# Introduction

This article is the first in a series I am planning that dives deep into the design and development of a simple game known as Pipes. While building the application, I will detail a number of key concepts or techniques that I have used. If you have programmed (before but not necessarily on a Pokitto) these concepts will seem obvious, if you are just starting out then hopefully they will be illuminating.

Most of you will be familiar with the game and would have seen it on mobile phones and even in printed form. Game play is very simple – connect each node with its partner by laying pipe between them. When laying a new pipe, it cannot cross any existing pipes. Once all of the nodes are connected, play moves on to a harder level.

I chose the Pipes game as it is complex enough to enable me to demonstrate the various techniques required to build a Pokitto game without being too complex. I hope you enjoy the articles and feel free to provide feedback via the community (<https://talk.pokitto.com/>)

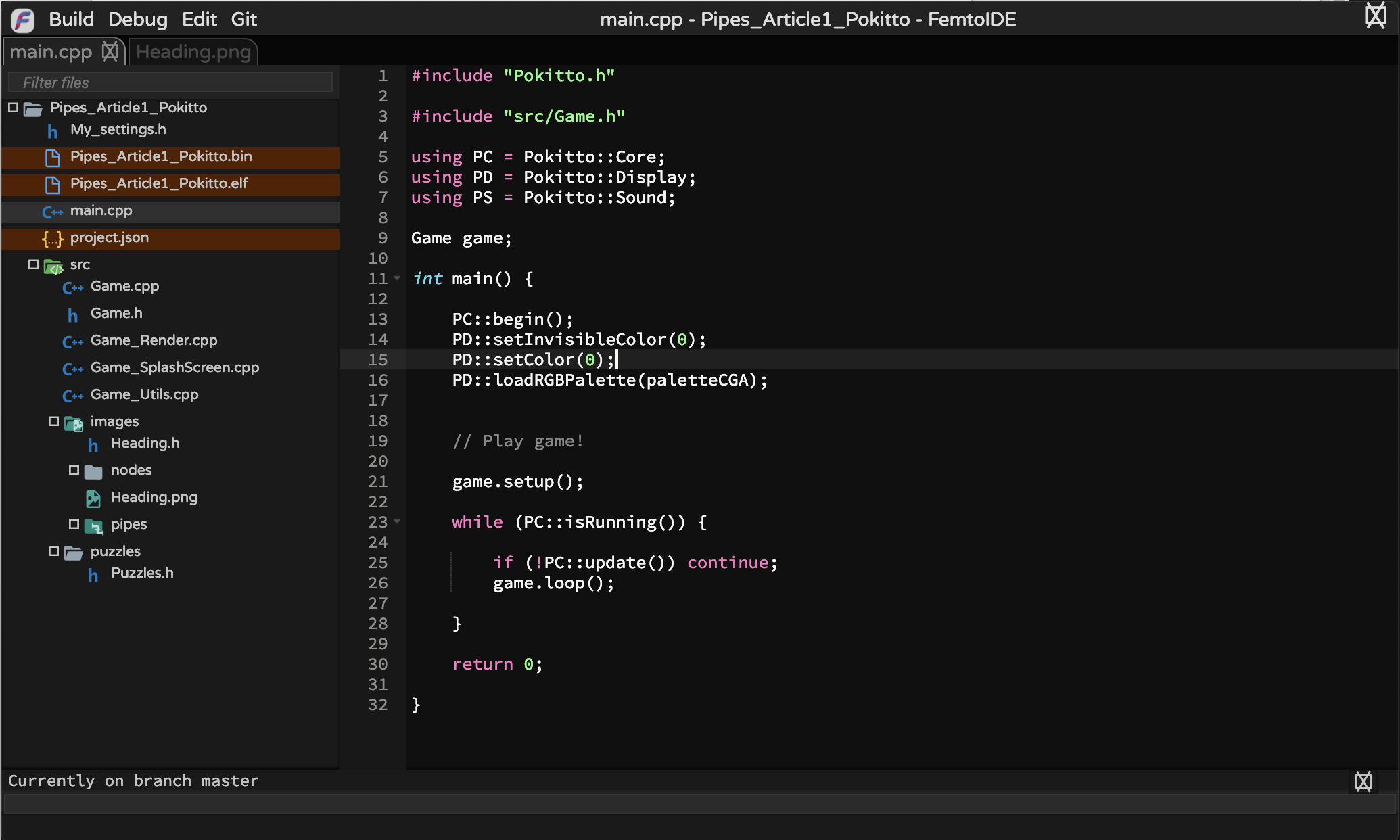
# Getting Started

Before we get started, I am going to assume that you have read the FemtoIDE tutorial xxx will this be in the magazine too? Xxx and have some basic programming experience in C++ or some other language. Even for a newbie, this tutorial is not extremely hard but I it is focused on building a game on the Pokitto.

It doesn’t delve into classes and uses basic data types – chars, bytes and integers. Most of the code could be easily improved using more advanced functions but that is not the intent of this, introductory tutorial.

With each article in this series, I will start by prompting you to download the code from my repository. I encourage you to review the structure of the code just as much as the detail of the code itself. For this article, you can download the code from <https://github.com/filmote/Pipes_Article1>\_Pokitto.

After opening the code in FemtoIDE, review the project. An image of the FemtoIDE is shown below. On the left-hand side is a navigation pane of the programs that make up the and on the right the file currently being edited.



A number of files are worth mentioning at this point. Again I encourage you to review them so that you become familiar with their content.

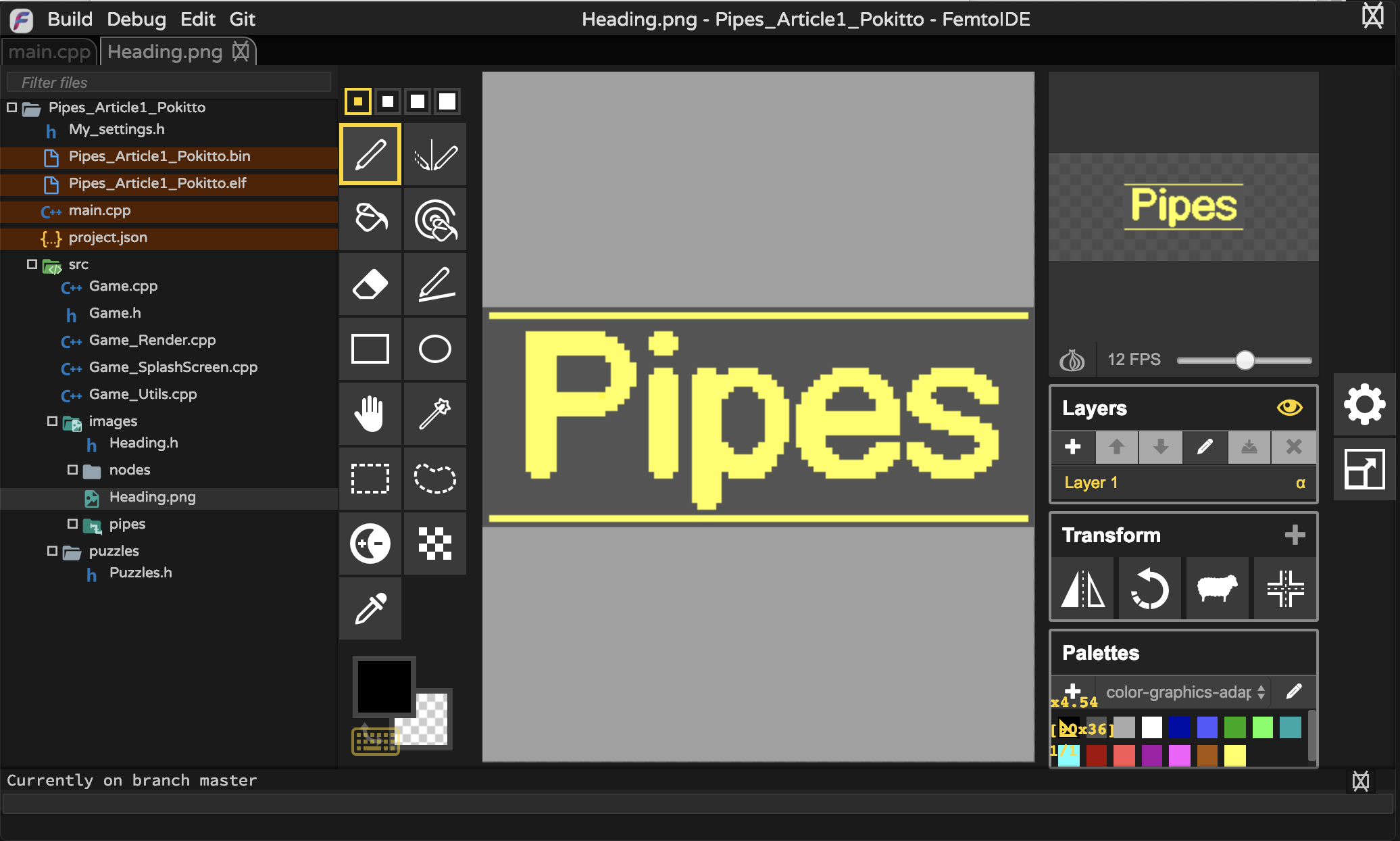
* *My\_settings.h* is a global configuration file used by the Pokitto library. It typically stores information such as what screen mode to use – low resolution, high resolution, 16 colours, etc. It can also be used to store custom settings for your particular program.
* *.bin* and *.elf* files are produced as part of the compilation process. The *.bin* can be loaded onto a Pokitto and played.
* *main.cpp* is the default or first executable file that a Pokitto program runs. The contents of this file is visible in the right hand side of the screen and you can see that it is typically quite simple. In the shown example, it performs a number of steps to initialise the Pokitto then calls a second class (in this example *Game*) which houses all of the game logic.   
    
  You can see that after the game initialisation, the process goes into a continuous loop. For those of you familiar with the Arduino platform, this is synonymous with the *setup()* and *loop()* constructs.
* *project.json* is a file that is unique to the FemtoIDE and stores many project and compilation specific details. Typically, you will not need to edit this file so little knowledge of its structure is required.

The remainder of the files are specific to our game. These consist of a class header (*Game.h*), a number of class files (ending in .cpp) and various support files.

# Artwork

Let’s start with the artwork. In this first instalment, we will work through specifying the artwork for the game’s splash screen and the nodes themselves. In next article, I will add the graphics for the actual pipes that connect the nodes.

One of the great things in the FemtoIDE is the inbuilt *Piskel* graphics editor that allows you to edit the game’s graphics without having to change tools. Furthermore, when you compile a program the graphics are automatically compiled into data files that can be included directly into your game. The *Piskel* editor is shown below:



By default, the Pokitto uses a colour palette known as CGA (Colour Graphics Adapter) which can be traced back to the earliest IBM PCs. The Pokitto library comes with many other, pre-defined palettes that emulate the GameBoy, ZX Spectrum and Pico8 amongst others.

Changing palettes in FemtoIDE

Node graphics:

Pipe graphics:

# 

# 

I encourage you to browse through the sample project and play with the existing images and the Piskel tool.

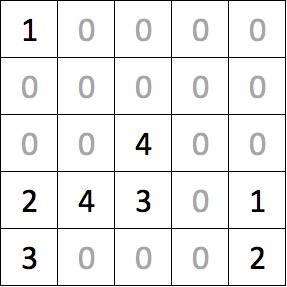
# Development

## Fitting 100s of Puzzles in Memory

Right, so we have an idea and the graphics sorted out. Now we need to consider how we will store the hundreds of puzzles in the limited memory that the Pokitto offers. In the side bar *Program Memory vs RAM?*, I discuss the two different types of memory that the Pokitto has. As the puzzle definitions are static and will be large, we will store them in the ‘program memory’ however even this is limited so any memory savings we can make are gold!

Callout to “Program Memory vs RAM”

Consider a simple puzzle like the one shown below:



The numbers represent the nodes and (obviously) they come in pairs. The puzzle is 5 columns wide and 5 rows deep equaling 25 cells. We could store this in an array that looks something like this:

const byte puzzles\_5x5[] = {

1, 0, 0, 0, 0,

0, 0, 0, 0, 0,

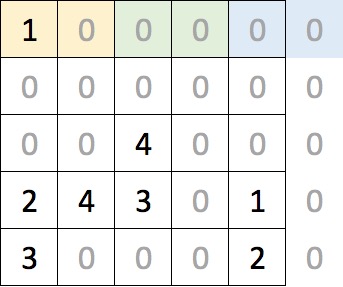
0, 0, 4, 0, 0,

2, 4, 3, 0, 1,

3, 0, 0, 0, 0

};

As we know there are, at most, 10 different node types on a 9x9 puzzle so we can compress the arrays a little using a hexadecimal notation. If we group the numbers in pairs we store two values into a single byte – effective halving our storage requirements. Unfortunately, our puzzle has an odd number of columns so we cannot fully realize this saving and have to be content storing each row using three bytes rather than the original five bytes. Still this is a 40% saving!



The array can now be expressed as shown below – in 15 bytes! Note how the hexadecimal values are expressed using the prefix ‘0x’. The first element, the 0x10, corresponds to the two yellow cells at the top left of the puzzle and the final 0x20 relates to the bottom-right corner. Remember that as the puzzle only has 5 columns, the ‘2’ relates to the right-most column and the ‘0’ is not used.

const byte puzzles\_5x5[] = {

0x10, 0x00, 0x00,

0x00, 0x00, 0x00,

0x00, 0x40, 0x00,

0x24, 0x30, 0x10,

0x30, 0x00, 0x20,

};

Additional puzzles can be added to the same array. The example below shows two puzzles and I have simply formatted the array to visually separate the puzzles. The array is a contiguous 30 bytes long where the first puzzle occupies bytes 0 through 14 with the second puzzle occupying bytes 15 to 29. Note that we – and the Arduboy - refer to the first byte as the 0th position.

const byte puzzles\_5x5[] = {

0x10, 0x00, 0x00,

0x00, 0x00, 0x00,

0x00, 0x40, 0x00,

0x24, 0x30, 0x10,

0x30, 0x00, 0x20,

0x10, 0x20, 0x40,

0x00, 0x30, 0x50,

0x00, 0x00, 0x00,

0x02, 0x04, 0x00,

0x01, 0x35, 0x00,

};

## Reading a puzzle into Memory

Whereas we stored the puzzles in a single dimensional array in memory, it is much easier for us to visualize and manipulate the puzzle if it is represented as a two-dimensional array. The snippet of code below shows a declaration of a multidimensional array of five columns and rows.

byte board[5][5];

The declaration of two-dimensional arrays in C / C++ is a little counter-intuitive (for me anyway!) as you specify the number of columns before the rows. When visualizing a puzzle, the cell in the top right-hand corner of a 5 x 5 grid can be described as x = 4 and y = 0. However, when referencing the same cell in the array it must be referred to as board[0][4] or board[y][x]. Of course, our puzzles all have equal dimensions so this is a little academic.

As this tutorial progresses, we will end up with an array for each of the puzzle sizes – 5x5, 6x6, 7x7, 8x8 and 9x9.

The code below repeats our puzzle array and shows how to read the first element of it. If you are unfamiliar with C++ (and most other languages), the first element in an array is index zero so the expression puzzles\_5x5[0] retrieves the first element. The expression puzzles\_5x5[14] would retrieve the last element of the below array as it has 15 elements.

const byte puzzles\_5x5[] = {

0x10, 0x00, 0x00,

0x00, 0x00, 0x00,

0x00, 0x40, 0x00,

0x24, 0x30, 0x10,

0x30, 0x00, 0x20,

};

byte byteRead = puzzles\_5x5[0];

Callout to “Bit Manipulation”

The previous code would read the first element from the array, 0x10. Once the value has been retrieved, we need to split the value into two using two functions whose operation is described in the side panel *Bit Manipulation*. Shown below, they simply return the left and right values of a two-digit hexadecimal number.

byte leftValue(byte val) {

return val >> 4;

}

byte rightValue(byte val) {

return val & 0x0F;

}

With these two functions in play, we can now render an entire puzzle. The initBoard() function accepts a parameter that defines which puzzle in the array of puzzles to retrieve. A value of 0 (the first puzzle) will result in the function reading the first 15 elements from the puzzles\_5x5 array and populate our two-dimensional board[][] array. Passing a value of 1 for the puzzle number will result in the function reading bytes 15 to 29 of the puzzles\_5x5 array and so on.

The code is a little convoluted for a simple 5 x 5 puzzle but you will see in later articles that we will flesh it out to handle any puzzle size. The main complexity of the code is due to it compensating for puzzles with odd number indices (5x5, 7x7 and 9x9) where the last byte on each row of the array is discarded.

#define PUZZLE\_X 5

#define PUZZLE\_Y 5

void initBoard(byte puzzleNumber) {

byte x = 0;

byte y = 0;

byte byteRead = 0;

for (int i = (puzzleNumber \* 15); i < (puzzleNumber + 1) \* 15; i++) {

byteRead = pgm\_read\_byte(&puzzles\_5x5[i]);

// Load up the left hand value ..

board[y][x] = 0;

if (leftValue(byteRead) > 0) {

board[y][x] = 0xF0 | leftValue(byteRead);

}

x++;

// Are we still in the confines of the board?

if (x <= PUZZLE\_X) {

board[y][x] = 0;

if (rightValue(byteRead) > 0) {

board[y][x] = 0xF0 | rightValue(byteRead);

}

}

x++;

if (x >= PUZZLE\_X) { y++; x = 0; }

}

}

I have purposely neglected to point out the little bit of code that logically ORs the retrieved value with the hexadecimal constant 0xF0. Take it for granted that there is a cunning plan for this later and that it will be revealed when we start actually laying pipe.

## Rendering the Board

In the section *Node Graphics* above, we designed 10 nodes to support our game. You can review these in the source code under the *src/images/Nodes* directory. You will notice that there is a graphics file and a corresponding .h class file which contains the encoded version of the graphic in a format that the Pokitto can use. The conversion magic was performed by the FemtoIDE. If you were to change a graphics, FemtoIDE will automatically regenerated the encoded data file.

Each node graphic has been given a name like *Node1*, *Node 2*, etc. To simplify the handling of the images, I have included a file named *Nodes.h* which simply includes all of the images into a single file and this, in turn, can be included in any class that needs to reference all of the node images.

To render the nodes out, we could write a lot of code that conditionally renders the correct sprite based on an input value like that below:

switch (nodeNumber) {

case 1:

PD::drawBitmap(x, y, Node1);

break;

case 2:

PD::drawBitmap(x, y, Node2);

break;

…

}

Or we can create an array that points to our images. Note the declaration is similar to the way we defined the array for the puzzle or the images although this time we have placed an asterisk after the data type ‘byte’ to indicate that the array contains pointers to the items rather than the actual values. Refer to *Stop pointing your finger at me!* for more information on pointers in C / C++.

const byte\* nodes[] = {nullptr, Node1, Node2, Node3, Node4, Node5,

Node6, Node7, Node8, Node9, Node10 };

I have also added a *nullptr* to the front of the array. As mentioned previously, arrays in C++ are zero based yet our nodes start numbering from 1 to 10. Adding the *nullptr* allows us to simply used the node value to look up the corresponding image – otherwise we would need to remember to subtract 1 each time!

Now we can simply render a node out with a single line of code:

PD::drawBitmap (x, y, nodes[3]);

Simple. The following function renders the board in the top left hand corner of the screen on a grid where each cell is 11 pixels wide by 11 pixels deep. This grid sizing allows space for a highlight square to be rendered around a node with a single pixel gap between the two.

In a later article, we will add additional functionality to ensure the board is properly centered on the screen regardless of the size of the puzzle itself. Likewise, scrolling functionality will be added for the puzzles that are too big to fit on the screen.

#define GRID\_WIDTH 11

#define GRID\_HEIGHT 11

void renderBoard() {

arduboy.clear();

// Draw nodes ..

for (int y = 0; y < PUZZLE\_Y; y++) {

for (int x = 0; x < PUZZLE\_X; x++) {

if (isNode(x, y)) {

sprites.drawOverwrite(x \* GRID\_WIDTH + 2, y \* GRID\_HEIGHT + 2, nodes[getNodeValue(x,y)], frame);

}

}

}

// Draw grid marks ..

for (int y = 0; y <= PUZZLE\_Y; y++) {

for (int x = 0; x <= PUZZLE\_X; x++) {

arduboy.drawPixel(x \* GRID\_WIDTH, y \* GRID\_HEIGHT, WHITE);

}

}

}

The functions isNode(x, y) and getNodeValue(x,y) will be discussed later.

## Next Article

In the next article, we will look at the actual game play itself.