GOTOP

陳寬達の

C++Builder 深度歷險

CD-ROM Included

碁峰
www.gotop.com.tw

陳昇瑋
（原名：陳寬達）

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

第八章

足球番

許多人應該都很熟悉「倉庫番」這個小遊戲，

在各個平臺及各式電視遊樂器上皆曾見它的蹤跡。

我的回味方式比較奇特，不是好好地玩玩它，

而是以 C++Builder 從頭到尾撰寫一套足球版的「足球番」。

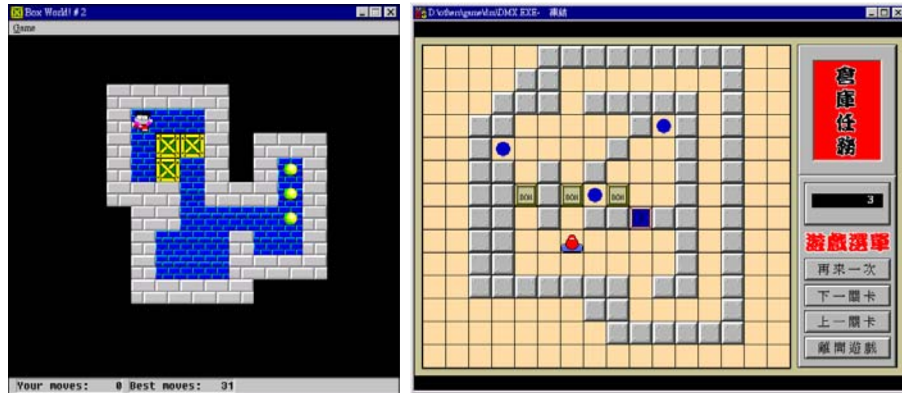

圖 8-1 / 兩個從 Internet 上取得的倉庫番遊戲

上圖是兩個從 Internet 上取得的倉庫番遊戲，皆是國人的作品。前者似乎以純 Windows SDK 開發，後者以 QBasic 4.5 英文版撰寫。

你一定也很熟悉這個小遊戲吧！在各個平臺及各式電視遊樂器上皆曾見它的蹤跡。規則十分簡單，只要操縱主角將箱子一一推至地圖上的標示點，就算過關。雖然規則就這麼簡單，但難度可不低唷，在細心安排設計下，有些關卡總要讓人傷透腦筋，嘗試個十來次甚至百來次才能過關[1]。

這個再簡單也不過的老遊戲，正是今天的主題。我們將以 C++Builder 從頭到尾撰寫一套足球版的「倉庫番」，就稱它為「足球番」好了。它的特性十分符合本章的教學目的：

1. 畫面處理 GDI 即可輕鬆解決，背景不用捲動，也不算即時遊戲。
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，不是只靠華麗畫面吸引人的遊戲類型。

先來訂立我們這套足球番的功能及特色：

---

[1] 好吧，我承認，上頭那兩個遊戲我就有些關卡過不了。:P

1. 遊戲規則與倉庫番皆相同，將所有目的地形利用覆蓋用物品掩住即可過關。
2. 地圖大小即是可視範圍，因此不用支援地圖捲動。
3. 所有圖片，包括角色圖形尺寸大小皆相同。
4. 角色的移動以圖片大小爲單位，因此沒有小碎步移動所帶來的碰撞處理問題。
5. 所有圖片，包括角色圖形皆是外掛方式，可以在不修改程式碼的情形下更動圖形。
6. 採用關卡制度，可供使用者自行編輯關卡。
7. 具有動作重播功能。

看起來，除了有些新鮮有點無聊的重播功能外，只不過是另一套簡單的倉庫番類型遊戲，連圖形、角色移動等方式都還一切從簡哩。希望你不會太失望，功夫從易處練，雖然一點都不炫，至少是自己寫出來的嘛。

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。因爲重覆性高，有些遊戲將地圖編輯器及主程式合併放在同一支程式內，但是，分開爲兩支程式有下列優點：

- 減少程式設計的複雜度。同一支程式提供的功能越多，程式邏輯必定越複雜。
- 不必要的 overhead。地圖編輯功能不一定會提供給 end user，若將這些功能置於主程式中而不去使用，徒然增加檔案大小而已。

好，那表示我們要分別撰寫三支應用程式。在動手之前，別急，讓我們先將系統必要的幾個重要類別切割出來。

TTiles 類別

用來定義一個「圖庫」（Tile Archive），一個圖庫包含多張圖片，畫面上除了角色圖形外的所有圖片都是由圖庫中取得。

因為提供給「地形」及「物品」兩層地圖的圖片屬性完全不同，因此必須為這兩個圖庫分別設計不同的圖庫類別，所以我將 *TTiles* 設定為抽象類別，但做好大部分的工作，如載入／儲存圖庫，圖片管理及繪製圖片等等，再分別由 *TTerrTiles* 及 *TItemTiles* 兩個類別繼承，分別提供設定及讀取圖片屬性的功能。

- 「地形」圖片有「可以通過」及「目的地形」兩種屬性。
- 「物品」圖片有「可以移動」及「覆蓋用物品」兩種屬性。

TMap 類別

用來定義一張地圖，或說，一個關卡的佈局。本遊戲的地圖設計為兩層，底下一層是地形，上面一層是物品，兩層獨立操作／貼圖互不相干，貼圖用的圖片分別由兩個圖庫提供，因此分別需要一個二維陣列來儲存圖片編號。

為求視覺逼真效果及操作編輯方便，現在的地圖往往都不只分為兩層，如 StarCraft 就分為五層（請見圖 8-2），英雄無敵 III 也至少分為四層（請見圖 8-3），分別是土地、河流、道路及地形物。我們的足球番由於地圖簡單，因此分為地形及物品兩層即足夠。

TMap 類別不但負責關卡的載入及儲存，另外也會儲存角色的初始位置。

圖 8-2 / StarCraft 的地圖編輯器，它有五層地形可供編輯。

圖 8-3 / 英雄無敵 III 的地圖編輯器，它有四層地形可供編輯。

TRole 類別

角色類別，負責角色的圖形載入及繪製，移動本身的位置計算及搬動物品。另外重播功也由此類別自行紀錄移動資訊，再交由主程式來播放。

總共才需五個類別，而且除了 *TTiles* 類別是抽象類別，不用來產生物件，其它四個類別在遊戲中都只要產生一個物件就夠了，爲什麼？因爲我們同一時間只可能使用一張地圖，一個「地形」圖庫，一個「物件」圖庫以及一個角色。

類別實作

先將類別的功能及需求定義出來，切割清楚後，一一將這些重要類別實作出來，到時候用「兜」的，便可輕易兜出三個程式來了，信不信？:）

首先得先定義一些到處用得著的常數（定義於 Util.pas）：

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

#define TILE_NUM_X 10                // 畫面橫軸格數
#define TILE_NUM_Y 10                // 畫面縱軸格數

const char* FN_TERR_ARCHIVE "TERR.TIA"  // 地形圖庫
const char* FN_ITEM_ARCHIVE "ITEM.TIA"  // 物品圖庫
const char* FN_ROLEBITS    "ROLEBITS.BMP" // 角色圖檔

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

圖片大小爲 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是隨手定義的，在我的 1280 x 1024 解析度畫面下看起來小小的，但

後來才發現 10 x 10 格設計不出複雜的關卡。:P 反正只要將這兒的常數更動，重新編譯主程式及地圖編輯器後，即可以新的畫面大小進行關卡設計及遊戲，覺得地圖範圍過小的讀者請自己修改。:p

SIG_MYFILE 用來作爲檔頭標籤，在讀取圖庫及地圖檔案時，先確認檔案開頭有沒有此字串，以確定讀取的是我們自己的檔案，不會有誤讀的情況發生。此外，我還在圖庫的檔頭標籤後頭加上「副檔頭標籤」，用以辨別、確認「地形」及「物品」圖庫。

TTiles 圖庫類別及子類別

TTiles 是第一個實作的類別，這是它的類別宣告（定義於 TileUnit.h）：

```
#0001  class TTiles {
#0002  private:
#0003    int FTileNum; // 圖片數量
#0004    Graphics::TBitmap* FBits; // 存放圖庫所有圖片的 bitmap
#0005
#0006    int GetTilePos_Left(int index);
#0007    int GetTilePos_Top(int index);
#0008
#0009    int GetTileNumPerRow();
#0010    int GetTileRowCount();
#0011  protected:
#0012    virtual void SetTileNum(int Value);
#0013
#0014    // 讀取及設定屬性都是虛擬函式，讓後代類別來改寫
#0015    virtual void LoadAttrs(TReader* filer) = 0;
#0016    virtual void WriteAttrs(TWriter* filer) = 0;
#0017  public:
#0018    TTiles();
#0019    virtual ~TTiles();
#0020
#0021    // 載入及儲存圖庫
#0022    void LoadFromFile(AnsiString Filename);
#0023    void SaveToFile(AnsiString Filename);
#0024
#0025    void ImportPicture(AnsiString Filename); // 匯入圖形檔
#0026
#0027    __property int TileNum = {read = FTileNum, write = SetTileNum};
#0028
```

```
#0029    // 每列的圖片數目
#0030    __property int TileNumPerRow = {read = GetTileNumPerRow};
#0031    // 共有幾列圖片
#0032    __property int TileRowCount = {read = GetTileRowCount};
#0033
#0034    __property Graphics::TBitmap* Bitmap = {read = FBits};
#0035
#0036    // 根據圖片編號就可取得圖片在圖形中的位置
#0037    __property int TilePos_Left[int index] = {read = GetTilePos_Left};
#0038    __property int TilePos_Top[int index] = {read = GetTilePos_Top};
#0039  };
#0040
#0041  const int TILE_BITS_NUM_X = 10; // 圖庫中每列圖片的數目
```

取巧的圖片儲存方式

圖庫的實作採用了一個很偷懶很取巧的函式：將所有的圖片擺在同一個 bitmap 中，同時定義了一個 *TILE_BITS_NUM_X* 常數，表示圖庫中每列圖片的數目。我將此常數設定爲 10，表示圖庫 bitmap 爲寬 = 320 點，高 = 圖片數目 / 10 x 32，所以編號爲 0 .. 9 的圖片會依序擺在第一列、編號爲 10 .. 19 的圖片會擺在第二列等等。

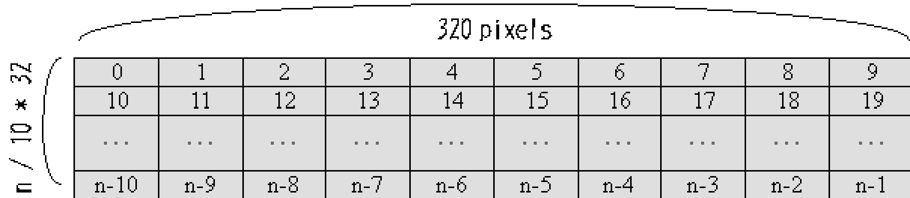

圖 8-4 / 圖庫中圖片的儲存方式

這種偷懶的圖片儲存函式讓設計者在畫好圖片後必須開啓影像編輯軟體，手動將圖片擺在正確的位置上，麻煩極了。例如，我畫好 No.13 圖片，這時必須打開原本儲存圖庫的圖形檔，將它擺在（96, 32）的位置上，差一點都不行。呵，我知道你們都會罵我很笨，饒了我吧，下不爲例，爲了程式簡單，我只好出此下策，下回我一定將它改成比較聰明的儲存方式。

爲了這種圖片儲存法，*TTiles* 類別提供 *TilePos_Left* 及 *TilePos_Top* 兩個陣列型態屬性，來讓外界依圖片編號就可取得圖片在圖檔中的 X 軸及 Y 軸位置。這兩個屬性皆屬於一維陣列屬性，使用時必須傳入索引值，而它們對應的 *GetTilePos_Left* 及 *GetTilePos_Top* 兩個函式才會根據此索引值去計算圖片位置。

另外圖片數目也一定爲 *TILE_BITS_NUM_X* 的倍數，是呼叫 *ImportPicture* 函式後根據此圖形檔的長寬大小自動計算而得，*ImportPicture* 函式同時會裁切載入的圖形檔，使圖形檔的寬度變成 *TILE_WIDTH * TILE_BITS_NUM_X*，高度變成 *TILE_HEIGHT* 的整數倍：

```
#0001  void TTiles::ImportPicture(AnsiString Filename)
#0002  {
#0003    FBits->LoadFromFile(Filename);
#0004    TileNum = FBits->Height / TILE_HEIGHT * TILE_BITS_NUM_X;
#0005
#0006    // 將 bitmap 裁減爲所需的大小
#0007    FBits->Width = TILE_WIDTH * TILE_BITS_NUM_X;
#0008    FBits->Height = TileNum / TILE_BITS_NUM_X * TILE_HEIGHT;
#0009  }
```

載入圖檔的 *TTiles.LoadFromFile* 函式使用 VCL 的 *TFileStream* 及 *TReader* 類別來開啓檔案及讀取資料，0018 行還呼叫了 *LoadAttrs* 函式供後代類別 *TTerrTiles* 及 *TItemTiles* 來改寫，讀取它們自己的屬性資料：

```
#0001  void TTiles::LoadFromFile(AnsiString Filename)
#0002  {
#0003    TFileStream* fs;
#0004    TReader* reader;
#0005
#0006    fs = new TFileStream(Filename, fmOpenRead);
#0007    reader = new TReader(fs, 2048);
#0008    try {
#0009      CheckSignature(reader, SIG_MYFILE); // 檢查檔頭標籤
#0010      CheckSignature(reader, typeid(*this).name()); // 檢查副檔頭標籤
#0011
#0012      // 爲了觸發 property 的 SetXXX method
#0013      TileNum = reader->ReadInteger();
#0014      reader->FlushBuffer();
#0015
#0016      // 這是虛擬函式, 因爲 TTiles 本身根本沒有定義屬性,
#0017      // 讓後代類別決定該如何讀取屬性
#0018      LoadAttrs(reader);
```

```
#0019
#0020     FBits->LoadFromStream(fs); // 讀入存放所有圖片的圖形
#0021
#0022     // 檢查圖形的長寬符不符合標準
#0023     if (FBits->Width < TILE_WIDTH * TILE_BITS_NUM_X)
#0024       throw Exception("Width of tile bitmap is invalid");
#0025
#0026     if (FBits->Height / TILE_HEIGHT * TILE_BITS_NUM_X < FTileNum)
#0027       throw Exception("Height of tile bitmap is invalid");
#0028   } __finally {
#0029     delete reader;
#0030     delete fs;
#0031   }
#0032 }
```

你可以在類別宣告中注意到，*LoadAttrs* 及 *WriteAttrs* 兩個函式不但爲虛擬函式，同時還宣告爲抽象函式，這是因爲 *TTiles* 類別根本沒有圖片屬性的概念，只是宣告好這兩個函式，完全不實作，留待後代改寫使用。

地形及物品圖庫類別

TTiles 類別撰寫完成後，接著就可以從它衍生出談論已久，呼之欲出的 *TTerrTiles* 及 *TItemTiles* 類別了（與 *TTiles* 類別同樣定義於 TileUnit.h）。

```
#0001 // 地形的屬性
#0002 typedef enum {taCanPass, taTarget} TTerrAttrElement;
#0003 typedef Set<TTerrAttrElement, taCanPass, taTarget> TTerrAttr;
#0004
#0005 // 物品的屬性
#0006 typedef enum {iaCanMove, iaSource} TItemAttrElement;
#0007 typedef Set<TItemAttrElement, iaCanMove, iaSource> TItemAttr;
#0008
#0009 class TTerrTiles : public TTiles {
#0010 private:
#0011   std::vector<TTerrAttr> FAttrs;
#0012   TTerrAttr GetAttrs(int index);
#0013   void SetAttrs(int index, const TTerrAttr& Value);
#0014 protected:
#0015   virtual void SetTileNum(int Value);
#0016
#0017   virtual void LoadAttrs(TReader* filer);
```

```
#0018   virtual void WriteAttrs(TWriter* filer);
#0019  public:
#0020   __property TTerrAttr Attrs[int index] =
#0021     {read = GetAttrs, write = SetAttrs};
#0022  };
#0023
#0024  class TItemTiles : public TTiles {
#0025  private:
#0026   std::vector<TItemAttr> FAttrs;
#0027   TItemAttr GetAttrs(int index);
#0028   void SetAttrs(int index, const TItemAttr& Value);
#0029  protected:
#0030   virtual void SetTileNum(int Value);
#0031
#0032   virtual void LoadAttrs(TReader* filer);
#0033   virtual void WriteAttrs(TWriter* filer);
#0034  public:
#0035   __property TItemAttr Attrs[int index] =
#0036     {read = GetAttrs, write = SetAttrs};
#0037  };
#0038
#0039  TTerrTiles Terrs;
#0040  TItemTiles Items;
```

第 0003、0007 行是這兩個新類別的重點，兩個圖庫類別分別擁有不同的圖片屬性。0011 及 0026 分別宣告兩個類別用來儲存屬性的容器物件，在此我使用 C++ Standard Library 所提供的 *vector* 類別。

兩個類別都改寫了 *TTiles* 類別的 *SetTileNum* 及 *LoadAttrs*、*WriteAttrs* 函式，目的十分簡單，改寫 *SetTileNum* 函式是爲了在圖片數目改變時同時變更 *FAttrs* 屬性陣列的長度，而 *LoadAttrs*／*WriteAttrs* 則經由 *TFileStream* 物件來讀取及寫入屬性陣列。以 *TTerrTiles* 類別的程式碼爲例：

```
#0001  void TTerrTiles::SetTileNum(int Value)
#0002  {
#0003   TTiles::SetTileNum(Value);
#0004   FAttrs.resize(TileNum);
#0005  }
#0006
#0007  void TTerrTiles::LoadAttrs(TReader* filer)
#0008  {
#0009   for (int i = 0; i < TileNum; i++)
#0010     for (TTerrAttrElement x = taCanPass; x <= taTarget;
```

```
#0011         x = (TTerrAttrElement)(x + 1)) {
#0012       bool b = filer->ReadBoolean();
#0013       if (b) FAttrs[i] = FAttrs[i] << x;
#0014     }
#0015
#0016   filer->FlushBuffer();
#0017 }
#0018
#0019 void TTerrTiles::WriteAttrs(TWriter* filer)
#0020 {
#0021   for (int i = 0; i < TileNum; i++)
#0022     for (TTerrAttrElement x = taCanPass; x <= taTarget;
#0023         x = (TTerrAttrElement)(x + 1))
#0024       filer->WriteBoolean(FAttrs[i].Contains(x));
#0025
#0026   filer->FlushBuffer();
#0027 }
```

儲存屬性的步驟有些麻煩：尋訪每個圖片的屬性，將 *TTerrAttrElement* 集合型態的值儲存起來，但因為 *TTerrAttrElement* 類別本身未提供輸出入介面，也未將內部的資料佈局公開給外界存取（這是當然的），所以我們只能以一個個可能的值來詢問，此集合是否包括某某值，如果是的話，就寫入 *true*，否則寫入 *false*。讀取屬性的步驟當然也一樣囉，讀入一個個布林值，根據它們的真偽值來還原圖片的屬性。

0039 及 0040 列分別為 *TTerrTiles* 及 *TItemTiles* 類別各宣告一個物件，因為它們只使用一個物件便已足夠，因此在此宣告，在三支程式中使用時就不用分別再為它們宣告了。

好了，完成了三個類別，它們皆置於 TileUnit 單元。

TMap 地圖類別

嘿，接下來輪到 *TMap* 類別了（好像在宰羊圈裏的待宰綿羊似的:p），*TMap* 類別定義於 MapUnit.h：

```
#0001 typedef Byte TMapArray[TILE_NUM_Y][TILE_NUM_X];
#0002
#0003 class TMap {
#0004 private:
```

```
#0005   TMapArray FTerrMap, FItemMap; // 地形及物品地圖
#0006
#0007   int FLevelNo; // 目前載入的關卡編號
#0008   Graphics::TBitmap* FInvBitmap; // 用於物品的透明貼圖
#0009
#0010   int FRole_X, FRole_Y; // 角色的起始位置
#0011
#0012   void SetLevelNo(int Value);
#0013
#0014   bool GetCanPass(int x, int y);
#0015   bool GetIsTarget(int x, int y);
#0016
#0017   bool GetCanMove(int x, int y);
#0018   bool GetIsSource(int x, int y);
#0019
#0020   TTerrAttr GetTerrAttr(int x, int y);
#0021   void SetTerrAttr(int x, int y, TTerrAttr Value);
#0022   TItemAttr GetItemAttr(int x, int y);
#0023   void SetItemAttr(int x, int y, TItemAttr Value);
#0024
#0025   Byte GetTerrMap(int x, int y);
#0026   Byte GetItemMap(int x, int y);
#0027   void SetTerrMap(int x, int y, Byte Value);
#0028   void SetItemMap(int x, int y, Byte Value);
#0029   void SetRole_X(int Value);
#0030   void SetRole_Y(int Value);
#0031 protected:
#0032 public:
#0033   TMap();
#0034   ~TMap();
#0035
#0036   AnsiString GetFileName(); // 根據關卡編號，傳回對應的檔名
#0037
#0038   void LoadFromFile();
#0039   void SaveToFile();
#0040
#0041   void DrawTerrMap(TCanvas* Canvas); // 將地形畫在 Canvas 上
#0042   void DrawItemMap(TCanvas* Canvas); // 將物品畫在 Canvas 上
#0043
#0044   // 重設整張地圖，或只重設地形或物品
#0045   void ResetMap();
#0046   void ResetTerrs();
#0047   void ResetItems();
#0048
#0049   __property int LevelNo = {read = FLevelNo, write = SetLevelNo};
#0050
```

```
#0051    // 取得某格的地形及物品圖片編號
#0052    __property Byte TerrMap[int x][int y] =
#0053     {read = GetTerrMap, write = SetTerrMap};
#0054    __property Byte ItemMap[int x][int y] =
#0055     {read = GetItemMap, write = SetItemMap};
#0056
#0057    // 取得初始的角色位置
#0058    __property int Role_X = {read = FRole_X, write = SetRole_X};
#0059    __property int Role_Y = {read = FRole_Y, write = SetRole_Y};
#0060
#0061    // 取得某格的地形屬性
#0062    __property TTerrAttr TerrAttr[int x][int y] =
#0063     {read = GetTerrAttr, write = SetTerrAttr};
#0064
#0065    __property bool CanPass[int x][int y] = {read = GetCanPass};
#0066    __property bool IsTarget[int x][int y] = {read = GetIsTarget};
#0067
#0068    // 取得某格的物品屬性
#0069    __property TItemAttr ItemAttr[int x][int y] =
#0070     {read = GetItemAttr, write = SetItemAttr};
#0071
#0072    __property bool CanMove[int x][int y] = {read = GetCanMove};
#0073    __property bool IsSource[int x][int y] = {read = GetIsSource};
#0074  };
#0075
#0076  TMap Map;
```

光是類別宣告就洋洋灑灑的七十六行，其實絕大部分是屬性宣告及對應的讀取／寫入函式，讓 *TMap* 類別使用起來更方便罷了。

首先宣告 *TMapArray* 型別，它是 *TILE_NUM_X* * *TILE_NUM_Y* 個元素的 *Byte* 陣列，*Byte* 型態的範圍為 0 ~ 255，表示圖片最多只能有 256 種，我知道這時又有人想 K 我了，不急，換個角度想，現在限制越多，表示改良空間越大，以後領到最佳進步獎的機會也越高說。不過 256 個圖片對於足球番這種小遊戲來講再怎麼說也夠多了，不夠的話隨時再將 *Byte* 改為 *Word* 或 *unsigned long*（四位元組的無號正整數）也成。

0005 列宣告 *FTerrMap* 及 *FItemMap* 兩個型態為 *TMapArray* 的地形及物品地圖。在這兒我做了特殊的設定：**「0 號地形為預設地形，而 0 號物品表示此處無物品」**。因為地形一定遍布整張地圖，但物品不是，所以一定得設定一個數字表示此處沒有東西，而 0 號是最佳選擇。

處理透明貼圖

0008 列所宣告的 *FInvBitmap* 物件是專門用來供物品圖片做透明貼圖用的。透明貼圖指的是將物品「貼」到背景上時，周圍不屬於物品本身之處，就應該讓背景顯現出來，如圖 8-5。

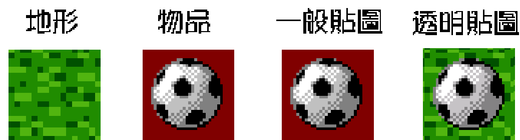

圖 8-5 / 一般貼圖及透明貼圖的比較

透明貼圖一直是 GDI 繪圖中十分傷腦筋的問題。從前 DOS 時代在 X mode 下，連貼圖動作都是自己寫的，利用兩層迴圈將像素「塞」進顯示卡的視訊記憶體，完全掌握著最低階的動作，因此達成透明貼圖的效果只是舉手之勞。而 DirectDraw 的 *ISurface* 也提供了 *SetColorKey* 等函式提供透明貼圖的解決方案。

但在 GDI 中要達成透明貼圖的效果可就麻煩了，一來我們對它的掌控能力不若 DOS 下直接填寫視訊記憶體那麼完整，二來 GDI 本身動作就慢，不容許我們花太多額外時間在處理透明貼圖。於是，最早最早，在 Windows 95 那個時代，GDI API 還沒有提供任何關於透明貼圖的解決方案前，達成透明貼圖通常必須經由下列的九大步驟：

實驗器材：

原始影像一，遮罩顏色一，遊戲畫面一。

實驗目的：

將原始影像貼到遊戲畫面上，但不貼原始影像中遮罩顏色的部分，使得那一部分仍然呈現原本的遊戲畫面。

實驗步驟：

1. 建立一個 DC 來放置原始影像，稱之爲「影像 DC」。

2. 將原始影像選擇至 DC 上。

3. 另外建立一個 memory DC 來存放最後的影像，暫且稱它為「目的 DC」。

4. 將畫面上將要貼圖的矩形區域圖形拷貝到目的 DC。

5. 建立一個「AND 遮罩」，讓原始影像所有以遮罩顏色繪製的部分通通變成透明的，建立「AND 遮罩」有下列三小步驟：

   ■ 將「影像 DC」的背景顏色設定為遮罩顏色。

   ■ 建立一個相同大小的單色 DC。

   ■ 將原始影像不管三七二十一貼到此單色 DC 上。這使得此單色 DC 變成原始影像透明貼圖用的「AND 遮罩」，此遮罩於原始影像為遮罩顏色部分呈現 1，而非遮罩顏色部分呈現 0。

6. 呼叫 *BitBlt* 函式，搭配 *SRCAND* 貼圖模式將「AND 遮罩」貼到「目的 DC」上。

7. 呼叫 *BitBlt* 函式，搭配 *SRCAND* 貼圖模式將經過反相的「AND 遮罩」貼到「影像 DC」上。

8. 呼叫 *BitBlt* 函式，搭配 *SRCPAINT* 貼圖模式將「影像 DC」貼到「目的 DC」上。

9. 最後將「影像 DC」貼到畫面上適當的位置。

呼，看得都很累了，這是 MSDN 裏建議使用的方法，當然不是唯一做法囉，只是方法大同小異，簡單不到哪去。

Tips

你可以發現這九道步驟完全沒有涉及調色盤的處理，所以這方法只適用於不使用調色盤的顯示模式，不適合 256 色或 16 色模式使用。

但是，若你的程式只想在 Windows NT 上執行，Windows NT 提供一道新的 *MaskBlt* 函式，讓我們可以很簡單地達成透明貼圖，只是這一點都不實用，有誰想要寫只能在 Windows NT 執行的遊戲呢？

雖然微軟知錯能改，亡羊補牢，早已提出 *TransparentBlt*、*AlphaBlend* 透明及半透明貼圖

等 GDI 函式，但只在 Windows 2000 才提供，遠水救不了近火，我們還是自求多福，自己動手做透明貼圖囉。

很幸運地，VCL 的 *TCanvas* 及 *TBitmap* 類別提供透明貼圖的機制，只要將 *TBitmap* 的 *Transparent* 屬性設為 *true*，將 *TransparentColor* 設為遮罩顏色，再呼叫 *TCanvas::Draw* 函式就行了：

```
#0001  void __fastcall TForm1::btnVCLClick(TObject *Sender)
#0002  {
#0003    Graphics::TBitmap* Bits;
#0004
#0005    Bits = new Graphics::TBitmap; // 建立暫時的 TBitmap
#0006    Bits->Assign(imgSrc->Picture->Bitmap); // 將原始影像拷貝過來
#0007    Bits->Transparent = True;
#0008    Bits->TransparentColor = RGB(128, 0, 0); // 設定遮罩顏色
#0009
#0010    imgDst->Canvas->Brush->Color = clBtnFace;
#0011    imgDst->Canvas->FillRect(imgDst->Canvas->ClipRect);
#0012    imgDst->Canvas->Draw(0, 0, Bits); // 複製到目的畫布上
#0013    delete Bits;
#0014  }
```

要注意的是，*TBitmap* 的這幾個 *TransparentXXXX* 屬性只針對 *TCanvas* 的 *Draw* 及 *BrushCopy* 函式有效，對於 *TCanvas* 的其它函式或 GDI 函式皆無效用，所以若你將 0012 行改成 *BitBlt* API 函式或 *TCanvas* 的 *StretchBlt* 函式時，會發現完全沒有透明貼圖的效果。你可以在 Graphics 單元中找到實作透明貼圖的 *TransparentStretchBlt* 函式，若在 Windows NT 下而且原始及目的影像大小一樣時，它就直接呼叫 *MaskBlt* GDI 函式；對於其它版本的 Windows，就仿前頭的九大步驟，先製作「AND 遮罩」，經過幾道邏輯貼圖，再加上調色盤的處理，以支援 256 色或 16 色模式下的透明貼圖效果。

Info

十分遺憾的是，*TCanvas* 的透明貼圖能力在 Windows 98 竟然出問題，無論 C++Builder 5 或新出爐的 C++Builder 6，這個問題依舊存在，Borland 真該打。

真是氣煞人也，沒關係，幸好我們有那九陽真經..哦，不，是透明貼圖九步心訣，大不了自己做一個就是，沒什麼了不起。我寫了一個小範例程式，分別使用 VCL 及 API 來達成

透明貼圖，執行結果請見下頁圖 8-6。

圖 8-6 / 分別使用 VCL 及 API 來達成透明貼圖的範例程式

好，回到正題，我們提到，*FInvBitmap* 物件是專門用來供物品圖片做透明貼圖用的，因此在 *TMap* 的建構函式中，可以看到 *FInvBitmap* 的屬性設定：

```
TMap::TMap()
{
  ...

  FInvBitmap = new Graphics::TBitmap;
  FInvBitmap->Width = TILE_WIDTH;
  FInvBitmap->Height = TILE_HEIGHT;
  FInvBitmap->Transparent = true;
  FInvBitmap->TransparentColor = (TColor)RGB(128, 0, 0);
}
```

待會便拿它來達成貼物品圖片的透明貼圖效果。而遮罩顏色 *RGB(128, 0, 0)* 是我任意選定的，但選定就不要更動，因爲遊戲中所有的物品及角色圖片都必須嚴格遵照這個遮罩顏色來繪製，若有更動便要大肆修改，十分麻煩。

關卡檔案的載入儲存

TMap 類別的載入及儲存函式會分別將兩層地圖，及初始角色位置放到檔案或從檔案讀出。這裏的改進空間極大，也可將其它與關卡相關的設定一併儲存，例如關卡描述啦、

過關提示啦、過關美女圖啦，都一起放到關卡檔案中。*TMap::LoadFromFile* 函式如下：

```
#0001  void TMap::LoadFromFile()
#0002  {
#0003    TFileStream* fs;
#0004    TReader* reader;
#0005
#0006    fs = new TFileStream(GetFileName(), fmOpenRead);
#0007    reader = new TReader(fs, 2048);
#0008    try {
#0009      CheckSignature(reader, SIG_MYFILE);
#0010      CheckSignature(reader, typeid(*this).name());
#0011
#0012      FRole_X = reader->ReadInteger();
#0013      FRole_Y = reader->ReadInteger();
#0014
#0015      reader->Read(FTerrMap, sizeof(TMapArray));
#0016      reader->Read(FItemMap, sizeof(TMapArray));
#0017    } __finally {
#0018      delete reader;
#0019      delete fs;
#0020    }
#0021  }
```

0009 及 0010 列同樣地檢查檔頭標籤及副檔頭標籤，接下來讀取角色位置，最後才是兩張固定大小的地形及物品地圖，過程平舖直述，十分簡單直覺。

TMap 類別的重頭戲是 *DrawTerrMap* 及 *DrawItemMap* 兩個分別繪製地形層及物品層畫面的函式，裏頭同樣地都是兩層迴圈，一一尋訪所有的圖格，根據那一格的地形或物品圖片編號，將圖片畫在 *Canvas* 上頭：

```
#0001  void TMap::DrawTerrMap(TCanvas* Canvas)
#0002  {
#0003    for (int y = 0; y < TILE_NUM_Y; y++)
#0004      for (int x = 0; x < TILE_NUM_X; x++)
#0005        BitBlt(Canvas->Handle, x * TILE_WIDTH, y * TILE_HEIGHT,
#0006          TILE_WIDTH, TILE_HEIGHT, Terrs.Bitmap->Canvas->Handle,
#0007          Terrs.TilePos_Left[FTerrMap[y][x]],
#0008            Terrs.TilePos_Top[FTerrMap[y][x]], SRCCOPY);
#0009  }
#0010
#0011  void TMap::DrawItemMap(TCanvas* Canvas)
#0012  {
#0013    // 統一 FInvBitmap 及 Items.Bitmap 的格式
```

```
#0014   FInvBitmap->PixelFormat = Items.Bitmap->PixelFormat;
#0015
#0016   for (int y = 0; y < TILE_NUM_Y; y++)
#0017     for (int x = 0; x < TILE_NUM_X; x++)
#0018       if (FItemMap[y][x]) {
#0019         int Left = Items.TilePos_Left[FItemMap[y][x]];
#0020         int Top = Items.TilePos_Top[FItemMap[y][x]];
#0021
#0022         FInvBitmap->Canvas->CopyRect(Rect(0, 0, TILE_WIDTH,
#0023           TILE_HEIGHT), Items.Bitmap->Canvas,
#0024           Rect(Left, Top, Left + TILE_WIDTH, Top + TILE_HEIGHT));
#0025
#0026         // 解決 VCL 在 Windows 98 下透明貼圖 bug 的 workaround
#0027         if (IsWindows98())
#0028           DrawTransparentBitmap(Canvas->Handle, FInvBitmap->Handle,
#0029             x * TILE_WIDTH, y * TILE_HEIGHT, RGB(128, 0, 0));
#0030         else
#0031           Canvas->Draw(x * TILE_WIDTH, y * TILE_HEIGHT, FInvBitmap);
#0032       }
#0033 }
```

0005 列繪製地形層比較單純，呼叫 *BitBlt* 函式，將對應的地形圖片拷貝到 *Canvas* 上的對應位置。但繪製物品層時，首先要檢查該格是否有物品，若 *FItemMap[y][x]* 爲零，表示沒有物品，就不畫；另外要繪製時，先將圖片拷貝至 *FInvBitmap* 上，再呼叫 *TCanvas::Draw* 函式貼上 *FInvBitmap*，目的就是爲了先前討論已久的透明貼圖。理論上是如此，但我們先前提過 VCL 的透明貼圖能力在 Windows 98 下失效，所以 0028 行就專爲解決 VCL 在 Windows 98 下透明貼圖 bug 而特別將 Windows 98 平臺下的貼圖動作獨立出來，先以 *IsWindows98* 函式（Util 單元）判斷目前是否處於 Windows 98 下，若是的話，再呼叫我們自己的 *DrawTransparentBitmap* 函式來達成透明貼圖。

大一統的 Win32 平臺？！

經常撰寫 Win32 程式的程式員，一定會對微軟吶喊的「統一的 Win32 平臺」這句好聽的口號印象深刻吶！這…真的只是口號罷了，我很少寫出一套不需針對不同 Windows 版本修正問題的軟體。判斷系統是否爲 Windows 98 並不難，檢查 VCL 提供的三個全域變數 *Win32Platform*、*Win32MajorVersion* 及 *Win32MinorVersion* 即可，詳情請查閱 *GetVersionEx*

API 函式。而 Windows 98 即等於版本爲 4.10 以上的 Windows：

```
bool IsWindows98()
{
  return (Win32Platform == VER_PLATFORM_WIN32_WINDOWS) &&
    ((Win32MajorVersion > 4) ||
    ((Win32MajorVersion == 4) && (Win32MinorVersion >= 10)));
}
```

若系統爲 Windows NT，則 *Win32Platform* 變數值爲 *VER_PLATFORM_WIN32_NT*，而 Windows 2000 的 *Win32MajorVersion* == 5，因爲其實就是 NT 5.0 嘛。

二維陣列屬性

最後再實作 *TMap* 所提供的一大票屬性，以 *CanPass* 屬性爲例，它是二維陣列屬性，宣告爲：

```
__property bool CanPass[int x][int y] = {read = GetCanPass};
```

它有一個對應的屬性讀取函式，因此就要另外實作此函式：

```
bool TMap::GetCanPass(int x, int y)
{
  return Terrs.Attrs[FTerrMap[y][x]].Contains(taCanPass);
}
```

於是 *TMap* 的物件使用者就可以輕易地使用此屬性，如經由 *Map.CanPass[2][3]* 取得一個布林值，判斷此地圖座標爲（2, 3）的格點是否能夠讓角色通過；由 *Map.CanMove[8][2]* 來判斷座標爲（8, 2）處是否有物品，若有物品，能不能推動等等。

TMap 類別宣告的 0076 列，宣告一個 *TMap* 類別的全域物件 *Map*，原因也跟 *TTerrTiles* 及 *TItemTiles* 類別一樣，因爲 *TMap* 物件只需要一個，因此宣告在這兒最清楚也最方便。

TRole 主角類別

終於剩下最後一個類別－*TRole*（定義於 Role.h）：

```
#0001 enum TDirection {drUp, drDown, drLeft, drRight};
#0002 typedef std::vector<TDirection> TDirectionArray;
#0003
#0004 class TRole {
#0005 private:
#0006   int FX, FY; // 角色座標
#0007   TDirection FDirection; // 行進方向
#0008   Graphics::TBitmap *FBits, *FInvBitmap; // 角色圖片及透明貼圖用圖片
#0009
#0010   TDirectionArray FPlayBackList, FMoveList; // 重播功能用的動作記錄
#0011
#0012   TDirectionArray& GetPlayBackList() {return FPlayBackList;}
#0013 public:
#0014   TRole();
#0015   ~TRole();
#0016
#0017   void LoadBits(); // 載入角色的 bitmap
#0018   void Draw(TCanvas* Canvas); // 繪製角色
#0019   void Move(TDirection Dir); // 移動角色（碰撞處理，移動物品）
#0020
#0021   void SavePlayBack(); // 將動作記錄移至重播紀錄
#0022   void CleanMoveList(); // 清除動作記錄
#0023
#0024   // 位置及方向
#0025   __property int X = {read = FX, write = FX};
#0026   __property int Y = {read = FY, write = FY};
#0027   __property TDirection Direction =
#0028     {read = FDirection, write = FDirection};
#0029
#0030   __property TDirectionArray& PlayBackList =
#0031    {read = GetPlayBackList};
#0032 };
```

0001 列先宣告 *TDirection* 列舉型態，定義出角色可能面對及移動的方向。*FInvBitmap* 變數的目的及用法與 *TMap* 的 *FInvBitmap* 一模一樣，同時是爲了透明貼圖功能而存在的。

TRole 的設定比較偷懶，因爲畫面上只有唯一一個角色，因此圖片也只要一份，上下左右各一張圖片，總共才需一張 128 x 32 的點陣圖。若遊戲中可同時出現多種角色，每種角色又有不同的圖形及行爲模式時，*TRole* 的設計可就沒這麼簡單。一般來說，走動時的圖片都常會一個方向提供三張，分別是站立不動、提起左腳及邁開右腳，若再加上蹲姿或射擊、中彈等其它動作表情等等，所需的圖片數量還真多，同樣地，留待日後再來加強角色的功能。

讀取角色圖形的函式很簡單，呼叫 *TBitmap::LoadFromFile* 從 bitmap 檔案中讀出即可：

```
#0001  void TRole::LoadBits()
#0002  {
#0003    // 讀取 BMP 檔，內含四個方向的圖形
#0004    FBits->LoadFromFile(FN_ROLEBITS);
#0005
#0006    if (FBits->Width < TILE_WIDTH * 4) // 四個方向
#0007      throw Exception("Width of role bits is invalid");
#0008
#0009    // 統一 FInvBitmap 及 FBits 的格式
#0010    FInvBitmap->PixelFormat = FBits->PixelFormat;
#0011  }
```

接下來是畫出角色的 *Draw* 函式，與 *TMap::DrawItemMap* 函式類似，先將對應的圖片拷貝到 *FInvBitmap* 上，再呼叫 *TCanvas::Draw* 將 *FInvBitmap* 以透明貼圖的方式貼上去。這裏也同樣有解決 VCL 在 Windows 98 下透明貼圖 bug 的 workaround。

```
#0001  void TRole::Draw(TCanvas* Canvas)
#0002  {
#0003    // 先將要秀出的區域拷至 FInvBitmap，再畫出 FInvBitmap
#0004    FInvBitmap->Canvas->CopyRect(Rect(0,0,TILE_WIDTH,TILE_HEIGHT),
#0005      FBits->Canvas, Rect((int)FDirection * TILE_WIDTH, 0,
#0006      ((int)Direction + 1) * TILE_WIDTH, TILE_HEIGHT));
#0007
#0008    // 解決 VCL 在 Windows 98 下透明貼圖 bug 的 workaround
#0009    if (IsWindows98())
#0010      DrawTransparentBitmap(Canvas->Handle, FInvBitmap->Handle,
#0011        FX * TILE_WIDTH, FY * TILE_HEIGHT, RGB(128, 0, 0));
#0012    else
#0013      Canvas->Draw(FX * TILE_WIDTH, FY * TILE_HEIGHT, FInvBitmap);
#0014  }
```

移動及推動

TRole 類別最重要的函式，也是與使用者操作最直接相關的，就屬 *Move* 函式了。它決定當使用者按上下左右方向鍵後，角色的移動及搬動物品的處理：

```
#0001  // small but useful function, update x & y location of role
#0002  void MoveAhead(int& X, int& Y, TDirection Dir)
#0003  {
#0004    switch (Dir) {
```

```
#0005     case drUp: Y--; break;
#0006     case drDown: Y++; break;
#0007     case drLeft: X--; break;
#0008     case drRight: X++; break;
#0009   }
#0010 }
#0011
#0012 void TRole::Move(TDirection Dir)
#0013 {
#0014   int X, Y, MX, MY;
#0015
#0016   X = FX; Y = FY;
#0017   MoveAhead(X, Y, Dir); // 計算角色的下一位置
#0018   FDirection = Dir;
#0019
#0020   // 超出畫面範圍了...
#0021   if (Y < 0 || X < 0 || X >= TILE_NUM_X || Y >= TILE_NUM_Y) return;
#0022
#0023   if (!Map.CanPass[X][Y]) return; // 不能走的地形
#0024
#0025   if (Map.ItemMap[X][Y] != 0) { // 如果要移動過去的位置有物品
#0026     if (!Map.CanMove[X][Y]) return; // 不能搬動耶
#0027
#0028     MX = X; MY = Y;
#0029     MoveAhead(MX, MY, Dir); // 計算物品的新位置
#0030     if (MY < 0 || MX < 0 || MX >= TILE_NUM_X || MY >= TILE_NUM_Y)
#0031       return; // 不可將物品移到範圍外
#0032
#0033     // 要搬過去的新位置上不能有東西，也必須是可以走動的地形
#0034     if (Map.ItemMap[MX][MY] != 0 || !Map.CanPass[MX][MY]) return;
#0035
#0036     Map.ItemMap[MX][MY] = Map.ItemMap[X][Y]; // 搬動過去
#0037     Map.ItemMap[X][Y] = 0; // 原來的地方沒有物品了
#0038   }
#0039
#0040   // 成功走出，將移動方向記錄下來
#0041   FMoveList.push_back(Dir);
#0042
#0043   // 更新角色位置
#0044   FX = X;
#0045   FY = Y;
#0046 }
```

先設計一個小小的 *MoveAhead* 函式來輔助位置的處理，根據傳入的 *Dir* 方向，更改 X 或 Y 軸位置，雖然簡單，但很有用。

上面處理角色移動的邏輯並不難，可歸納如下：

1. 0017 列先計算下一步的位置。
2. 0021 列判斷是否超出畫面範圍了，是的話就跳離處理函式。
3. 0023 列判斷是否將走到不能穿過的地形，是的話就跳離處理函式。
4. 0025 列判斷是否將走到有物品的圖格，是的話繼續步驟 5，否則進行步驟 9。
5. 0026 列判斷該物品能否搬動，否的話就跳離處理函式。
6. 0030 列判斷是否會將物品搬出畫面範圍了，是的話就跳離處理函式。
7. 0034 列判斷物品的新位置上是否有東西，是否爲可以穿越的地形？若沒有擺置物品且地形可以穿越就繼續，否則跳離處理函式。
8. 0036、0037 列將物品搬動到新位置。
9. 0041 列將移動方向記錄在 *FMoveList* 物件中，留待重播功能使用。
10. 0044、0045 列更新角色位置，大功告成。

程式乍看之下可能不怎麼懂，但若用文字敘述出來，就變成很簡單的幾道判斷敘述而已，而這已是整個遊戲中最麻煩重要的邏輯處理呢。所以我說的沒錯吧，撰寫遊戲一點都不難，難的通常是顯示技術、速度及程式複雜度，而非程式邏輯。讓我們繼續往下看。

TRole 類別宣告的 0009 列中，將重播動作記錄用的兩個變數 *FPlayBackList* 及 *FMoveList* 宣告爲 *TDirectionArray* 類別，

在這兒，我想要將角色的每一步移動方向都記錄下來，但是角色的移動步數未知，可能只有兩三步，也可能是兩三千步，所以不適合靜態地配置記憶體空間。但自己呼叫 *malloc*、*realloc*、*free* 等函式來管理移動記錄所需的記憶體空間又稍嫌麻煩，*vector* 物件在此是最適宜的容器。

角色動作錄影支援

TRole 的 *SavePlayback* 及 *CleanMoveList* 函式分別將 *FMoveList* 的記錄移至 *FPlaybackList* 中，以及將 *FMoveList* 的內容清除：

```
#0001  void TRole::SavePlayBack()
#0002  {
#0003    // 將行動記錄放到 FPlayBackList 中，以便重播
#0004    FPlayBackList = FMoveList;
#0005  }
#0006
#0007  void TRole::CleanMoveList()
#0008  {
#0009    FMoveList.clear(); // 行動記錄清除以方便下次使用
#0010  }
```

不知不覺間，我們已將 *TTiles*、*TMap* 及 *TRole* 等三個核心類別實作完成，好的開始是成功的一半，緊接著，就要利用上頭介紹的這些類別來架構遊戲的三支程式囉。

圖庫編輯器、地圖編輯器，以及遊戲主程式的實作也有一定的順序。圖庫編輯器還未完成前，沒有圖庫，就無法編輯地圖；而地圖編輯器還未完成前，沒有地圖，就無法進行遊戲。所以看來非從圖庫編輯器下手不可。

圖庫編輯器

RAD 開發工具的好處是，可以在開始撰寫第一行程式碼前，先透過「所視即所得」的整合環境編輯方式，將使用者介面，所有的控制項，元件，選單及視覺佈局擺好在視窗上，待程式一執行，嘿，就是設計時期擺放的那個模樣。

我個人十分偏愛這樣的設計方式，先將使用者介面設計好，再下手來撰寫程式碼。只要介面制訂後不再更動，程式碼就好寫；若是介面甚至元件種類還變來變去，那程式碼肯定就得改來改去，越修越複雜，麻煩的不得了。

好，選取【New / Application】開啓一個新專案順便建立 main form，在 form 上將介面佈局擺好，如圖 8-7，看起來很陽春，我知道，是我的錯，以後改進。圖 8-8 是其選單設計畫面，可由其得知圖庫編輯器準備提供的功能。

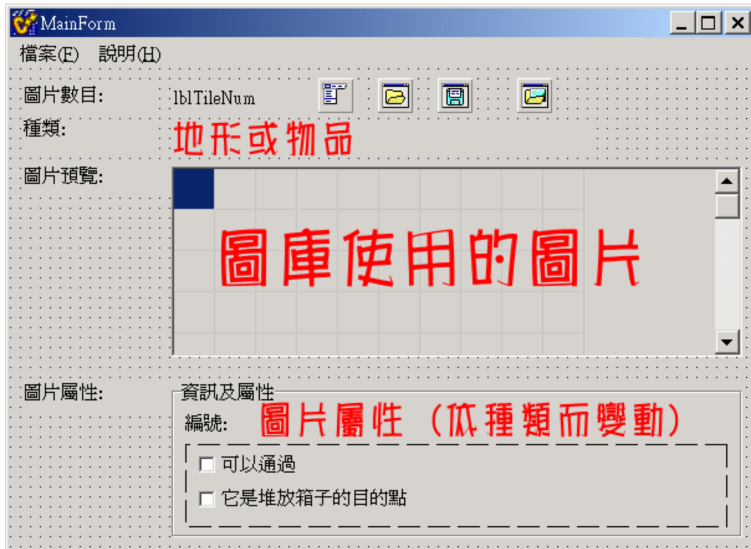

圖 8-7 / 圖庫編輯器的設計畫面

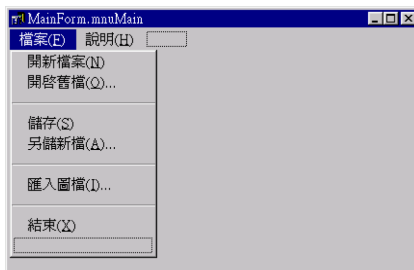

圖 8-8 / 圖庫編輯器的選單設計畫面

假設我準備好 16 張地形圖片（大小皆是 32 x 32），想要納入遊戲使用。典型的操作方式是這樣的：

1. 請先在影像編輯軟體中編輯 320 x 64（因爲每列 10 格，16 張需佔用兩列）大小的影像，分別將 16 張圖片放到適當的位置，第一張擺在（0, 0）、第二張擺在（32, 0）、第三張擺在（64, 0）...，最後儲存爲 BMP 檔案。

2. 接著執行圖庫編輯器，選擇【檔案 / 開新檔案】，圖庫類型選擇爲「地形」，建立新的圖庫。

3. 匯入事先準備好的 BMP 圖檔。

4. 針對每張圖片一一設定它們的屬性，決定角色是否可通過、是否爲目的地形等等。

5. 最後，存檔成 *FN_TERR_ARCHIVE* 常數所指定的檔名，就可以順利供地圖編輯器及遊戲主程式載入使用。

6. 物品圖庫也依上述步驟如法泡製。

我打算利用同一套程式，相同的介面，幾乎相同的程式碼來進行地形圖庫及物品圖庫的製作及處理。

雙重「物」格的 FTiles

要怎麼做呢？地形圖庫及物品圖庫分別屬於 *TTerrTiles* 及 *TItemTiles* 類別，而不同型態，不同類別的物件通常要分別撰寫程式碼處理才行。別忘了，它們來自同一個父類別－*TTiles*，藉著多型機制的幫忙，我們可以讓 *TTerrTiles* 及 *TItemTiles* 物件共享同一段程式碼，甚至共用同一個變數：

```
#0001  class TMainForm : public TForm
#0002  {
#0003  ...
#0004  private:
#0005    TTiles* FTiles;
#0006
#0007    void __fastcall CreateTiles(const std::type_info&);
#0008  };
#0009
#0010  void __fastcall TMainForm::CreateTiles(const std::type_info&
#0011    typeinfo)
#0012  {
#0013    if (FTiles) delete FTiles;
#0014
#0015    if (typeinfo == typeid(TTerrTiles))
#0016      FTiles = new TTerrTiles();
#0017    else if (typeinfo == typeid(TItemTiles))
```

```
#0018     FTiles = new TItemTiles();
#0019   else // 不是預期的類別
#0020     throw Exception("Invalid class type info");
#0021 }
```

你瞧，0005 列宣告著 *TTiles* 類別的 *FTiles* 物件，因為同一時間只能編輯地形或物品圖庫其中之一，因此我使用同一個 *FTiles* 變數來儲放這兩種物件。

我們希望 *FTiles* 變數有時指向 *TTerrTiles* 物件，有時指向 *TItemTiles* 物件，並使用一道專門的函式 *CreateTiles* 來負責 *FTiles* 物件的建立及釋放。那麼，*CreateTiles* 函式一定需要接收參數，來得知應該建立哪一個類別的物件。在 C++ 裡頭，類別只是個抽象的描述，類別的作用就是拿來建立物件，以及衍生新的資料型別，從來無法把它當作參數傳遞，所以我們不能這麼宣告：

```
void CreateTiles(const ????? class) { ... }
...
CreateTiles(TTerrTiles); // 建立 TTerrTiles 物件
CreateTiles(TItemTiles); // 建立 TItemTiles 物件
```

每個變數及物件都有它的資料型別，但類別屬於什麼資料型別？我們可以說 *MainForm* 的資料型別為 *TMainForm*、變數 *i* 的資料型別為 *int*，但 *TMainForm* 以及 *int* 的資料型別是什麼？

答案是 C++ 並沒有規範「資料型別」的「資料型別」。所以我們不能將「資料型別」**直接**當成函式參數傳遞。在這兒，我們可以依賴 C++ 的 RTTI（Run-Time Type Information，執行時期型別資訊）機制來**間接**傳遞「資料型別」。

可別被這串陌生的英文縮寫嚇到了，事實上它一點都不困難，簡單地說，RTTI 就是把「資料型別」的各項資訊，也編譯入目的碼，讓我們可在程式執行時期，取得「資料型別」的各種資訊。這些資訊其實就是我們（人類）可以由原始程式碼輕易得到的訊息及定義，但由於 C++ 是靜態型別語言，所以在編譯之後，許多在程式執行時（CPU）用不上的資料型別定義就失去了，不會帶進目的碼裡頭，這是很直覺的作法，也兼顧目的碼大小與執行效率。例如以下宣告的列舉型別變數 *Alignment*：

```
enum {Evil, Neutral, Good} Alignment;
if (Alignment == Evil) ...
else if if (Alignment == Good) ...
```

經過編譯之後，編譯器會將每一個列舉型別的值編號，例如 *Evil* 爲 0、*Neutral* 爲 1、*Good* 爲 2，而 *Alignment* 只不過是數值可能爲 0 ~ 2 的無號整數。每一個在程式碼中使用 *Evil*、*Neutral* 及 *Good* 之處，都會自動以 0、1、2 來看待。基本上，跟下列寫法無異，只不過以整數值來表示三種陣營，容易讓程式員搞混，大大降低程式維護性而已：

```
unsigned int Alignment; // possible values: 0 ~ 2
if (Alignment == 0) ...
else if if (Alignment == 1) ...
```

CPU 執行程式時只需要這些資訊即足夠，但相對的我們已失去了資訊型別的資訊。例如我們已經無法從 *Alignment* 的數值 0，取得“Evil”這個字串，因爲“Evil”這個字串只在原始碼中有效，並沒有編入目的碼中。

C++ 的 RTTI 就是爲了解決這類問題而出現，此機制會將各種資料型別的資訊記錄下來，一併編譯入目的碼中，讓我們在執行時期時可透過程式取用。事實上，雖然它看起來只是很簡單的機制，C++ 的某些語言設施（如多型）以及 C++Builder 的 RAD 開發環境，皆是由於 RTTI 在背後支援，才能達到。

C++ 的 RTTI 支援，以 *typeid* 保留字爲起始點，由此我們可以取得 *std::type_info* 類別，對於描述資料型別的資訊進行比較、比對及取得型別名稱等動作。例如：

```
int x;
ShowMessage(typeid(x).name());  // 顯示 int

TComponent* b = new TButton(NULL);
ShowMessage(typeid(b).name());  // 顯示 TComponent*
ShowMessage(typeid(*b).name()); // 顯示 TButton (支援多型, see ?)

if (typeid(x) == typeid(y)) ... // 檢查兩個變數的資料型別是否相等
```

對於 *std::type_info* 類別，請參閱 C++Builder 線上說明的 *typeid* 及 *type_info* 等主題。

於是，藉由 *typeid* 保留字及 *std::type_info* 類別，我們可讓 *CreateTiles* 接收 *std::type_info&* 參數，傳入的參數再與 *typeid(TTerrTiles)* 及 *typeid(TItemTiles)* 比對，就

可得知程式現在想要建立何種類別的物件了。*CreateTiles* 函式的呼叫方式為：

```
CreateTiles(typeid(TTerrTiles)); // 建立 TTerrTiles 物件
CreateTiles(typeid(TItemTiles)); // 建立 TItemTiles 物件
CreateTiles(typeid(TButton));     // 來鬧場的？CreateTiles 會丟出例外
```

以 RTTI 驗明「物」格

現在我們能確定的是，*FTiles* 指向的不是 *TTerrTiles* 物件就是 *TItemTiles* 物件，那麼要如何辨別目前指向的物件究竟是哪一種呢？這又得依賴 RTTI 機制了。我們必須使用 *dynamic_cast* 運算子來驗證物件的實際類別。它的使用方式為：

```
dynamic_cast<T>(ptr)
```

T 為類別指標或參考型別，或是 *void**；*ptr* 必須是型態為指標或參考的 expression。若轉型成功，*dynamic_cast* 會回傳型別為 *T* 的指標或參考；若轉型失敗，會傳回 *NULL* 或引發 *Bad_cast* 例外。詳細的轉型規則請見線上說明的「dynamic_cast」主題。

例如以下這道 expression：

```
dynamic_cast<TBarClass*>(FooObjectPtr)
```

如果 *FooObjectPtr* 指向 *TBarClass* 類別或衍生類別的物件，會得到型別的 *TBarClass** 的指標，否則得到 *NULL*。因此，我們可以使用 *dynamic_cast* 運算子來作保證安全的物件向下轉型：

```
// 如果 FooObjectPtr 指向 TBarClass 類別及衍生類別物件的話...
if (TBarClass* p = dynamic_cast<TBarClass*>(FooObjectPtr)) {
  ... // 使用物件指標 p
}
```

你可以在範例程式中看到我大量地用 *dynamic_cast* 運算子，尤其在事件處理函式內：

```
#0001  void __fastcall TMainForm::mnuSaveClick(TObject *Sender)
#0002  {
#0003    // 若是"另存新檔" 或還未指定檔名，就先問使用者檔名
#0004    if (dynamic_cast<TComponent*>(Sender)->Tag == 1 ||
#0005        FFileName == "") {
#0006      dlgSave->Filter = dlgOpen->Filter;
#0007      // 詢問使用者檔名，若按取消就離開
```

```
#0008     if (!dlgSave->Execute()) return;
#0009     FFileName = dlgSave->FileName; // 將檔名記起來
#0010   }
#0011
#0012   FTiles->SaveToFile(FFileName); // 儲存圖庫
#0013   UpdateControlStatus(); // 更新視窗標題
#0014 }
```

VCL 中絕大部分的事件處理函式都帶有一個 *Sender* 參數，代表觸發此函式的物件，因爲不一定是什麼類別，所以 *Sender* 宣告爲指向所有 VCL 類別的始祖－*TObject* 的指標，但其實它可能是一個 *TButton* 元件、一個 *TImage* 元件甚至一個 *TTimer* 計時器元件。只要我們確定它應該是什麼類別的物件，就可以放心地將它轉型，然後呼叫函式或使用屬性。

以上面的 *mnuSaveClick* 函式爲例，由於我只將這個事件處理函式指派給 *mnuSave* 及 *mnuSaveAs* 兩個 *TMenuItem* 物件，所以當 *mnuSaveClick* 函式被觸發時，理論上 *Sender* 參數必定是兩者其中之一（除非程式有其它地方呼此函式），於是我便可放心地將 *Sender* 參數轉型爲 *TComponent* 類別，取得它的 *Tag* 屬性來使用。爲何轉型爲 *TComponent* 類別，而不轉爲 *TMenuItem* 類別呢？其實兩者都行。只是我個人習慣讓程式碼擁有較大的彈性，若將它轉型爲 *TMenuItem* 類別，萬一日後我又想將 *mnuSaveClick* 函式指派給另一個 *TButton* 物件時，型別轉換部分勢必得修改。因此，我個人的習慣是，若要存取某個屬性或呼叫函式，就將它轉型爲該屬性或函式出現的類別。此處來說，*Tag* 屬性是 *TComponent* 類別介紹的新屬性，因此我就將 *Sender* 參數轉爲 *TComponent* 類別。

提到 *Tag* 屬性，它是所有 VCL 元件都擁有的一個屬性，爲四個位元組的長整數，我們可以任意地使用它。如上述的例子，我將【儲存】及【另儲新檔】兩個元件指派同一個事件處理函式，但代表【儲存】的 *TMenuItem* 元件的 *Tag* 屬性爲 0，而代表【另儲新檔】的 *TMenuItem* 元件的 *Tag* 屬性則設爲 1，以此來區分兩者的不同，進行對應的動作。若我不這樣做，而以傳統的做法分別爲兩個 *TMenuItem* 元件撰寫不同的事件處理函式，就會有很多重覆的程式碼，佔空間，維護起來也較麻煩。

Ouch，離題似乎又遠了~~。於是呢，藉由 *dynamic_cast* 運算子之助，我們可以輕易判別出，目前 *FTiles* 變數指向的究竟是地形或者物品圖庫。

```
if (dynamic_cast<TTerrTiles*>(FTiles))
  … // 是地形圖庫唷！
else if (dynamic_cast<TItemTiles*>(FTiles))
  … // 是物品圖庫耶～
```

雙重「物」格變換

什麼情況下，*FTiles* 會指向不同類別的物件呢？只有兩個地方，一是開啓新檔時，詢問使用者準備建立的圖庫類型，二是開啓舊檔時：

```
#0001  void __fastcall TMainForm::mnuNewClick(TObject *Sender)
#0002  {
#0003    // 建立詢問對話盒
#0004    TTileKindDlg* dlg = new TTileKindDlg(this);
#0005    try {
#0006      // 秀出對話盒，使用者按下"確定"後會傳回 mrOK
#0007      if (dlg->ShowModal() == mrOk) {
#0008        if (dlg->rgpTileKind->ItemIndex == 0)
#0009          CreateTiles(typeid(TTerrTiles));    // 地形圖庫
#0010        else
#0011          CreateTiles(typeid(TItemTiles));   // 物品圖庫
#0012
#0013        FOldSelection = -1;
#0014        UpdateControlStatus();
#0015      }
#0016    } __finally {
#0017      delete dlg; // 無論如何，摧毀詢問對話盒
#0018    }
#0019  }
```

開新檔時比較簡單，我另外設計一個詢問使用者圖庫類型的對話盒，當使用者按下「確定」後會傳回 *mrOK*，再根據它的選擇呼叫 *ChangeTileClass* 函式建立地形或物品圖庫物件。

圖 8-9 / 詢問使用者圖庫類型的對話盒

開啓舊檔就比較有趣了，還記得之前設計 *TTiles* 類別時，在 *TTiles::SaveToFile* 函式中有寫入檔頭標籤及副檔頭標籤嗎？我們就靠著這個及 Object Pascal 的例外捕捉機制（try .. except .. end）來判斷讀入的是何種圖庫，同時建立對應的 *FTiles* 物件：

```
#0001  void __fastcall TMainForm::mnuOpenClick(TObject *Sender)
#0002  {
#0003    if (dlgOpen->Execute()) {
#0004      try {
#0005        // 先試試看是否爲地形圖庫
#0006        // 讀取有錯的話, 就會產生例外, 讓我們的例外補捉機制處理
#0007        CreateTiles(typeid(TTerrTiles));
#0008        FTiles->LoadFromFile(dlgOpen->FileName);
#0009        FFileName = dlgOpen->FileName; // 讀取地形圖庫成功
#0010      } catch (...) {
#0011        // 如果不是地形圖庫, 檢查副檔頭標籤時會產生例外,
#0012        // 所以跳到這兒來執行
#0013        // 再試試看是否爲物品圖庫, 否則就什麼都不是
#0014        CreateTiles(typeid(TItemTiles));
#0015        // 有錯的話, 也會產生例外, 我們就不處理了
#0016        FTiles->LoadFromFile(dlgOpen->FileName);
#0017        FFileName = dlgOpen->FileName;
#0018      }
#0019
#0020      FOldSelection = -1;
#0021      UpdateControlStatus();
#0022    }
#0023  }
```

讀取圖庫時採取「**試誤法**」，先試試看是否爲地形圖庫，不是的話，再試試看是否爲物品圖庫，否則就不理它。我憑藉的是 *TTiles::LoadFromFile* 函式中會呼叫 *CheckSignature* 函式來檢查檔頭標籤，而 *CheckSignature* 函式會在檔頭標籤與預期不符時丟出一個例外：

```
#0001  void CheckSignature(TReader* reader, const AnsiString Sig)
#0002  {
#0003    AnsiString S = reader->ReadString();
#0004    if (Sig != S)
#0005      throw Exception("File signature not match");
#0006  }
```

這個看起來很 dirty 又很 smart 的「**試誤法**」，其實是懶得另外撰寫檢查檔頭標籤函式的結果，不然正常的寫法會像是這樣，你喜歡哪個呢？

```
if (IsTerrArchive(dlgOpen->FileName))
  CreateTileClass(typeid(TTerrTiles));
else
  CreateTileClass(typeid(TItemTiles));

FTiles->LoadFromFile(dlgOpen->FileName);
```

繪製圖庫圖片

視窗中央那個 *TDrawGrid* 物件 *grdPreview* 會根據目前的 *FTiles* 內容將圖片顯示出來，*TDrawGrid* 元件本身並不儲存任何資訊，顯示的結果端視我們如何處理它的 *OnDrawCell* 事件而定。以下是 *grdPreview* 的 *OnDrawCell* 事件處理函式：

```
#0001  void __fastcall TMainForm::grdPreviewDrawCell(TObject *Sender, int
#0002    ACol,
#0003        int ARow, TRect &Rect, TGridDrawState State)
#0004  {
#0005    // 由行及列換算圖片編號
#0006    int No = ARow * grdPreview->ColCount + ACol;
#0007
#0008    if (No < FTiles->TileNum) // 將對應的圖形畫出來
#0009      BitBlt(grdPreview->Canvas->Handle, Rect.Left, Rect.Top,
#0010        TILE_WIDTH, TILE_HEIGHT, FTiles->Bitmap->Canvas->Handle,
#0011        FTiles->TilePos_Left[No], FTiles->TilePos_Top[No], SRCCOPY);
#0012    else {
#0013      grdPreview->Canvas->Brush->Color = clBlack; // 編號大於圖片數目
#0014      grdPreview->Canvas->FillRect(Rect);
#0015    }
#0016  }
```

同樣地，還是經由 *TTiles* 的 *TilePos_Left* 及 *TilePos_Top* 兩個屬性來取得圖片在圖庫中的座標，再利用 *BitBlt* API 將該圖片貼到 *grdPreview* 的對應位置上；對於沒有圖片的那些

圖格，就塗黑。

而每當使用者選擇 *grdPreview* 上的某一格時，*grdPreview* 的 *OnSelectCell* 事件處理函式 *grdPreviewSelectCell* 就必須先將目前設定的屬性寫回 *FTiles* 物件中，接著再讀出即將選擇的圖片屬性，依屬性更新 *TCheckBox* 元件狀態：

```
#0001 void __fastcall TMainForm::grdPreviewSelectCell(TObject *Sender,
#0002       int ACol, int ARow, bool &CanSelect)
#0003 {
#0004   int No, OldNo = FOldSelection;
#0005
#0006   // OldNo 爲原本選擇的圖片編號
#0007   if (OldNo < FTiles->TileNum) { // 將使用者設定的圖片屬性寫回去
#0008     // 若是地形圖庫的話...
#0009     if (TTerrTiles* TerrTiles = dynamic_cast<TTerrTiles*>(FTiles)){
#0010       TTerrAttr& Attr = TerrTiles->Attrs[OldNo];
#0011       // 根據 cbxCanPass 及 cbxTarget 兩個 checkbox 來設定圖片屬性
#0012       Attr = TTerrAttr();
#0013
#0014       // 可以穿越的地形
#0015       if (cbxCanPass->Checked) Attr = Attr << taCanPass;
#0016       // 它是目的地形
#0017       if (cbxTarget->Checked) Attr = Attr << taTarget;
#0018     } else if (TItemTiles* ItemTiles =
#0019       dynamic_cast<TItemTiles*>(FTiles)) {
#0020       // 若是物品圖庫的話...
#0021       TItemAttr& Attr = ItemTiles->Attrs[OldNo];
#0022       Attr = TItemAttr();
#0023
#0024       // 可以搬動的物品
#0025       if (cbxCanMove->Checked) Attr = Attr << iaCanMove;
#0026       // 覆蓋用物品
#0027       if (cbxSource->Checked) Attr = Attr << iaSource;
#0028     }
#0029   }
#0030
#0031   No = ARow * grdPreview->ColCount + ACol; // 計算即將選擇的圖片編號
#0032   if (No < FTiles->TileNum) { // 秀出目前所選的圖片屬性
#0033     lblTileNo->Caption = "編號: " + IntToStr(No);
#0034
#0035     // 將所選擇的圖片屬性由 checkbox 元件表現出來
#0036     if (TTerrTiles* TerrTiles = dynamic_cast<TTerrTiles*>(FTiles)){
#0037       TTerrAttr& Attr = TerrTiles->Attrs[No];
#0038       cbxCanPass->Checked = Attr.Contains(taCanPass);
```

```
#0039        cbxTarget->Checked = Attr.Contains(taTarget);
#0040      } else if (TItemTiles* ItemTiles =
#0041        dynamic_cast<TItemTiles*>(FTiles)) {
#0042        TItemAttr& Attr = ItemTiles->Attrs[No];
#0043        cbxCanMove->Checked = Attr.Contains(iaCanMove);
#0044        cbxSource->Checked = Attr.Contains(iaSource);
#0045      }
#0046
#0047      // 更新選擇的 tile no
#0048      FOldSelection = No;
#0049    } else
#0050      CanSelect = false; // 選擇不合法的圖格，不給選
```

嗯，到此為止，也許你不相信，但是圖庫編輯器一不小心就這樣完工了，圖 8-10 及圖 8-11 分別是圖庫編輯器在製作地形及物品圖庫時的執行畫面。從畫面中可以看到，地形我只提供四張圖片，編號 0 號為草地，以它做為預設地形；物品圖片更少，只有一顆足球，放在編號 1 的位置上，因為編號 0 具有特殊意義，代表「此地無任何物品」。

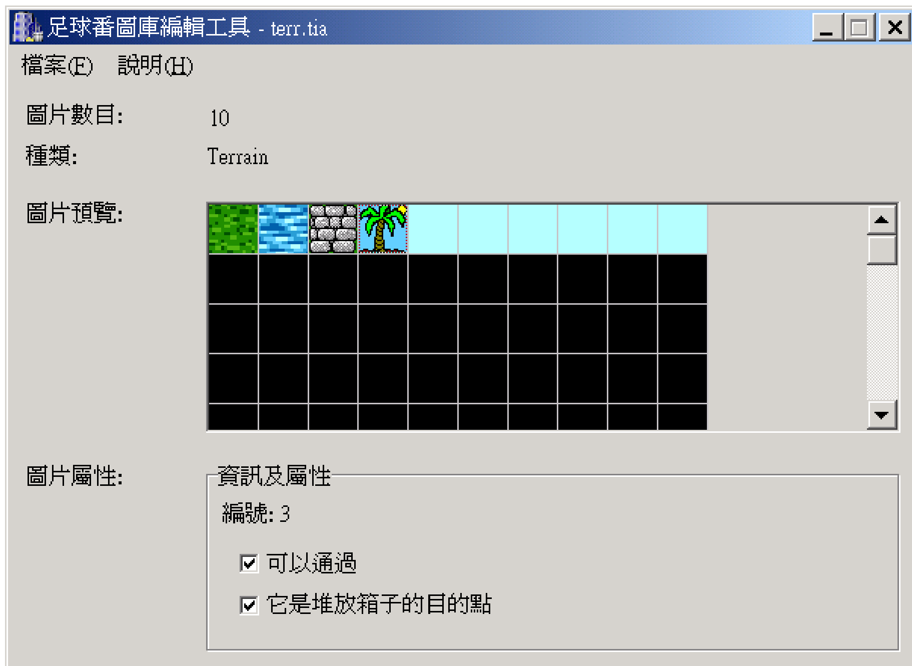

圖 8-10 / 圖庫編輯器的執行畫面（地形圖庫）

圖 8-11 / 圖庫編輯器的執行畫面（物品圖庫）

事實上這大概是全世界最陽春的圖庫編輯器了，跟圖 8-2 及圖 8-3 列出的 StarCraft 及英雄無敵 III 的地圖編輯器一比，咱們的圖庫編輯器羞得無地自容，差點離家出走，還是我好說歹說才將它留住。我想，就算是第二陽春的圖庫編輯器至少也有圖片拉曳、更換位置編號等功能，再者圖片群組及物件的概念也該支援，可以發揮的地方還多得是，讓我們以後慢慢玩吧。

地圖編輯器

有了圖庫編輯器，製作出圖庫後，接著就可以編輯地圖。地圖編輯器的目的很簡單－提供一個 WYSIWYG 的介面讓使用者可以方便地編輯存放那兩層地圖的二維陣列元素值。呵，很繞口吧。

說得清楚點，*TMap* 不是擁有兩個 *TMapArray* 型態的 *TerrMap* 及 *ItemMap* 變數嗎？*TMapArray* 型態是 *TILE_NUM_X * TILE_NUM_Y* 大小的二維 *Byte* 陣列，而**地圖編輯器的目的就是提供與遊戲進行時相同的圖片及畫面以及方便的操作介面，供關卡設計者編輯陣列內容**。

目的很簡單，說來只有幾句話，但寫來比我們那個世界第一笨的圖庫編輯器還稍微複雜一滴滴（以程式碼行數比較的話:p）。還是先將介面設計出來：

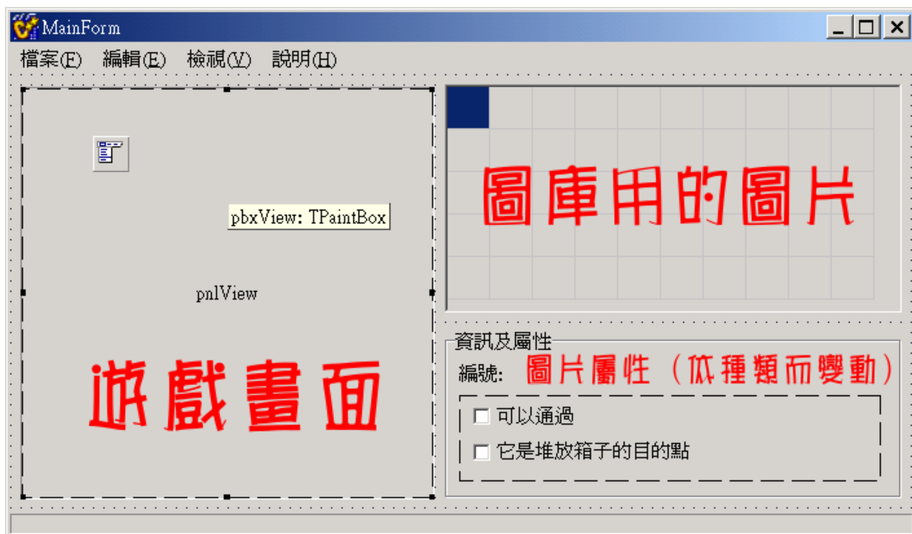

圖 8-12 / 地圖編輯器的設計畫面

程式規劃如下：

1. 有三種編輯模式，分別是地形，物品及角色編輯模式。
2. 與遊戲畫面不同的是，可以顯示格線及特殊區域以利編輯。
3. 能夠檢查關卡地形及物品擺設是否合法。

首先宣告編輯模式型別及視窗類別 *TMainForm*：

```
#0001 // 三種編輯模式: 地形, 物品及主角
#0002 enum TEditKind {ekTerrs, ekItems, ekRole};
#0003
#0004 class TMainForm : public TForm
#0005 {
#0006 ...
#0007 private:
#0008     TEditKind FEditKind; // 目前編輯模式
#0009     int FLevelNo; // 目前編輯的關卡
#0010     bool FModified; // 載入關卡後是否更動過
#0011
#0012     TRole FRole; // 角色物件
#0013     // double-buffering 用的背景 bitmap
#0014     Graphics::TBitmap* FBackBitmap;
#0015
#0016     int FCursorX, FCursorY; // 滑鼠游標位置
#0017     TMouseButton FButtonPressed; // 滑鼠按鍵狀態
#0018
#0019     void __fastcall DrawBackBitmap(); // 繪製背景 bitmap
#0020     void __fastcall UpdateView(); // 更新編輯畫面
#0021     void __fastcall UpdateControlStatus();
#0022
#0023     void __fastcall SetLevelNo(int Value);
#0024
#0025     void __fastcall MySaveMap(); // 儲存地圖檔案
#0026     bool __fastcall AskSaveMap(); // 確認是否儲存地圖
#0027     bool __fastcall ValidateMap(); // 檢查地圖是否合法
#0028
#0029     __property int LevelNo = {read = FLevelNo, write = SetLevelNo};
#0030 ...
#0031 };
```

0012 列宣告了 *TRole* 物件 *FRole*，這是 *TRole* 類別第一回派上用場呢。不過在此只是虛晃幾招，只用它來做顯示及定位角色的用途而已，那複雜的移動邏輯仍派不上用場。

0014 列的 *FBackBitmap* 就是文章前頭談到 double-buffering 時所說的背景 bitmap。在 *UpdateView* 函式中，會先呼叫 *DrawBackBitmap* 將應該出現在畫面上的所有東東繪製於 *FBackBitmap*，再呼叫 *TCanvas::Draw* 函式一口氣將 *FBackBitmap* 畫上 *pbxView*，也就是代表編輯畫面的 *TPaintBox* 畫布上頭。所以 *UpdateView* 函式只有簡單的短短兩行：

```
#0001 void __fastcall TMainForm::UpdateView()
#0002 {
#0003   DrawBackBitmap(); // 將畫面放到 FBackBitmap
```

```
#0004   pbxView->Canvas->Draw(0, 0, FBackBitmap); // 複製到 pbxView
#0005  }
```

在地圖編輯器中，因為必須同時使用地形圖庫及物品圖庫，因此就不像圖庫編輯器那樣，兩者共用一個 *TTiles* 變數，而直接使用宣告在 TileUnit 單元中的全域變數 *Terrs* 及 *Items*。

程式初始化

在 *OnCreate* 事件處理函式中，初始化所有的物件，並載入角色圖片及地形、物品圖庫，另外還設定好 *FBackBitmap*，讓它跟編輯畫面一樣大：

```
#0001  void __fastcall TMainForm::FormCreate(TObject *Sender)
#0002  {
#0003    FEditKind = ekTerrs; // 預設為地形編輯模式
#0004    FButtonPressed = mbMiddle; // 表示目前滑鼠鍵沒有按著
#0005
#0006    try {
#0007      FRole.LoadBits(); // 讀入主角的圖形
#0008
#0009      Terrs.LoadFromFile(AppDir + FN_TERR_ARCHIVE); // 讀入地形
#0010      Items.LoadFromFile(AppDir + FN_ITEM_ARCHIVE); // 讀入物品
#0011    } catch (Exception& E) {
#0012      ShowMessage(E.Message);
#0013      // 以 asynchronous 方式關閉視窗
#0014      PostMessage(Handle, WM_CLOSE, 0, 0);
#0015    }
#0016
#0017    // According tile num and size, adjust dimension of pnlView
#0018    pnlView->ClientWidth = TILE_WIDTH * TILE_NUM_X;
#0019    pnlView->ClientHeight = TILE_HEIGHT * TILE_NUM_Y;
#0020
#0021    FBackBitmap = new Graphics::TBitmap; // 緩衝用 bitmap
#0022    FBackBitmap->Width = TILE_WIDTH * TILE_NUM_X;
#0023    FBackBitmap->Height = TILE_HEIGHT * TILE_NUM_Y;
#0024
#0025    LevelNo = 1;
#0026    UpdateControlStatus();
#0027
#0028    // 滑鼠游標位置
#0029    FCursorX = -1;
#0030    FCursorY = -1;
#0031  }
```

0009 及 0010 列載入圖庫檔案時所用的 *AppDir* 字串記錄著程式執行檔所在的目錄，由程式庫的 xFiles 單元提供。

0021 ~ 0023 列建立 double-buffering 用的背景 bitmap *FBackBitmap*，並將大小設定與遊戲畫面相同，這使得 *UpdateView* 函式中，將 *FBackBitmap* 內容複製到編輯畫面 *pbxView* 的動作省事許多。

奇妙的 LevelNo 屬性

LevelNo 是整數型態的屬性，每當設定新值時，就會呼叫 *SetLevelNo* 函式去設定（見類別宣告 0029 列）：

```
#0001  void __fastcall TMainForm::SetLevelNo(int Value)
#0002  {
#0003    try {
#0004      // 讀取地圖檔及角色位置
#0005      Map.LevelNo = Value;
#0006    } catch(...) {
#0007      // 若讀取地圖檔失敗, 清除整張地圖
#0008      Map.ResetMap();
#0009    }
#0010
#0011    FRole.X = Map.Role_X; // 將角色位置由 Map 物件中抄出來
#0012    FRole.Y = Map.Role_Y;
#0013
#0014    FLevelNo = Value;
#0015    FModified = False; // 地圖尚未更動（當然:p）
#0016    UpdateControlStatus();
#0017    UpdateView(); // 別忘了更新遊戲畫面
#0018  }
```

這就是我爲什麼喜歡使用屬性的原因。瞧 *FormCreate* 函式中簡簡單單的一道敘述，將 *LevelNo* 設爲 1，事實上它的屬性寫入函式－*SetLevelNo* 就會爲我讀取第一關的地圖出來，同時取得角色位置，設定其它變數，並且更新編輯畫面等等，多麼優雅的撰寫方式呀。屬性機制讓所有讀／寫的邊際效應都漂亮地隱藏在屬性值讀／取的簡單敘述背後。

LevelNo 屬性甚至可以這樣使用：

```
#0001  void __fastcall TMainForm::mnuRestoreLevelClick(TObject *Sender)
#0002  {
#0003    // 會觸發 property write method (SetLevelNo)
#0004    LevelNo = LevelNo;
#0005  }
```

mnuRestoreLevelClick 是選擇【恢復此關卡原狀】選項的事件處理函式，看到上面那行程式碼，不曉得 *LevelNo* 是屬性的傢伙一定會認爲我花轟了，竟然把一個變數指派給自己，只是浪費 CPU 時間的無意義動作呀。但是別忽略隱藏在 *LevelNo* 屬性背後的 *SetLevelNo* 函式，將 *LevelNo* 指派給 *LevelNo* 的結果是，*SetLevelNo* 函式會去重新載入目前的關卡地圖，達成「恢復此關卡原狀」的目的，很特別吧。

繪製編輯畫面

前頭剛提過，呈現編輯畫面的核心函式只有一個－*DrawBackBitmap*，它先繪製地形層，接著物品層，最後是角色。可以想像出，待會才要進行的遊戲主程式的 *DrawBackBitmap* 函式，似乎也只需要這三個動作就夠了。但在地圖編輯器中，還要能夠顯示格線及特殊區域，所以繪製物品層後，繪製角色前，另外加上三段程式碼，分別將格線，目的地形及覆蓋用物品標示出來：

```
#0001  void __fastcall TMainForm::DrawBackBitmap()
#0002  {
#0003    Map.DrawTerrMap(FBackBitmap->Canvas); // 繪製地形層
#0004    Map.DrawItemMap(FBackBitmap->Canvas); // 繪製物品層
#0005
#0006    // 如果使用者想看格線, 就將格線畫出來
#0007    if (mnuShowGrid->Checked) {
#0008      FBackBitmap->Canvas->Pen->Color = clBlack;
#0009      FBackBitmap->Canvas->Pen->Style = psSolid;
#0010      FBackBitmap->Canvas->Pen->Width = 1;
#0011
#0012      // 先畫橫線
#0013      for (int y = 0; y < TILE_NUM_Y; y++) {
#0014        FBackBitmap->Canvas->MoveTo(0, y * TILE_HEIGHT);
#0015        FBackBitmap->Canvas->LineTo(TILE_NUM_X * TILE_WIDTH,
#0016          y * TILE_HEIGHT);
#0017      }
#0018
```

```
#0019     // 再畫直線
#0020     for (int x = 0; x < TILE_NUM_X; x++) {
#0021       FBackBitmap->Canvas->MoveTo(x * TILE_WIDTH, 0);
#0022       FBackBitmap->Canvas->LineTo(x * TILE_WIDTH,
#0023         TILE_NUM_Y * TILE_HEIGHT);
#0024     }
#0025   }
#0026
#0027   // 如果使用者想看特殊區域，將目的地地形標出來
#0028   if (mnuShowSpeicalArea->Checked) {
#0029     FBackBitmap->Canvas->Pen->Color = clRed;
#0030     FBackBitmap->Canvas->Pen->Width = 1;
#0031     FBackBitmap->Canvas->Pen->Style = psDash;
#0032     FBackBitmap->Canvas->Brush->Style = bsClear;
#0033
#0034     for (int y = 0; y < TILE_NUM_Y; y++) // 逐一檢查
#0035       for (int x = 0; x < TILE_NUM_X; x++)
#0036         if (Map.IsTarget[x][y]) { // 外框加上右上畫到左下的紅色虛線
#0037           FBackBitmap->Canvas->Rectangle(x * TILE_WIDTH,
#0038             y * TILE_HEIGHT, (x + 1) * TILE_WIDTH - 1,
#0039             (y + 1) * TILE_HEIGHT - 1);
#0040           FBackBitmap->Canvas->MoveTo((x + 1) * TILE_WIDTH,
#0041             y * TILE_HEIGHT);
#0042           FBackBitmap->Canvas->LineTo(x * TILE_WIDTH,
#0043             (y + 1) * TILE_HEIGHT);
#0044         }
#0045    }
#0046
#0047   // 如果使用者想看特殊物品，將覆蓋用物品標出來
#0048   if (mnuShowSpeicalItems->Checked) {
#0049     FBackBitmap->Canvas->Pen->Color = clLime;
#0050     FBackBitmap->Canvas->Pen->Width = 1;
#0051     FBackBitmap->Canvas->Pen->Style = psDash;
#0052     FBackBitmap->Canvas->Brush->Style = bsClear;
#0053
#0054     for (int y = 0; y < TILE_NUM_Y; y++) // 逐一檢查
#0055       for (int x = 0; x < TILE_NUM_X; x++)
#0056         if (Map.IsSource[x][y]) { //
#0057           外框加上左上畫到右下的亮綠色虛線
#0058           FBackBitmap->Canvas->Rectangle(x * TILE_WIDTH,
#0059             y * TILE_HEIGHT, (x + 1) * TILE_WIDTH - 1,
#0060             (y + 1) * TILE_HEIGHT - 1);
#0061           FBackBitmap->Canvas->MoveTo(x * TILE_WIDTH,
#0062             y * TILE_HEIGHT);
#0063           FBackBitmap->Canvas->LineTo((x + 1) * TILE_WIDTH,
#0064            (y + 1) * TILE_HEIGHT);
```

```
#0065          }
#0066     }
#0067
#0068     FRole.Draw(FBackBitmap->Canvas); // 最後畫出角色圖案
#0069   }
```

你可以先偷偷看一下圖 8-13，看看這段程式碼繪出的畫面長得什麼樣子。當然囉，可以獨立開關這三個顯示選項，預設值爲關，所以你自己執行時不會看到格線及特殊區域，選取功能表將它們打開後才看得到。

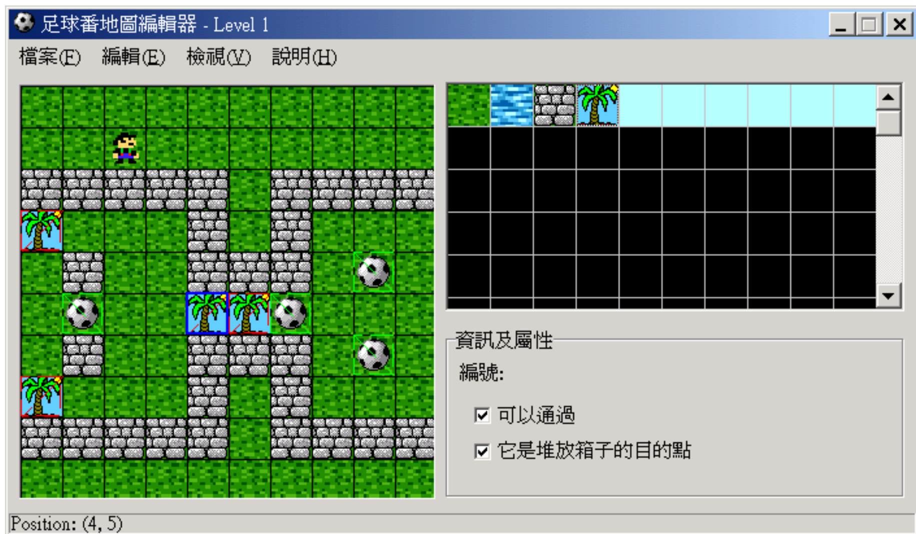

圖 8-13 / 地圖編輯器的執行畫面（將三個顯示選項都打開了）

右上方顯示圖庫圖片的 *grdView* 及右下角顯示圖片屬性的 *TCheckBox* 都跟圖庫編輯器中一樣，因此不再贅述。

選取【編輯 /（地形、物品、人物）】時，會觸發 *mnuEditRoleClick* 函式，首先將觸發此函式的 *TMenuItem* 元件打勾（即 *Checked* 屬性設爲 *true*），設定 *FEditKind* 的新值，此處我又使用前述的函式，將三個 *TMenuItem* 元件的 *OnClick* 事件全指派到同一個事件處理函式，以 *Tag* 屬性區分彼此。所以 0009 列只要輕鬆的一行指派敘述，到 *Kinds* 陣列中查表， 就可以將 *FEditKind* 設定爲對應的編輯模式。此函式的其它部分都在處理切換編輯模式時控制項的介面差異問題（顯示顏色，控制項是否致能等等）。

```
#0001 void __fastcall TMainForm::mnuEditKindClick(TObject *Sender)
#0002 {
#0003   const TEditKind Kinds[3] = {ekTerrs, ekItems, ekRole};
#0004   const TColor Colors[2] = {clBtnFace, clWindow};
#0005
#0006   TMenuItem* item = dynamic_cast<TMenuItem*>(Sender);
#0007
#0008   item->Checked = true;
#0009   FEditKind = Kinds[item->Tag]; // 設定選取的編輯模式
#0010
#0011   UpdateControlStatus();
#0012   UpdateView();
#0013
#0014   ...
#0015 }
```

滑鼠的拉曳及物品置放

真正的好菜現在上桌，不論在何種編輯模式下，當滑鼠游標在編輯畫面範圍內時，游標底下對映的圖格就會有個水藍色的框框跟著它跑，這是怎麼做到的呢？這效果來自於 *grdView* 的 *OnMouseMove* 事件處理函式 *grdViewMouseMove*：

```
#0001 void __fastcall TMainForm::pbxViewMouseMove(TObject *Sender,
#0002     TShiftState Shift, int X, int Y)
#0003 {
#0004   // 已經跑出範圍外了...
#0005   if (X < 0 ||  X >= TILE_WIDTH * TILE_NUM_X || Y < 0 ||
#0006     Y >= TILE_HEIGHT * TILE_NUM_Y) return;
#0007
#0008   FCursorX = X / TILE_WIDTH; // 換算座標，以圖格爲單位
#0009   FCursorY = Y / TILE_HEIGHT;
#0010
#0011   // 一邊拉曳一邊置放物品的效果
#0012   pbxView->OnMouseDown(Sender, FButtonPressed, Shift, X, Y);
#0013
#0014   UpdateView(); // 重繪遊戲畫面
#0015
#0016   // 畫出目前所選取的區域外框
#0017   pbxView->Canvas->Pen->Width = 2;
#0018   pbxView->Canvas->Pen->Color = clBlue;
#0019   pbxView->Canvas->Brush->Style = bsClear;
#0020   pbxView->Canvas->Rectangle(FCursorX * TILE_WIDTH,
#0021     FCursorY * TILE_HEIGHT, (FCursorX + 1) * TILE_WIDTH,
```

```
#0022     (FCursorY + 1) * TILE_HEIGHT);
#0023
#0024    stbMain->SimpleText = Format("Position: (%d, %d)",
#0025      OPENARRAY(TVarRec, (FCursorX, FCursorY)));
#0026  }
```

每當滑鼠指標在控制項上移動時，就會不斷產生 *OnMouseMove* 事件。那麼，你也許會問，既然如此，那麼滑鼠指標一定是在控制項範圍內呀，又何必要有 0005 ~ 0006 列的範圍檢查碼呢？原因是，若使用者在控制項內按下滑鼠任一鍵然後「拉曳」的話，此控制項就會不斷地收到 *OnMouseMove* 事件，不論滑鼠指標是否早已移出控制項範圍，直到滑鼠鍵放開，所以 0005 ~ 0006 列的範圍檢查是必要的。

0008 ~ 0009 列將座標值由以像素為單位換算為以圖格為單位。0012 列是為了達成一邊拉曳滑鼠一邊置放物品的效果，主動觸發 *pbxView* 的 *OnMouseDown* 事件處理函式以設定或清除圖格。接著，畫出所選取區域的外框。藍色外框是仿英雄無敵 III 地圖編輯器而來的，我覺得挺顯眼好看。

接著介紹最重要的一個，也就是達成地圖編輯器目的的重要函式－修改地圖內容的 *pbxView OnMouseDown* 事件處理函式：

```
#0001  void __fastcall TMainForm::pbxViewMouseDown(TObject *Sender,
#0002       TMouseButton Button, TShiftState Shift, int X, int Y)
#0003  {
#0004    // 將按下的按鍵記錄起來，配合 OnMouseMove event handler
#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    // 計算目前選擇的圖片編號
#0011    int No = grdPreview->Row * grdPreview->ColCount + grdPreview->Col;
#0012
#0013
#0014    if (Button == mbLeft) { // 左鍵是設定
#0015      switch (FEditKind) { // 根據編輯模式不同進行不同的設定動作
#0016        case ekTerrs:
#0017          Map.TerrMap[FCursorX][FCursorY] = No; // 將新地形擺上
#0018          break;
#0019
#0020        case ekItems:
```

```
#0021         // 物品不可擺在角色身上
#0022         if (FRole.X == FCursorX && FRole.Y == FCursorY) return;
#0023
#0024         Map.ItemMap[FCursorX][FCursorY] = No; // 將新物件擺上
#0025         break;
#0026
#0027       case ekRole:
#0028         // 角色不可以擺在物品上
#0029         if (Map.ItemMap[FCursorX][FCursorY] != 0) return;
#0030         // 角色不可以擺在不可走動的地形上
#0031         if (!Map.CanPass[FCursorX][FCursorY]) return;
#0032
#0033         FRole.X = FCursorX; // 設定角色位置
#0034         FRole.Y = FCursorY;
#0035         break;
#0036     }
#0037   } else { // 右鍵是清除
#0038     switch (FEditKind) {
#0039       case ekTerrs: Map.TerrMap[FCursorX][FCursorY] = 0; // 清除地形
#0040                     break;
#0041
#0042       case ekItems: Map.ItemMap[FCursorX][FCursorY] = 0; // 清除物品
#0043                     break;
#0044
#0045       case ekRole:   // 角色不能清掉, so do nothing
#0046                     break;
#0047     }
#0048   }
#0049
#0050   FModified = true;
#0051   UpdateView();
#0052 }
```

這段程式碼的註解相當清楚，邏輯也十分簡單，首先判斷使用者按下的是左鍵或右鍵，左鍵代表置放，右鍵代表清除。接著再根據目前的編輯模式，置放地形、物品、角色或是清除地形或物品。

唯一要注意的就是置放物品及角色前，要小心會不會讓地圖產生不合法，或是遊戲無法進行的窘況，例如角色不可擺在物品上，也不可擺在無法穿越的地形上這類的合法性檢查。

FButtonPressed 變數是用來實作拉曳設定效果的關鍵處，即是你可以按著滑鼠左鍵隨意

在遊戲畫面上游走，經過之處就會擺上目前選擇的地形或物品，編輯起來爽快多了。*FButtonPressed* 是 *TMouseButton* 列舉型態，其值可能爲 *mbLeft*、*mbRight*、*mbMiddle* 三者之一，分別代表滑鼠左鍵、右鍵及中鍵。在此利用 *FButtonPressed* 記錄著使用者目前按下的滑鼠鍵，原本還須用一個布林變數來記錄目前是否真正按著滑鼠鍵，但因爲我們沒用到中鍵，所以設定當 *FButtonPressed* 爲 *mbMiddle* 時，就代表滑鼠鍵沒有按著，其值爲 *mbLeft* 或 *mbRight* 時，才代表滑鼠左鍵或右鍵正被按壓著。

擁有 *FButtonPressed* 資訊後，就可以在 *OnMouseMove* 事件處理函式中，主動呼叫 *OnMouseDown* 的事件處理函式，設定或清除對應的圖格。哦對了，別忘了撰寫 *OnMouseUp* 事件處理函式，在滑鼠鍵放開時，將 *FButtonPressed* 設爲 *mbMiddle*：

```
void __fastcall TMainForm::pbxViewMouseUp(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
{
  // 設定爲 mbMiddle, 表示使用者已放開滑鼠鍵
  FButtonPressed = mbMiddle;
}
```

關卡合法性檢查

地圖編輯器還有一項重要的功能，就是在關卡設計完成後，儲存入檔案前，檢查地圖的合法性，覆蓋用物品及目的地形數目，角色是否位於不能移動的地形上等等，合法的關卡才能順利地進行遊戲。檢查地圖合法性的 *ValidateMap* 函式程式碼如下：

```
#0001  // 檢查地圖是否合法
#0002  bool __fastcall TMainForm::ValidateMap()
#0003  {
#0004    int x, y;
#0005    int SourceCount, TargetCount;
#0006
#0007    SourceCount = 0; // 覆蓋用物品數目
#0008    TargetCount = 0; // 目的地形數目
#0009    for (int y = 0; y < TILE_NUM_Y; y++)
#0010      for (int x = 0; x < TILE_NUM_X; x++) {
#0011
#0012        // 計算覆蓋用物品及目的地形數目
#0013        if (Map.IsSource[x][y]) SourceCount++;
```

```
#0014       if (Map.IsTarget[x][y]) TargetCount++;
#0015
#0016       // 此格地圖中記錄的圖片編號是不是大於圖庫的圖片數目 ?
#0017       // 是的話就調回預設值
#0018       if (Map.TerrMap[x][y] > Terrs.TileNum) Map.TerrMap[x][y] = 0;
#0019       if (Map.ItemMap[x][y] > Items.TileNum) Map.ItemMap[x][y] = 0;
#0020     }
#0021
#0022   // 是否沒有目的地形 ? (無法過關)
#0023   if (TargetCount == 0 &&
#0024     !YesNoBox("沒有目的地形, 是否繼續？", MB_DEFBUTTON1))
#0025     return false;
#0026
#0027   // 是否覆蓋用物品少於目的地形 ? (無法過關)
#0028   if (TargetCount > SourceCount &&
#0029     !YesNoBox("覆蓋用物品少於目的地形, 是否繼續？", MB_DEFBUTTON1))
#0030     return false;
#0031
#0032   // 角色位於不能移動的地形上
#0033   if (!Map.CanPass[Map.Role_X][Map.Role_Y] &&
#0034     !YesNoBox("主角位於不能移動的地形上, 是否繼續？",MB_DEFBUTTON1))
#0035     return false;
#0036
#0037   return true;
#0038 }
```

至此，地圖編輯器也順利完工，讓我們再看一次執行畫面，這回三個特殊顯示選項沒有打開。嗯，遊戲畫面也可從這兒的編輯畫面看出大概了，是不是對即將完成的遊戲更充滿期待呢？

圖 8-14 / 地圖編輯器的執行畫面

明明是倉庫番類型的遊戲，不過物品圖庫竟然沒有箱子的蹤影，嘻，這是因爲我找不到箱子的圖片，只好以足球代替，這也是這套遊戲之所以稱爲「足球番」的原因。:p 無論如何，這套「足球番」已呼之欲出，加把勁就要完成了，休息一下，咱們繼續。

「足球番」主程式

一路過關斬將，砍了圖庫編輯器，宰掉地圖編輯器，最後來到大魔王－「足球番」主程式前...

老法子，先在將視窗介面設計好。在設計時期看起來，這遊戲主程式比前兩支程式都還簡單，因爲只有一個 *TPaintBox* 元件 *pbxView*，上面再放著一個 *TMainMenu* 元件及三個計時器元件而已。

圖 8-15 / 「足球番」主程式的設計畫面

在下列的 *TMainForm* 類別宣告中，0002 列首先宣告 *TGameStatus* 型態，定義四種狀態，分別是「歡迎畫面」、「遊戲進行中」、「關卡完成後的慶祝畫面」及「過關動作回顧」（即重播功能）。有許多變數，物件及函式的命名及含意都與地圖編輯器一模一樣，如代表角色的 *FRole*，擔任 double-buffering 緩衝區的 *FBackBitmap*，繪製背景 bitmap 的 *DrawBackBitmap* 函式及更新遊戲畫面的 *UpdateView* 函式等等。比較新鮮的是存放目前遊戲狀態的 *FGameStatus* 變數，記錄可以播放過關回顧關卡編號的 *FPlayBackLevelNo* 變數，可在遊戲畫面中央或正上方畫出字串及方框的 *DrawStatusBox* 函式，以及檢查是否已完成任務的 *CheckFinished* 函式等。類別宣告的原始程式碼列表如下：

```
#0001  // 有四種狀態, Title 畫面, 遊戲中, 完成某關卡及過關回顧
#0002  enum TGameStatus {gsTitle, gsPlaying, gsSuccess, gsPlayBack};
#0003
#0004  class TMainForm : public TForm
#0005  {
#0006  ...
#0007  private:
#0008      TGameStatus FGameStatus; // 遊戲狀態
#0009      int FLevelNo, FPlayingTime; // 目前關卡及遊戲進行時間
#0010      TRole FRole; // 角色物件
#0011
#0012      // double-buffering 用的背景 bitmap
```

```
#0013     Graphics::TBitmap* FBackBitmap;
#0014     int FPlayBackLevelNo; // 可以 playback 的 LevelNo
#0015
#0016     void __fastcall DrawBackBitmap(); // 繪製背景 bitmap
#0017     void __fastcall UpdateView(); // 更新編輯畫面
#0018     void __fastcall UpdateControlStatus();
#0019
#0020     // 在遊戲畫面中央或正上面繪出字串及外圍方框
#0021     void __fastcall DrawStatusBox(AnsiString S, bool TopOrCenter);
#0022     bool __fastcall CheckFinished(); // 檢查是否已完成任務
#0023
#0024     void __fastcall SetGameStatus(TGameStatus Value);
#0025     void __fastcall SetLevelNo(int Value);
#0026
#0027     __property int LevelNo = {read = FLevelNo, write = SetLevelNo};
#0028     __property TGameStatus GameStatus =
#0029       {read = FGameStatus, write = SetGameStatus};
#0030  ...
#0031  };
```

TMainForm 的 *OnCreate* 事件處理函式與地圖編輯器中幾乎完全一樣:同樣地建立及初始化 *TRole*、*TMap*、*TTiles* 物件及緩衝用 bitmap 等等，並沒有新的工作。

三個小時鐘

而丟在 form 上的三個 *TTimer* 計時器元件，它們的任務是什麼呢？

- *tmrTitle*
  負責在歡迎畫面時，讓角色飛快地胡亂移動，帶來爆笑效果，觸發間隔設爲 50 毫秒。

- *tmrTimer*
  爲遊戲進行中的計時器，每 1000 毫秒，即一秒鐘觸發一次，同時更新螢幕上的時鐘，讓使用者曉得此關卡已花費多少時間。

- *tmrPlayback*
  重播過關記錄時，每 500 毫秒觸發一次，讓角色每半秒鐘根據先前的動作記錄移動到下一步，若不使用計時器來放慢速度，直接用迴圈來播放，幾百步驟的動作記錄可能一眨眼就播完了:p

它們的觸發事件處理函式分別如下：

```
#0001  void __fastcall TMainForm::tmrTimerTimer(TObject *Sender)
#0002  {
#0003    FPlayingTime++; // 遊戲進行時間加一秒
#0004    UpdateView(); // 強制更新畫面
#0005  }
#0006
#0007  void __fastcall TMainForm::tmrTitleTimer(TObject *Sender)
#0008  {
#0009    FRole.Move((TDirection)random(4)); // 讓主角任意走動
#0010    UpdateView(); // 更新畫面
#0011  }
#0012
#0013  void __fastcall TMainForm::tmrPlayBackTimer(TObject *Sender)
#0014  {
#0015    // 按照之前的動作循序走動，使用 tmrPlayBack 的 Tag
#0016    //  屬性來記錄目前走到第幾步
#0017    TDirectionArray& plist = FRole.PlayBackList;
#0018    FRole.Move(plist[tmrPlayBack->Tag]);
#0019
#0020    // 這一步走過了，遞增至下一步
#0021    tmrPlayBack->Tag = tmrPlayBack->Tag + 1;
#0022
#0023    // 全部播完了，過關畫面
#0024    if ((unsigned int)tmrPlayBack->Tag == FRole.PlayBackList.size())
#0025      GameStatus = gsSuccess;
#0026
#0027    UpdateView(); // 更新畫面
#0028  }
```

tmrTimer 觸發時只要遞增 *FPlayingTime*，並且強制更新畫面，*DrawBackBitmap* 函式就會根據 *FPlayingTime* 的值將此關卡已花費時間畫在右上角。

tmrTitle 觸發時十分放心地呼叫 *random* 函式取得上下左右任一方向，接著呼叫 *TRole::Move* 函式將角色移向亂數取得的方向，有點不知死活的樣子，不過這樣子不會出問題，因爲我們的移動碰撞檢查碼都放在 *Move* 函式裏，所以若檢查爲不合法的移動，角色就會留在原地不動，這樣一來，因爲移動方向有時合法有時不合法，還可模擬出時走時停的效果呢。

tmrPlaybackTimer 的觸發事件處理函式中，將已播放的步數存在它本身的 *Tag* 屬性，然

後經由 *TDirectionArray* 物件 *PlaybackList* 查得目前這一步的走法。0022 列檢查，若是全部移動記錄播放完畢，則立即進入過關畫面，反正我們只有使用者過關後才會將移動記錄保留下來，因此播放的一定是過關走法，所以移動記錄全部播完畢時一定正好過關。

遊戲狀態的初始化

這些計時器由 *GameStatus* 屬性的屬性寫入函式來控制啓動狀態，*SetGameStatus* 任務重大，負責在進入各個遊戲狀態時，開關這三個計時器，並分別將該狀態所需的變數或物件初始化：

```
#0001  void __fastcall TMainForm::SetGameStatus(TGameStatus Value)
#0002  {
#0003    FGameStatus = Value;
#0004
#0005    // 根據新的遊戲狀態開關三個計時器
#0006    tmrTimer->Enabled = FGameStatus == gsPlaying;
#0007    tmrPlayBack->Enabled = FGameStatus == gsPlayBack;
#0008    tmrTitle->Enabled = FGameStatus == gsTitle;
#0009
#0010    switch (FGameStatus) {
#0011      case gsTitle: LevelNo = 1; // 歡迎畫面顯示第一關地圖
#0012        break;
#0013
#0014      case gsPlaying:
#0015        FPlayingTime = 0; // 計時歸零
#0016        FRole.CleanMoveList(); // play back 歸零
#0017        break;
#0018
#0019      case gsSuccess:
#0020        // 過關了, 將關卡記錄起來, 表示要重播時就回此關卡
#0021        FPlayBackLevelNo = FLevelNo;
#0022        break;
#0023
#0024      case gsPlayBack:
#0025        LevelNo = FPlayBackLevelNo; // 切換到記錄重播回顧的關卡
#0026        tmrPlayBack->Tag = 0; // 從第一步開始播放
#0027        break;
#0028    }
#0029
#0030    UpdateControlStatus(); // 更新標題列及其它控制項
```

```
#0031   UpdateView(); // 更新遊戲畫面
#0032 }
```

因為三個計時器只有分別在歡迎畫面，遊戲中，及重播過關回顧時時才須啓動，因此 0006 ~ 0008 列根據新的遊戲狀態設定它們的 *Enabled* 屬性，同一時間最多只有一個計時器為啓動狀態。0025 列，進入重播過關回顧狀態時，必須主動載入記錄重播回顧的關卡（同時會將角色擺在初始位置），如此才可順利進行重播，要不然若目前處於第一關地圖，而動作記錄是第二關記下來的，就會看到主角到處碰壁，亂走一通的蠢模樣。

設定 *LevelNo* 屬性，也就是間接呼叫 *SetLevelNo* 函式時，裏頭再呼叫 *TMap::LevelNo* 來載入關卡：

```
#0001 void __fastcall TMainForm::SetLevelNo(int Value)
#0002 {
#0003   Map.LevelNo = Value;
#0004   FRole.X = Map.Role_X;
#0005   FRole.Y = Map.Role_Y;
#0006
#0007   FLevelNo = Value;
#0008   UpdateControlStatus();
#0009   UpdateView();
#0010 }
```

唯一要特別注意的是，載入關卡後，千萬別忘了將角色的初始位置由 *Map* 物件的 *Role_X* 及 *Role_Y* 屬性中讀出，更新 *FRole* 的位置。

繪製遊戲畫面

有了這些幕後工作人員控制著流程，在分工清楚的前提下，前景的演員只要盡守本分，依照模式好好地繪製遊戲畫面就夠了。下列是畫出遊戲畫面的 *DrawBackBitmap* 函式：

```
#0001 void __fastcall TMainForm::DrawBackBitmap()
#0002 {
#0003   Map.DrawTerrMap(FBackBitmap->Canvas); // 繪製地形層
#0004   Map.DrawItemMap(FBackBitmap->Canvas); // 繪製物品層
#0005   FRole.Draw(FBackBitmap->Canvas); // 畫出角色圖案
#0006
#0007   int M = FPlayingTime / 60; // 已花費時間的分鐘數
```

```
#0008    int S = FPlayingTime % 60; // 已花費時間的秒鐘數
#0009
#0010    FBackBitmap->Canvas->Font->Color = clWhite;
#0011    FBackBitmap->Canvas->Font->Name = "FixedSys";
#0012    FBackBitmap->Canvas->Font->Style = TFontStyles() << fsBold;
#0013    FBackBitmap->Canvas->Font->Size = 14;
#0014    FBackBitmap->Canvas->Brush->Style = bsClear;
#0015
#0016    TRect R;
#0017    switch (FGameStatus) {
#0018      case gsTitle:
#0019        // 畫出上面的標題大字及下方的作者名稱
#0020        R = Rect(0, 0, TILE_WIDTH * TILE_NUM_X, TILE_HEIGHT *
#0021          TILE_NUM_Y - FBackBitmap->Canvas->TextHeight("我") / 2);
#0022        DrawText(FBackBitmap->Canvas->Handle, "作者: 陳寬達", - 1, &R,
#0023
#0024          DT_BOTTOM | DT_CENTER | DT_SINGLELINE);
#0025        DrawStatusBox("歡迎光臨 足球番", true);
#0026        break;
#0027
#0028      case gsPlayBack: FBackBitmap->Canvas->TextOut(5, 5,
#0029        Format("過關回顧 (LEVEL %d) ...",
#0030          OPENARRAY(TVarRec, (FLevelNo))));
#0031        break;
#0032
#0033      default:
#0034        // 在右上角顯示時間, 以 XX:XX 的格式顯示
#0035        AnsiString Str = Format("%.2d:%.2d",
#0036          OPENARRAY(TVarRec, (M, S)));
#0037        FBackBitmap->Canvas->TextOut(TILE_WIDTH * TILE_NUM_X -
#0038          FBackBitmap->Canvas->TextWidth(Str) - 5, 5, Str);
#0039
#0040        // 在左上角顯示關卡
#0041        Str = Format("LEVEL %d", OPENARRAY(TVarRec, (FLevelNo)));
#0042        FBackBitmap->Canvas->TextOut(5, 5, Str);
#0043        break;
#0044    }
#0045
#0046    // 這是過關畫面
#0047    if (FGameStatus == gsSuccess)
#0048      DrawStatusBox("哇, 成功了 !!", false);
#0049  }
```

0003 ~ 0005 列分別繪出地形、物品及角色，剩下的程式碼則根據目前的遊戲狀態，在畫面的不同區域畫出標題，時間，關卡及祝賀訊息等等。你會發現只要當初遊戲的狀態區

分的清楚無模糊地帶，顯示畫面的程式邏輯就會簡單的不得了，根據狀態顯示不同的訊息就一切 OK。

嗯，其實已經可以看到遊戲畫面，九牛只差一毛了。還差什麼？原來居十分關鍵地位的使用者輸入處理，不能讓玩家控制的遊戲就不能叫做遊戲，叫 DEMO 版本。:p

處理使用者輸入

第一個步驟，先將 *TMainForm* 的 *KeyPreview* 屬性設定爲 *true*，這樣可以保證不論鍵盤輸入焦點位於哪個控制項上，*MainForm* 本身一定會第一個收到鍵盤事件，並且還可以進行過濾處理，讓控制項本身看不到鍵盤事件哩。接著爲 *TMainForm* 的 *OnKeyDown* 事件撰寫處理函式，以上下左右四個方向鍵來控制角色的移動：

```
#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: FRole.Move(drUp); break; // 向上走
#0008
#0009        case VK_DOWN: FRole.Move(drDown); break; // 向下走
#0010
#0011        case VK_LEFT: FRole.Move(drLeft); break; // 向左走
#0012
#0013        case VK_RIGHT: FRole.Move(drRight); break; // 向右走
#0014
#0015        default: return; // 亂按一通，不理它
#0016      }
#0017
#0018      UpdateView(); // 更新畫面
#0019
#0020      if (CheckFinished()) { // 是否完成任務 ??
#0021        FRole.SavePlayBack(); // 將行動記錄存起來以便重播
#0022        GameStatus = gsSuccess; // 進入過關狀態
#0023      }
#0024    }
#0025  }
```

首先注意只有在遊戲中狀態，鍵盤控制才有效。接下來就簡單啦，檢查按鍵是否爲方向鍵，讓角色往對應的方向移動，如果不是方向鍵就離開，省得麻煩。角色更新後，呼叫 *CheckFinished* 函式檢查是否完成任務，是否已將所有目的地形利用覆蓋用物品掩住了？若是的話，就將行動記錄存起來以便重播，並且進入過關狀態。*CheckFinished* 檢查函式是這樣的：

```
#0001  bool __fastcall TMainForm::CheckFinished()
#0002  {
#0003    // 逐一檢查每個目的地形是否已被覆蓋用物品覆蓋住了 ?
#0004    for (int y = 0; y < TILE_NUM_Y; y++)
#0005      for (int x = 0; x < TILE_NUM_X; x++)
#0006        if (Map.IsTarget[x][y] && !Map.IsSource[x][y])
#0007          return false; // 哦, 有一個目的地形還沒被掩住, 失敗 !!
#0008
#0009    return true; // Yeah, 任務達成
#0010  }
```

哇哈，它成功了，我們也成功了。再補上操作介面上的一些其它功能，遊戲主程式也完成了，迫不及待看看它的執行畫面：

圖 8-16 / 足球番的歡迎畫面

圖 8-17 / 足球番的遊戲中畫面一

圖 8-18 / 足球番的遊戲中畫面二

圖 8-19 / 足球番的過關畫面

圖 8-20 / 足球番的過關回顧畫面

圖片確實單調了些，關卡也是我隨手拉出來的，不要又打我呀，這些不是重點，遊戲寫出來才是重點咩。

你是否已有幾分感覺，撇開技術層面不談，遊戲設計與一般的程式設計其實沒有太大的

差異。只要別因為「撰寫遊戲」這四個字而興奮過頭，好好地訂立企劃，將程式中的類別、型態、模組規劃出來，再一步步慢慢兜，你將發現，遊戲程式的撰寫不但沒有想像中那麼難，所獲得的成就感還不是一般程式比得上的唷。