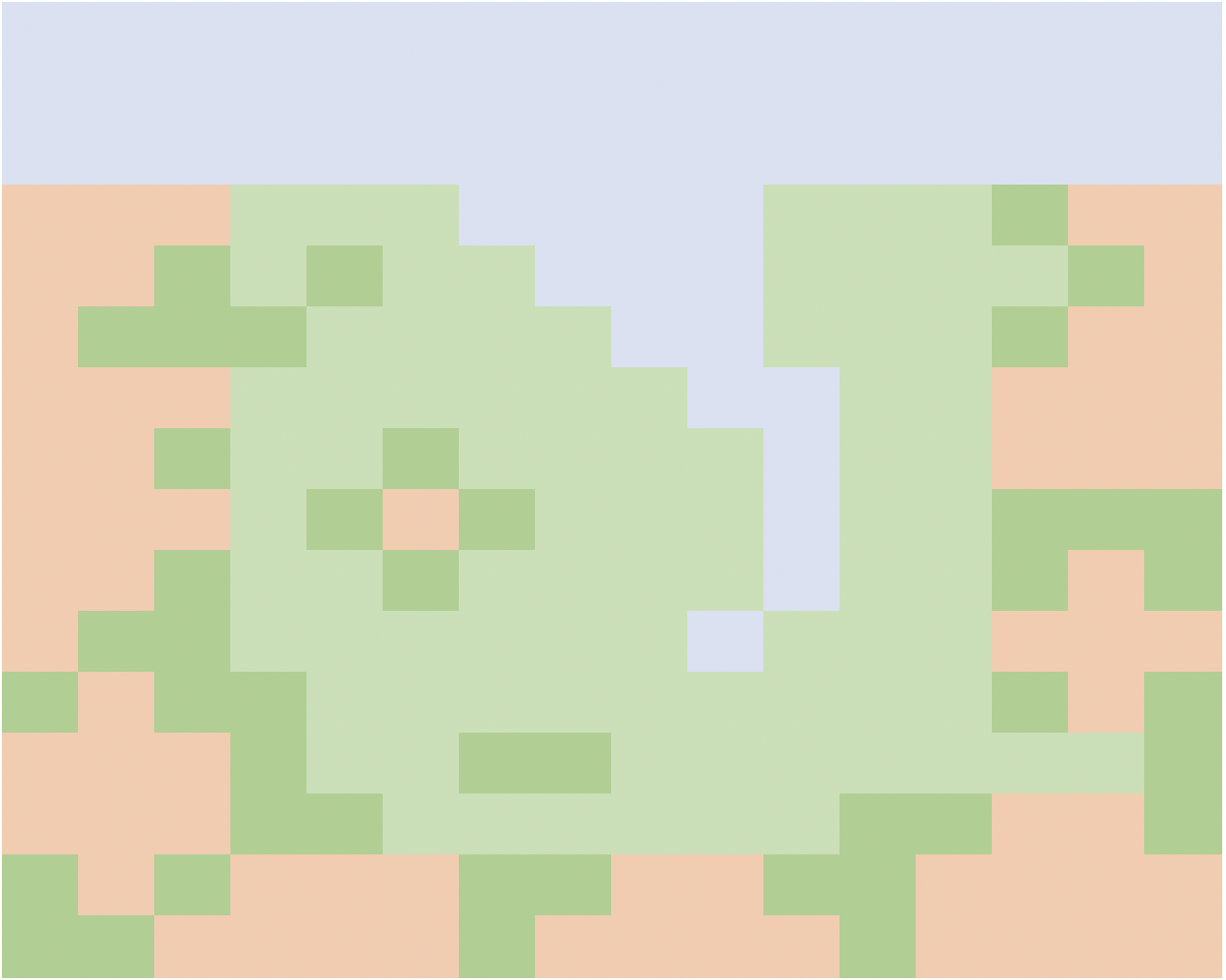
Reference Hanski project.

# 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).

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,

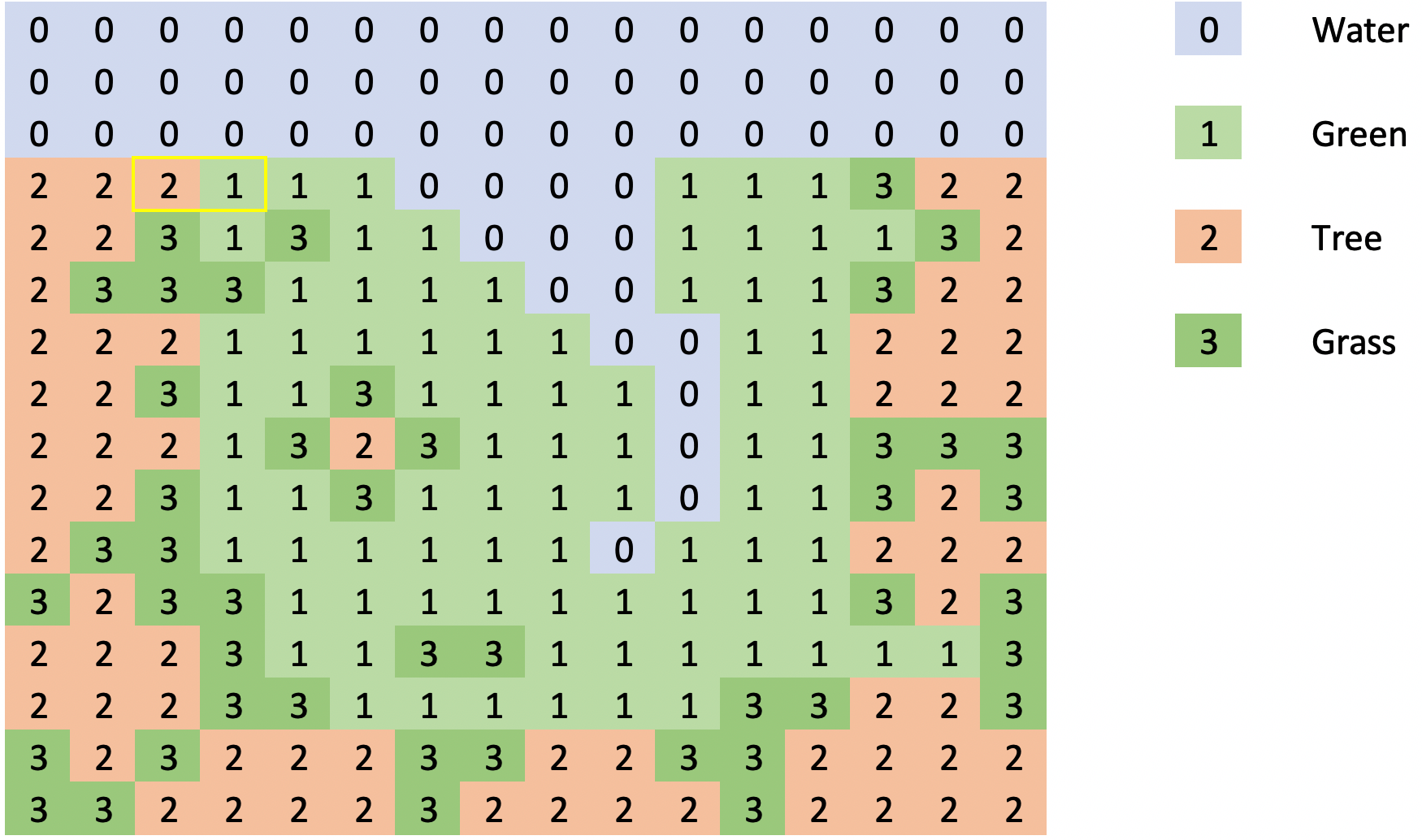
0x22, 0x23, 0x11, 0x33, 0x11, 0x11, 0x11, 0x13,

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;

…

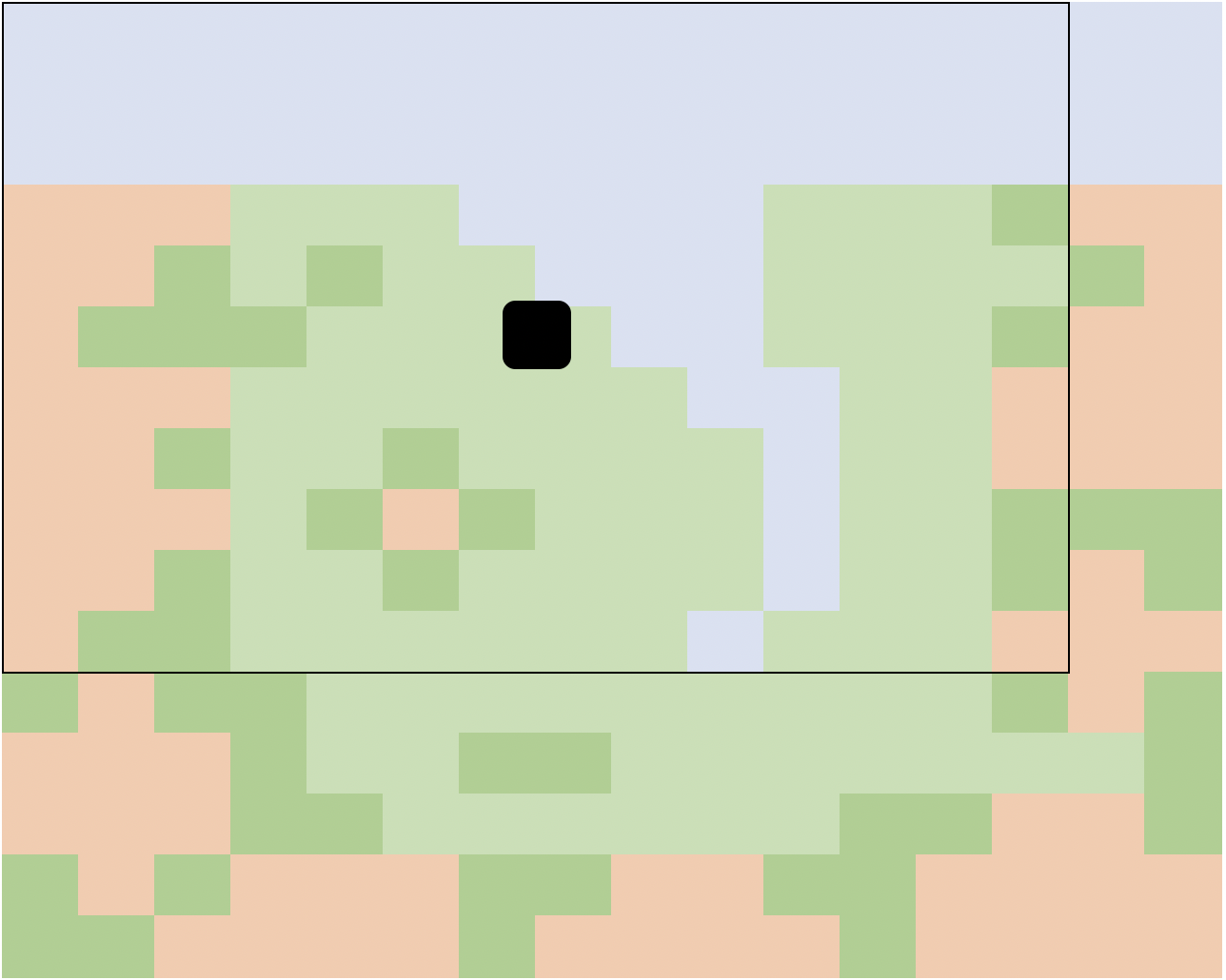
When we are ready to render our viewport, we simply call **tilemap.draw(x, y);**

# Moving a player around the world.

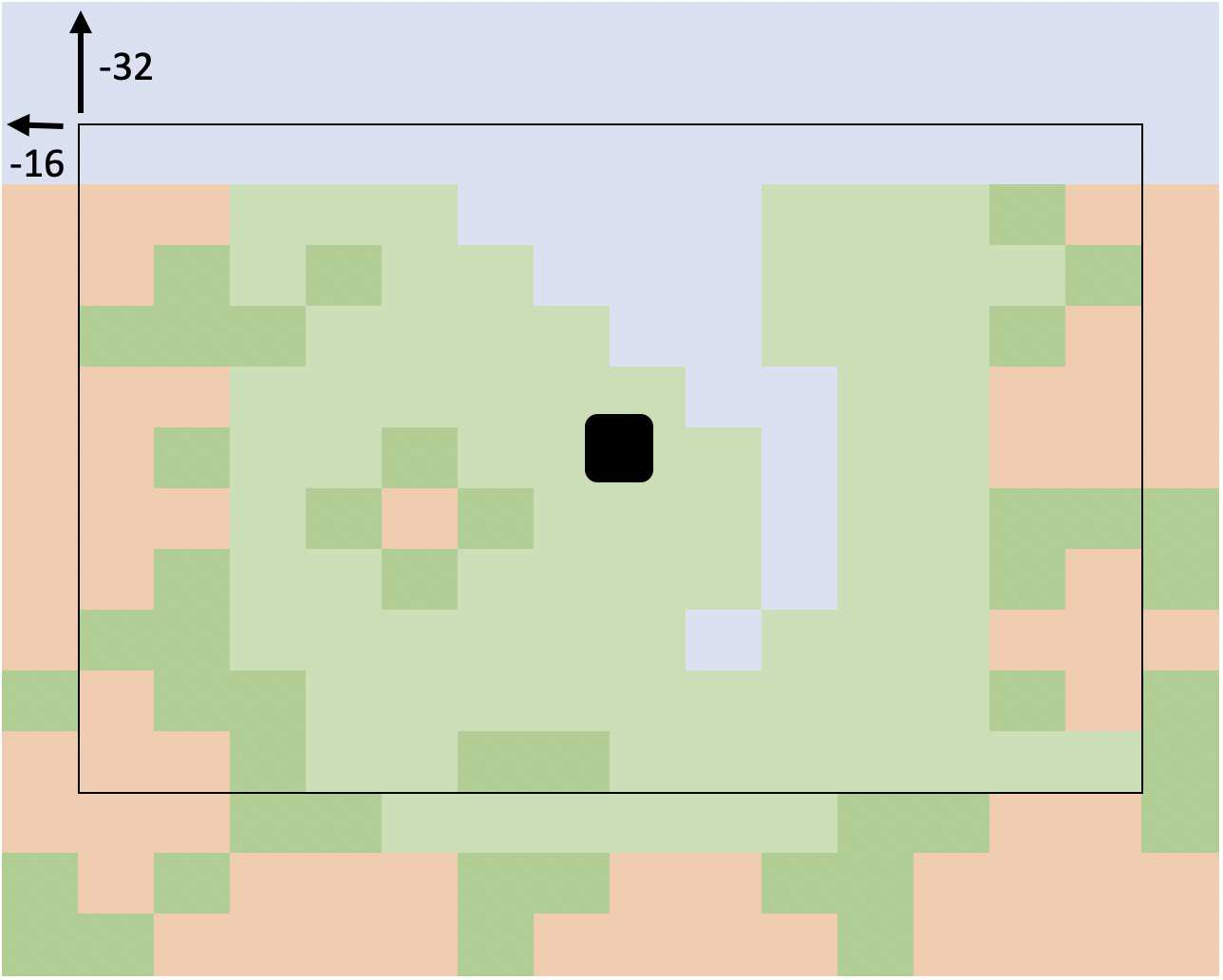
Now that we have defined our world, we can add the player and allow him to explore.

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’.

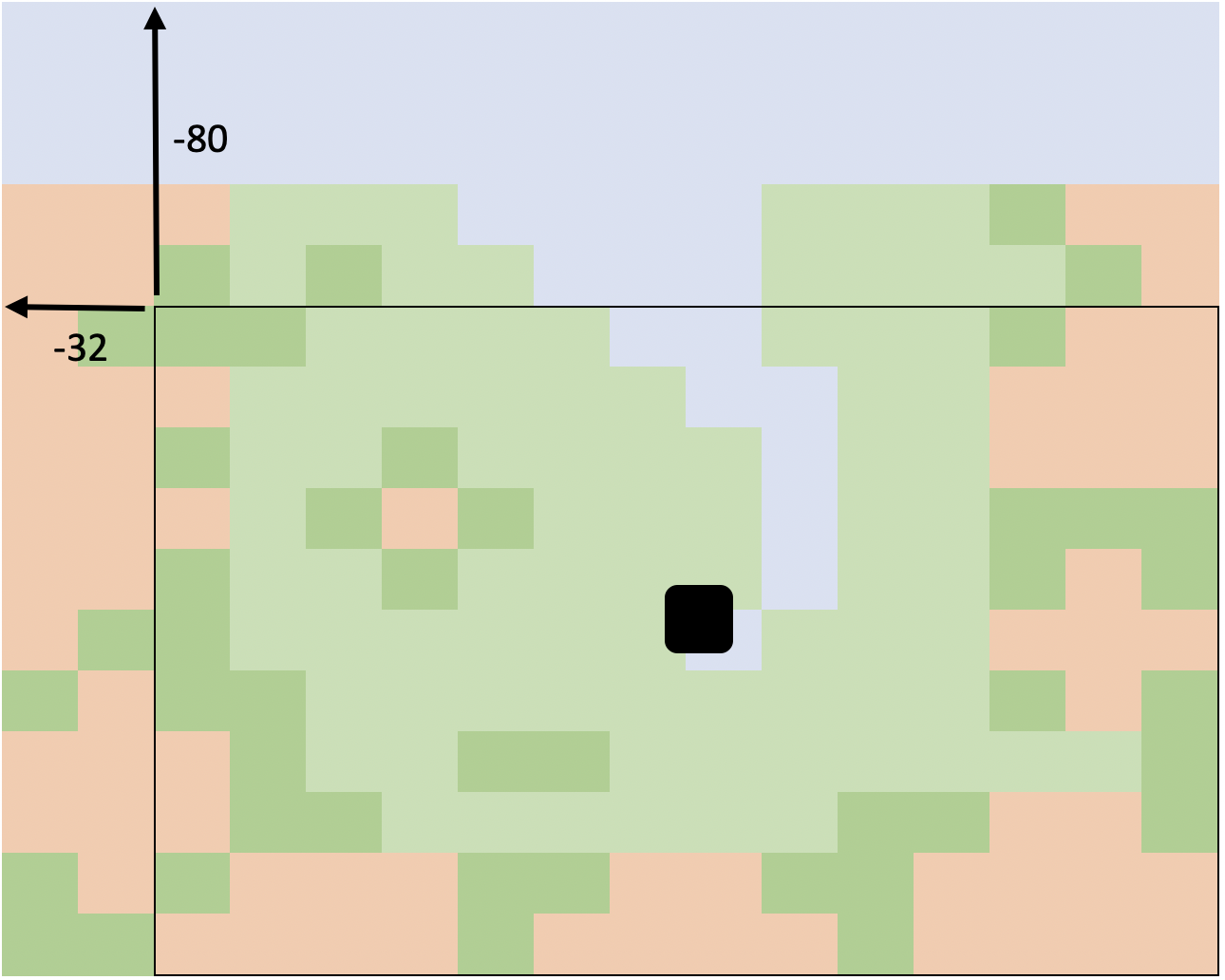
Imagine the player starts near the top-left of the world. When rendering the screen, we typically render the player in the middle of the viewport and hence the middle of the screen. When the viewport is positioned in the top, left-hand corner, as shown below, **the world has an offset of x = 0, y = 0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

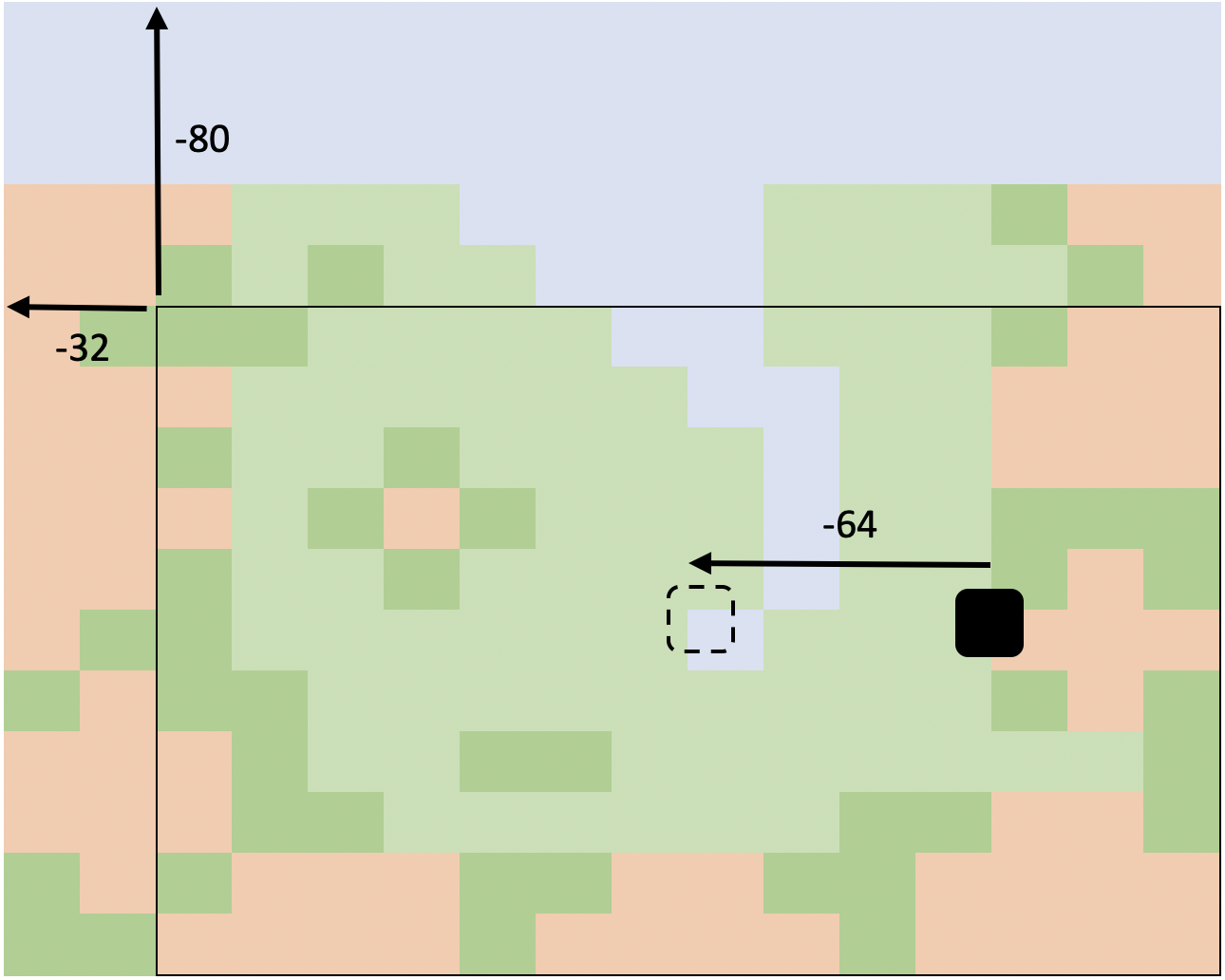
This boundary check is the biggest limitation of the original sample application and it prevents the user from moving close to the edges or corners of the world. Run the application and see how the player moves around the screen but note the limitations.

# 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++;

}

}

Review the code for moving right, up and down. The code for moving up is near identical to the code shown above. The right and down code are common but the extent to which you can move is limited by the screen width.

# Detecting Obstacles

Download the code in the repo <https://github.com/filmote/Tilemap_2> 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;

int16\_t oldXOffset = xOffset;

int16\_t oldYOffset = yOffset;

… handle player movements.

// Check the map tile under the player.

int16\_t playerCenterInMapX = (screenCentreX + 6) - x - xOffset;

int16\_t playerCerterInMapY = (screenCentreY + 7) - y - yOffset;

uint8\_t tileId = tilemap.GetTileId(playerCenterInMapX, playerCenterInMapY, 16);

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

if (tileId != TileType::Green) {

x = oldX;

y = oldY;

xOffset = oldXOffset;

yOffset = oldYOffset;

}

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_3> 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 some helper methods that calculate the position in the world from the offset information.

struct Entity {

int16\_t x;

int16\_t y;

int16\_t xOffset;

int16\_t yOffset;

const uint8\_t width = 12;

const uint8\_t height = 15;

int16\_t getPositionInMapX() { return Constants::screenCentreX - x - xOffset; }

int16\_t getPositionInMapY() { return Constants::screenCentreY - y - yOffset; }

};

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** an **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 (checkMovement(player, positionInMapX - 1, positionInMapY, Direction::Left)) {

// If we are already on the right hand side of the screen, come back to the centre ..

if (player.xOffset < 0) {

player.xOffset++;

}

…

}

}

# Adding Enemies