Spring 2017

Creating a Game with Procedural Generation

Kirsten Baker
The University of Akron, knb58@zips.uakron.edu

The Game:

The created game is a procedurally generated maze. The first version of the game was completed in Java while the current version using Unity. While the Unity version mostly uses the same code as the Java version, Unity only allows for C# or JavaScript to be used. Therefore, the scripts were translated into C# and use Unity code.

Before the maze is generated, the player chooses a size for the board. Initially, all the walls are constructed on the playing field. Then, the algorithm randomly chooses a starting point on one side (left side for the Java project, bottom for the Unity project) and creates a maze that ends on the opposite side. After a starting location is selected, a random direction is chosen and the algorithm looks to see if the move is allowed and then if it has previously been to the square. If the direction chosen points to an unmarked location, it moves to it, marks it, and destroys the wall it passed through. If the chosen spot has already been discovered, the generator will choose another direction. If the algorithm traps itself, it will randomly select a previously visited location and attempt to continue the maze. The maze is considered completed when the generator reaches the opposite side of the board.

This algorithm allows the maze to be completed without cycles and guarantees uniqueness for each maze. Instead of filling in the completed maze with dead ends like traditional mazes, I decided to allow empty areas that created various abstract patterns. Figure 1.1 shows the maze generated using Java and Figure 1.2 shows the top-down view of the maze using Unity. In both figures, the player is represented by a blue square.

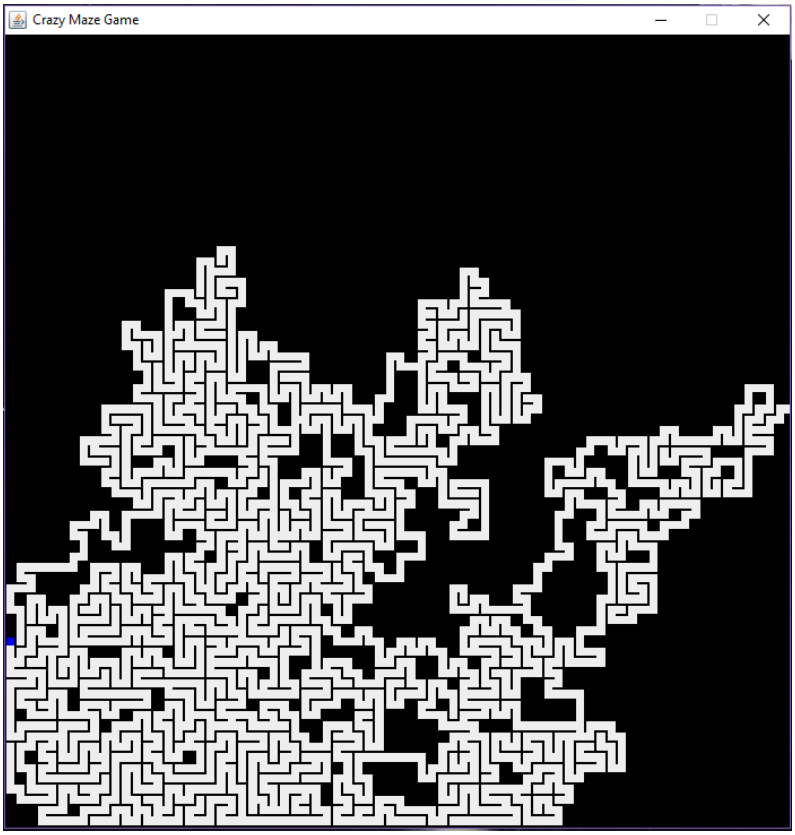


FIGURE 1.1

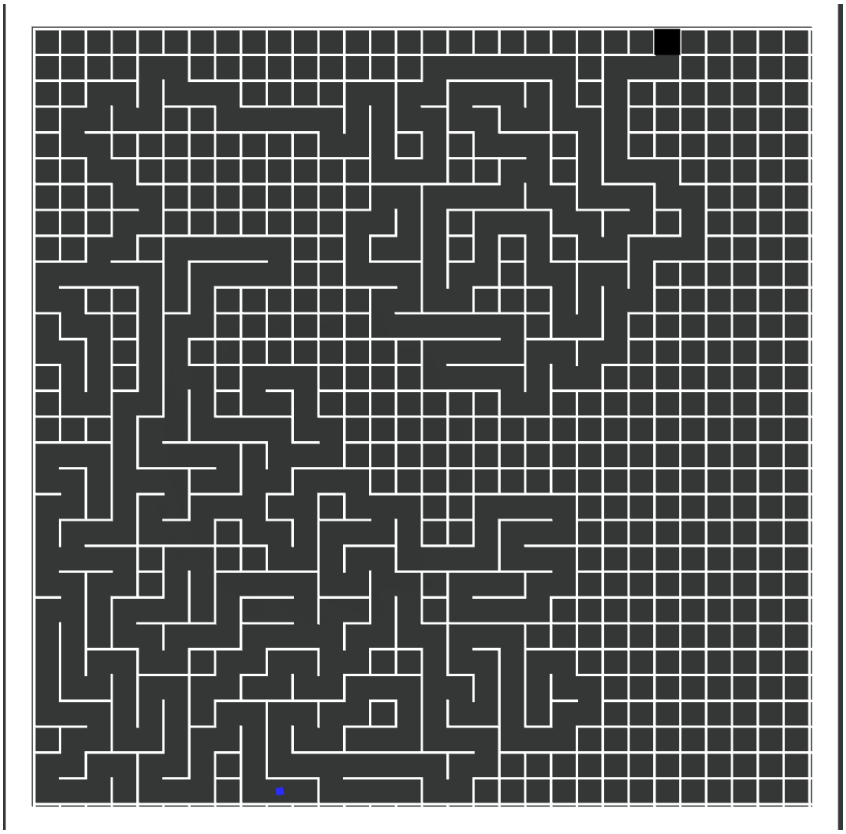


FIGURE 1.2

Before Unity:

Before deciding to shift the project into the game engine, Unity, I implemented an escape feature in the game that would allow the player to return to the main menu. This would occur when a player pressed the escape key. As shown in Figure 2, a screen would pop up that the player quit and show the current time and number of moves. The buttons at the bottom would allow the player to either return to the main menu or quit the game.

FIGURE 2

Furthering the improvements on the game, I refined the maze creation by separating pieces of code that were jammed together in a large function into smaller functions. As shown in Figure 3.1, the code that allowed the computer to change the direction of the maze was moved into a function called mazeDirection. To call the function, you include the character that told what direction the maze was to move in next. Because the code confirmed that it can move in the specified direction before the call, the function returns nothing. Figure 3.1 also displays the section of code that changed the position of the generator after the maze creator was trapped. This portion was moved into a function labeled repositionMazeCreator.

Figure 3.2 shows the function used to create the blackout for the extreme modes. Because the lines to create the blackout for the first player were the same for player two, one set was transferred into the createBlackout function. To make sure that the function can draw the area of light around the correct player, it takes in the player's position.

And finally, Figure 3.3 displays the code that moved the player(s) around the board. The class was renamed as PlayerMover which handled which keys moved the player and the direction. The code that changed the location of the player was transferred into the movePlayer function. Making these changes greatly increased the readability and flexibility of the code.

Unity:

Upon taking the advice of a faculty member, I decided to move the project into a game engine called Unity. This would allow my game to become three-dimensional and look more professional. However, I soon realized I couldn't fully reuse my previous code as Unity has special functions and only allows the use of C# or JavaScript. Because Java and C# are similar, I decided to translate my current code into it. Figure 4 shows what the Unity engine looks like to a developer.

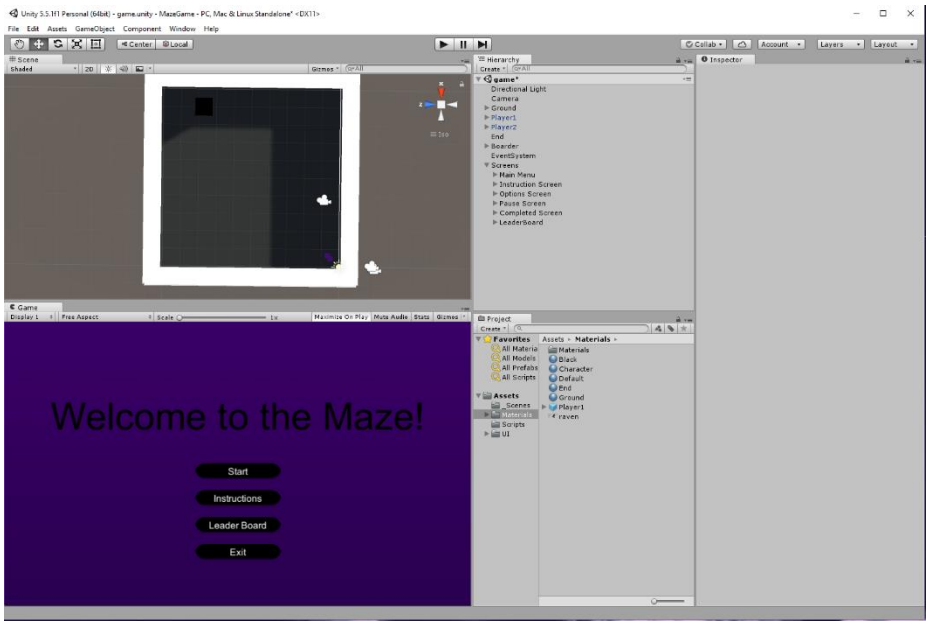


FIGURE 4

Before:

```
public void graph(int size) {
    ...
    if (d == 'n') {
        if (yLoc > 0 && !corners[xLoc][yLoc-1]) {
            hWalls[xLoc+1][yLoc] = false;
            yLoc--;
            corners[xLoc][yLoc] = true;
        }
    }
    else if (d == 's') {
        if (yLoc < size-1 && !corners[xLoc][yLoc+1]) {
            yLoc++;
            hWalls[xLoc+1][yLoc] = false;
            corners[xLoc][yLoc]=true;
        }
    }
    else if (d == 'e') {
        if (xLoc < size-1 && !corners[xLoc+1][yLoc]) {
            xLoc++;
            vWalls[xLoc][yLoc + 1] = false;
            corners[xLoc][yLoc]=true;
        }
    }
    else if (d == 'w') {
        if (xLoc > 0 && !corners[xLoc-1][yLoc]) {
            vWalls[xLoc][yLoc + 1] = false;
            xLoc--;
            corners[xLoc][yLoc]=true;
        }
    }
    if (trapped(xLoc, yLoc, size)) {
        int possX, possY;
        boolean exist=false;
        while (!exist){
            possY = rand(size);
            possX = rand(size);
            if (corners[possX][possY]) {
                xLoc = possX;
                yLoc = possY;
                exist=true;
            }
        }
    }
    ...
}
```

After:

```
public void graph(int size) {
    ...
        mazeDirection(d);

        //if trapped, move to a another location to
continue
        if (trapped(xLoc, yLoc, size)) {
            repositionMazeCreator();
        }
    ...
}

//move maze creator in given direction
public void mazeDirection(char d){
    if (d == 'n') {
        if (yLoc > 0 && !corners[xLoc][yLoc-1]) {
            hWalls[xLoc+1][yLoc] = false;
            yLoc--;
            corners[xLoc][yLoc] = true;
        }
    }
    else if (d == 's') {
        if (yLoc < size-1 && !corners[xLoc][yLoc+1]) {
            yLoc++;
            hWalls[xLoc+1][yLoc] = false;
            corners[xLoc][yLoc]=true;
        }
    }
    else if (d == 'e') {
        if (xLoc < size-1 && !corners[xLoc+1][yLoc]) {
            xLoc++;
            vWalls[xLoc][yLoc + 1] = false;
            corners[xLoc][yLoc]=true;
        }
    }
    else if (d == 'w') {
        if (xLoc > 0 && !corners[xLoc-1][yLoc]) {
            vWalls[xLoc][yLoc + 1] = false;
            xLoc--;
            corners[xLoc][yLoc]=true;
        }
    }
}

//if maze creator is trapped, move to another location
to coninue
public void repositionMazeCreator(){
    int possX, possY;
    boolean exist=false;
    while (!exist){
        possY = rand(size);
        possX = rand(size);
        if (corners[possX][possY]) {
            xLoc = possX;
            yLoc = possY;
            exist=true;
        }
    }
}
```

FIGURE 3.1

Before:

```
public void paintComponent(Graphics g) {
    ...

    if (modeSel > 0) {
        Area a = new Area(new Rectangle2D.Double(0, 0, xLen, xLen));
        if (modeSel == 1) {
            a.subtract(new Area(new Ellipse2D.Double(((xStart - 5.5) * length) - 4 * length,
                    ((yStart - 5.5) * length) - 4 * length, length * 20, length * 20)));
        }
        else if (modeSel == 2) {
            a.subtract(new Area(new Ellipse2D.Double(((xStart - 2.5) * length) - 2 * length,
                    ((yStart - 2.5) * length) - 2 * length, length * 10, length * 10)));
        }
        else {
            a.subtract(new Area(new Ellipse2D.Double(((xStart - 1) * length) - length,
                    ((yStart - 1) * length) - length, length * 5, length * 5)));
        }
        if (play2) {
            if (modeSel == 1)
                a.subtract(new Area(new Ellipse2D.Double(((x2Start - 5.5) * length) - 4 * length,
                        ((y2Start - 5.5) * length) - 4 * length, length * 20, length * 20)));
            else if (modeSel == 2)
                a.subtract(new Area(new Ellipse2D.Double(((x2Start - 2.5) * length) - 2 * length,
                        ((y2Start - 2.5) * length) - 2 * length, length * 10, length * 10)));
            else
                a.subtract(new Area(new Ellipse2D.Double(((x2Start - 1) * length) - length,
                        ((y2Start - 1) * length) - length, length * 5, length * 5)));
        }
        g2.fill(a);
    }

    ...
}
```

After:

```
public void paintComponent(Graphics g) {
    ...

    //create the blackout and flashlight
    if (modeSel > 0) {
        Area a = new Area(new Rectangle2D.Double(0, 0, xLen, xLen));
            createBlackout(g2,a,Player1X,Player1Y);
        if (play2) {
            createBlackout(g2, a, Player2X, Player2Y);
        }
        g2.fill(a);
    }
    ...
}

//creates the flashlight around the player for the extreme modes
public void createBlackout(Graphics2D g2, Area a, int playerX, int playerY){
    switch (modeSel) {
        case 1:
            a.subtract(new Area(new Ellipse2D.Double(((playerX - 5.5) * length) - 4 * length,
                    ((playerY - 5.5) * length) - 4 * length, length * 20, length * 20)));
            break;
        case 2:
            a.subtract(new Area(new Ellipse2D.Double(((playerX - 2.5) * length) - 2 * length,
                    ((playerY - 2.5) * length) - 2 * length, length * 10, length * 10)));
            break;
        case 3:
            a.subtract(new Area(new Ellipse2D.Double(((playerX - 1) * length) - length,
                    ((playerY - 1) * length) - length, length * 5, length * 5)));
            break;
        default:
            break;
    }
}
```

FIGURE 3.2

Before:

```
class KeyPressListener implements KeyListener {
    @Override
    public void keyPressed(KeyEvent event) {
        String m="";
        String m2="";
        if (!play2){
            switch (event.getKeyCode()) {
                case 87: m = "n"; moveCounter++; break;
                case 38: m = "n"; moveCounter++; break;
                case 83: m = "s"; moveCounter++; break;
                case 40: m = "s"; moveCounter++; break;
                case 68: m = "e"; moveCounter++; break;
                case 39: m = "e"; moveCounter++; break;
                case 65: m = "w"; moveCounter++; break;
                case 37: m = "w"; moveCounter++; break;
                default: break;
            }
            switch(m){
                case "n":
                    if(!hWalls[xStart+1][yStart])
                        yStart--;
                    break;
                case "s":
                    if(!hWalls[xStart+1][yStart+1])
                        yStart++;
                    break;
                case "e":
                    if(!vWalls[xStart+1][yStart+1])
                        xStart++;
                    else if(xStart+1 == size) {
                        getRootPane().getParent().setVisible(false);
                        new MazeCompleted(moveCounter, -1, 0, gameTime);
                    }
                    break;
                case "w":
                    if(!vWalls[xStart][yStart+1])
                        xStart--;
                    break;
                default: break;
            }
        }
        else{
            switch (event.getKeyCode()) {
                case 87: m2 = "n"; move2Counter++; break;
                case 38: m = "n"; moveCounter++; break;
                case 83: m2 = "s"; move2Counter++; break;
                case 40: m = "s"; moveCounter++; break;
                case 68: m2 = "e"; move2Counter++; break;
                case 39: m = "e"; moveCounter++; break;
                case 65: m2 = "w"; move2Counter++; break;
                case 37: m = "w"; moveCounter++; break;
                default: break;
            }
            switch(m2){
                case "n":
                    if(!hWalls[x2Start+1][y2Start])
                        y2Start--;
                    break;
                case "s":
                    if(!hWalls[x2Start+1][y2Start+1])
                        y2Start++;
                    break;
                case "e":
                    if(!vWalls[x2Start+1][y2Start+1])
                        x2Start++;
                    else if(x2Start+1 == size) {
                        getRootPane().getParent().setVisible(false);
                        new MazeCompleted(moveCounter,
                                move2Counter, 2, gameTime);
                    }
                    break;
                case "w":
                    if(!vWalls[x2Start][y2Start+1])
                        x2Start--;
                    break;
                default: break;
            }
            switch(m){
                case "n":
                    if(!hWalls[xStart+1][yStart])
                        yStart--;
                    break;
                case "s":
                    if(!hWalls[xStart+1][yStart+1])
                        yStart++;
                    break;
                case "e":
                    if(!vWalls[xStart+1][yStart+1])
                        xStart++;
                    else if(xStart+1 == size) {
                        getRootPane().getParent().setVisible(false);
                        new MazeCompleted(moveCounter,
                                move2Counter, 1, gameTime);
                    }
                    break;
                case "w":
                    if(!vWalls[xStart][yStart+1])
                        xStart--;
                    break;
                default: break;
            }
        }
        repaint();
    }
    public void keyTyped(KeyEvent event) {}
    public void keyReleased(KeyEvent event) {}
}
```

FIGURE 3.3.1

After:

```
class PlayerMover implements KeyListener {
    @Override
    public void keyPressed(KeyEvent event) {
        if(event.getKeyCode() == KeyEvent.VK_ESCAPE){
            new MazeCompleted(moveCounter, -1, -1, gameTime);
            getRootPane().getParent().setVisible(false);
        }
        else{
        String m="";
        String m2="";
        if (!play2){
            switch (event.getKeyCode()) {
                case 87: m = "n"; moveCounter++; break;
                case 38: m = "n"; moveCounter++; break;
                case 83: m = "s"; moveCounter++; break;
                case 40: m = "s"; moveCounter++; break;
                case 68: m = "e"; moveCounter++; break;
                case 39: m = "e"; moveCounter++; break;
                case 65: m = "w"; moveCounter++; break;
                case 37: m = "w"; moveCounter++; break;
                default: break;
            }
                movePlayer(m,"");
        }
        else{
            switch (event.getKeyCode()) {
                case 87: m2 = "n"; move2Counter++; break;
                case 38: m = "n"; moveCounter++; break;
                case 83: m2 = "s"; move2Counter++; break;
                case 40: m = "s"; moveCounter++; break;
                case 68: m2 = "e"; move2Counter++; break;
                case 39: m = "e"; moveCounter++; break;
                case 65: m2 = "w"; move2Counter++; break;
                case 37: m = "w"; moveCounter++; break;
                default: break;
            }
            movePlayer(m, "");
            movePlayer("",m2);
        }
       repaint();
    }
    }

    public void keyTyped(KeyEvent event) {}
    public void keyReleased(KeyEvent event) {}
}
```

```
//move player in the desired direction if allowed
public void movePlayer(String m, String m2){
    if("".equals(m2)){
        switch(m){
            case "n":
                if(!hWalls[Player1X+1][Player1Y])
                    Player1Y--;
                break;
            case "s":
                if(!hWalls[Player1X+1][Player1Y+1])
                    Player1Y++;
                break;
            case "e":
                if(!vWalls[Player1X+1][Player1Y+1])
                    Player1X++;
                else if(Player1X+1 == size) {
                    getRootPane().getParent().setVisible(false);
                    new MazeCompleted(moveCounter, -1, 0, gameTime);
                }
                break;
            case "w":
                if(!vWalls[Player1X][Player1Y+1])
                    Player1X--;
                break;
            default: break;
        }
    }
    if ("".equals(m)){
        switch(m2){
            case "n":
                if(!hWalls[Player2X+1][Player2Y])
                    Player2Y--;
                break;
            case "s":
                if(!hWalls[Player2X+1][Player2Y+1])
                    Player2Y++;
                break;
            case "e":
                if(!vWalls[Player2X+1][Player2Y+1])
                    Player2X++;
                else if(Player2X+1 == size) {
                    getRootPane().getParent().setVisible(false);
                    new MazeCompleted(moveCounter,
                            move2Counter, 2, gameTime);
                }
                break;
            case "w":
                if(!vWalls[Player2X][Player2Y+1])
                    Player2X--;
                break;
            default: break;
        }
    }
}
```

FIGURE 3.3.2

Creating the Walls:

Unlike a 2D board where X and Y are the flat axes, in a 3D Unity project, the X and Z axes lay flat while the Y axis is going up or down. To cause less confusion, I changed the Y labels in the code to Z to suit the new axes. Since the first version of the game was completed in Java, the walls were just lines drawn on the screen. Because the walls now needed to be 3D, I changed the code to work with game objects.

When first creating game objects, such as the outer walls and floor, they are centered on the origin. At the start, I simply moved the walls into the proper place on the sides of the floor to create the game barrier. When first making the playing field, this posed no problem. However, when trying to create the inner walls of the maze, I realized that the walls would include negative positions and I would have to do unnecessary calculations to figure out where they would be. Instead of this, the problem was rectified when I made the ground and outer walls reposition themselves to start at the origin and move in a positive direction. I could then continue using the algorithm I had already created.

Placing the inner walls proved a challenge because they needed to be sized and placed properly so that when walls were removed, the other walls looked seamless. I realized I had already solved the problem, because all I needed were two intersecting walls which were already created as the boarder. I created copies of two of the intersecting outer walls and sized them down to be the inner walls.

The next problem was that the walls were being improperly sized relative to the maze size. To fix this, I used the editor to move and resize the walls to get the calculations that would allow them to flow seamlessly into the others on every level. Solving this problem also helped solve the same issue with the outer walls. When I finally completed the calculations for the inner

walls, I created and placed copies of each wall to fill the board to create the initial state of the maze.

Maze Destroyer:

At the end of the level, the inner walls need to be destroyed so that a new maze can be created. If the walls are not destroyed between levels, as shown in Figures 5.1 and 5.2, the previous walls would interfere with the creation of a new maze. To destroy the walls, while they are being created, the game objects were placed in an array that would delete its contents when the level was completed or the player decided to create a new one.

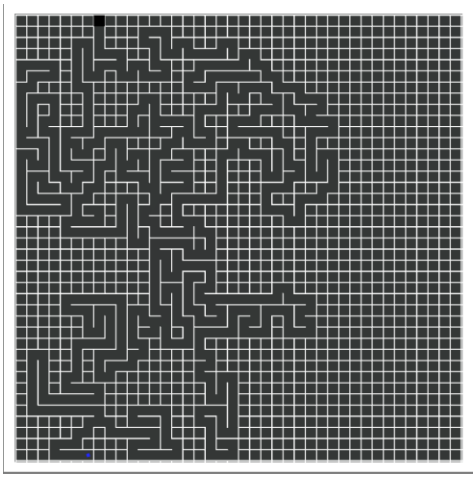


FIGURE 5.1

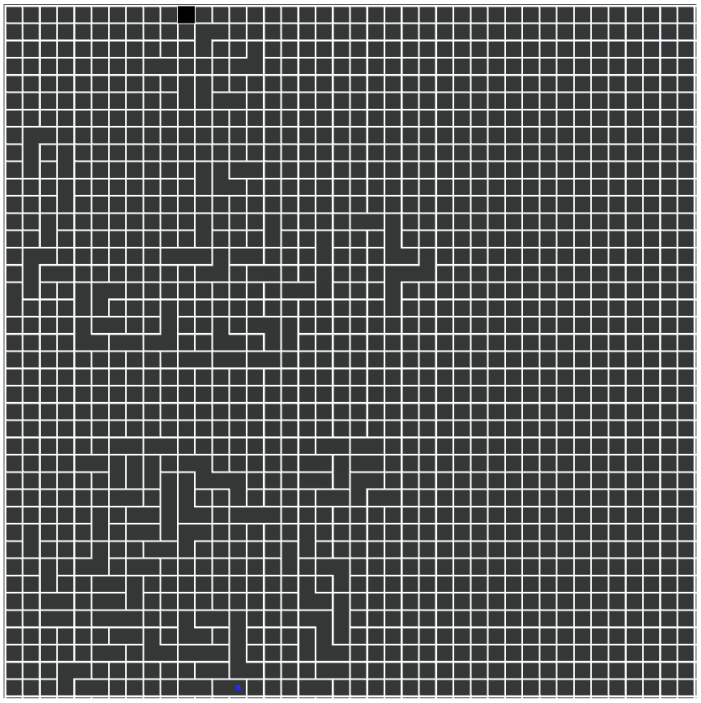


FIGURE 5.2

Finish line:

Once the maze was being created and the player able to move through it, I needed to add an object that would signal the finish line. At first, I simply created a spotlight to mark the end. However, I realized that I could not add in a detector to see when the object encountered it because a spotlight is not a solid object. Instead, as seen in Figure 6, I created a solid black box that took over the entire space. Upon colliding with the end box, the win screen would appear.

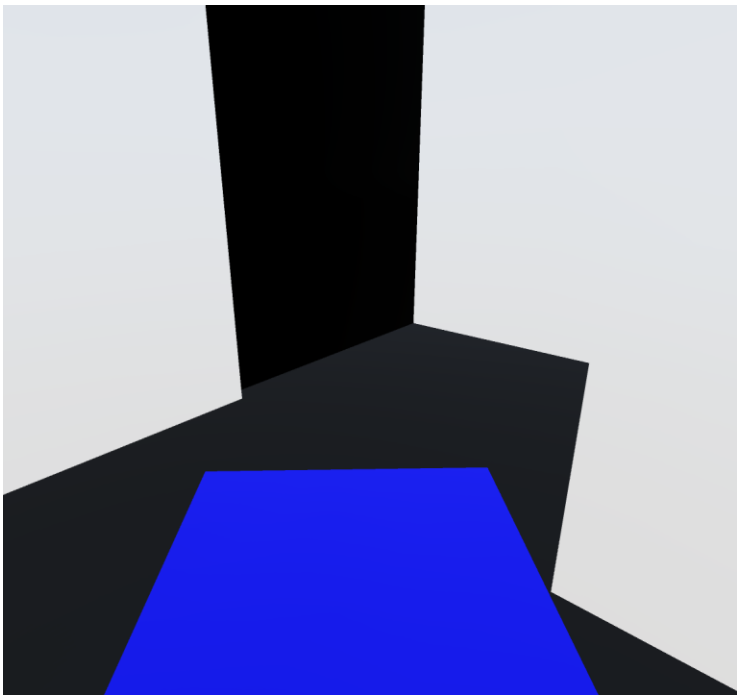


FIGURE 6

Screens:

For almost the entire project, I encountered a problem in that the text on the various menu screens refused to resize in relation to the computer's screen. In the Unity editor, there is a space that shows how the game currently looks. Using this section, I placed and sized the text. However, when playing the game, the size of the text didn't grow in proportion to the screen

dimensions. After some research and testing, I found the setting (shown in Figure 7) that would allow the text to fit perfectly on any resolution and screen.

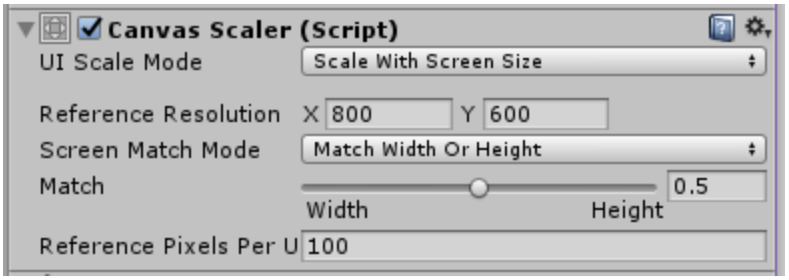


FIGURE 7

Getting the screens to switch between each other was a challenge. In the editor, there is a section to choose what happens when a button is pressed. At first, I had the screens going to different displays when they were not in use. However, when I realized that they may show up on different monitors if players had more than one, I knew I needed to change the implementation. To do this, I created a script, ScreenController, that would would allow screens to be flipped between without sending the old one to a different display. Instead, as shown in Figure 8, the script activates the new screen and deactivates the old one.

```
public class ScreenController : MonoBehaviour {

    [SerializeField]
    private Button b;
    [SerializeField]
    private GameObject current;
    [SerializeField]
    private GameObject target;

    // Use this for initialization
    void Start () {
        b.onClick.AddListener(() => { screen(); });
        if (current.name != "Main Menu") { current.SetActive(false); }
    }

    void screen () {
        target.SetActive(true);
        current.SetActive(false);
    }
}
```

FIGURE 8

There are six non-game screens: Main Menu, Options, Instructions, Leaderboard, Paused, and Completed. They are shown in Figures 9.1 – 9.6 respectively.

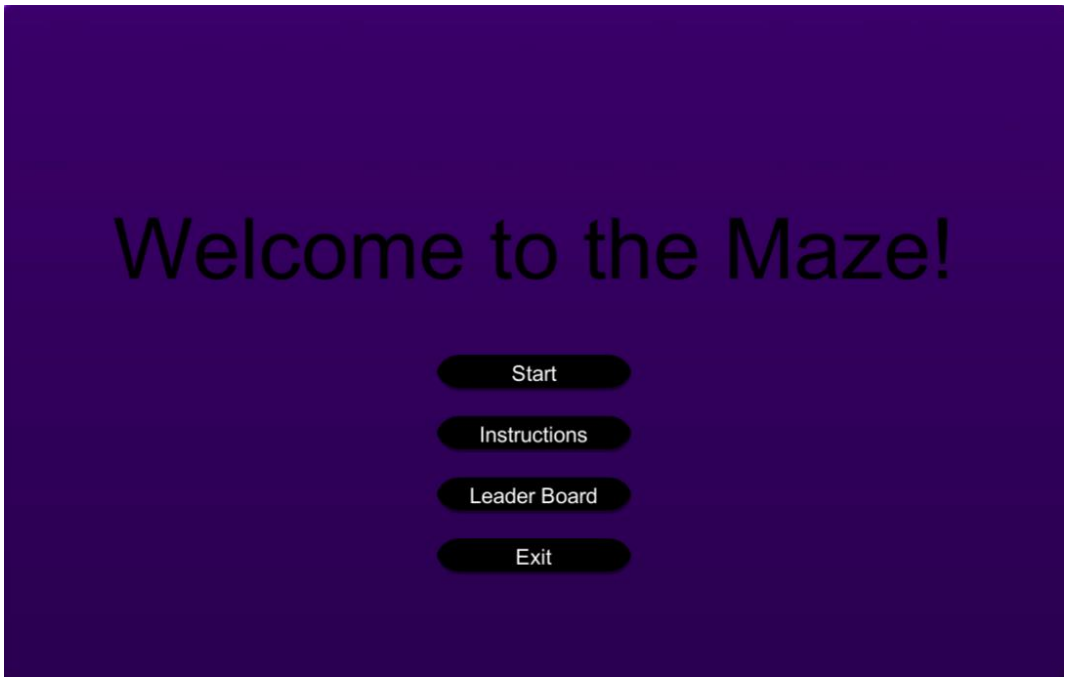


FIGURE 9.1

The Main Menu houses the options to go to the Options menu (Figure 9.2) to start the game, look at the Instructions (Figure 9.3), look at the Leaderboard (Figure 9.4), or quit the game. The Main Menu button on the Paused screen (Figure 9.5) and the Play Again button on the Completed screen (Figure 9.6) both lead back to this screen.

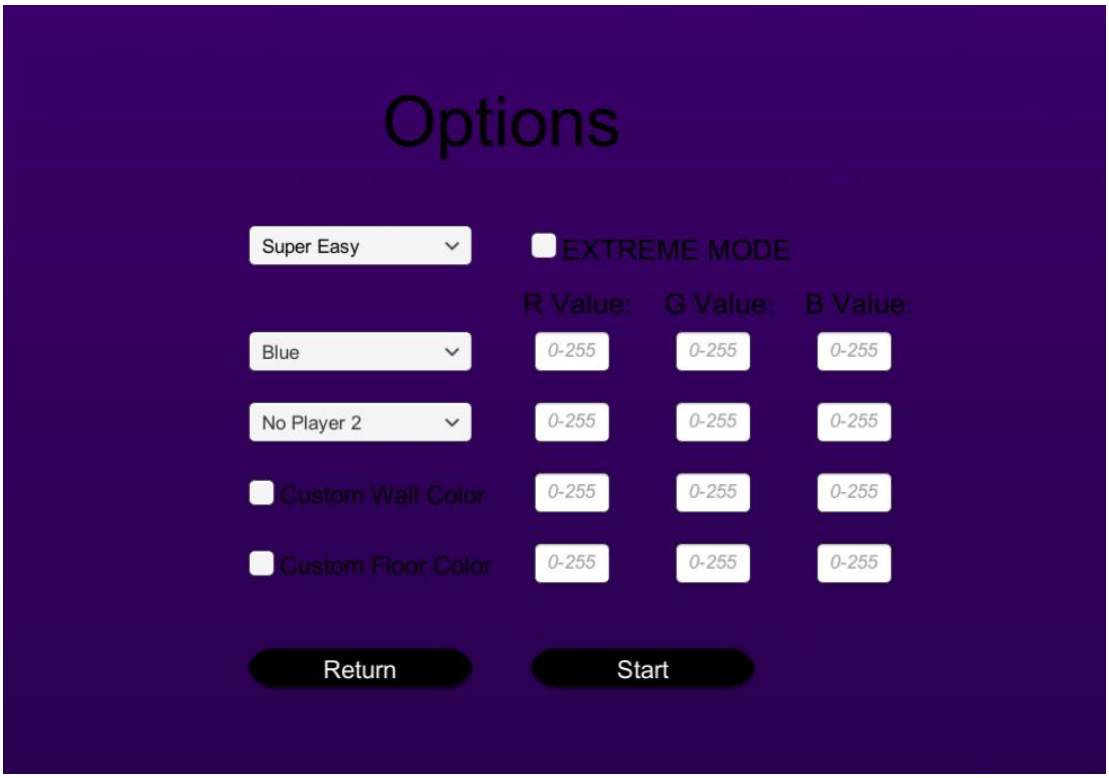


FIGURE 9.2

The Options screen shows the options for the game. Here, the player chooses the difficulty level (size), whether they want to play in extreme mode, and the colors of the players, walls, and floor. The colors for the players can be custom using RGB values or chosen from pre-set colors. The walls and floors can only be customized using RBG values. The Return button takes the player back to the Main Menu (Figure 9.1) while the Start button starts the game using the settings.

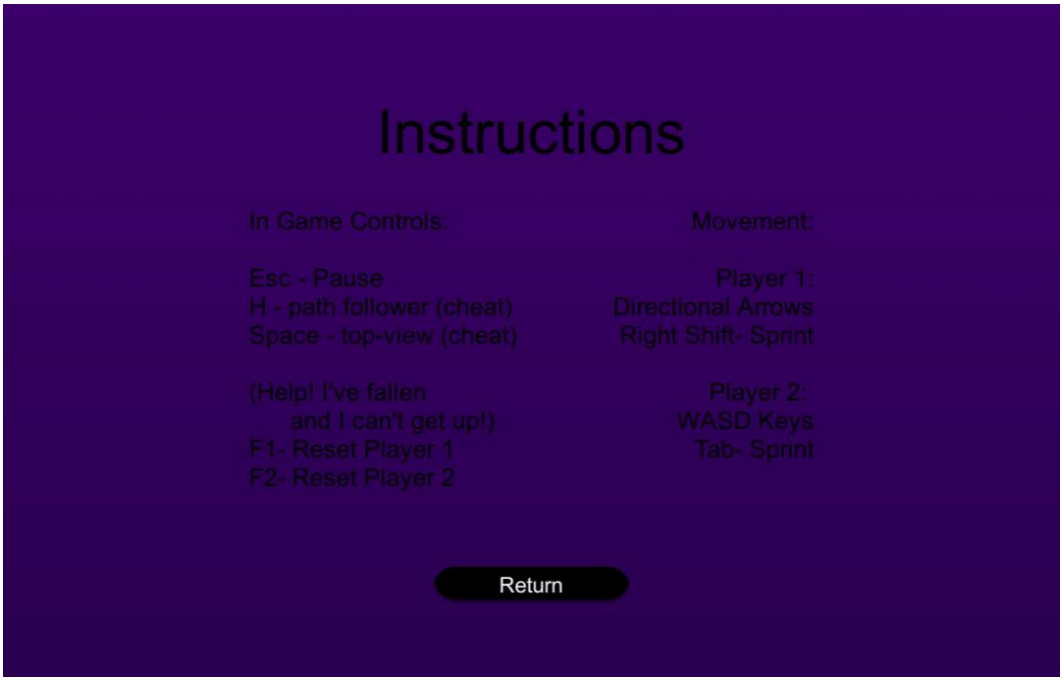


FIGURE 9.3

The Instructions show all of the directions for playing the game. It includes the in-game controls to use the Pause screen (Figure 9.5), features that are considered cheats to help the player navigate the maze, and buttons that will straighten the player in case they tip over their character. The movement section of the controls tells which buttons the players use to move their character. The Return button at the bottom of the screen returns the user to the Main Menu (Figure 9.1).

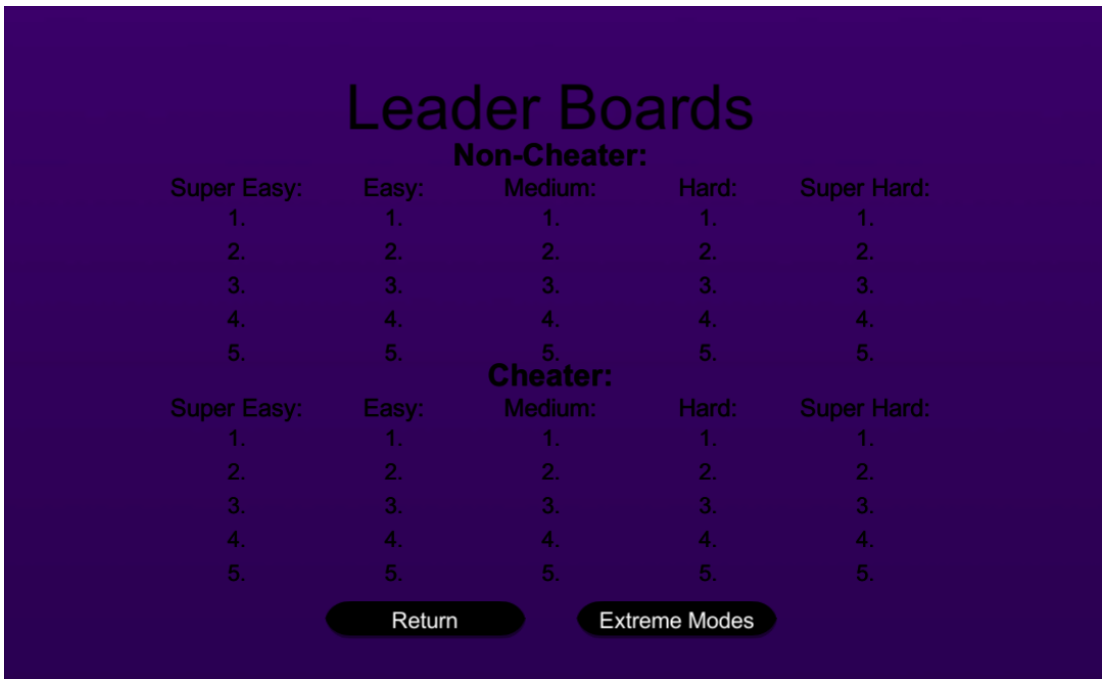


FIGURE 9.4

The Leaderboard was the last screen to be implemented. The top five times of each level are displayed. If a cheat was used, then the time is displayed on the Cheater section of the board. If not, it's displayed on the Non-Cheater portion. There are four different scoreboards to keep track of: the Cheater and Non-Cheater sections of both the Regular and Extreme Modes. The button at the bottom of the screen toggles between the Regular and Extreme Mode scoreboards. The Return button returns the user to the Main Menu (Figure 9.1).

The board shown above has no times shown because times cannot be added to the file that they are saved in. Each scoreboard has one file associated with it for a total of four files. To reset a board, the associated file must be deleted. The next time the game is played, it will create a new, empty file.

FIGURE 9.5

While doing a test-run of the game, I realized that a pause option would be beneficial for the user. The Paused screen has its own script to control how it interacts with the game. When activated by pressing the escape key, the screen pauses the clock and deactivates the player(s) so they can't continue moving. The Resume button will deactivate the screen and resume the game. If resumed, the clock will continue its count and the player would be activated again. The Main Menu button quits the current level and puts the user back at the Main Menu (Figure 9.1). The Exit button quits the game.

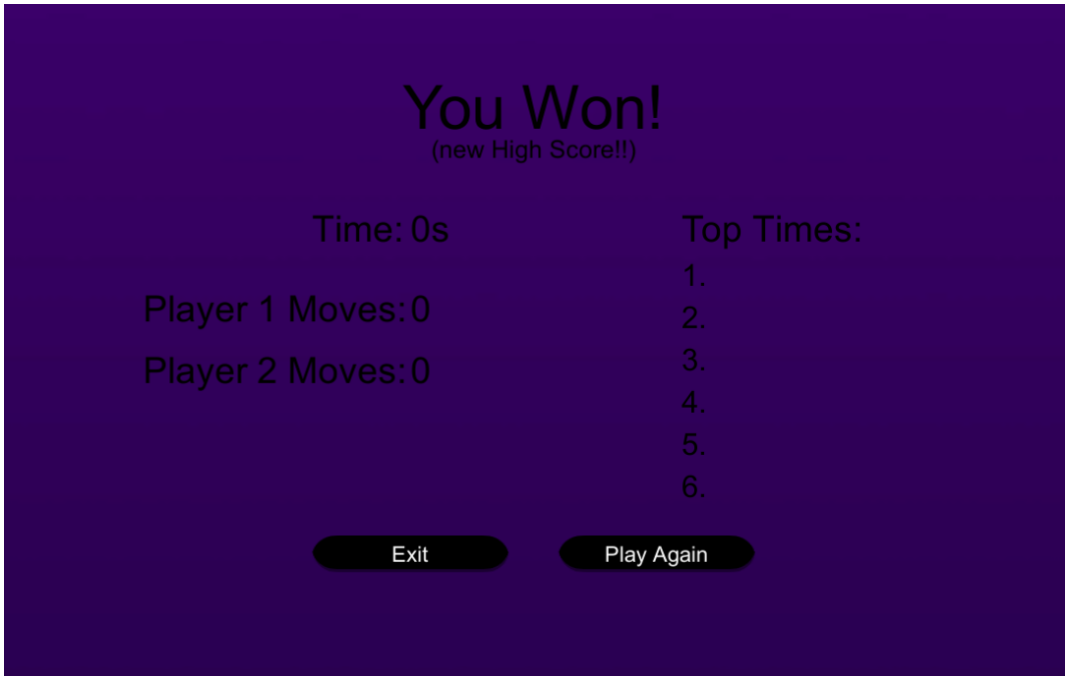


FIGURE 9.6

The Completed screen appears only when a character has completed the maze. It shows the time and number of moves it took to complete the maze. If the completed time is a new high score, the player is alerted and the Leaderboard (Figure 9.4) is updated along with the Top Times section. The Top Times shows the fastest times for the current level for either, depending on if a cheat was used, the Cheater or Non-Cheater board. The Play Again button will direct the player back to the Main Menu (Figure 9.1) and the Exit button will quit the game.

Like with the Paused screen (Figure 9.5), the player(s) is deactivated so that they can't move their character. Otherwise, if a player backed up then moved forward, "winning" again, then the moves and time would get updated to the reflect the new win. If the user decides to play another game, the player gets activated again.

Playing:

The game consists of five levels of various difficulties ranging from super easy to super hard. Originally, the sizes were 10, 25, 50, 75, and 100 squares. However, while testing out the 100x100 maze, it became apparent that the computer's performance became hindered with the number of objects needing rendered. The levels were then reduced to 10, 20, 30, 40, and 50 squares to reduce the strain on the computer. As shown in Figure 10, the blackout mode, EXTREME MODE, would make the walls and floor a deep black and allow the player little light to navigate the maze by. The Finish Line is especially hard to discern and very easy to pass in this level because it, too, is black.

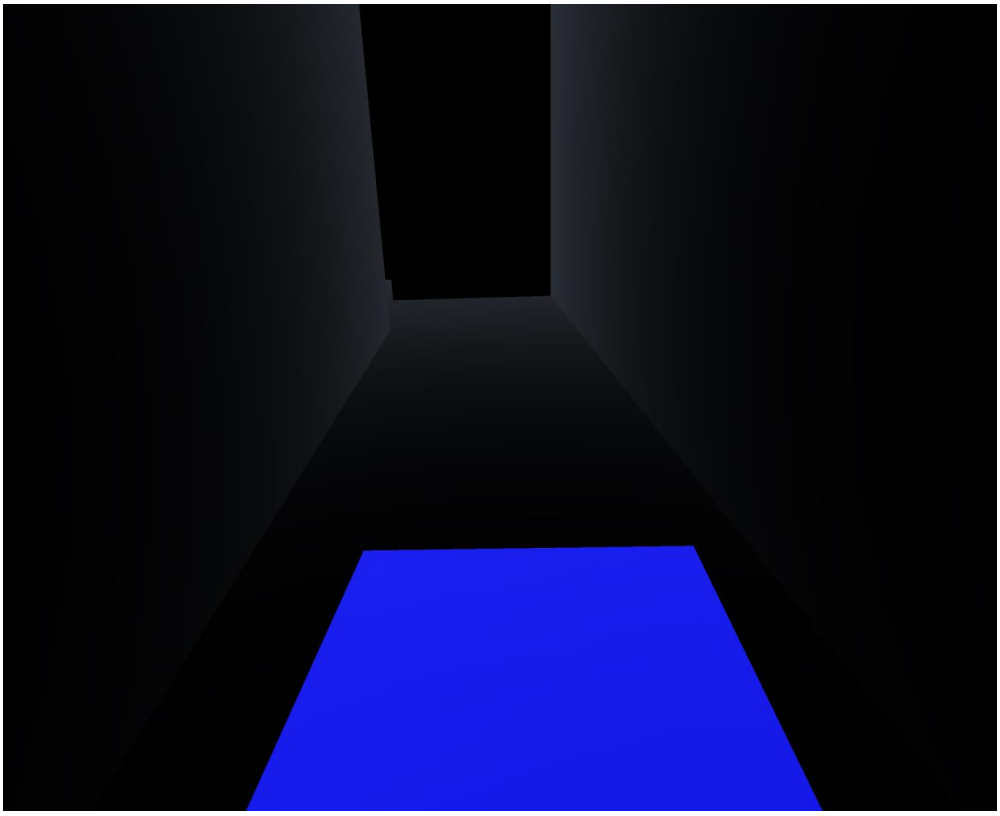


FIGURE 10

After getting the game running for one player, I created the option to have a second local player. Both players have the option to choose their character and color from the Options Screen (Figure 9.2) before starting the game. In game, because Player 1 uses the directional keys and Player 2 uses WASD. As shown in Figure 11, the screen is split down the middle with Player 2 on the left.

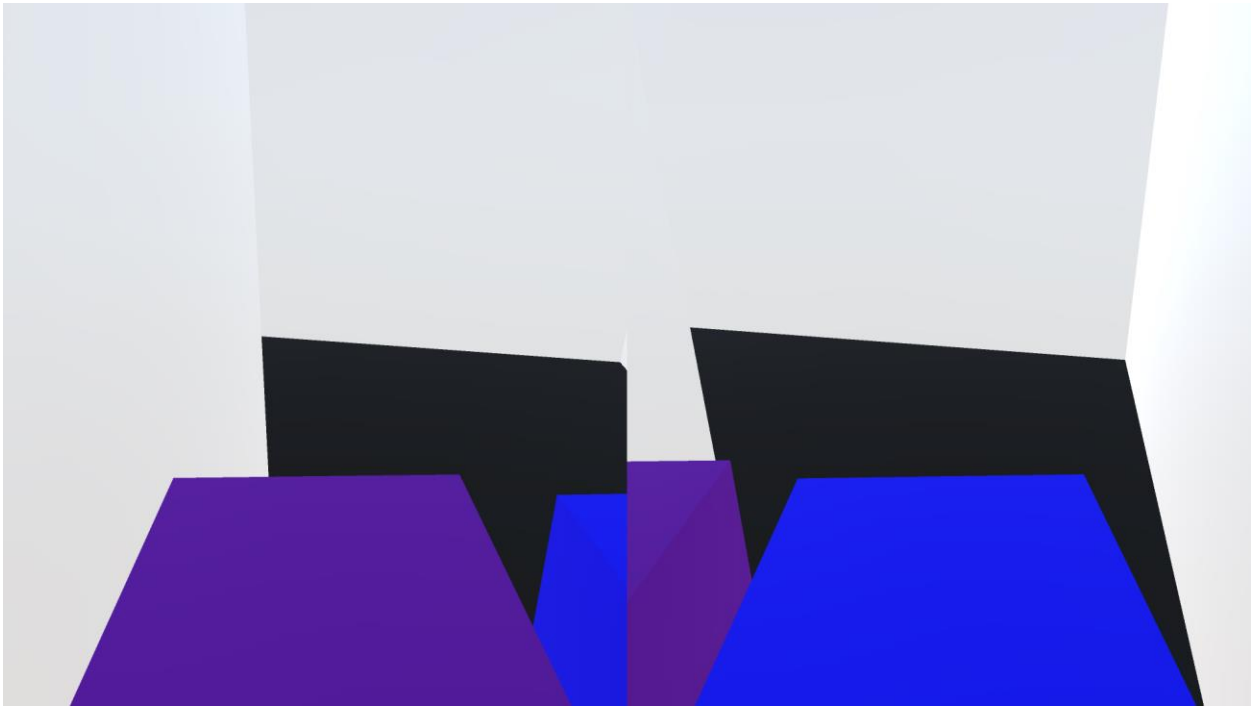


FIGURE 11

As displayed in Figure 12, there is a bug with the players that, when ran continually into a wall, there is a chance of them “falling down”. This means that the character got rotated in a manner that would not allow completion of the maze. This problem would not be solved if the player quit the maze and started a new one. The only solution is to restart the game. To combat

this, a player reset button was added that would return the player to the proper rotation in its current position. Player 1 and 2 each have their own buttons for reset, F1 and 2 respectively.

FIGURE 12

Another bug encountered was that when one character is sprinting using the shift button closest to them, the other could not sprint. Sprinting allows the player to travel faster which is useful when racing. The solution to this problem was simply changing the sprint buttons to be CTRL and TAB for players one and two respectively.

Upon running into the Finish Line (Figure 6), the game would be completed and the player would be directed to the Completed screen (Figure 9.6).

Cheats:

Once the maze was fully functional, I created various options to help the player complete the maze that are considered cheats. Using these will result in the completed time to be on the Cheater section of the Leaderboard (Figure 9.4). One such option is the top-down view of the maze (Figures 1.2 or 5.1) that is toggled by the space bar. To create this cheat, I placed a camera perpendicular to the board and made the projection orthographic which removed the shadows and made the picture clearer. Each increasing maze difficulty level would increase the size of the picture the camera had to capture. Therefore, it becomes increasingly difficult to view the character(s) in the maze. In the extreme mode, the top-down view is almost obsolete because the entire screen is black apart from the character and any trails it created (Figure 13)

Another available cheat a trail that can be turned on that shows where you've been. The player object comes with a trail renderer that is turned on by pressing the H key. This would allow a player to find their way out a dead end easier and would inform them of places they already explored. Because the following trail is white, it shows on the extreme mode very well. (Figure 13)

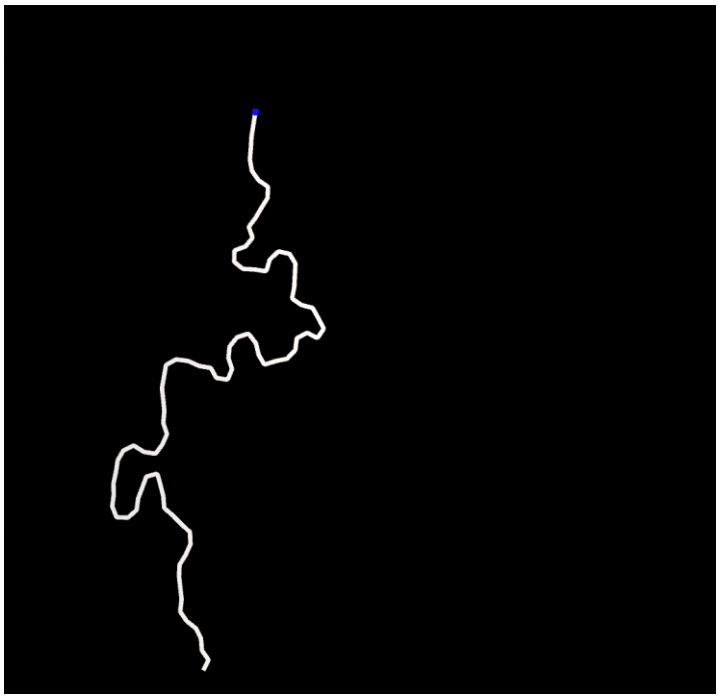


FIGURE 13

Customizations:

The Options screen houses all the available customizations for the maze. The trail renderer and player color customizations are executed in the InitializePlayer script. The trail renderer is part of this script because it follows the player. The wall and floor color customizations are in the Modes script. These customizations are done by the player choosing RGB values (Figures 14.1 and 14.2). As shown in Figure 10, if the extreme mode is checked, the wall and floor customizations would be overridden to black.

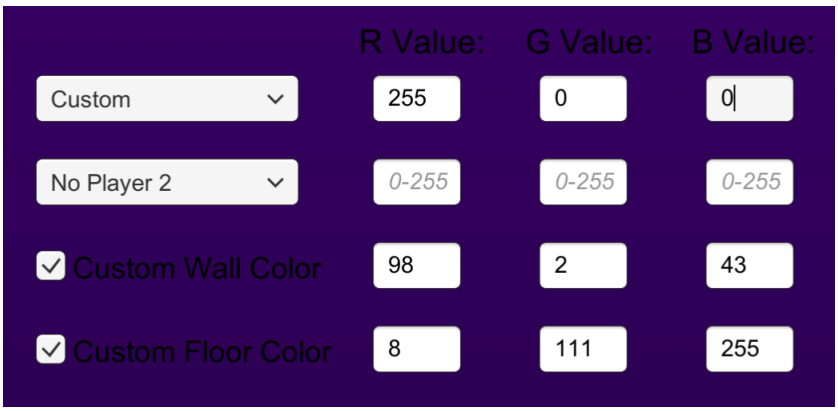
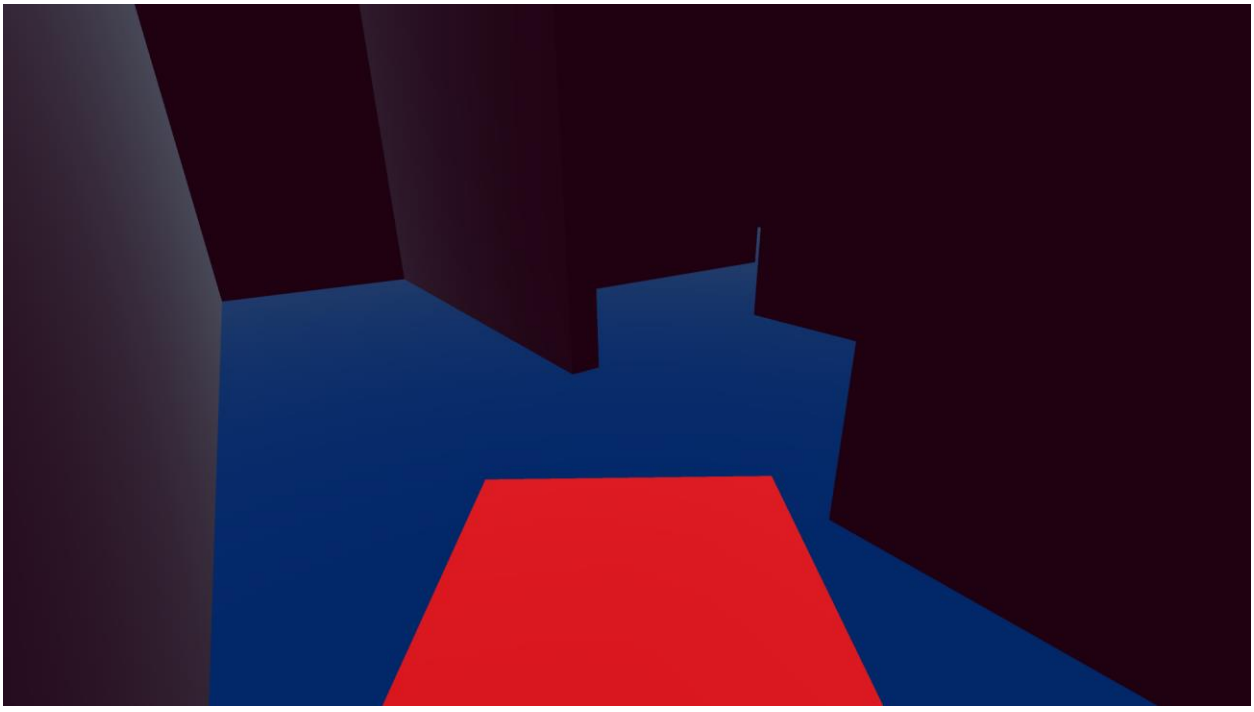


FIGURE 14.1

FIGURE 14.2



Figures 14.1 and 14.2 show floor, wall, and player customization. If a wall or ground color isn't chosen by the player, they are set to their default values (white and black respectively). If customization is selected and the player forgets to type in a value, it is defaulted to 0. If the value surpasses 255, its value will stay at 255. One problem that occurred when implementing this feature was that the inner walls were being created before the outer walls were being colored. Because the inner walls are clones of the outer ones, without the outer walls being colored first, the inner walls were created white, while the outer ones were the custom color. To fix this problem, I forced the Modes script to execute before the MazeCreator script. This ensured that any customizations would be applied to both the inner and outer walls.

Unlike its Java counterpart, the Unity version of the game only allows for one mode, extreme mode, instead of three modes of increasing difficulty. In Java, as shown by Figure 15, these modes would black out the screen apart from a small circle of light around the player(s). The more extreme the mode, the smaller the circle would be.

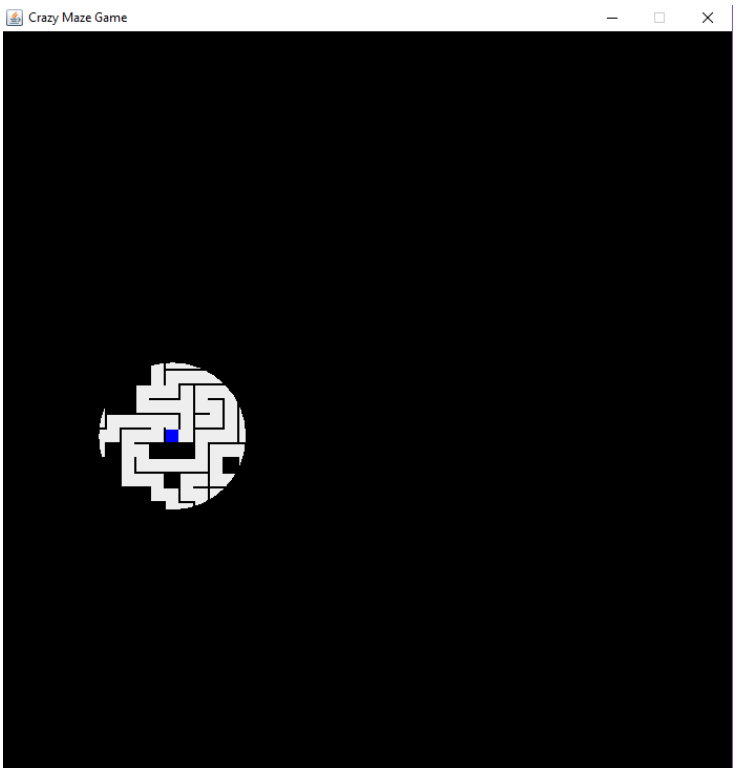


FIGURE 15

In the Unity version, I opted out of multiple modes after many failed attempts at blacking out the screen apart from a flashlight used by the player. After making the walls and ground the deepest black I could create, the game refused to allow a flashlight to be used; the light wouldn't appear. However, as seen in Figure 10, the player would still be able to see a small way in front of them just like the halo in the Java version. This happy coincidence also showed that the mini map was completely black apart from the player (Figure 13). Happy with this, I opted to only have the one mode.

Conclusion/Future Planning:

The game is a procedurally generated maze for the user to traverse. After customizing the maze to suit their taste, one or two players can go through and race the other to the end. High scores are kept for the user to strive to beat. After completing this project, I have further increased my knowledge of software engineering practices and problem solving abilities. This project has also broadened my experiences to include using a game engine that could be beneficial to future projects. I have also expanded my knowledge of languages to include C# and the Unity format.

In the future, I would like to expand this game to have online multiplayer access instead of just local multiplayer. It would also expand to have an all-time scoreboard that would show the fastest times of all players. Another expansion would be to allow players to create a custom maze and upload it for other users to complete. Later, more characters, shapes, would be available for the player to choose from and there would be an option for the player to upload an avatar to use instead of the shapes.