Design

  Game Concepts

  Artwork

o   Large pipes

o   small pipes

o   numbers (5x4).

Programming

  Main structure

o   Various modes

  Init, select a level, no node selected, a node selected, game over

  Grid array

o   How the game play is recorded in the array[5][5]

  What are hex numbers?

  Masking bits and bytes?

  The concepts of Left-Right (or Right-left) and Top\_bottom or (bottom\_top).

o   Storing the puzzles in Progmem

o   Populating the array[5][5] from progmem

  Rendering the board

o   The joys or sprites that are not multiples of 8 bits high

  Using masks, or

  Top to bottom rendering (without masks)

  Game play

o   Navigating around the grid

o   Selecting an item and changing mode

  Checking the selection is valid?

  Navigating back on yourself

·         Using Left-Right (or Right-left) and Top\_bottom or (bottom\_top).

  Selecting a corresponding node

·         Is it game over?

·         Is it level over?

  Expanding the game to 9x9

o   introducing scrolling and screen offsets

o   configuration in progmem

o   render a scroll bar as a visual indicator

  Splash and level selects and sound

o   Saving the user progress in EEPROM

Debugging

  Debugging techniques

o   Refer other article

o   Rendering the board to debug

Launch

  Publishing code to git

  Preparing a HEX file

  Preparing a .Arduboy file

  Wait for adulation

Introduction

Only 10 node types on a 9x9

# Design

Game Concepts

## Artwork

Only 14 node types

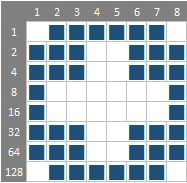
## Arduboy Sprites

Callout to “What are sprites”

For a small library, the Arduboy2 support for sprites is quit thorough. Looking at the code, it has been optimized for speed and as such imposes one restriction – sprites can be any width but they must be a multiple of 8 pixels high. This may sound like a major restriction but, as I will show later, the library also has some tricks in it to effectively get around this too. In this first installment of the article, I will restrict the sprites we use to simple 8 x 8 and 16 x 16 pixel graphics.

@Gaveno112 posted a great article in Volume 3 of the Arduboy Magazine and I encourage you to read it.

Take the simple sprite below. It is 8 pixels wide by 8 pixels high – an optimal size for a screen whose coordinates are a multiple of 8 (132 x 64 pixels).



Callout to “What’s n array”

The definition for a sprite consists of the sprites dimensions followed by an array (check out the *What’s an array?* Sidebar for more information) of numbers each of which describe a column of 8 pixels each.

The array definition for this sprite is shown below.

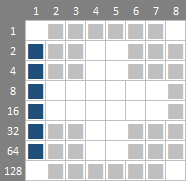
byte myFirstSprite[] = {

8, 8,

126, 231, 231, 129, 129, 231, 231, 126,

};

I have formatted the array to make it a little more readable. The first line contains the width and height of the array, in this case 8 pixels by 8 pixels. The remaining 8 bytes contain the pixel data for each column of the sprite and are calculated using a simple formula shown below.



Notice how I have labelled the side of the graphic with 1, 2, 4 and so on. To calculate that the first column’s value is 126, I simply added up all of the values adjacent to the pixels I want to be turned on (white). 2 + 4 + 8 + 16 + 32 + 64 = 126. The remaining columns are calculated in exactly the same way. If you haven’t realized it already, you have just had your first experiences with binary numbers (see the sidebar Decimal vs Hexadecimal vs Binary Numbers for more information).

Callout to “Decimal vs Binary vs Hexadecimal” Numbers

”

Sprites are often expressed in hexadecimal rather than decimal but this purely convention and makes no difference to the execution of the code.

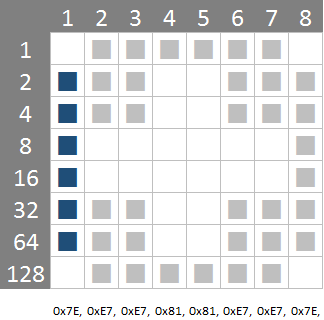
byte myFirstSprite[] = {

8, 8,

0x7E, 0xE7, 0xE7, 0x81, 0x81, 0xE7, 0xE7, 0x7E,

};

I have included my graphic design tool of choice in the repository – an Excel spreadsheet – and in it you will see a little trick I use to both design and calculate the array.



Below the graphic, my array is calculated automatically, using the formula shown below:

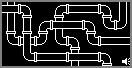
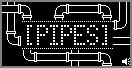
="0x"&DEC2HEX((SUMIF(B$2:B$9,"<>",$A$2:$A$9)),2)&","

I would be guessing that if you are considering programming an Arduboy that you will be able to pull apart an Excel formula and understand how it works!

## Node Graphics

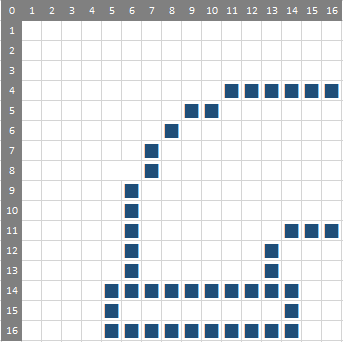
## Splash Screen Pipes

All good games have a splash screen (or a title screen) and Pipes is no exception. I wanted to have an animated sequence where the pipes where laid on the screen piece by piece before the final game title is shown.

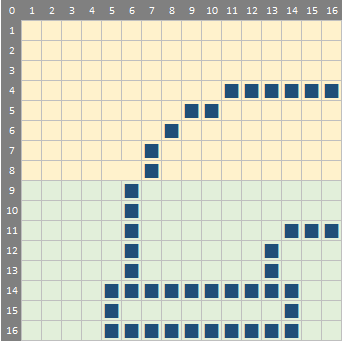
 

To do this, I prepared more graphics in Excel with these 16 x 16 pixels wide – again these dimensions are optimal for a screen that s 128 pixels wide and 64 pixels deep. The size allows 8 pipe images to be laid out horizontally and 4 images vertically.

A pipe elbow is shown below. You may have noticed that the center of the pipe itself is not central to the image and this is intentional to allow all of the pipe flanges to fit on an image. If I placed the pipe centrally, the flange would need to be pushed closer to the inner radius of the image – and it looked cramped – or the flange would need to be spread across this and any adjoining image (too hard!).



In the examples above, our sprites were 8 pixels wide by 8 pixels high. When defining a sprite that is 16 pixels or more in height, the array is specified from left to right then top to bottom. An example is shown in the graphic and array below:



byte logo\_elbow[] = {

16, 16,

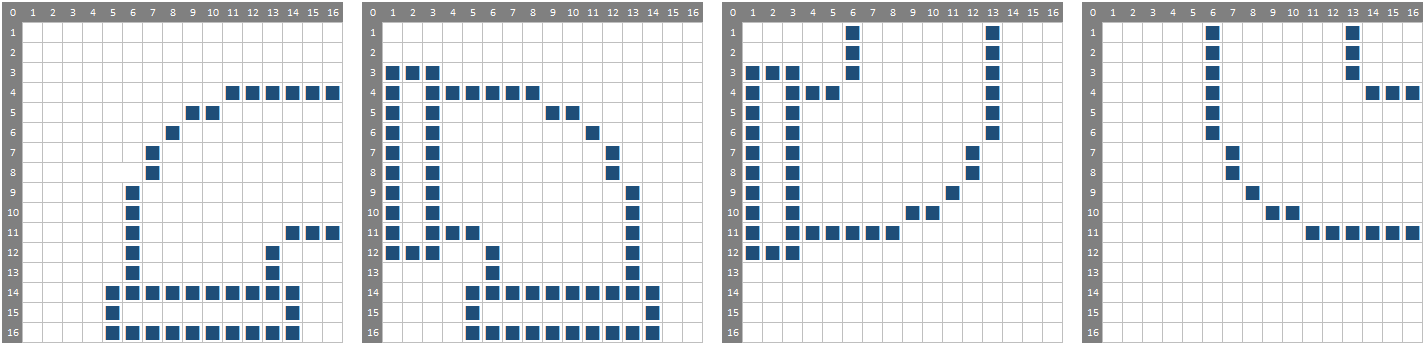
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xC0, 0x20, 0x10, 0x10, 0x08, 0x08, 0x08, 0x08, 0x08, 0x08,

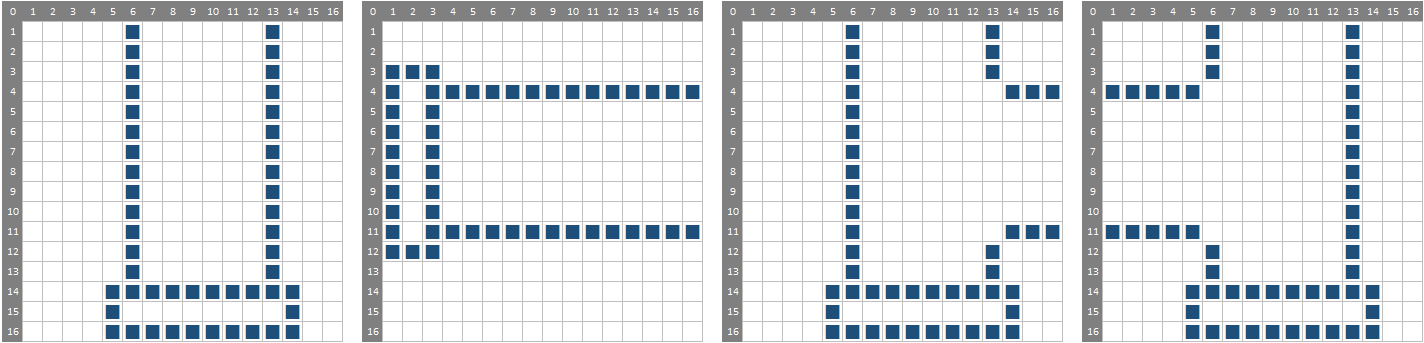
0x00, 0x00, 0x00, 0x00, 0xE0, 0xBF, 0xA0, 0xA0, 0xA0, 0xA0, 0xA0, 0xA0, 0xB8, 0xE4, 0x04, 0x04,

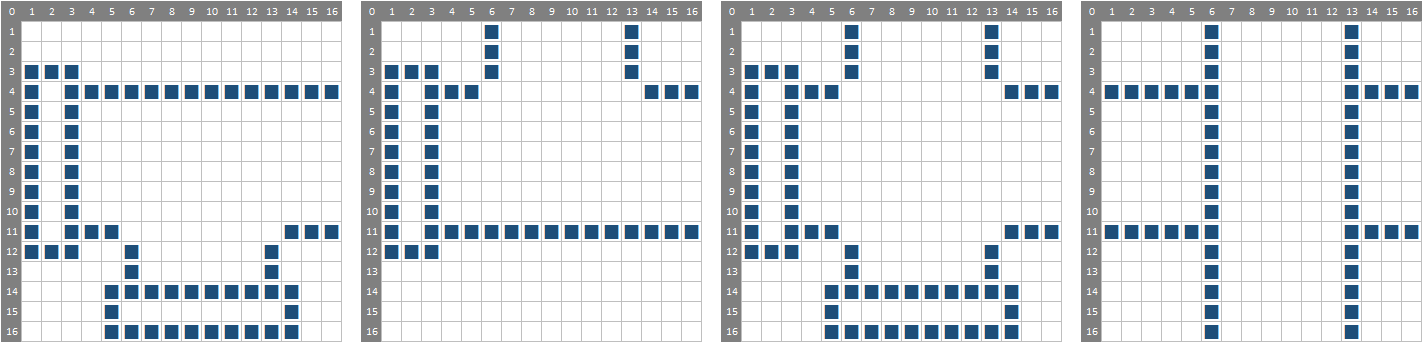
};

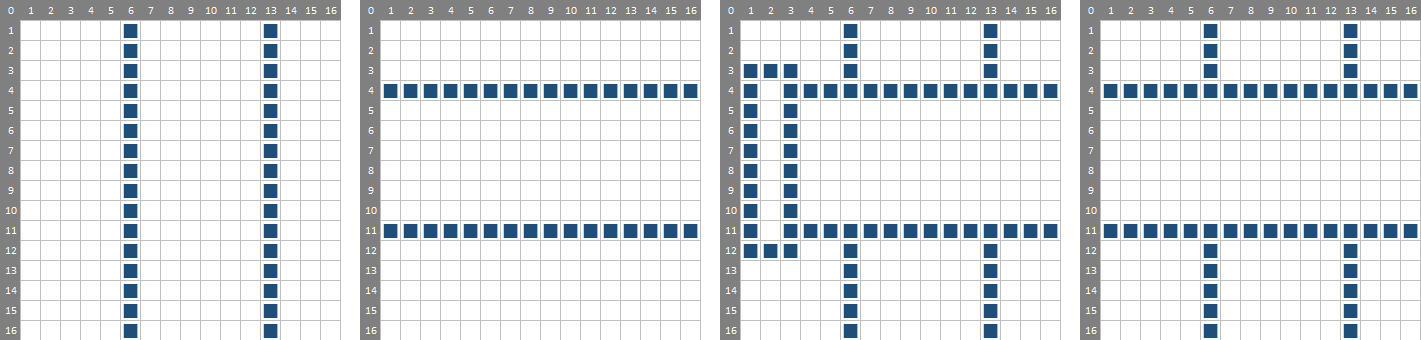
The array follows a similar pattern to the 8 x8 example - the first two bytes describe the width and height of the sprite (in this case 16 pixels wide x 16 pixels high) followed by the image data. The first row of the array - shaded in yellow - corresponds to the top row of the image and the second row corresponds to the green section in the sprite.

The images for all of the pipes are shown below. The corresponding data arrays have been populated in the Images.h file in the source files of the solution.









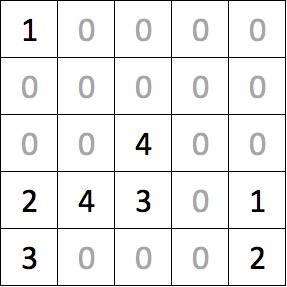
# Development

## 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 Arduboy offers. In the side bar Progmem vs RAM?, I discuss the two different types of memory that the Arduino 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 “Progmem vs RAM”

Consider a simple puzzle like the one shown below:



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 PROGMEM puzzles\_5x5[] = {

1, 0, 0, 0, 0,

0, 0, 0, 0, 0,

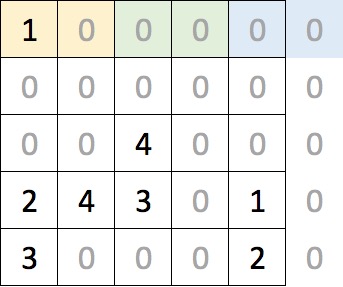
0, 0, 4, 0, 0,

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 PROGMEM 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 PROGMEM 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 PROGMEM, 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. This requires a little explanation – the pgm\_read\_byte function is used to read a single byte from the program memory at the location specified. There are other variations of this pgm\_read\_word() and pgm\_read\_dword() which read in an integer and long datatype respectively. The &puzzles\_5x5[0] indicates the address of the first element of array in memory (whereas simply specifying puzzles\_5x5[0] would refer to the value of the array’s first element).

const byte PROGMEM puzzles\_5x5[] = {

0x10, 0x00, 0x00,

0x00, 0x00, 0x00,

0x00, 0x40, 0x00,

0x24, 0x30, 0x10,

0x30, 0x00, 0x20,

};

byte byteRead = pgm\_read\_byte( &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] = 0xF0 | leftValue(byteRead);

x++;

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

if (x <= PUZZLE\_X) {

board[y][x] = 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 asdasdas above, we designed 10 nodes to support our game. These are defined in the *Images.h* source file and have been given names using the format of node\_1[]*.* 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:

sprites.drawOverwrite(x, y, node\_1, frame);

break;

case 2:

sprites.drawOverwrite(x, y, node\_2, frame);

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[] = {node\_1, node\_2, node\_3, node\_4, node\_5, node\_6, node\_7, node\_8, node\_9, node\_10 };

No we can simply render a node out with a single line of code:

sprites.drawOverwrite(x, y, nodes[3], frame);

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.

## Wiring it Together

The sample code at sdfsdfsd contains the code that is detailed in this article. I have strung the functions together to show how they work – it renders a splash screen, flips to the grid and back again. Unfortunately, no game play yet .. you will need to wait for the next installment of this article to see how this is implemented.

Alternatively, you can download the working game at dfewf and pull it apart to see how I did it.