# Building a simple Tilemap game.

By Filmote

The following article builds on a sample program that @Hanksi published to show how the Tilemap class can be used. His code was written in Python but I have changed it to my preferred language, C++, to provide a corresponding resource in that language.

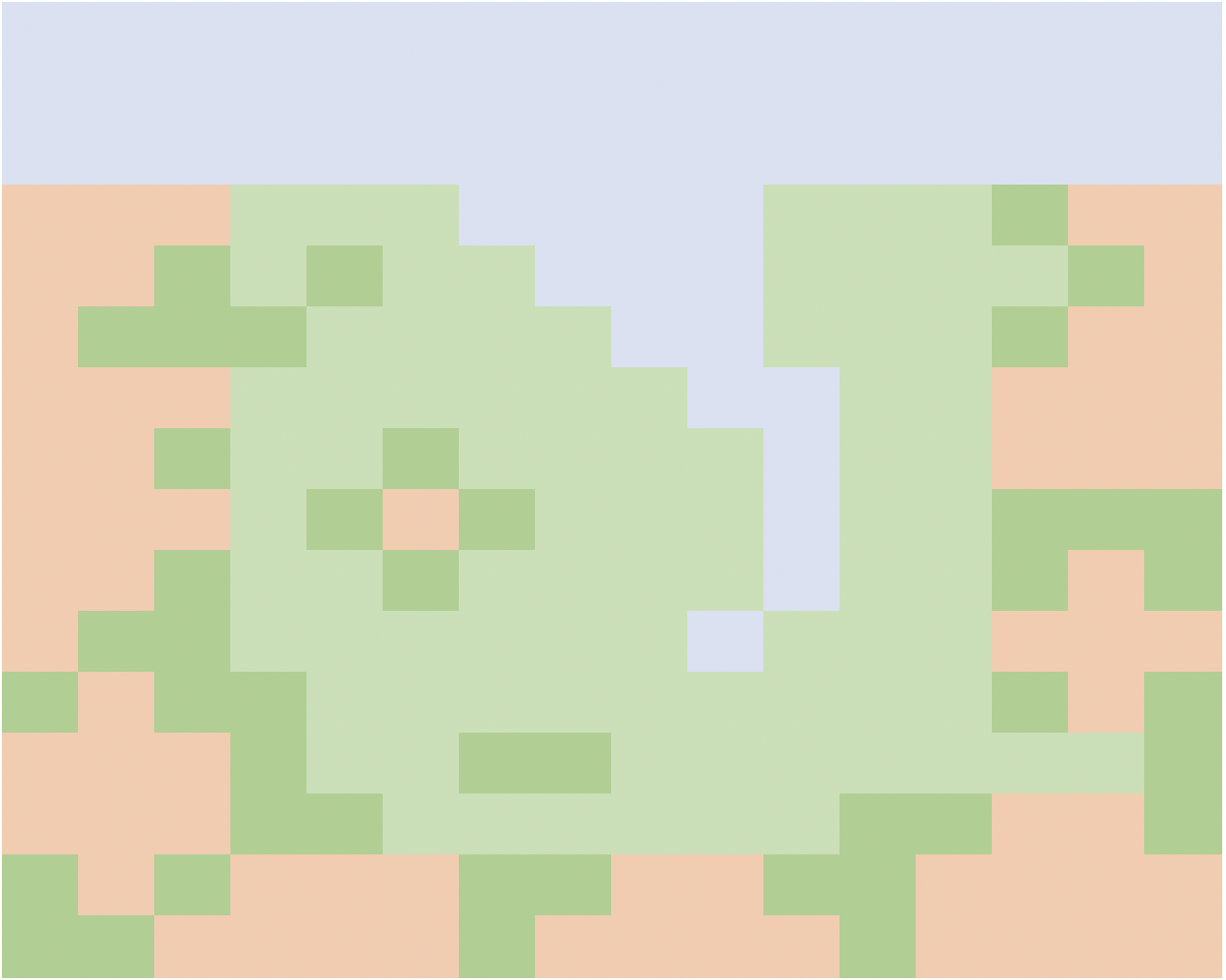
This tutorial is not necessarily trying to teach people to program in C++ and assumes you are familiar with the basics. The code is relatively simple so even if you are comping from a different language you will find the code easy to understand.

# What is a Tilemap?

Tilemaps are a very common programming technique for creating 2D worlds out of reusable tiles. Typically, the world is very large and the player can only see a small section at a time as they move around. Using common tiles allows for extremely large worlds to be made on devices that have surprising little memory. The Pokitto API includes a small library that makes building a game based on a tile map easy!

# Defining the World

Take a look at the world map below that is featured in Hanski’s Python sample program. It consists of a world that is 16 tiles wide and 16 tiles high and utilises only 4 different tiles representing a green area that can be walked on and three obstacles - a bushy grass area, a tree and water.



The world itself is simply an array or values that tell the tilemap class what tile to render in each of the 16 x 16 locations. Each tile is assigned a number and, in the sample application, these are captured in an enumeration as shown below.

enum TileType {

Water = 0,

Green = 1,

Tree = 2,

Grass = 3,

};

To save memory, the current tilemap implementation limits the number of tiles to 16 and takes advantage of this by compressing two tiles into a single byte within the array. The array for our world is shown below alongside our world map. The highlighted cell shows a value of 0x21 corresponds to the area highlighted in the world map. As you can see the two nibbles (that’s what half a byte is called corresponds to a tree and a green tile).

const unsigned char mapPixels[] = {

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x22, 0x21, 0x11, 0x00, 0x00, 0x11, 0x13, 0x22,

0x22, 0x31, 0x31, 0x10, 0x00, 0x11, 0x11, 0x32,

0x23, 0x33, 0x11, 0x11, 0x00, 0x11, 0x13, 0x22,

0x22, 0x21, 0x11, 0x11, 0x10, 0x01, 0x12, 0x22,

0x22, 0x31, 0x13, 0x11, 0x11, 0x01, 0x12, 0x22,

0x22, 0x21, 0x32, 0x31, 0x11, 0x01, 0x13, 0x33,

0x22, 0x31, 0x13, 0x11, 0x11, 0x01, 0x13, 0x23,

0x23, 0x31, 0x11, 0x11, 0x10, 0x11, 0x12, 0x22,

0x32, 0x33, 0x11, 0x11, 0x11, 0x11, 0x13, 0x23,

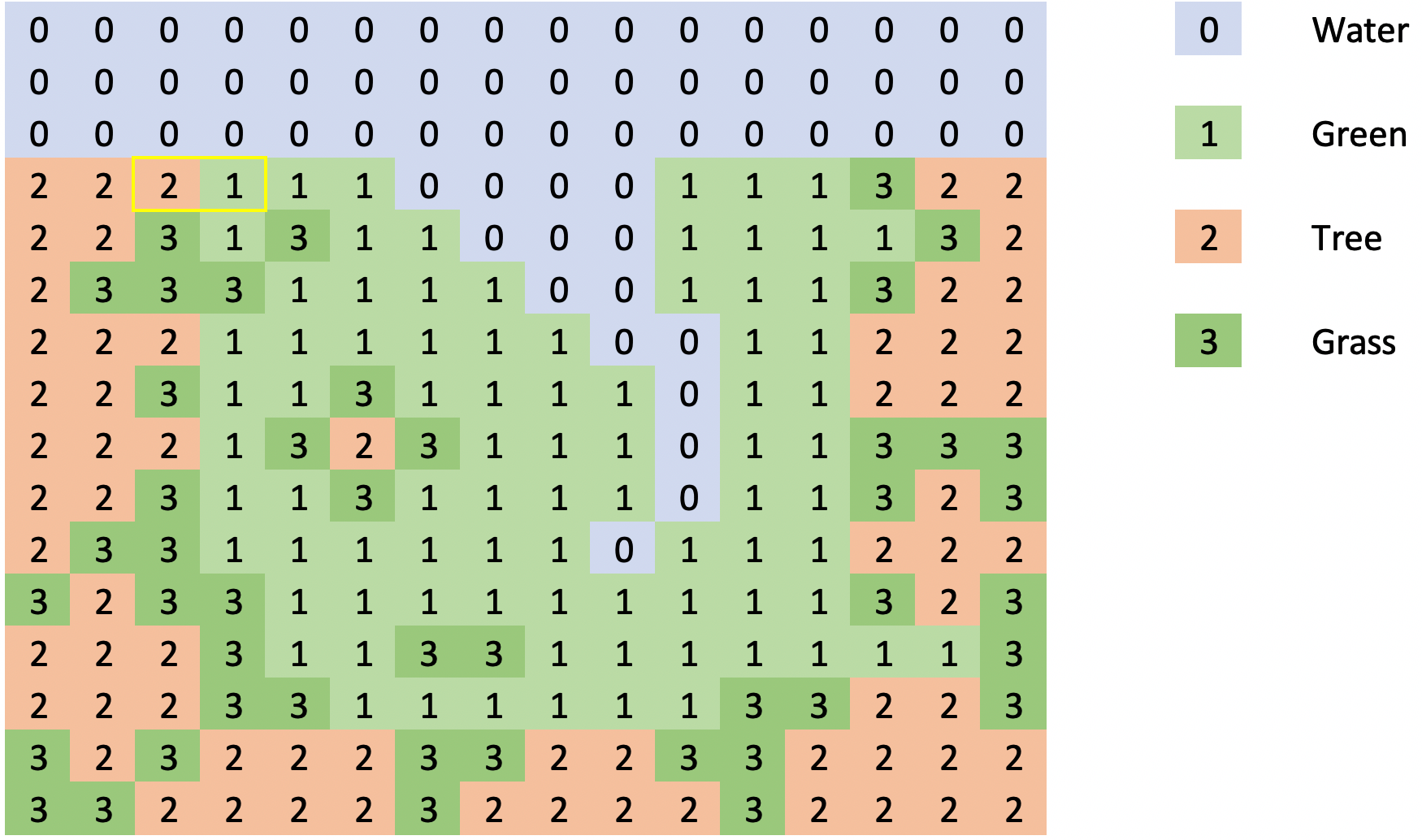
0x22, 0x23, 0x11, 0x33, 0x11, 0x11, 0x11, 0x13,

0x22, 0x23, 0x31, 0x11, 0x11, 0x13, 0x32, 0x23,

0x32, 0x32, 0x22, 0x33, 0x22, 0x33, 0x22, 0x22,

0x33, 0x22, 0x22, 0x32, 0x22, 0x23, 0x22, 0x22,

};



Using this simple compression method, the array used to store our 16 x 16 tile world is only 128 bytes in size. It is not hard to imagine that a game with a single large world or multiple levels stored as multiple arrays or even an array of arrays! The Pokitto has enough memory to support a large universe.

Download and review the code in the repo <https://github.com/filmote/Tilemap_1>

This is a literal translation of @hanski’s original python version with minor changes for clarity. In addition to the enumeration, defined above, the code to initialise the tilemap class is shown below. The **tilemap.set()** command configures the tilemap object with size of the world (16 tiles wide by 16 tiles high) and the data array to use, **Data::mapPixels**.

The remaining four lines of the code assign the map data indexes to the images to be rendered. Using the enumeration values rather than literals makes the code much more readable.

enum TileType {

Water = 0,

Green = 1,

Tree = 2,

Grass = 3,

};

Tilemap tilemap;

int main(){

…

tilemap.set(16, 16, Data::mapPixels);

tilemap.tiles[TileType::Green] = Data::green16;

tilemap.tiles[TileType::Tree] = Data::tree16;

tilemap.tiles[TileType::Grass] = Data::grass16;

tilemap.tiles[TileType::Water] = Data::water16;

…

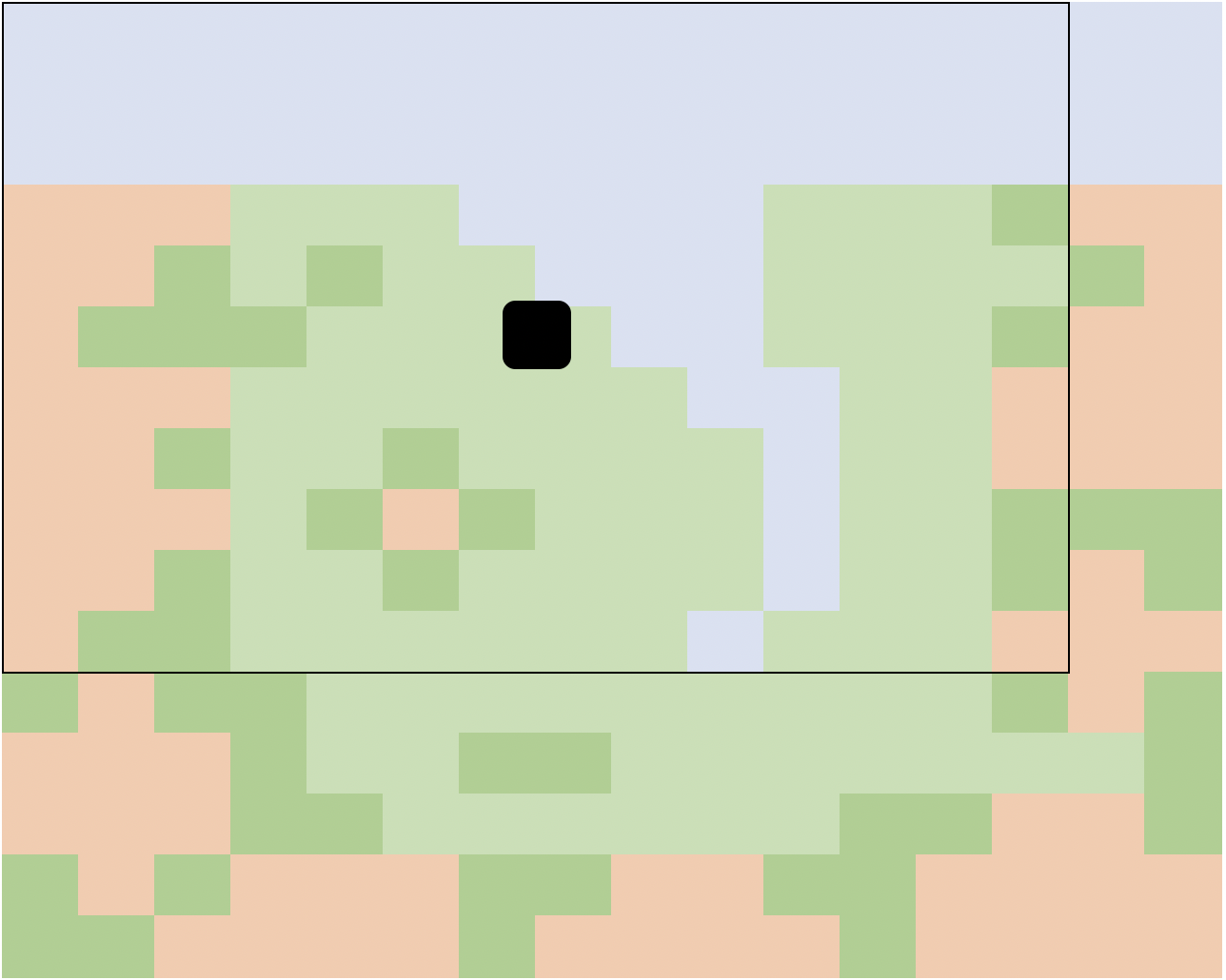
When we are ready to render our viewport, we simply call **tilemap.draw(x, y);**

# Moving a player around the world.

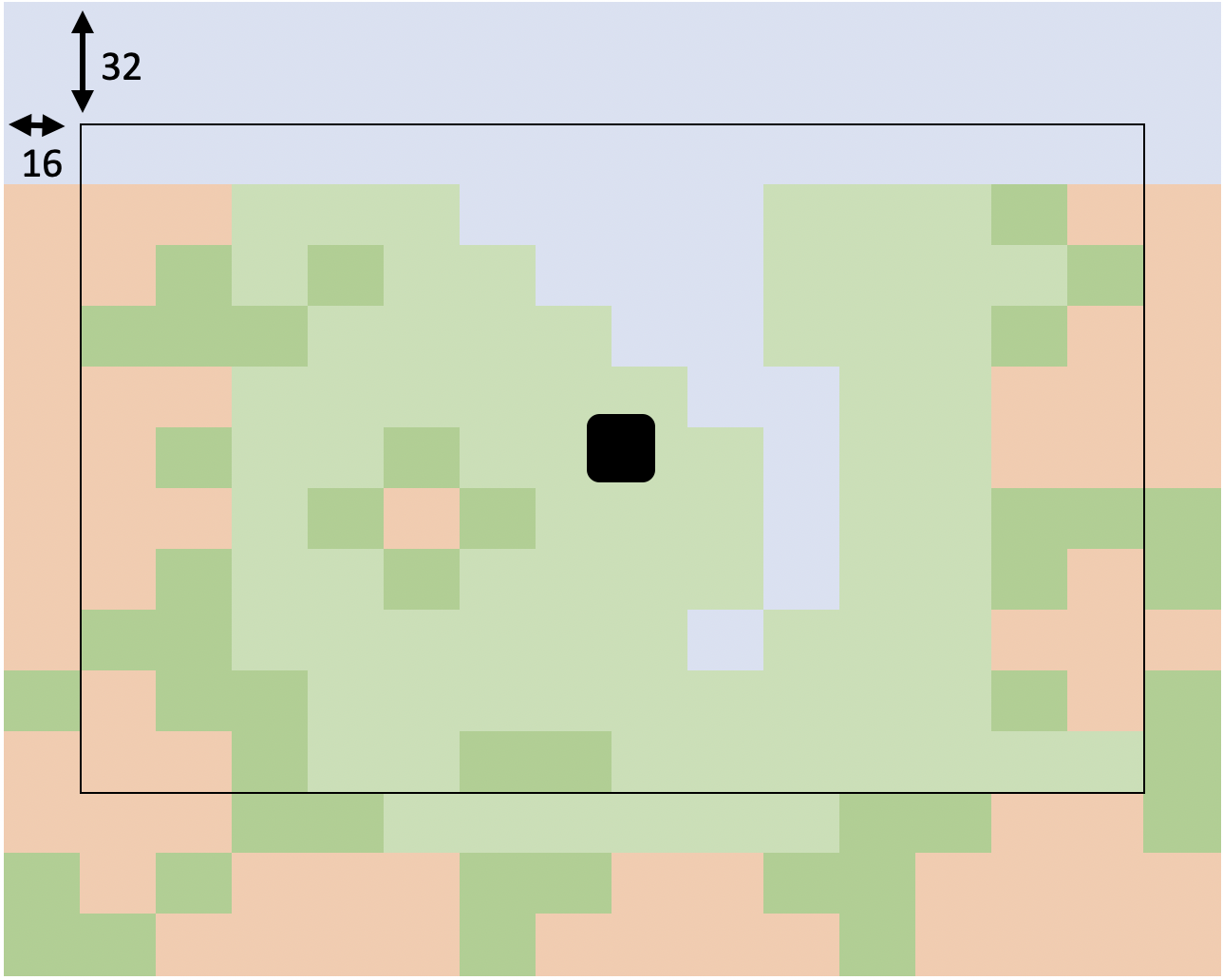
Now that we have defined our world, we can add the player and allow him to explore.

Previously we mentioned that our world is 16 x 16 tiles in size. If our tiles are also 16 x 16 pixels in size then we can calculate that our world is 256 pixels wide by 256 pixels in height. Using mode 15 on the Pokitto allows us to display 220 x 176 pixels of that world. As such we can only show a portion of the world and will need to scroll this view as the player moves. The visible section of the world is known as the ‘viewport’.

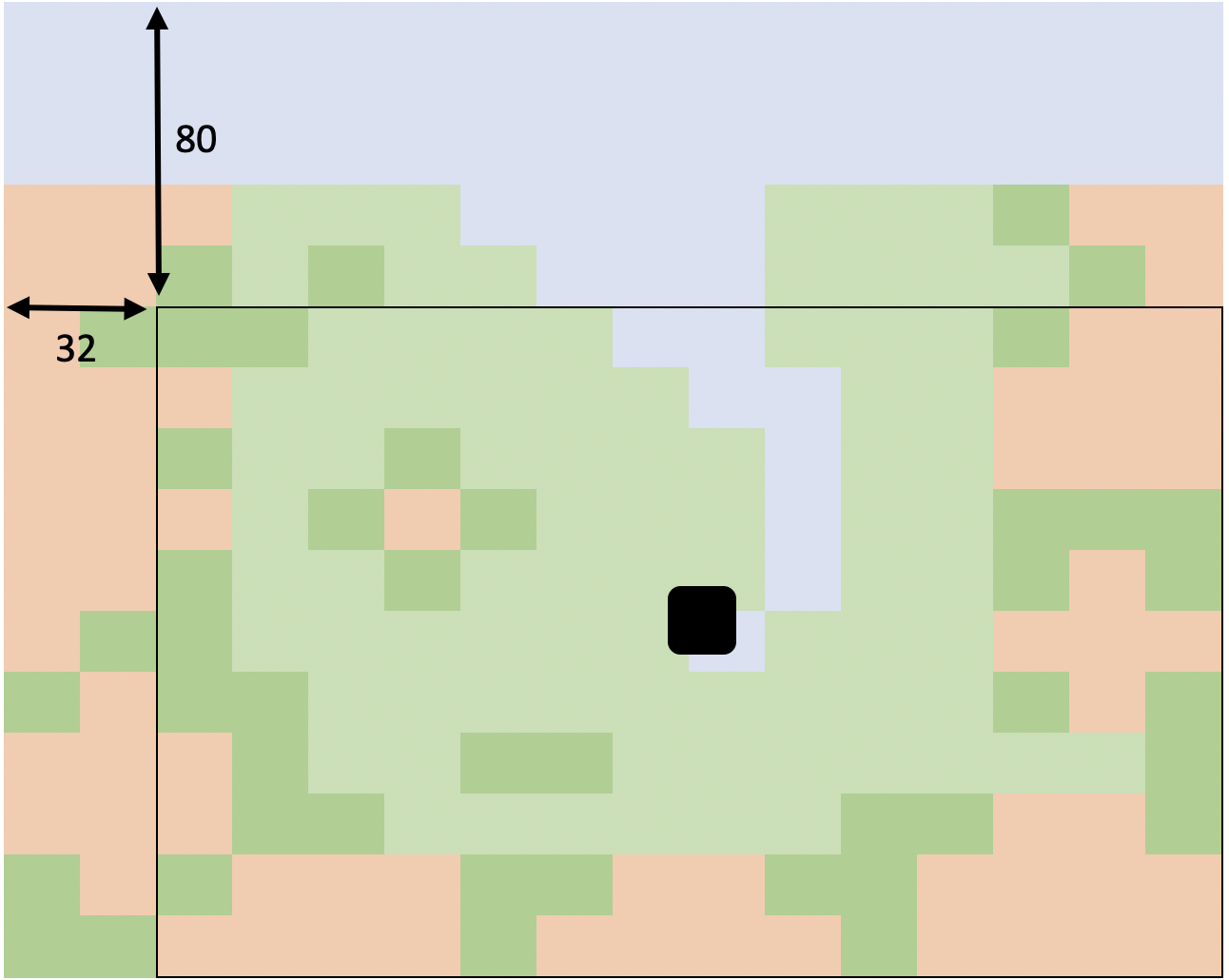
Imagine the player starts near the top-left of the world. When rendering the screen, we typically render the player in the middle of the viewport and hence the middle of the screen. When the viewport is positioned in the top, left-hand corner, as shown below, **the world has an offset of x = 0, y = 0 relative to the viewport.**



Now let’s assume the player moves diagonally down towards the bottom, right-hand corner. The viewport moves along with the player so that when rendering the player is still in the centre. At this stage, **the viewport has on offset of x = 16, y = 32 relative to the top-left corner of the world.**



Finally, the player reaches a point where the viewport is now touching the right-hand and bottom edges of the world. The viewport offsets (x = 32 and y = 80) can be calculated as the difference in size between the world and the size of the viewport. The world’s width is 256 pixels wide and the viewport is only 220 pixels resulting in a difference of 32 pixels. Likewise, the height of our world, 256 pixels, is 80 pixels larger than our viewport height, 176 pixels.



Download and review the code in the repo <https://github.com/filmote/Tilemap_1> if you haven’t already.

The code below shows how the pressing of the directional buttons on the Pokitto will control the movement of the viewport. Pressing the left button will decrease the player’s x position and pressing right will increase it.

int16\_t x = -20;

int16\_t y = -50;

while (PC::isRunning()) {

...

if (PC::buttons.pressed(BTN\_LEFT) || PC::buttons.repeat(BTN\_LEFT, 1)) {

x = x + 1;

}

if (PC::buttons.pressed(BTN\_RIGHT) || PC::buttons.repeat(BTN\_RIGHT, 1)) {

x = x + 1;

}

if (PC::buttons.pressed(BTN\_UP) || PC::buttons.repeat(BTN\_UP, 1)) {

y = y - 1;

}

if (PC::buttons.pressed(BTN\_DOWN) || PC::buttons.repeat(BTN\_DOWN, 1)) {

y = y + 1;

}

...

tilemap.draw(x, y);

PD::drawBitmapData(centreScreenX, centreScreenY,

12, 15, Data::girl12x15Pixels);

}

A simple check of the player coordinates ensures that the player is unable to move outside of the world. As mentioned earlier, **worldHeight** and **worldWidth** describe the dimensions in pixel and the two ‘magic’ numbers 12 and 15 are the players width and height, respectively.

if (x < 0) x = 0;

if (x > worldWidth - 12) x = worldWidth;

if (y < 0) y = 0;

if (y > worldHeight - 15) y = worldHeight;

# Rendering the Viewport and Player

From the player’s coordinates, we can calculate the coordinates of the view port. Typically, the player will be rendered in the centre of the screen and the viewport will show the world relative to the player’s position in the world.

A function for calculating the view port’s coordinates is shown below. The calculation for the **X** and Y values are similar and divided into three tests – is the player within one half the width of the view port of the left-hand side of the world in which case the viewport should be rendered alongside the world’s left-hand side (**x = 0**).

If not, is the player within one half the width of the view port of the right-hand side of the world? If so, then the viewport should be rendered alongside the right-hand side of the world (**x = worldWidth – PD::width**).

If neither of these are true, the viewport should be rendered relative to the player (**x = player’s X position – PD::width / 2**).

void calculateViewPortPosition(int16\_t &xViewPort, int16\_t &yViewPort) {

// Calculate the x position ..

if (x < PD::width / 2) {

xViewPort = 0;

}

else if (x > worldWidth - PD::width / 2) {

xViewPort = worldWidth - PD::width;

}

else {

xViewPort = x - PD::width / 2;

}

…

}

Using this function, we can calculate the viewport’s offset and use this when calling the **Tilemap** classes **draw()** function.

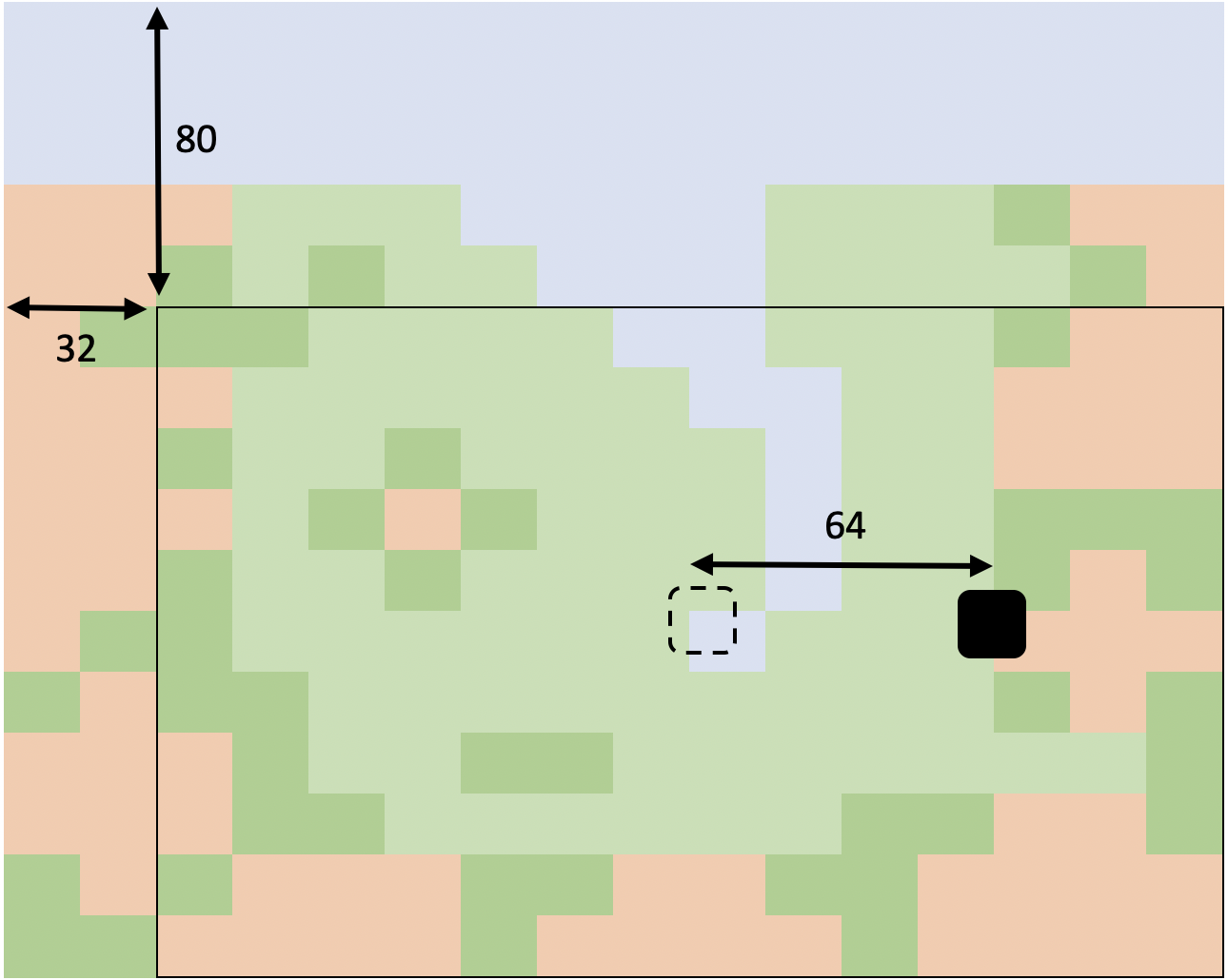
int16\_t xViewPort;

int16\_t yViewPort;

calculateViewPortPosition(xViewPort, yViewPort);

tilemap.draw(xViewPort, yViewPort);

So far the scenarios shown have not included the case where a player is moving to an edge of the world. In the image below, the player has moved to the right and the viewport is now hard up against the right-hand side of the world. If the player was to move further right, we cannot move the viewport any more so to compensate, we render the player right of the viewport’s centre line.



Download and review the code in the repo <https://github.com/filmote/Tilemap_1> It contains a function, similar to the **calculateViewportPosition()** above, which determines the coordinates at which to render the player. The calculation for the **X** and Y values are similar and divided into three tests – is the player within one half the width of the view port of the left-hand side of the world in which case the player should be rendered alongside the world’s left-hand side (**x = player’s x position**).

If not, is the player within one half the width of the view port of the right-hand side of the world? If so, then the player should be rendered at the current x position less the difference in size between the worlds and viewport’s width (**x = player’s x position – (worldWidth – PD::Width)**).

If neither of these are true, the player should be rendered in the centre of the screen (**x = PD::width / 2**).

void calculatePlayerPosition(int16\_t &xPlayer, int16\_t &yPlayer) {

if (x < PD::width / 2) {

xPlayer = x;

}

else if (x > worldWidth - PD::width / 2) {

xPlayer = x - (worldWidth - PD::width);

}

else {

xPlayer = PD::width / 2;

}

…

}

Review the omitted code from the **calculateViewportPosition()** and **calculatePlayerPosition()** functions. The two functions are similar in structure and the calculation of the Y coordinate in both is nearly identical to the calculation of the X coordinate.

# Detecting Obstacles

Download the code in the repo <https://github.com/filmote/Tilemap_1> if you haven’t already.

If you run this code, you will notice that you can only walk on the plain green tiles. This is done by capturing the current location of the player before any move and then detecting the tile the player may have just moved on to. If it is not a green tile, the player is moved back to their original position. As this is all done prior to rendering the screen, the two movements are not visible.

This code is shown below.

// Capture the old coordinates in case we need to move back ..

int16\_t oldX = x;

int16\_t oldY = y;

… handle player movements.

// Check the map tile under the player.

uint8\_t tileId = tilemap.GetTileId(x + 6, y + 7, 16);

// If the tile is not green, do not move.

if (tileId != TileType::Green) {

x = oldX;

y = oldY;

}

Although this code works well, it uses the centre of the player to detect the tile underneath which allows the player to move half-way onto a tree or water before being stopped. To prevent this, we need to check the tile below the left side of the player if we are moving left, the right-side of the player if we are moving right and so on. In a later section of this tutorial I want to add enemies to the world and want to share the movement logic between the player and the enemy. Externalising the logic into a function will support both requirements.

Download the code in the repo <https://github.com/filmote/Tilemap_2> and run it. You will notice that the player cannot ‘half’ move onto a tree or the water.

The code below shows how to detect whether a player movement is possible **before** the movement is made. But before we look at the code, you may have noticed that I introduced two new constructs into the code – an **enum** and a **struct**ure.

Previously we used an **enum** to define the tile types which made our code more readable. Using this same logic, I have created an **enum** that defines the four direction of movements. We will use these later when testing whether a movement is ‘legal’.

enum Direction {

Up,

Down,

Left,

Right

}

The **Entity** structure is used to capture our player and viewport offset details and allows us to pass them as a whole to our **checkMovement()** function. In addition to the offsets, the structure also contains some constant information describing the player size and can be extended later to capture inventory or health details.

struct Entity {

int16\_t x;

int16\_t y;

const uint8\_t width = 12;

const uint8\_t height = 15;

};

Using the **enum** and **struct** above, we can now construct a function that checks whether a move is valid or not. This is partially repeated below. The first thing you will notice is the parameters that are passed – a reference to the player entity, the x & y position the player is moving to and the direction they are moving. I could easily have not passed the reference to the entity as it is already a global variable but in the next section you will see that I can pass an enemy reference to the same function!

Depending on the direction we nominate, the tiles that the player will move onto are calculated. As our player could be straddling two world tiles, they are both retrieved for evaluation. When moving left or right, the tile at the nominated **x** and **y** position and the tile immediately below this must be considered. If you are moving up or down, the tile immediately to the right must be considered.

Once the tiles are determined, they are checked to ensure the player can move and, if so, the function returns a **true** otherwise it returns a **false**. This logic could be extended to include other tiles - doors, portals etc – in a multi-level game.

bool checkMovement(Entity &entity, int16\_t x, int16\_t y, Direction direction) {

int8\_t tileId1 = 0;

int8\_t tileId2 = 0;

switch (direction) {

case Direction::Left:

tileId1 = tilemap.GetTileId(x, y, 16);

tileId2 = tilemap.GetTileId(x, y + entity.height, 16);

break;

case Direction::Right:

tileId1 = tilemap.GetTileId(x + entity.width, y, 16);

tileId2 = tilemap.GetTileId(x + entity.width, y + entity.height, 16);

break;

case Direction::Up:

…

case Direction::Down:

…

}

// If either tile is not green, do not move.

if (tileId1 != TileType::Green || tileId2 != TileType::Green) {

return false;

}

return true;

}

Now that we have a function which tells us whether a move is valid or not, we can condition the player movement logic shown previously.

if (PC::buttons.pressed(BTN\_LEFT) || PC::buttons.repeat(BTN\_LEFT, 1)) {

// Can we move to the left?

if (player.xOffset < 0 &&

checkMovement(player, player.x - 1, player.y, Direction::Left)) {

player.x--;

}

}

# Adding Enemies

Now the fun part – adding enemies!

Download the code in the repo <https://github.com/filmote/Tilemap_4> and run it. As you can see, there are now three enemies who will chase you around the small world – if they touch you its game over. You will also notice that the enemies can only move on the green tiles just like the player. The following code shows how to use the previously defined **checkMovement()** function to test enemy movement as well.

But before we get there, let’s have a quick look at the changes I have made to the **entity** class we used previously. In the code below, we have a base struct called **Entity** and two separate classes, **Player** and **Enemy**, that extend it. This simple model allows us to have specific methods for the player class to assist in calculating positions within the world for collision detection and so forth. Currently the **Enemy** and **Player** struct simply extend the base **Entity** class but these could be altered later to capture details such as health, inventory, etc.

struct Entity {

int16\_t x;

int16\_t y;

const uint8\_t width = 12;

const uint8\_t height = 15;

};

struct Player : Entity { };

struct Enemy : Entity { };

Finally, we can declare our player and array of enemies:

Player player;

Enemy enemies[Constants::numberOfEnemies];

In this simple example game, the enemies will try to move towards the player but at the same time avoiding obstacles they cannot move through. This logic is encapsulated in the function shown below.

The position of the player is compared to the current location of each enemy in the array. If the player is to the left of the enemy and the enemy can move in that direction (determined using the existing **checkMovement()** function) then the player is moved. Depending on the position of the enemy relative to the player, the enemy may move both horizontally and vertically.

The code has been abbreviated and only shows the logic for the left movement. The three other directions contain similar logic and can be reviewed in the source code downloaded earlier.

void handleEnemyMovements() {

// Move each enemy individually ..

for (uint8\_t i = 0; i < Constants::numberOfEnemies; i++) {

// Move left?

if (player.x < enemies[i].x) {

if (checkMovement(enemies[i], enemies[i].x - 1, enemies[i].y,

Direction::Left)) {

enemies[i].x--;

}

}

…

}

}

Finally, detecting a collision between an enemy and player is achieved by testing whether the player ‘rectangle’ overlaps the enemy ‘rectangle’ in any way. This logic for this is shown below ..

bool collide(Player player, Enemy enemy) {

return !(enemy.x >= player.x() + player.width ||

enemy.x + enemy.width <= player.x() ||

enemy.y >= player.y() + player.height ||

enemy.y + enemy.height <= player.y());

}

# But I want to use FemtoIDE and Piskel to create graphics!

While browsing the code, you may have noticed that the graphics images used are just the pixel data and do not have embedded dimensions. If you have used FemtoIDE and have created graphics within it, you are probably familiar with its process of converting images into **.h** files for immediate inclusion into a program.

These image also allow you to render the images using the overloaded method **drawBitmap(x, y, image)** where you do not need to specify the dimensions.

If you download the code in the repo <https://github.com/filmote/Tilemap_4> you will see how I have used the image **.h** files from the FemtoIDE conversion process to populate the tile data. The magic happens in the images/Images.h class where I have removed the embedded data from previous projects and included the .h files.

Tile definitions expect a reference to the image data (without embedded dimensions) and width and height parameters. When creating the tile definition using embedded dimensions, I have pointed it to the 2nd element of the image data which is the starting point of the pixel data. When specifying the width and height, I have referred to the 0th and 1st elements to retrieve the embedded dimensions.

#include "Green.h"

#include "Grass.h"

#include "Tree.h"

#include "Water.h"

Tilemap::Tile green16 = { &Green[2], Green[0], Green[1] };

Tilemap::Tile tree16 = { &Tree[2], Tree[0], Tree[1] };

Tilemap::Tile grass16 = { &Grass[2], Grass[0], Grass[1] };

Tilemap::Tile water16 = { &Water[2], Water[0], Water[1] };

# Finally ..

The current implementation of the **TileMap** class is optimised to speed up rendering of tiles in screen mode 2 which has 16 colours and a resolution of 110 x 88. My sample code is running using screen mode 15 which also has 16 colours but a higher resolution of 220 x 176.

You can switch between mode 2 and mode 15 by editing the **MySettings.h** file. If you ‘clean’ and recompile the code, you will see that it works in both modes.

You have probably noticed that the rendering is a little slow in mode 15 but this can be improved dramatically by one simple little change in the Pokitto library. If you open up the file named Tilemap.hpp in the /Pokitto/POKITTO\_LIBS/Tilemap directory, you will see a line 26 the following code:

#if POK\_SCREENMODE != MODE\_FAST\_16COLOR

Changing the line to the following enables faster rendering on mode 15 as well.

#if POK\_SCREENMODE != MODE\_FAST\_16COLOR && POK\_SCREENMODE != MODE15

Save the file, recompile the game and note the speed improvements! This modification is likely to be merged into the base library very soon so you may have found it has already been done by the time you are reading this.