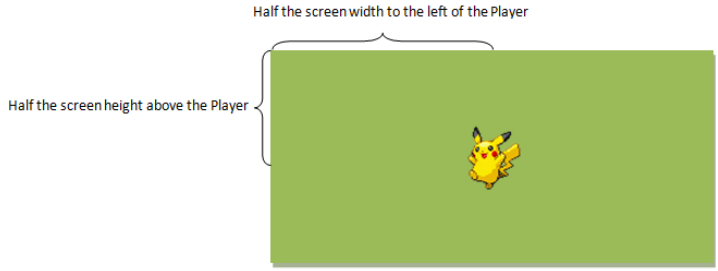
IN628 2020 Practical 11 – Sprite/NPC Scrolling Tile Map 1

Introduction

In today's practical, you will extend the functionality of your **Viewport** class to support games that have a main player character. This will require you to add a method to your **Viewport** class, add a method to your Sprite class, and adjust your game cycle code accordingly.

Centering the Viewport around the player

In a scrolling game, the game world is much bigger than the computer screen. We have termed the visible portion of the world the 'viewport'. In the previous practical, we moved the **Viewport** around manually with the arrow keys. In many games, the **Viewport** is not moved directly. Rather, the player character is controlled with the arrow keys and the viewport is automatically centred around the player. Thus, the game always displays the player in the centre of the screen. To centre the **Viewport** around the player, you simply adjust the **Viewport's** position whenever the character moves. Position the upper left corner of the **Viewport** half the canvas width to the left of the player character, and half the canvas height above the player character. You then draw the viewport exactly as before.



Add a new method to your Viewport class to adjust the Viewport's coordinates relative to the player's world coordinates. A general solution is:

```
void Viewport::MoveRelativeToPlayer(int spriteX, int spriteY)
{
    ViewportWorldX = spriteX - ((viewportTilesWide * 32) / 2);
    ViewportWorldY = spriteY - ((viewportTilesHigh * tileSide) / 2);
    // Do necessary bounds checking to prevent the viewport from running
    // off the edges of the world
}
```

Then modify the code of your KeyDown handler to move the Player (and change the Player character's direction if necessary), rather than moving the Viewport.

Thus, on KeyDown:

1. Move the player (adjust his world coordinates xPos and yPos).
2. Update the player's animation frame counter.
3. Set the viewport's position centred around the player.
4. Draw the viewport to the bufferGraphics.
5. Draw the player at the centre of the bufferGraphics (see below).

6. Draw the bufferImage to the form.

NB: If you place all the drawing code in the KeyDown handler, you will need to press a key to perform the initial draw. You can solve this problem by duplicating your drawing code in the Paint event, but do this cautiously. Paint is raised in response to several different system events, and you should be careful not to produce strange behaviours. We will discuss Paint in detail later; if you don't want to deal with it now, just use the "press a key to start" technique.

Drawing the player at the centre of the buffergraphics

When the world becomes bigger than the screen, we must change the way we draw our sprites. Up until now, the screen and the world have been the same size, so we have simply drawn each sprite at its xPos, yPos world pixel coordinates. But now, a **Sprite's** position in the World may be something like 20,000 while the screen remains no wider than 1024. In our next practical, we will consider how to handle the general case for all sprites. Today we will focus on how to manage the player character.

In the description of the KeyDown functionality given above, you move the player by adjusting his world coordinates (xPos and yPos), as you have always done. As the player moves through the world, his xPos and yPos can become very large; yet he must always appear in the centre of the visible screen, at 512 x 384 (or whatever the centre coordinates of your visible screen are). Thus, we now have two important coordinate pairs: where the Sprite is in the world, and where he must appear on the screen when drawn.

To implement this, your existing **Sprite::Draw()** method is no longer sufficient, because it automatically draws to the **Sprite's** xPos, yPos -- you pass those values into the **canvas->DrawImage** method call. Now you want to be able to force the draw to some other location (in this case, the centre of your screen). You could simply modify your existing **Draw()** to accept a location to draw to instead of defaulting to xPos, yPos. However, when developing a complex class, if we have a method that works, we prefer not to change it, because we might want it again someday. It is better to make a new method to do a new job. This is an important principle of OO development: Add functionality through extension, not through modification. You will revisit this principle often as your programming projects become more complex.

In this case, therefore, we add a new method to our **Sprite** class called **ForcedDraw** that accepts a location (two integers or a point) and draws the Sprite at that location on the canvas. The code for this method is extremely like the code for your existing Draw. (This may feel like bad code duplication. Don't worry about that now, we will fix it later.)

Add **Sprite::ForcedDraw(int forcedX, int forcedY)** to your Sprite class and use that method in your KeyDown handler for today's practical.

Testing your new functionality

Build an application that demonstrates player-centred scrolling. The map and images from the Resources folder, or you can use your own.