Functionality

I created a story driven puzzle game for my programming assessment. The game is called “Spirit Gardens”, in which the player moves into their new home to discover that it has been taken over by spirits that have fled the forest. The player must discover why the spirits have abandoned their forest home and solve the mystery so they can move into their new house, without the unexpected visitors. They may do so by exploring the mazes via the arrow keys, listening to the NPCs and collecting items to help them on the way.

Environment Structure and Design

I used the grid class and the game object class from Week 4 Lecture 1 of PDM2 to structure my game, following the examples we had been shown. From this I created a child class for the maze wall, maze exit, items, npcs and player from game object before working on the main sketch. This was so each of the objects in the game would have the ‘get’ methods and I could easily access the coordinates for different comparisons I would need for the object detection.

Each of the environments throughout the game is a combination of instances of maze walls, a maze exit and occasional featured NPCs or collectable items for the player to interact with. I used a combination of 3 different functions, following the example shown in Week 4 Lecture 1, which I then modified to suit the requirements of my game. The first of these functions is the create maze function which creates an array, storing each different instance of MazeWall, and also creates the exit which is kept separate from the array.

The maze walls will then be added to the grid in the addMazeWallsToGrid() function. Due to the nature of object detection being reliant on available squares within the grid, the exit must be kept separate so the player can ‘enter’ the same grid cell the maze exit occupies in order to advance to the next stage.

The final function is to draw each of the maze walls and the maze exit by calling the draw method that is available in both classes. During the early stages of building the game, these methods would draw the same shape in different colours to clearly define the exit for further testing and a clearer structure.

```
/**
 * adds maze 2 to grid
 */
function addMaze2ToGrid() {
    for(const MazeWall of maze2) {
        gridM2.addToGrid(MazeWall);
    }
    // gridM2.addToGrid(mazeExit2);
    gridM2.addToGrid(npc6);
}

/**
 * draws maze2
 */
function drawMaze2() {
    for(const MazeWall of maze2) {
        MazeWall.draw();
    }
    mazeExit2.draw();
    image(mazeImg2,0,0);
    npc6.draw(npcImg2);
    // item1.draw()
    // for(i = 0; i < itemArray.length; i++) {
    //     itemArray[0].draw();
    // }
}
```

Figure 1 Maze 2 'add to grid' and 'draw' functions.

Each environment being created through a combination of these three functions is also specific to a different instance of the grid class. This is so when creating the advanceLevel() function, I could swap between the instance of the grid class being displayed. This was a

simpler alternative to clearing the original grid of each of the objects currently occupying each row and column before then adding a new structure to the emptied grid and drawing this on the canvas. I had attempted to alter the grid class by adding my own method which would use two for loops to count through each index in the 2D array and set each row and column's grid cells to false.

```
//my add ons to grid class
/**
 * this method empties the grid of all occupants so I can reset the contents in the maze for each different area
 */
clearGrid() {
    //attempt 1
    // for(let row = 0; row < 11; row++) {
    //     this.#cells[row] = false;
    // } for(let col = 0; col < 11; col++) {
    //     this.#cells[col] = false;
    // }


    for(let row = 0; row < 11; row++) {
        this.#cells[row][col] = false;
        for(let col = 0; col < 11; col++) {
            this.#cells[row][col] = false;
        }
    }
}
```

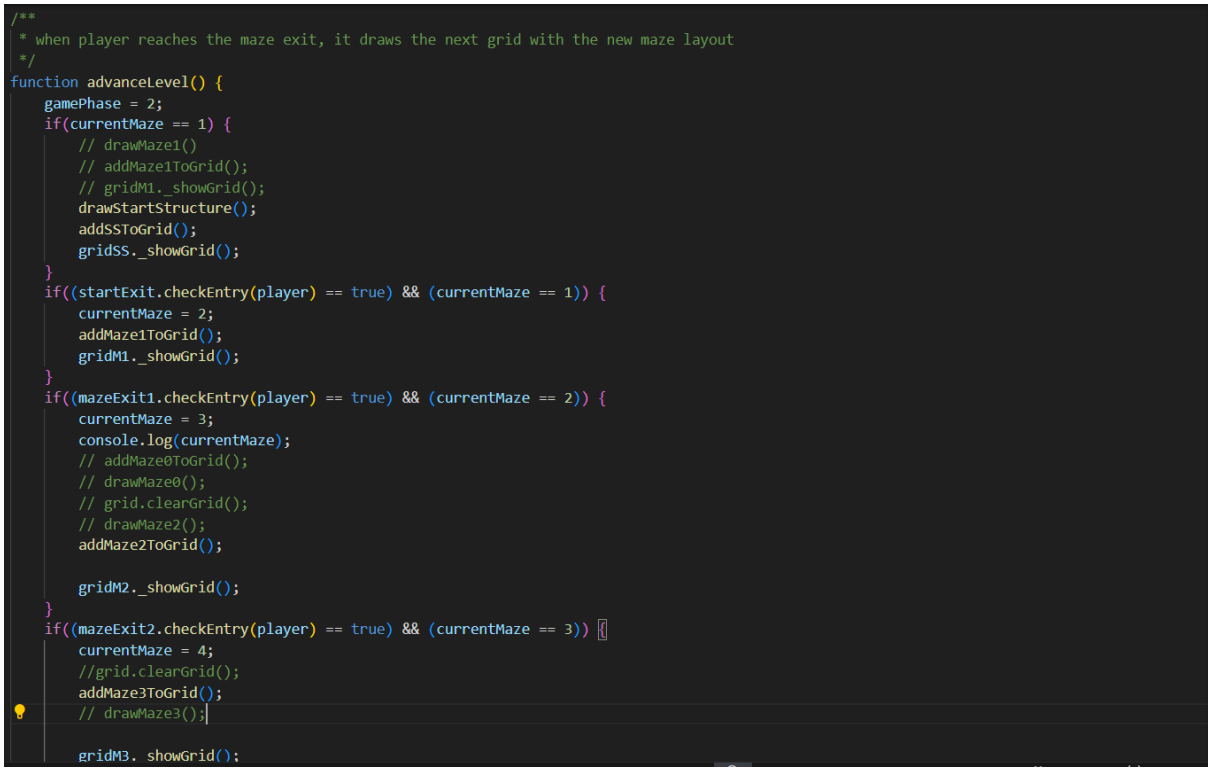
Figure 2 attempt at creating a method for the grid class to clear the grid.

This idea didn't work how I had hoped. Originally, I anticipated the arrays would overwrite each other, but this wasn't the case, so I implemented various grids creating different instances of the class in setup. I decided to treat the introduction environment and the end of game environments the same way I treated the grids for consistency in my code and make it easier to transition between locations. Each grid was still made with 50 cells.

```
grid = new Grid(50);
gridM1 = new Grid(50);
gridM2 = new Grid(50);
gridM3 = new Grid(50);
gridFS = new Grid(50);
gridSS = new Grid(50);
```

Figure 3 all instances of the grid class.

The advanceLevel() function tracks which is the current maze using the currentMaze variable and also tracks if the player has entered the exit using the checkEntry() class method on each of the environment's different MazeExit instances.



```
/**
 * when player reaches the maze exit, it draws the next grid with the new maze layout
 */
function advanceLevel() {
    gamePhase = 2;
    if(currentMaze == 1) {
        // drawMaze1()
        // addMaze1ToGrid();
        // gridM1._showGrid();
        drawStartStructure();
        addSSToGrid();
        gridSS._showGrid();
    }
    if((startExit.checkEntry(player) == true) && (currentMaze == 1)) {
        currentMaze = 2;
        addMaze1ToGrid();
        gridM1._showGrid();
    }
    if((mazeExit1.checkEntry(player) == true) && (currentMaze == 2)) {
        currentMaze = 3;
        console.log(currentMaze);
        // addMaze0ToGrid();
        // drawMaze0();
        // grid.clearGrid();
        // drawMaze2();
        addMaze2ToGrid();

        gridM2._showGrid();
    }
    if((mazeExit2.checkEntry(player) == true) && (currentMaze == 3)) {
        currentMaze = 4;
        //grid.clearGrid();
        addMaze3ToGrid();
        // drawMaze3();

        gridM3._showGrid();
```

Figure 4 a section of the advanceLevel() function.

When creating this function, I initially added the functions that draw the different mazes to the if statements. However, when the function was called, it successfully added the maze values to the grid but couldn't draw the mazes. This was confirmed through the showGrid() testing method within the grid class which revealed the walls were still being added to the grid. To fix this, I separated the code that draws the mazes and kept advance level only occupying the grids. The code to draw the maze was placed in draw within a different set of

conditionals that, like the original, track the current maze using the variable and draw the corresponding maze to that value.

```
if(currentMaze == 2) {
    drawMaze1();
    gridM1._showGrid();
}
if(currentMaze == 3) {
    drawMaze2();
    gridM2._showGrid();

}
if(currentMaze == 4) {
    drawMaze3();
    gridM3._showGrid();
}
if(currentMaze == 5) {
    drawFSS();
    gridFS._showGrid();
}
```

Figure 5 conditional created to draw the correct maze for each grid being displayed on the canvas.

The checkEntry() method returns an object, in this instance it returns the player, and then checks if that objects x and y coordinates are less than or equal to the coordinates of the exit. It then also checks if the player's x coordinate is less than or equal to the x coordinate of the exit plus its width, and then if the player's y coordinate is less than or equal to the exit's y coordinate plus its height. This is necessary because the player is smaller than the maze exit so the game needs to check the full space of the cell the exit occupies for the player's interaction to avoid the interaction only being counted when the player reaches the specific location the exit is drawn from. This was more successful than my first plan which was to compare vector values between the player and the maze exit objects.

```
checkEntry(player) {

    if((player.getX() >= this.getX()) && (player.getY() >= this.getY()) && (player.getX() <= this.getX() + this.getHeight()) && (player.getY() <= this.getY() + this.getHeight())){
        return true;
    }
    // distance = this.position.sub(playerVector);
    //     if(distance == playerVector ) {
    //         console.log(distance.mag());
    //         return true;
    //     }
    //     return false
    //     if((player.getX() + player.getWidth()) && (player.getY() + player.getHeight()) > (this.getX() + this.getWidth()) && (this.getY + this.getHeight())) {
    //         return false;
    //     }
    // // }
    // console.log("Player x: " + player.getX())
    // console.log("Player y: " + player.getY())
    // console.log("exit x: " + this.getX())
    // console.log("exit y: " + this.getY())
}


}
```

Figure 6 current checkEntry() method in the MazeExit class.

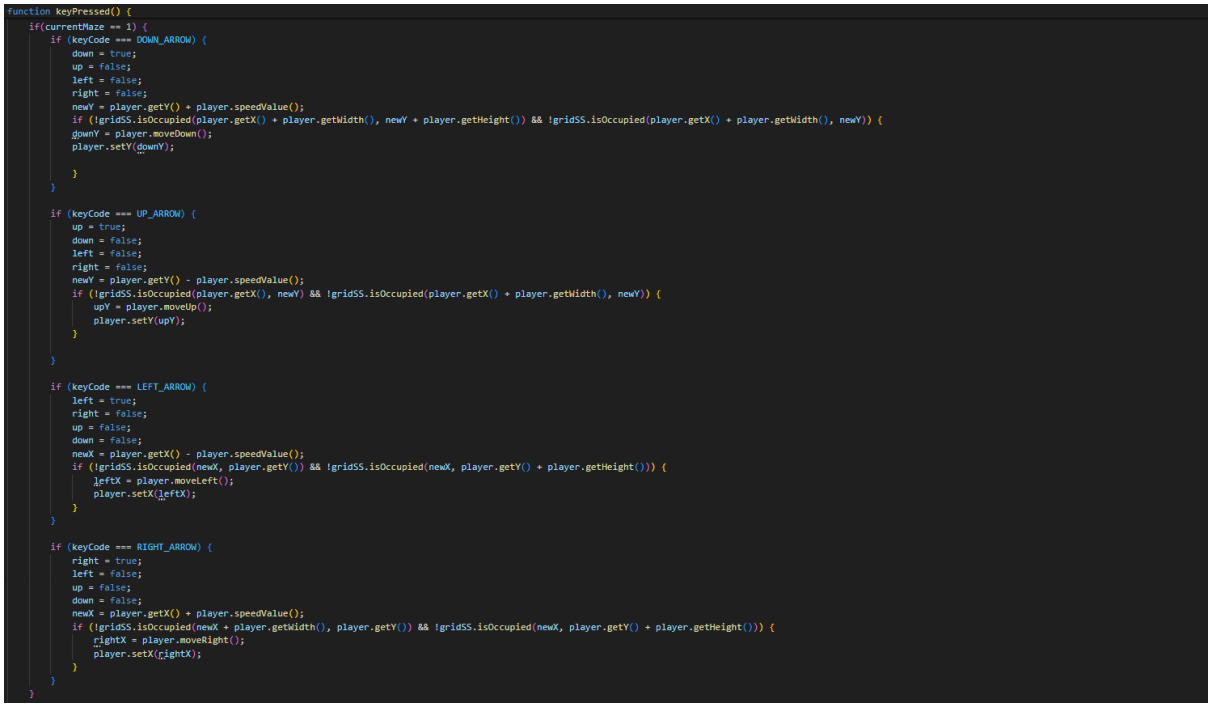
Movement controls and object detection

For the player to move around the different environments, I used the p5.js function keyPressed() and tailored each of the arrow keys to the corresponding direction for the player's movement. The code uses a currentMaze variable which tracks which of the environments in the game are being displayed on the canvas which is important because each of the environments is added to a different grid. This function is how I implemented object detection to stop the player from moving through the maze walls and the NPCs. It detects whether the grid cell that the player is moving into is occupied, and if it isn't occupied, the player can move, following the structure shown in the lectures.

```
if (keyCode === DOWN_ARROW) {
    newY = player.getY() + player.speedValue();
    if (!gridSS.isOccupied(player.getX() + player.getWidth(), newY + player.getHeight()) && !gridSS.isOccupied(player.getX() + player.getWidth(), newY)) {
    downY = player.moveDown();
    player.setY(downY);

    }
}

if (keyCode === UP_ARROW) {
    newY = player.getY() - player.speedValue();
    if (!gridSS.isOccupied(player.getX(), newY) && !gridSS.isOccupied(player.getX() + player.getWidth(), newY)) {
        upY = player.moveUp();
        player.setY(upY);
    }

}

if (keyCode === LEFT_ARROW) {
    newX = player.getX() - player.speedValue();
    if (!gridSS.isOccupied(newX, player.getY()) && !gridSS.isOccupied(newX, player.getY() + player.getHeight())) {
        leftX = player.moveLeft();
        player.setX(leftX);
    }
}

if (keyCode === RIGHT_ARROW) {
    newX = player.getX() + player.speedValue();
    if (!gridSS.isOccupied(newX + player.getWidth(), player.getY()) && !gridSS.isOccupied(newX, player.getY() + player.getHeight())) {
        rightX = player.moveRight();
        player.setX(rightX);
    }
}
```

Figure 7 first working attempt at collision detection using the keyPressed() function.

However, the original code for the player's movement had to be repeated to detect the available cells in each of the grids being drawn. I placed the code inside each conditional that checked the currentMaze variable's value and then would repeat the detection for that maze. Having the detection calculated made it easy to transfer the syntax to apply to various different grids needed for the game.



```
function keyPressed() {
    if(currentMaze == 1) {
        if (keyCode === DOWN_ARROW) {
            down = true;
            up = false;
            left = false;
            right = false;
            newY = player.getY() + player.speedValue();
            if (!gridSS.isOccupied(player.getX() + player.getWidth(), newY + player.getHeight()) && !gridSS.isOccupied(player.getX() + player.getWidth(), newY)) {
            downY = player.moveDown();
            player.setY(downY);

            }
        }

        if (keyCode === UP_ARROW) {
            up = true;
            down = false;
            left = false;
            right = false;
            newY = player.getY() - player.speedValue();
            if (!gridSS.isOccupied(player.getX(), newY) && !gridSS.isOccupied(player.getX() + player.getWidth(), newY)) {
                upY = player.moveUp();
                player.setY(upY);
            }

        }

        if (keyCode === LEFT_ARROW) {
            left = true;
            right = false;
            up = false;
            down = false;
            newX = player.getX() - player.speedValue();
            if (!gridSS.isOccupied(newX, player.getY()) && !gridSS.isOccupied(newX, player.getY() + player.getHeight())) {
                leftX = player.moveLeft();
                player.setX(leftX);
            }
        }

        if (keyCode === RIGHT_ARROW) {
            right = true;
            left = false;
            up = false;
            down = false;
            newX = player.getX() + player.speedValue();
            if (!gridSS.isOccupied(newX + player.getWidth(), player.getY()) && !gridSS.isOccupied(newX, player.getY() + player.getHeight())) {
                rightX = player.moveRight();
                player.setX(rightX);
            }
        }
    }
```

Figure 8 modified keyPressed() function for my desired output.

I also decided to use keyPressed() for changing which version of the player is being displayed. I used variables for each direction and them to true if the key that represents that direction had been pressed. These variables were them used in a playerDirection() function to create a conditional statement that tracked which variables were true so it would then draw the correct image.

```
function playerDirection() {

    player.drawImg(pDown);
    if(left === true) {
        player.drawImg(pLeft);
    }
    if(right === true) {
        player.drawImg(pRight);
    }
    if(up === true) {
        player.drawImg(pUP);
    }
    if(down === true) {
        player.drawImg(pDown);
    }
}
```

Figure 9 playerDirection() function.

NPCs and Collectable Items

I wanted the NPCs to speak when they saw the player at the start of the game. This would kickstart the narrative and give directions through the dialogue. I chose to combine narrative progression within game instructions through NPC interaction because I felt it best complimented the overall design of the game, maintaining a consistent narrative progression throughout. I also felt that this feature created different personalities for the different NPCs in the game through their different responses to leaving the forest. This created a stronger impression of the ‘magical forest’ the ‘forest spirits’ left behind.

The code for interaction with NPCs at the start of the game is stored in a function I created called npcInteraction(). This function uses conditionals to first check the current maze the player is in and then checks if the player’s x and y are facing a chosen side of the npc. For the beginning of the game, it checks the player is either on the left or right of the npcs so as they run up the pathway, they receive the directions to start the game.

```
/**
 * triggers interaction with NPCs by detecting if the player has interacted with the designated location the NPC is in
 */
function npcInteration() {
    if( currentMaze == 1) {
        if((player.getX() && player.getY()) == ((npc.getX() - 50) && npc.getY())) {
            npc.dialogue1();
        }
        if((player.getX() && player.getY()) == ((npc2.getX() + 50) && npc2.getY())) {
            npc.dialogue2();
        }
        if((player.getX() && player.getY()) == (npc5.getX() && npc5.getY())) {
            npc.introHintDialogue();
        }
    }
    if(currentMaze == 2 ) {
        if((player.getX() && player.getY()) >= ((npc4.getX() + 100) && npc4.getY())) {
            npc.dialogue3();
        }
    }
    if(currentMaze == 3) {
        if((player.getX() < npc6.getX() - npc6.getWidth()) && (player.getY() < npc6.getY() + npc6.getHeight())) {
            npc.dialogue7();
        }
    }
    if(currentMaze == 4) {
        if((player.getX() <= npc7.getX()) && (player.getY() <= npc7.getY() - npc7.getHeight() || player.getY() >= npc7.getY() + npc7.getHeight())) {
            npc.dialogue8();
        }
    }
    if(currentMaze == 5) {
        if((player.getX() && player.getY()) >= (npc5.getX() && npc5.getY() + npc5.getHeight())) || (player.getX() && player.getY()) >= ((npc5.getX() + npc5.getWidth()) && (npc5.getY() + npc5.getHeight())) ) {
            npc.dialogue5();
        }
        if((player.getX() && player.getY()) <= (npc5.getX() && npc5.getY())) {
            npc.dialogue4();
        }
        // if((player.getX() && player.getY()) == ((npc4.getX()) && (npc4.getY() + 100))) {
        //     npc.dialogue6();
        // }
    }
}
```

Figure 10 npcInteraction() function.

For the separate lines of dialogue, I considered different ways at storing the lines of text and then displaying them for the player when appropriate. I decided to create separate methods within the NPC class that store each line of dialogue. This keeps all the responses separate and makes it easier to call the class method within each of the conditionals by keeping the responses specific to the NPCs.

```
class NPC extends GameObject {
    /**
     * draws text box and fills it with dialogue
     */
    textBox() {
        stroke(0);
        fill(255);
        rect(0,450,300,100);
        // text(this.#dialogue[i], 0, 500);
    }

    /**
     * draws dialogue
     */
    dialogue1() {
        text("please help us! there's a monster!", 270,300);
    }

    dialogue2() {
        text("you must go through the forest maze ahead!",200,250);
    }

    dialogue3() {
        text("*shivers*",0,350);
    }

    dialogue4() {
        text("please don't be scared! I'm not a monster", 250,300);
    }

    dialogue5() {
        text("if you splash me with that bucket, it'll prove i'm not a monster", 250,300)
    }

    dialogue6() {
        text("ah, you saved me from the mud! Now i can go home and see my friends!", 250,300);
    }
    introHintDialogue() {
        text("look out for items on the way to help you",150,150);
    }

}
```

Figure 11 a section of the NPC class' dialogue methods.

For the player to collect items within the different locations in the game, they must ‘walk over’ them. I created a function to check this. The code tracks the x and y coordinates of the player and the x and y coordinates of the object in the conditional statement and compares these values. If they are the same, the code prints text that tells the player the item has been

collected. With their new item, the player can move on to the next maze location. Each of the items is a tear that has been left behind by the spirits as they fled the forest. The NPC at the end location will then set a quest for the player that requires them to use the items gathered in the maze.

```
function itemCollected() {
    if(item1.collected(player) == true) {
        // drawMaze2NoObj();
        text("you found an item!", item1.getX(), item1.getY());
        // /** change current maze to second display of maze 2 */
        // currentMaze == 32

    }
}
```

Figure 12 itemCollected() function.

This function uses the collected() method from the item class. It contains the information for drawing the different items and also contains the collected() method. This method works in a very similar way to the maze exit's checkEntry() method, as it also compares the players coordinates with the objects own coordinates to return true. Both methods also take an object as an argument, referring to the player in both situations.

```
class Item extends GameObject {
    #img;

    constructor(x,y,w,h) {
        super(x,y,w,h);
        this.#img;
    }

    draw() {
        fill(255,191,0);
        rect(super.getX(), super.getY(), super.getWidth(), super.getHeight());
    }

    collected(player) {
        if((player.getX() >= this.getX()) && (player.getY() >= this.getY()) && (player.getX() <= this.getX() + this.getHeight()) && (player.getY() <= this.getY() + this.getHeight
            return true;
        }
    }
}
```

Figure 13 collected() method in the item class.

Additional Features and Aesthetics

Using the grid based detection method worked best for my idea because when the collision detection code was completed, I drew each stage in the game and placed the images over the original display on the canvas to give the illusion the player is colliding with the trees in each maze.

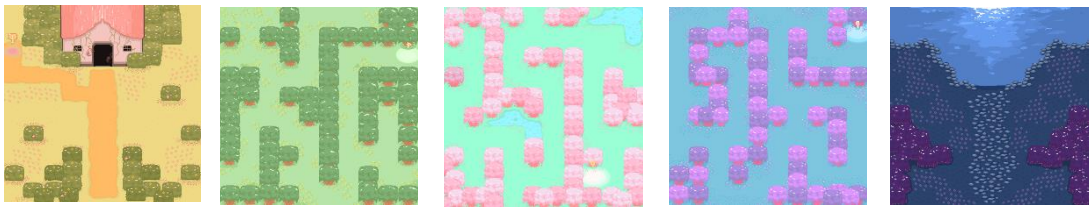


Figure 14 Maze images.

Using separate image files for the player worked well for changing the player's direction visually. I drew each of the different images on procreate by creating my own pixel art brush and changing the canvas size to match the size of the grid cell each character in the game would occupy. I also used this approach for the different settings.

Figure 15 player images.

Figure 16 NPC images.

The final NPC is drawn from an 100x100 pixel grid because the sludge design takes up a larger portion of the screen. I wanted the size to reflect the NPCs responses to running away from a ‘monster’.

Figure 17 final NPC ('in the mud') image.

The spirit being revealed after the player uses the magic tears to save them will be the original size, but I wanted to emphasize the joy the spirit feels after being saved to I drew stars around the spirit.

Figure 18 final NPC (celebration) image.

I added a sound track which is loaded using the preload() function and then looped when the player presses the ‘START GAME’ button.

```
/**
 * removes button and starts running game once pressed
 */
function buttonClicked() {
    gameRun = true;
    buttonSTART.hide();
    soundTrack.loop();
    gamePhase = 2;
    currentMaze = 1;
}
```

Figure 19 buttonClicked() function to start the game.

Reflection

There are still existing bugs in Spirit Gardens. There is a bug in the transition to the first maze location from the opening scene where the player spawns on the maze walls and not in the open which I would like to fix when developing the game further. Some of the dialogue in the mazes from the NPCs will also be repeated, but I think for some of the dialogue this surprisingly worked well, so I kept it. I think Spirit Gardens overall successfully incorporates a simple narrative with enjoyable game play and a new world for the player to explore. However, I do think the gameplay feels a bit repetitive after the first two mazes. I would like to further develop the game with cutscenes for the introduction to set the scene for the player and cutscenes for each of the NPCs to bring out their personalities and curate a richer storyline. The NPCs could also have mini games for the player to do while in the maze to break up the maze exploring and make it more engaging. This could be further improved with sound effects played during each interaction and item collection.

I would also like to improve the solutions to some of the issues I had in the programming. For the item collection, the maze image is redrawn over the item, but I would like to add more efficient solutions like a way to remove the item. The item collection could also be improved by having an onscreen display of the items that have been collected.

Figure 20 concept for item collection display.

Description	Source	Licence
Relaxing lo-fi music	Music by Calvin Clavier from <a href="https://pixabay.com//?utm_source=link-attribution&utm_medium=referral	Free to use under Pixabay Content License https://pixabay.com/service/license-summary/

	&utm_campaign=music&utm_content=207243">Pixabay	

Word Count: 2254