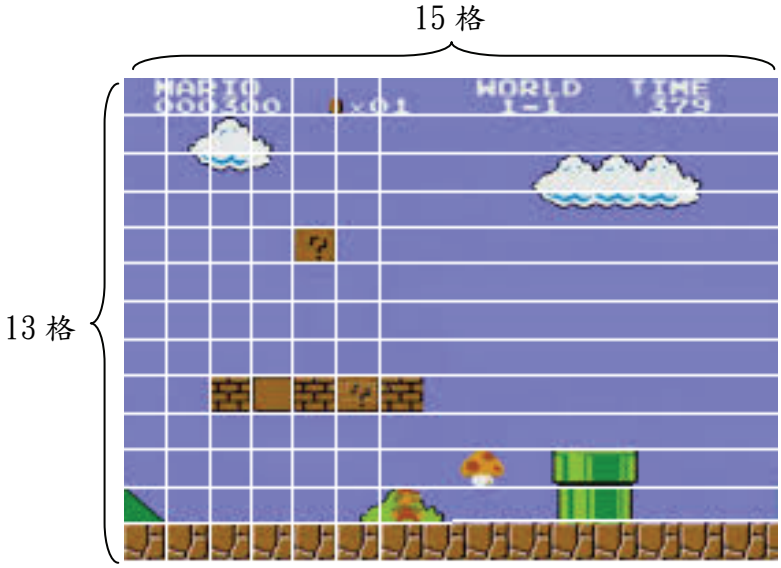
遊戲地圖與座標系統概論

摘要

地圖與座標的處理是 2D 遊戲中很重要的議題。初學者很容易在觀念不清的情形下，把座標相關的程式寫的一團亂，衍生出很多 bug。本文以有系統的方式，介紹地圖與座標系統的關係，釐清觀念，並建議一套程式設計的方法。

一、明顯格狀地圖

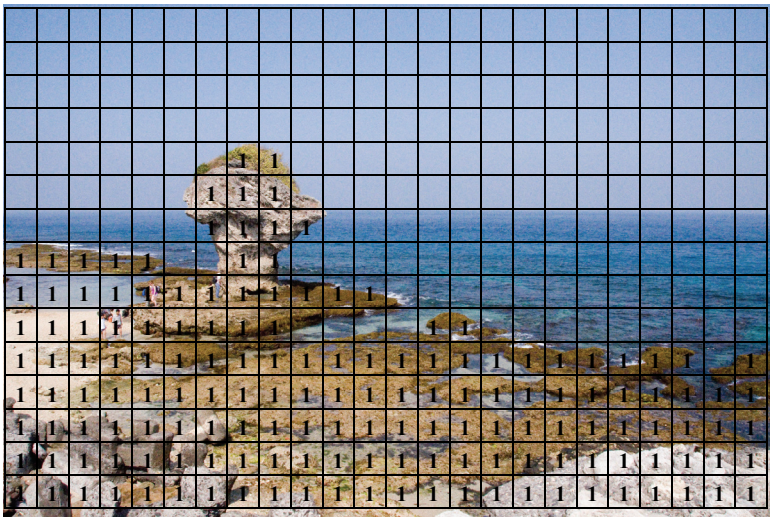
在橫向或直向卷軸的**電腦遊戲裡經常有地形或障礙物**出現，通稱地圖。而**當寫作程式判斷地形或障礙物時，我們需要地形或障礙物的資料**，以便各種角色在地圖中移動時能使用適當的判斷式決定其行為。有些地圖的障礙物**格狀**很明顯、很規則，例如下圖。這一類的地圖的地形或障礙物(磚、地板、水管)可以直接用 **2D 陣列**來表達，例如下圖可以想成是一個 15*13 的陣列，陣列中每一個位置用一個整數編碼來代表對應的障礙物(例如：空氣編為 0、磚編為 1、地板編為 2、水管編為 3)。當使用 2D 陣列時，一個角色的座標只要經過換算(詳述於後)就可以查出他是落在哪一格，知道是否有障礙物，因此，運算很有效率。這種地圖參考 Windows 練習「A sample map class」即可繪出。



二、沒有明顯格狀的地圖

有些遊戲的地圖地形沒有規則，或障礙物沒有明顯的格狀，如下圖。此時用陣列的編碼同時代表障礙物及其圖形並不方便，但是，我們還是可以用 2D 陣列來表達障礙物，

也就是說，把**地圖的繪圖功能和障礙功能分開處理**。例如下圖中有障礙物的位置標示(編碼)為 1，其他位置為 0，繪圖時直接將整張圖貼上去，但是，當計算某個角色是否碰到障礙物時，則查詢 2D 陣列決定是否有障礙物。使用這種方式時，格子的大小決定障礙物位置的準確度，例如下圖整個海岸線是斜的，因此，有些地方標示為 1 並不是很準確，不過，遊戲的顯示往往只要大約對到位置就可以了，因此，通常格子不用取得很細。



三、座標系統

有許多橫向或直向捲軸遊戲，其地圖比螢幕還大，在遊戲進行中，螢幕只能顯示地圖的一小部分。此時，遊戲程式需要一個**統一的座標系統**，才能簡化程式的設計。以下簡介座標系統的表示與處理方法。假設螢幕解析度為 640×480 點，地圖共有 10×6 格，每格佔 160×120 點(為了方便說明，此處的一格是很大的，其實大部分的遊戲，一格都沒有這麼大)，則一個螢幕可以顯示其中的 4×4 格，如下圖所示。

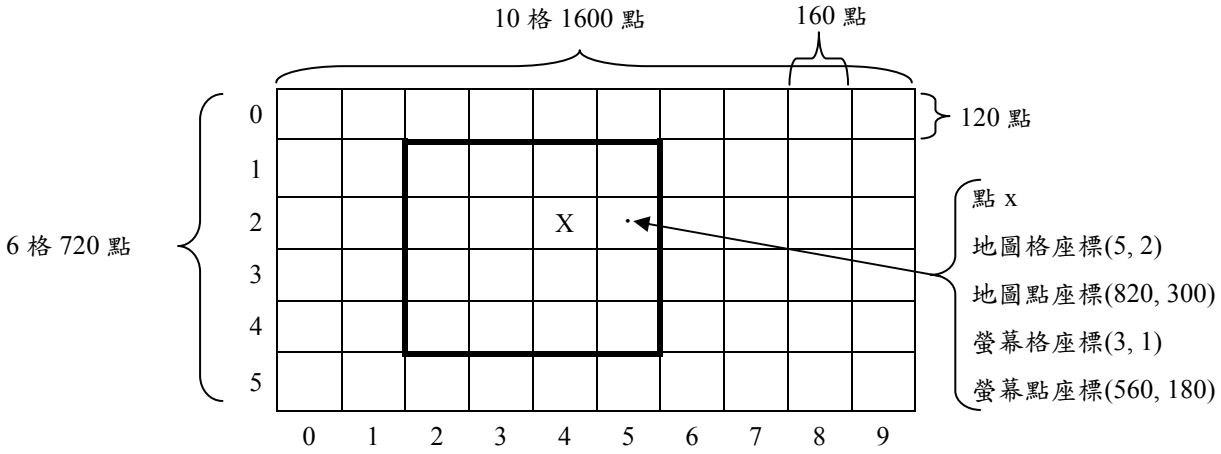


圖 1. 一個比螢幕大的地圖(**粗線**為螢幕看到的範圍)

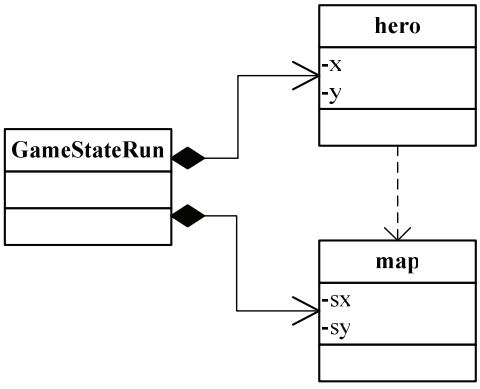
在這裡，「地圖」與「螢幕」是兩個不同的系統，而座標又有兩種：「**格座標**」與「**點座標**」，因此，組合起來總共有四種不同的座標表達方式：

1. 地圖的格座標：如 X 這一格，相對於地圖的左上角，其**格座標**為(4, 2)
2. 地圖的點座標：如 x 這一點，相對於地圖的左上角，其**點座標**為(820, 300)
3. 螢幕的格座標：如 X 這一格，相對於螢幕的左上角，其**格座標**為(2, 1)
4. 螢幕的點座標：如 x 這一點，相對於螢幕的左上角，其**點座標**為(560, 180)

在這四種表達方式中，**最位重要的是「地圖的點座標」**，依據地圖的點座標我們可以推算出「地圖的格座標」，以及「螢幕的點座標」，至於「螢幕的格座標」則大部分遊戲都用不到。

當我們設計座標系統時，有下列問題要考慮：(1) **統一座標**：座標系統必須統一，在地圖上不同位置的不同角色，才能互相追蹤或判斷碰撞，(2)**角色平滑的移動**：我們希望主角(或其他角色)能平滑的移動(以點的方式移動)，而不是一次跳躍一格，(3) **螢幕平滑的移動**：我們希望當主角的位置改變時，螢幕能平滑的跟著移動，而不是一次跳躍一格，(4)**簡明的計算**：我們希望座標的計算與座標系統的換算很簡單，以簡化程式，(5) **唯一座標**：每一個角色，只儲存一種座標，當需要其他座標時，當場換算(不儲存)，因為，當儲存兩套座標時，只要程式有一點點小疏忽，兩套座標就會不同步，而一但座標不同步後，就會產生累計誤差，越錯越多(最常見的現象就是螢幕移到下一區，再回來，反覆幾次，整個畫面就亂掉了)。

在這四種座標表達方式中，**最重要的是「地圖的點座標」**。我們知道除了主角以外，地圖上還有很多敵人(NPC)，通常每一個角色(的 class)都必須儲存自己的座標，以便移動、追蹤，以及判別碰撞時使用。此時，最佳的座標就是地圖的點座標，因為，如果所有的角色都**儲存**(使用)地圖的點座標，那麼**座標系統是一致的**，所以，我們可以直接拿這個座標做追蹤或碰撞的判斷，程式就很簡單。反之，如果，某些角色儲存地圖的點座標，某些又儲存螢幕的點座標，那麼座標的追蹤或碰撞的判斷就會很難處理。接著我們利用下列的 UML class diagram，解釋主角(hero)、地圖(map)與螢幕之間的座標關係。



一般而言，我們可以在主角的 class hero 中宣告主角在地圖上的點座標，例如下列的程式：

```
class hero {
public:
    …
private:
    int x, y;          // (x, y)  為主角在地圖上的點座標
    int sx, sy;        // (sx, sy)  為主角在螢幕上的點座標
```

```
};
```

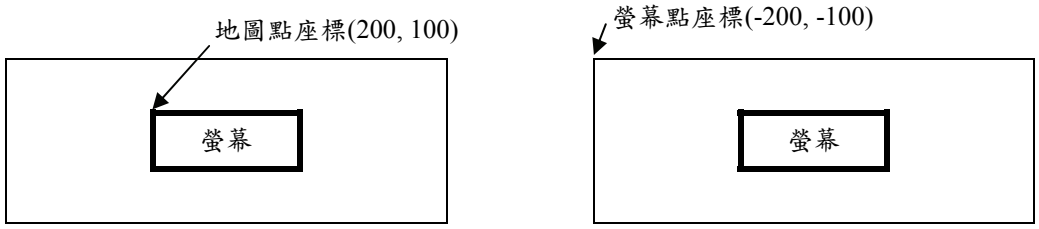
此時，(x, y)座標自(0, 0)起至(1599, 719)為止，都是合法的範圍，換句話說，主角的座標可以是整個地圖上的任一點。如果主角要往右移一點，x+=1 即可，同理，往下移一點，y+=1 即可，因此，**主角的移動，可以處理得很平滑**，主角座標與地圖邊界的判斷也很容易。但是，主角總得顯示在螢幕上，所以，程式必須「算出」主角在螢幕上的點座標，計算方法我們在後面再介紹。但是，請注意，**千萬不要在 hero 中宣告(或儲存) 主角在螢幕上的點座標**，因為，這個座標可以經由換算而得到，存起來有百害而無一利(只有極少數情形下，為了顧慮運算效能，不得不儲存此座標)。

接下來，我們來看螢幕和地圖的關係。假設地圖的 class 是 map，那麼地圖的宣告可能類似下列程式：

```
class map {
public:
    …
private:
    int map[10][6];      // 地圖共有10×6格
    int sx, sy;          // (sx, sy)為螢幕(的左上角)在地圖上的點座標
};
```

在這個宣告中，我們直接把「螢幕(的左上角那一點)在地圖上的點座標」宣告(儲存)在地圖中。由於螢幕的位置是以地圖的點座標方式儲存，所以螢幕與主角都使用相同的座標系統。以圖 1 為例，螢幕左上角剛好在(2, 1)這一格上，其地圖的點座標為(320, 120)，所以 sx 為 320，而 sy 為 120，整個螢幕可以看到的範圍是地圖的(320, 120)至(959, 599)這些點。由於螢幕的位置是以點座標的方式儲存，所以，**螢幕的移動，可以處理得很平滑**，如果螢幕要往右移一點，sx+=1 即可，同理，往下移一點，sy+=1 即可。

那麼，地圖怎麼貼到螢幕上呢？我們必須把**地圖的點座標轉換為螢幕的點座標**，才能顯示地圖。寫程式時，我們可以想像成把整張地圖(的左上角)貼到螢幕的(-sx, -sy)這個點，那麼看起來就相當於是在螢幕上看到地圖上的(sx, sy)到(sx+639, sy+479)範圍了。如下圖所示，假設，sx = 200、sy = 100，那麼，把整張地圖貼到(-200, 100)的螢幕座標就對了，也就是說，把圖 2(a)想像成圖 2(b)，作為顯示。請注意，在這樣的設計下，不論是地圖、螢幕或是主角，我們都不儲存其「螢幕的點座標」，因為，螢幕的點座標是可以算出來的。



(a) 地圖點座標系統：螢幕在地圖上的(200, 100)

(b) 螢幕點座標系統：將地圖顯示在螢幕的(-200,-100)

圖 2. 計算螢幕顯示地圖的哪一部份

地圖顯示的程式很容易寫，例如，map 的 OnShow 可能寫成類似下列的程式。在這個程式中，sx 與 sy 是 map 物件的變數，只要修改 sx 與 sy 就能改變螢幕看到地圖的哪一部份。其實對於捲軸遊戲而言，地圖上大部分的格子在螢幕上都看不到，因此，下列程式的巢狀式 for loop，**可以縮小 loop 的範圍**，改寫成只對「會被顯示」的格子做 ShowBitmap 的動作，以節省 CPU time，這部份的改寫很簡單(當地圖很大時，會很重要)，留給同學們自行處理。

```
void map::OnShow()
{
    for (int i = 0; i < 10; i ++) {
        for (int j = 0; j < 6; j++) {
            int x = i * 160 – sx; //  算出第(i, j)這一格的 x 螢幕座標
            int y = i * 120 – sy; //  算出第(i, j)這一格的 y 螢幕座標
            switch (map[i][j]) {
            case 1:
                xx.SetTopLeft(x, y); //  指定第(i, j)這一格的座標
                xx.ShowBitmap();
            ...
            }
        }
    }
}
```

接下來我們討論主角與地圖的關係。當主角想移動時，必須先判斷移動的方向有沒有障礙物，假設主角在地圖上的點座標為(x, y)，且主角想往右移動，則必須判斷(x+1, y)這個點有沒有障礙物，由於障礙物在地圖上是以格狀的方式存在，所以，我們必須把**地圖的點座標**轉換為**地圖的格座標**，才能判斷是否有障礙物。轉換公式很單純，假設地圖的格座標為(gx, gy)，則(gx, gy) = (x / 160, y / 120)。例如，對於地圖的點座標(160, 120)這個點而言，其地圖格座標為(1, 1)，所以我們只要查(1, 1)這一格是否為障礙物即可。寫程式時，除法運算最好交給地圖負責(因為地圖才知道每格的寬和高有多大)。舉例說，我們可以賦予地圖一個副程式 IsEmpty (回覆是否為空的位置，即不是障礙物)，則主角就可以利用這個副程式決定是否可以移動：

```
bool map::IsEmpty(int x, int y)     // (x, y)  為地圖的點座標
{
    int gx = x / 160;               //  轉換為格座標(整數除法)
    int gy = y / 120;               //  轉換為格座標(整數除法)
    return map[gx][gy] == 0;        //  假設 0 代表空的
}

void hero::OnMove(map *m)           //  注意：這是 hero，不是 map class
{
    if (想往右) {
        if (m->IsEmpty(x+1, y)) {     //  右邊是空的
```

```
            x += 1;
            m->SetSX(...);              // 修正 sx 與 sy，改變螢幕的位置
        } else // 右邊不是空的

            ...
    }
}
```

最後，我們說明主角與螢幕的關係。當主角顯示在螢幕上時，必須先將主角在地圖的點座標轉換為螢幕的點座標。這個轉換也很簡單，如前面的 class 宣告，假設主角在地圖的點座標為(x, y)，螢幕在地圖上的點座標為(sx, sy)，則**主角在螢幕上的點座標**顯然是(x – sx, y – sy)。實作時，座標轉換的責任最好還是交給地圖負責(因為地圖儲存螢幕在地圖上的點座標)，例如下列程式。由於主角在 OnShow 的時候用到地圖(計算螢幕座標)，因此，地圖的 pointer 必須傳給主角的 OnShow。

```
int map::ScreenX(int x)  // x 為地圖的點座標
{
    return x - sx;          // 回傳螢幕的 x 點座標
}

int map::ScreenY(int y)  // y 為地圖的 y 點座標
{
    return x - sy;          // 回傳螢幕的點座標
}

void hero::OnShow(map *m)       // 注意：這是 hero，不是 map class
{
    xx.SetTopLeft(m->ScreenX(x), m->ScreenY(y)); // 轉換為螢幕座標
    xx.ShowBitmap();                                    // 顯示圖形
}
```