Daniel Andjelic, Liam Kynaston, Torin Stephens
CPSC 1490 – 001
15/04/2018

CPSC 1490 – Final Project: *Cavern of Imps*

Abstract:

For this project we set out to make an Arduino controlled, turn-based video game. This project's primary hardware consists of an Arduino Due, three push-buttons, four limit switches and a 2.2" 18-bit colour TFT LCD, wired using jumper wires and a breadboard (see figure 1, below). Inspiration for the game came from the Legend of Zelda franchise and the Rogue-like genre.

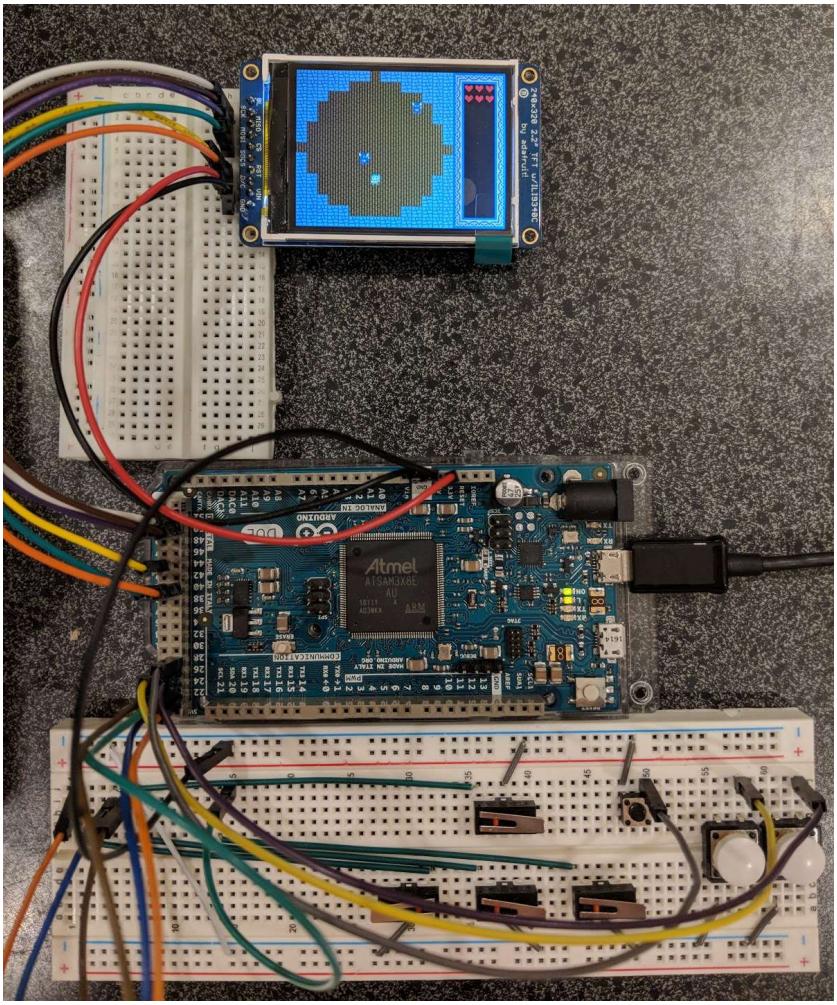

Figure 1. Image of final hardware setup.

The game is set in a dungeon consisting of 9 rooms, in the shape of a square. Each area (left, right and centre) of the dungeon has a unique theme. The dungeon has a variety of enemies, breakable

objects, weapons, traps, and possibly a few secrets. The left side of the display contains the playable space, while the right side displays the player character's health, and has space for an inventory and a world map. The game is turn based, but it can be changed to real-time with the use of timers. The purpose of the project was to make a fun handheld game which can be used to relax and de-stress students

This project can be used as a basis for a variety of video games, including Rogue-likes, Zelda-like games, RPGs, and more. These are all possible with the current code, by adjusting the world map matrix and enemy positions.

RCGs:

	Explanation	Label
Requirement 1	Working screen	**R1**
Requirement 2	Player controls	**R2**
Requirement 3	Character controls	**R3**
Requirement 4	Enemies	**R4**
Requirement 5	AI	**R5**
Requirement 6	Sprites/art assets	**R6**
Requirement 7	Boss battle	**R7**
Requirement 8	Health system	**R8**
Constraint 1	Commands in LCD library	**C1**
Constraint 2	Budget	**C2**
Constraint 3	Screen resolution	**C3**
Goal 1	Multiple weapons	**G1**
Goal 2	Multiple enemy types	**G2**

Daniel Andjelic, Liam Kynaston, Torin Stephens
CPSC 1490 – 001
15/04/2018

Goal 3	Breakable objects	**G3**

Design Process:

Hardware:

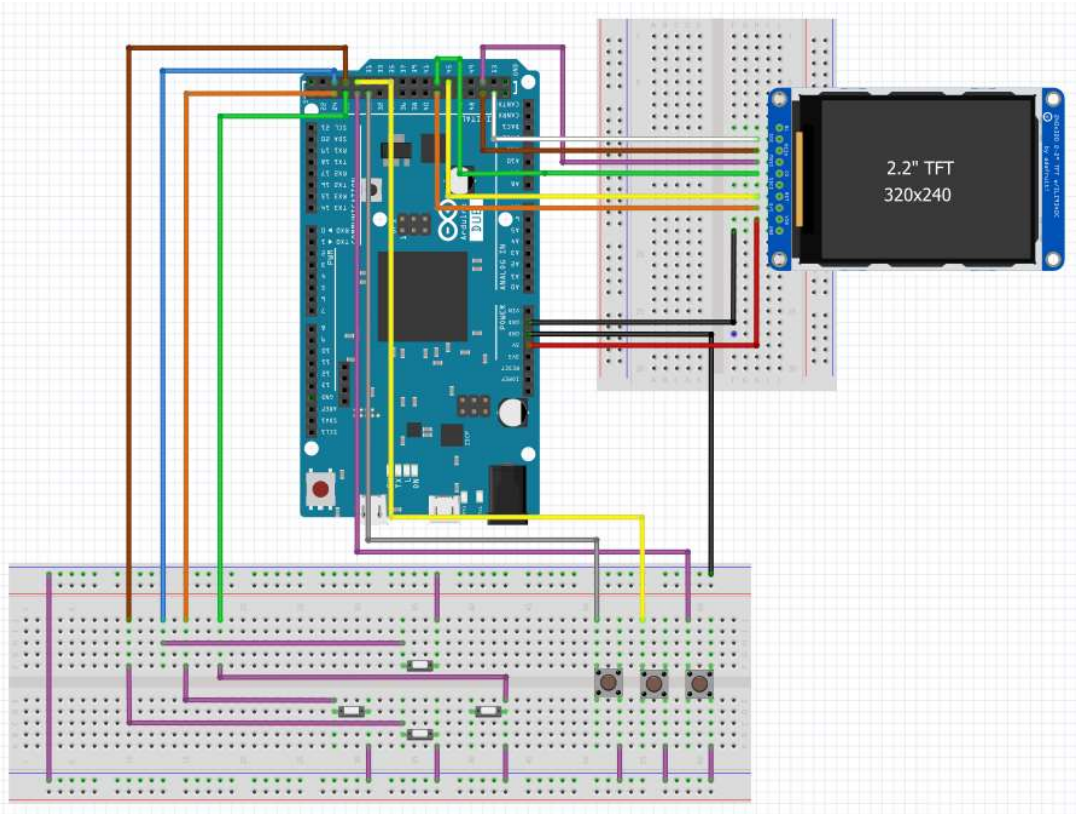

Figure 2. Fritzing diagram of hardware setup

R1: For our project we wanted a top down game with colour graphics. To fulfill this requirement we needed a screen with better visual capabilities than the supplied character LCD screen. To fulfill this requirement, we researched a variety of screens for one which would

meet our visual requirements as well as our budget. We concluded that Adafruit's 2.2" 18-bit colour TFT LCD best met this requirement while remaining considerate of C2.

R2: For our game, we wanted intuitive controls that were comfortable to use. To meet this requirement we searched online and in person for a variety of buttons for ones that felt tactile and were responsive. We decided to get four limit switches for directional control, and three pushbuttons for movement, attacking, and weapon switching.

C2: After purchasing these items, our total spending was $47.15. This is below our budget of $50. Thus, the project was kept within C2.

Game Design and Implementation:

The world map for the game is stored in a four-dimensional array, "wMap". The first two dimensions indicate which room a tile is in, while the last two dimensions store the coordinates of the tile within the room. This allows all tiles to be stored within a single matrix, while also allowing functions to easily check a tile or character's position within a room.

Player input and movement:

R3: To allow for player input, we used two separate techniques. These helped satisfy R2. Both techniques modify an object of the class "CHARACTER". Objects of this class were given the following properties:

- "row" - row of the room within the world map (first dimension in the world map)
- "col" - column of the room within the room map (second dimension in the world map)
- "posX" - x-coordinate within a room (fourth dimension in the world map)
- "posY" - y-coordinate within a room (third dimension in the world map)
- "cDirection" - direction the character is facing
- "cTile" - the tile type of the player's current position

The player character is initialized in the bottom centre row, in the bottom centre of the screen (wMap[2][1][15][7]).

Daniel Andjelic, Liam Kynaston, Torin Stephens
CPSC 1490 – 001
15/04/2018

To choose what direction the player character faces, we created a function ("faceDirection") that sets the player character's cDirection property based on which of four limit switches is pressed.

To control attacks and movement, we added two if statements to the main loop. These each check if a button has been pressed. If the attack button has, a function, "heroAttack", is called, which allows the player to attack enemies (see R8 and G1). If the movement button has been pressed, another function is called, "heroMove".

heroMove works by incrementing or decrementing one of the player character's xPos or yPos properties, based on which direction the character is facing. Before it does this, though, it checks if the player character can move into the new position. If it can, the player character is placed in the new, adjusted position. The character's previous position is set to the player character's cTile property, and the tile type of the character's new position is stored as its cTile property. If the player cannot move into the new position (ex., a wall is in that position), the player character's position is not changed.

Gameplay:

R4: To satisfy R4, we created additional character objects, using row and col properties set to each character's room, and posX and posY properties to set their positions within that room. Enemies are differentiated from the player character by the new CHARACTER property "type", which is 0 for the player character, and was initially set to 1 for all enemies.

R5: Meeting R5 involved creating functions for enemy movements ("enemyMove"). To allow this function to easily be applied to all enemies, we created a CHARACTER matrix ("Characters") that stores all objects of the CHARACTER class. This made it so that another function ("enemiesMove") could call enemyMove for each enemy by simply incrementing position within the matrix.

The function alters one of the character's posX or posY values by one. When the distance between the enemy and the player character ("D") is greater than six the alteration is random. When D is less than or equal to six, the function chooses the

change that will minimize D. If D is exactly one, the enemy will attack the player, decreasing the player's health by a set amount.

Before the enemy moves, the function checks if the enemy is able to move into the selected position (i.e., that position is currently occupied by a floor tile). If D is less than six and not equal to one, the function also checks if the enemy was in that position the previous turn (a value indicating this is stored as the character property "lastMove").

If the enemy cannot move into the selected tile, or if it occupied that tile last turn, the function selects another adjustment to either posX or posY (if D <= 6, it selects the next best option to minimize D). This allows enemies to navigate around most obstacles by preventing them from moving between two positions indefinitely. An example is shown in figure 3, below.

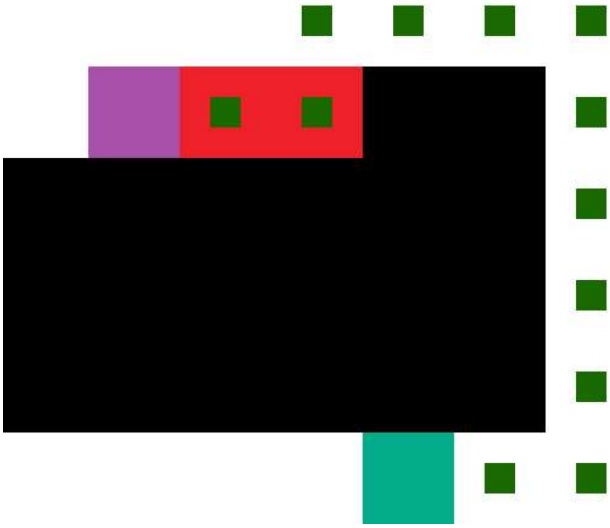

Figure 3. Example of improvement offered by storing previous move.
(Red – without lastMove; Green – using lastMove)

Every time the player moves or attacks, all enemies in the room will move or attack, as well. Then the player can move or attack again. This cycle makes up one turn in the game.

R7: One issue in implementing the boss battle was the boss's size and how he is drawn. The boss occupies nine tiles, so it didn't seem practical to have him move around the room, as he would leave limited space for the player to move around in. Instead, the boss spawns lava on random tiles in the room. The boss is activated either by attacking him or walking a set distance into the room he occupies. He spawns lava after every player attack or move, and the lava lasts for two turns. To prevent players from attacking the boss without moving, he spawns lava in different ways depending on which weapon he is attacked with. To increase difficulty, the boss "enrages" when ⅓ of his health remains. When the boss is enraged, lava is spawned at an increased rate, decreasing the tiles available for the character to occupy without taking any damage.

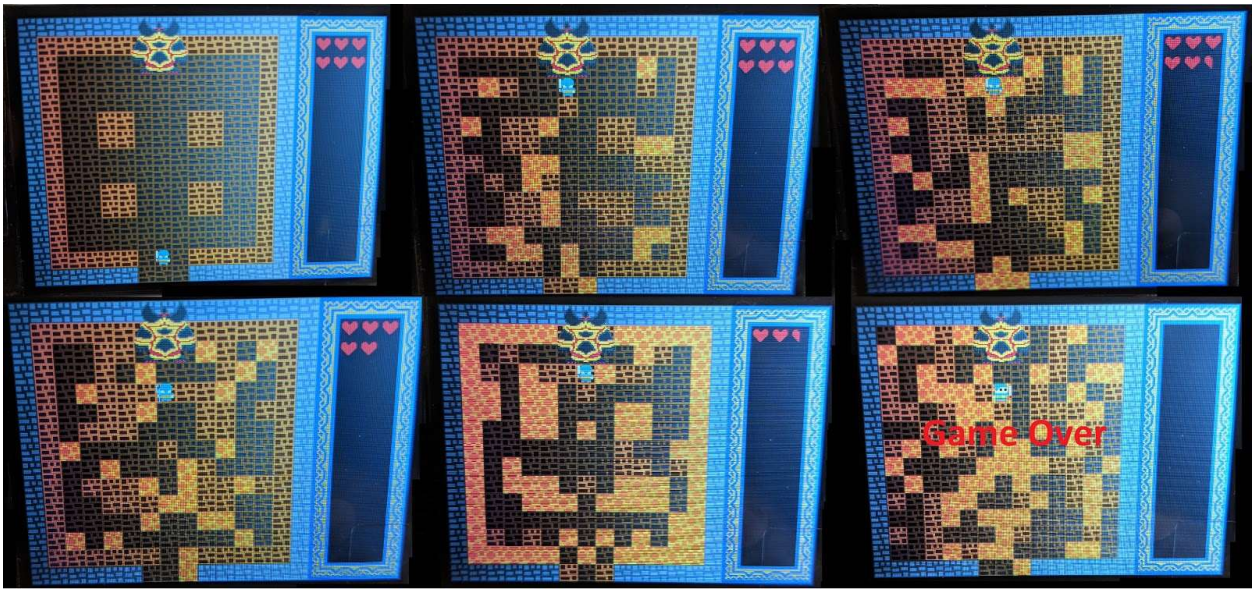

Figure 4. Boss room at various stages.

R8: R8 was met by implementing two new CHARACTER properties, "health" and "cStatus". The player character's health property is decreased when an enemy attacks (by one or two) and when the player steps in lava (by two) or on a trap (by one). Player health is increased when the player moves onto a heart, up to a maximum of 24. An enemy's health is modified when the player attacks that enemy (by one if using the bow, by two if using the trident, and by three if using the sword). The boss's health is

not stored as a property of a character object. Instead, it is a global variable. However, it is decreased in the same way a regular enemy's health is.

If the player character's health reaches zero, its cStatus is set to zero, and the player is no longer able to change direction, move, attack, or switch weapons. At this point, a "game over" message is printed on-screen. The Arduino must be reset to start the game over. When an enemy's health reaches zero, the enemy's cStatus is set to zero. It is then removed from the world map array, its cTile is placed where it was, and there is a chance of a heart being placed over that.

G1: Additional weapons were added by adding two boolean global variables, "trident" and "bow". These are initialized to false. When the player character encounters a 7 on the world map, one of the two booleans is set to true – if the world row is 0, "trident", and if the world row is 2, "bow". Another button was added to the player controls, and was set up similarly to the attack and move buttons. Pressing this button will increment a global variable, "weapon", until it reaches 2, at which point pressing the button will set "weapon" to 0. If "trident" and/or "bow" is false, the values 1 and/or 2 will be skipped, respectively.

To implement different behaviours for the additional weapons, if statements were added to the "playerAttack" function. When "playerAttack" is called, one if statement is executed, depending on the value of "weapon": if "weapon" is 0, and if there is an enemy directly in front of the player, it has its health reduced by 3; if "weapon" is 1, and if there is an enemy in front of the player, but exactly 2 tiles away from the player, it has its health reduced by 2; if "weapon" is 2, and if there is an enemy in front of the player, at any distance from the player, it has its health reduced by 1.

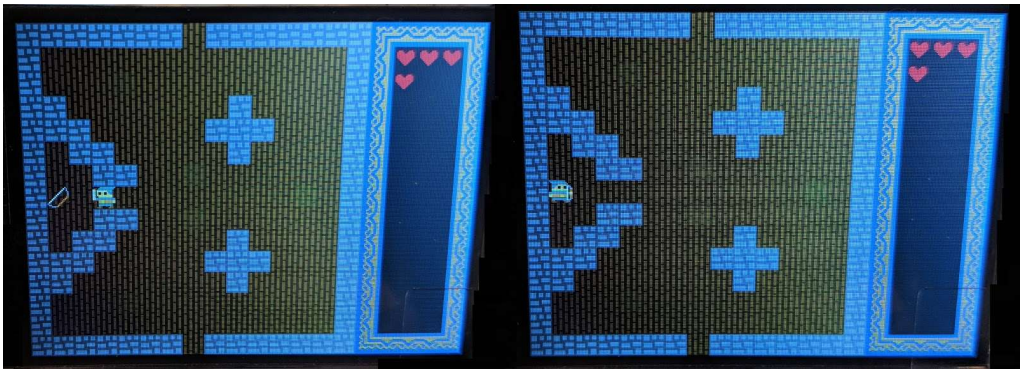

Figure 5. The bow on the map (left) and in use (right).

G2: To introduce additional enemy types, we added new character types (2, 3, 4 and 5). This allowed us to vary the appearance of enemies. It also let us have different enemies deal different amounts of damage to the player character's health.

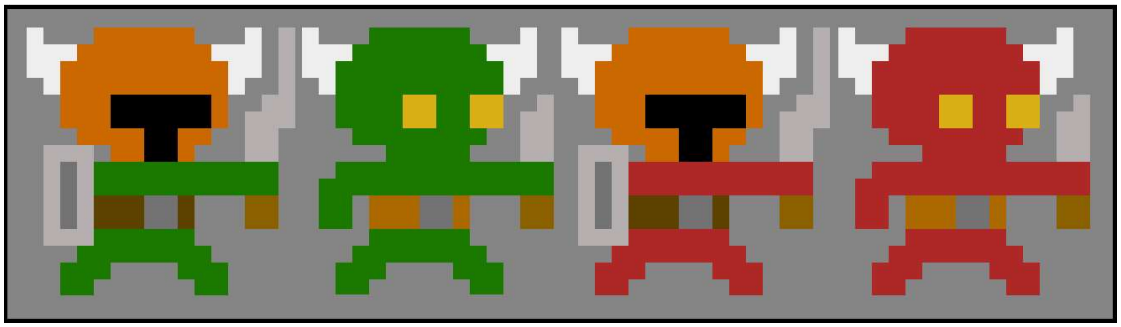

Figure 6. Various enemy types.

G3: To implement breakable objects, we needed tiles that could disappear and modify elements of the game world when attacked. Part of this functionality already existed in enemies. Because of this, we made new enemy types (6 and 7) for two kinds of breakable objects: switches and pots. Additionally, we added a new property to the ROOM class: switches.

To allow for the game world to be modified, we created a new class, "ROOM", to manage rooms. This class was given the following properties:

- "enemies" - the number of enemies within a room
- "switches" - the number of switches within a room

A two-dimensional array was created to store objects of class ROOM. This allowed a room's coordinates within the ROOM array to be equal to its first two coordinates within the world map array.

When a switch's health reaches zero, it decrements the switches property of the room it occupies, then calls a function that checks the number of switches in that room. If the number of switches is zero, certain actions are taken depending on the room (opening a new path, for example). Then, as with all enemies, the switch is removed from the world map and replaced with its cTile.

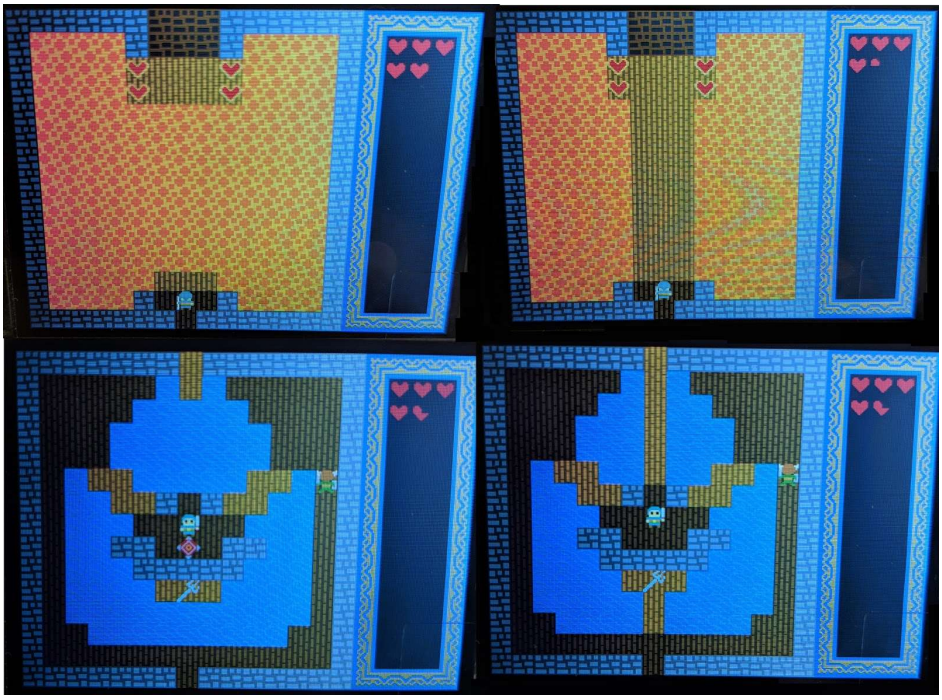

Figure 7. Examples of actions due to a switch being attacked.

When a pot's health reaches zero, it is removed from the world map, its cTile is placed in its place, and there is a chance of a heart being placed over that.

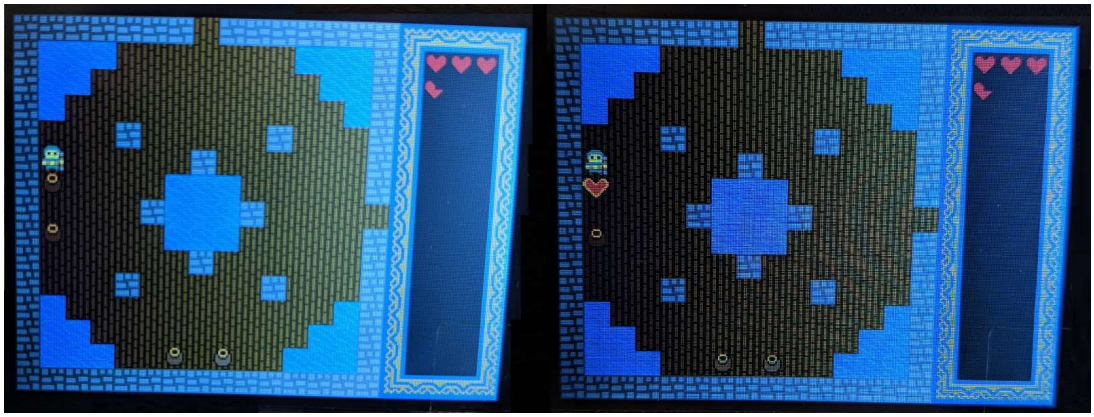

Figure 8. A pot being replaced with a heart.

Graphics/Display:

R6: For the game we wanted high quality art with consistent style. Initially, we used paint to create sprites pixel by pixel.

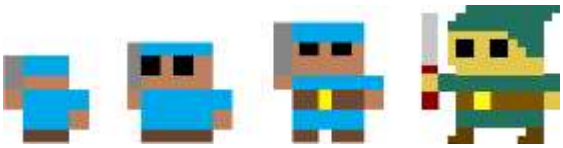

Figure 9. Progression of character designs.

Later, we researched different tools to draw pixel art. We ended up using Piskel, an online sprite editor, for most our art assets. During our research we looked at Creative Commons art, and found some sprites similar to what we wanted. We adapted a couple of these for use in the game, and based the style of our other, original sprites on them.

We kept sprites to only a few colours because of the difficulty in drawing the sprites on the LCD (C1). The LCD library does have commands for bitmaps, but they are only monochrome, and so we had to use the library's existing commands for drawing lines, squares, and pixels. Since these commands are limited, getting art assets into the game was labour intensive.

For the purpose of the game, the display is divided into an imaginary grid of 16×16 pixel blocks. To draw something, the sprite has to be made up of monochrome lines, shapes and pixels. Each drawing command has to be given specific x and y

coordinates in terms of the whole 320×240 grid. Each sprite is drawn on one of the 16×16 pixel blocks, with the origin at the top left corner of the block.

The sprite's position within a room is selected by multiplying the x- and y-coordinates (the last two of the sprite's world map coordinates) by 16, and then each command's x and y-coordinate within the sprite is selected by adding them from the top left corner of the 16×16 block. An example is shown in figure 10, below.

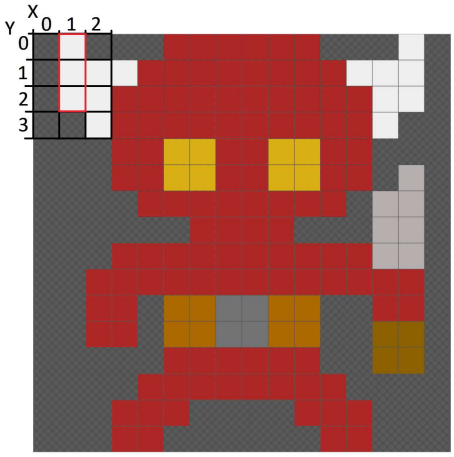

Figure 11. Result of LCD command: tft.drawFastVLine(x*spacing+1, y*spacing, 3, 0xFFFF);

Implementing R4, R7, R8, and all three goals increased the amount of work necessary to meet this requirement, as they all required the addition of art assets to the game.

C1: C1 made it difficult to input art assets into the game. Though it resulted in a significant amount of work, we were able to work within this constraint by drawing sprites into the game using the LCD library's built-in functions.

C3: The resolution of the screen is a constraint on the game. The screen being 320×240 pixels had a significant impact on how we designed the game. We wanted the playable space of the game to be square, so the maximum playable space is 240×240 pixels. We chose the game tiles to be 16×16 pixels because it gives us a sprite size that is versatile and easy to see. Also, it allows the game to have a 15×15 grid, providing a sizable playable space for the player. The sprites that were smaller than 16×16 were hard

to see on the screen, and their design was limited by their pixel count. Choosing a square playable space also let us improve the game's user interface by displaying player health and inventory.

Summary:

Over the course of this project, we learned a variety of programming and teamwork skills.

We had to learn how to use the LCD to display our game. The LCD screen came with its own library that had a variety of commands, which made it relatively easy to use.

A large part of our project was learning how to use classes and objects. We made the hero, enemies, switches, and pots objects of the CHARACTER class, and made rooms objects of the ROOM class. We learned how to give each object a variety of properties which made it easy to keep track of their health, position, status, etc.

We wanted to be able to easily call all of the enemies to have them take their actions, but we weren't able to do this easily with individual objects. To address this, we had to learn how to make an object array. Using an object array, we were able to call all of the enemies in a loop. The enemies would only take their turn if they were currently in the same room as the hero, which we could easily check by having the world map coordinates be properties of the objects.

To allow the player character to easily transition from room to room, and we figured that the easiest way to do this was to make the map an array. To do this and allow positions within rooms to be easily accessible, we needed to learn how to manage four-dimensional arrays. Accessing the array was easy and intuitive. We found the hardest aspect of using a four-dimensional array was initializing it with every tile of every room. Doing this is counter-intuitive, as the way it is input makes the rooms appear rotated and reflected in our code.

To have enemies follow the player character around objects, we had to create and implement a basic AI system. By storing each enemy's previous position, we were able to have them navigate most obstacles within the game.

Furthermore we had to learn a variety of team oriented skills. The first team skill we had to learn was how to work on code with collectively with other. we learned how to use Github. The biggest problem with multiple people coding one project came from merging different branches of code.

Daniel Andjelic, Liam Kynaston, Torin Stephens
CPSC 1490 – 001
15/04/2018

To improve the game the most important change would be to design and implement a better method for managing graphics. Drawing sprites and moving them based on where their pixels are is very basic and limits how much can be done with the game. It would be convenient to be able to design and implement sprites that were bitmaps with multiple colours. It would also reduce that amount of time it took to make new art assets and add them to the game.

Another way to improve the game would be to fix the controls in place. Currently the controls are placed into a breadboard, but the game would be easier to play if the controls were placed into a 3D printed mold with a single directional pad over top of the limit switches.

One final way to improve the game would be to implement real time gameplay. We could accomplish real time gameplay by making the enemies move on a timer, and each time a second passed they would each make an action.

Attachments:

- Game code (FinalProject_1.0.1)
- Adafruit graphics library (Adafruit-GFX-Library-Master)
- Adafruit TFT LCD library (Adafruit-ILI9340-master)