Fix in code:

initBoard

Nodes names should be node\_1 to node\_10;

Nodes array should have a 0 as fist element.

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

Progmem vs RAM?

The Arduboy offers 32KB of Flash to store programs (known as Progmem) and 2.5KB of RAM to store variables used when executing a program. Once the Arduboy core library has been included, these figures reduce to approximately 22KB of flash and 1.3Kb RAM! Without forward planning, it is very easy to write programs that run out of either memory quickly.

Static data like the puzzle configurations in Pipes lend themselves to being stored in PROGMEM as they are relatively large (especially when storing hundreds of puzzles) and static.

PROGMEM acts differently to normal RAM and needs to be accessed using the commands pgm\_read\_byte(), pgm\_read\_word() and pgm\_read\_dword() for the data types byte (or char), int and long respectively.

  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.

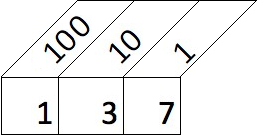
Introduction

Only 10 node types on a 9x9

Decimal vs Binary vs Hexadecimal Numbers

Humans have ten fingers and it’s no wonder we have a number system based on that fact. The Hindu-Arabic numbering system which we all use is known as a positional, decimal numbering system where each column increases in value by a factor of ten. This is totally different to Roman numerals which use a range of different letters to represent different values.

With a decimal number 137, we can break it down to 1 x 100, 3 x 10 and 7 x 1.

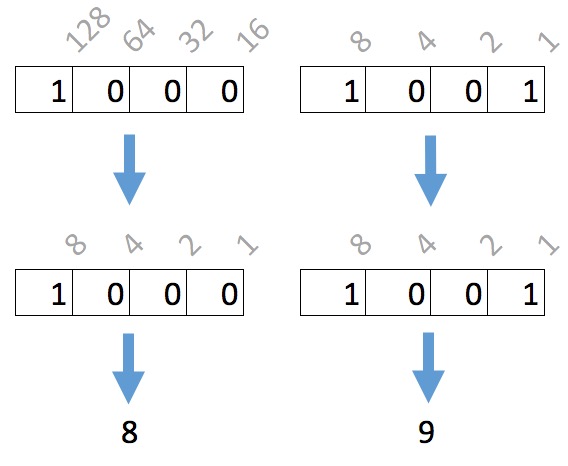


Computers store numbers as ones and zeroes in a system known as binary. Like the decimal system, each column is two times the value of the previous one. The decimal number 137 is represented as 10001001 in binary. The image below shows the number 10001001 with the decimal equivalent column value above. If you break it down, 1 x 128, 1 x 8 and 1 x 1 = 137 decimal.



But what about Hexadecimal? Using ones and zeroes to communicate with a computer is not really human-friendly so we convert the binary numbers into a number system known as hexadecimal (or hex for short) where each column is 16 times greater than the one before it. The numbers range from 0 to 9 and then from A to F to represent the 16 ‘numbers’. So the decimal number 3 is the same as the hex value 3 and the decimal number 9 is the same as the hex number 9. The decimal number 10 becomes the hex ‘number’ A, likewise 11 becomes B and 15 becomes F. The decimal number 16 is the equivalent of the hex number 10, 17 decimal equals 11 hex and so on.

But why hexadecimal? If you recall I said computers use zeroes and ones – referred to as a bit - and groups them together in lots of eight which are known as a byte (8 bits equals 1 byte). A byte can hold a value between 0000 0000 (decimal 0) and 1111 1111 (decimal 255). Four bits can hold a value between 0000 (decimal 0) and 1111 (decimal 15) – which just happens to align with the range of a single hexadecimal value. So we use hexadecimal as it is easy for humans to read and dead simple for a computer to convert into binary.



# Design

Game Concepts

## Artwork

Only 14 node types

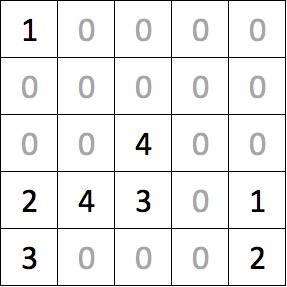
Arduboy Sprites

XXXX posted

# Development

## Fitting 100s of Puzzles in Memory

Right, so we have an idea and the graphics sorted out we need to consider how we will store the hundreds of puzzles in the limited memory that the Arduboy offers. 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,

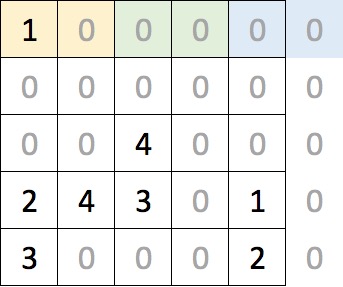
0, 0, 4, 0, 0,

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,

Bit Manipulation

As we discussed earlier, we can use hexadecimal notation when specifying arrays and save space by combining two values into one byte. When we read the data back we need to easily be able to split the two digits apart.

As we saw, the decimal value 137 was equal to the hexadecimal value 0x89 and the binary number 10001001.

If we want to retrieve the ‘8’ from 0x89 we can perform what is known as a ‘4 bit left shift’. A ‘left shift’ moves all values one location to the left while discarding the left most value. After four shifts, 0x89 becomes ..

1000 1001 (Original number of 0x89)

0100 0100 (1st shift)

0010 0010 (2nd shift)

0001 0001 (3rd shift)

0000 1000 (4th shift)

.. and convert this back to hex, we end up with 0x08. In C / C++ notation this is represented as 0x89 >> 4.

If we want the ‘9’ value from 0x89, we can perform some Boolean trickery by logically ANDing the value with 0x0F. This is best visualized using binary representations of the number:

1000 1001 (0x89)

AND 0000 1111 (0x0F)

---------

0000 1001 (0x09).

In C / C++ notation this is represented by the code 0x89 & 0x0F.

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] );

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 convulted 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[]*.* Unfortunately, this is only a naming convention and we want to be able

const byte\* nodes[] = {0, node\_1, node\_2, node\_3, node\_4, node\_5, node\_6, node\_7, node\_8, node\_9, node\_10 };