Isha Sathe
CSE 205 Honors Project
Maze Escape!
04/18/2020

INTRODUCTION

I chose to complete this honors project because it gave me a chance to expand my knowledge in GUI and create a game that I found interest in while simultaneously earning honors college credit for the course. The goal of this project is to create a game that is in the form of an advanced GUI application with an attractive and user-friendly interface.

REQUIREMENTS

For this project I've chosen to implement a maze escape game for which I need to meet the following requirements:

- A selection of mazes to choose from
- As well as a selection of backgrounds and characters
- A points system for completing mazes
- A way of keeping the character in bounds
    - Screen bounds
    - As well as maze bounds to prevent cheating
- Player control for the character
- A restarting mechanism

APPROACH

Deciding Game/Basic Design

The first step of this project was to decide on the game I wanted to create. When I was younger I always enjoyed doing mazes so I thought it would be a cool thing to create using JavaFX. I decided that I want to be able to choose different difficulty levels and have different characters and backgrounds which would add to the user experience. As for the interface I wanted it to have a fun and colorful design. I decided that the best way to implement this using JavaFX would be to create a separate class for each the background, maze and character. I decided to start by creating the main application class MazeEscape where I could test each of the other classes I was creating by adding them to the scene and stage. I also decided that each class besides the MazeEscape and Menu class will extend the Pane class because that allows me to the add an instance of each of my classes to the MazeEscape class as a Pane, otherwise there is no way of adding the classes I've created to the main scene and then stage of my MazeEscape class.

Designing The Background Class

I started off with the Background class because it seemed like the easiest task. At first I created the class and used the constructor as the way to create each background that I wanted. But when I tried to use this in the EscapeMaze class it didn't make much sense because there was no way to change the background without instantiating the object again. So I changed my approach and created a method inside the Background class called setBackground that takes a string as a parameter which can be 1 of 3 background choices, "day", "night" or "sunset". I chose these backgrounds because it seemed like an attractive user interface and one that I personally liked. When creating the "day" background I wanted to create a bunch of clouds, which I could have

approached in many ways. Each cloud consists of 4 circles made from the JavaFX Circle class so I could have created a loop to make all of these circles and then a loop to add all of them to the Pane. Another way, which I chose, was to create a nested class whose constructor created a cloud. This way I could create an ArrayList of type Cloud and then add that to my Pane. The Cloud class also extends Pane which is important because otherwise there would be no way to add the clouds to my background Pane. This seemed like the easier approach to me so I implemented the same approach when creating the "night" background and wanted to create a lot of stars. The Stars class is also a nestled class that creates a star (diamond) for the "night" background. The third background "sunset" is much simpler but I needed to research how to create a gradient in JavaFX in order to implement it. Each background is in a "switch case" and the backdrop is an instance variable because it is common between all backgrounds. Since each "case" of the switch-case adds to the Pane, the Pane first needs to be cleared so that the new background isn't being stacked on top of an old one when a new one is selected which would be inefficient.

Designing the Maze Class

The Maze class was an easier task but by far the most tedious. Just as the Background class it has a method called setMaze which takes a string as a parameter which can be one of the three difficulty levels, "easy", "medium" or "hard". To create the maze I declared the ArrayList lines of type Lines as a protected instance variable, in order to promote encapsulation and also so that I could refer to these Lines later in the MazeEscape class to check bounds. In each switch-case a different number of lines is added to the ArrayList depending on the difficulty of the maze. As it can be seen in the program the harder the maze the more lines it has. Just as the Background class, in each case of the switch-case I start off by clearing the Pane. This is even more important in the Maze class because if the lines were to stack on top of each other the maze would be entirely unplayable. Each of the cases also set the width of the Line strokes to 5 and add all of the Lines to the Pane once they are all instantiated. The Maze class also has the private method makeMarkers() that adds a "start" and "end" marker to each maze and is called at the beginning of each case in the switch-case.

Designing the Character Class

The Character class was also fairly straight forward but took a lot of trial and error to get the characters to look "friendly". Each character has the same circle shaped body so I created the instance variable "body" of type Circle for it. And just like the Background and Maze class the Character class has a setCharacter method that takes a string as a parameter that can be one of three characters, "bird", "pig" or "frog". Each of the cases in this switch-case also start off by clearing the pane and then once all the "body parts" (shapes) have been created they're added to the Pane.

Designing the Menu Class

The Menu class is the class that has all the nodes where the player will be selecting their backgrounds, mazes and characters. To do this I declared a ComboBox of type String for each of these selections, and a label to label each one. As well a restart button to accomplish the restart

requirement, and a score label to accomplish the requirement of a score indicator. And also a welcome label to welcome users to the game and a message label to report various messages such as a win or an out of bounds message. The ComboBoxes, Button, score and message Label were all declared as protected in order to be able to access them in the MazeEscape class which is where I will set an action on each of those nodes. The difference between this class and the rest is that it extends VBox rather than Pane and this is simply because a VBox would allow an easier configuration of the nodes rather than a Pane or any other extension of Pane.

Designing the MazeEscape Class

The MazeEscape class was more of an in between step of each of the previous steps but can only be focused on after each of those classes were made. First the MazeEscape class was treated as a testing class. An object of Background, Maze, Character and Menu were created in this class and then each set method was used to test the methods and then each object was placed on the root pane and then displayed using the scene and stage. This is the only class that does this because MazeEscape is the main application and extends the java class Application. Some of the instance variables are objects of the Background, Maze, Character, and Menu classes along with their following default Strings that will be used to set each. The rest of the instance variables are required for the event handlers implemented as private nested classes in the MazeEscape class and will be better explained in the explanation of each of these classes.

Designing the CharacterMover Class

The CharacterMover class extends the EventHandler class and listens for a KeyEvent. This was by far the most challenging part of the project. First I had to understand how the character object was being placed onto the root pane of the MazeEscape class in order to be able to understand how to move it. First I had to go through some JavaFX documentation to find the relocate method which allowed me to relocate the character pane to a certain point. The handle method has a KeyEvent parameter and then the switch-case in the method gets the key code of the key pressed. On my laptop the arrow keys aren't fully functional so I chose to implement "A, S, W, D" as my arrow keys as some other games use as alternatives. First, before the switch case I have to find the xPos and yPos of the character and I do this by calling the getLayoutX and getLayoutY methods of the Pane class (which is possible because Character extends Pane). Then in each case I first check if the increment is in bounds and if it is then it can move (20 positions up for "W", down for "S", left for "A" and right for "D"). Then it gets more complicated. In order to check if a player is cheating (crossing over the maze lines) we have to check if the character is ever in the same position as any of the lines in the maze. In order to do this we first have to access the ArrayList of Lines from the Maze object. Then in a for loop (incrementing for the size of the ArrayList) we have to cycle through all of the lines and get their starting and ending x and y positions, this can be done using the methods from the Line class: getStartX(), getEndX(), getStartY(), getEndY(). These methods are called on maze.lines.get(i) because this is where each Line object is stored. While cycling through this loop an if statement checks if the character object has the same xPos as the line and if it does and if it's in the y range of the Line then the character is outBounds, setting the boolean value to true. Else it checks if the character

has the same yPos as the line and if it does and it's in the x range of the line then the character is also outBounds, setting the boolean to true. I check if the x is the same and then y is in range (or vice versa) because these are straight lines so if the character is in the same x (or y) as any of the lines it's safe unless it's also in the y (or x) range of that line (which is basically how a line works). And all of this is done before the switch-case so that right after all the cases it can be determined if the player has either won, cheated, or is still playing. If they're still playing nothing happens and if they cheated the "message" label from the Menu class is set to "OUT OF BOUNDS" until the player takes the next move. And if the player wins the message is set to "YOU WIN" and a point is added to the score count but this only happens if the game is "reset". This can happen a number of ways, as explained in other Handlers but in this handler the game is reset if a player goes out of bounds (cheats). Another key aspect of this handler is that when the setOnKeyPressed(new CharacterMover()) is called it is called on the scene. This took a lot of trial and error to figure out and it's because you can't place this action on a pane (the character itself) and the scene as a whole will "accept" the keyPressed event and then you can move the character (pane) based on that key.

Designing the Other Event Handlers

The other handlers were much easier to implement. The BackgroundHandler uses the setBackground(String) method from the Background class to set the background during any point and time of the game. This action is set on the bkgComboBox from the Menu object (which is why it's declared as protected) and the String chosen from the ComboBox is found using menu.bkgChoice.getValue() which is used in the switch-case in the handle method.
The CharacterHandler does the same thing and uses the setCharacter(String) method from the Character class and the action is set on the ComboBox for the character choice, this can also be done during any point of the game. The MazeHandler does the same thing with the setMaze(String) method but when a maze is changed the character is placed back at start and the game is restarted (boolean restart = true). The RestartHandler sets the character back to start, restart=true and the message is set to empty again.

Miscellaneous

Throughout the coding process a lot of the steps consisted of commenting to make sure I understood the program for whenever I need to look back at it and so anybody else who looks at it can understand what is happening. There was also a lot of pseudo code for each class, I would sketch out the basics I needed for each class to help me get started, which also helped me draw the UML diagram. There were a lot of System.out.println("this works!"); statements that really helped me troubleshoot to see why something wasn't working, generally it was a scope issue and when the print statement wasn't reached this was a quick way to realize that.

DESIGN

The following is the final design of Maze Escape Game depicted as a UML diagram

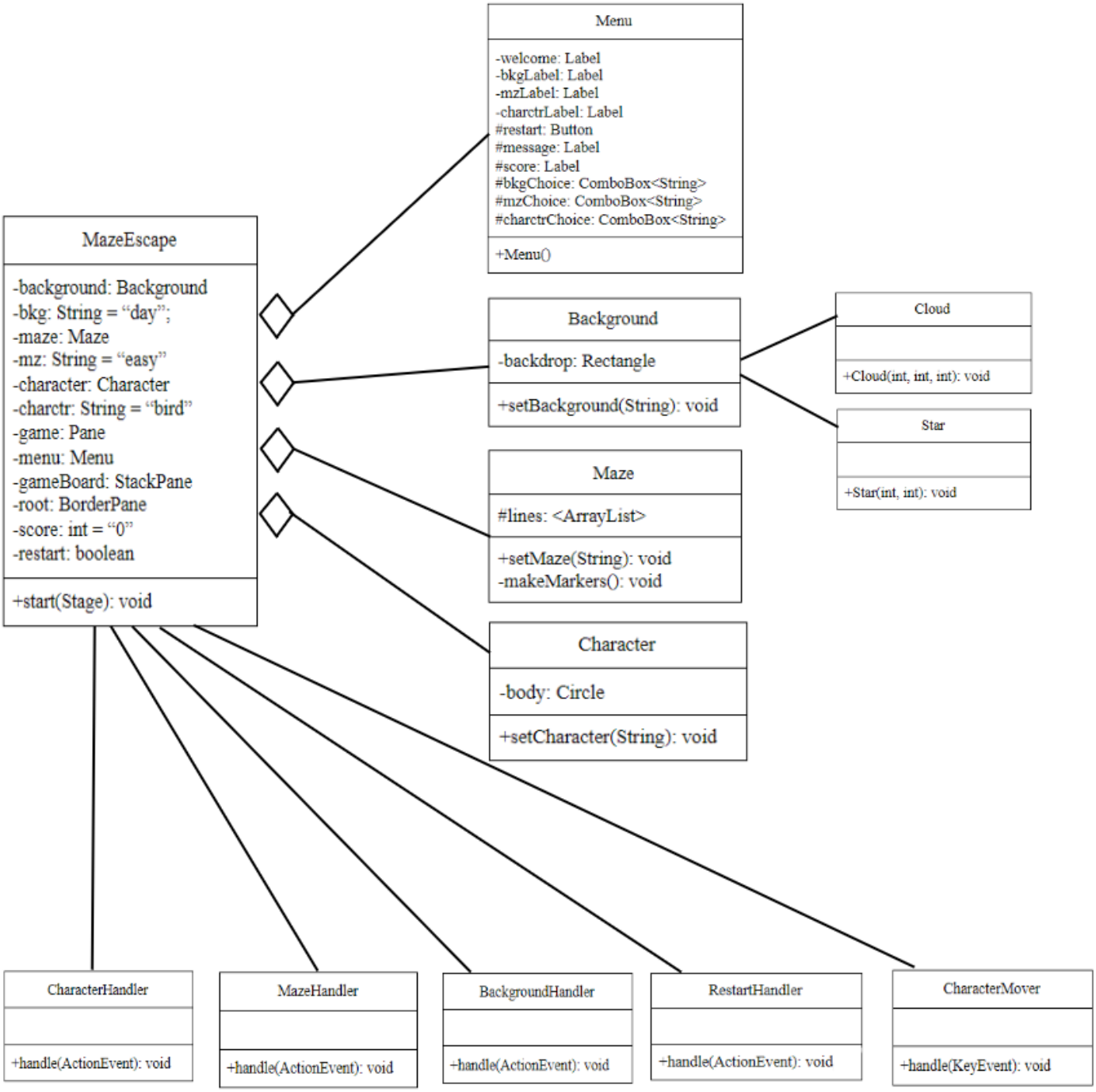

Figure 1

SIMULATION

When the program is first run:

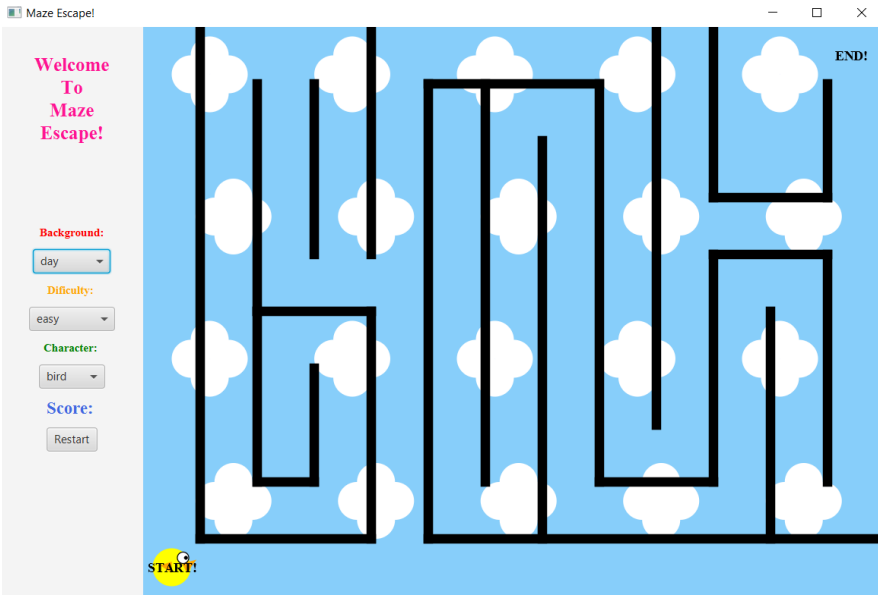

Figure 2

After moving around, sunset background and frog character chosen:

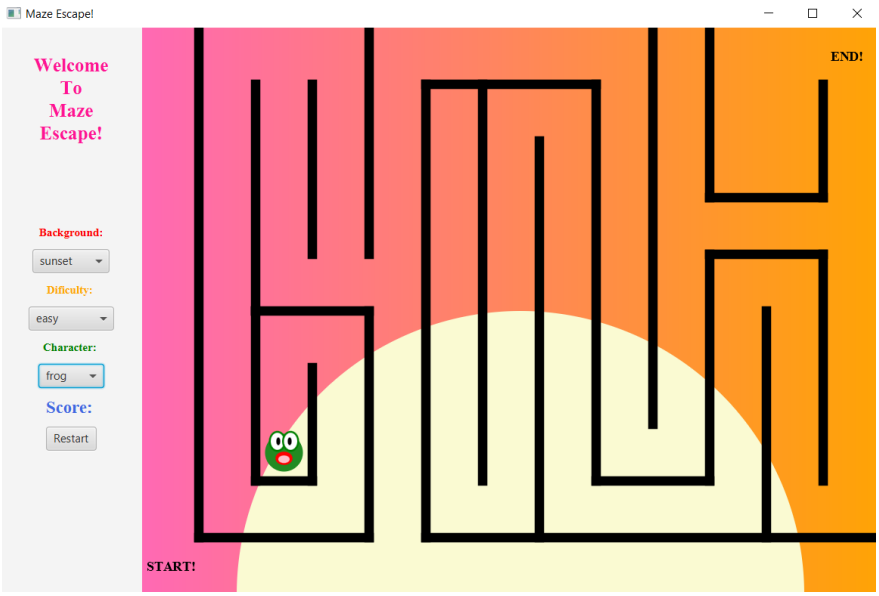

Figure 3

After choosing night background and pig character and clicking restart button:

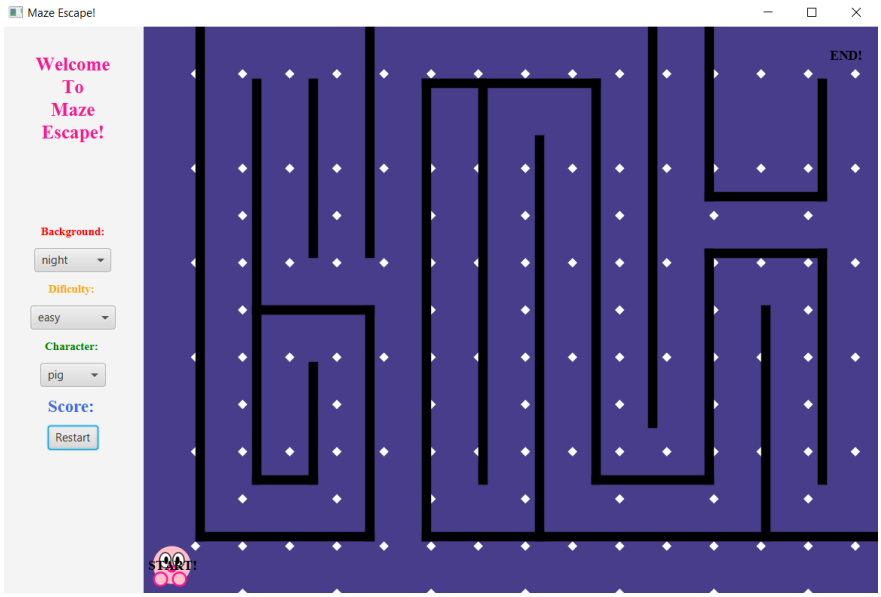

Figure 4

After choosing the medium level, day background and wining

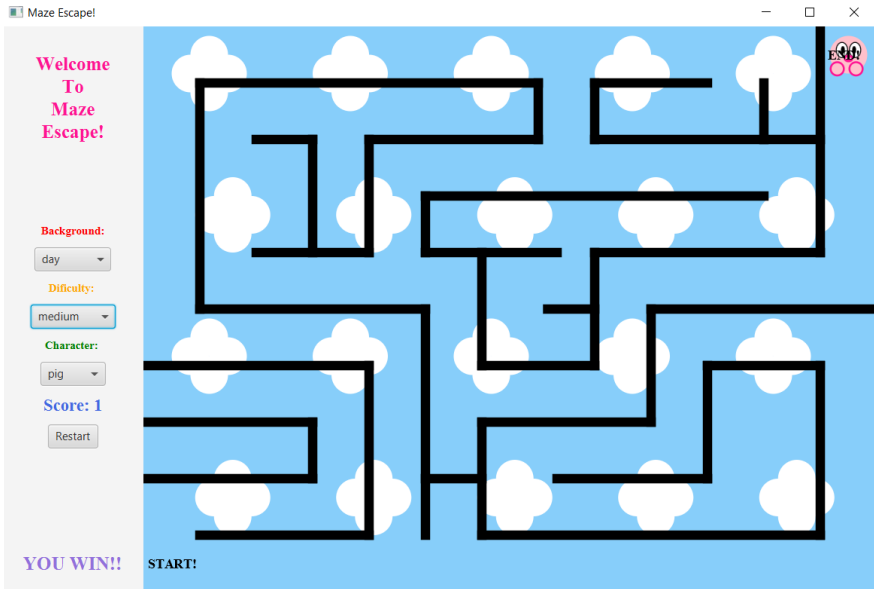

Figure 5

After already winning one, choosing sunset background, frog character and trying to cheat:

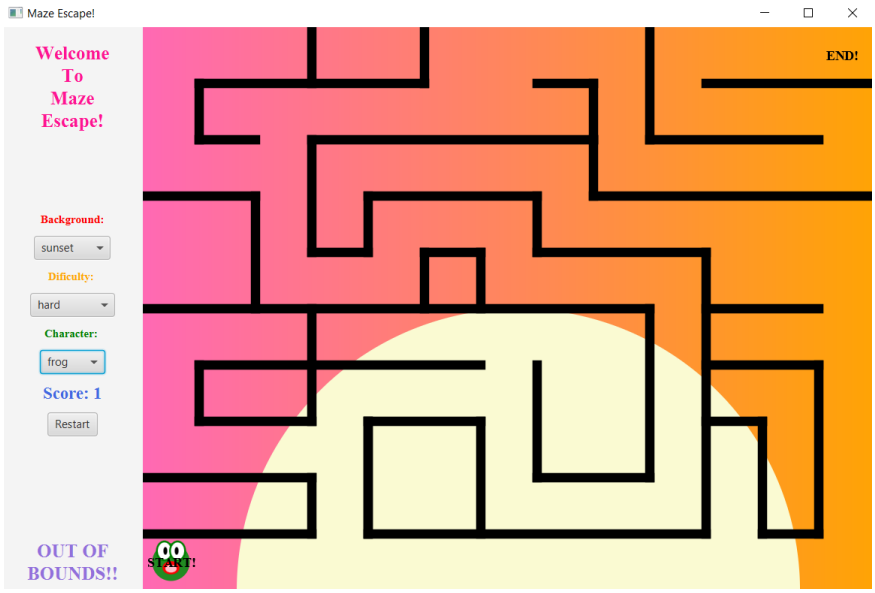

Figure 6

NOTE: There are many other test cases that could be run but that would require much more space than is realistically possible. Although, each feature is shown (out of bounds, restart, selection ability, scoring) and each selection of each category is also shown.