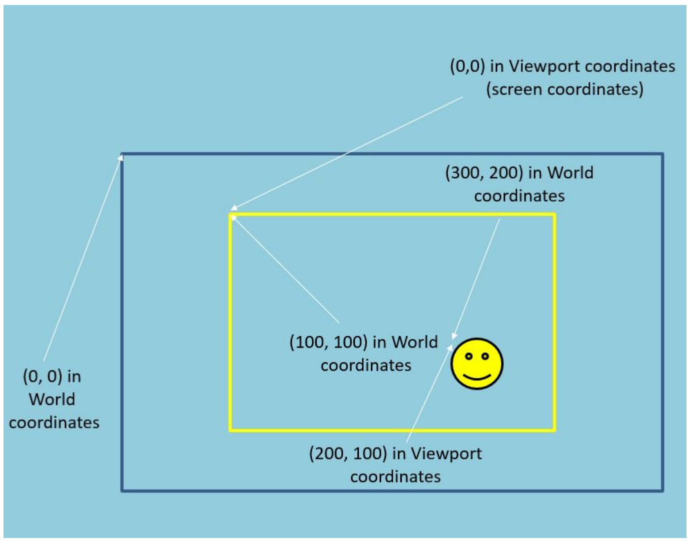
IN628 Programming 4
Semester 1, 2020

IN628 2020 Practical 10 – Manual Scrolling Tile Map

Introduction
Today you will extend your **Tile Map** to permit scrolling. As discussed in lecture, a scrolling background can be viewed as consisting of a **Tile Map** which is bigger than the visible part of the game screen. The game screen itself can be thought of as a **Viewport** window that slides around on the large **Tile Map**.

We will start by building a tiled background that simply scrolls in response to the arrow keys. We will later add a player character **(Sprite)** to the scene, and arrange to keep the **Viewport** window centred on the player character, who will be under arrow key control.

Coordinate systems
When working with a scrolling background, you must always keep in mind two coordinate systems – the world system, which contains the entire **Tile Map**, and the **Viewport** system, which involves only the visible game screen. All elements in the game must have world coordinates. You determine which portion of the world map needs to be displayed in the **Viewport**, and translate all visible elements' world map coordinates to **Viewport** coordinates.



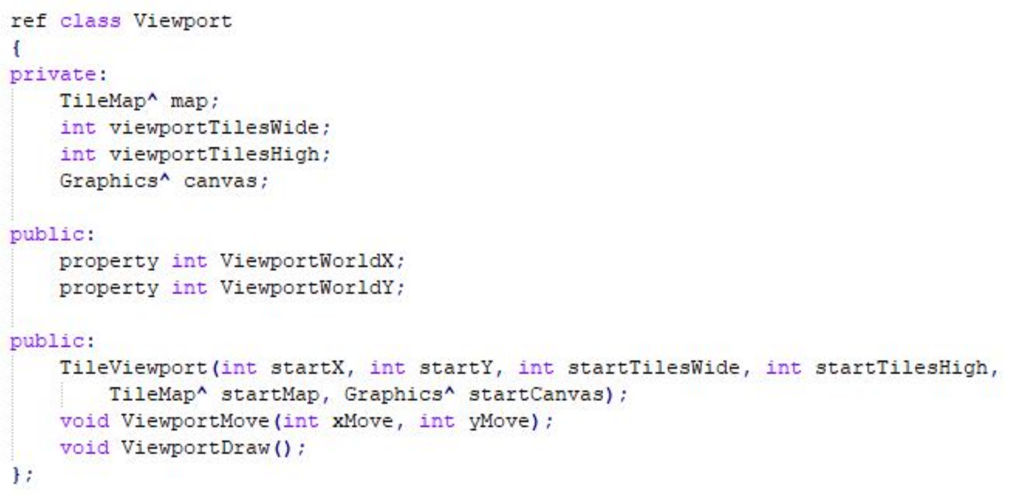
(0,0) in Viewport coordinates (screen coordinates)
(300, 200) in World coordinates
(100, 100) in World coordinates
(0, 0) in World coordinates
(200, 100) in Viewport coordinates

For example, in the figure above, the smiley face is an NPC (i.e. he is not the player character). The Smiley face is at location (300,200) in the world map, i.e. those are his World Coordinates. If the **Viewport's** top left corner is at position (100,100) in the World, the smiley face is 200 pixels to the right and 100 pixels down from that corner. Thus his position in **Viewport** coordinates is (200,100). Since the **Viewport** for us will be the same size as the canvas, the smiley face needs to be drawn at location (200,100) on the canvas. Note that handling the player character is a little different, because we always want to keep him in the center of the screen. We'll deal with the player character in detail later.

Viewport class

We need to define a class to represent the **Viewport**. The **Viewport's** job will be to display the visible area of the world map. The **Viewport** will need to know the world coordinates of its top left corner and its width and height in tiles. It will need to determine which tiles from the world map are in its area, and to draw those tiles at the correct locations on the Form. The **Viewport** will need to make its top left coordinates available to other objects. For example, you might want your **SpriteLists** to access these values to determine which of their sprites to display, and where to display them (cf. the figure above).

A possible architecture for the **Viewport** class is:



```
ref class Viewport
{
private:
    TileMap^ map;
    int viewportTilesWide;
    int viewportTilesHigh;
    Graphics^ canvas;

public:
    property int ViewportWorldX;
    property int ViewportWorldY;

public:
    TileViewport(int startX, int startY, int startTilesWide, int startTilesHigh,
        TileMap^ startMap, Graphics^ startCanvas);
    void ViewportMove(int xMove, int yMove);
    void ViewportDraw();
};
```

Note that the **Viewport** has a reference to the **TileMap** that it is displaying. By accessing **TileMap**, the **Viewport** can determine which images to display on the **Form** (see below for discussion).

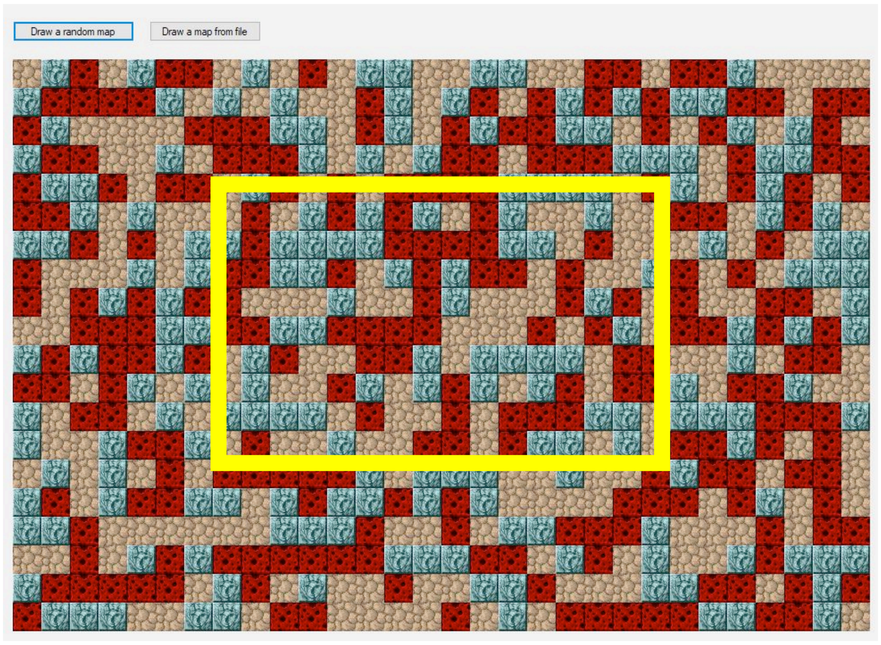
Viewport::ViewportMove(int xMove, int yMove)

The **ViewportMove** method as defined here accepts two parameters: **xMove** and **yMove**. It adjusts the X and Y pixel world coordinates of the **Viewport's** upper left corner by adding **xMove** to the X coordinate and **yMove** to the Y coordinate. The object generating the **ViewportMove** call (it will be the **Form's KeyDown** event handler in the first instance) is responsible for passing positive and negative values in to the method as required. (NB: This is a temporary method. Our technique for controlling the position of the **Viewport** will change when we add a player character to the screen.)

The **Viewport** must not run off the edge of the world. Make sure that you check for this, and adjust the coordinates as required. The top and left limits for the **Viewport** are always 0 (the top and left edges of its canvas). The right and bottom limits are dependent on the size of the world map and the **Viewport**, and must be computed.

Viewport::ViewportDraw()

This is the tricky part of implementing a scrolling map. Consider the figure below:



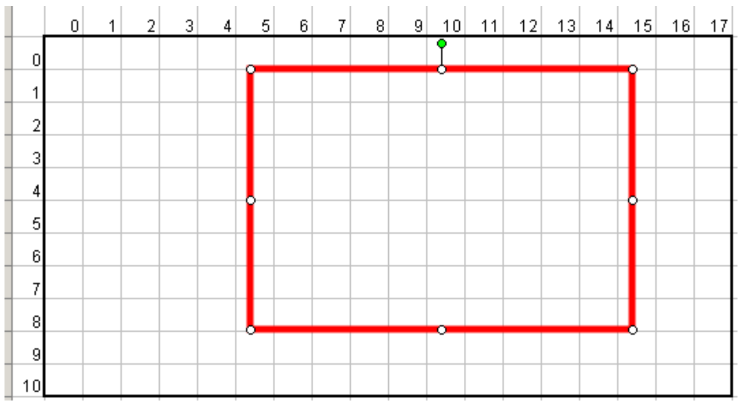
The main image is the world map; the interior square represents the area of the **Viewport**. You want to draw the pixels inside that area to the **Form** (we aren't drawing the sprite yet). You must perform the following steps:

1. Convert the pixel world coordinates of the **Viewport's** upper left corner to Tile Map coordinates, so you know which tile to start with. Starting with that tile, you intend to draw **viewportTilesWide** columns and **viewportTilesHigh** rows of tiles to the canvas.

2. Determine the ***offset*** of the **Viewport's** upper left corner from the edge of the tile boundary. To obtain smooth scrolling, your **Viewport** must move in steps of a size much smaller than the size of an individual tile (the Viewport usually moves only 2 or 3 pixels at each draw cycle). Otherwise your scrolling will jump along in large tile size leaps instead of scrolling smoothly. Therefore, you will often be displaying only a partial tile in the leftmost column and top row. The number of pixels away from the tile edge where you begin drawing is called the offset (see the figure below for more discussion). You must take this offset into account when deciding where on the canvas to draw each tile image.

3. Using a nested for loop, cycle through all the tiles you need to draw. Access the correct image using the **TileMap**, as discussed in lecture. Determine the correct screen location for the tile, and draw it (remember to shift the tiles as needed for the offset computed in step 2). This will usually cause the tiles in the leftmost column and topmost row to be drawn to a negative location, while the tiles in the right most column and bottom most row will be only partially displayed. This is called clipping, and is quite tricky to handle in Windows programming, but not in Visual Studio, which automatically takes care of the mechanics of it for you. You simply need to pass the negative pixel location to the **DrawImage** method call.

To make the offset management process clearer, consider the figure below:



This represents a **Tile Map** that is 18 tiles wide and 11 tiles high. The interior square is the **Viewport**, which is 10 tiles wide and 8 tiles high. You want to draw the area inside the **Viewport** to the screen. Assume each tile is 10 pixels wide.

1. The **pixel** world coordinates of the upper left corner of the **Viewport** are (52, 10). That pixel location is in the **tile** in column 5 and row 1 of the map. That is, the pixel coordinates (52, 10) correspond to a location in tile (5, 1). These values can be obtained by dividing the pixel coordinate by the dimension of the individual tiles. Thus, you can convert from pixel coordinate to map coordinate using this equation (remember that / is integer division in this situation in C++; any remainder is discarded):

   TileCoordinate = PixelCoordinate/tileSide

2. Note that the upper left corner of the Viewport does not lie precisely on a tile boundary, it is "inside" tile (5, 1). When we draw tile (5, 1) to the form, we can't position it precisely at (0, 0) on the canvas. It must be shifted to the left by the offset of the Viewport from the tile boundary (in this case that is 2 pixels along the x-axis). The offset of the pixel world coordinate can be determined by taking the remainder of the pixel coordinate divided by the tile dimension. (For example, to get to pixel 52, you cross 5 complete tiles, and two further pixels. The offset is 2.) This "get the remainder" operation is simply modulo arithmetic. Thus, we can compute:

   TileOffset = PixelCoordinate % tileSide

3. Having computed the starting tile and the offset, you now need to draw the tiles in the following nested for loop (in pseudocode):

   ```
   for c from StartingTileColumn to StartingTileColumn + ViewportTilesWide
       for r from StartingTileRow to StartingTileRow + ViewportTilesHigh
   ```

   The first tile **StartingTileColumn**, **StartingTileRow** needs to be drawn at pixel location (0, 0) on the canvas, minus the x and y offsets. The second tile (going along the first row) needs to be drawn at pixel location (0, TILESIDE), minus the x and y offsets. The third tile (going along the first row) needs to be drawn at (0, TILESIDE * 2) minus the x and y offsets. And so on. Look carefully at this pattern, and you should be able to work out the equations necessary to derive the screen pixel coordinates for each tile using the loop-driving variables.

4. Once you know which tile you are drawing, and where on the canvas to draw it, access the appropriate **TileMap** method to determine the bitmap to draw. (If you don't already have the necessary methods for this in your **TileMap** class, add it now.) Then use **canvas->DrawImage** to paint each tile's bitmap to the target canvas at the appropriate location.