Video Game Design Final Report: Lazer

Jinwoo Yom

ECE 4974: Independent Study

Dr. Hsiao

May 10th, 2016

Table of Contents:

Introduction pg.3

Objective pg.3

Design Approach pg.3

- Map Generation pg.5
- Player Control pg.5
- Game Logic pg.6
- Non-Player Artificial Intelligence pg.7
- User Interface pg.9
- Special Effects & Art pg.12
- Play Testing / Updates pg.12

Conclusion pg.14

List of Tables:

Figure 1: Random Map Generation .... pg. 4

Figure 2: Procedural Map Generation .... pg. 4

Figure 3: Blinking Red Light During “warningTime” .... Pg. 6

Figure 4: No Blinking Red Light During “timeBetweenLazers” .... Pg. 7

Figure 5: A* Algorithm Practice project (Light blue characters searching for Dark blue characters) (1/2) .... Pg. 8

Figure 6: A* Algorithm Practice project (Light blue characters searching for Dark blue characters) (2/2) .... Pg. 9

Figure 7: Diagram of Lazer’s Game Structure .... pg . 10

Figure 8: Main Screen .... Pg. 10

Figure 9: Level Selection Screen .... Pg. 11

Figure 10: Game Play UI .... Pg. 11

Figure 11: Game Statistics accessed from Game Screen .... Pg. 13

Figure 12: Game Statistics accessed from Level Selection .... Pg. 13

Introduction

Lazer is a fast-paced, single-player survival game. The player races against the clock and other non-player characters (NPCs) to collect items on randomly generated maze-like maps while dodging deadly lasers coming from all directions.

Players can progress through three different episodes that each have six increasingly difficult levels. Difficulty depends on the frequency of laser blasts, map terrain, time limit, and the number of items to collect. Players can also monitor their progress through the game on the statistics board, measured by collecting and analyzing game telemetry.

Lazer strives to entertain players of all ages. It was built with Unity3D Game Engine as an independent research project at Virginia Tech by Jinwoo Yom '17.

Objective

The objective of this independent study was to fully design and build a complete game utilizing and expanding upon the knowledge and skills acquired from Video Game Design 1 (ECE4982 Fall 2015). Concepts that were used from this course were: A* path finding algorithm, NPCs that exhibit realistic personalities and traits, FSM / State-based game logic design, and procedural tilemap generation. Two new concepts that were explored and integrated were using coroutines to split up execution time and different AI techniques to control NPCs. The use of AI aimed to create NPCs that exhibited unique personalities and traits and to pass the Turing test.

The success of this game was measured through play testing and user feedback. As of May 9, 2016, nine players provided positive feedback. Further play tests are scheduled.

Design Approach

This independent study began with researching and brainstorming a variety of game concepts and ideas. The main inspirations for Lazer came from strategic, maze-based games like Bomberman and Crazy Arcade after wanting to really highlight both pathfinding algorithms and map generation concepts from the course.

Lazer was designed and built over these seven major milestones listed in chronological order below:

1. Map Generation: generating grid-based tilemaps, game space boundaries, procedurally spawning rocks, item generations.
2. Player Control: keyboard control, player positioning, collision detection.
3. Game Logic: finite state machine logic, randomly spawning laser blasts, laser warning signal, timer, progressive difficulty, win/lose scenario.
4. Non-Player Character AI: pathfinding algorithm (A*), different AI personalities.
5. User Interface: start screen, buttons, how-to-play, pop-up screens, level selection screen, progression memory management.
6. Art/Special effects: re-polishing code, art work using Adobe Photoshop, adding sound effects, adding explosions and other particle effects.
7. Play Testing/Updates: collecting player feedback, making necessary updates or adjustments, testing edge cases.

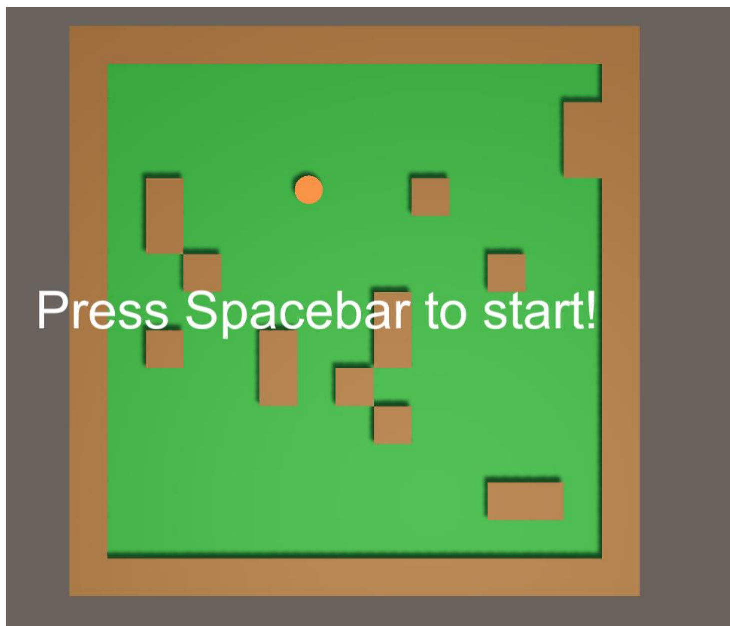

Figure 1. Random Map Generation

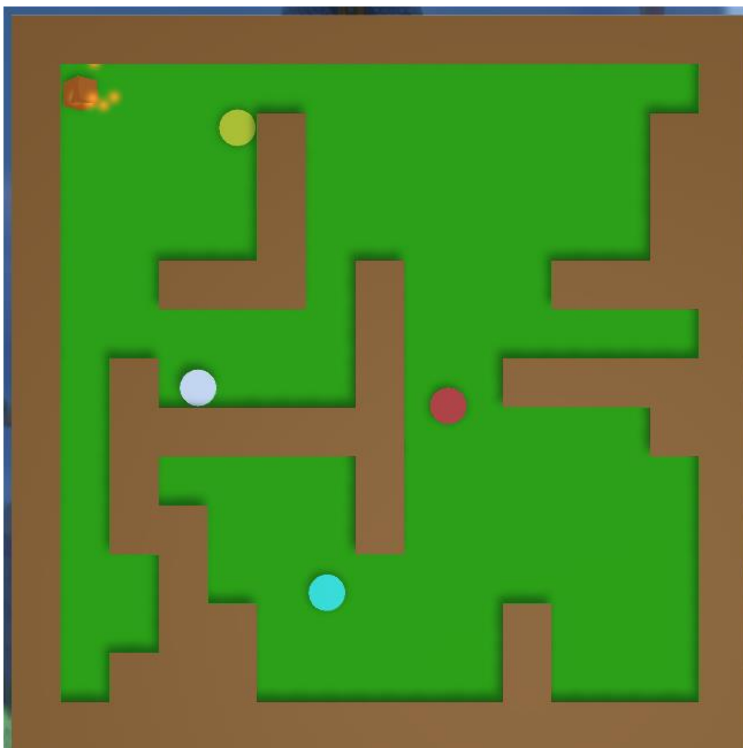

Figure 2. Procedural Map Generation

Map Generation

This was the first milestone of the design approach. It started with building the basic structure of the game scene. This includes a plane, where the game takes place, and boundary walls. The boundary walls are collections of smaller cube-shaped walls saved onto an array of

GameObjects. These walls have different tags depending on their orientations. For example, the walls on the top of the screen will be tagged as Northwall, whereas, walls on the left of the screen will be tagged as WestWall. These tags are used for the lasers to differentiate between different types of walls. Through this, the lasers can determine their starting location, traveling direction, and destination location.

In order to achieve procedurally generated maps, being able to have successful randomly generated map was the first step. To do this, a struct called emptySpaces was made. This struct holds information about each individual block space on the map. Each of the structs on emptySpaces are pushed onto a list called emptyBlocks, which is then used to easily choose a random empty block.

With random map generation complete, transitioning it into a procedural map generation was the last step of this milestone. A function called generateGroupOfRocks() in the GameController script uses a probability algorithm to group rocks. This algorithm generates a straight wall of rocks 70% of the time. However, it has 15% chance of turning either right or left to create a small "shelter" for the players. Additionally, the wall should never landlock anything, which means there has to exist at least one path from a player's position to anywhere in the map. A function called floodCheck() uses recursion method to check for this by counting all of the connected empty blocks from a single location on the map. It will then compare the number of of connecting blocks to the total number of empty blocks. If these values are equal, then there is no landlocking. If they are not equal, then there is landlocking and the block does not get placed in that location. Figure 1 and 2 above show the side-by-side comparison between random and procedural map generation.

Scripts that were written for this milestone are:

- GameController - The main script which manages most of the game such as the main game loop, wall generations, player generations, rock generations, and item generations. However, for this milestone, only the logic for wall generations were implemented. The rest were implemented later in other milestones.
- Rotater – Animates the pickup items by rotating the item in all three axis: x, y, and z.
- TileType – Allow the tiles to be serializable gameObjects.

Player Control

This is a simple, but very crucial, milestone since which provides an additional method of debugging (by playing the game). In order to do this, a class called PlayerBase is made, which holds all of the private and protected variables that a player needs. Additionally, it has a OnTriggerEnter() function, which is called during a trigger event and checks to see if the trigger object was either a point block, speed boost block, shield block, or laser.

PlayerController is a class that controls the playable character's movement using inputs from the keyboard. It inherits from the PlayerBase class to have all of the common player characteristic. Thus, PlayerController and PlayerBase are the only two scripts for this milestone.

Game Logic

This milestone mainly focuses on laser generation. Most of the laser logic resides in a script called WallScript. This script is owned by the outer edge blocks, which interacts with the main game controller to know when it is or when it is not appropriate to instantiate a laser block,

The variables warningTime and timeBetweenLazers refer to the amount of time the lasers blink before firing and the frequency of blasts, respectively (Figure 3 and 4 shows an example of this). It also keeps track of the game state to make sure it stops laser production when the game is not being played.

Once instantiated, LazerScript, held by the laser blocks, figures out which direction lasers need to travel by checking the tag of its currently colliding block. It will then destroy itself once it reaches its destination.

The three scripts that were made for this milestone:

- WallScript - Main script that manages laser production.
- Rockscript - Held by the rocks to destroy the laser.
- LaserScript - Figures out which direction the laser needs to travel.

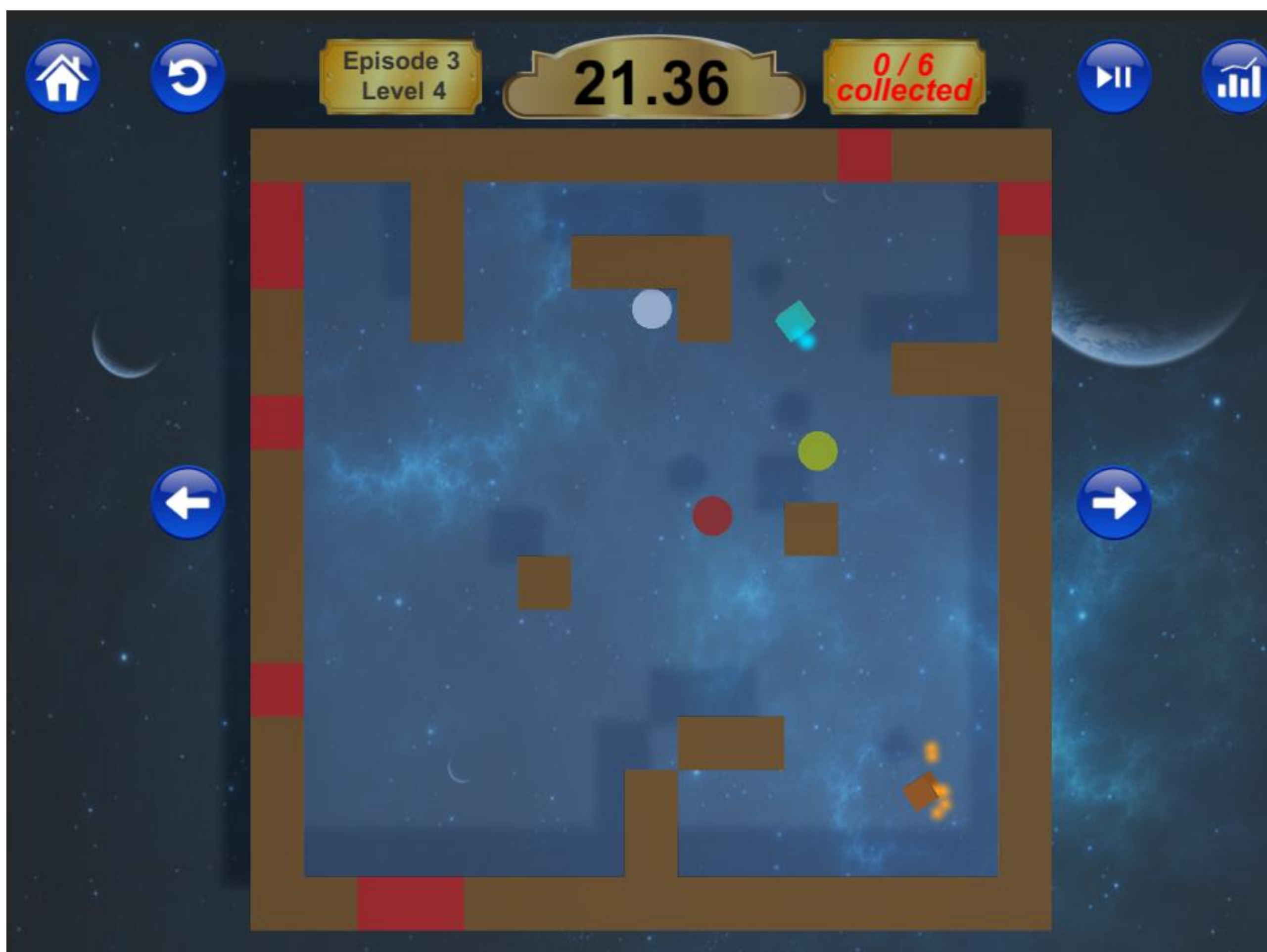

Figure 3.Blinking red warning light during "warningTime"

Figure 4. No blinking red light during "timeBetweenLazers"

Non-Player Artificial Intelligence

This milestone provided the basis for NPC behavior. Some of methods considered were Depth First Search (DFS), Breadth First Search (BFS), A*, and Dijkstra's method. After comparing these methods, it was concluded that the A* algorithm was the most efficient because Dijkstra's method, DFS, and BFS visited too many unnecessary nodes. The A* algorithm considers three values (H, G, and F) to make comparisons among nodes. Then, the nodes are stored in either an open or closed list, depending on if they should become part of the path or not. A* efficiently updates itself if the current path turns out to not be the most efficient path.

Before implementing the AI in Lazer, a separate Unity project was made to practice and perfect the path finding algorithm (Figure 5 and 6 shows this separate practice project). In a separate project, a pseudocode of the A* algorithm from a website (http://web.mit.edu/eranki/www/tutorials/search/) was translated to C# syntax. This was a very useful source that helped understand what the algorithm is doing step-by-step. Once the algorithm was successfully implemented, an optimization function was written to further eliminate any path redundancy.

For example, if the object path is traversing through a linear path, the optimization function removes all of the nodes in between and just keeps the starting and the ending points. Furthermore, a heap is implemented to store the nodes. This heap acts much like a binary tree, but the children nodes will always have less value than the parent nodes. This is so that it will have easy access to a node with the least value when implementing the A* algorithm. Finally, for processing multiple paths that are requested by multiple NPCs, a path request managing script

was required to queue all of the requests. Then, using Coroutine, each request is processed per frame to minimize the delay time of each frame.

For the AI players, an empty class called AIPlayer, which inherits from PlayerBase, is created to better organize the code structure. The reason it can be an empty function is because a Unit script already handles the AI movements for the AI players. Once the algorithm implementation was complete in the side project, merging the implementation over to Lazer was the biggest problem. There were many issues with incorrect positioning, since the algorithms needed to be modified slightly from world positioning to grid positioning.

After successfully merging the algorithm to Lazer, the final step of this milestone was to add characteristics. Two characteristics were implemented: itemPercentage and LazyPercentage. The itemPercentage dictates how likely the AI player will chase after items rather than points, and the lazyPercentage dictates how likely the AI is to give up on obtaining the item if it has lower chance of getting to it compared to other AIs. Instead, it will wander until an item appears in a location which it has better chance of obtaining. The main logic is coded in the function AIPathRequests() within the GameController script. Then, the function return control signals back to Unit in order to execute the AI's behavior.

These seven scripts were written to direct the AI:

- AIPlayer - An empty class that inherits from PlayerBase.
- Grid - Creates walkable and non-walkable node-grid systems for AI
- Heap - Container system used to ease the process of pathfinding algorithm.
- Node - Scripts for the nodes of A* algorithms paths. They contain gCost, hCost needed for A* algorithm.
- Pathfinding - Implements A* pathfinding algorithm.
- pathRequestManager - Managing script that enqueues pathfinding requests and uses coroutine to process one request per frame to minimize the delay time of each frame.
- Unit - Script that controls AI movement. It makes path requests, then on success, follows the returned path.

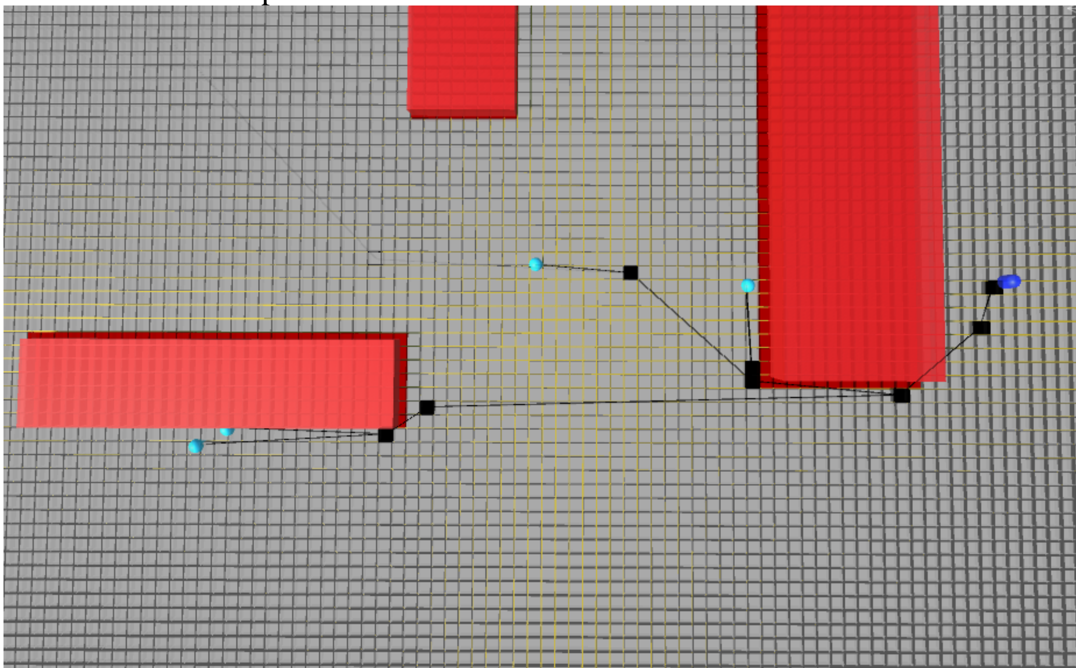

Figure 5. A Search Algorithm practice project (light blues characters searching for a dark blue character) (1/2)*

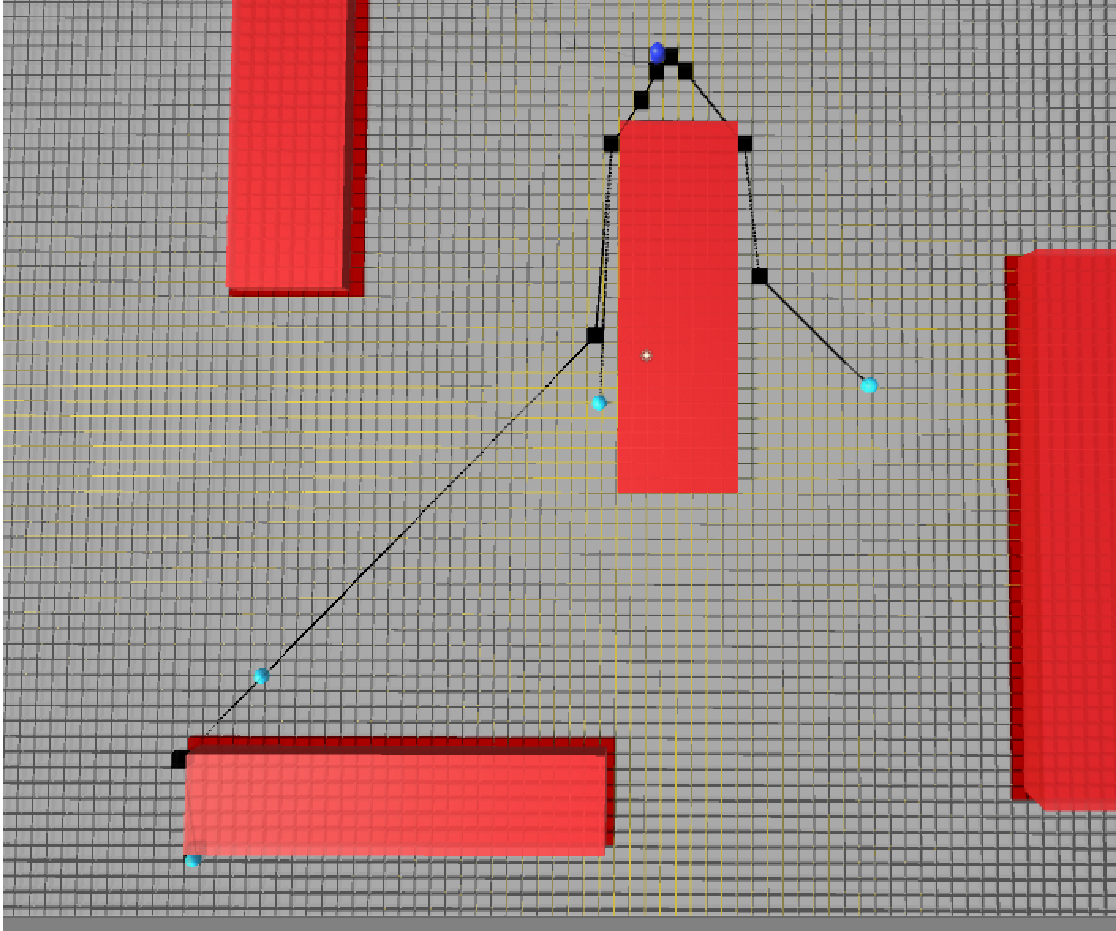

Figure 6. A Search Algorithm practice project (light blues characters searching for a dark blue character) (2/2)*

User Interface

User Interface (UI) is a crucial part of program, especially in a game. Presentation and user experience are the keys to success in today's society, so it is important to take the time to perfect it as much as possible. In other words, if a game has a horrible user interface, then the less likely players would want to play or have a positive experience.

The first step in designing the UI was creating a diagram of what the structure should look like before implementing it in the game. Figure 7 Below is a diagram of Lazer's UI structure diagram. Although it is simple and straight forward, it does cover all possible scenarios. The scripts menuScript and levelSelect are the main scripts that run the logic behind the menu screen and the level selection screen. During the game, UIController controls all of the UI. Each of these three scripts contain functions that control all of the buttons and displays for their screen. Two additional scripts, Destroyer and StartScreenLazers, are made for the laser effects on the two starting screens.

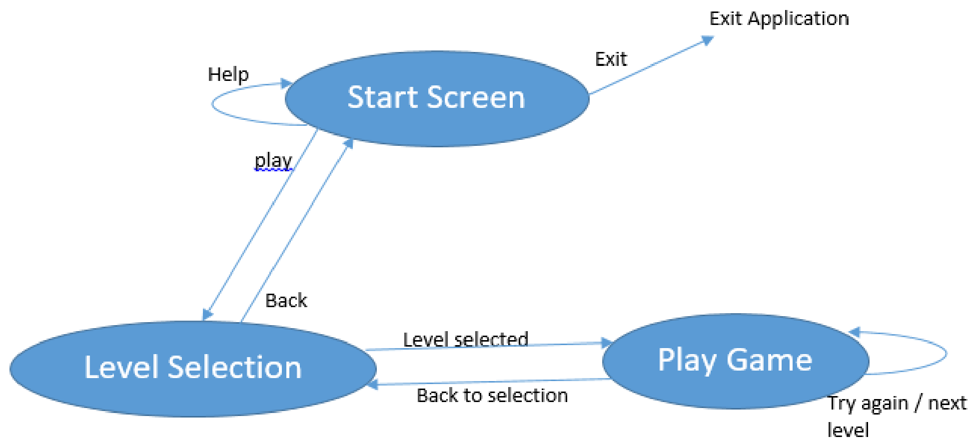

Figure 7.Diagram of Lazer's game Structure

Figure 8, 9 and 10 Below shows the UI of main menu, level selection, and gameplay. As it will be mentioned in Special Effects & Art section below, all of the art (not including the backgrounds) are made using Photoshop CSS5.

Figure 8. Main Screen

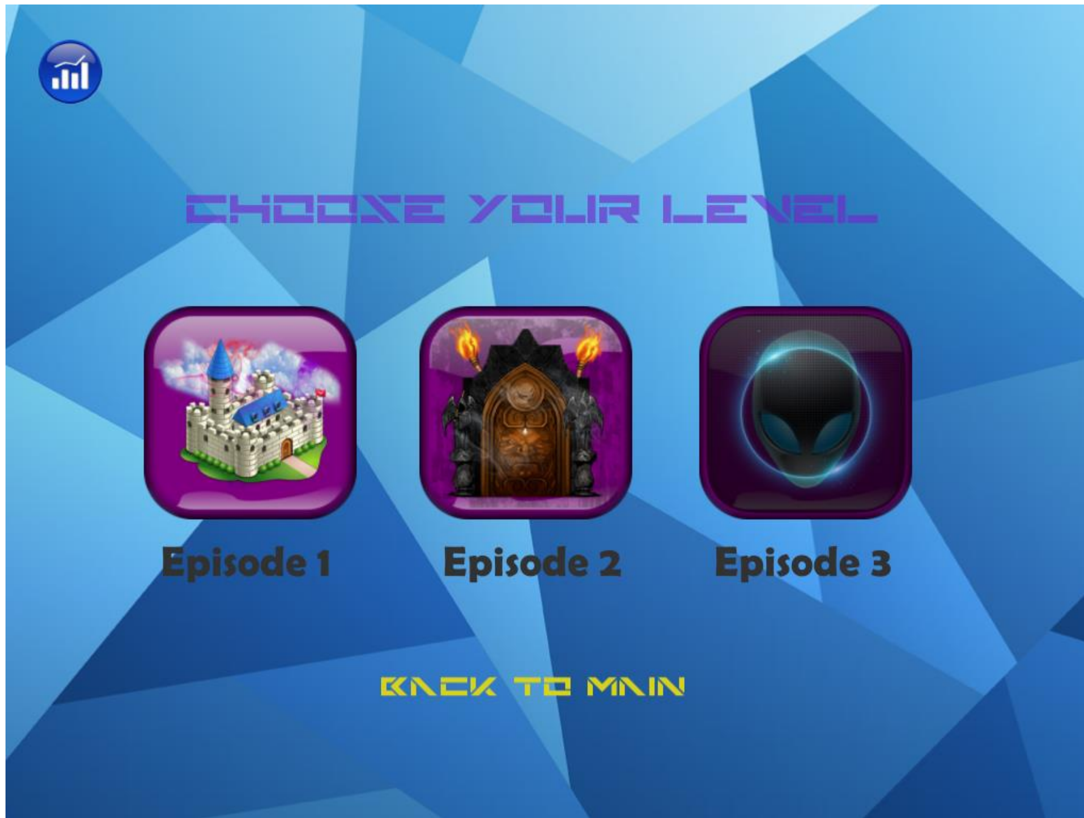

Figure 9. Level Selection Screen

Figure 30. Gameplay UI

The scripts made for this milestones are:

- Destroyer - A script applied to the invisible boundary walls to destroy the flying lasers in the two starting screens.
- levelSelect - A main logic script that controls UI in Level Selection screen.
- menuScript - A main logic script that controls UI in Main Menu screen.
- StartScreenLazers - Runs the laser effects on the two starting screens.
- UIController - Runs all of the UI on the game screen.

Special Effects & Art

All of the special effects used in Lazer were explosion effects for laser collisions, laser trail effects, particle effects for the items, and halo effects for players who obtained a shield. The trail, particle, and halo effects were all done using the special effect component in Unity's library. However, explosion effect was done by importing a free VFX effect available in the Unity store. One script called DestroyByTime is attached to this explosion effect so that the explosion gets deleted after certain amount of time.

As for the art, all of the pop-up screens and buttons were made in Adobe Photoshop. However, the backgrounds of the screens were imported from free-to-use websites. More information can be found in Credits from the Main Menu screen.

All of the music and sounds effect were imported from PlayOnLoop.com and FreeSound.org. All rights are reserved to these websites.

Play Testing / Updates

A total of nine players has play tested the game and gave positive feedback by May 9, 2016. Many gave great suggestions or reported bugs that helped improve Lazer significantly. One of the best suggestions was to implement a memory script that saves user's progress and prevents the user from jumping ahead without properly progressing from Level 1. Another great suggestion (shown in Figure 11 and 12) was to collect game telemetry so that users can view all of their stats on their progress. Users can view this window by clicking on either the icon on the top right in the game screen or level selection screen.

Figure 11. Game Statistics Accessed from Game Screen

Figure 12. Game Statistics Accessed from Level Selection

In order to create a memory script, a function called DontDestroyOnLoad() was used inside of memoryScript to keep the object that is holding this script from being deleted during scene changes. Inside, there are many private variables which were used for storing game data. These private variables are accessible by get and set methods from other scripts.

Conclusion

The objectives of this project were to design an AI to control NPCs, implement NPCs with realistic personalities and traits, implement FSM/State-based game logic design, and procedurally generate tilemaps. I believe that I have accomplished all of these objectives and even more. I have learned a lot about what it takes to be a full-stack game developer and how tedious the process can be. I am glad that I was able to utilize all that I learned in ECE4982 to guide me through this independent study. I also learned how to better manage my time when building a game that more complex and on a much larger scale.

In conclusion, I learned a lot about how to design and develop a video game from scratch. I feel confident utilizing a finite state machine to implement different states of the game, programming AIs and implementing different personalities and traits to make them more realistic, and designing procedural map generations by understanding how it enhances replayability to a game. Thus, I developed a larger sense of awareness in design planning and the full development of a game.

I would like to thank Dr. Hsiao for his guidance throughout this independent study and the play testers for providing critical feedback to improve the game.