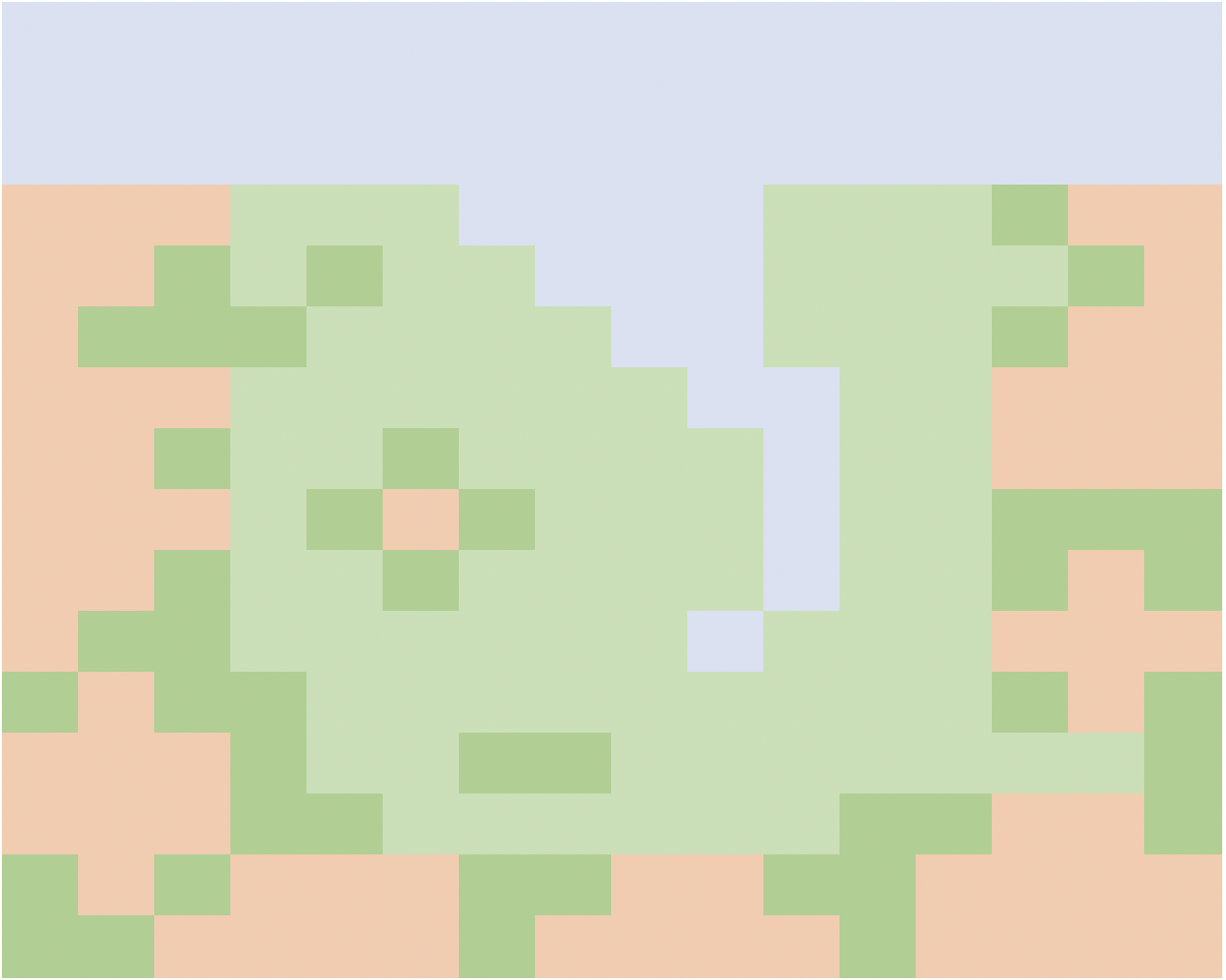
Reference Hanski project.

# 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,

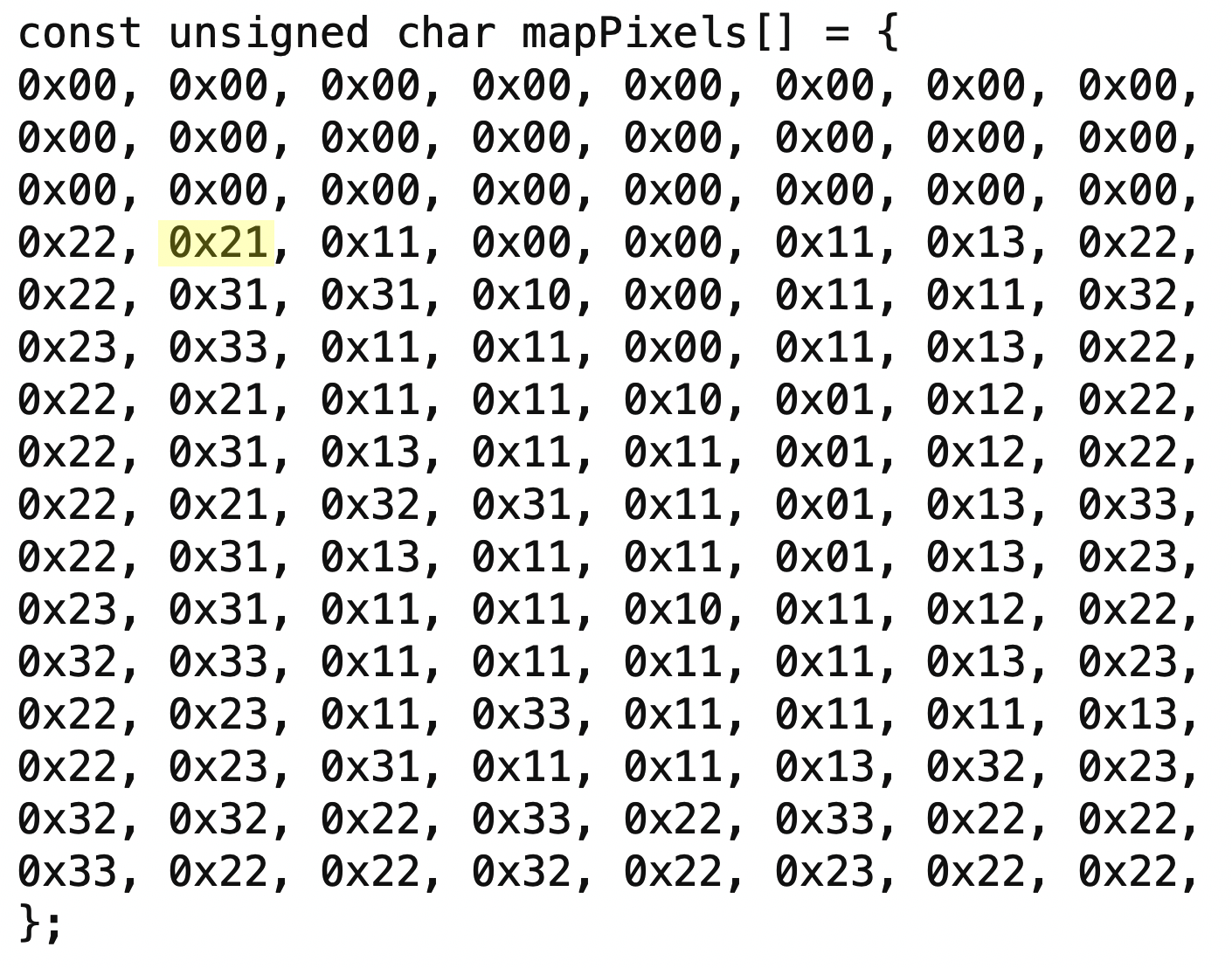
Green = 1,

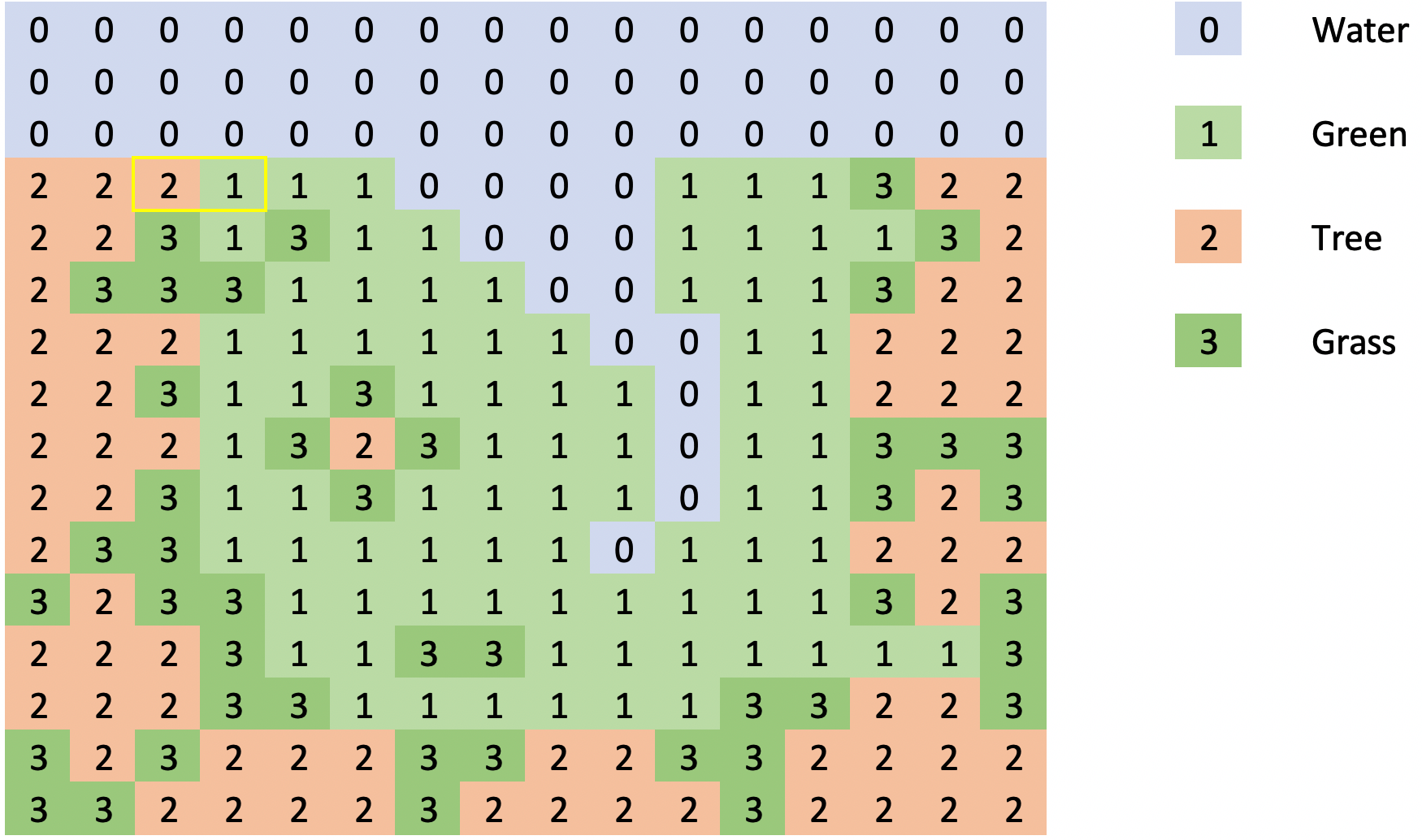
Tree = 2,

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).





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;

…

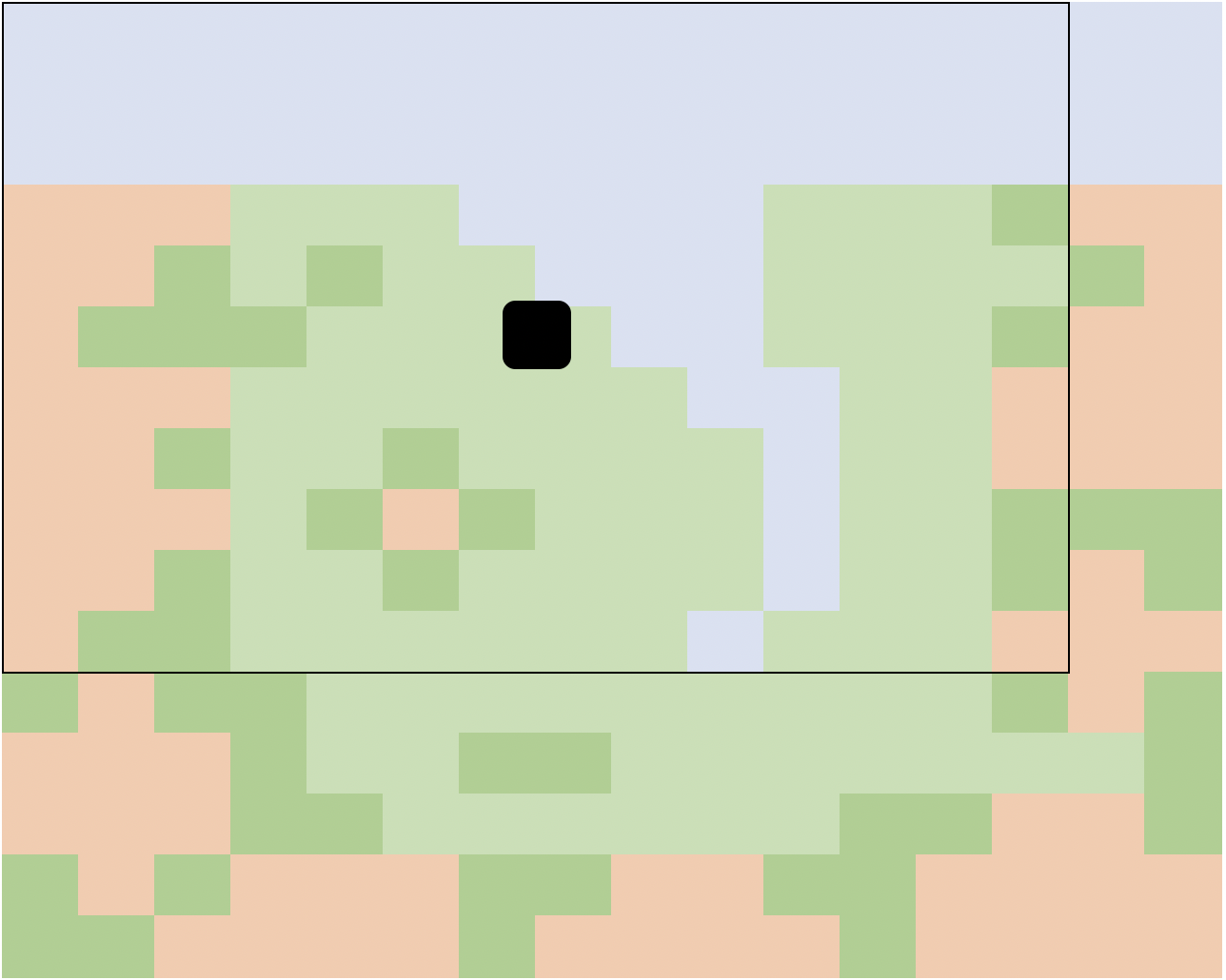
When we are ready to render our viewport, we simply call **tilemap.draw(x, y);**

# Moving a player around the world.

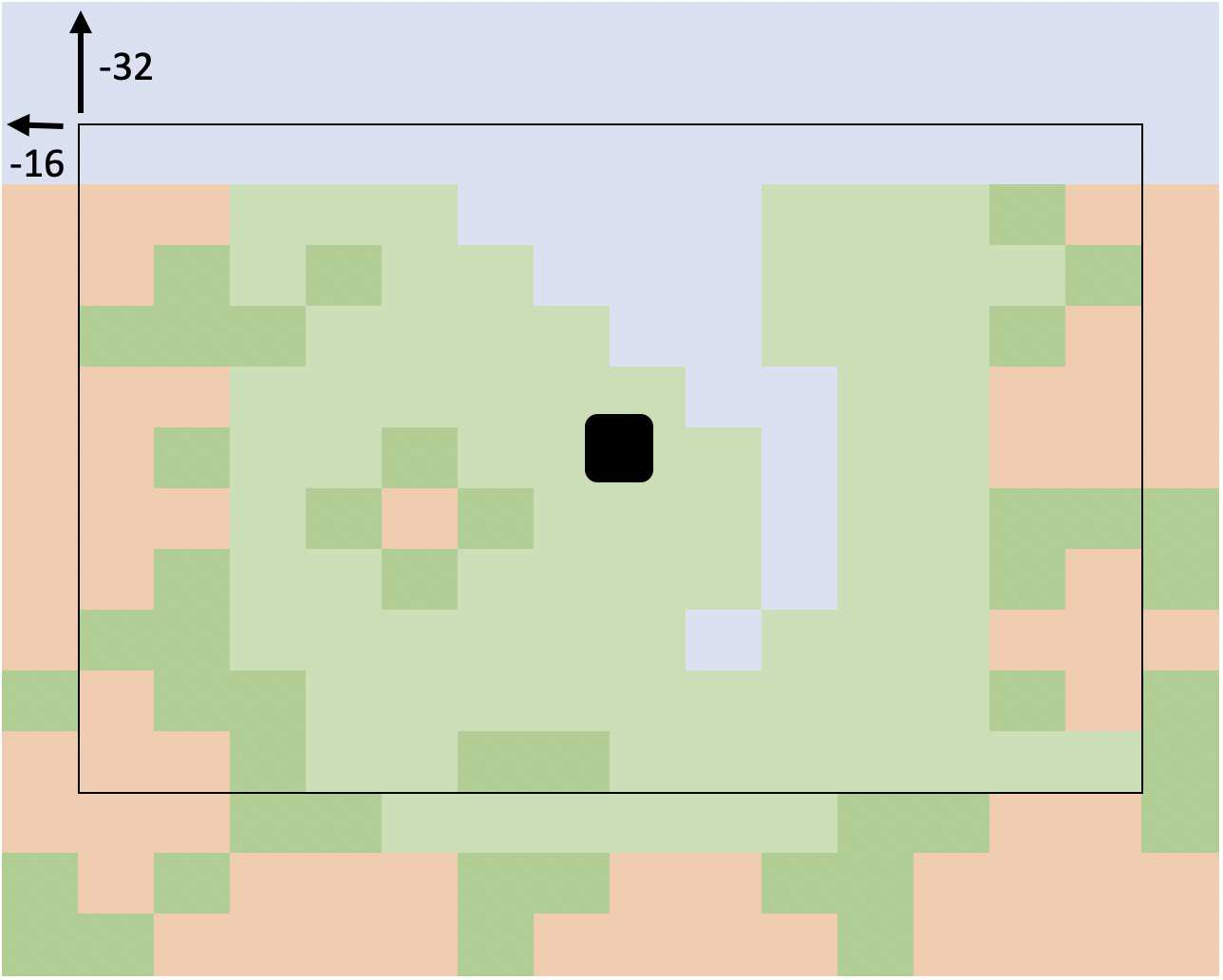
Now that we have defined our world, we can add the player and allow him to explore.

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’.

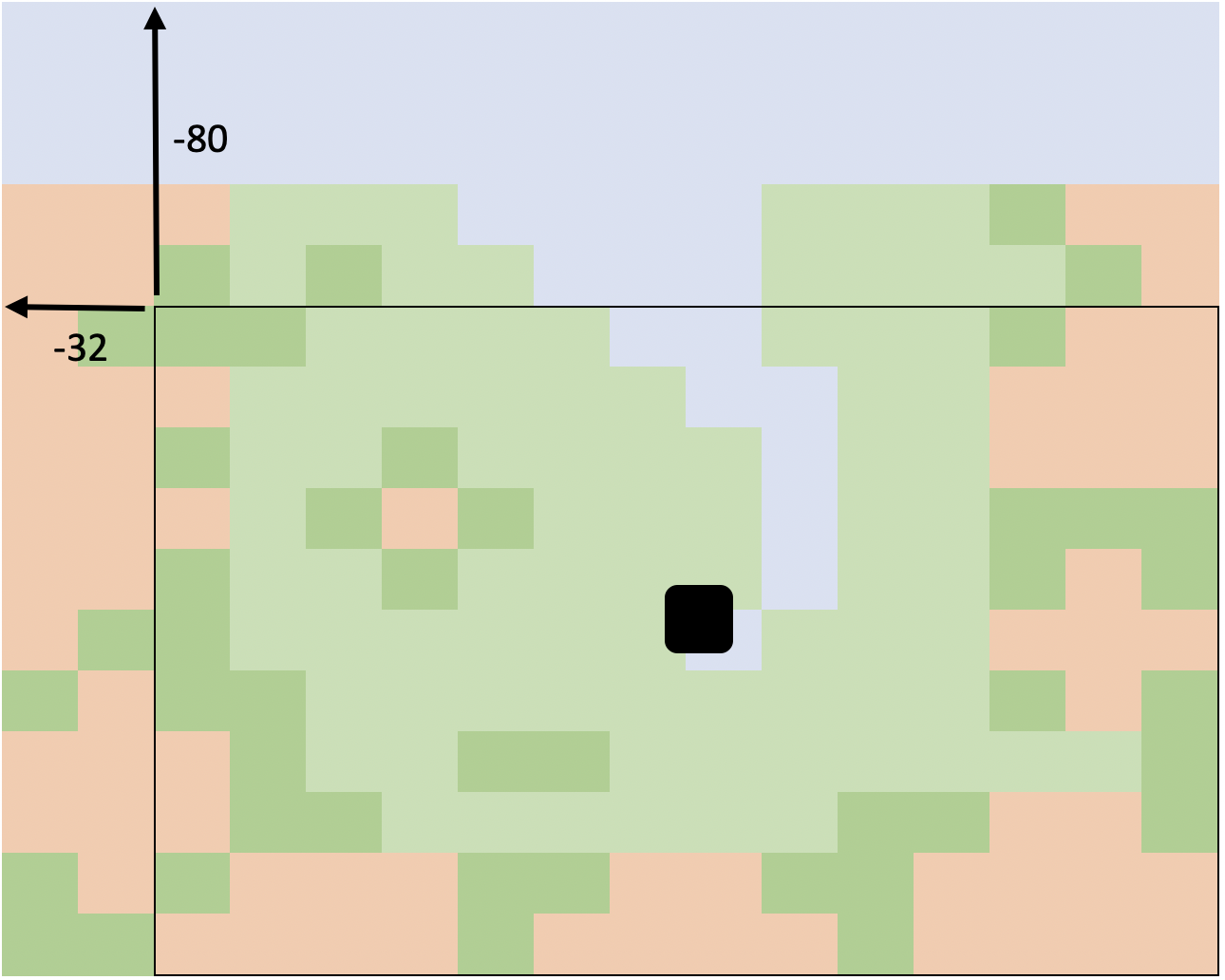
Imagine the player starts near the top-left of the world. When rendering the screen, we typically render the player in the middle of the viewport and hence the middle of the screen. When the viewport is positioned in the top, left-hand corner, as shown below, **the world has an offset of x = 0, y = 0 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 world has on offset of x = -16, y = -32 relative to the viewport.**



Finally, the player reaches a point where the viewport is now touching the right-hand and bottom edges of the world. The 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** **increase** the x position or offset of the viewport relative to the world whereas **pressing the right button will decrease** the x offset.

This may seem counter-intuitive but if you keep referring to the previous images and think of the x and y values as **offsets of the world relative to the viewport** it will make sense.

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);

}

The original **@hanski** code also includes the following checks to ensure that the rendering of the tilemap does not try to render tiles outside of the array bounds, as shown below. This ensures that the x value must be between 0 and -32 (which is the screen width, 220 pixels, minus the world width, 256 pixels). The y values are similarly restricted between 0 and -80.

if (x > 0) x = 0;

if (x + worldWidth < PD::width) x = PD::width - worldWidth;

if (y > 0) y = 0;

if (y + worldHeight < PD::height) y = PD::height - worldHeight;

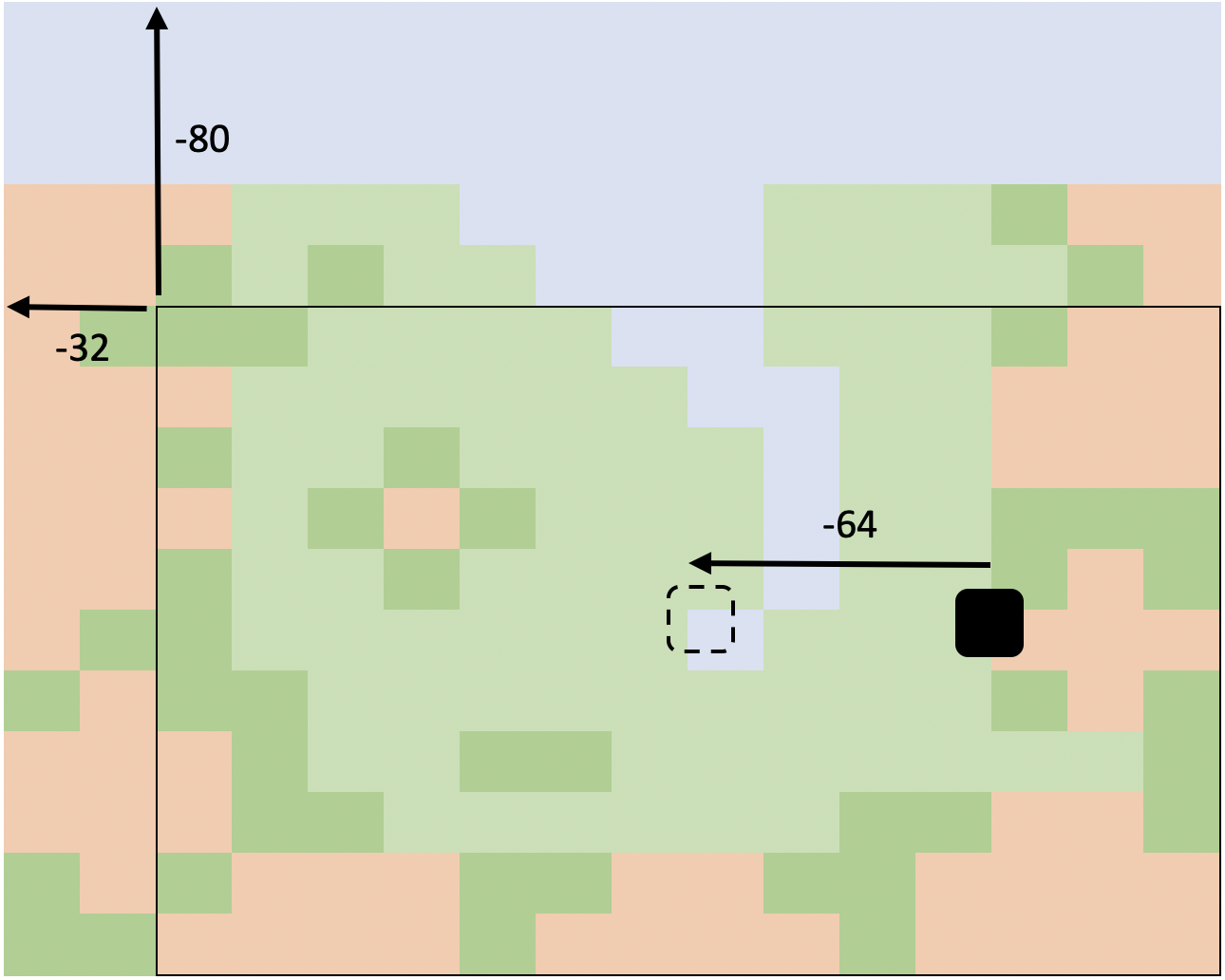
This boundary check is the biggest limitation of the original sample application and it prevents the user from moving close to the edges or corners of the world. Run the application and see how the player moves around the screen but note the limitations.

# Viewport Offsets and Player Offsets

So far we have looked at the offset of the world relative to the viewport but to allow our player to move right to the edges of the world we need to introduce another concept – the player offset relative to the middle of the viewport.

In the image below, our player has moved towards the bottom left to the point where the viewport cannot move any more, at offsets x = -32, y = -80. If the player was to continue moving right, the viewport cannot move anymore and we have to move the player itself relative to the **centre of the viewport**. This is called the **player offset**.

When the player has reached the position near the right of the screen, it now has a viewport / world offset of x = -32, y = -80 and a player offset of x = -64.



Download and review the code in the repo <https://github.com/filmote/Tilemap_2> If you run this code, you will notice that you can move closer to the edges of the screen that the previous code sample and that when you approach the edges, the viewport stops moving and the player moves to the edge and away from the centre of the screen.

Shown below is the code that handles the player movement. For brevity, I have shown only the code to handle the left hand button press. In summary, it checks to see if we can move the viewport first (by checking that the **x** viewport offset is less than zero) and, if not, then alters the **xOffset** value. The offset value is limited to half the width of the viewport.

Similar code exists for the right, up and down buttons. You can review these in the code if you wish.

Note that when rendering the player, we must include the offset when determining the position whereas previously we always rendered the player in the centre of the screen.

int16\_t x = -20;

int16\_t y = -50;

int16\_t xOffset = 0;

int16\_t yOffset = 0;

while (PC::isRunning()) {

...

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

// Otherwise scroll the screen itself if we can ..

if (x < 0) {

x++;

}

// If we cannot scroll the screen, move the player to the left ..

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

xOffset++;

}

}

...

tilemap.draw(x, y);

PD::drawBitmapData(centreScreenX - xOffset, centreScreenY - yOffset,

12, 15, Data::girl12x15Pixels);

}

The code above is a simplification and omits the case where the player presses the left button and they are already on the right hand side of the screen. In this scenario, we should move the player back towards the centre of the screen before moving the viewport or increasing the xOffset value. The full code is repeated below for completeness.

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

// If we are already on the RHS of the screen, come back to the centre ..

if (xOffset < 0) {

xOffset++;

}

// Otherwise scroll the screen itself if we can ..

else if (x < 0) {

x++;

}

// If we cannot scroll the screen, move the player to the left ..

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

xOffset++;

}

}

# Detecting Obstacles

# Adding Enemies