者檔名
#0026    if (dynamic_cast<TComponent*>(Sender)->Tag == 1 ||
#0027        FFileName == "") {
#0028      dlgSave->Filter = dlgOpen->Filter;
#0029      // 詢問使用者檔名, 若按取消就離開
#0030      if (!dlgSave->Execute()) return;
#0031      FFileName = dlgSave->FileName; // 將檔名記起來
#0032    }
#0033
#0034    BackupAttr(grdTile->Col, grdTile->Row); // 儲存目前圖片屬性
#0035
#0036    TTiles::Instance().SaveToFile(FFileName); // 儲存圖庫
#0037    // 儲存圖片群組描述
#0038    WriteComponentResFile(
#0039      ChangeFileExt(FFileName, ".TVW"), tvwTiles);
#0040    UpdateControlStatus(); // 更新視窗標題
#0041  }
```

0010 列的 *ReadComponentResFile* 函式及 0038 列的 *WriteComponentResFile* 函式即是關鍵所在，就是這短短的兩行呼叫，解決了群組描述的資料永續需求。這兩個強力函式的原型如下：

```
TComponent* __fastcall ReadComponentResFile(const AnsiString FileName,
  TComponent* Instance);

void __fastcall WriteComponentResFile(const AnsiString FileName,
  TComponent* Instance);
```

兩者參數相同，皆需傳入一個檔案名稱及一個元件。*WriteComponentResFile* 會將 *Instance* 元件的 *__published* 區段屬性值寫入 *FileName* 檔案；而 *ReadComponentResFile* 的功能恰恰相反，將由 *WriteComponentResFile* 寫入的元件資訊讀取回來，回復 *Instance* 元件的原來狀態。藉由這兩支函式，可以很輕鬆、很偷懶地將 VCL 元件的狀態、資料儲存起來，供日後讀取，回復爲元件原來的狀態。

Info

這兩支函式內部依賴的正是 VCL 的 streaming 機制，所以儲存格式與 DFM 檔相同。*WriteComponentResFile* 函式並不真正將所有的 *__published* 區段屬性值寫入檔案，而只存入與預設屬性值不同的屬性；除此之外，元件也可自由決定是否儲存額外的內部資訊。

藉著 VCL 的永續機制，這些辛辛苦苦由 *mnuAddClick* 函式建立的群組描述，只要將儲放群組描述節點的 *tvwTiles* 整個備份起來，存到檔案中，下次需要的時候再整批還原即可，連帶記錄著群組頭頭圖片編號的 *TTreeNode*::*Data* 屬性也一併儲存還原，如同啥事都沒發生過，方便極了。

偷懶法的利與弊

這當然不是最理想的辦法，正規的方法是將群組描述隨著首張圖片與其它群組資訊一塊存放，並在讀取圖庫後尋訪所有圖片以重新建立樹狀檢視元件節點，這必須多寫一些程式碼。而我們的偷懶法雖然簡單省事，但天底下總沒有那麼完美的事，這方法的弊端是，將圖庫和群組描述分爲兩個獨立的檔案存放，若是不小心刪除其中一個，整套圖庫就毀了，無法使用。而且於架構上，於資料封裝的觀點看來，群組描述由圖片頭頭自行處理才是正道。

不過意外的好處是，因爲圖片群組只對圖庫編輯器及地圖編輯器有用，對遊戲主程式則完全沒有意義，因爲遊戲中完全不需圖片群組的概念，只消將正確的圖片繪出即可。因此移交遊戲給外界時，只須附上圖庫檔案，群組描述檔案自己留著，可以減少遊戲所佔用的磁碟空間。

預視圖片群組

視窗右上角提供即時預視圖片群組能力的是 *TDrawGrid* 元件，欲讓它可以正確地顯示圖片群組，只要撰寫 *OnDrawCell* 事件處理函式即可：

```
#0001  void __fastcall TMainForm::grdTileDrawCell(TObject *Sender,
#0002    int ACol, int ARow, TRect &Rect, TGridDrawState State)
#0003  {
#0004    if (!tvwTiles->Selected) return; // 一定要選擇某群組才行
#0005
#0006    // 此格所對應的圖片編號 = 首張圖片編號 + ARow * 橫軸數目 + ACol
#0007    int No = (int)tvwTiles->Selected->Data +
#0008      ARow * grdTile->ColCount + ACol;
#0009    if (No >= TTiles::Instance().TileNum) return; // 是否爲合法編號 ?
#0010
#0011    // 畫出對應的圖片
#0012    grdTile->Canvas->Draw(Rect.Left, Rect.Top,
#0013      TTiles::Instance().Tile[No].Bitmap);
#0014  }
```

是否覺得圖庫編輯器的挑戰性不高呢？沒關係，將遊戲所需的圖片群組通通加入圖庫，儲存爲 TILES.TIA 檔案後，將它擱置一旁。緊接著要將地圖編輯器也一口氣拼出來，小陳飛刀既出，請接招囉。

地圖編輯器

地圖編輯器，按照定義，可稱呼爲**「以所見即所得方式編輯地圖圖片編號的編輯器」**。所以呢，只要忠實地將目前的地圖畫出來，並配合使用者的輸入改變地圖資訊，讓使用者可以輕鬆地編輯地圖，就是成功的地圖編輯器。聽起來一點也不難，那就開始拉元件吧！

哦不，忘了說明一件事，*TMap* 類別除了存放關卡的四層地圖外，另外也儲存每道關卡我們的初始位置，但是我們還沒有寫出角色子系統呀，所以主角位置設計這部分得先略過，爲它預留空間，以後待 *TMyTank* 類別完成後再補進去即可。

圖 9-9 是地圖編輯器主視窗的設計時期畫面，看起來，嗯，很沮喪，空洞洞的，只有三個元件，左方的地圖畫面，右上方用來列出圖片群組的樹狀檢視元件及右下角可供預視及選擇圖片的 *TDrawGrid* 元件。

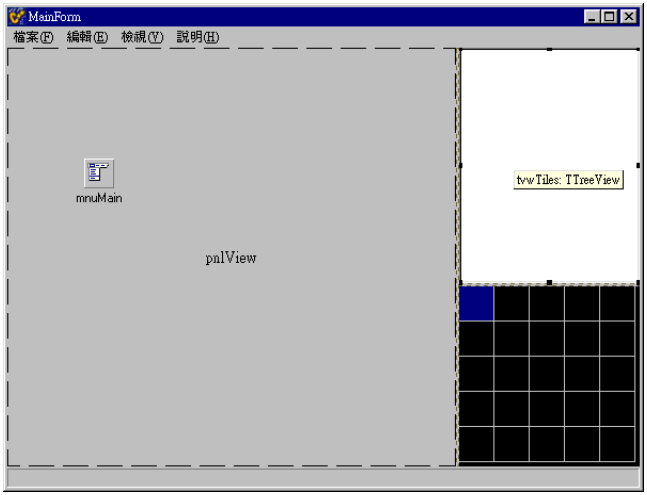

圖 9-9 / 地圖編輯器的設計畫面

地圖編輯器中所使用的技巧，包括使用 double-buffering 來繪出地圖、利用 *OnMouseMove* 事件來達成小藍框框跟著滑鼠指標跑的效果、繪製特殊區域等等，都與前一章「足球番」的地圖編輯器一模一樣，在此就不重覆介紹。

程式一開始，在 *TMainForm* 的 *OnCreate* 事件處理函式中，首先由全域物件變數 *Tiles* 載入圖庫，接著再呼叫 *ReadComponentResFile* 函式將搭配圖庫使用的樹狀檢視元件資料還原至右上角的 *tvwTiles*，使圖片群組描述重現；而右下角的 *TDrawGrid*，也和圖庫編輯器中的 *grdTiles* 負責相同的任務，當使用者選擇某圖片群組時，繪出此圖片群組。

靈活的圖片群組操作功能

TDrawGrid 還提供一個功能，讓使用者能在其上拉曳一塊矩形區域，只取圖片群組的某

部分貼到地圖上，ㄟ，十分難表達，請看看圖 9-10 的執行畫面。你瞧，雖然咱們可愛的「草怪」佔有 3 x 3 張圖片，但我可以只取其左上角 2 x 2 張，貼在地圖上，很靈活吧。地圖上跟隨著滑鼠指標移動的藍色矩形區域，會隨著你在 *grdTile* 上拉曳的矩形區域大小而變，這能力是由 *TDrawGrid* 的 *Selection* 屬性而得：

```
#0001  // 取得選擇區域的寬及高
#0002  int __fastcall TMainForm::GetSelectionWidth()
#0003  {
#0004    return grdTile->Selection.Right -
#0005      grdTile->Selection.Left + 1;
#0006  }
#0007
#0008  int __fastcall TMainForm::GetSelectionHeight()
#0009  {
#0010    return grdTile->Selection.Bottom -
#0011      grdTile->Selection.Top + 1;
#0012  }
```

GetSelectionWidth 及 *GetSelectionHeight* 函式分別是 *SelectionWidth* 及 *SelectionHeight* 屬性的屬性存取函式，因此在程式中隨時取用這兩個屬性，都能夠正確地回傳目前 *grdTile* 元件使用者選擇的區域大小。

圖 9-10 / 「草怪」群組佔有 3 x 3 張圖片，但可只取其左上角 2 x 2 張圖片

地圖編輯模式

編輯模式共有五種，分別是四層地圖層及角色位置設定，如圖 9-11。程式中使用 *FEditLayer* 變數來記錄，若 *FEditLayer* 爲 0 ~ *LAYER_MAX*，表示正在編輯對應的圖層，否則爲我方坦克位置設定模式。

圖 9-11 / 可選定任一種編輯模式來進行設計工作

地圖圖層的資料設定

程式中最重要的動作就屬按下滑鼠鍵時，對於地圖圖層的資料設定及清除工作了，這份工作是由 *pbxView* 的 *OnMouseDown* 事件處理函式來擔綱：

```
#0001  void __fastcall TMainForm::pbxViewMouseDown(TObject *Sender,
#0002       TMouseButton Button, TShiftState Shift, int X, int Y)
#0003  {
#0004    // 記錄按下的滑鼠鍵, 配合 OnMouseMove 事件處理函式
#0005    // 產生拉曳設定效果
#0006    FButtonPressed = Button;
```

```
#0007
#0008   if (Button == mbMiddle) return; // 滑鼠中鍵不做任何事
#0009
#0010   TMap& Map = TMap::Instance();
#0011
#0012   if (Button == mbLeft) { // 左鍵是設定
#0013     if (!tvwTiles->Selected) return; // 沒有選定任何群組
#0014
#0015     int No = (int)tvwTiles->Selected->Data; // 圖片群組的頭頭編號
#0016
#0017     // 左上角, 正上方及右上角三處是敵方坦克的出生點, 不能放東西
#0018     if (FEditLayer != LAYER_TERR && FSelectionY == 0 &&
#0019       (FSelectionX == 0 || FSelectionX == TILE_NUM_X / 2 + 1 ||
#0020       FSelectionX == TILE_NUM_X - 1)) return;
#0021
#0022     if (FEditLayer <= LAYER_MAX) {
#0023       // 將新地形擺上
#0024       for (int MY = 0; MY < GetSelectionHeight(); MY++)
#0025         for (int MX = 0; MX < GetSelectionWidth(); MX++)
#0026           Map.GetCell(FEditLayer, FSelectionX + MX,
#0027           FSelectionY + MY).TileNo =
#0028             No + TTiles::Instance().Tile[No].XNum *
#0029             (MY + grdTile->Selection.Top) +
#0030             (MX + grdTile->Selection.Left);
#0031     } else {
#0032       // 角色不可以擺在不可走動的地形上
#0033       if (!Map.GetCell(LAYER_TERRITEM, FSelectionX,
#0034         FSelectionY).CanPass) return;
#0035
#0036       FTank->PosX = TILEWIDTH[FSelectionX]; // 設定主角初始位置
#0037       FTank->PosY = TILEHEIGHT[FSelectionY];
#0038     }
#0039   } else { // 右鍵是清除
#0040     if (FEditLayer > LAYER_MAX) return; // 角色不用清除
#0041
#0042     for (int MY = 0; MY < GetSelectionHeight(); MY++) // 清除此地形
#0043       for (int MX = 0; MX < GetSelectionWidth(); MX++)
#0044         Map.GetCell(FEditLayer, FSelectionX + MX,
#0045           FSelectionY + MY).TileNo = 0;
#0046   }
#0047
#0048   FModified = true; // 此地圖已更改
#0049   UpdateView(); // 更新地圖畫面
#0050 }
```

0024 ~ 0030 列將新地形擺上時，不但要利用兩層迴圈一一設定 *FSelectionX* x *FSelectionY*

個圖格，取得圖片編號時還得小心圖片選擇區域不見得由左上角開始，可能由群組中任一張圖片開始拉曳，所以必須考慮 *grdTile->Selection->Top* 及 *grdTile->Selection->Left* 兩屬性。至於清除地形時，就不用考慮這麼多，通通指定爲零就成了。

破碎圖格的編輯能力

有一點沒跟上任天堂版坦克大決戰的地方是，它的地圖編輯器提供破掉一半的磚牆可供編輯，而我們的版本則沒有。其實加入此功能的空間早已預留，你在前頭已經看過，若某個 *TCell* 圖格物件呈破碎狀態，呼叫它的 *SaveToStream* 函式儲存時，*FBreakPtr* 指向的圖格破碎表格也會一併寫入資料流，並且也可從資料流完整地重現圖格破碎表格。依著這樣的設計，只要再加上編輯圖格破碎表格的能力，就可以使用地圖編輯器設計出包含破碎圖格的關卡。這一點都不難，我想差不多也是一塊蛋糕的等級，留待日後再發揮補強囉。嗚，你看，圖 9-13 及圖 9-14 分別是任天堂版及我們的第一道關卡地圖，地圖編輯器才少了一個破碎圖格編輯功能，關卡看起來就遜多了，怨嘆啊。

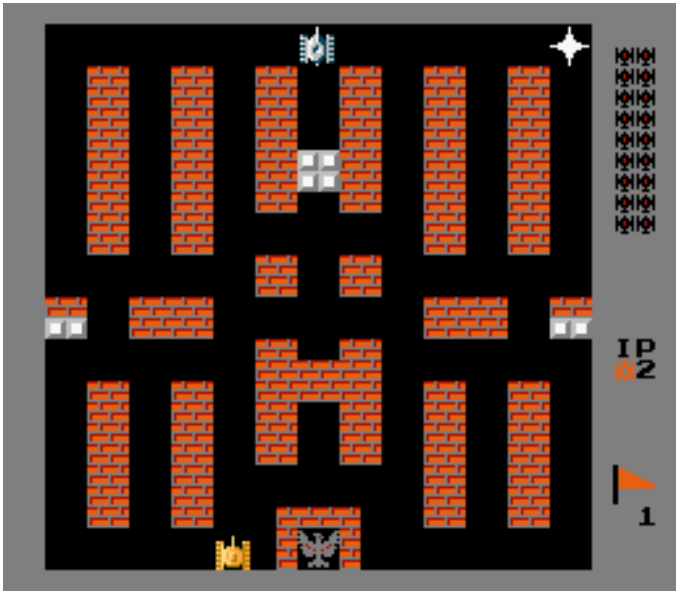

圖 9-13 / 任天堂版第一道關卡

圖 9-14 / 我們的比較遜的第一道關卡

圖層檢視選擇

在操縱地圖編輯器的同時，你可以即時見到對地圖所做的任何更動，不過這樣的操作介面仍有些不便。例如，很可能在某些圖格上放置不必要的圖片，雖然被上層地圖的圖片蓋住了，看不見，但繪製地圖時仍會爲它耗費執行時間。所以我加入圖層檢視選擇功能，讓使用者可選擇是否只想見到目前編輯的那層地圖，如圖 9-12，這就是單獨檢視地形物層的結果。

圖 9-12 / 單獨檢視地形物層

就這樣，說著說著地圖編輯器也開始能讓我編些關卡來「看」了。不過只能「看」，不能玩哪，連角色子系統的影子都還沒見著，辛辛苦苦「按」出來的關卡不曉得何時才能身歷其境地玩它一場。唉，還是認命點，左腳都踏入火坑了，右腳豈有不進來溫暖一下的道理，歡迎進入角色子系統的實作。

角色子系統

由圖 9-5 的類別大局觀看來，*TSprite* 類別是角色子系統所有類別的祖先類別，因此只要好好地設計 *TSprite* 類別，處處預留合理的擴充性，讓繼承它的後代類別們做起事來不至綁手綁腳，這是十分重要的設計原則。虛擬及抽象函式也必須運用得當，有許多情況並不是後代類別單純地改寫（override）幾道虛擬函式就可達成的；函式的切割、類別之間的任務分派等等規劃事宜，絕對會大大影響實作困難度及成品品質。

對於像 *TSprite* 這麼重要的類別，正是訓練自己類別規劃能力的大好時機，而訓練的方法通常是**「設計、實作、驗證、思考、改良、再來一次」**這幾個步驟的循環。對於程式架

構的規劃能力，重新撰寫似乎是設計上、實作上絕佳的訓練方法。

還記得高一時，初得知創世紀的作者Richard當初將創世紀一還是創世紀三重寫了二十多遍，只爲求得更高竿的規劃能力、更面面俱到的設計觀點、更靈活的程式撰寫技巧時，心中的驚訝及感動，真是無可言喻。我自己也曾嘗試重寫自己以往的作品，每每翻閱幾個月前還自鳴得意寫出的程式碼，幾個月後卻爲之臉紅，心想「我怎麼會寫出這種狗屁不通、盤根錯結、旗正飄飄[3]的程式碼？天啊，我不敢承認這是我寫的！」時，大略可以感受到自己在程式設計方面的進展。

系上有位大我一屆，對於程式設計的概念、思考、經驗、實作上都是駭客級的學長，時常對我說，他哪時又把自己的某某程式或程式庫重寫了一遍，又獲得了好多好多感覺、好多好多感動云云。我只有想著，啊，這真是步入程式設計涅盤境界的絕佳法門呀。

呵，老毛病，又離題遠了，送你一支蒼蠅拍，下回再離題時請拍我回來，謝謝。

以下列出角色子系統所需的常數（定義於 Util.h 中）：

```
const char* DR_IMAGES               = "img\\";  // 角色影像檔的存放目錄

const int SPRITE_DEFAULT_SPEED      = 4;        // 角色的預設速度
const int SMOOTH_MOVE_THRESHOLD
     = SPRITE_DEFAULT_SPEED * 3 - 1;            // 平滑移動的門檻值

const int MYTANK_DEFAULT_SPEED      = 5;        // 我方坦克預設速度
const int BULLET_DEFAULT_SPEED      = 10;       // 子彈預設速度
const int BULLET_DEFAULT_BLOW_RANGE = 4;        // 子彈預設爆炸範圍

const int MAX_ETANK_PER_SCENARIO    = 20;       // 每道關卡的敵方坦克數目
const int MAX_TANK_ON_SCREEN        = 5;        // 畫面上最多坦克數目
const int MAX_ETANK_COLLISION_COUNT = 5;        // 敵方坦克連續碰撞次數上限

const float PROBAB_ETANK_SHOT_BULLET  = 0.1;    // 敵方坦克射擊子彈機率
const float PROBAB_ETANK_RANDOM_TURN  = 0.02;   // 敵方坦克隨意轉彎機率
const float PROBAB_ETANK_BORN         = 0.02;   // 敵方坦克出生機率
const float PROBAB_GEM_BORN           = 0.005;  // 寶物出現機率
```

[3] 當程式邏輯不順時，最直覺也最dirty的方法就是：再加一個flag變數進去！

```
#define TIMER_ID_GEM          1        // 寶物生滅所使用的 Timer 編號
#define TIMER_ID_GEM_CLOCK    2        // 時鐘寶物效果所使用的 Timer 編號
#define TIMER_ID_GEM_HAT      3        // 帽子寶物效果所使用的 Timer 編號
#define TIMER_ID_GEM_ARROW    4        // 弓箭寶物效果所使用的 Timer 編號

// 欲傳遞給遊戲主迴圈的視窗訊息編號
#define WM_DESTROY_OBJECT       (WM_USER + 0)     // 摧毀物件指令
#define WM_GAMEOVER             (WM_USER + 1)     // 遊戲結束訊息
#define WM_SPECIAL_CONDITION    (WM_USER + 2)     // 吃到寶物時產生效果指令
#define WM_INIT_LEVEL           (WM_USER + 3)     // 關卡重新開始
```

TSprite 類別

廢話不多說，請進入戰戰兢兢、冷汗直冒的備戰狀態，我們面對的是整套遊戲最艱難的 *TSprite* 類別（定義於 Sprite.h）：

```
#0001  enum TDirection {drUp, drDown, drLeft, drRight}; // 方向
#0002
#0003  // 屬性
#0004  enum TSpriteAttrElement {
#0005     saUndirectionalBitmap, // 圖形不具方向性
#0006     saNoCellCollision, // 不跟圖格碰撞
#0007     saNoTankCollision, // 不跟坦克碰撞
#0008     saNoBulletCollision, // 不跟子彈碰撞
#0009     saAlignWithTerrItem} ; // 走動時會自動對齊地形物, 走起來比較順
#0010
#0011  typedef Set<TSpriteAttrElement, saUndirectionalBitmap,
#0012    saAlignWithTerrItem> TSpriteAttr;
#0013
#0014  // 動畫結束後是否停止或重頭開始
#0015  enum TAdvanceFrameMode {afWrap, afStop} ;
#0016
#0017  // 傳回碰撞結果的陣列型態
#0018  typedef std::vector<TCell*> TCellArray;
#0019  typedef std::vector<TSprite*> TSpriteArray;
#0020
#0021  // 在畫面上行走活動的角色物件
#0022  class TSprite {
#0023  private:
#0024    int FX, FY; // 座標
#0025    int FFrameNo; // 目前顯示的 frame 編號
#0026    int FCollisionCount; // 連續碰撞次數
```

```
#0027
#0028    bool FActive; // 是否進行動作
#0029    bool FVisible; // 是否可見
#0030    bool FOnAir; // 是否在天空
#0031    int FSpeed; // 行進速度
#0032
#0033    TRect FRect; // 佔用矩形區域
#0034    TDirection FDirection; // 行進方向
#0035    TSpriteAttr FAttr; // 屬性
#0036
#0037    bool FPostToDead; // 是否已登記要摧毀
#0038
#0039    Graphics::TBitmap *FBits, *FInvBitmap; // 圖片及貼圖用圖片
#0040
#0041    // 角色中心點所在的圖格位置
#0042    int GetTile_X();
#0043    int GetTile_Y();
#0044
#0045    // 取得角色的寬及高度
#0046    int GetObjectWidth();
#0047    int GetObjectHeight();
#0048
#0049    void SetDirection(TDirection Value);
#0050
#0051    // helper functions for CheckCollisions()
#0052    bool IsRealCellCollision(int Layer, int CellXPos, int CellYPos,
#0053      int& x, int& y);
#0054    void CheckCellCollisionsPos(TCellArray& Collisions, const TRect&
#0055      SpriteRect, int Layer, int CellXPos, int CellYPos,
#0056      int& x, int& y);
#0057  protected:
#0058    // 不同角色有不同的資訊
#0059    TRect FObjectRect; // 角色尺寸
#0060    AnsiString FFileName; // 角色圖形檔名
#0061    int FFrameMax; // 動畫框數
#0062
#0063    int FMoveDelay, FMoveDelayCount; // 下次動作前的延遲次數
#0064    TAdvanceFrameMode FAdvanceFrameMode; // 動畫結束後處理方式
#0065
#0066    virtual void ResetStatus(); // 重設角色狀態
#0067
#0068    // 碰撞檢查觸發函式, 負責呼叫所有的碰撞檢查函式
#0069    virtual bool CheckCollisions(int& x, int& y);
#0070
#0071    // 邊界碰撞檢查
#0072    virtual bool CheckBoundCollisions(int& x, int& y);
```

```
#0073    // 地形物碰撞檢查
#0074    virtual bool CheckCellCollisions(int Layer, int& x, int& y,
#0075      TCellArray& Collisions);
#0076    // 角色碰撞檢查
#0077    virtual bool CheckSpriteCollisions(TSpriteArray& Sprites,
#0078      int& x, int& y, TSpriteArray& Collisions);
#0079  public:
#0080    TSprite();
#0081    virtual ~TSprite();
#0082
#0083    void LoadBits(); // 載入角色圖形
#0084
#0085    virtual void Draw(TCanvas* Canvas); // 繪製角色
#0086
#0087    virtual void Move(); // 進行下一步動作
#0088
#0089    void PostToDie(WPARAM Param = 0); // 登記欲摧毀本身
#0090
#0091    void RandomDirection(); // 隨意選擇方向
#0092    void RandomPosition(); // 隨意擺置
#0093    void CenterWith(TSprite& ASprite); // 與另一角色置中對齊
#0094    void CenterBy(int x, int y); // 使中心點爲 (X, Y)
#0095
#0096    __property int PosX = {read = FX, write = FX};
#0097    __property int PosY = {read = FY, write = FY};
#0098
#0099    __property int Tile_X = {read = GetTile_X};
#0100    __property int Tile_Y = {read = GetTile_Y};
#0101
#0102    __property TDirection Direction =
#0103      {read = FDirection, write = SetDirection};
#0104    __property int Speed = {read = FSpeed, write = FSpeed};
#0105    __property bool Active = {read = FActive, write = FActive};
#0106    __property bool Visible = {read = FVisible, write = FVisible};
#0107    __property bool OnAir = {read = FOnAir, write = FOnAir};
#0108    __property bool PostToDead = {read = FPostToDead};
#0109    __property int CollisionCount = {read = FCollisionCount};
#0110
#0111    __property TRect Rect = {read = FRect};
#0112    __property TSpriteAttr Attr = {read = FAttr, write = FAttr};
#0113
#0114    __property TRect ObjectRect = {read = FObjectRect};
#0115    __property int ObjectWidth = {read = GetObjectWidth};
#0116    __property int ObjectHeight = {read = GetObjectHeight};
#0117  };
```

大部分的變數定義及函式宣告都已加上詳細的註解，請多瀏覽幾回。這兒的 *TSprite* 類別和你心目中的 *TSprite* 類別有哪些相異處？爲什麼？孰優孰劣？爲什麼？多多問自己類似的問題，多多思考，有助於規劃能力的增長。

0026 列宣告的 *FCollisionCount* 變數用來偵測角色的「碰壁」狀況，每當碰到東西時就加一，若順利行走，什麼東西都沒碰到則歸零，改變方向時也歸零。如此一來，此變數就可視爲「連續碰壁計數器」，對於亂數控制的角色而言，若此計數值大於某個上限時，就必須請它轉彎，否則一直卡在牆壁旁或邊界，很難看的耶！

0030 列 *FOnAir* 設定此角色「是否在空中」，這是配合本遊戲的四層地圖層而設。畫面重繪的步驟是，先繪出地形層、地形物層及物品層，接著才畫出角色，最後畫出高地形物層。但是這樣一來就無法將飛行器加入遊戲中，哪有飛機會飛在樹棚底下的呀。所以我再加上 *OnAir* 屬性，若其值爲 *true*，則此角色會在高地形層之後才繪出，*OnAir* 屬性爲 *False* 的角色則按照原設定，在高地形物層之前繪出（也就是在下面）。待會你就會看到，爲了示範這個屬性，我真的在咱們的坦克大決戰中加入飛行器了！:p

0033 列定義*FRect*，記錄著此角色所佔用的矩形區域，會隨著角色移動自動更新。我們必須使用此矩形區域來測試碰撞情形，還記得嗎？*TCell*圖格類別也有個*Rect*屬性，將角色的*Rect*及圖格的*Rect*做交集運算，便可得知角色有沒有撞到地形物了。不過使用矩形做爲碰撞運算單位是極危險的事，萬一你的坦克長的像十字架那種形狀，也就是在其佔用矩形區域內，真正佔用面積極少的情況，就很容易發生子彈明明距離坦克本體還好幾步，但是莫名其妙就爆炸了，這不是因爲坦克有替身使者[4]，是因爲它們的矩形區域已經產生碰撞了。但是以矩形區域來測試碰撞是最有效率的辦法，這是速度與品質的交易，也留待日後改善。

將 *FObjectRect*、*FFileName*、*FFrameMax* 三個變數宣告於 *protected* 區段的用意是，讓後代類別去改變它們，以取得該類別使用的圖形檔名、動畫框數及角色尺寸。*FObjectRect*

[4] 日本漫畫「Jo Jo冒險野郎」裏許多角色擁有的特殊能力，例如承太郎的「白金之星」可使全世界暫停十幾秒。十分好看的漫畫，筆者強力推薦。:P

預設值設爲地圖圖格大小的矩形區域（即 *TILE_WIDTH* x *TILE_HEIGHT*），後代類別的角色若與圖格同樣大小的話就不必改變它。

0063 列 *FMoveDelay* 及 *FMoveDelayCount* 是針對動作特別遲緩的角色而設，每當正常角色行動 *FMoveDelay* 次，它才行動一次。可用於播放爆炸、閃爍效果，讓一個動畫在畫面上持續久一點，而不是一閃而逝。

碰撞處理總管

0069 列的 *CheckCollisions* 函式極爲重要，每一步行動時都會呼叫它來測試角色的碰撞情況。*CheckCollisions* 函式本身並不進行碰撞測試，只負責呼叫所有的碰撞處理函式。*TSprite* 類別提供三種碰撞處理函式，分別是測試邊界碰撞的 *CheckBoundCollisions* 函式、測試地形物碰撞的 *CheckCellCollisions* 函式及測試角色相互碰撞的 *CheckSpriteCollisions* 函式。

CheckCollisions 是個虛擬函式，所以後代類別可以改寫它，讓它呼叫更多種類的碰撞處理函式，例如主角的 *TMyTank* 類別就改寫它以呼叫 *TMyTank* 的寶物碰撞處理函式；而 *TBullet* 類別也改寫它加入子彈碰撞處理函式等等。

這種處理方式可使碰撞測試處理的開放性大增，不必將所有的碰撞測試函式通通放在 *TSprite* 類別，只要提供一個嵌入點（*CheckCollisions* 函式），有需要的後代類別便可將它專屬的碰撞測試函式納入。

所有的碰撞處理函式都宣告爲虛擬函式，讓後代類別有機會改寫。它們都提供傳回值，傳回「判定爲碰撞對象的物件陣列」，這使後代類別改寫碰撞處理函式時，可對父類別的判定結果重新翻案，本來被打入黑五類的傢伙也可來個鹹魚大翻身。舉例而言，*TTile* 類別有個 *taBulletCanPass* 屬性，指這張圖片無論是否能被角色通過，子彈一定可以通過。假設地形物層的某個圖格是不能通過的（不含 *taCanPass* 屬性），當角色經過此圖格時，*TSprite::CheckCellCollisions* 地形物碰撞處理函式會將此圖格納入碰撞名單內，所以坦克

及其它角色都會卡住，無法通過。但是子彈類別 *TBullet*，改寫 *CheckCellCollisions* 函式，取得重新審核的機會，若被判定爲碰撞的圖格包含 *taBulletCanPass* 屬性的話，就無條件釋放，改判無碰撞。這種比蕭薔腿上絲襪更具彈性的**「多審制度」**，賦予各種物件完全宰制行動的能力，完全避免類別設計不良所帶來的挖東牆補西牆或旗正飄飄的可能性。

角色圖形

角色的圖形採用很簡單的處理方式：角色圖形可爲無方向性（例如圓形的飛碟，怎麼轉都是同一個模樣）或上下左右四個方向；而每個方向可有任意張數的輪替動畫。這些圖形通通放在同一張 BMP 圖形，繪製角色時再依需求將對應的區域拷貝出來。

以遊戲中的主角坦克爲例，它有四個方向的圖形，每個方向兩張，一律爲 32 x 32 大小。

圖 9-15 / 主角坦克的角色圖形檔

從主角坦克的角色圖形可看出，四個方向上下左右的圖形依序由左至右排列，而每個方向的動畫則由上至下置放。只要按照規矩設計角色圖形，並適當地改寫虛擬函式宣告類別使用的角色圖形檔名、角色尺寸及動畫張數，就可讓角色擁有正確的外觀。

天經地義，繪製角色圖形的工作由 *TSprite::Draw* 函式來負責：

```
#0001  void TSprite::Draw(TCanvas* Canvas)
#0002  {
#0003    //若不可見或已登記摧毀則不畫
#0004    if (!FVisible || FPostToDead) return;
#0005
#0006    // 動畫框編號不合法則不畫
#0007    if (FFrameNo > FFrameMax || FFrameNo < 0) return;
#0008
#0009    // 先將要秀出的區域拷至 FInvBitmap
```

```
#0010    // 若只有一張圖形, 不分方向性
#0011    if (Attr.Contains(saUndirectionalBitmap))
#0012      FInvBitmap->Canvas->CopyRect(ObjectRect, FBits->Canvas,
#0013      Classes::Rect(0, FFrameNo * ObjectHeight, ObjectWidth,
#0014      (FFrameNo + 1) * ObjectHeight));
#0015    else  // 依目前方向
#0016      FInvBitmap->Canvas->CopyRect(ObjectRect, FBits->Canvas,
#0017        Classes::Rect((int)FDirection * ObjectWidth,
#0018        FFrameNo * ObjectHeight, ((int)FDirection + 1) * ObjectWidth,
#0019        (FFrameNo + 1) * ObjectHeight));
#0020
#0021    // 再畫出 FInvBitmap
#0022    Canvas->Draw(FX, FY, FInvBitmap);
#0023  }
```

角色圖形繪製一律採用透明貼圖，依它的圖形是否具有方向性而有不同的座標計算公式，這些座標計算看似複雜，實則簡單，不信你細看便知，皆是十分直覺的倍數運算罷了。

角色的運作

角色一定要會活動才叫角色，否則就成了盆栽。*TSprite* 類別裡的運作核心爲 *Move* 函式，幾乎可說 *TSprite* 類別所有的變數、狀態、屬性及函式都與 *Move* 函式有直接或間接的關係。

每次呼叫 *Move* 函式，該角色就進行「一動」，這「一動」，包括：

- 若角色已登記準備被摧毀（*FPostToDead* 爲 *true*）或停止運作（*FActive* 爲 *false*），則角色全部活動停止。
- 若有行動延遲設定，則待行動延遲次數達到設定值後才真正行動。
- 若有多張動畫，則換下一張。若動畫已播到最後一張且 *FAdvanceFrameMode* 爲 *afStop*，則此角色停止運作（*FActive* 變成 *false*）。
- 計算下一步的位置，然後針對此新座標進行各種碰撞測試，碰撞測試處理可能帶來各種影響，可能是修正座標，也可能引發爆炸等等。

- 更新角色座標。

遊戲進行中會不斷呼叫所有角色的 *Move* 函式，每個輪迴各呼叫一次，每秒鐘至少十二個輪迴，這可使得遊戲中的角色們持續活動，栩栩如生，像真的有生命一樣。

TSprite::Move 函式程式碼如下：

```
#0001  // 最重要的動作函式, 控制角色的動作
#0002  void TSprite::Move()
#0003  {
#0004    if (FPostToDead) return; // 已經登記準備被摧毀了
#0005    if (!FActive) return; // 不進行任何動作
#0006
#0007    // 每次動作前的延遲次數
#0008    if (FMoveDelay != 0) {
#0009      // 每 FMoveDelay 次才真正 Move 一次
#0010      FMoveDelayCount++;
#0011      if (FMoveDelayCount != FMoveDelay) return;
#0012      FMoveDelayCount = 0; // 歸零
#0013    }
#0014
#0015    if (FFrameMax > 0) { // 若有動畫的話
#0016      FFrameNo++; // 遞增動畫編號
#0017      if (FFrameNo > FFrameMax) { // 播放一個輪迴了
#0018        if (FAdvanceFrameMode == afWrap)
#0019          FFrameNo = 0; // 重新再來
#0020        else {
#0021          FFrameNo = FFrameMax;  // 維持在最後一張動畫
#0022          FActive = false; // 停止動作
#0023        }
#0024      }
#0025    }
#0026
#0027    if (FSpeed == 0) return; // 若速度爲零, 不必移動
#0028
#0029    // 取得目前位置
#0030    int x = FX;
#0031    int y = FY;
#0032
#0033    // 計算下一步的位置
#0034    switch (FDirection) {
#0035      case drUp: y -= FSpeed; break; // 往上走
#0036      case drDown: y += FSpeed; break; // 往下走
#0037      case drLeft: x -= FSpeed; break; // 往左走
#0038      case drRight: x += FSpeed; break; // 往右走
```

```
#0039   }
#0040
#0041   // 檢查所有碰撞
#0042   if (CheckCollisions(x, y))
#0043     FCollisionCount++; //  撞到什麼東西了...
#0044   else
#0045     FCollisionCount = 0; // 什麼都沒撞到, 連續碰撞計數器歸零
#0046
#0047   // 更新位置
#0048   FX = x;
#0049   FY = y;
#0050
#0051   // 更新佔用的矩形區域
#0052   FRect = ObjectRect;
#0053   OffsetRect(&FRect, FX, FY);
#0054 }
```

0042 列呼叫的 *CheckCollisions* 函式是 *Move* 函式的關鍵，也是將具有活動力的角色與位於背景的地圖子系統連結起來的繩引。此處若未進行碰撞處理，則一堆角色在畫面上只是愚蠢地行動著，絲毫不受腳下地形的影響，跋山涉水，如履平地。這還不算什麼，若是於角色之間也沒有碰撞處理，就可以看到一堆坦克疊在一起滑動，太拙了。

麻煩的碰撞檢查

CheckCollisions 函式並不真正擔任碰撞檢查者的工作，它的角色像是「碰撞檢查部門」的主管，負責發號司令，所有的碰撞檢查工作都必須透過它來實行。

TSprite::CheckCollisions 程式碼如下：

```
#0001 // (X, y) 是欲使用的新座標, 傳回值表示是否發生任何碰撞
#0002 bool TSprite::CheckCollisions(int& x, int& y)
#0003 {
#0004   if (FPostToDead) return false; // 已經登記準備被摧毀了
#0005
#0006   bool bCollision = false;
#0007
#0008   // 檢查是否撞到地形物
#0009   if (!bCollision && !FAttr.Contains(saNoCellCollision)) {
#0010     TCellArray CellCollisions;
#0011     bCollision = CheckCellCollisions(LAYER_TERRITEM, x, y,
```

```
#0012       CellCollisions);
#0013    }
#0014
#0015    // 檢查是否撞到邊界
#0016    if (!bCollision)
#0017      bCollision = CheckBoundCollisions(x, y);
#0018
#0019    return bCollision;
#0020  }
```

0002 列中的參數 *x* 及 *y* 指角色的新座標，也就是用來測試碰撞的座標。所有碰撞處理函式皆需要 *x*、*y* 這兩個參數，並且以傳址方式傳遞，使碰撞處理函式**擁有修正新座標的能力**，待會你就可以看到，座標將如何被修正。

此函式傳回一布林值，表示這一動是否發生任何碰撞。在 *Move* 函式中，若 *CheckCollisions* 傳回 *true*，則遞增連續碰撞計數值 *FCollisionCount*；若爲 *false*，則將 *FollisionCount* 重置爲零。

TSprite::CheckCollisions 函式中，依序進行兩種基本的碰撞測試，分別是角色與地形物、角色與邊界的碰撞測試。因爲 *CheckCollisions* 是虛擬函式，所以後代類別可以改寫它，加入新的碰撞測試。

每當完成一項碰撞測試，若此項碰撞測試得知有碰撞發生，該碰撞處理函式就會將傳入的布林參數設爲 *true*，然後返回。而呼叫碰撞處理函式的 *CheckCollisions* 函式得知後，就會省略其它的測試工作，直接跳離函式。這種做法的特性是，一個角色在「一動」中只會發生一種碰撞，亦即，它不是碰到牆壁，就是撞上其它角色，要不然就是超出邊界，絕不會有兩者以上同時發生的情況。好處是可以省下碰撞處理的時間，其次是簡化碰撞結果的處理。否則，當一顆子彈同時打到磚牆及坦克時，要在哪邊爆炸？還是兩者皆爆？何況若兩者皆爆的話，就得爲每顆子彈建立一個爆炸物件陣列，多麻煩哪。

邊界碰撞測試

TSprite::CheckCollisions 所呼叫的三種碰撞測試中，以邊界測試最爲簡單：

```
#0001  bool TSprite::CheckBoundCollisions(int& x, int& y)
#0002  {
```

```
#0003   int OrgX, OrgY;
#0004
#0005   // 邊界檢查
#0006   OrgX = x; OrgY = y;
#0007
#0008   if (y < 0) y = 0; // 是否超出上方
#0009   if (x < 0) x = 0; // 是否超出左方
#0010   if (x + ObjectWidth >= WORLD_WIDTH) // 是否超出右方
#0011     x = WORLD_WIDTH - ObjectWidth;
#0012   if (y + ObjectHeight >= WORLD_HEIGHT) // 是否超出下方
#0013     y = WORLD_HEIGHT - ObjectHeight;
#0014
#0015   // 是否撞到邊界 ?
#0016   return (OrgX != x || OrgY != y);
#0017 }
```

這個函式中，分別檢查角色的四邊是否超出畫面的範圍了，若是的話，則直接修正其座標，讓角色由超出畫面邊緣成爲緊貼著畫面邊緣，這樣一來，無論再怎麼走，角色都不可能離開畫面範圍，十分簡單又有力的邊界檢查。

地形物碰撞測試

檢查地形物碰撞的 *CheckCellCollisions* 函式，由於程式碼實在太長，礙於篇幅無法列出，不過它的檢查方式可以一言蔽之，就是針對以角色爲中心的九個圖格（角色所在圖格加上周圍環繞的八個圖格），一一檢查圖格的矩形區域與角色的新矩形區域是否產生碰撞（呼叫 *IntersectRect* API 函式來判斷），函式十分直覺簡單，只是一堆座標的處理使程式碼寫來十分冗長而已。

另外，*CheckCellCollisions* 函式還包括走動時自動切齊地形物的座標修正功能。此功能是針對擁有 *saAlignWithTerrItem* 屬性的角色設計，本遊戲中所有坦克都具有此屬性，這個功能會讓我們在操作坦克時，若要走進或彎入一條巷道中時，即使坦克位置不是對得很準，只要差距小於 *SMOOTH_MOVE_THRESHOLD* 點數，此功能就會自動幫你修正座標，讓玩者不必爲了讓坦克走某條巷道就得瞄準個好半天。

角色碰撞測試

CheckSpriteCollisions 會檢查角色與其它角色的碰撞情形，並傳回與該角色發生碰撞的角

色陣列（因爲可能同時撞上好多個角色）：

```
#0001  bool TSprite::CheckSpriteCollisions(TSpriteArray& Sprites,
#0002    int& x, int& y, TSpriteArray& Collisions)
#0003  {
#0004    TRect SpriteRect, R;
#0005
#0006    // 計算新的矩形區域
#0007    SpriteRect = ObjectRect;
#0008    OffsetRect(&SpriteRect, x, y);
#0009
#0010    // 一一尋訪傳入的角色
#0011    for (TSpriteArray::iterator p = Sprites.begin();
#0012      p != Sprites.end(); p++) {
#0013      TSprite* &S = *p;
#0014      TRect SRect = S->Rect;
#0015
#0016      if (this != S && !S->PostToDead && S->Visible &&
#0017        IntersectRect(&R, &SpriteRect, &SRect)) {
#0018
#0019        // 將撞到的角色加入碰撞結果陣列
#0020        Collisions.push_back(S);
#0021      }
#0022    }
#0023
#0024    return Collisions.size() > 0; // 表示有碰撞發生
#0025  }
```

CheckSpriteCollisions 函式需要傳入一個 *TSpriteArray* 陣列 *Sprites* 參數，包含所有要拿來進行碰撞測試的角色，此函式會將碰撞到的角色加入同樣是 *TSpriteArray* 陣列型別的 *Collisions* 參數。由於並不是所有角色物件都需要主動進行與其它角色的碰撞測試，因此 *CheckSpriteCollisions* 函式目前只是備而不用，若 *TSprite* 的子類別需要進行角色的碰撞測試，只要改寫 *CheckCollisions* 函式，在於其中呼叫 *CheckSpriteCollisions* 函式即可。

至於 *TSprite* 其它的變數、函式及屬性皆屬座標、面積、屬性管理之類，註解說明得十分清楚，不再一一介紹。

TSprite 類別至此可說是大功告成，因爲它已預留許多空間，供後代類別改寫發揮，角色子系統可說完成了一半。接下來的工作就簡單了，依序由 *TSprite* 類別衍生出角色子系統的其它類別，按圖索驥，將圖 9-5 類別大局觀的類別依階層關係一個個實作出來便成。

TTank 坦克抽象類別

最早的構想是，從 *TSprite* 類別衍生出一個坦克類別，供遊戲中所有坦克使用。不過再仔細想想，我方坦克及敵方坦克差異頗大，例如說我方坦克可吃寶物；敵方坦克可由亂數決定其行動、在出生前會先發出金光閃閃效果等等。這些功能全部由同一個類別負責容易使類別十分複雜，難以處理，所以決定先建立一個擁有坦克功能及特性的抽象類別 *TTank*，我方坦克及敵方坦克再分別由 *TTank* 建立自己的類別。

TTank 類別宣告如下（定義於 Sprite.h 中）：

```
#0001  // 坦克類別, 我方及敵方坦克皆從此類別繼承
#0002  class TTank : public TSprite {
#0003  private:
#0004    int FHP; // 裝甲 (生命力)
#0005    int FBulletNum; // 目前子彈數
#0006    bool FSuperMode; // 無敵模式
#0007
#0008    friend class TBullet;
#0009  protected:
#0010    // 每種角色不同
#0011    int FScore; // 被摧毀後, 主角的得分
#0012    int FMaxBulletNum; //   同一時間子彈數上限
#0013    int FBulletBlowRange; // 子彈爆炸威力 (範圍)
#0014
#0015    // 碰撞檢查觸發函式, 負責呼叫所有的碰撞檢查函式
#0016    virtual bool CheckCollisions(int& x, int& y);
#0017
#0018    // 新增坦克碰撞檢查
#0019    virtual bool CheckTankCollisions(int& x, int& y,
#0020      TSpriteArray& Collisions);
#0021  public:
#0022    TTank();
#0023    virtual ~TTank();
#0024
#0025    virtual void ResetStatus(); // 重設坦克狀態
#0026    virtual TBullet* FireBullet(); // 發射子彈
#0027
#0028    __property int HP = {read = FHP, write = FHP};
#0029    __property int MaxBulletNum =
#0030      {read = FMaxBulletNum, write = FMaxBulletNum};
#0031    __property bool SuperMode =
```

```
#0032     {read = FSuperMode, write = FSuperMode};
#0033  };
```

0013 列宣告 *FBulletBlowRange* 變數，記錄此坦克所發射子彈的爆炸威力（範圍），這是不同於任天堂版坦克大決戰的一點改良，因我們完全使用矩形區域來進行碰撞測試，包括子彈爆炸範圍及可爆破地形物的矩形交集測試。透過此爆炸威力的設定，遊戲中的子彈不再只能呆呆地永遠只破壞八分之一或四分之一塊磚牆，威力弱一點的子彈可能只損壞十六分之一個圖格（因爲 *TCell* 的圖格破碎表格爲 4 x 4 大小），威力強大的子彈甚至能夠一口氣毀壞九個圖格（因爲地形物的碰撞測試只針對圍繞角色的八塊圖格及角色本身所在圖格進行測試），威力相差 144 倍。

咻 ~ 子彈發射

TTank 類別除了加入一堆坦克資訊，如生命力、子彈數目、爆炸範圍外，最有趣的功能就屬 0026 列宣告的 *FireBullet* 子彈發射函式了。想想，若沒有發射子彈的能力，坦克大決戰就不再是坦克大決戰，而是一拖拉庫坦克互相輾來輾去的「碰碰坦克」，多無趣啊。

發射子彈是典型的「由角色產生其它角色」的例子，*FireBullet* 函式程式碼如下：

```
#0001  // 發射子彈, 傳回值爲發射出去的子彈物件
#0002  TBullet* TTank::FireBullet()
#0003  {
#0004    // 目前子彈數若已達上限則不容許再發射
#0005    if (FBulletNum >= FMaxBulletNum) return NULL;
#0006
#0007    TBullet* b = new TBullet(this); // 建立子彈物件
#0008    b->LoadBits(); // 載入子彈圖形
#0009    b->Direction = Direction; // 子彈與坦克本身同樣方向
#0010
#0011    b->CenterWith(*this); // 先將子彈座標設定與坦克置中對齊
#0012
#0013    switch (Direction) { // 根據行進方向來調整子彈座標
#0014      case drUp: b->PosY = Rect.Top; break;
#0015      case drDown: b->PosY = Rect.Bottom - b->ObjectHeight; break;
#0016      case drLeft: b->PosX = Rect.Left; break;
#0017      case drRight: b->PosX = Rect.Right - b->ObjectWidth; break;
#0018    }
```

```
#0019
#0020    FBulletNum++; // 遞增坦克的目前子彈數
#0021    return b;
#0022  }
```

這段程式碼十分易懂，首先檢查目前子彈數目是否已達上限，然後建立子彈物件，適當地設定它的屬性，最後再遞增坦克的目前子彈數目。這顆子彈會自動加入遊戲所維護的子彈陣列中，和其它角色一樣朝著自己眼前的方向一步一步行進，直到擊中什麼東西爲止。

FireBullet 函式會傳回剛發射出去的子彈物件，而且宣告爲虛擬函式，這與碰撞處理函式有異曲同工之妙－留給後代類別平反既定事實的機會。後代類別若有需要，即可改寫 *FireBullet* 函式，取得 *TTank*::*FireBullet* 所發射的子彈物件，將此子彈物件修改爲更合適的狀態，才放行。

坦克與坦克的碰撞行為

TTank 類別加入 *CheckTankCollisions* 函式來處理坦克與坦克的碰撞行爲。*CheckTankCollisions* 函式藉由 *TSprite::CheckSpriteCollisions* 函式的輔助，將畫面上所有坦克傳入，取得碰撞角色列表後，一一重新審查，若發現碰撞的坦克擁有 *saNoTankCollision* 屬性，則此坦克不應該與其它坦克發生碰撞，就將它自碰撞結果陣列移除。程式碼列表如下：

```
#0001  bool TTank::CheckTankCollisions(int& x, int& y, TSpriteArray&
#0002    Collisions)
#0003  {
#0004    // 首先呼叫 CheckSpriteCollisions 函式取得碰撞坦克列表
#0005    TSpriteArray& r_Tanks = *Tanks(); // r_Tanks 指向所有坦克列表
#0006    bool bCollision = TSprite::CheckSpriteCollisions(r_Tanks, x, y,
#0007      Collisions);
#0008
#0009    if (bCollision) { // 若與任何坦克發生碰撞
#0010      for (TSpriteArray::iterator p = Collisions.begin();
#0011        p != Collisions.end(); ) {
#0012        TSprite* &T = *p;
#0013
```

```
#0014       if (T->Attr.Contains(saNoTankCollision))
#0015         Collisions.erase(p);
#0016       else
#0017         p++;
#0018     }
#0019
#0020     bCollision = Collisions.size() > 0; // 重新裁決是否發生碰撞
#0021     // 亡羊補牢, 猶未晚也. 其實根本不算有碰撞的..
#0022     if (!bCollision) return false;
#0023
#0024     // 根據碰撞到的坦克及自己的方向來調整座標
#0025     for (TSpriteArray::iterator p = Collisions.begin();
#0026       p != Collisions.end(); p++) {
#0027       TSprite* &T = *p;
#0028
#0029       switch (Direction) {
#0030         // 把自己放在對方的下方
#0031         case drUp: y = T->Rect.Bottom; break;
#0032         case drDown: y = T->Rect.Top - ObjectHeight; break;
#0033         // 把自己放在對方的右方
#0034         case drLeft: x = T->Rect.Right; break;
#0035         case drRight: x = T->Rect.Left - ObjectWidth; break;
#0036       }
#0037     }
#0038   }
#0039   return bCollision;
#0040 }
```

我方坦克

坦克的抽象類別也定義好後，只要再由其分別衍生我方坦克及敵方坦克，稍加改寫即可。

我方坦克 *TMyTank* 繼承自 *TTank* 類別，改寫 *CheckCollisions* 碰撞觸發函式，呼叫 *TMyTank::CheckGemCollisions* 寶物碰撞處理函式來檢查我方坦克與寶物的碰撞情形。

寶物爲 *TSprite* 的子類別 *TGem* 物件，由於遊戲中任一時間最多只可能有一個寶物存在，所以我宣告一個全域的 *TGem* 物件，*Gem*。當它爲 *NULL* 時，表示目前畫面上沒有寶物，否則指向該寶物物件。

我方坦克加入吃掉寶物能力的程式碼爲：

```
#0001  bool TMyTank::CheckGemCollisions(int& x, int& y)
#0002  {
#0003    if (!Gem) return false; // 寶物不存在, 不可能碰到
#0004
#0005    // 根據新座標計算坦克所佔用的矩形區域
#0006    TRect SpriteRect = ObjectRect;
#0007    OffsetRect(&SpriteRect, x, y);
#0008
#0009    // 坦克是否與寶物所佔矩形區域產生交集
#0010    TRect R, R1 = Gem->Rect;
#0011    if (IntersectRect(&R, &SpriteRect, &R1))
#0012      return true; // 吃到寶物了
#0013
#0014    return false;
#0015  }
#0016
#0017  bool TMyTank::CheckCollisions(int& x, int& y)
#0018  {
#0019    bool bCollision = TTank::CheckCollisions(x, y);
#0020
#0021    // 如果吃到寶物的話 (不影響 bCollision) ...
#0022    if (CheckGemCollisions(x, y)) {
#0023
#0024      // 根據寶物的種類, 傳送訊息給遊戲迴圈處理, 或自己處理掉
#0025      switch (Gem->GemKind) {
#0026        case gkClock:
#0027        PostMessage(0, WM_SPECIAL_CONDITION, TIMER_ID_GEM_CLOCK, 0);
#0028        break;
#0029
#0030        case gkHat:
#0031        PostMessage(0, WM_SPECIAL_CONDITION, TIMER_ID_GEM_HAT, 0);
#0032        break;
#0033
#0034        case gkArrow:
#0035        PostMessage(0, WM_SPECIAL_CONDITION, TIMER_ID_GEM_ARROW, 0);
#0036        break;
#0037
#0038        case gkStar: FMaxBulletNum = 10; // 能夠連發子彈
#0039        break;
#0040
#0041        case gkBlow: FBulletBlowRange = 32; // 超強子彈爆炸威力
#0042        break;
#0043
#0044        case gkApple: Speed = 2 * MYTANK_DEFAULT_SPEED; // 兩倍速度
#0045        break;
#0046      }
```

```
#0047
#0048     // 將寶物摧毀(被吃掉了)
#0049     delete Gem;
#0050   }
#0051
#0052   return bCollision;
#0053 }
```

若 *CheckGemCollision* 函式回傳 *true*，確定吃到寶物後，0025 列會根據寶物的型態做出適當的回應。有些變更只純粹發生在我方坦克上，例如能夠連發子彈、超強子彈爆炸威力、兩倍行走速度的改變，只要更動 *TMyTank* 本身變數即可；但有些寶物的能力，例如暫停敵方坦克動作、爲軍旗加上防護罩等等，必須依賴遊戲系統的支援才能達成。此處你可以先看到我以呼叫 *PostMessage* API 函式將自訂的視窗訊息丟到目前執行緒的訊息佇列中，這些自訂訊息會由誰來處理，爲我們達成特殊效果的請求呢？先賣個關子，後頭再敘。

敵方坦克

所有敵方坦克類別的父類別－*TETank*，也和 *TTank* 一樣都是抽象類別，而所有敵方坦克類別皆繼承它。*TETank* 類別最大的貢獻是，出現時產生金光閃閃效果。

金光閃閃效果是由另一個直接衍生自 *TSprite* 的 *TStar* 類別所負責，*TETank* 負責產生及摧毀 *TStar* 物件，並在適當時候（出生前）不繪出自己，而繪出 *TStar* 物件，即可達成先出現金光閃閃而後敵方坦克才出現的效果。*TStar* 的建構函式如下，它的角色圖形如圖 9-16。

```
#0001 // 敵方坦克出生時金光閃閃的效果類別
#0002 TStar::TStar()
#0003 {
#0004   Speed = 0; // 不動
#0005   Attr = TSpriteAttr() << saUndirectionalBitmap; // 沒有方向性
#0006   FMoveDelay = 8; // 每八個 frame 才變一次
#0007   FAdvanceFrameMode = afStop; // 動畫播完後就停止
#0008
#0009   FFileName = "star.bmp";
#0010   FFrameMax = 2; // 三張動畫
#0011   FObjectRect = Classes::Rect(0, 0, 38, 32);
#0012 }
```

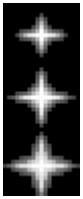

圖 9-16 TStar－金光閃閃類別所用的角色圖形

由此可知，*TStar* 類別即是個不行動、圖形無方向性、緩慢播放動畫，且三格動畫播完即功成身退的角色。而 *TETank* 敵方坦克以如下方式使用 *TStar* 來達成出生效果的目的：

```
#0001  // 敵方坦克的行動函式
#0002  void TETank::Move()
#0003  {
#0004    if (FStar) { // 還在出生中..
#0005      FStar->Move(); // 金光閃閃物件進行下一動
#0006
#0007      if (!FStar->Active) { // 若金光閃閃動畫秀完了..
#0008        delete FStar; // 摧毀金光閃閃物件, 下一動敵方坦克就會出現了..
#0009        FStar = NULL;
#0010      }
#0011    } else {
#0012      // 依亂數"可能"發射子彈
#0013      if (random(100) < PROBAB_ETANK_SHOT_BULLET * 100)
#0014        FireBullet();
#0015
#0016      // 若連續碰撞次數過多, 或亂數許可, 則轉向
#0017      if (CollisionCount > MAX_ETANK_COLLISION_COUNT ||
#0018        random(100) < PROBAB_ETANK_RANDOM_TURN * 100)
#0019        RandomDirection();
#0020
#0021      TTank::Move(); // 進行下一動
#0022    }
#0023  }
#0024
#0025  void TETank::Draw(TCanvas* Canvas)
#0026  {
#0027    if (!FStar)  // 金光閃閃物件是否存在？
#0028      TTank::Draw(Canvas); // 正常的繪製動作
#0029    else
#0030      FStar->Draw(Canvas); // 畫出金光閃閃的星星
#0031  }
```

可以看出，*TETank* 類別除了提供金光閃閃出生效果外，還以亂數控制，主動發射子彈及轉向，當然囉，若連續碰撞次數過多也會轉向。

既然 *TETank* 已具有所有敵方坦克該具備的功能及特色，五種不同的敵方坦克只要在建立 *TETank* 物件及進行某些特定動作時時，按照各自的特徵修改對應的變數即可。哦，還有設定個別的 *FScore* 變數，代表其被主角摧毀後可獲得的分數。

- 第一種敵方坦克
  一切正常，改都沒改。
- 第二種敵方坦克
  將 *FSpeed* 設爲 *SPRITE_DEFAULT_SPEED* * 2，足足比別的敵方坦克快一倍。
- 第三種敵方坦克
  在 *FireBullet* 函式中，將子彈的速度設定爲正常速度的兩倍。
- 第四種敵方坦克
  將 *FSpeed* 設爲 *SPRITE_DEFAULT_SPEED* - 1，*FHP* 設爲 4，跑得慢一點，但打四次才會死。
- 第五種敵方坦克
  嘻，這是我自行新增的「飛行」坦克，狀似蝴蝶。將 *FOnAir* 設爲 *true*，表示在空中飛行；將 *FSpeed* 設爲 *SPRITE_DEFAULT_SPEED* / 2，只有一般敵方坦克的一半速度，但皮實在厚，*FHP* 爲 10。

子彈及爆炸

坦克之外，坦克所發射出的子彈也是戰場上最耀眼的明星之一。它的實作重心在於碰撞處理，不論碰到邊界、地形物、坦克甚至其它的子彈時，都要視情況啓動爆炸效果及產生其它邊際效應，如設定地形物破碎表格、遞減坦克生命力、摧毀其它子彈等等。

子彈類別 *TBullet* 的類別宣告如下（定義於 Sprite.h）：

```
#0001  class TBullet : public TSprite {
#0002  private:
#0003    TTank* FTank; // 產生此子彈的坦克
#0004    TExplosion* FExplosion; // 此子彈所產生的爆炸物件
#0005    bool FTankBulletNum_Dropped; // 是否已將坦克的子彈數目遞減
#0006
```

```
#0007    // 啓動爆炸效果 (建立爆炸物件)
#0008    void FireExplosion(const std::type_info& typeinfo);
#0009    void LocateExplosion(TRect R); // 將爆炸置於矩形區域中心
#0010  protected:
#0011    // 改寫碰撞檢查觸發函式
#0012    virtual bool CheckCollisions(int& x, int& y);
#0013
#0014    // 改寫所有的碰撞處理函式, 同時新增與子彈的碰撞處理函式
#0015    virtual bool CheckBoundCollisions(int& x, int& y);
#0016    virtual bool CheckCellCollisions(int Layer, int& x, int& y,
#0017      TCellArray& Collisions);
#0018    virtual bool CheckTankCollisions(int& x, int& y,
#0019      TSpriteArray& Collisions);
#0020    virtual bool CheckBulletCollisions(int& x, int& y,
#0021      TSpriteArray& Collisions);
#0022  public:
#0023    TBullet(TTank* ATank);
#0024    virtual ~TBullet();
#0025
#0026    virtual void Draw(TCanvas* Canvas);
#0027    virtual void Move();
#0028
#0029    __property TTank* Tank = {read = FTank, write = FTank};
#0030  };
```

子彈所引發的爆炸效果由 *TExplosion* 類別提供，它的功用如同 *TStar* 類別，也只是擺在那邊好看，慢速播放效果動畫的物件。而 *TExplosion* 又是個抽象類別，真正的爆炸由它的兩個子類別 *TSmallExplosion* 及 *TBigExplosion* 提供。這兩個類別唯一的不同就是圖形大小及動畫數目罷了，分別使用不同的角色圖形，如圖 9-17。

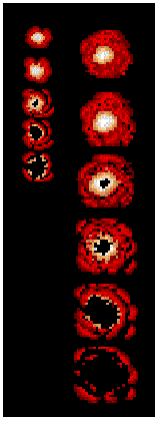

圖 9-17 / 大小爆炸所用的圖形，分別有六張及五張動畫

而 *TBullet* 使用 *TExplosion* 物件的方式也和 *TETank* 使用 *TStar* 物件的方式無二，不再重覆說明。

TBullet 的建構及解構函式中，會將本身加入 *Bullets* 全域陣列及將本身自陣列中移除。這也就是爲什麼在 *TTank::FireBullet* 函式中，產生子彈物件後，什麼記錄都不必留，也不怕遺失對於子彈物件的參考，因爲子彈會負責自己的登錄工作。

TBullet 類別的重頭戲在於個碰撞處理函式的改寫：

- 改寫 *CheckBoundCollisions* 函式
  撞上任一邊界時，引發小爆炸。
- 改寫 *CheckCellCollisions* 函式
  去除帶有 *taBulletCanPass* 屬性的圖格，引發小爆炸，並針對每個碰撞到的圖格計算破碎程度，更新圖格的破碎表格。若炸到的是帶有 *taFlag* 屬性的圖格，表示此圖格放置軍旗，就發個訊息告知遊戲結束。
- 新增 *CheckTankCollisions* 函式
  若撞到的坦克是發射子彈本身的坦克當然沒事（不過坦克要被自己發射的子彈 K 中倒也不容易）；若撞到的坦克及主人皆屬敵方坦克，則也沒事，讓子彈繼續飛行；否則的話，表示主角子彈打到敵方坦克或敵方子彈打到主角，先製造一場大爆炸，同時遞減被炸到坦克的生命力，若生命力等於零，則摧毀該坦克。
- 新增 *CheckBulletCollisions* 函式
  將子彈與畫面上所有其它子彈皆進行碰撞測試。若子彈主人互爲敵我，則兩子彈抵消掉，將自己及對方子彈皆申報作廢；否則，當作沒事，子彈繼續飛行。

好囉，重點在於瞭解類別及函式的任務分派，所以只有簡述，沒有程式碼，本文的重點在於如何架構一個完整的遊戲程式，而非撰寫細密繁雜的程式碼。就這樣，快刀亂麻、秋風落葉般地將角色子系統完成。好累吧～該是將它們全部兜在一塊的時刻了。

遊戲的誕生

下圖是遊戲主視窗的設計時期畫面，哇，夠簡單的了。比足球番程式用的元件還少，不過別擔心，因爲這是遊戲寫作，而不是資料庫程式設計課程，遊戲畫面與元件的使用無關，只要給我一個視窗，一張畫布，任是什麼樣的遊戲畫面也生得出來。

圖 9-18 / 遊戲主視窗的設計時期畫面

有了先前設計的兩大子系統爲靠山，遊戲主程式的撰寫真的可以「用兜的」，不必再煩惱什麼碰撞處理，也不必在意圖片、圖層及圖格之間曖昧不明的三角關係。身處直接面對遊戲玩家的第一線上，遊戲主程式只要負責將**畫面**、**使用者輸入**及**遊戲邏輯**三部分打點好，其餘的交給背後的地圖及角色子系統去操心吧。

繪製遊戲畫面

無可避免地，由於速度上的考量，我們不直接在畫布上進行貼圖動作，而先暗地裡在另一個相同大小的 bitmap 上作畫，等到全部繪製完成後再一口氣複製到視窗上。

DrawBackBitmap 做的就是這些動作，首先依序畫出地形層、地形物層、物品層、寶物、

地面坦克、子彈、高地形物層及空中坦克。接著才根據目前的遊戲狀態繪出玩家坦克生命力、得分等等。*DrawBackBitmap* 函式如下：

```
#0001  // 先將畫面繪製在緩衝用 bitmap, 再複製到視窗上
#0002  void __fastcall TMainForm::DrawBackBitmap()
#0003  {
#0004    TMap& Map = TMap::Instance();
#0005
#0006    if (mnuDrawLayer1->Checked)
#0007      Map.Draw(FBackBitmap->Canvas, LAYER_TERR); // 繪製地形層
#0008    else { // 若沒畫地形層, 就塗黑
#0009      FBackBitmap->Canvas->Brush->Color = clBlack;
#0010      FBackBitmap->Canvas->FillRect(FBackBitmap->Canvas->ClipRect);
#0011    }
#0012
#0013    if (mnuDrawLayer2->Checked) // 繪製地形物層
#0014      Map.Draw(FBackBitmap->Canvas, LAYER_TERRITEM);
#0015
#0016    if (mnuDrawLayer3->Checked) // 繪製物品層
#0017      Map.Draw(FBackBitmap->Canvas, LAYER_ITEM);
#0018
#0019    // 若有寶物, 畫出寶物
#0020    if (Gem) Gem->Draw(FBackBitmap->Canvas);
#0021
#0022    DrawTanks(FBackBitmap->Canvas, false); // 畫出地面上所有坦克
#0023
#0024    DrawBullets(FBackBitmap->Canvas); // 畫出子彈
#0025
#0026    if (mnuDrawLayer4->Checked) // 繪製高地形物層
#0027      Map.Draw(FBackBitmap->Canvas, LAYER_HITERRITEM);
#0028
#0029    // 畫出"空中"所有坦克
#0030    DrawTanks(FBackBitmap->Canvas, true);
#0031
#0032    FBackBitmap->Canvas->Font->Color = clWhite;
#0033    FBackBitmap->Canvas->Font->Name = "FixedSys";
#0034    FBackBitmap->Canvas->Font->Style = TFontStyles() << fsBold;
#0035    FBackBitmap->Canvas->Font->Size = 14;
#0036    FBackBitmap->Canvas->Brush->Style = bsClear;
#0037
#0038    TRect R;
#0039    switch (FGameStatus) {
#0040      case gsTitle:
#0041        // 畫出上面的標題大字及下方的作者名稱
#0042        R = Rect(0, 0, TILE_WIDTH * TILE_NUM_X, TILE_HEIGHT *
#0043          TILE_NUM_Y - FBackBitmap->Canvas->TextHeight("我") / 2);
```

```
#0044       DrawText(FBackBitmap->Canvas->Handle, "作者: 陳寬達", -1, &R,
#0045         DT_BOTTOM | DT_CENTER | DT_SINGLELINE);
#0046       DrawStatusBox("歡迎光臨 坦克大決戰", true);
#0047       break;
#0048
#0049      case gsOver: // Ouch, 軍旗被幹掉或主角死掉了
#0050         DrawStatusBox("任務失敗", false);
#0051       break;
#0052
#0053     default:
#0054        // 在右上角顯示生命力及分數
#0055        AnsiString Str = Format("裝甲 %.2d 得分 %.4d",
#0056          OPENARRAY(TVarRec, (FTank->HP, FScore)));
#0057        FBackBitmap->Canvas->TextOut(WORLD_WIDTH -
#0058          FBackBitmap->Canvas->TextWidth(Str) - 5, 5, Str);
#0059
#0060        // 在左上角顯示關卡
#0061        Str = Format("LEVEL %d", OPENARRAY(TVarRec, (FLevelNo)));
#0062        FBackBitmap->Canvas->TextOut(5, 5, Str);
#0063
#0064       break;
#0065    }
#0066
#0067    // 這是過關畫面
#0068    if (FGameStatus == gsSuccess)
#0069      DrawStatusBox("任務成功 !!", false);
#0070  }
```

程式共分爲五種狀態，分別爲：

```
enum TGameStatus {gsTitle,     // 歡迎畫面
                  gsPlaying,   // 遊戲進行中
                  gsSuccess,   // 過關畫面
                  gsOver,      // GAME OVER
                  gsTerminate};    // 程式即將關閉
```

我希望能在歡迎畫面時，一邊秀出第一關的地圖，一邊有一群坦克們在裡頭忙碌地移動及破壞地形物，所以進入 *gsTitle* 狀態時，隨機產生 *MAX_TANK_ON_SCREEN* 輛敵方坦克，並將我方坦克藏起來，免得使用者與遊戲中畫面搞混，煞有其事地拿起搖桿來砰砰砰。另一方面利用 *tmrTitle* 計時器，每 20 毫秒進行一動，便可模擬遊戲進行時敵方坦克的行爲。

設定 *GameStatus* 屬性時，會呼叫到它的屬性存取函式 *SetGameStatus*：

```
#0001  void __fastcall TMainForm::SetGameStatus(TGameStatus Value)
#0002  {
#0003    FGameStatus = Value;
#0004
#0005    // 根據新的遊戲狀態開關計時器
#0006    tmrTitle->Enabled = FGameStatus == gsTitle;
#0007
#0008    switch (FGameStatus) {
#0009      case gsTitle:
#0010        LevelNo = 1; // 歡迎畫面顯示第一關地圖
#0011
#0012        // 隨機產生 MAX_TANK_ON_SCREEN 輛敵方坦克
#0013        for (int i = 1; i <= MAX_TANK_ON_SCREEN; i++)
#0014          CreateETank(random(5) + 1);
#0015
#0016        FTank->Visible = false; // 將自己坦克藏起來
#0017        FTank->Active = false; // 自己坦克不要動
#0018        break;
#0019
#0020      case gsPlaying:
#0021        UpdateControlStatus(); // 更新標題列及其它控制項
#0022        UpdateView(); // 更新遊戲畫面
#0023
#0024        GameLoop(); // 進入遊戲主迴圈 (遊戲進行中都一直在此迴圈內)
#0025        break;
#0026
#0027      case gsSuccess:
#0028        break;
#0029    }
#0030
#0031    if (FGameStatus != gsTerminate) { // 使用者是否欲關閉視窗 ??
#0032      UpdateControlStatus(); // 更新標題列及其它控制項
#0033      UpdateView(); // 更新遊戲畫面
#0034    } else
#0035      Close(); // 關閉視窗
#0036  }
```

0024 列呼叫 *GameLoop* 函式，這個函式可說是集天下之大成，整個遊戲進行中都會一直處於這個函式內執行。爲什麼要這麼麻煩？怎麼不像上回足球番程式那樣，使用者按一下，行動一步，畫面重繪一次，不是很簡單嗎？

差別在於：足球番是非即時遊戲，而坦克大決戰是即時遊戲。所謂即時遊戲意指遊戲不

論是否有玩者輸入事件（按鍵或扳動搖桿等等），都會持續不間斷地進行份內該做的事。例如，射出子彈後，不論玩者跑去上廁所或不斷地敲擊空白鍵，子彈還是得繼續飛，敵方坦克也得繼續忙碌著亂跑。

你可能會想到使用計時器，就像我們的歡迎畫面那樣，每隔很短的一段時間就觸發一次，讓所有角色進行一動，再更新畫面。

聽來很合理，不過現實總是不同。重點是 Windows 裡的計時器功能並不值得依賴，這組計時器功能依賴 *WM_TIMER* 視窗訊息來觸發事件或呼叫回呼函式，而 *WM_TIMER* 視窗訊息又是幾百個標準 Windows 視窗訊息裡優先權最低的幾個，只要有其它事發生，*WM_TIMER* 一定會遲到甚至不到，這樣沒有時間觀念的傢伙你還敢讓它擔任維持遊戲時程的重責大任嗎？（請參閱第四章「分秒必爭，細說計時器」，對計時器機制有十分詳盡的解說。）

因此我們通常選擇取得完全主控權的方法，換句話說，讓遊戲進行中，不論有無任何事件發生，執行的仍是我們爲它準備好的程式碼。靠著 application framework 的幫忙，平日撰寫應用程式時，我們絲毫不必知道當程式處於閒置狀態時，它在做什麼？反正只要有人傳遞訊息過來，將感興趣的訊息攔截下來，做出適當回應就好了，其它就不用管了。

說得具體一點，VCL的*TApplication*類別不但背負整個應用程式的生滅，也包括訊息的處理。Win32 架構中，一個執行緒只要擁有視窗，就同時擁有一個訊息佇列，此執行緒必須常常去詢問取得佇列中的視窗訊息，並分派給訊息的目的視窗的視窗函式。而負責取得分派視窗訊息的迴圈就稱爲訊息迴圈，對於擁有視窗的執行緒而言，終其一生皆在訊息迴圈內度過。打開專案原始碼，看到那行熟悉的*Application->Run*呼叫了沒？裏頭包含的正是主執行緒的訊息迴圈[5]。

說得再具體一點，我們必須另開新局，自行建立一個訊息迴圈，在遊戲進行中，暫時將

---

[5] 三言兩語要道盡訊息、訊息佇列、訊息迴圈及VCL的訊息處理流程是不可能的任務，若希望能有徹底認識，請閱讀錢達智先生的Delphi學習筆記Win32 基礎篇。

TApplication 類別的任務接下，處理即時遊戲中所有的行進、重繪及訊息處理等工作。

遊戲主迴圈

GameLoop 函式中，有一個 *while* 迴圈，我稱之爲「遊戲主迴圈」，遊戲開始後、結束前，執行緒會一直處於此 *while* 迴圈內執行，不曾離開。事實上，它就是接替 *TApplication* 工作的新訊息迴圈，負責視窗訊息的分派處理。不過不只這些，它還有額外的任務：

- 若訊息佇列內尚有訊息，則取出訊息來處理。
- 若訊息佇列內沒有訊息，且遊戲視窗保有輸入焦點，則讓所有角色進行一動，並且重繪畫面。
- 若遊戲視窗未取得輸入焦點，則呼叫 *WaitMessage* API 函式將控制權轉讓系統中其它執行緒（反正沒事幹），直到有新訊息進來爲止。

TMainForm::*GameLoop* 函式程式碼列表如下：

```
#0001  void __fastcall TMainForm::GameLoop()
#0002  {
#0003    InitLevel(); // 初始化
#0004
#0005    TMsg Msg;
#0006    int iStopTime;
#0007    TSprite* Sprite;
#0008    TMap& Map = TMap::Instance();
#0009
#0010    // 遊戲迴圈
#0011    while (FGameStatus == gsPlaying) {
#0012      if (PeekMessage(&Msg, 0, 0, 0, PM_REMOVE)) { // 取得訊息
#0013        switch (Msg.message) {
#0014          case WM_QUIT: // 程式結束, 離開遊戲迴圈
#0015            FGameStatus = gsTerminate;
#0016            goto OutGameLoop;
#0017
#0018          case WM_DESTROY_OBJECT: // 要求釋放某物件
#0019            Sprite = (TSprite*)Msg.wParam;
#0020
#0021            if (Sprite == FTank) { // 若死掉的是主角
#0022              FTank->Visible = false;
```

```
#0023            FGameStatus = gsOver; // Game over 囉 ~~
#0024            goto OutGameLoop; // 離開遊戲迴圈
#0025          }
#0026
#0027          // 坦克摧毀的話要做特別處理
#0028          if (dynamic_cast<TTank*>(Sprite)) {
#0029
#0030            // 若被摧毀的是敵方坦克, 則 Msg.LPARAM 代表其分數
#0031            FScore += Msg.lParam;
#0032
#0033            // 將坦克與與其有關係的子彈解除關係
#0034            FreeBulletsForTank(dynamic_cast<TTank*>(Sprite));
#0035
#0036            // 若敵方坦克全部出現且死光光了, 則到下一關去
#0037            if (FETankUsed == MAX_ETANK_PER_SCENARIO &&
#0038              Tanks()->size() == 1) {
#0039              // 這裡應該加一些過場畫面...
#0040              LevelNo = LevelNo + 1;
#0041              InitLevel();
#0042            }
#0043          }
#0044
#0045          // 釋放物件
#0046          delete Sprite;
#0047
#0048         break;
#0049
#0050        case WM_GAMEOVER: // 軍旗被打掉了, 遊戲結束
#0051          FGameStatus = gsOver;
#0052          goto OutGameLoop; // 離開遊戲迴圈
#0053
#0054        case WM_SPECIAL_CONDITION: // 進入特殊狀態
#0055          iStopTime = 10; // 特殊狀態預設有效時間十秒鐘
#0056
#0057          switch (Msg.wParam) {
#0058            // 暫停敵人行動
#0059            case TIMER_ID_GEM_CLOCK: FEnemyStopped = true;
#0060
#0061            break;
#0062
#0063            // 無敵
#0064            case TIMER_ID_GEM_HAT: FTank->SuperMode = true;
#0065            break;
#0066
#0067            case TIMER_ID_GEM_ARROW:
#0068              // 在軍旗周圍擺上打不破的鐵牆, 且修復它
```

```
#0069              Map.GetCell(LAYER_TERRITEM, 5, 11).TileNo = 26;
#0070              Map.GetCell(LAYER_TERRITEM, 5, 11).DisposeBreakMap();
#0071              Map.GetCell(LAYER_TERRITEM, 6, 11).TileNo = 26;
#0072              Map.GetCell(LAYER_TERRITEM, 6, 11).DisposeBreakMap();
#0073              Map.GetCell(LAYER_TERRITEM, 7, 11).TileNo = 26;
#0074              Map.GetCell(LAYER_TERRITEM, 7, 11).DisposeBreakMap();
#0075              Map.GetCell(LAYER_TERRITEM, 5, 12).TileNo = 26;
#0076              Map.GetCell(LAYER_TERRITEM, 5, 12).DisposeBreakMap();
#0077              Map.GetCell(LAYER_TERRITEM, 7, 12).TileNo = 26;
#0078              Map.GetCell(LAYER_TERRITEM, 7, 12).DisposeBreakMap();
#0079
#0080              iStopTime = 20; // 鐵牆持續二十秒
#0081            break;
#0082           }
#0083           // 設定計時器
#0084           SetTimer(Handle, Msg.wParam, iStopTime * 1000, NULL);
#0085         break;
#0086
#0087         case WM_TIMER: // 計時器時間到 (特殊狀態時間到)
#0088           KillTimer(Handle, Msg.wParam); // 取消計時器
#0089
#0090           switch (Msg.wParam) {
#0091             case TIMER_ID_GEM: // 寶物擺夠久了, 還不吃, 拿掉
#0092               if (Gem) delete Gem;
#0093               break;
#0094
#0095             case TIMER_ID_GEM_CLOCK: FEnemyStopped = false;
#0096             break;
#0097
#0098             case TIMER_ID_GEM_HAT:
#0099               FTank->SuperMode = mnuSuperMode->Checked;
#0100               break;
#0101
#0102             case TIMER_ID_GEM_ARROW: // 在軍旗周圍擺回磚牆
#0103               Map.GetCell(LAYER_TERRITEM, 5, 11).TileNo = 10;
#0104               Map.GetCell(LAYER_TERRITEM, 6, 11).TileNo = 10;
#0105               Map.GetCell(LAYER_TERRITEM, 7, 11).TileNo = 10;
#0106               Map.GetCell(LAYER_TERRITEM, 5, 12).TileNo = 10;
#0107               Map.GetCell(LAYER_TERRITEM, 7, 12).TileNo = 10;
#0108             break;
#0109           }
#0110          break;
#0111
#0112         case WM_INIT_LEVEL:
#0113           InitLevel(); // 關卡重新開始
#0114           break;
```

```
#0115       }
#0116
#0117       // 正常的訊息處理程序
#0118       TranslateMessage(&Msg);
#0119       DispatchMessage(&Msg);
#0120     } else if (Focused()) { // 若視窗擁有輸入焦點才動作
#0121       // 若此關卡及目前敵方坦克都沒達到上限,
#0122       // 則按照亂數"可能"出現敵方坦克
#0123
#0124       if (FETankUsed < MAX_ETANK_PER_SCENARIO &&
#0125         (int)Tanks()->size() < MAX_TANK_ON_SCREEN &&
#0126         random(100) < 100 * PROBAB_ETANK_BORN)
#0127         CreateETank(random(5) + 1); // 五種坦克任選一種
#0128
#0129       // 若目前沒有寶物, 則按照亂數"可能"出現寶物
#0130       if (!Gem && random(100) < 100 * PROBAB_GEM_BORN)
#0131         CreateGem(random(6)); // 六種寶物任選一種
#0132
#0133       // 移動我方坦克及／或敵方坦克
#0134       if (FEnemyStopped)
#0135         FTank->Move();
#0136       else
#0137         MoveTanks();
#0138
#0139       // 移動子彈
#0140       MoveBullets();
#0141
#0142       // 更新畫面
#0143       UpdateView();
#0144     } else {
#0145       // 若視窗沒有取得輸入焦點, 則將控制權交給其它執行緒,
#0146       // 直到有訊息進來
#0147       WaitMessage();
#0148     }
#0149   }
#0150
#0151   OutGameLoop: ;
#0152 }
```

0012 列呼叫的 *PeekMessage* API 函式，配合 *PM_REMOVE* 參數，效果與平日常用的 *GetMessage* API 函式幾乎一樣。唯一的差別是，*PeekMessage* 的傳回值爲布林值，表示是否取得訊息，若沒有訊息等待，函式立即返回；而 *GetMessage* 的傳回值代表取得的訊息是否爲 *WM_QUIT*，函式會等待取得訊息後才返回。這就是遊戲迴圈中，一邊能讓程式正常運作，一面能保持遊戲即時性的關鍵：**「不斷呼叫 *PeekMessage* 函式，若有訊息，**

則處理訊息；若沒訊息，則進行遊戲狀態的更新」。

不過，當遊戲視窗沒有取得輸入焦點時，遊戲狀態就不再需要不斷更新。一方面玩家可能只是想切換到 BBS 連線程式回個熱訊，沒想到切換回遊戲視窗時，才發現主角坦克已經被轟爛了，人家打到三十關了誰賠他呀？所以必須防止此情況的發生。另一個理由是，不讓遊戲繼續佔用大量 CPU 時間，遊戲狀態不斷更新會耗掉大量 CPU 時間，所以必須在視窗未取得輸入焦點時暫停動作。因此 0120 列會先呼叫 *Focused*，得知目前是否擁有輸入焦點，若有的話，更新遊戲狀態；沒有的話，則呼叫 *WaitMessage* API 函式。此函式會將控制權交給其它執行緒使用，若此次執行周期尚未結束前又有新的視窗訊息進入，控制權會自動再交還我們。

視窗訊息的處理

0013 ~ 0115 列是視窗訊息的處理邏輯，一一處理感興趣的訊息：

WM_QUIT 訊息

將 *FGameStatus* 設爲 *gsTerminate*，程式將跳離遊戲迴圈，並於離開 *GameLoop* 函式後立即結束程式。

WM_DESTROY_OBJECT 訊息

這是由 *TSprite* 物件的 *PostToDie* 函式所丟出的訊息，表示此角色「申請」自毀。訊息裡的 *wParam* 參數事實上是指向申請的物件本身的指標，*lParam* 參數則記錄玩者所得的分數。若 *wParam* 參數指向主角坦克 *FTank*，表示主角被幹掉了，則將 *FGameStatus* 設爲 *gsOver*，離開遊戲迴圈。

在此將 *wParam* 參數轉型爲 *TSprite*，以 *delete* 保留字摧毀此物件。由於 *wParam* 參數必定指向遊戲中的某個 *TSprite* 物件，在多型機制的輔助下，物件摧毀時一定會呼叫到正確的物件解構函式。

最後檢查是否所有該出現的敵方坦克已全部被摧毀，若是的話，遞增關卡編號，並將遊

戲狀態初始化，直接進入下一關繼續遊戲。理論上，關卡與關卡之間會有一些過場畫面，如上一關的得分、下一關的任務說明等等，請原諒我的偷懶。

WM_GAMEOVER 訊息

表示軍旗被打爛了，將 *FGameStatus* 設爲 *gsOver*，離開遊戲迴圈。

WM_SPECIAL_CONDITION 訊息

這是我方坦克吃到寶物時對於特殊效果的請求，由 *TMyTank::CheckCollisions* 函式發出。訊息裡的 *wParam* 參數爲特殊效果代碼，遊戲迴圈必須根據代碼提供正確的服務：

- *TIMER_ID_GEM_CLOCK* 暫停敵人行動
- *TIMER_ID_GEM_HAT* 將我方坦克加上防護罩，變成無敵狀態
- *TIMER_ID_GEM_ARROW* 在軍旗周圍擺上打不破的鐵牆

最後的步驟是設定計時器，讓特殊效果只持續一段時間後即恢復。

WM_TIMER 訊息

當我們設定的計時器時間到時便會收到此訊息，此時再根據計時器代碼來進行適當的動作，我們也利用它來進行特殊效果的還原：

- *TIMER_ID_GEM* 表示寶物擺夠久了，主角還不吃，拿掉
- *TIMER_ID_GEM_CLOCK* 恢復敵人行動
- *TIMER_ID_GEM_HAT* 將我方坦克恢復原來非無敵狀態
- *TIMER_ID_GEM_ARROW* 在軍旗周圍擺回磚牆

WM_INITLEVEL 訊息

將遊戲狀態重設爲目前關卡的初始狀態。

最後，對於其它尚未處理的訊息，一律採用最標準的訊息處理方式，呼叫 *TranslateMessage* API 函式剖析鍵盤訊息以及呼叫 *DispatchMessage* API 函式將訊息分派給適當的視窗程序。

例行的遊戲狀態更新

但如果 *PeekMessage* 未取得任何視窗訊息呢？0121 ~ 0143 列就進行例行的遊戲狀態更新動作：

1. 若此關卡及目前敵方坦克都沒達到上限，則按照亂數「可能」出現敵方坦克。
2. 若目前沒有寶物，則按照亂數「可能」出現寶物。
3. 移動我方坦克及／或敵方坦克。
4. 移動子彈。
5. 更新畫面。

如此一來，遊戲就會即時不斷地更新狀態，你可以看到敵方坦克不停地走動，子彈不斷地發射，不停地飛行、碰撞、爆炸，一幕幕活生生的坦克大戰景象不停地在遊戲視窗中上演著。

上面程式所呼叫的 *CreateETank* 及 *CreateGem* 函式分別依參數產生對應的敵方坦克及寶物。建立寶物時，必須同時呼叫 *SetTimer* API 函式來設定計時器，以使寶物能自動在三十秒後消失。兩個函式的程式碼列表如下：

```
#0001  // 產生敵方坦克
#0002  void __fastcall TMainForm::CreateETank(int Kind)
#0003  {
#0004    FETankUsed++; // 遞增此關卡已產生的敵方坦克
#0005
#0006    TETank* T;
#0007    T = new TETank(Kind);
#0008    T->LoadBits();
#0009  }
#0010
#0011  // 產生寶物
#0012  void __fastcall TMainForm::CreateGem(int Kind)
#0013  {
#0014    Gem = new TGem(TGemKind(Kind));
#0015    Gem->LoadBits();
#0016    Gem->RandomPosition(); // 隨意擺置
```

```
#0017    // 寶物出現 30 秒後自動消失
#0018    SetTimer(Handle, TIMER_ID_GEM, 30 * 1000, NULL);
#0019  }
```

處理使用者輸入

最後的最後，讓我們來畫龍點睛，加上使用者控制部分。先將 *TMainForm* 的 *KeyPreview* 屬性設爲 *true*，讓它無時無刻都能優先收到鍵盤訊息。再分別撰寫它的 *OnKeyDown* 及 *OnKeyUp* 事件處理函式：

```
#0001  void __fastcall TMainForm::FormKeyDown(TObject *Sender, WORD &Key,
#0002       TShiftState Shift)
#0003  {
#0004    // 注意, 只有遊戲中狀態, 鍵盤控制才有效
#0005    if (FGameStatus == gsPlaying) {
#0006      switch (Key) {
#0007        case VK_UP: case VK_DOWN: case VK_LEFT: case VK_RIGHT:
#0008          switch (Key) {
#0009            case VK_UP: FTank->Direction = drUp; break; // 向上走
#0010
#0011            case VK_DOWN: FTank->Direction = drDown; break; // 向下走
#0012
#0013            case VK_LEFT: FTank->Direction = drLeft; break; // 向左走
#0014
#0015            case VK_RIGHT: FTank->Direction = drRight; break;//向右走
#0016
#0017          }
#0018
#0019          FTank->Active = true; // 主角開始"動"
#0020          break;
#0021
#0022        case VK_SPACE:
#0023          FTank->FireBullet(); // 發射子彈, 咻 ~~
#0024        break;
#0025      }
#0026    }
#0027  }
#0028
#0029  void __fastcall TMainForm::FormKeyUp(TObject *Sender, WORD &Key,
#0030       TShiftState Shift)
#0031  {
#0032    // 注意, 只有遊戲中狀態, 鍵盤控制才有效
#0033    if (FGameStatus == gsPlaying)
```

```
#0034     switch (Key) { // 放開鍵盤, 我方坦克就停止動作
#0035       case VK_UP: case VK_DOWN: case VK_LEFT: case VK_RIGHT:
#0036       FTank->Active = false;
#0037       break;
#0038     }
#0039  }
```

與足球番的使用者輸入最大的不同是，這兒希望做到讓我方坦克連續行走的效果。也就是，若按下任一方向鍵，坦克開始移動後，只要在該鍵放開前，坦克都會不斷地移動，即使另外按下空白鍵發射子彈也一樣。

因此我們不再採用，每收到 *WM_KEYDOWN* 視窗訊息，就更新主角座標一次的方式；而是收到 *WM_KEYDOWN* 視窗訊息時，讓 *FTank->Moving* 爲 *true*，而收到 *WM_KEYUP* 時，讓 *FTank->Moving* 爲 *false*，讓遊戲主迴圈時時更新坦克的位置，如此便可達成上述效果。

熬呀熬出頭

終於，好不容易，千辛萬苦，披荊斬棘，排除萬難，我們來到最令人興奮的一刻！儲存整個專案，按下【F9】執行鍵，讓下面幾張圖來道感言吧。

圖 9-19 / 歡迎畫面

圖 9-20 / 第一關遊戲畫面

圖 9-21 / 第二關遊戲畫面

圖 9-22 / 軍旗被烤焦，遊戲結束畫面

第一關遊戲畫面中，你可以看到我新增的第五種坦克－一隻狀似蝴蝶的飛行器在天上慢慢移動，不斷轟炸地面的可怕景象。第二關遊戲畫面中，兩隻可愛的草怪用磚牆圍住，一輛敵方坦克正穿過樹棚自裡頭跑出來，而主角的子彈正越過海洋快要偷襲到它呢。

而任務失敗的畫面中，我方坦克跑到畫面中央想將剛出現的大蝴蝶轟掉，太興奮了，沒注意到下方什麼時候偷跑來兩輛敵方坦克，狠心將小鳥軍旗烤焦，唉。

雖然速度慢了點，聲音單調了點（由於聲道的限制，目前只播放大爆炸音效），平衡度差了點（所有的坦克參數皆未經過精心考慮，隨手設定的結果），不過還是很好玩，好像重回任天堂版坦克大決戰時代。

坦克大決戰實作，成功！