# Building a simple Tilemap game.

By Filmote

The following article builds on a sample program that @Hanski 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.

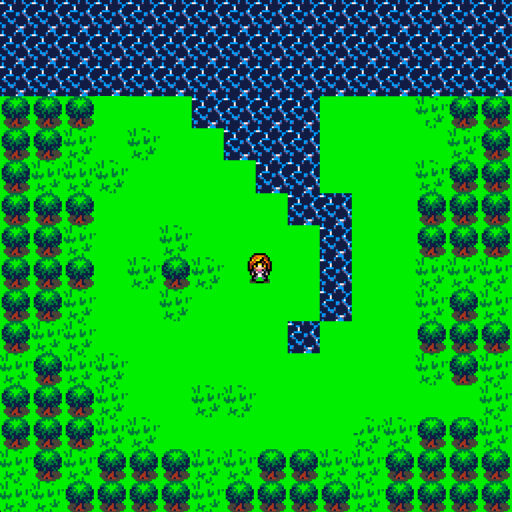
This tutorial is not necessarily trying to teach people to program in C++ and assumes you are familiar with the basics. The code is relatively simple so even if you are comping from a different language you will find the code easy to understand.

# 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,

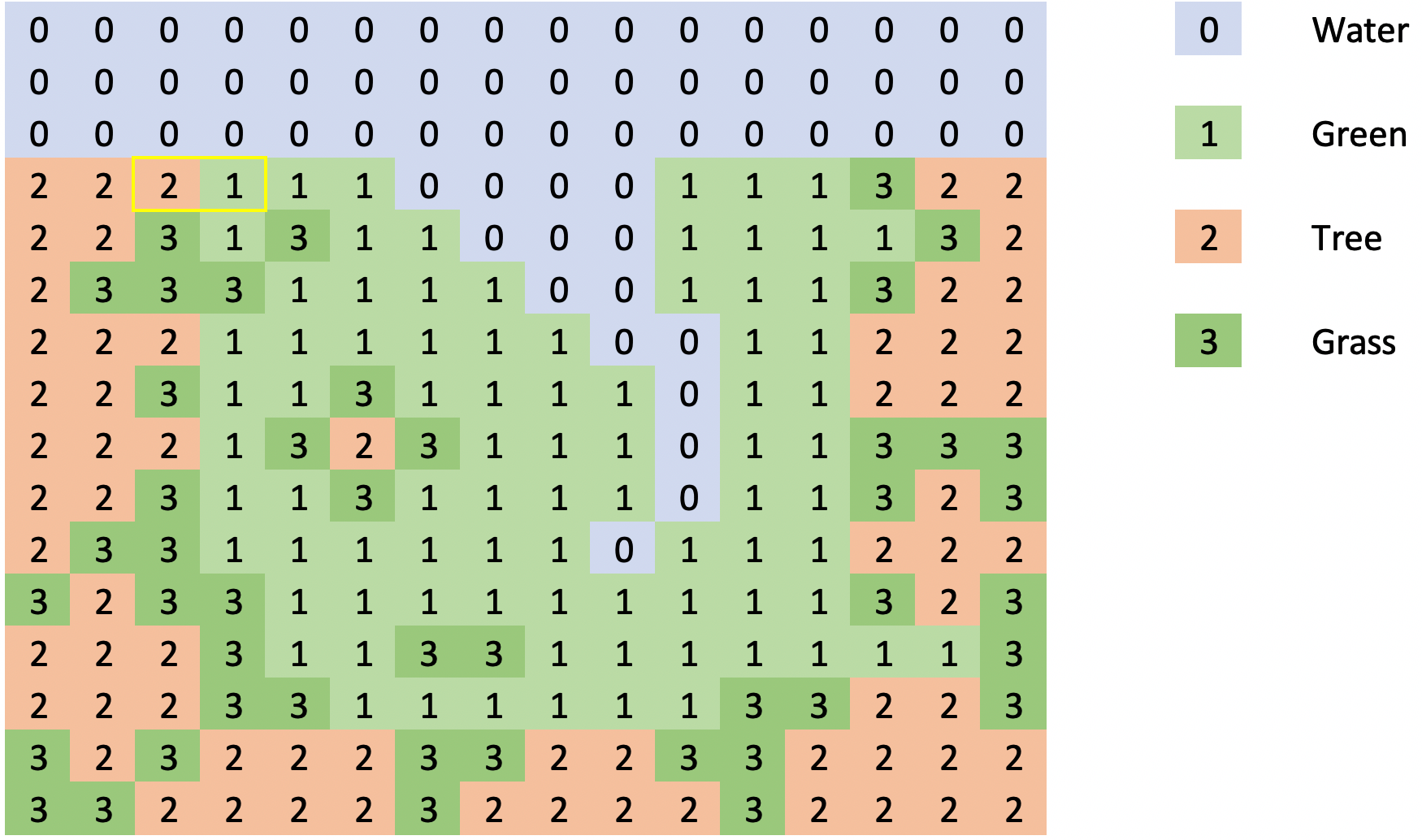
0x22, 0x23, 0x11, 0x33, 0x11, 0x11, 0x11, 0x13,

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.setTile(TileType::Green, 16, 16, Data::green16);

tilemap.setTile(TileType::Tree, 16, 16, Data::tree16);

tilemap.setTile(TileType::Grass, 16, 16, Data::grass16);

tilemap.setTile(TileType::Water, 16, 16, Data::water16);

…

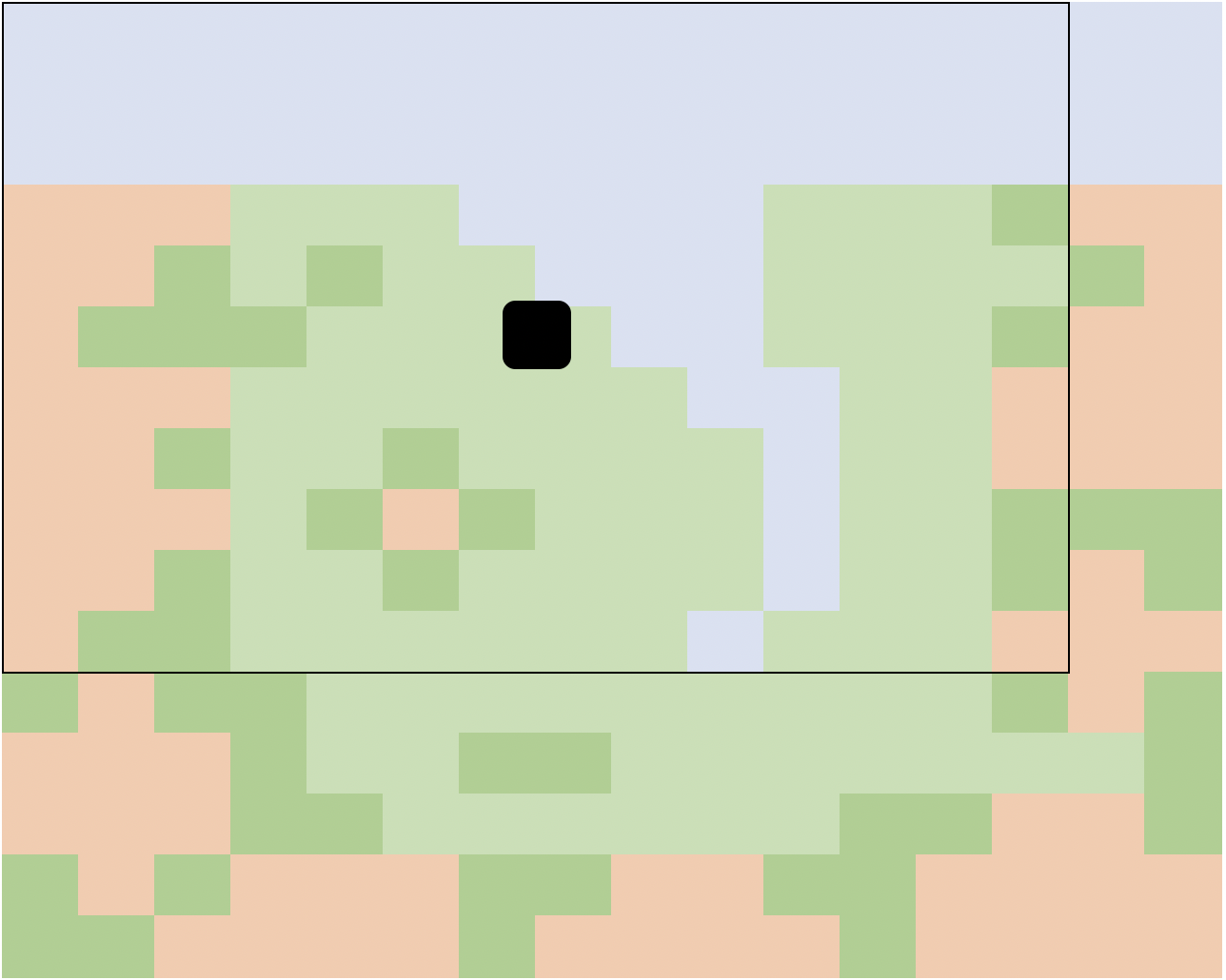
When we are ready to render our viewport, we simply call **tilemap.draw(x, y);**

# Moving a player around the world.

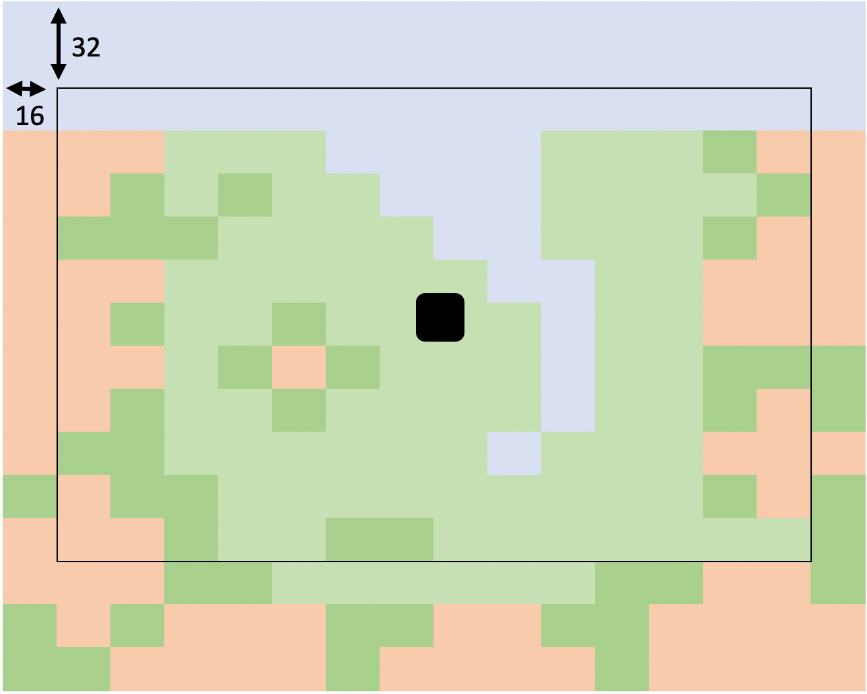
Now that we have defined our world, we can add the player and allow him to explore.

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’.

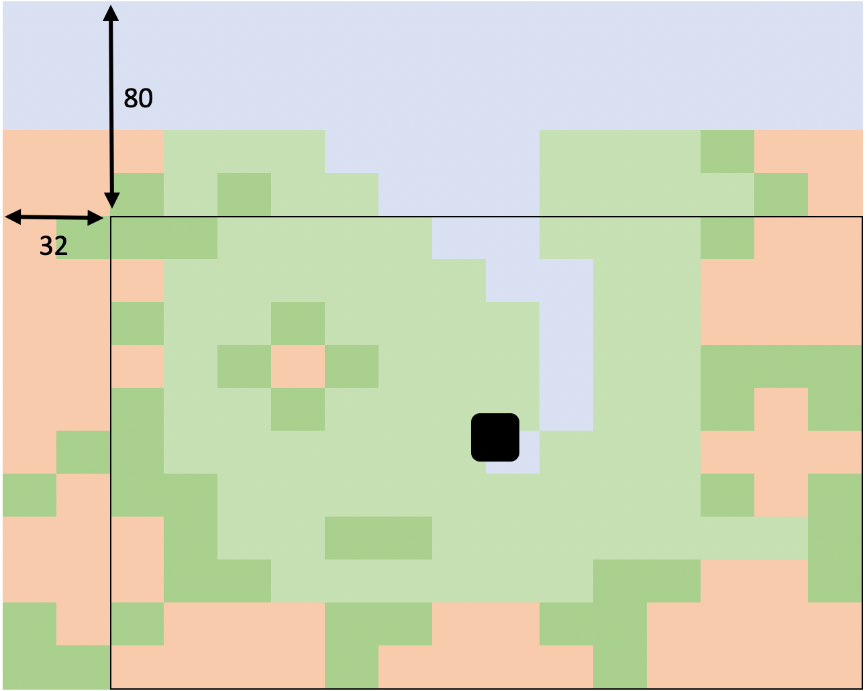
Imagine the player starts near the top-left of the world. When rendering the screen, we typically render the player in the middle of the viewport and hence the middle of the screen. When the viewport is positioned in the top, left-hand corner, as shown below, **the world has an offset of x = 0, y = 0 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**). By the way, **PD::width** and **PD::height** are constants defined in the Pokitto library and represent the current display modes’ width and height. The sample code in this article run using Mode 2 and Mode 15 which have a width and height of 110 x 88 and 220 x 176 respectively.

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.

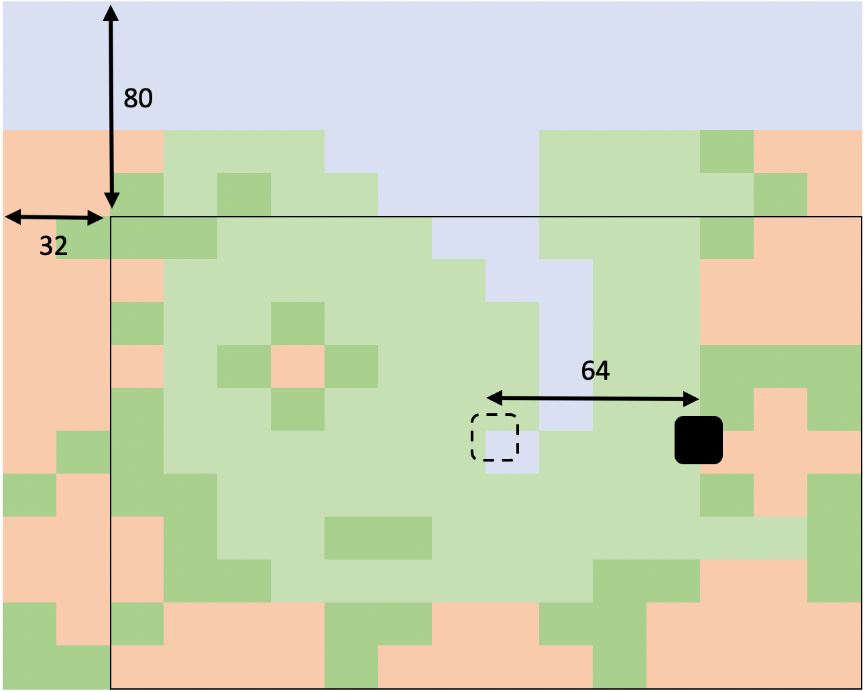
int16\_t xViewPort;

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. For the sake of clarity, I have also moved all of the constant values into a namespace, brilliantly named **Constants**, so that you can see in the code exactly which values these are.

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_3> 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. This is done using a helper method of the **tilemap** class, as shown below.

tilemap.setTile(TileType::Green, Images::Green);

tilemap.setTile(TileType::Tree, Images::Tree);

tilemap.setTile(TileType::Grass, Images::Grass);

tilemap.setTile(TileType::Water, Images::Water);

tilemap.setTile(TileType::Brick, Images::Brick);

tilemap.setTile(TileType::Door\_Open, Images::Door\_Open);

tilemap.setTile(TileType::Door\_Closed, Images::Door\_Closed);

tilemap.setTile(TileType::Key, Images::Key);

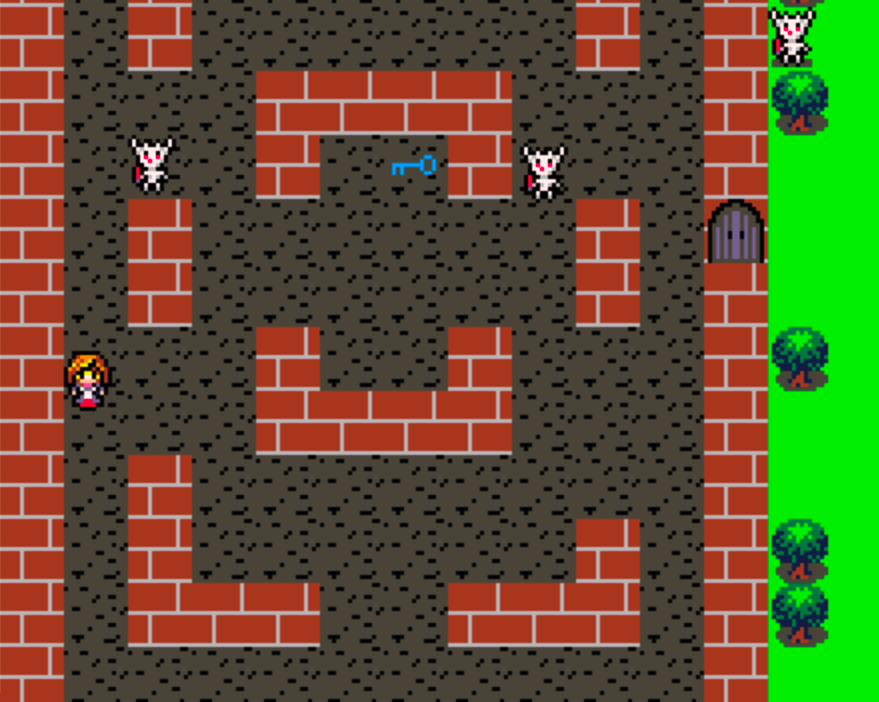
tilemap.setTile(TileType::Carpet, Images::Carpet);

# Doors and Inventory Items

What is a game without doors, keys and other items to pick up along the way?

In this section of the tutorial, we will change the world to include a castle with a locked outer door. To escape, you need to collect the key and before attempting to unlock the door and escape, as shown in the picture below.

Download the code in the repo <https://github.com/filmote/Tilemap_5> and run it. You will see a world similar to that above and will notice that there is both a key and a door in view. Attempt to go through the door without a key and you will be blocked – attempt it a second time after picking up the key and you will be able to pass through it to the green grass beyond.



I will present one method for rendering and tracking items but there are a number of other approaches – each with their own pros and cons. The method I will describe uses the same data map approach we have used in the tutorial so far but we update elements as the player interacts with them.

As you can see from the image above, our new world includes some scenic items that do not change, such as the brick and carpet, and some new items we can interact with including the door and key. These new tiles have been added to the **TileType** enumeration and are used when defining the initial world map.

enum TileType {

Water = 0,

Green = 1,

Tree = 2,

Grass = 3,

Brick = 4,

Door\_Closed = 5,

Door\_Open = 6,

Key = 7,

Carpet = 8,

};

However, when the player touches the key we will want to pick it up and from that point onwards show the carpet (the brown / grey) tile. To do this, we need to be able to modify the world map data .. but if you recall, it is marked as **const** thus preventing update. Marking an array as **const** does ensure they are stored in flash - rather than RAM – which is important for large arrays.

The simplest fix would be to change this would be to remove the **const** declaration but in the next section I want to introduce the ability to have multiple worlds, so an alternative approach is to copy the world from flash into an array in memory. This performed by the function below.

A new array **worldMap** is declared as global variable with an initial size of the width multiplied by the height divided by two. The ‘divide by two’ clause has been added as the world data is actually packed into the array using ‘two bytes per tile’ hence the array is only half of the normal size. Once declared, the function simply copies the data from the flash array to the RAM array using a simple loop.

uint8\_t worldMap[mapTileWidth \* mapTileHeight / 2];

void initWorld() {

// Populate the world data ..

for (uint8\_t i = 0; i < mapTileWidth \* mapTileHeight / 2; i++) {

worldMap[i] = Data::worldMap[i];

}

}

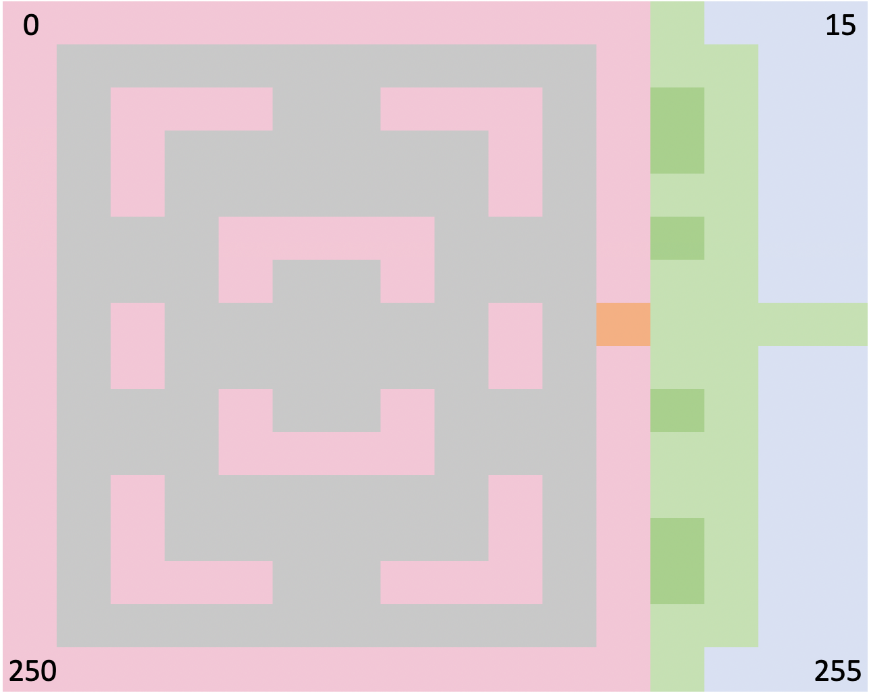
Now that we have our world information in RAM we can update it as needed. Let’s look at the code that detects whether a player has touched a key or not. Firstly, I have added a simple property to the **Player** structure to record whether we have the key or not.

struct Player : Entity {

bool hasKey = false;

};

To update a tile, we need to know its index in the world array of the tiles. Our world is 16 tile wide x 16 tiles high resulting in 256 tiles - the index of the top, left-hand tile is 0, the top right-had side is 15, and the bottom right is 255.



A simple helper function allows us to calculate the tile based on a player’s **x** and **y** position as shown below. The third parameter, **width**, is the width of the world measured in tiles which for our world is 16.

uint16\_t getTileIndex(int32\_t x, int32\_t y, uint16\_t width) {

uint32\_t tx = x / width;

uint32\_t ty = y / width;

return (ty \* width + tx);

}

The **checkMovement()** function presented earlier has been modified to calculate the tile type and index for the two tiles they could possibly touch depending on the direction they are moving. After calculating the tile types and indices, the function then tests to see if the player has touched a key. If so, the **hasKey** property of the **player** is set to true and the tile set back to **TileType::Carpet** thus preventing the key from being shown again.

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

int8\_t tileId1 = 0;

int8\_t tileId2 = 0;

uint16\_t tile1Index;

uint16\_t tile2Index;

switch (direction) {

case Direction::Left:

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

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

tile1Index = getTileIndex(x, y, 16);

tile2Index = getTileIndex(x, y + entity.height, 16);

break;

case Direction::Right:

…

break;

…

}

// Handle player actions

if (&entity == &player) {

// If we have found a key, pick it up ..

if (tileId1 == TileType::Key) {

player.hasKey = true;

updateTileType(tile1Index, TileType::Carpet);

}

…

}

The **updateTileType()** function, as its name suggests, update the tile type in the world array in RAM. As mentioned earlier, the data in this array is packed ‘two tiles to a byte’ and this function test to see if the tile being update is odd or even, using the **%** (modulus) function, and then updating the left or right portion of the byte appropriately.

void updateTileType(uint16\_t tileIndex, TileType tileType) {

if (tileIndex % 2 == 0) {

worldMap[tileIndex / 2] = (worldMap[tileIndex / 2] & 0x0f) | (tileType << 4);

}

else {

worldMap[tileIndex / 2] = (worldMap[tileIndex / 2] & 0xf0) | tileType;

}

}

The logic to handle the opening of the door to the **checkMovement()** function is almost as easy as the key logic with a simple check added to ensure the player actually has the key.

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

…

// Handle player actions

if (&entity == &player) {

…

// Open the door? Only if we have a key ..

if (player.hasKey) {

if (tileId1 == TileType::Door\_Closed) {

player.hasKey = false;

updateTileType(tile1Index, TileType::Door\_Open);

}

…

}

}

…

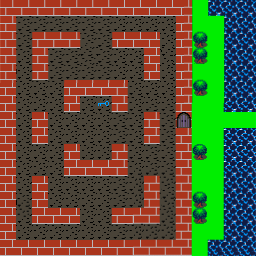
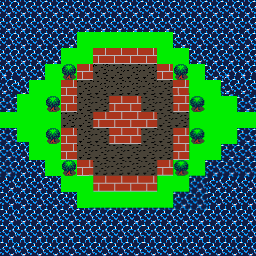
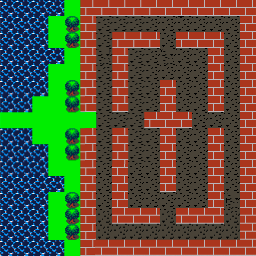
}

We can now pick up inventory items, open doors and escape the castle. But what is outside?

# Multiple Worlds

So far, we have only dealt with only one world map which, although fun, is kind of limiting. In this section, we will expand our world to contain three separate ones. For simplicity, I have kept the dimensions of the three worlds identical but this doesn’t need to be the case. At the end of the section I will detail what needs to be changed in order to have the worlds different dimensions.

Our three worlds are shown below. I have designed them so that moving from one world to the next brings you into the new world from the opposite side but same height as the previous world. This is also a simplification I have made but - as with the worlds of different dimensions - I will detail the changes that need to be made to accommodate different starting locations.

Download the code in the repo <https://github.com/filmote/Tilemap_6> and run it. You can now move between the different worlds freely. You will notice that each level has a different number of enemies and that they start in different locations.

Firstly, let’s look at the data for the three worlds as defined in the **Data.h** file. These are defined as three separate arrays using the same logic we have used so far. Once defined, pointers to the three arrays are included in a fourth array called **worldMaps** which allows the rest of the code to refer to the world array by index.

namespace Data {

const uint8\_t worldMap\_0[] = {

0x44, 0x44, 0x44, 0x44, 0x44, 0x44, 0x10, 0x00,

…

};

const uint8\_t worldMap\_1[] = {

…

};

const uint8\_t worldMap\_2[] = {

…

};

const uint8\_t \* const worldMaps[] = {

worldMap\_0, worldMap\_1, worldMap\_2

};

};

The number of enemies and starting positions for each are also defined in the **Data.h** file alongside the world data. There are three definitions - one for each world – and they consist of 7 values. The first value specifies the number of enemies in the world and the remaining 6 values the **x** and **y** positions of each enemy.

As with the world data, pointers to the enemy starting position arrays are added to a fourth array to allow our code to access them by index.

I have limited the number of enemies to three but this could easily be extended.

namespace Data {

…

// Enemy starting positions ..

const uint16\_t startingPostions\_0[] = {

3, 81, 49, 161, 49, 193, 64

};

const uint16\_t startingPostions\_1[] = {

1, 80, 80, 0, 0, 0, 0

};

const uint16\_t startingPostions\_2[] = {

2, 98, 49, 161, 49, 0, 0

};

const uint16\_t \* const startingPostions[] = {

startingPostions\_0, startingPostions\_1, startingPostions\_2

};

};

With the world data and starting positions defined in arrays, we can alter the **initWorld()** function presented earlier to accept a parameter to indicate which world to load. The revised code is shown below and loads the world data in a similar way to that shown previously and the enemy starting positions.

Previously, the variable **numberOfEnemies** had a constant value of three. It has now been made a global variable and is populated as part of the **initWorld()** function however the code that moves and renders the enemies remains unchanged as it used this constant as an upper limit when looping.

uint8\_t worldMap[Constants::mapTileWidth \* Constants::mapTileHeight / 2];

uint16\_t numberOfEnemies = 0;

void initWorld(uint8\_t worldIndex) {

// Populate the world data ..

for (uint16\_t i = 0; i < Constants::mapTileWidth \* Constants::mapTileHeight / 2; i++) {

worldMap[i] = Data::worldMaps[worldIndex][i];

}

// Populate the enemy starting positions ..

numberOfEnemies = Data::startingPostions[worldIndex][0];

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

enemies[i].x = Data::startingPostions[worldIndex][(i \* 2) + 1];

enemies[i].y = Data::startingPostions[worldIndex][(i \* 2) + 2];

}

}

I have conveniently laid out my three worlds so that moving to the right from one world to another increases the index of the current world and moving from right to left decreases it. This sort of transition can easily be accommodate by testing to see if the player is moving off the edge of the world map as part of the **handlePlayerMovements()** function.

Moving off the map to move to the next world is common however it may not always be the case. For example, you might traverse to the next world by climbing up (or down) a staircase located in the middle of the world itself. These transitions can also be handled in the **handlePlayerMovements()** function in much the same way as the key was picked – by testing to see if the player has moved onto a special tile.

The code below shows the changes that have been made to cater for the left and right movements off the world map. Whereas previously we prevented the player from ever being able to move outside of the world map, we now allow this movement and increase or decrease a global variable that contains the current world index accordingly. When the player has moved completely off the left or right hand side, we reposition them to the opposite side of the world so that they can enter from the correct side.

uint8\_t currentWorld = 0;

void handlePlayerMovements() {

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

if (player.x - 1 <= 0) {

player.x--;

if (player.x = -1) {

player.x = Constants::worldWidth - 1;

currentWorld--;

initWorld(currentWorld);

}

}

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

player.x--;

}

}

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

if (player.x + 1 >= Constants::worldWidth - player.width) {

player.x++;

if (player.x == Constants::worldWidth) {

player.x = -player.width + 1;

currentWorld++;

initWorld(currentWorld);

}

}

else if (checkMovement(player, player.x + 1, player.y, Direction::Right)) {

player.x++;

}

}

…

}

As pointed out earlier, this approach is a simplification and relies on a number of key points. Firstly, it assumes the worlds are lined up from left to right in order. Secondly, it assumes that you will leave the current world and enter the next at the same **y** height. The final two compromises are that the worlds are of the same dimensions and that they are designed to prevent you from exiting where you should not. I have achieved this by bordering the left and right edges of the individual world maps with bricks and water.

The first compromise can easily be overcome by coding the conditions to handle the progression through worlds rather than assuming they are linear.

The second compromise can be overcome by altering the **Data.h** file and **initWorld()** code to include a player starting location similar to the enemy starting positions.

Changing the dimension for each world is a little more complex. The initialisation of our tilemap, which currently is in our **main()** function, could be moved to the **initWorld()** function with the world dimensions (16 x 16) being read from a configuration array similar again to the enemy start positions.

Doing so also allows you to reassign the tiles to a different set if, for example, one world was indoors and the next outside. This following pseudo-code shows how this might be achieved. This approach would allow you to change your tile set for different game levels / worlds while still using only 16 tiles.

tilemap.set(16, 16, worldMap);

if (indoors) {

tilemap.setTile(TileType::Brick, Images::brick16);

tilemap.setTile(TileType::Door\_Open, Images::door\_open16);

tilemap.setTile(TileType::Door\_Closed, Images::door\_closed16);

tilemap.setTile(TileType::Key, Images::key16);

tilemap.setTile(TileType::Carpet, Images::carpet16);

}

else {

tilemap.setTile(TileType::Green, Images::green16);

tilemap.setTile(TileType::Tree, Images::tree16);

tilemap.setTile(TileType::Grass, Images::grass16);

tilemap.setTile(TileType::Water, Images::water16);

}

# Sixteen tiles only?

In the previous section, I introduced an approach to swapping the tile set you are using based on the level or world you are currently playing. This allows you to restrict each individual world to 16 tiles which, in turn, allows the compression of the world using the ‘two tiles per byte’ encoding shown throughout this tutorial so far.

But what if your levels are more complex and require more than 16 tiles?

A recent PR submitted by @hanski allows you to increase the tile count from 16 to a high value up to 256 tiles. This is typically enough for even the most complex worlds especially when combined with the ability to replace tiles per level. The downside of having more than 16 tiles is that the tiles cannot be compressed into the world map array (easily) and your world map will literally double in size. This may not be an issue for the maps stored in flash but copying these to RAM may be a concern.

Download the code in the repo <https://github.com/filmote/Tilemap_7> This is essentially the same code as Project 6 however it has been changed to support additional tiles.

Adding support for additional tiles starts with adding a simple **#define** in the **My\_settings.h** configuration file, as shown below. The number of tiles you specify (32 in my case) can be any number from 16 to 256. Omitting the declaration will result in the default value of 16 being used.

#define MAX\_TILE\_COUNT 32

In the sample project, have used an **#ifdef**, **#else** and **#endif** compilation directives to conditionally compile sections of code depending on this value. In this way, you can see the original code and its replacement side by side. The sample code actually does not use more than 16 tiles so feel free to change this value from between 16 and any other value and recompile the code. If you are using FemtoIDE, make sure you select the ‘Clean’ option before building to ensure the changes are detected.

Changing the setting to anything more than 16 will require a change to the data in the world maps from the ‘two tiles per byte’ encoding to a simple ‘single byte per tile’. This is clearly shown in the examples below - the first section of code show our original world map whereas the second shows the same map in the ‘single byte per tile’ encoding. The highlighted values in both samples highlights the difference in size between the two approaches – the second approach literally taking twice as many bytes as the first.

const uint8\_t worldMap\_0[] = {

0x44, 0x44, 0x44, 0x44, 0x44, 0x44, 0x10, 0x00,

0x48, 0x88, 0x88, 0x88, 0x88, 0x84, 0x11, 0x00,

0x48, 0x44, 0x48, 0x84, 0x44, 0x84, 0x21, 0x00,

0x48, 0x48, 0x88, 0x88, 0x84, 0x84, 0x21, 0x00,

…

};

const uint8\_t worldMap\_0[] = {

4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 1, 0, 0, 0,

4, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 4, 1, 1, 0, 0,

4, 8, 4, 4, 4, 8, 8, 4, 4, 4, 8, 4, 2, 1, 0, 0,

4, 8, 4, 8, 8, 8, 8, 8, 8, 4, 8, 4, 2, 1, 0, 0,

…

};

Other changes are required when declaring the worldMap array and the functions that subsequently populate and update it. These changes are shown below with both the original and changed code. In all cases, the changes simply account for the two encoding types.

#if MAX\_TILE\_COUNT == 16

uint8\_t worldMap[Constants::mapTileWidth \* Constants::mapTileHeight / 2];

#else

uint8\_t worldMap[Constants::mapTileWidth \* Constants::mapTileHeight];

#endif

void initWorld(uint8\_t worldIndex) {

// Populate the world data ..

#if MAX\_TILE\_COUNT == 16

for (uint16\_t i = 0; i < Constants::mapTileWidth \* Constants::mapTileHeight / 2; i++) {

worldMap[i] = Data::worldMaps[worldIndex][i];

}

#else

for (uint16\_t i = 0; i < Constants::mapTileWidth \* Constants::mapTileHeight; i++) {

worldMap[i] = Data::worldMaps[worldIndex][i];

}

#endif

…

}

void updateTileType(uint16\_t tileIndex, TileType tileType) {

#if MAX\_TILE\_COUNT == 16

if (tileIndex % 2 == 0) {

worldMap[tileIndex / 2] = (worldMap[tileIndex / 2] & 0x0f) | (tileType << 4);

}

else {

worldMap[tileIndex / 2] = (worldMap[tileIndex / 2] & 0xf0) | tileType;

}

#else

worldMap[tileIndex] = tileType;

#endif

}