# Recap

In the last article, we managed to get a basic version of the Pipes application running – albeit a simple one that only plays a single 5 x 5 puzzle. In this instalment, we will add the code to handle puzzles of different sizes and the ability to select a level and puzzle number.

The sample code detailed in this article can be found in the repository at [https://github.com/filmote/Pipes\_Article3](https://github.com/filmote/Pipes_Article2)\_Pokitto. The game is totally playable now but needs more cowbells to make it complete. If you don’t understand that reference, you need to watch the SNL comedy sketch here <https://www.youtube.com/watch?v=DLeNQKk4uuI>

# Adding Levels

To date, our game play has been limited to two 5 x 5 puzzles. By now you are probably sick of them and can do both with your eyes closed! Let’s fix that immediately by adding a puzzles of different sizes to the *Puzzles.h* file. Due to the different sizes of the puzzles, they must all be declared in their own array. A subset of the puzzles is shown below.

const byte puzzles\_5x5\_count = 2;

const byte puzzles\_5x5[] = {

  0x10, 0x20, 0x40,

  0x00, 0x30, 0x50,

  0x00, 0x00, 0x00,

  0x02, 0x04, 0x00,

  0x01, 0x35, 0x00,

  0x10, 0x00, 0x00,

  0x00, 0x00, 0x00,

  0x00, 0x40, 0x00,

  0x24, 0x30, 0x10,

  0x30, 0x00, 0x20,

}

const byte puzzles\_6x6\_count = 2;

const byte puzzles\_6x6[] = { ... }

const byte puzzles\_7x7\_count = 2;

const byte puzzles\_7x7[] = { ... }

const byte puzzles\_8x8\_count = 2;

const byte puzzles\_8x8[] = { ... }

const byte puzzles\_9x9\_count = 2;

const byte puzzles\_9x9[] = {

0x00, 0x00, 0x00, 0x00, 0x00,

0x01, 0x23, 0x00, 0x00, 0x00,

0x00, 0x02, 0x40, 0x43, 0x00,

0x00, 0x00, 0x00, 0x05, 0x00,

0x06, 0x06, 0x10, 0x00, 0x00,

0x75, 0x00, 0x00, 0x09, 0x80,

0x08, 0x70, 0x00, 0x00, 0x00,

0x09, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00,

...

}

## Loading the Board (Revisited)

The initBoard() function has undergone a few changes to date and the final tweak is to change it to accept a level parameter in addition to the original puzzle number one. The original code in the function remains unchanged however additional functionality has been added to allow the program to read from the different puzzle arrays we declared above.

The puzzleType parameter is a byte which indicates the size of the puzzle from 5 to 9. Using this number, we can work out how many bytes to read per puzzle. In our original code, this value was hardcoded to 15 bytes which equated to 3 bytes per row multiplied by 5 rows. As you may have worked out by now, a 6 x 6 puzzle also uses three bytes per row but has 6 rows.

When determining how many bytes to read, we determine whether the puzzle size is an even number - puzzleType % 2 == 0 - and, if so, we need to read in one half of the width multiplied by the number of row bytes - (puzzleType / 2) \* puzzleType. If the puzzle size is an odd number, we need to round the puzzle size up to the next even number, halve that and multiply in by the number of rows - ((puzzleType + 1) / 2) \* puzzleType.

Callout to “Modulus in C / C++”

When reading the actual puzzle data from the array, a simple switch statement is used to retrieve from the correct array.

#define PUZZLE\_5X5 5

#define PUZZLE\_6X6 6

#define PUZZLE\_7X7 7

#define PUZZLE\_8X8 8

#define PUZZLE\_9X9 9

void initBoard(byte puzzleType, byte puzzleNumber) {

byte x = 0;

byte y = 0;

byte byteRead = 0;

byte bytesToRead = (puzzleType % 2 == 0 ? (puzzleType / 2) \* puzzleType :

((puzzleType / 2) + 1) \* puzzleType);

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

switch (puzzleType) {

case PUZZLE\_5X5:

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

break;

case PUZZLE\_6X6:

byteRead = pgm\_read\_byte(&puzzles\_6x6[i]);

break;

case PUZZLE\_7X7:

byteRead = pgm\_read\_byte(&puzzles\_7x7[i]);

break;

case PUZZLE\_8X8:

byteRead = pgm\_read\_byte(&puzzles\_8x8[i]);

break;

case PUZZLE\_9X9:

byteRead = pgm\_read\_byte(&puzzles\_9x9[i]);

break;

}

...

}

}

## Rendering the board.

The existing code to render the board will happily render puzzles of any size – however with the larger puzzles portions of the board will not be visible. Later in this article, we will look at how we can make the board scroll as the user moves around.

# Storing the Player’s Progress.

My version of the Pipes game will not let a player advance to the next puzzle in each level until they have completed the one before that. I store and retrieve the progress of each level in the EEPROM using functionality provided as part of the Pokitto library, known as *Cookies*.

In Pokittoland, a cookie is a class that exposes a bunch of variables relevant to your game or application that you want to store in the EEPROM. As you can see below, the cookie used for Pipes has five variables that are used to store the progress of each level. I have also defined another field, called *initialised* that is used to determine whether the cookie has been initialized before its first use.

class GameCookie : public Pokitto::Cookie {

public:

byte initialised;

byte level5x5 = 0;

byte level6x6 = 0;

byte level7x7 = 0;

byte level8x8 = 0;

byte level9x9 = 0;

public:

void initialise() {

initialised = COOKIE\_INITIALISED;

level5x5 = 0;

level6x6 = 0;

level7x7 = 0;

level8x8 = 0;

level9x9 = 0;

saveCookie();

}

};

You may also have noted that the cookie extends *Pokitto::Cookie*. By extending this class, our cookie knows how to save itself using the inherited *saveCookie()* method.

The logic for creating and initializing the cookie is shown below and is taken from the *main.cpp* file which, as discussed in the first article, is used to initialize the Pokitto before going into an endless loop of gameplay.

In addition to the code Pokitto library, the program now includes the *PokittoCookie* library and our own *GameCookie* implementation. An instance of a *GameCookie* is created followed by a call to an inherited method, *begin()*. This method, accepts a unique name for the cookie, the size of the cookie and the cookie value itself. The method first checks to see if a cookie exists with that name and, if so, populates the cookie with values retrieved from the EEPROM.

If no matching cookie is found, the method reserves enough memory in the EEPROM to store the cookie. Cookies can be any length and this process will allocate enough memory to store the entire cookie.

#include "Pokitto.h"

#include "PokittoCookie.h"

#include "src/utils/GameCookie.h"

#include "src/Game.h"

#define COOKIE\_INITIALISED 34

Game game;

GameCookie cookie;

int main() {

cookie.begin("PIPES", sizeof(cookie), (char\*)&cookie);

Once the cookie has been created and loaded, the *initialised* variable is tested to see if the cookie has just been created. If it has, the variables are often filled with random data that was read from the EEPROM rather than known values. The value of *COOKIE\_INITIALISED* is irrelevant but allows us to test the retrieved cookie for a known value.

// Has the cookie been initialised?

if (cookie.initialised != COOKIE\_INITIALISED) {

cookie.initialise();

}

Finally, the created and initialised cookie can be passed to the game class and the game started!

// Play game!

game.setup(&cookie);

while (PC::isRunning()) {

if (!PC::update()) continue;

game.loop();

}

return 0;

}

# Selecting a Level

Now that we have all of the levels defined and have a way of tracking progress, we need a way for the player to select which of the five levels they would like to play.



The code below shows the button handling for the menu and is taken from the file *Game\_Menus.cpp*. It stores the selected menu item in a variable called levelSelect\_selectedItem which is incremented or decremented as the user clicks the up or down buttons within the range of the six level options. If the user presses the ‘B’ button, control is returned back to the splash screen.

void Game::levelSelect() {

clearHighlightAndSelection();

if (PC::buttons.pressed(BTN\_UP) && levelSelect\_selectedItem > 0) {

levelSelect\_selectedItem--;

}

if (PC::buttons.pressed(BTN\_DOWN) && levelSelect\_selectedItem < sizeof(levels) – 1) {

levelSelect\_selectedItem++;

}

if (PC::buttons.pressed(BTN\_B)) {

gameState = STATE\_SPLASH\_INIT;

}

…

If the player presses the ‘A’ button, the progress of the selected level is retrieved from the EEPROM via the *GameCookie*. If they select a level where they have completed at least one of the puzzles, they are invited to clear the level and start again or continue on from the last completed puzzle. If the player selects a new level, they are first puzzle is automatically selected.

In either scenario the gameplay variable is updated to indicate which routine the main loop() should call on the next pass. I have also included a second variable, prevState, and have set this to STATE\_LEVEL\_SELECT. This variable will be used later to determine how a user got to the puzzle select screen and is described in the section *Selecting a Puzzle* below.

if (PC::buttons.pressed(BTN\_A)) {

puzzle.level = levels[levelSelect\_selectedItem];

puzzle.index = 0;

if (cookie->getLevelIndex(puzzle.level) > 0) {

gameState = STATE\_PUZZLE\_SELECT;

prevState = STATE\_LEVEL\_SELECT;

puzzleSelect\_selectedItem = 0;

}

else {

gameState = STATE\_INIT\_GAME;

}

}

The screen is rendered using the code below. The *renderBackground()* function is the same one we used when creating the splash screen. After rendering the background, two rectangles are rendered and filled in white and then again in a slightly smaller size in dark grey.

renderBackground();

PD::setColor(1);

PD::fillRect(OVERALL\_X\_POS, OVERALL\_Y\_POS, OVERALL\_WIDTH, OVERALL\_HEIGHT);

PD::setColor(8);

PD::fillRect(OVERALL\_X\_POS + 1, OVERALL\_Y\_POS + 1, OVERALL\_WIDTH - 2, OVERALL\_HEIGHT - 2);

renderLevelDetail(ITEM\_X\_POS, ITEM\_1\_Y\_POS, levels[0], levelSelect\_selectedItem == 0);

renderLevelDetail(ITEM\_X\_POS, ITEM\_2\_Y\_POS, levels[1], levelSelect\_selectedItem == 1);

renderLevelDetail(ITEM\_X\_POS, ITEM\_3\_Y\_POS, levels[2], levelSelect\_selectedItem == 2);

renderLevelDetail(ITEM\_X\_POS, ITEM\_4\_Y\_POS, levels[3], levelSelect\_selectedItem == 3);

renderLevelDetail(ITEM\_X\_POS, ITEM\_5\_Y\_POS, levels[4], levelSelect\_selectedItem == 4);

The renderLevelDetail() function renders a single menu item at positions X and Y. If the variable highlight is true, it first renders a white filled rectangle and sets the text colour to grey on a white background. If the row is not highlighted, it sets the text colour to yellow on dark grey.

void Game::renderLevelDetail(byte x, byte y, byte level, byte highlight) {

if (highlight) {

PD::setColor(1);

PD::fillRect(x - 1, y - 1, ITEM\_WIDTH, ITEM\_HEIGHT);

PD::setColor(8, 0);

}

else {

PD::setColor(4, 8);

}

PD::setCursor(x, y);

switch (level) {

case PUZZLE\_5X5:

PD::print("Practice");

PD::setCursor(x + 52, y);

PD::print(cookie->getLevelIndex(PUZZLE\_5X5) < 10 ? " " : "");

PD::print(cookie->getLevelIndex(PUZZLE\_5X5), 10);

PD::print("/");

PD::print(puzzles\_5x5\_count, 10);

break;

case PUZZLE\_6X6: ...

case PUZZLE\_7X7: ...

case PUZZLE\_8X8: ...

case PUZZLE\_9X9: ...

}

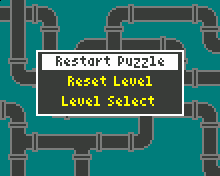
arduboy.setTextColor(WHITE);

arduboy.setTextBackground(BLACK);

}

# Selecting a Puzzle

As stated earlier, if the player selects a level where they have completed at least one of the puzzles, they are invited to clear the level and start again or continue on from the last completed puzzle. If the player selects a new level, they are first puzzle is automatically selected. The puzzle selection screen looks like this:



The code below handles the selection of a puzzle. I will not dissect it as it is functionally similar to the level section above but there are a few variations.

The first is when the user selects the option to continue playing the next puzzle from the level they have chosen. If the retrieved EEPROM setting reveals that the user has completed all puzzles in the current level, then they are presented with the last puzzle again.

You will also notice that when rendering the first menu item, the words *Continue Playing* are rendered if the player has progressed from the level selection screen. If they have pressed the ‘B’ button whilst playing an actual puzzle, they will be presented with the option *Restart Puzzle* instead.

void Game::puzzleSelect() {

clearHighlightAndSelection();

if (PC::buttons.pressed(BTN\_UP) && puzzleSelect\_selectedItem > 0) {

puzzleSelect\_selectedItem--;

}

if (PC::buttons.pressed(BTN\_DOWN) && puzzleSelect\_selectedItem < 1) {

puzzleSelect\_selectedItem++;

}

if (PC::buttons.pressed(BTN\_B)) {

gameState = STATE\_LEVEL\_SELECT;

}

if (PC::buttons.pressed(BTN\_A)) {

if (puzzleSelect\_selectedItem == 1) {

puzzle.index = 0;

cookie->updateLevel(puzzle.level, puzzle.index);

}

if (puzzleSelect\_selectedItem == 0) {

// If all puzzles in the current level are completed, simply re-show the last puzzle ..

if (cookie->getLevelIndex(puzzle.level) == getNumberOfPuzzles(puzzle.level)) {

puzzle.index = cookie->getLevelIndex(puzzle.level) - 1;

}

else {

cookie->getLevelIndex(puzzle.level);

}

}

gameState = STATE\_INIT\_GAME;

}

// Render menu ..

renderBackground();

PD::setColor(1);

PD::fillRect(OVERALL\_X\_POS, OVERALL\_Y\_POS, OVERALL\_WIDTH, OVERALL\_HEIGHT);

PD::setColor(8);

PD::fillRect(OVERALL\_X\_POS + 1, OVERALL\_Y\_POS + 1, OVERALL\_WIDTH - 2, OVERALL\_HEIGHT - 2);

const char s4[] = {" Level Select"};

if (prevState == STATE\_LEVEL\_SELECT) {

renderPuzzleOption(ITEM\_X\_POS, ITEM\_1\_Y\_POS, s1, (puzzleSelect\_selectedItem == 0));

}

else {

renderPuzzleOption(ITEM\_X\_POS, ITEM\_1\_Y\_POS, s2, (puzzleSelect\_selectedItem == 0));

}

renderPuzzleOption(ITEM\_X\_POS, ITEM\_2\_Y\_POS, s3, (puzzleSelect\_selectedItem == 1));

renderPuzzleOption(ITEM\_X\_POS, ITEM\_3\_Y\_POS, s4, (puzzleSelect\_selectedItem == 2));

}

# Enabling Scrolling and Centering the Board

To complete the game play, we need to enable scrolling so that the player can reach the lower rows of the bigger puzzles. While modifying the code to support this, we will add some extra functionality to center the board in the screen and to add a ‘scrollbar’ on the right hand side that indicates where the player’s highlighted cell is relative to the rest of the board. When finished, the game play will look like this :

To start with, I have defined another array in the *Puzzles.h* file that stores a row of information per puzzle type. For each puzzle type we define four parameters that include the X and Y position at which to render the board and a ‘unit length’ and ‘bar length’ for the scrollbar. The scrollbar parameters are used simulate the behaviour of a scroll bar where the ‘slider’ length is proportional to the visible screen length compared to the full puzzle size. More about that later.

const byte puzzles\_details[] = {

27, 16, 0, 0, // 5x5

21, 10, 0, 0, // 6x6

15, 4, 0, 0, // 7x7

6, 4, 7, 68, // 8x8

1, 4, 6, 64, // 9x9

};

The puzzle structure that is populated at the start of a game and maintains game play information has been extended to capture this additional information. When populating the puzzle structure via the *initBoard()* function, these additional parameters are retrieved and stored along with the puzzle definition itself.

struct Slider {

byte unit; // Number of pixels / row for the slider.

byte overall; // Height of the slider portion in pixels.

};

struct Puzzle {

byte level;

byte index;

Node maximum;

Node offset;

Slider slider;

byte board[9][9];

}

puzzle;

In the previous article we looked at the two functions that handle the node selection and pipe laying, play\_NoSelection() and play\_NodeSelected() respectively. If you recall, both of these functions finished with a call to renderBoard() which simply rendered the board in the top left corner of the page. To cater for the offsets, I have added some additional parameters for them as shown below:

void renderBoard(int xOffset, int yOffset, byte topRow) {

PD::fillScreen(11);

...

for (int y = 0; y < puzzle.maximum.y; y++) {

for (int x = 0; x < puzzle.maximum.x; x++) {

if (isPipe(x,y)) {

PD::drawBitmap((x \* GRID\_WIDTH) + xOffset, (y \* GRID\_HEIGHT) + yOffset,

connectors[getPipeValue(x, y)]);

}

}

}

...

}

When rendering the any component of the board, the X and Y offsets are included in position calculations.

To implement the scrolling, I have altered the play\_NoSelection() and play\_NodeSelected() to calculate a ‘Y’ offset that takes into account the current highlighted row. The function to calculate the top row to display is shown below. The code is very similar to that which we utilised when displaying the levels menu and aims to keep the highlighted cell in the middle of the screen where possible.

byte calculateTopRow() {

byte topRow = 0;

if (player.highlightedNode.y <= 2) {

topRow = 0;

}

else {

if (player.highlightedNode.y >= 3 && player.highlightedNode.y <= puzzle.maximum.y - 7) {

topRow = player.highlightedNode.y - 2;

}

else {

topRow = (puzzle.maximum.y - 7 >= 0 ? puzzle.maximum.y - 7 : 0);

}

}

return topRow;

}

The play\_NoSelection() and play\_NodeSelected() utilize the new calculateTopRow() function to determine the top row to display. When the top row evaluates to zero, the rendering of the puzzle starts at the offset position defined in puzzle.offset.x and offset.position.y, as retrieved from the puzzles\_details[] array. As the top row increases the calculated offset gets increasingly larger and negative which forces the board to be rendered from a negative ‘Y’ position.

renderBoard(puzzle.offset.x, puzzle.offset.y - calculateTopRow() \* GRID\_HEIGHT,

calculateTopRow());

The renderBoard() routine will simply render the items regardless of their ‘Y’ values. The Pokitto library contains code that detects when an image will be completely rendered outside the dimensions of the screen. However, it needs to calculate this and this adds some overhead to the rendering. This does not pose a problem for a game like Pipes but would be a consideration for a fast moving, action game.

# Rendering a Scroll Bar

Our last little visual trick for his edition of the series is to add a scroll bar that will indicate to the player that the puzzle they are playing is bigger than what the Arduboy can display and where their highlighted cell is relative to the board.

The renderBoard() function is altered once again to include some logic to render the scrollbar. The 5 x 5 puzzle does not need a scroll bar but all other puzzle sizes do. To cater for this, the rendering of the puzzle is conditioned by the puzzle.slider.unit value (one of the values retrieved from the puzzles\_details[] array earlier) being greater than 0. You may have noticed earlier that the 5 x 5 puzzle had this value set to zero.

The ‘slider’ part of the toolbar is positioned on the scrollbar to indicate the relative position of the highlighted cell. This is done by calculating the top position using the topRow parameter multiplied by the puzzle.slider.unit value. The height of the slider is a constant per puzzle type.

void renderBoard(int xOffset, int yOffset, byte topRow) {

…

if (puzzle.slider.unit > 0) {

PD::setColor(0);

PD::fillRect(SCROLLBAR\_X, SCROLLBAR\_Y, SCROLLBAR\_WIDTH + 1, SCROLLBAR\_HEIGHT);

PD::setColor(9);

PD::drawRect(SCROLLBAR\_X, SCROLLBAR\_Y, SCROLLBAR\_WIDTH, SCROLLBAR\_HEIGHT);

PD::fillRect(SCROLLBAR\_X + 2, SCROLLBAR\_Y + 5 + (topRow \* puzzle.slider.unit),

SCROLLBAR\_WIDTH - 3, puzzle.slider.overall);

// Top arrow ..

PD::drawBitmap(SCROLLBAR\_X + 2, SCROLLBAR\_Y + 2, ArrowUp);

// Bottom arrow ..

PD::drawBitmap(SCROLLBAR\_X + 2, SCROLLBAR\_Y + SCROLLBAR\_HEIGHT - 3, ArrowDown);

}

## Next Month

In the next article, we will add some additional features to make the game a little more professional – sound, an animated splash screen and other decorations. If we have space, I will discuss how to create a distributable *BIN* file and a *POP* file.