迷路の再生

Maze of Rebirth

A Virtual Reality Horror Maze Game

Computer Science & Interactive Media and Game Development

A Major Qualifying Project Report

Submitted to the Worcester Polytechnic Institute faculty

In partial fulfillment of the requirements for

The Degree of Bachelor of Science

By:

Corey Dixon

William Kelley

Patrick Malone

Advised by:

Professor Clifford Lindsay

Professor Jennifer deWinter

Abstract

This report describes the creation process of *Maze of Rebirth*, a Virtual Reality game in which the player must escape the world of darkness, Yomi: a procedurally-generated multi-floor maze filled with a variety of dangerous creatures that the player must trick lead into traps in order to defeat. The project was developed over the course of three months at Ritsumeikan University Biwako-Kusatsu Campus in Japan with the help of the Noma and Takada labs, as well as a couple of months in Worcester, Massachusetts. This paper goes into detail about various technical aspects, the research done in preparation for the development of the game, and the way in which we tested players before and after having them play the game.

Acknowledgements

We would like to thank our advisors' Professor deWinter and Professor Lindsay for their advice and guidance throughout our project. The help provided by our advisors kept our development on track and shaped our design decisions for the better. We would also like to thank Worcester Polytechnic Institute for providing us with the opportunity to undertake this project.

We would like to thank Ritsumeikan University and its faculty for all of the assistance lent to us over the course of this project. In particular, we would like to thank Professor Noma, Professor Lopez, and Professor Takada for their help throughout our stay in Japan. We would also like to thank the students in the Media and Design lab and Distributed and Collaborative Systems lab. We would also like to extend our thanks to Edgar Handy for giving us advice and play testing our game.
We would like to thank Marco Duran for creating all sound and music assets for our game. Marco created three music tracks with inspiration from the anime Aku no Hana. Marco also created seventeen sound effects, such as footprints and various trap activation sounds.

We would like to thank Laurie Mazza for her advice on creating art assets for our game. Additionally, we would like to thank her for letting us use her skeleton model and animations for our Oni type enemy.

We would like to thank Brian Moriarty for play testing our game and providing feedback.

We extend our thanks to individuals who provide models for free online as their anonymous contributions aided the development of our game's aesthetics.

Authorship

Section	Writing headed by	Editing headed by
Abstract	All	Patrick
Acknowledgements	All	Corey
1. Introduction	All	Patrick
2.1. Experience Goal	All	Patrick
2.2 Background Research	William	Corey
2.3 Gameplay	All	Corey
2.4 Tutorial Design	William	Patrick
2.5 Level Design	Patrick	William
2.6 Environment Design	Corey	Patrick
2.7 Enemies	William	Patrick
2.8 Traps (Design)	Corey	William
2.9 Items (Design)	Corey	Patrick
2.10 User Story	William	Patrick
2.11 Conclusion	William	Patrick
3.1 Software	Corey	William
3.2 Telemetry	Corey	Patrick
3.3 Virtual Reality	Corey	William
3.4 Maze Generation	Patrick	William
3.5 A.I. Implementation	Patrick and William	Corey
3.6 User Interface	Patrick and William	Corey
3.7 Traps (Technical)	Corey	Patrick
3.8 Footprints (Technical)	Corey	Patrick
3.9 Items (Technical)	Corey	Patrick
4.1 Introduction	Patrick	William
4.2 Paper Prototype	Patrick and William	Corey
4.3 Study Design	William	Patrick
4.4 Experiment Design	William	Patrick
4.5 Methodology	Patrick and William	Corey

4.6 Results	Patrick and William	Corey
4.7 Conclusions	All	William
5.1 What Went Right	All	William
5.2 What Went Wrong	All	William
5.3 What We Learned	All	William
References	All	William

Table of Contents

List of Figures

List of Tables

List of Code Listings

1. Introduction

Maze of Rebirth is a VR game in which the player must escape a multi-floor maze filled with yokai (creatures from Japanese folklore), traps and items. The player must use the items at his disposal to navigate the maze, while also trying to kill dangerous enemies by luring them into traps.

We developed our game using Unity because another one of our goals was to support a wide variety of platforms. Our game was developed to work on both Oculus Rift and the HTC Vive by using Valve's SteamVR library. We were able to develop and test our game using the resources provided by Ritsumeikan University. The game could be ported to a variety of consoles with ease and while we put a strong focus on Virtual Reality, we have also included traditional control methods such as a keyboard and mouse, and an Xbox 360 Controller.

In *Maze of Rebirth*, an individual plays as a deceased person trying to navigate the maze of the underworld so that, upon completion, she can be reborn. All she has at her disposal is a compass, some chalk that can be used to draw on the walls, and talismans which can stun enemies. Using the compass to gain a sense of direction, using the chalk to mark where she has been so she knows where not to go, and uses the talismans to save herself from dangerous situations, the player cautiously makes her way through the maze in hopes of reaching the end.

A challenge we faced in creating this game was how we were going to handle VR sickness in our game. Movement was extremely important as our game was set in a maze in which the player must flee enemies all whilst in first person. With such a focus on running, we could not use the popular Virtual Reality movement schemes such as dashing or teleporting. After testing a variety of games with Virtual Reality support, we decided to emulate one of Minecraft VR's movement schemes (segmented camera turning) in order to minimize VR sickness. However, during testing, many players told us that they preferred when the game did not emulate that particular movement scheme, so we changed the movement back to normal.

Other challenges we faced included designing the game in such a way that it allowed for lots of re-playability, as well as providing an uneasy atmosphere that fills the players with nervousness and perhaps even dread. To solve the re-playability problem, we decided to make the game randomly-generated, so that players could come back and replay the game as many times as they want, getting a new experience every time. For the overall atmosphere of the game, we did research into various creatures from Japanese folklore, also known as yokai, and based the enemies off of those. We chose assets that were especially creepy-looking to represent these monsters, so that players would feel frightened, or at least startled, whenever they ran into one. Randomizing the lighting, while not letting it get too bright, also helped reinforce the creepy atmosphere we were aiming for.

Yet another major hurdle in designing our game was balancing out the levels so that they could all be beatable. Since enemies were un-killable without traps, it was very possible for a level to be loaded where enemy placements were such that making it through the level would be almost impossible. To overcome this, we added the stun talismans, so that players could survive through otherwise-impossible situations, thus giving them the time necessary for them to become accustomed to the game. Criteria were also added to the enemy spawning to ensure that they did not spawn too close to the player's starting position.

This report delves into the specifics of *Maze of Rebirth*'s development, as well as the methods we used for testing our game, and what the results of that testing were. More specifically, Chapter 2 focuses on the design choices of the game, as well as the overall gameplay and mechanics. Chapter 3 looks at the games aesthetics, both auditorily and visually. Chapter 4 is where the paper goes into the most depth; exploring the actual technical side of how we got every element of the game to work, from a programming perspective. The fifth chapter looks at the methods we used for testing, and the results of such testing, and the sixth and final chapter looks at what our thoughts on the project are, looking back at it.

In order to give a good idea of what our game is like, it is necessary to give examples of how our game is played. Since the levels are generated for each playthrough, giving an exact representation of what an example user will experience is difficult, as it will depend on what randomly-generated maze her or she gets placed in. As such we instead explain the general process of an example user interacting with our game before we give a more concrete example.

A large part of the player's experience will hinge on whether the user wishes to play the game in Virtual Reality(VR). Once that decision is made and the real-world components are properly setup, the user may then open our game. Once there, the user may do one of three things, quit the game, enter the settings menu to adjust the control scheme, or enter the play menu. Inside of the play menu the user has three options to adjust prior to beginning the main game. The first option is to adjust the seed in order to make different maps spawn. The second option is the difficulty level which will determine how large the maze is and what the user will encounter within the maze. The last option is whether the user wishes to enable the tutorial or not, which is described in depth in Section 2.4. Once into the game proper, the user will start in a relatively safe area due to no enemies or traps being able to spawn in adjacent maze units. The user must then navigate the maze in order to find the ladder to progress to the next floor. The user will quickly encounter enemies and traps and have to evade them or make good use of the items the user started with or acquired from within the maze. The user is likely to die before he or she completes the maze he or she began with. Once dead the user may return to the play menu and either retry the same maze using the same combination of seed and difficulty or he or she may try another maze by changing either or both of those options. Eventually the player will

complete a maze leaving them with the option of either trying other mazes or trying to complete the same maze again but faster.

Inside the game itself there are two primary considerations for the user at any given time. Those two being which direction does she decide to go and should she use an item. These decisions will have to take into account the game world around them. Factors that would affect these decisions include but are not limited to: which directions have walls prohibiting movement, what currently occupies the maze units in any given direction, and the current status of the player's inventory.

Knowing how the game plays, we can now create a more precise example of a user playing our game. The player would start on the tutorial and figure out how to move. The player then proceeds through the four tutorial floors learning how to deal with enemies and traps as well as learning how to use items along the way. From here the player enters into the main game where additional enemy and trap types are added to what they have to deal with. A more detailed user story can be found in section 2.10.

2. Design and Gameplay

2.1 Experience Goal

The player's motivation in the game is to escape Yomi. The sense of being lost, the sound of enemies, and the knowledge that there could be an un-killable enemy nearby, all work to create a feeling of dread within the player. This feeling of despair and anxiety will give way to momentary bouts of relief as the player finishes levels, as well as a tremendous amount of relief when the maze is finally completed.

2.2 Background Research and Decisions

For our game, we conducted background research. In order to better create the enemy yokai and the general atmosphere we researched into traditional Japanese folklore. Amongst the sources we read were the *Kojiki* and the *Nihon Shoki*, which are the two oldest compilation of legends about the Kami (gods) and the trials that they faced as well as the history of Japan (Chamberlain 1982; Toneri 720). We included in our game multiple references to Yomi, the Japanese equivalent to Hades, as a place as well as encounters with yokai. In the instances in which Yomi is mentioned, there is never a distinct description of what exactly it is like. All that is mentioned is that it is a physical place where people go when they die, and that a person who eats at the hearth in Yomi is bound there forever, even if that person is still alive. This lack of physical description of Yomi's appearance allowed us to take the creative freedom to imagine Yomi as an underground maze. This helped us understand the unique traits of various Yokai, while also providing us with a large variety of possible enemies to choose from.

Additionally, we obtained general descriptions as well as drawings of yokai from the website Yokai.com (Meyer 2013). We also watched Japanese horror stories such as *Yami Shibai* and *Kakurenbo* that gave us references to use for Japanese style horror which focuses on a slow encroaching doom that the characters are usually powerless to defend themselves from. We used these as references because a large part of the horror we planned to inflict on players is derived from the same sense of helplessness focused on in those shows (Kumamoto 2013; Morita 2005). Finally, we made use of the existence of ofuda in order to give the player a fitting way to have some option to defend himself/herself. Ofuda are objects, typically made of paper or wood, that have words inscribed on them to charge them with the power of Kami. Ofuda are known for only having so much of charge before having to be replaced and we matched that with our ofuda being a temporary stun against enemies (D. 2011).

Another element of the game that required research was the incorporation of Virtual Reality. In an attempt to minimize the possibility of motion sickness for the players, research was done into various techniques that other VR games have used to prevent motion sickness. The

majority of sources all generally agreed on certain facts: traditionally user interfaces do not work, too much motion in peripheral vision is bad, and teleportation-based movement tends to work the best. We decided to go against teleportation, however, as that type of movement would quickly lead to players get lost in a maze as teleportation-based movement ruins spatial awareness (RealityShift 2016; Blain 2016; Riviere 2017; Mason 2017; Sebastian 2017). Additionally, while looking up examples of previous Virtual Reality games we encountered that Minecraft uses a staggered turn mechanic in an attempt to reduce simulation sickness, as a result we experimented with giving both smooth and staggered turning to the players of our game.

2.3 Gameplay

One of the main elements of gameplay is exploring the maze. Players must wander through various turns and intersections in order to find the floor's exit. Players can find chalk within the maze, which they can use to leave marks indicating where they have been and what they have done. Players start with a mirror which enables them to look around corners without alerting any potential enemies. Both players and the enemy yokai will leave behind footprints when they move a certain distance, which can be used by both parties to their advantage when navigating the maze. Players can also use a compass in order to keep track of their orientation within the maze. Lastly as players explores they can find pickups like chalk or ofuda (stun talismans) along the way.

Players cannot directly damage the yokai and as such must run away if engaged. The yokais' ability to follow the player's footprints will make escape difficult, but will also allow the player to lead the various yokai into traps that will kill them as shown in Figure 1. The player should be careful, however, as he or she can fall victim to the same traps as the enemies. If the player is caught off guard, or needs to buy some time from an enemy for one reason or another, then he or she can stun any enemy by throwing an ofuda, an enchanted object with "words of power" inscribed onto them, at the enemy.

[Figure 1: Example Gameplay]

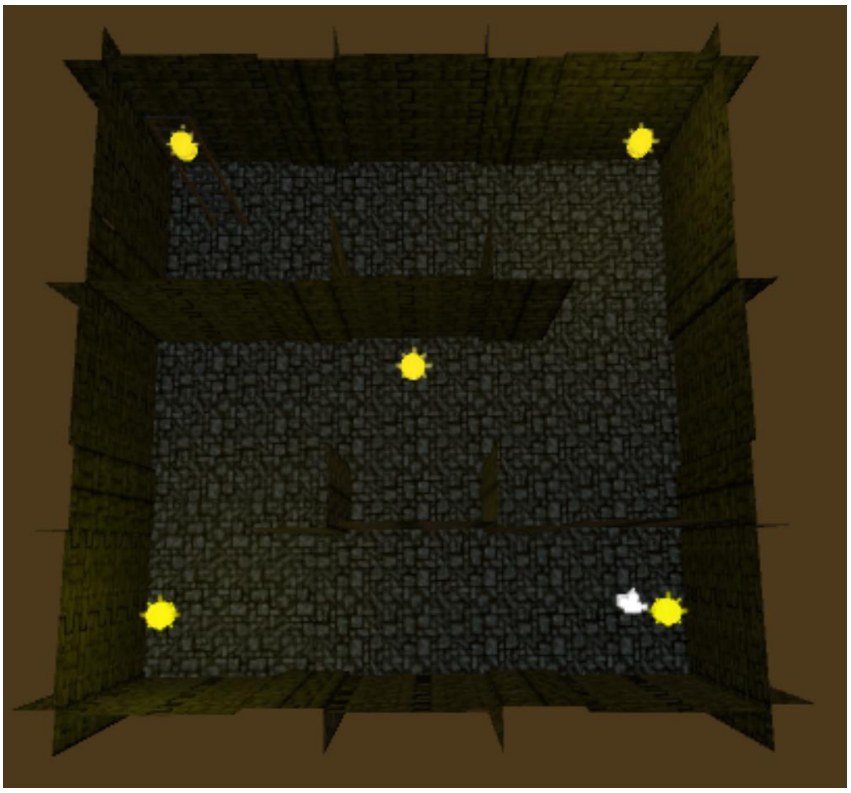
2.4 Tutorial Design

The tutorial section for any game is supposed to provide the player with a fundamental understanding of how the game plays and how the player can interact with the game. In order to create an effective tutorial, a game designer first needs to establish what the player needs to know. For our game the tutorial covered the follow lessons:

1. How to navigate the maze in order to find the objective that is the ladder at the end of the current section of the maze. This is fundamental underlying gameplay that the player needs to know before anything else.
2. Using chalk, as chalk helps the player figure out where he or she has been and what he or she has done which can reduce the amount of time lost.
3. Using the compass. The player can get disoriented easily in a maze so giving the player a compass and teaching him or her how to use it with ease the mental burden.
4. What are the enemies of the game. If the player doesn't know what he or she is encountering is an enemy he or she can respond in a manner which can cause him or her to lose.
5. The fact that traps exist and how to use them. Without the players knowing this, they will have no way to react to enemies beyond running away which will frustrate and annoy them.
6. How to use ofuda, as it enables another option against enemies.
7. Using the mirror. This is important because the mirror can be a useful tool to aid the player.

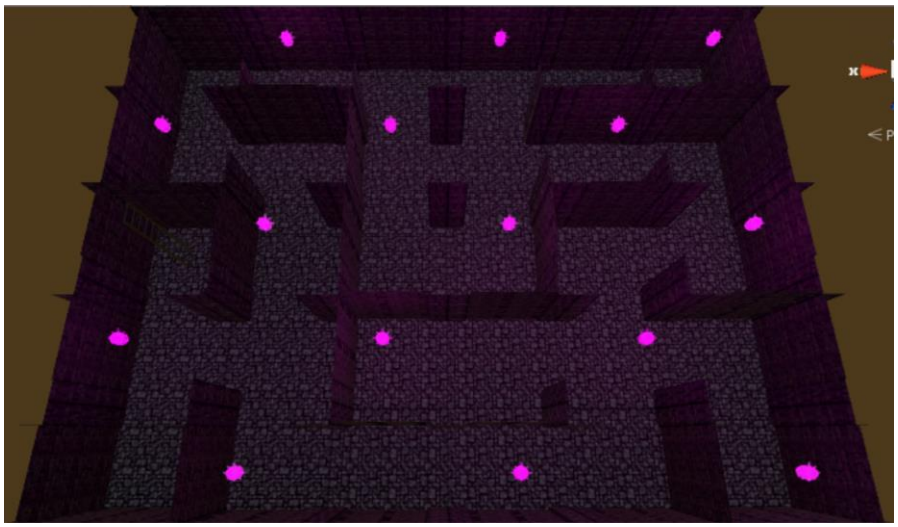
We endeavored to design the tutorial around teaching these concepts.

On the first tutorial floor the player starts facing a wall forcing them to turn to do anything, teaching them turning. Then the player needs to walk to the end of the hallway, which teaches the player how to move. The player then has to go around corners, which teaches the player that he or she will have to navigate down paths that are not straight. Lastly the player wanders until he or she finds the ladder, which teaches the player that the ladder is the goal. Figure 2 is a picture of the first tutorial floor showing that all of the above steps are indeed necessary to happen.



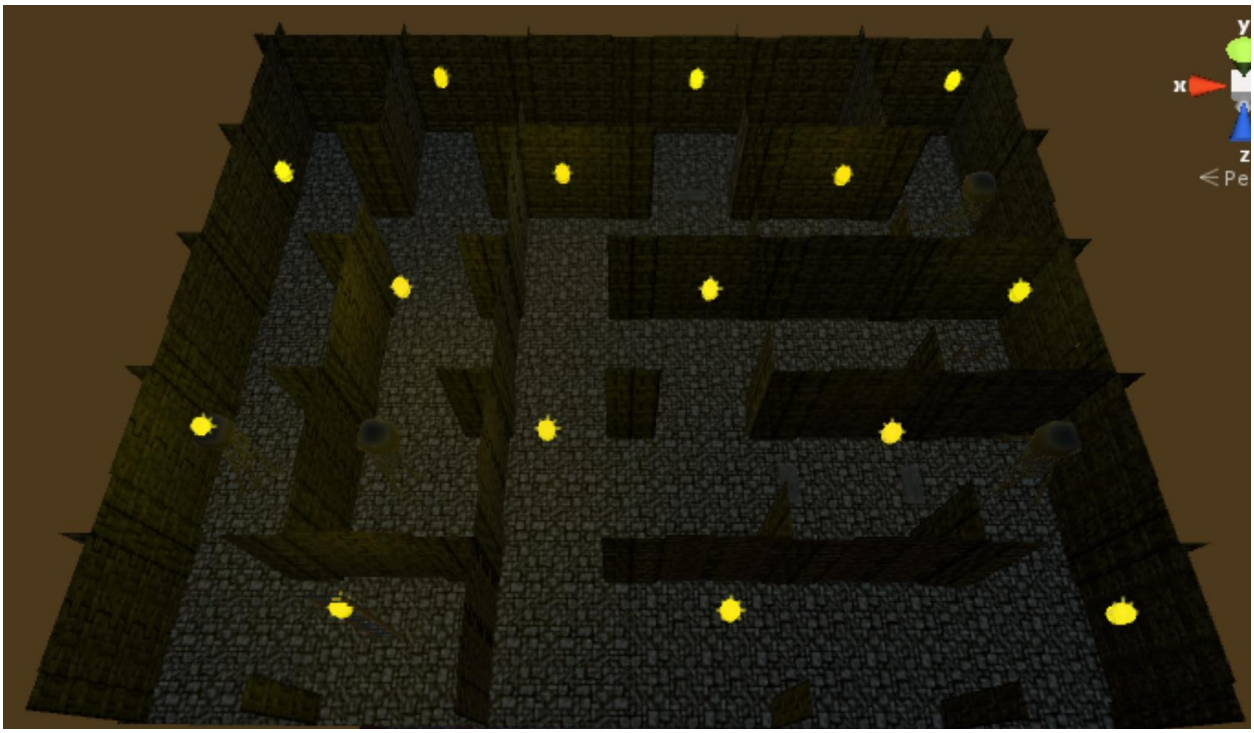
[Figure 2: An overhead picture of our first tutorial floor.]

On the second floor of the tutorial, the player encounters chalk. The player then navigates maze and can use the chalk where he or she wishes and eventually the player finds the ladder for that floor. Figure 3 is a picture of the second tutorial showing that this does indeed have multiple dead ends which enforces the concept of navigating a maze.



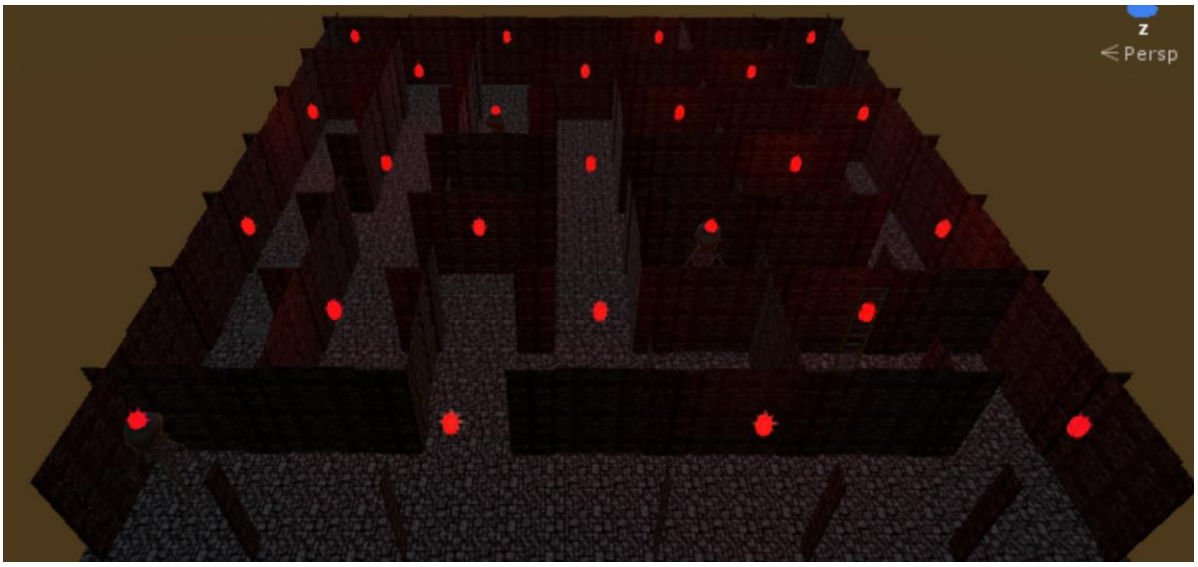
[Figure 3: An overhead picture of our second tutorial floor.]

On the third floor, the player turns a corner and sees an Oni. The Oni notices the player and moves toward the player only to fall into a trap and dies as a result. This teaches the player that the enemies exist, moves towards them, can walk into traps, and that traps can kill Oni. The player then advances further into the maze and sees it happen again. This reinforces the previous lesson. The player then goes around a corner and sees there is a trap behind them. The player keeps going forward until he or she encounters an Oni, the player can then lead the Oni back to the trap, which teaches the player that he or she can lure Oni into traps. Lastly the player finds the ladder. Figure 4 is a picture of the third tutorial floor and shows that Oni and traps are placed in such a way that forces the player to learn how to deal with them.



[Figure 4: An overhead picture of our third tutorial floor.]

On the fourth floor, the player wanders around and eventually finds an ofuda (stun talisman). Afterwards, the player runs into an Oni, which he or she can use the talisman against. The player can find more talismans and Oni as he or she explores the maze. Eventually the player proceeds and turns the correct corner, there is an Oni to the right and a new area to the left, which teaches the player the benefit of simply running away, from there the player will continue down the hallway to find the ladder. Figure 5 is a picture of the fourth tutorial floor and shows that the fourth floor is a combination of the previous floors to ensure that the player has learned the concepts they need to know.



[Figure 5: An overhead picture of our fourth tutorial floor.]

2.5 Level Design

The levels in *Maze of Rebirth* are randomly generated with specific criteria being implemented to ensure that certain elements are consistent across all levels.

These elements include:

- Splitting each floor into multiple sections
    - Building walls at certain points within the game to separate the floors
    - Number of sections per floor determined by difficulty
- Generating loops within each section
    - Deleting walls at specific locations creates multiple paths and loops
- The placement of
    - Items
        - Ofuda
        - Chalk
    - Traps
        - Spike trap
        - Pit trap
        - Crushing trap
    - Enemies
        - Oni
        - Taka-Nyudo
        - Ikari Inu
    - Ladders

Each of these elements will be explained in the Maze/Level Generation and Spawning section of the paper, also listed as section 3.4.

Each floor is divided into sections that are completely disconnected and are as equal in size as possible. This was done to force players to go to different floors in order to advance: going up and down between sections to eventually reach the end. This helps provide the game with the feeling of a "3-dimensional" maze, rather than just a collection of 2-dimensional mazes stacked on top of each other. The player can move between floors and sections using the ladders mentioned above.

Loops were added as a means of making the maze more difficult to navigate, so that even the earlier levels could pose somewhat of a navigation challenge. The game attempts to make the loops as big as possible so that their existence has as much impact on the gameplay as possible. The number of loops generated is dependent on the level of difficulty the player chooses to play on when he starts the game. The same goes for enemies, loops, sections, and so on. The higher the difficulty, the more there is of everything, from harmful enemies to helpful items.

Items, traps, and enemies are all placed randomly across the area, with nothing being spawned closer than three spaces apart. In addition to that, traps are never spawned at dead ends, as they would be useless in such locations. Furthermore, enemies, traps, and items are all in places that the player doesn't need to go to in order to reach the end of the section. This is done to ensure that enemies aren't spawned in an area that makes reaching the end of the maze section impossible. Due to this game taking part in a maze, even though it might be possible for players to reach the end of the maze without encountering enemies, the fact is that they most likely will. When that happens, the player will be forced to explore the maze for the traps needed to kill the enemies, as well as the other items in the game.

Lastly, ladders are generated in the farthest dead-end space from where the player spawns, meaning that no section will be too easy to complete.

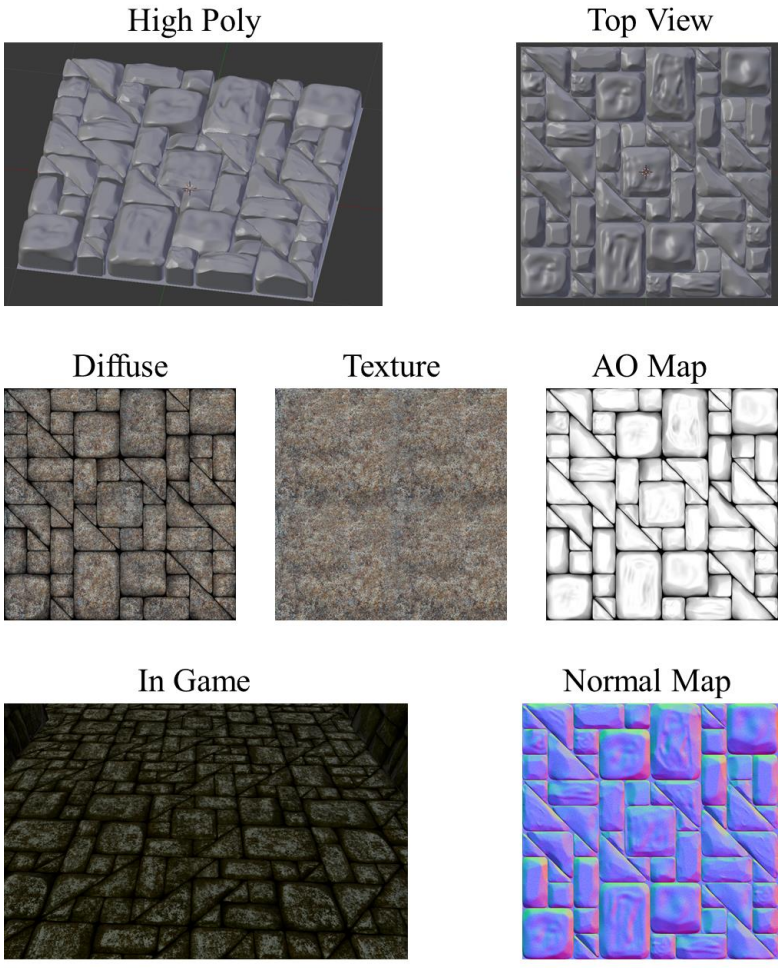
2.6 Environment Design

Maze of Rebirth takes place in the Shinto world of darkness, Yomi. We wanted the environment to reflect this darkness in order to maintain the creepy atmosphere of a horror game.

Since our game takes place in a maze, the environment assets were designed to be modular. The maze is made up of five cell types, dead end, hallway, corner, three-way intersection, and four-way intersection. Each cell is a six by six cube with a variable number of

walls and a pillar in each corner. Since the maze is made up of many cells, the polygon count for each cell had to be low.

The walls, floor, and pillar were made by baking high poly models into texture maps and combining them with textures made from photographs taken in Japan. To create the high poly models, first several bricks are modeled and placed in a pattern to fill up the area. Each brick is then sculpted to look worn and beaten up. The high poly mesh is then exported and used to generate a normal map, specular map, and ambient occlusion map. This allowed us to bake the detail of the high poly mesh into a plane (stym, 2014). In order to add additional detail to the texture maps, they were combined with textures created from photographs taken around Ritsumeikan. Photographs were layered together to create highly detailed stone textures (See Figure 6).

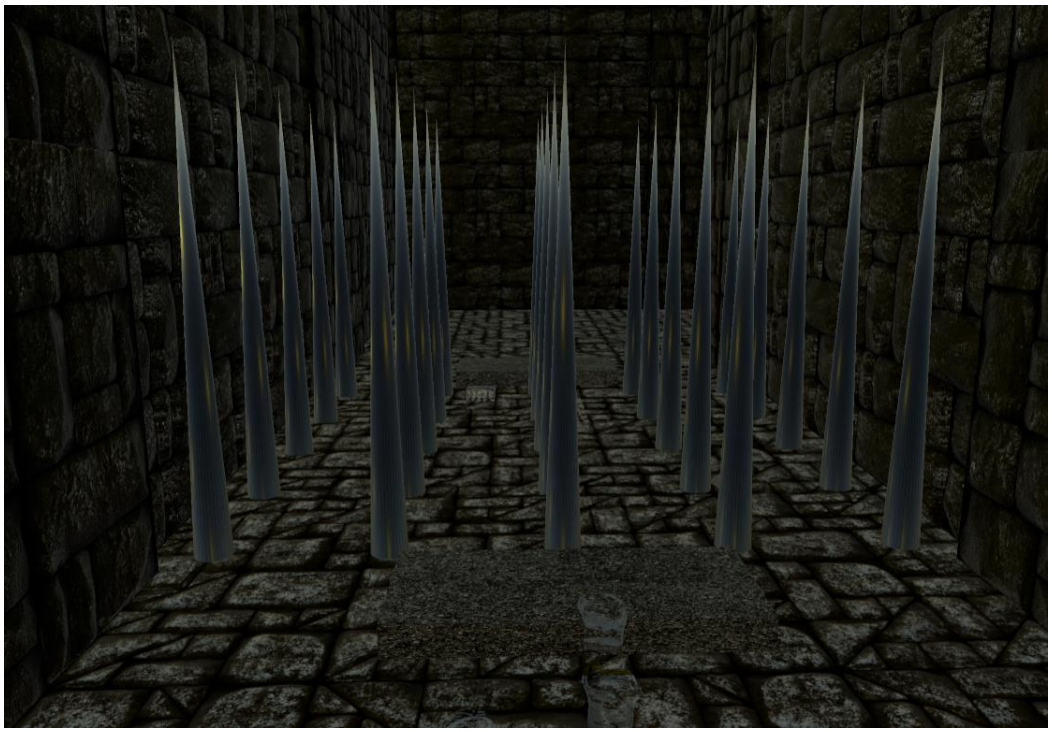


[Figure 6: High poly floor model, texture maps, and in game]

Lanterns are used to light up the maze. They can be found on the ceiling throughout the maze (See Figure 7). Each section has its own color for the lanterns, however the brightness throughout the section varies.

[Figure 7: Lantern in game]

Various traps can be found throughout the maze and are meant to help the player have a way to fight back against monsters. Both the spike trap and the floor trap use the spike model, which is made up of a five by five matrix of cones. For the spike trap, they are raised out of the ground (See Figure 8), while the floor trap has them at the bottom of a pit.

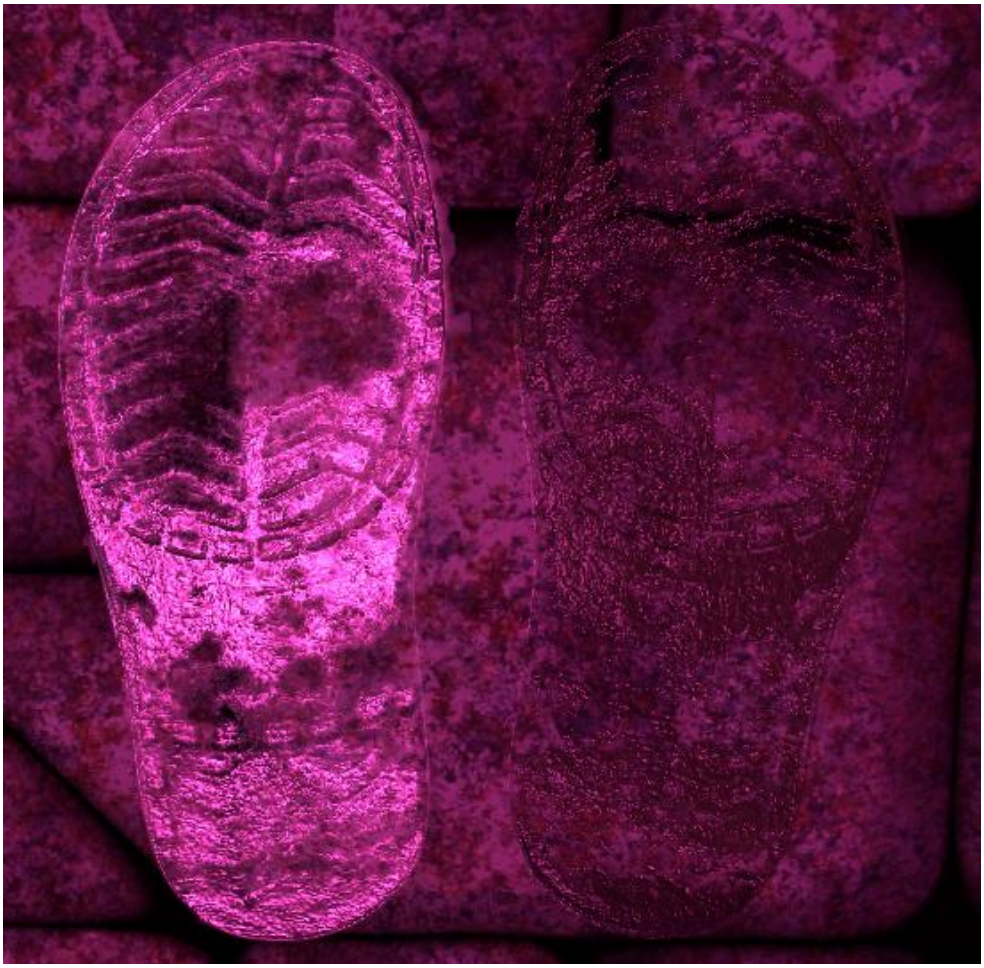


[Figure 8: Spike model in game]

In addition to traps, there are ladders in spread throughout the maze, which connect each section to the next. Two ladders models were needed as one is used to go up a floor, and one to go down a floor (See Figure 9).

[Figure 9: Down ladder in game]

Yet another element of the environment is footprints, which each character leaves behind as it walks through the maze. Footprints are made up of a diffuse map, normal map, and specular map. Each footprint uses the specular shader for increased visibility. Specular shading reflects light, this highlights the footprint much more than diffuse shading (See Figure 10).



[Figure 10: Specular Footprint (left), Diffuse Footprint (right)]

The player has two footprints made from a shoe. The shoe is cutout from the image and processed to look faded. Fading in different ways creates new variations of the footprint. Next, specular and normal maps must be created for each variation. The specular map is made by converting to greyscale and scaling the color towards white. In a specular map the color information is used to determine how reflective the texel(texture pixel) is. White represents completely reflective and black is not reflective at all. The normal map was then generated using a Gimp plugin on the diffuse texture. Figure 11 shows each individual map and the original picture used to create them (See Figure 11). The Taka Nyudo and Okuri Inu both have four variants footprint textures while the Oni only has three. The Taka Nyudo, Oni, and player have

different materials for their left and right feet. The left foot materials are the right foot textures mirrored on the X axis. All together this makes 22 footprint materials (See Figure 12).

[Figure 11: Original photograph (left), two variations of diffuse, specular, and normal maps of the players footprints]

[Figure 12: All 22 footprint materials]

2.7 Enemies

Enemies are creatures that patrol the maze, looking for the player. They are based on creatures from Japanese folklore, and aren't killable unless they fall into a trap. They are included in the game because, without them, there would be no way to lose the game, and the feeling of dread and tension that we want the player to experience as part of our game would disappear.

Oni

An Oni is a large, loud, slow monster that walks through the maze looking for the player. The Oni is the primary enemy of the game and is constantly trying to chase the player down to kill them. Figure 13 shows an illustration of how Oni are typically depicted in traditional Japanese works on the left and shows the model of an Oni in our game on the right. The model we used in our game is a placeholder that would be replaced should we receive the aid of a dedicated artist. We included the traditional art of an Oni in order to provide what the model in our game may eventually look like.

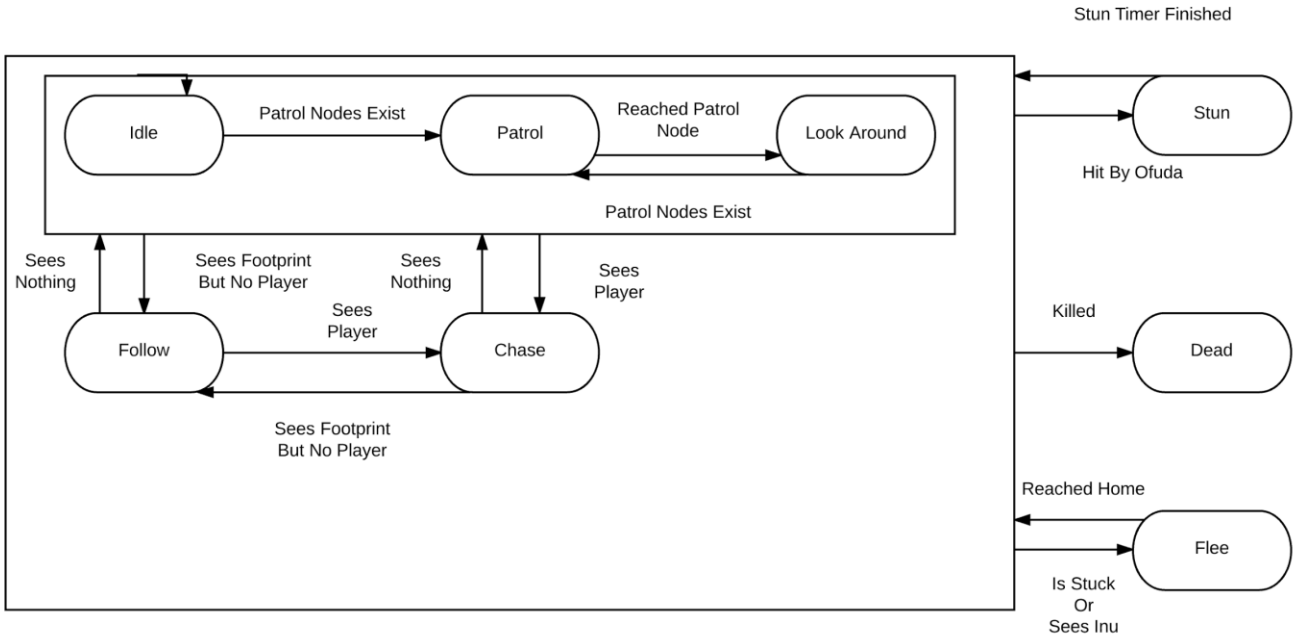
[Figure 13: (Left) Drawing of an Oni. (Meyer 2013), (Right) Model of an Oni in our game. (Mazza 2017)]

Oni Behavior

The Oni is capable of several different behavioral patterns:

- Idle
- Patrol
- Look Around
- Chase
- Flee
- Dead
- Follow
- Stun

In the idle behavior has the Oni do nothing and wait for stimulus to interact with the world around it. In the patrol behavior, the Oni wanders from intersection to intersection and keeps a lookout for the player, with the Oni obtaining a new destination once it is close enough to its current destination. In the look around behavior, the Oni has reached a patrol point and scans around for the player before moving on with its patrol. In the chase behavior, the Oni sees the player and moves directly towards them by setting their destination to be the player's current location. In the flee behavior the Oni has either encountered an Okuri Inu or has gotten stuck and returns to its home position by setting the location stored at spawn as its destination. In the dead behavior the Oni has died and no longer is active. In the follow behavior the Oni sees player footprints but no player and follows the footprints as a result. When the Oni has reached a small distance from the current footprint it is moving towards it then accesses said footprint's next field to determine where it should go next. In the stun behavior, the Oni has been hit by an ofuda and is stunned for a small-time period. If the Oni ever touches the player while in the chase behavior state then the player dies. The Oni can hear the player's footsteps and rotates towards them if the Oni is aware that a player exists. The Oni avoids colliding with other enemies during its patrol state thanks to its navigation A.I. Figure 14 is a state machine diagram for the Oni. The state machine is an illustration of how the states relate to each other and how the Oni can be expected to act. Further information about the technical implementation of the Oni is in Section 3.5.



[Figure 14: A representation of the state machine for Oni's A.I.]

Okuri Inu

The Okuri Inu is a dog type yokai that stalks the player and will attack when cornered or if it has followed the player long enough. It is said that prior to an Okuri Inu's attack, a person will hear the chirping of a Yosuzume, a bird type yokai that alerts people to the presence of an Okuri Inu. It never follows too close to the player and will try to avoid dead ends while not passing by the player. Figure 15 shows an illustration of how Okuri Inu are typically depicted in traditional Japanese works on the left and shows the model of an Okuri Inu in our game on the right. The model we used in our game is a placeholder that would be replaced should we receive the aid of a dedicated artist. We included the traditional art of an Okuri Inu in order to provide what the model in our game may eventually look like.

[Figure 15: (Left) Drawing of an Okuri Inu. (Meyer 2013), (Right) Model of an Okuri Inu in our game (3dhaupt 2015)]

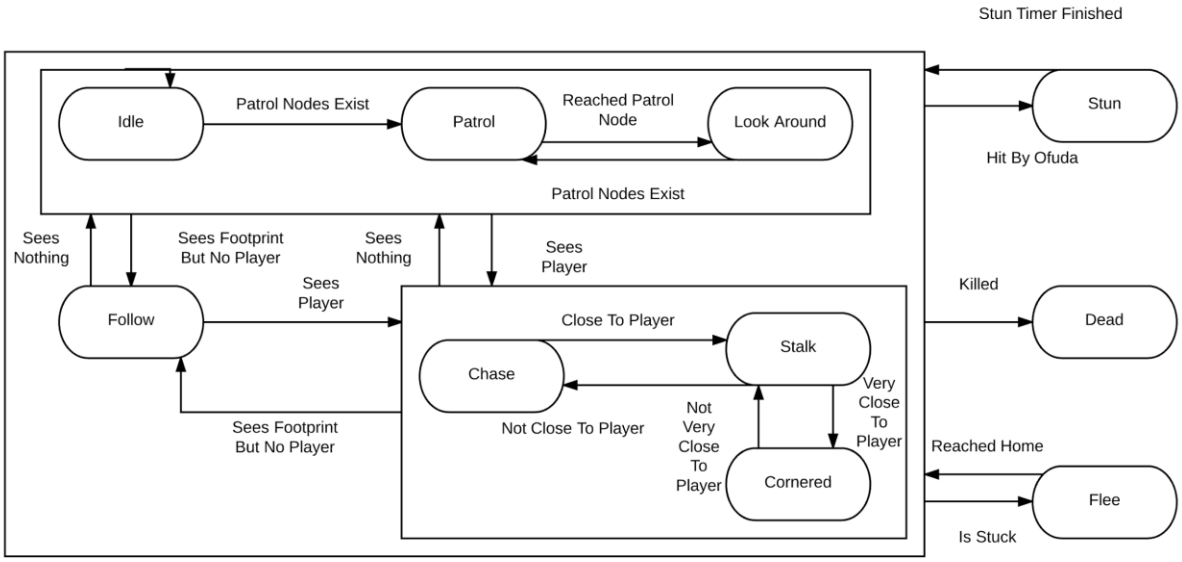
Okuri Inu Behavior

The Inu is capable of several behavioral patterns:

- Idle
- Patrol
- Look Around
- Chase
- Stalk
- Cornered
- Flee
- Dead
- Follow
- Stun

In the idle behavior the Inu does nothing and wait for stimulus to interact with the world. In the patrol behavior the Inu wanders from intersection to intersection and keeps a lookout for the player. In the look around behavior the Inu has reached a patrol point and scans around for the player before moving on with its patrol. In the chase behavior the Inu sees the player and

moves directly towards them until it has reached stalking distance. In the stalk behavior the Inu is close to the player and maintains a distance to follow behind the player until the player corners it or enough time has passed. While in stalk the Inu will move backward by calculating which maze node to move to according to the angle between the player and the Inu. In the cornered state, the Inu will play a growl sound to warn the player to back off and will kill the player if he or she further corners it. What's more, when the Inu is in the cornered state, the player has less time get away from the Inu. In the flee behavior the Inu has gotten stuck and returns to its home position to reset. In the dead behavior the Inu has died and no longer is active. In the follow behavior the Inu sees player footprints but no player and follows the footprints as a result. In the stun behavior the Inu has been hit by an ofuda and is stunned for a small-time period. If the Inu ever touches the player when attacking then the player dies. The Inu can hear the player's footsteps and rotates towards them if the Inu is aware that a player exists. The Inu avoids colliding with other enemies during its patrol state thanks to its navigation A.I. Further information about the technical implementation of the Okuri Inu is in Section 3.5 and there is a state machine for the Inu in figure 16.



[Figure 16: A representation of the state machine for Inu's A.I.]

Taka Nyudo

The Taka Nyudo is a giant type yokai, that the player must avoid looking up at so that it won't kill him. The Taka Nyudo is faster than the player when the player is not sprint moving. Figure 17 shows an illustration of how Taka Nyudo are typically depicted in traditional Japanese works on the left and shows the model of an Taka Nyudo in our game on the right. The model we used in our game is a placeholder that would be replaced should we receive the aid of a

dedicated artist. We included the traditional art of an Taka Nyudo in order to provide what the model in our game may eventually look like.

[Figure 17: (Left) Drawing of an Taka Nyudo. (Meyer 2013), (Right) Model of an Taka Nyudo in our game. (Melvinsalcedo169 2016)]

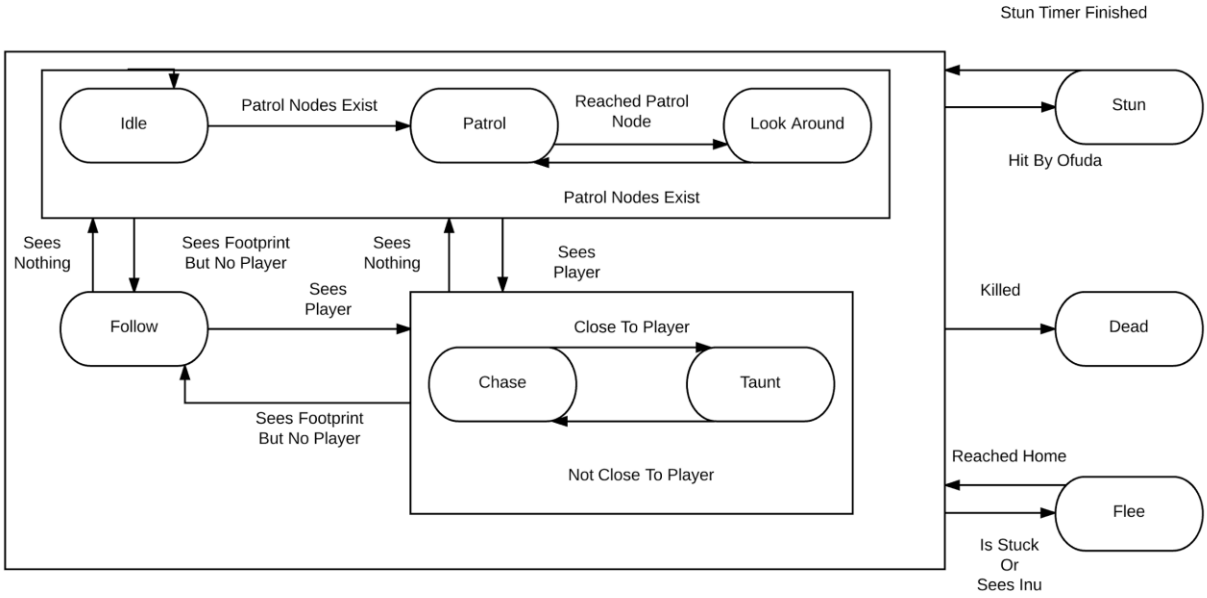
Taka Nyudo Behavior

The Taka is capable of several behavioral patterns:

- Idle
- Patrol
- Look Around
- Chase
- Taunt
- Flee
- Dead
- Follow
- Stun

In the idle behavior the Taka does nothing and wait for stimulus to interact with the world. In the patrol behavior the Taka wanders from intersection to intersection and keeps a lookout for the player. In the look around behavior the Taka has reached a patrol point and scans around for the player before moving on with its patrol. In the chase behavior the Taka sees the player and moves directly towards them. In the taunt behavior the Taka will maintain a distance from the player and spout insults and taunts at the player to try and get the player to look up, the Taka grows during this period of time, and if the player looks up during the Taka taunting the player will die. If the player lasts long enough without looking up at the taunting Taka, it will shrink and return to its home position. In the flee behavior the Taka has either encountered an Okuri Inu or gotten stuck and returns to its home position to reset. In the dead behavior the Taka has died and no longer is active. In the follow behavior the Taka sees player footprints but no player and follows the footprints as a result. In the stun behavior the Taka has been hit by an

ofuda and is stunned for a small-time period. The Taka can hear the player's footsteps and rotates towards them if the Taka is aware that a player exists. The Taka avoids colliding with other enemies during its patrol state thanks to its navigation A.I. Further information about the technical implementation of the Taka Nyudo is in Section 3.5 and there is a state machine for the Taka in figure 18.



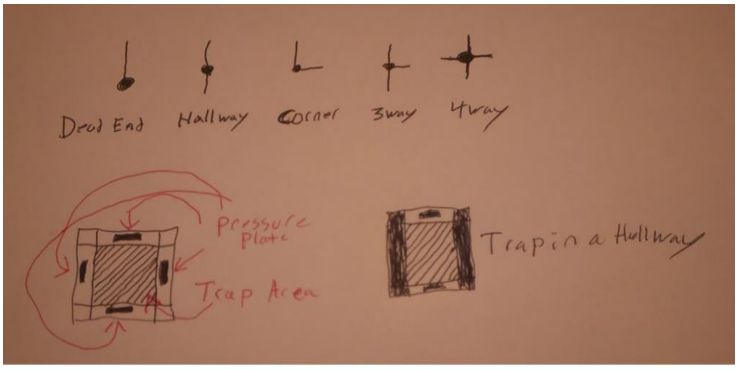
[Figure 18: A representation of the state machine for Taka's A.I.]

2.8 Traps

The traps were designed to fit into any of the maze cells. We have 5 variations of cells (See Figure 19). This required each trap to work with or without any of the four walls. To solve this problem, a pressure plate was added to each side of the trap. The pressure plate is located in the center of each side which gives the player room to carefully walk around it. However, the enemies are unable to easily navigate around the pressure plate, often resulting in their death. Depending on the cell type the trap is located in, some pressure plates may not be needed. In this case, the pressure plate is hidden inside the walls of the maze.

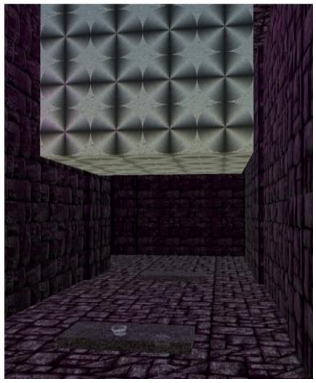
In our prototype we decided to limit ourselves to traps that did not require a wall. We had originally envisioned a dart trap. The dart trap would require a wall to shoot the darts from. This would require more strategic design to determine a good placement of the trap. Additionally, this would limit us to only supporting specific cell types. Ideally, in the future we would add additional trap variations that better fit into the environment and match the cell type. An example of this would be a crushing trap where two stones come out from the walls of a hallway.

Our final trap designs are the spike trap, crushing trap, and the floor trap. The spike trap operates by raising spikes out of the floor, impaling anything above. The crushing trap drops a large block on the floor. The floor trap opens to reveal a pit of spikes below.

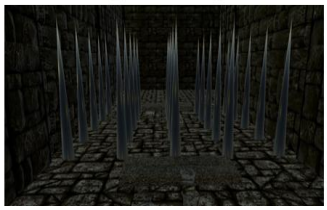


Floor Trap

Crushing Trap

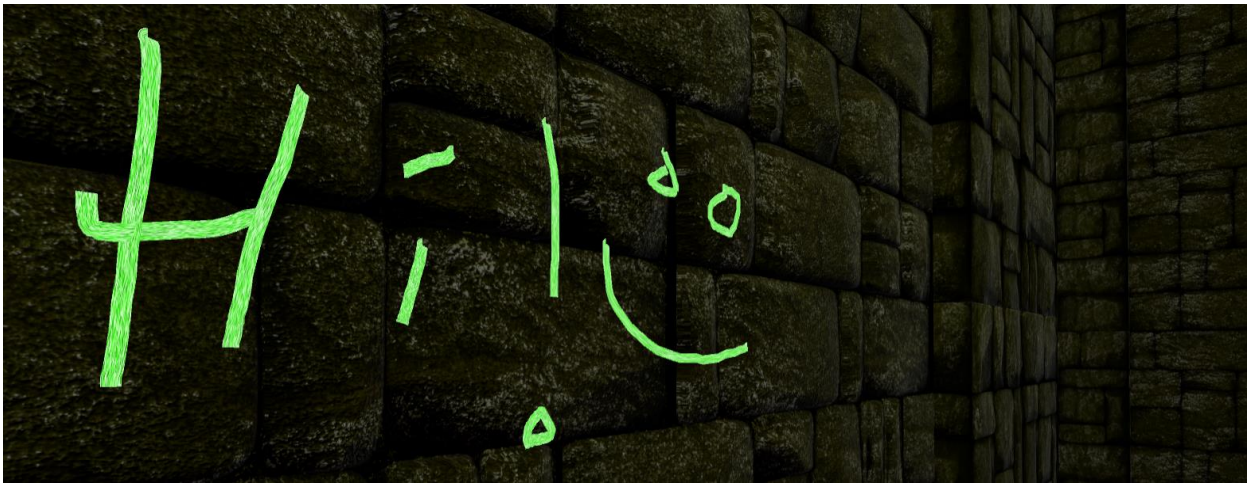


Spike Trap



[Figure 19: Pictures of traps in our game.]

2.9 Items



[Figure 20: Picture of a wall with a chalk drawing on it.]

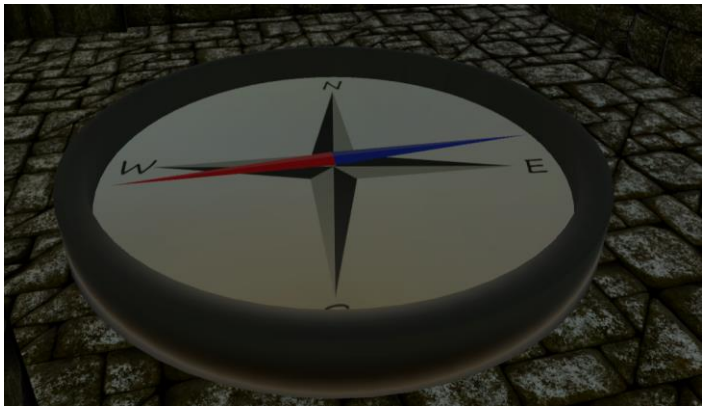
There are four items in our game the player may use to their advantage: chalk, the compass, the mirror, and the ofuda. Players can use chalk to draw on the environment. The player also has a compass to aid in navigation. The player has a mirror to cautiously navigate the maze. Lastly, the player has talismans called ofuda, that can be used in an emergency.

Chalk

Chalk can be used to draw on walls, the floor and even traps (See Figure 20). This allows the player to mark locations and recognize where they have been. The player has a limited supply of the chalk and can find more throughout the maze. Finding additional chalk is a reward for exploring the maze.

Compass

The compass can help the player to navigate the maze. It has a pin that always points North, which will aid in the player in keeping their orientation as he or she navigates the maze (See Figure 21).



[Figure 21: Traditional player's compass in game]

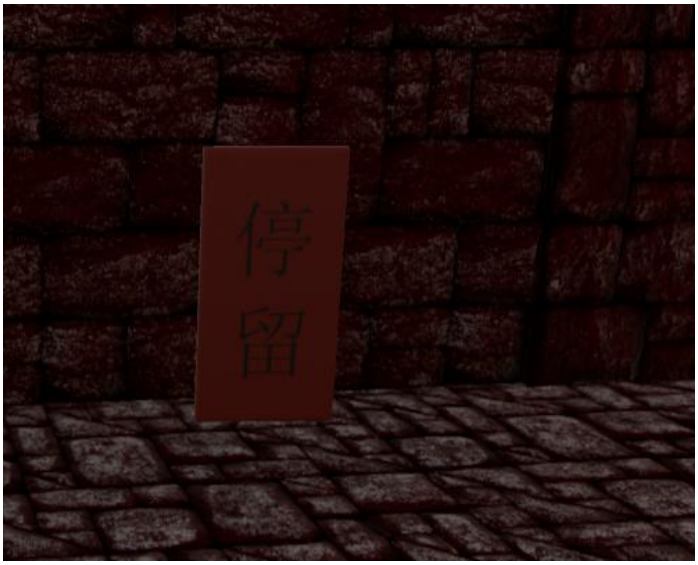
Mirror

The mirror can be used to look around corners and look behind the player (See Figure 22). This helps players to spot enemies before they are seen by an enemy. It allows the player to see if they are being chased, without turning around.

[Figure 22: Traditional player's mirror in game]

Ofuda

In the event the player was caught by an enemy and must escape, the player may throw an ofuda at the enemy to temporarily stun it. Additional ofuda can be found throughout the maze (See Figure 23). Ofuda are Shinto talismans, typically made out of wood or a card (D. 2011).



[Figure 23: Ofuda in game]

2.10 User Story

A man named John owns his own VR setup and wished to play a game that makes use of it. John after searching for such a game finds *Maze of Rebirth* online and downloads it (note our game is not available for download yet). John then proceeds to start our game. Once at the main menu John presses the play button which brings him to the play menu. Once there, John decides to take it easy for his first time playing, and as such he lowers the difficulty to easy and enables the tutorial before pressing play.

Now in the game John notices that he is in a hallway with a right turn up ahead. After some quick experimentation with the controllers John figures out how to move and turn and proceeds to walk through the first tutorial level. While walking John continues to experiment with the controls and discovers chalk drawing and ofuda throwing. John immediately wastes all of his chalk making pictures on the walls and throws all of his ofuda on the floor before picking the ofuda back up. Arriving at the end of the first tutorial level John spots a ladder which he walks into taking him onto the second floor of the tutorial.

Arriving on the second tutorial floor, John sees he can take a left turn or walk forward slightly and take a right turn. John takes the left turn and shortly finds himself in a four-way intersection. John turns left again and runs into a dead end. Turning around John checks the other two paths from the intersection to find they are also dead ends. John then returns the starting point of the second floor and takes the right turn. After some walking John sees that he can turn right or walk forward and turn left. John decides to turn right and finds the ladder to the third floor.

On the third floor, John sees that he can take a left turn and does so. As John makes the left turn, he sees something moving at him from down the corridor. As John realizes that it is an enemy, the Oni triggers a trap on the ground causing spikes to shoot out and kill the Oni in front of John. John, realizing that the pressure plates on the floor triggered the spike trap, proceeds to walk around the pressure plates. After continuing down the only path John turns right and has the same scenario repeat itself. This time though John has to choose between walking across the trap to where the Oni was or turning left and walking across a second adjacent trap. John decides to go left and sees that there is a one square space to the right of the second trap and a path to the left. Following the path John turns left and sees another Oni coming after him. There is not a trap between him and the Oni this time but John remembers the square he just passed. Turning around John crosses the trap a second time to reach the one safe space on the other side of it. Once there, John sees the Oni chasing him run into the trap John lead it into. Afterwards John continues down the path only to have to lead another Oni into a trap. Having done so John finally manages to find the ladder to the fourth floor.

Arriving on the fourth floor, John sees that he spawned in a four-way intersection. Turning around, John sees that on the path a spinning ofuda in the air. John decides to go get it

and walks into the ofuda picking it up. However, as John does so, he sees that an Oni is coming at him from his right. Rushing to react John throws the ofuda he just picked up at the Oni as well as a few more causing the Oni to get stunned. John sees this and proceeds to check behind the Oni to see it came from a dead end. John then decides to return to the starting area and take another path. This time John chooses the path leading right, which results in John finding a dead end with a chalk pick up. Now having chalk again, John returns to the starting area and indicates the two paths he went done. As John does this, the Oni that he ran into earlier finds John again and proceeds to chase him. Running away, John takes a right and then a left, leading him into a corridor where another Oni begins to chase him. After reaching the end of the corridor John finds the ladder and completes the tutorial.

Having finally arrived at the real game, John finds himself having to turn left to enter a hallway. Once there, John sees a new type of enemy (the Taka Nyudo) walking towards him from his left, causing John to run to his right. Following that path, John encounters an Okuri Inu, whom John stuns with his second to last ofuda as he runs past. After taking another left turn, John finds the ladder to the second floor.

Once on the second floor, John sees that there is an Inu coming at him from his front and a Taka walking towards him from his right. Before John can make a decision the situation changes. The Taka sees the Inu and proceeds to flee. John, seeing that Taka is running away, decides to follow after it and watches as the Taka walks into a dead end. This leaves a new path on the left for John to follow as the Inu chases him. After following the winding path John finds the stairs to the third floor.

Soon after reaching the third floor, John feels that he has gotten used to the game and rushes forward to see he can turn left or right. Continuing to rush, John decides to turn right and runs straight into an Oni and dies as a result. Feeling slightly frustrated at himself John returns to the play menu. Feeling confident that he knows how to play the game, John decides to not use the tutorial this time and retries the same maze he played before. Several attempts later, John has managed to complete the maze he started with. Having had fun, John decides to share the game with his friends and challenges them to try and complete the maze with fewer attempts then he made.

3. Technology

This section looks at the more technical aspects of how we actually implemented certain elements of our game. It delves into how we recorded play information into our database, how we implemented our game to utilize Virtual Reality, and the process by which levels were generated. This is also the section where we talk about how the various states of the enemy A.I. was implemented.

3.1 Software

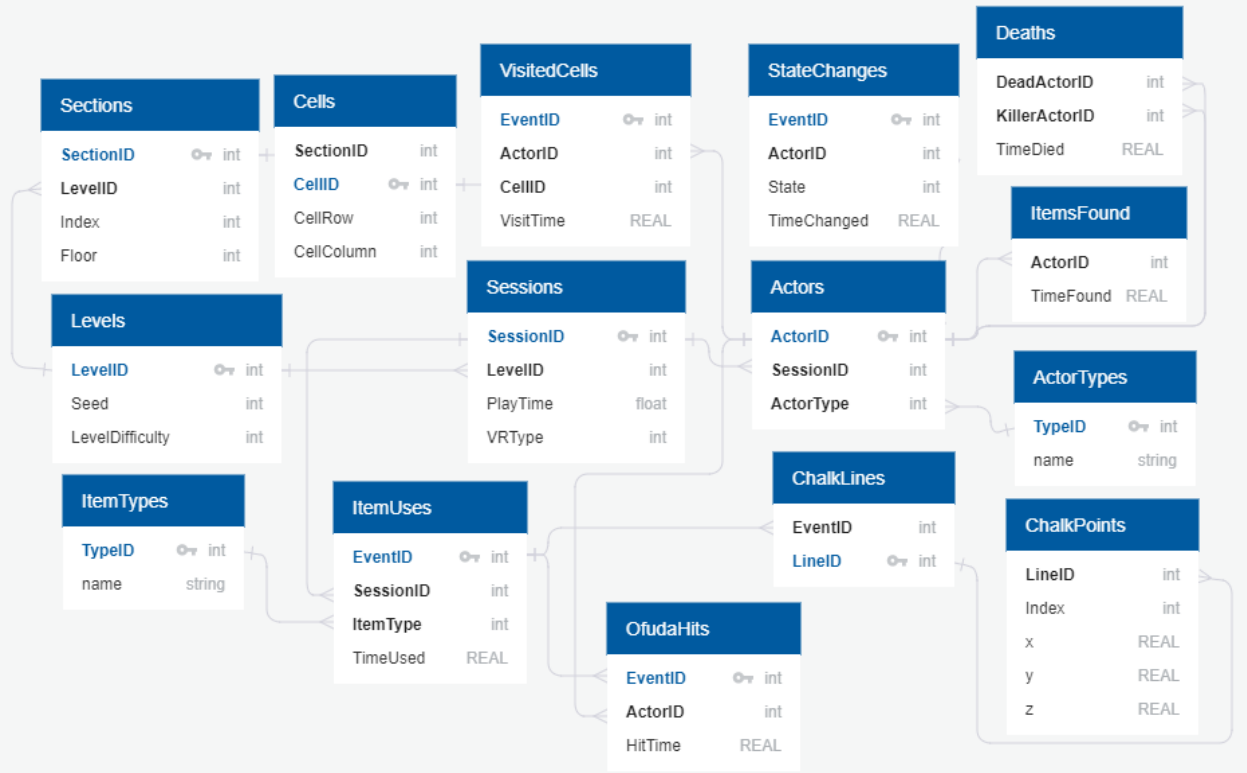
We developed our game using Unity 2017. We chose this engine for a variety of reasons. While Unity and Unreal Engine 4 both are commonly used throughout the industry and have comparable features, we decided to use Unity due to its extensive documentation and community support. Unity's support for C# allowed easy integration of the SQLite3 library and access to Microsoft's .NET API. Additionally, Unity provided support for both major competing virtual reality devices through a plugin on the asset store.

We used Git for the version control of our project. The repository for our project is hosted on Github at https://github.com/Zimhey/JapanActionGo. It is recommended to use the official gitignore for Unity as this will keep all unnecessary files out of the repository. Git allowed us to rapidly sync our work through our iterative design process. When using Git with a group it is important to communicate what you are working on in order to prevent merge conflicts.

On the art side, we used the free open source software Blender and Gimp. Blender was primarily used for creating models and UV mapping them. Each model was exported in the OBJ format as Blender's FBX exporting does not work well with Unity. Gimp was used for image editing, post processing and texture creation. A Gimp plugin called Normalmap was used to generate normal maps of the textures. XNormal was used to generate a variety of texture maps from 3d models.

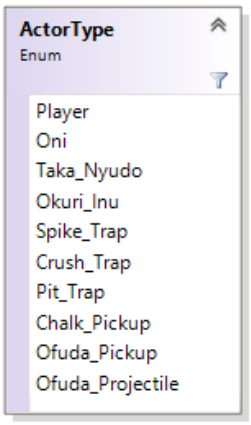
3.2 Telemetry

One of the major features of our game is to record most major events into a database which we then use later to determine if players are meeting our experience goals. We designed and developed a system to collect the important data, which could then be used to determine things about each play through, such as determining how lost a player was in the maze or how often they used their items and what they did with them. We developed a relational database schema, which contained important data for each type of event (See Figure 24).



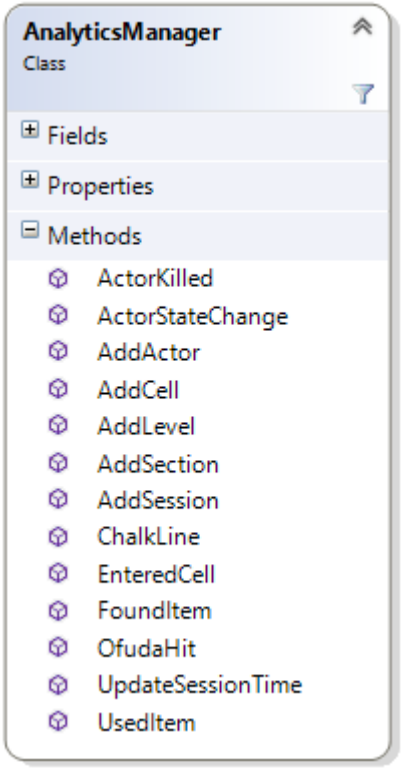
[Figure 24: Telemetry Database Schema]

The schema covers both level data and player session data. The level data is made up of a set of sections where each section has a set of cells. The play session data records each actor and their actions A list of actors recorded can be seen in figure 25. This includes which cells they visited and when, when they died and what killed them, and so on. Additionally, the state changes for actors that have Artificial Intelligence is recorded. The player's actions are recorded as well. Each item use generates an event ID which corresponds to things such as a collection of chalk lines for a drawing or when ofuda hit an enemy.



[Figure 25: Actor Types List]

The schema was implemented in SQLite3 using a C# library to interface with the database. A class called AnalyticsManager was created in order to provide an easy to use interface for recording events to the database (See Figure 26). It is accessed through our GameManager which handles all major game events. The AnalyticsManager abstracts the SQLite syntax to allow for ease of use. The AnalyticsManager divides queries into time sensitive and non-time sensitive queries. Time sensitive queries are run immediately as they require data back. An example of this is the item use query, as the EventID is used later when recording a collection of chalk lines. Queries that are not time sensitive are stored in a queue. A separate thread handles the **non-time sensitive** queries. The thread only runs the queries when the player is in the menu. This allows the **time sensitive** queries priority during game play sequences. In the event the query queue becomes too large, it is converted to a list and serialized to an xml file with the date and time. This prevents the database from getting bogged down between long play sessions.



[Figure 26: Analytics Manager Class Diagram]

In order to make the information in the database easier to use, multiple classes were created to help store the values in the database within various structures and objects. One such class was the GameSessionReports class, which extracts information from the entire database, and stores it within its fields. The second class was SessionReports, which was meant to contain the database information associated with a single playthrough. The GameSessionReports class

contains a list of SessionReports objects, and it fills in the fields of those objects by combing through the information contained within its own fields and using various IDs to figure out which entries are a part of the same game. Once the data has been separated by playthrough, we can more easily learn about individual players' action, and from that information, we can figure out patterns about what players tend to do as a whole. For instance, we can see if a player revisited the same area multiple times, and if there are no traps or items there, we could then extrapolate that perhaps the player got lost. Analyzing where they got lost in the maze could help us understand elements of maze design that cause players to get lost, which we could use to perhaps alter our maze generation algorithms to create more or less difficult mazes. We could also use data analyses to see what types of enemy placements tend to give the player the most trouble, and alter our level generation based on that. Unfortunately, this element of the game was never quite finished. We can organize the data and place it into separate classes, but we have yet to add code that actually analyzes this data to extrapolate trends and other information. In a future version of the game, we would like to have this feature fully-implemented so we can learn more through testing.

3.3 Virtual Reality

Virtual reality is an immersive technology that has long been anticipated by the video game industry. It allows a user to put on a headset and feel as though they are inside of a virtual world. This allows players to interact with the world in a new way not offered by traditional peripherals. There are several different virtual reality products currently available. These include the Oculus Rift (Figure 27), HTC Vive (Figure 28), and Google Cardboard (Figure 29). Oculus Rift and the HTC Vive support PC while Google Cardboard is made for mobile android devices. Our primary focus was on supporting Microsoft Windows and thus we decided to work with the Oculus Rift and HTC Vive. Both peripherals come with head mounted displays (HMD) and a pair of controllers. The HTC Vive features touch pads on their controllers in addition to a few buttons. The Oculus Rift went the traditional route and features analog sticks on each remote. Both devices now support room scale which allows the system to track where a user is in physical space. Room scale allows for additional immersion by translating physical movement into virtual movement.

Both the Oculus Rift and HTC Vive support Unity through their own software libraries; however, this would require us to abstract both libraries in order to support both. Rather than creating our own abstractions, we decided to use SteamVR, which abstracts both devices for us. This does come with its own set of problems as the HTC Vive does not have analog sticks. SteamVR fixes this by emulating an analog stick using the Vive's touchpads. While this is functional, it does not provide the same feedback as an analog stick and may feel weird to the user at first. SteamVR supports Unity through a plugin on the Asset Store. It provides a developer with the basic functionality required to develop for both devices.

[Figure 27: Oculus Rift, 1. Headset, 2. Touch controllers, 3. Sensors. (Oculus Rift, 2018)]

[Figure 28: HTC Vive Headset (center), Tower Sensors (upper left and right), Hand Controllers (lower left and right). (VIVE™, 2018)]

[Figure 29: Google Cardboard headset. (Google Cardboard, 2018)]

Locomotion is a difficult problem to overcome when developing for virtual reality. One of the major difficulties when working with virtual reality is simulation sickness. Simulation sickness is a type of motion sickness that occurs from the use of virtual reality. "VR Sickness is partly due to the real-world user's body being stationary while their virtual point-of-view moves around a virtual environment" (Unity, 2018). Room scale helps to solve this problem by tracking the player's physical position in the room and replicating it in the virtual space. This helps reduce simulation sickness as player's virtual body moves in sync with their physical one, however the virtual space then becomes limited by the size of the physical space. A common approach to navigating large 3-dimensional spaces in first-person is to use teleporting or dashing. Teleporting is often accompanied by vision fading which prevents the user from detecting discrepancy between the virtual movement and physical movement. While these locomotion schemes work, they do not fit into our experience goal well. One of our core mechanics is fleeing enemies in the maze. Teleporting away defeats the purpose of trying to escape the enemies. To accomplish our goal, we combined both traditional first-person movement with virtual reality's locomotion.

[Figure 30: VR Player Controller objects]

The implementation of the virtual reality player controller needs to combine both room scale movement with traditional analog stick movement. A traditional player controller uses a CharacterController component to move the player's character as it prevents them from walking through objects. The CharacterController's move method takes a vector that it will use to calculate its next position. The traditional movement method calculates a movement vector based on the position of the left analog stick (See Code Listing 1).

```
Vector2 input = getAnalog();
Vector3 move = transform.forward * input.y + transform.right * input.x;
```

[Code Listing 1: Analog Movement]

SteamVR provides the developer with a camera rig object with a head object and two controller objects (See Figure 30). These objects move relative to the camera rig. We can get the player's room scale movement vector by tracking the head's change in position relative to the camera rig (See Code Listing 2).

```
//The change in position is the last position - the new position
Vector3 deltaOffset = cameraOffset - (rigPosition - cameraPosition);
//Set the last position to the new position
cameraOffset = rigPosition - cameraPosition;
```

[Code Listing 2: Roomscale Movement]

The character can then be moved using both inputs while preventing the player from sticking their head through a wall (See Code Listing 3).

```
controler.Move(move * Time.fixedDeltaTime + deltaOffset);
```

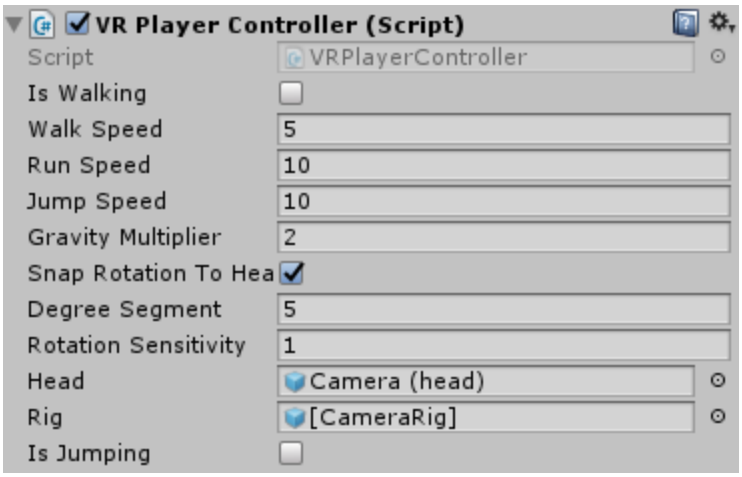
[Code Listing 3: Combined Movement]

Lastly, we need the HMD to follow the character's position so that the player sees what the character should. This is accomplished by moving the camera rig around the character object (See Code Listing 4).

```
//camera rig's position = character position + relative camera position
Rig.transform.position = transform.position + cameraOffset -
                         new Vector3(0, controller.height / 2F);
```

[Code Listing 4: Updating Camera Rig position]

A big part of the discomfort, according to Rubin, comes from movement in our peripheral vision. He pointed to Ubisoft's Eagle Flight as a game that's solving it by blacking out the periphery, which he calls 'tunneling' (PC Gamer 2016). One solution we found to this problem is in the virtual reality version of Minecraft snaps the player's rotation to 45-degree segments. This makes the rotation feel more like warping and prevents rapid changes in the peripheral vision. The level of VR sickness varies from person to person and as such we included the ability to change the number of degrees each segment covers and many other movement properties (See Figure 31). The outcome of giving the players the option of segmented turning is discussed in the beginning of the Results section.



[Figure 31: VR Player Controller Script]

Virtual reality allows us to interact with the environment in a more immersive manner. SteamVR's room scale tracking allows us to see the position and orientation of each controller. It also provides a pointer component which allows the player see what they are pointing at. In our game, one of the primary player actions is their ability to draw on the walls. Using the left-hand controller, the player may point and pull the trigger to draw a line on most surfaces. This method feels more natural than the traditional method where the player's look direction determines where the line will be drawn. Similarly, the right hand can be used to point in the direction the player wishes to throw an ofuda. The player is also given both a mirror and a compass which are attached to the controllers. This allows the player manipulate the mirror's position and orientation to see around corners by moving the controller. This solution works more elegantly than the traditional method which requires the player to press a button to show the mirror in front of them.

3.4 Maze/Level Generation & Spawning

One of the most notable aspects of our game is the way that the levels are randomly generated. We do this using depth-first search to create a "perfect" maze: a maze with only one path from beginning to end, where any point in the maze can be reached from any other point in the maze. This method is also known as recursive backtracking (Pullen 2015). An alternative algorithm we looked into using involved taking a square and dividing it into four sections of various size based on random number generation. This is also known as recursive division (Pullen 2015). After connecting the sections at random locations, once again using number generation, the process is repeated on the four sections of the original square. This process is repeated until there is a perfect maze. We chose to use to use the depth-first search method of random maze generation because the mazes generated tended to have more variety. After generating multiple mazes with this algorithm, the mazes are then split into separate sections,

which are connected to sections on other floors. After that, each section is filled with enemies, traps, and items.

Each level is generated according to several parameters:

- Number of rows
- Number of columns
- Number of floors the maze has
- Number of sections that each floor will be divided into
- Number of loops that will be in each section
- A seed that is used so mazes can be recreated when necessary.

Original Maze

The maze generator uses depth-first-search to create a random maze, the size of which is determined by the various parameters given to it.

The pseudo-code below represents how we used depth-first search to actively generate a perfect maze.

```
ConnectNodes(list(MazeNode) nodes) {
	//start out with a grid of unconnected nodes, all marked as “unused”
	//each node has an x and y value
	Node currentNode = node with x = 0 and y = 0;
	Stack(Node) prevNodes;
	while(there are still “unused” nodes) {
		Mark currentNode as used;
		if(all nodes adjacent to currentNode are marked as “used”)
			currentNode = pop(prevNodes);
		Else {
			push currentNode to prevNodes;
			Choose random node n adjacent to current node that is unused;
			Connect currentNode to n;
			currentNode = n;
		}
	}
}
```

[Code Listing 5: Pseudo-code for perfect maze generation]

In the code above, it can be seen that we used the “unused” and “used” marker to determine whether or not there were still nodes that needed to be connected, and we used a Stack to store the nodes that had been visited. This way, we could backtrack through previously-searched nodes in case the maze-generation process hit a dead-end. We have both a Stack and a field that informs us of whether or not a node has been connected because, when a programmer uses pop() to pop something off of a Stack, it is removed from said stack.

Exit Nodes

After creating the maze, depth-first-search is used yet again to find the path from the start of the maze to the end of it, with each node along said path being marked as an ExitNode as shown in Code Listing 6.

```
SetExitNodes(MazeNode root) {
	//all the nodes are connected to at least one other node, if a node isn't
connected
	//to an adjacent node, that means there is a wall between the two nodes
	//endNode is the node in the opposite corner of the startNode
	MazeNode current = root
	Stack(List(MazeNode)) PossiblePathsSoFar;
	List(MazeNode) startPath;
	Push startPath to PossiblePathsSoFar;
	Stack(MazeNode) nodes;
	while(current != endNode) {
		List(MazeNode) p = pop(PossiblePathsSoFar);
		Push current to p;
		For each (node n connected to current) {
			push n to nodes;
			Push p to PossiblePathsSoFar;
		}
		Current = pop(nodes);
	}
	List(MazeNode) p = pop(PossiblePathsSoFar);
	Push current to p;
	For each(MazeNode n in p)
		n.exitNode = true;
}
```

[Code Listing 6: Pseudo-code for finding the path through a maze]

The Stack of paths (which are just lists of nodes) allows us to record every possible path, ensuring that we will eventually reach the endpoint of the maze. Note how we also store the same path multiple times. This is because, for every node that is checked, there is a specific path that has led to this node, and there are certain nodes for which this path is the same. Because a path is removed from the Stack every time it's checked, we thus need duplicates of each path.

Node Interconnectedness

After generating the maze and assigning ExitNodes, the maze is then iterated through yet again, this time using breadth-first search, to calculate what we call the "connected node values." To understand what those values are, imagine a space in a maze that is connected to three other

spaces: one in front, one to the left, and one to the right. If there are 30 spaces that can be reached by going through the space in front of the current space, then the forward "connected node value" is 30. Similarly, if there are 33 spaces that can be reached by first going to the left, and 36 spaces that can be reached by going to the right, and the left and right "connected node values" would be 33 and 36, respectively. Since the space we are talking about in this example is not connected to the space behind it, then the backwards "connected node value" is zero. These values are calculated for every 1x1 space on any given floor, and are used to figure out how large an individual section would be if a wall were placed between two spaces.

```
SetConnectingValues(MazeNode root) {
	MazeNode current = root;
	while(!(every connecting value is set for every node)) {
		For each(node n2 adjacent to current) {
			If (node current is not connected to n2) { [2]
				n1.connectedNodesX = 0;
				Break;
				//in this case, X is either 1, 2, 3, or 4, depending on
				//the direction of adjacent node being checked
			}

			if(node n2's values are not set) {
				Current = n2; [1]
				break;
			}

			Else
				n1.connectedNodesX = n2.connectedNodes!X1 +
				   n2.connectedNodes!X2 + n2.connectedNodes!X3 + 1
		}
	}
}
```

[Code Listing 7: Pseudo-code for connecting nodes]

As shown in Code Listing 7, the way that the "current" node is set to the node n2 if n2's values have not been set [1], but only after checking if n2 is connected to the "current" node [2] ensures that dead ends will be evaluated first, allowing the nodes connected to dead ends to be evaluated next, and so on.

Sections

After setting the connected values, the maze generator then iterates through the ExitNodes, and finds the perfect place to decide where to divide the map into sections such that each section is close to equal in size, using the "connected node values" mentioned earlier. The number of sections that the maze is divided into is also a parameter given to the maze generation algorithm.

```
SeparateSections(MazeNode root) {
	Int nodesCutOff =  0
	// totalSections is how many sections there will be,
	// determined by difficutly
	Int remainingCells = size //# of nodes, by difficulty
	while(remainingCells != 0) {
		foreach(MazeNode n that is an “ExitNode”, starting from root) {
			nodesCutOff = n.connectedValuesX
			// X is based on the direction of the adjacent ExitNode
			If (remainingCells - nodesCutOff  =
			(approx)  size/totalSections) {
				remainingCells = nodesCutOff;
			}
		}
	}
}
```

[Code Listing 8: Pseudo-code for dividing the maze into sections]

As shown in Code Listing 8, the function takes the number of interconnected nodes and subtracts the number of Nodes that are being separated to determine the size of the section being created (see section of the above pseudocode that shows “remainingCells - nodesCutOff”). To determine if this section is the right size it is compared to the value of the size of each floor, divided by the number of sections on each floor. This is all done to ensure that each section created is as close to being equal as possible.

Loops

After the maze is split into sections, each section has loops generated in it. The maze generator finds two nodes that aren’t connected, but could be, and finds out how large of a loop would be created by connecting them. The largest potential loops are then created by connecting the necessary nodes. These loops make navigating the maze difficult and increases the chances of a player getting lost. The number of loops created is also determined by a parameter input into the initial algorithm. Code Listing 9 consists of pseudo-code describing this process.

```
AddLoops(MazeNode root) {
	List(MazeNode) node1s;
	List(MazeNode) node2s;
	List(MazeNode) largestLoops;
	// the size of these arrays is determined
	// by how many loops there will be in
	// a section, determined by the difficulty
	foreach(pair of adjacent nodes (n1, n2) not connected) {
		X = distance between n1 and n2
		if(x is bigger than the smallest value in largestLoops) {
			Int j = index of smallest value in largestLoops;
			largestLoops[j] = x;
			Node1s[j] = n1;
			Node2s[j] = n2;
		}
	}
	for(int i = 0; i++; i < largestLoops.length)
		Form connection between node1s[i] and node2s[i];
}
```

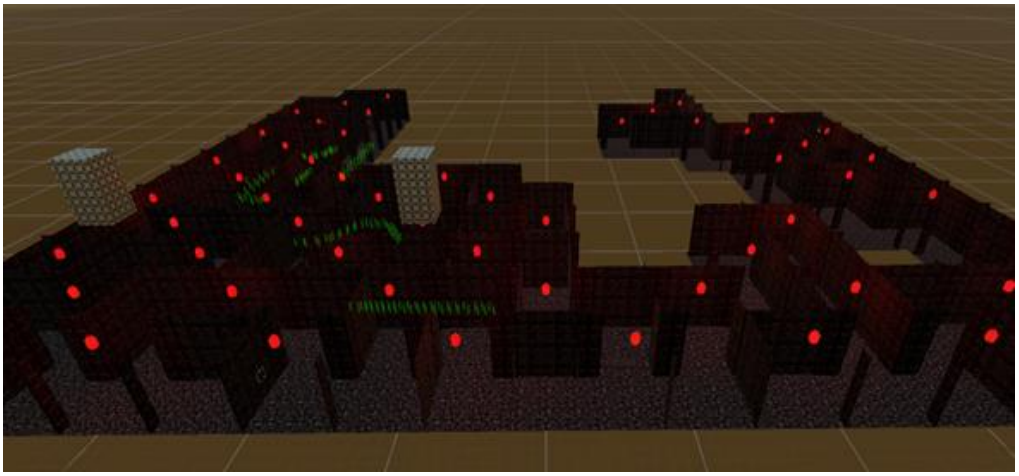
[Code Listing 9: Pseudo-code for generating loops]

Comparing the size of the loop currently being checked to the smallest value in largestLoops ensures that the values stored in largestLoops are all the largest loops possible, as opposed to having one loop that is the biggest, and then others that are not as big.

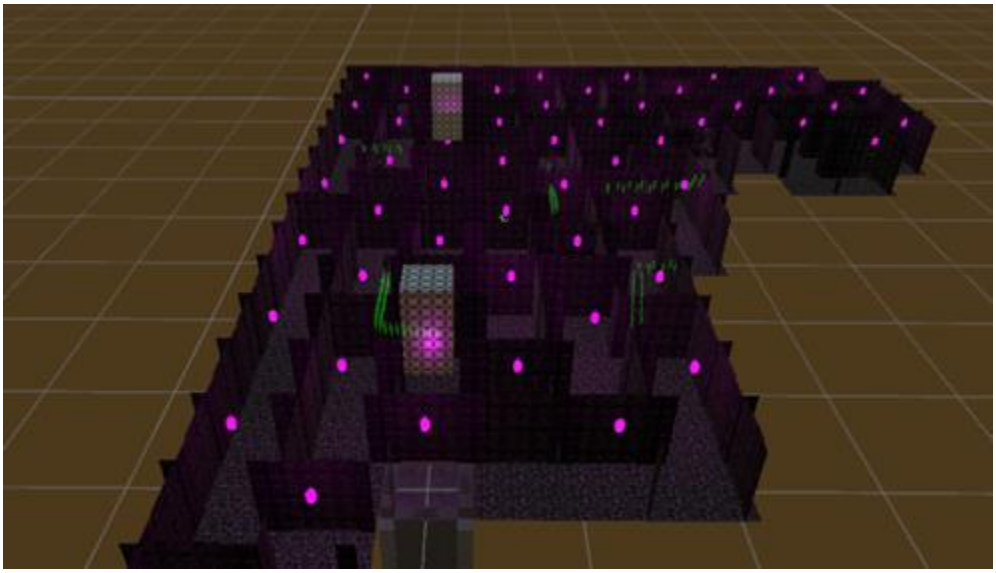
Lighting Generation

In addition to creating the maze's overall structure and form through random generation, the lighting of each section is also randomly generated. This is done by choosing a color at random from C#'s color library, and then increasing or decreasing the RBG values of those colors by random amounts. While the base color is the same throughout a given section, the RBG values are randomized from lantern to lantern so as not to make the lighting seem uniform throughout the section. In addition, the brightness for the light is also randomized from lantern to lantern. As a result of all of this, the lighting of each section is as unique as the layout.

Below are some overhead images of individual sections of some randomly generated mazes. We used one-sided walls to in constructing the maze, and as a result, a birds-eye view does not do a good job of showing the placement of the walls, so instead these images are from angle in an attempt to show more of the walls.



[Figure 32: Randomly-Generated Maze section example 1]



[Figure 33: Randomly-Generated Maze section example 2]

For figures 32 and 33, notice how each of the above sections is different in its shape and size. Even though we try to make the section as even as possible, the algorithm is not perfect, and so some individual sections are bigger than others. Also notice the lighting and the placement of traps. Even though the only traps visible are the crushing traps, they are in obviously different locations. Similarly, the base color of the lighting is different in each of these examples as well. It should also be noted that each of these sections shown are by themselves: the other sections that make up the maze are not shown. This is because our game only spawns the section of the game that the player is in.

Once the loops are generated, all of the ExitNodes are reset to be normal nodes. Once this is done, breadth-first search is used to find the farthest dead end from the player's starting point, and then depth-first search is used to mark all the nodes on the shortest path to said dead end as the new ExitNodes as shown in Code Listing 10. The point of this is to ensure that the start and end of the section are not too close to each other, which was a possibility otherwise.

```
setNewExitNodes(MazeNode root) {
	foreach(node n marked as an exitNode) {
		n.exitNode = false;
	}
	find the deadEnd node deNode that is farthest away from root;
	List(MazeNode) path = path from root to deNode;
	foreach(node n in path)
		n.exitNode = true;
}
```

[Code Listing 10: Pseudo-code for finding the new paths through sections]

Lastly, multiple floors are generated and then connected through ladders, which are placed at the first and last ExitNode in that section as shown in Code Listing 11. These ladders are placed as a means to connect sections that aren't connected on a single floor.

```
ladders(List(MazeNode) sectionRoots) {
	List(MazeNode) sectionEnds;
	foreach(MazeNode n in sectionRoots) {
		MazeNode m = furthest ExitNode from n;
		Add m to sectionEnds;
	}
	Connect the members of sectionEnds to one member of sectionRoots
	//which sectionEnd connects to which sectionRoot varies based on difficulty
}
```

[Code Listing 11: Pseudo-code for generating ladders]

The parameters that determine the how the maze is generated will be provided by the difficulty level that the player chooses when they start the game. Below is a table explaining the sizes of the maze based on difficulty.

Maze Size Table

	Small	Medium	Large	Excessive	Already Lost	Just Why?
Rows	10	15	20	25	30	40
Columns	10	15	20	25	30	40
Floors	3	3	3	4	4	5

[Table 1: Maze Sizes.]

The above table provides an understanding of the overall size of the level generated by multiplying the rows, columns, and floors. This helps give an idea of how much the levels change in scale based on difficulty.

The Tables Below Indicate the number of each type of item, enemy and trap appears within each section the level, based on difficulty level

Item Number Table

	Small	Medium	Large	Excessive	Already Lost	Just Why?
Chalk	3	3	3	3	3	3
Ofuda	3	3	3	3	3	3

[Table 2: Items per maze size.]

As shown, the number of items does not change for difficulty, meaning that the player does not get additional help for harder difficulties, adding to the challenge.

Enemy Number Table

	Small	Medium	Large	Excessive	Already Lost	Just Why?
Oni	1	3	9	15	20	25
Inu	1	1	2	3	4	5
Taka	1	2	4	10	12	18

[Table 3: enemies per maze size.]

Just as the number of cells in increased in difficulty, so too does the number of enemies, meaning that the player has more danger to look out for.

Trap Number Table

	Small	Medium	Large	Excessive	Already Lost	Just Why?
Spike	1	2	2	2	3	4
Crushing	1	2	3	4	4	5
Pit	1	1	1	2	3	3

[Table 4: Traps per maze size.]

As with the size of the maze and the number of enemies, there are also more traps in the higher difficulties as show in Table 4. However, the increase is not as great, meaning the player must do well to memorize the locations and paths to the traps, so that he/she knows what to do when they run into an enemy. This need for strategy is a part of making the game more of a challenge at the higher difficulty levels.

One of the major issues with this algorithm deals with placement of traps and enemies. While there are countermeasures to prevent an enemy from being placed incredibly close to a trap, it is still possible for an enemy to be spawned in between two traps, leaving the enemy nowhere to go. While this doesn't break the game in any way, it does make the existence of certain enemies useless, as if they see the player while he is on the other side of a trap, the enemy will run into the trap and die without the player needing to do anything. Another issue with this method of level generation is that it can make certain levels incredibly difficult. If an enemy is placed near where the player spawns, and begins to chase the player immediately, then the only way the player can survive would be to find a trap as soon as possible and lead the enemy there. However, if the player finds his way into a dead end or another enemy before that happens, the player won't be able to survive. While this is good in that it can provide a good challenge for the player, it is also quite possible to make certain levels nearly impossible to beat without knowledge of where traps are placed beforehand.

It should be noted that when we say a Maze is "generated," it simply means that all of the information needed to construct the maze is randomly produced and stored. The maze itself is not actually created in the game until it is "spawned." To minimize the amount of computing power being used at any given time, there is only one maze section spawned at any given moment: the section that the player is in. When a player collides with a ladder that teleports her

to a new section, the section that she will be teleported to spawns, the player is teleported to the new section, and the previous section is de-spawned as shown in Code Listing 12.

```
collideWithLadder() {
	Section x =section this ladder teleports player to;
	SpawnSection(x);
	TeleportPlayerTo(x);
	DespawnSection(this);
}
```

[Code Listing 12: Pseudo-code for moving between floors]

3.5 A.I. Implementation

Shared Functionality

While each type of enemy has code specified to how it works, there are commonalities between the various A.I. we implemented. Each type of enemy makes use of a state machine to determine what situation the enemy is in and what sort of behaviors it should exhibit. There is a function that defines behavior for every state. During the game loop the A.I. will update its current state and then run the appropriate function. Many states are present amongst all of our enemy types but there are enemy-specific states.

In order to implement our A.I.'s behavior we needed to have various requirements in place for the game objects in Unity. All enemies make use of our maze generator and footprint placer scripts as well as a slightly modified version of the default unity Navmesh agent. The maze generator script is accessed by the A.I. for calculating patrol paths through the maze. The A.I. also make use of footprints as described in section 3.8.

The Navmesh agent was used with some changes to the default settings. We used standard unity functions to turn off the automatic updating of position and rotation which meant the agent only conducted pathfinding. At this point, we make use of the default navigation agent by providing said agent with a destination. The agent then calculates the proper path to follow in the maze to reach the given destination and returns the path to the A.I. Finally, we implemented a custom movement function to allow our A.I. to follow the provided path. We did this because the normal Navmesh movement did not account for gravity when not using rigidbodies. As such we recreated simple lateral movement with a constant downward vector to simulate gravity. In addition to this, we abstracted several functions shared between the enemies into a single controller class.

In order to promote code reuse and maintainability, we made use of a shared superclass for the controllers to get shared functionality from. The shared code contained our functions to:

- Move the enemies
- Check if an enemy can see a particular object
- Manage Patrol Nodes

The function to check if an enemy can see an object must clear four conditions:

- Is the desired object close enough to register?
- Is the desired object in the direction that the A.I. is facing?
- Is the desired object in the same row and column vicinity?
- Does a ray cast to the position not hit a wall along the way?

If the all four conditions return true than the object is considered to be seen by the A.I. Below is some pseudo code that works roughly in the same way as our actual code.

```
SeeObject(thingToLookAt) {
	objectInFrontOfObserver = dot product of
		direction observer is looking and the
		direction vector to thingToLookAt;
	if (objectInFrontOfObserver){
		objectCloseToObserver = distance to
			thingToLookAt < threshold;
		if (objectCloseToObserver) {
			objectInSimilarRowAndColumn =
				result of helper function to
				calculate if the two objects are in ±1
				row and column of each other;
			if (objectInSimilarRowAndColumn){
				noWallBetweenObserverAndThingToLookAt = does a
					raycast to thingToLookAt not hit a wall?;
			if(noWallBetweenObserverAndThingToLookAt){
					return true;
				}
			}
		}
	}
}
```

[Code Listing 13: Pseudo-code for A.I. object detection]

Our custom movement function makes use of the path provided by the Navmesh agent to calculate the desired velocity that the enemy should use to move. Our function then adds gravity to the velocity. Finally, we then call the default character controller movement function with the

combined velocity multiplied by delta time to make movement time dependent rather than frame dependent.

There are also two modified versions of the function to see objects that we use for detecting footprints and Inus. The functions are necessary as the A.I. need to know if they can see a player's footprint or an Inu at all times. The functions initially made use of raycasting but this was changed due to the expensive nature of raycasting necessitating optimization. These functions now work by retrieving all objects marked with the tag "Inu", which is specific to the Inu enemy type, or by retrieving data from the footprint dictionary. The found Inu are then stored into an array for processing. The list of Inu is then refined using the location and orientation of enemies to determine if a given A.I. is capable of seeing any Inu. On account of the potentially large number of footprints to be searched through we implemented spatial hashing into the qualifiers for valid footprints to follow. When a footprint is instantiated, it is added to the footprint dictionary which contains lists of footprints as values with the cell location as the key. We got the idea to make use of spatial hashing from the paper "Optimization of Large-Scale, Real-Time Simulations by Spatial Hashing", which was one of the first academic papers to outline the use of spatial hashing for A.I. decision making. The paper posits that with spatial hashing the performance analysis has O(1) in the best case scenario and O(N^2) in the worst case scenario, which is not likely to happen in our game as it would require all of the footprints to be in a single cell.(Hastings et. al 2005). When checking to see if it can see any footprints a given A.I. will determine what cells it can see at that time. From that list of cells the A.I. will then use the cells locations to retrieve values from the dictionary slots that have keys that match the locations of the cells. If any of the values contain a footprint the A.I. will register that footprint as seen.

Another shared function is to turn the enemy towards the player and this function has the enemy calculate the quaternion it needs to slowly turns towards the player like in traditional Japanese horror.

Our enemy A.I. has a shared functionality in the form of patrolling. The A.I. will navigate the maze using individual cells as patrol points. Path nodes occur at intersections of paths, in other words in any maze cell that is a corner or has more than two openings. Accordingly, we have abstracted the function to set the first patrol node and the function to update the enemy's desired patrol node made to help with patrolling. These functions, called SetClosest and UpdateClosest respectively, determine what maze node the enemy should patrol towards next by finding the closest valid node. For the first a valid node merely does not have a trap along the path towards it, for any further nodes the node must also not be either of the two previous nodes patrolled to. The way the A.I. knows if a path is valid comes from accessing each individual cell along a potential path and checking to see if said cell is occupied by another enemy or a trap.

Specific Implementation

On a more specific level, each A.I. shares several states with slight variations as well as Taka Nyudo and Okuri Inu having their own specific states. The Oni was created first, and all of its states and functions serve as a base for the design of the other two A.I.

The Oni A.I. behaves in a straightforward manner: it has a series of states governing its decisions and acts according to if it should be patrolling or chasing the player for the most part. The idle state simply contains transitions to other states as it is simply an initial state to allow the A.I. to receive input. Below is some pseudo code of how the idle state was generally implemented.

```
idle(){
	if(SeeInu){
		state = flee;
	}
	if(SeePlayer){
		state = chase;
	}
	if(SeeFootprint){
		state = follow;
	}
	if(PatrolValid){
		state = patrol;
	}
}
```

[Code Listing 14: Pseudo-code for the idle state's function]

The chase state has the Oni check if it can see the player, if the answer is no it transitions out of the chase state. If the answer is yes however it does a distance check to see if the player is within killing distance and if the player is the Oni kills the player and ends the game, otherwise the Oni simply paths to the player as shown in Code Listing 15.

```
chase(){
	Transitions to other states like in idle;
	if(SeePlayer){
		Destination = player's location;
	}
	if(CloseToPlayer){
		Trigger game over;
	}
}
```

[Code Listing 15: Pseudo-code for the chase state's function]

The follow state simply checks to see if the Oni can see a player's footprint, if the Oni cannot or if it sees the player then it transitions out of the follow state. If the Oni can see a footprint but not the player then the Oni will utilize the previous implementation of finding footprints in order to get the most recent footprint it can see and then path to that specific footprint. Once the Oni has reached the specified footprint then the Oni will check to see what the next linked footprint is and path to that instead of searching for another footprint as shown in Code Listing 16. This was done to both cut down on time and space usage as well as enable the Oni to follow the player around corners.

```
follow(){
	Transitions to other states like in idle;
	if(nextFootprint == null){
		if(SeeFootprint){
			nextFootprint = found footprint;
		}
		else{
			state = patrol;
		}
	}
	Destination = nextFootprint's location;
	if(CloseToNextFootprint){
		nextFootprint = nextFootprint.getNext(); (getNext()
			is a getter function to retrive the object in a
			footprint's next field);
	}
}
```

[Code Listing 16: Pseudo-code for the follow state's function]

The patrol state checks three things before operating, can the Oni see the player, can the Oni see a footprint, and is there a viable set of path nodes to use for patrolling. If either of the first two are true, the Oni transitions to the appropriate state. The third check is a safety measure to ensure patrol does not run when it would cause errors, this check is also needs to be passed in order to enter the patrol state, so it should never fail but was included just in case the patrol state was ever entered erroneously. While in the patrol state the Oni checks to see if it has a node to patrol to. If it does, it simply navigates to that node; otherwise, it uses the function to get a patrol node that was described previously, updateClosest. Once it has reached its current patrol node, the Oni will enter the look around state as shown in Code Listing 17.

```
patrol(){
	set = false;
	Transitions to other states like in idle;
	if(currentNode == null){
		currentNode = SetClosest(nodes, Oni’s location);
		set = true;
	}
	if(set == false){
		if(CloseToCurrentNode){
			state = lookAround;
		}
	}
	Destination = currentNode’s location;
}
```

[Code Listing 17: Pseudo-code for the patrol state’s function]

A major element of the code regarding the patrol state is the code that prevents enemies from patrolling into traps, as well as each other. This is because we don’t want enemies to accidentally kill themselves while patrolling, and the NavMesh we use to allow enemies to move is too narrow for enemies to move past each other. We could have expanded the NavMesh on the ground to allow the enemies to pass one another, but doing so would allow enemies to move around the triggers of traps, rendering the traps useless to the player. To ensure that these problems were avoided, a boolean field was added to the MazeNode class, which would essentially state whether that node was on a path that a yokai was taking. This would allow for other yokai deciding where to go to check to see if the path to the nearest viable node is obstructed by either another enemy or a trap. The first function in pseudocode written below is a representation of the function used to check if there are any traps or enemies in the way. It is called both in SetClosest and UpdateClosest, when trying to determine a viable node for the yokai to patrol to, with the parameter thisNode referring to the node the yokai is currently on, and targetNode being the Node it is checking. The second function of pseudocode is called in SetClosest and UpdateClosest when a proper node is actually selected for an enemy to patrol to.

```
PathClear(thisNode, targetNode) {
	List<MazeNode> path = getPath(currentNode, targetNode);
	Foreach node in path {
		if(node.actor is a trap-type actor)
			Return false;
		Else if(node.enemyPathNode)
			Return false;
		Else
			Return true;
	}
}

SetEnemyPathNodes(thisNode, targetNode) {
	List<MazeNode> path = getPath(currentNode, targetNode);
	Foreach node in path {
		node.enemyPathNode = true;
	}
}
```

[Code Listing 18: Pseudo-code for an A.I. claiming its path]

The problem with these methods is that they make it very easy for an enemy to be spawned in a place where it won't move at all, because all viable patrol nodes will be blocked by either an enemy or trap. To prevent this, additional code was added to the UpdateClosest and SetClosest functions such that if there was not a valid patrol node it could go to, it would walk to a space right in front of a trap or enemy path, as if it were a valid patrol node. Below is a chunk of pseudocode for that will hopefully give the reader an idea of what we mean.

```
getNextPatrolNode() {
	List<MazeNode> intersections = list of all intersection nodes in the section;
	List<MazeNode> validIntersections;
	List<MazeNode> invalidPatrolNodes;
	Foreach node in intersections {
		MazeNode previous = enemy's current node;
		List<MazeNode> path = getPath(previous, intersection);
		Foreach node in path {
			if(node is an enemy path node or is a trap node)
				Add previous to invalidPatrolNodes;
				Break;
			if(node = intersection)
				Add node to validIntersections;
			Previous = node;
		}
	}

	if(validIntersections is empty)
		return invalidPatrolNodes for the node closest to the enemy;
	Else
		return for the closest node in validIntersections;
}
```

[Code Listing 19: Pseudo-code for retrieving the next valid patrol node]

If the path to a patrol node is blocked, then the node right before the blockage is added to invalidPatrolNodes, and if there is nothing in the validIntersections list, then it is this list that is checked through to figure out where the enemy should go. Once the destination is chosen, both the validIntersections list and the invalidPatrolNodes list are cleared.

The look around state has the Oni rotate around its Y axis for four seconds, if at any time during this rotation the Oni sees either the player or a footprint then it will transition to the appropriate state. If the Oni completes its rotation, then it will acquire a new node to patrol to and return to the patrol state as shown in Code Listing 20. It is important to note that the currentNode is variable held by the Oni and not any particular function.

```
look(){
	Transitions to other states like in idle;
	Spin Oni;
	Timer decrement;
	if(timerFinished){
		currentNode = UpdateClosest(nodes, Oni's location, currentNode);
		state = patrol;
	}
}
```

[Code Listing 20: Pseudo-code for the look around state's function]

If the Oni sees the Okuri Inu in any state other than stun, then the Oni will enter the flee state where it returns to its starting cell as shown in Code Listing 21.

```
flee(){
	Transitions to other states like in idle;
	Destination = starting location;
}
```

[Code Listing 21: Pseudo-code for the flee state's function]

If the Oni is hit with an ofuda at any time, it will be transitioned into the stun state for fifteen seconds during which the Oni will do nothing but rotate towards the player. We allowed the Oni to still rotate towards the player as an indication that the Oni is still alive and needs to be considered. Once the duration of the stun has been completed, the Oni will transition into the appropriate state by taking into account what it can see at that timer interval as shown in Code Listing 22.

```
stun(){
	Timer decrement;
	if(timerFinished){
		Transitions;
	}
}
```

[Code Listing 22: Pseudo-code for the stun state's function]

Finally, if the Oni is hit by a trap at any point, the Oni will report its death to our analytics system and delete itself.

The Taka Nyudo A.I. behaves largely in the same way as the Oni A.I. The main differences between the two A.I. are the inclusion of a taunt state for the Taka Nyudo and the Taka's chase and flee states working slightly differently, besides those three differences the other differences for the Taka are numerical value changes for the individual states, like the stun timer being twenty seconds for the Taka for example. The main change to the chase state for the Taka is that instead of a kill distance the Taka performs a distance check to see if it is close enough to begin taunting the player. If the Taka is close enough, it will transition into the taunt state as shown in Code Listing 23. In the taunt state, the Taka will grow taller by directly changing the local scale of the Taka's transform component over the course of two and a half seconds. If the player looks up at the Taka during this time period, the player will be killed and the game will end. If the player manages to not look up at the Taka until after the time period has expired, then the Taka will enter the flee state and shrink back to normal size over the course of two and a half seconds before continuing to flee normally. We detect if the player is looking up by taking the dot product of the forward vector of the player and the positive Y unit vector.

```
taunt(){
	Transitions to other states like in idle;
	if(CloseToPlayer){
		grow;
	}
	if(FullyGrown){
		state = flee;
	}
	if(PlayerLookingUp){
		Trigger game over;
	}
}
```

[Code Listing 23: Pseudo-code for the taunt state's function]

The Okuri Inu A.I. differs from the Oni A.I. in three main ways:

1. The first is that the Okuri Inu does not flee from other Okuri Inu.
2. The second is that chase state behaves slightly differently.
3. The third is the inclusion of the stalk and cornered states.

The chase state for the Inu behaves like the Taka in that instead of a kill distance the Inu instead checks to see if the player is close enough to stalk and if yes the Inu will transition into the stalk state. The stalk state works by trying to maintain a distance from the player that falls within a set range of values. If the Inu ends up further away than the upper bound value, it will check if it can see the player or the player's footprints and transition to the appropriate state. If the Inu is closer than the lower bound value, it will attempt to back away from the player. The way the Inu does this is by calculating which way to backup away from the player through taking into account the orientation of the player and itself as well as which of the surrounding cells are viable to travel into, that is, they do not have a wall blocking passage. Through this, the Inu will choose a node which is not on the other side of the player from the Inu and navigate towards that node if there is a valid node to path to. If there is not a valid node the Inu will walk directly backwards away from the player until it hits a wall. If the player backs the Inu into a wall, the Inu will enter the cornered state. In addition to this, while in stalk state, the Inu will decrement an attack timer as the seconds go by and will transition into the cornered state if the attack timer reaches zero or negative values as shown in Code Listing 24.

```
stalk(){
	Transitions to other states like in idle;
	if(TooCloseToPlayer){
		Try to back up from player;
		if(Cannot back away){
			state = cornered;
		}
	}
	Attack timer decrement;
	if(Attack timer finished){
		state = cornered;
	}
	if(NotCloseEnoughToPlayer){
		Move closer until within range of acceptable values;
	}
}
```

[Code Listing 24: Pseudo-code for the stalk state's function]

In the cornered state, the Inu will attack the player by chasing them down if the player continues to push the Inu into a wall. If the player enters within the Inu's killing distance during the cornered stat, the Inu will kill the player and trigger a game over. If the player backs away

from the Inu, the Inu will either enter the stalk, follow, or idle states based upon what the Inu can see. The Inu will also attack if the attack timer runs out in the cornered state as shown in Code Listing 25.

```
cornered(){
	Transitions to other states like in idle;
	if(TooCloseToPlayer){
		Try to back up from player;
		if(Cannot back away){
			Attack player;
		}
	}
	Attack timer decrement;
	if(Attack timer finished){
		Attack player;
	}
}
```

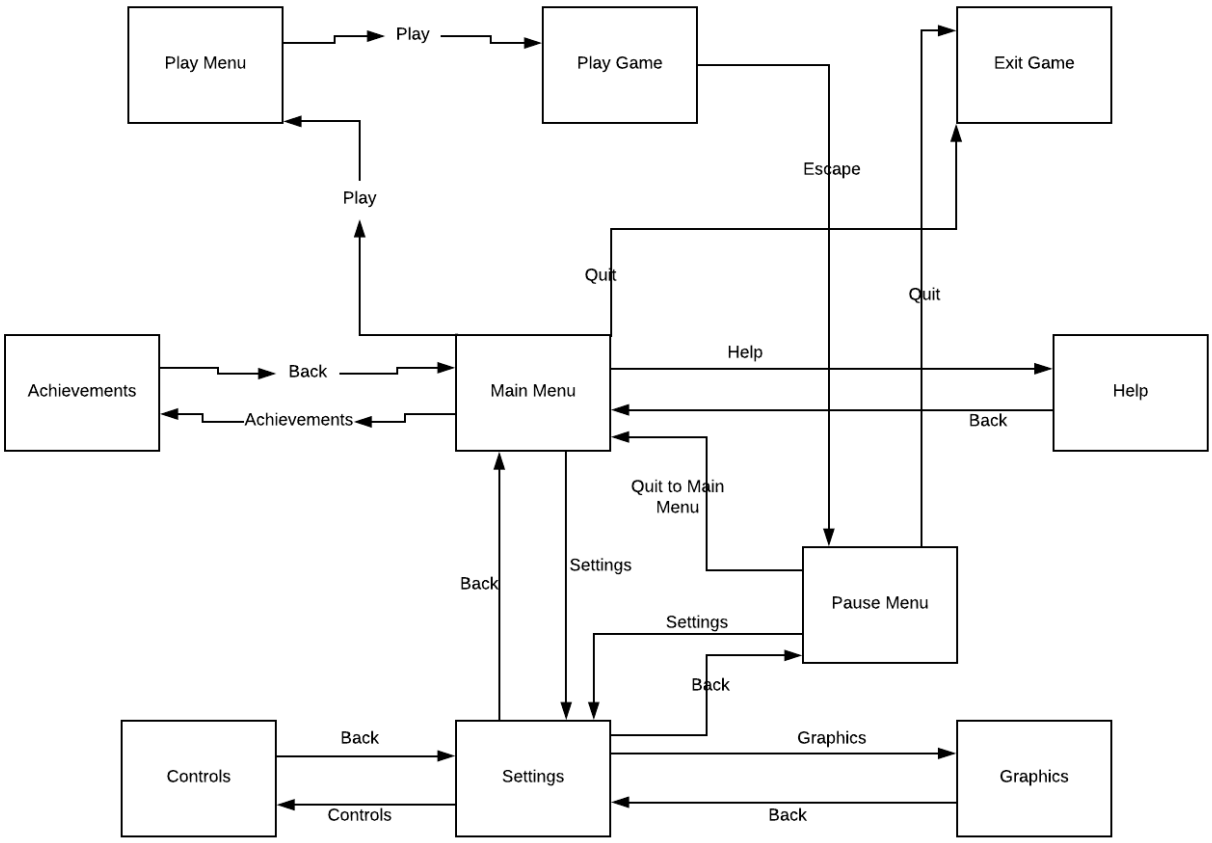
[Code Listing 25: Pseudo-code for the cornered state's function]

3.6 User Interface

In order for a player to interact with our game there needs to be an interface to allow for input and provide feedback. In this section we detail how we designed and implemented our User Interface (UI) system, including the feedback system and the menus used.

Menus

In order to enable users to interact with our game there needed to be menus to start the game and choose options. We implemented menus in our game using the Unity User Interface (UI) components called Canvas and Panel. In Unity, a Canvas is a digital representation of a computer screen and contains all other UI components of the game. A Panel, on the other hand, is what contains scenario specific UI components, an analogy would be if a Canvas were a video game a Panel would be a level of that game. Panels allow for various interfaces to be presented by swapping what Panels are active at a given time. For our game, we created our menu system by having buttons perform this swapping. Below is a flowchart showing the relationships between all of the Non-VR menus in the game:



[Figure 34: Flowchart of our menus]

Additionally, in Unity, functions can be assigned to buttons so we had functions that coordinated between inputs in the menus with the game manager to have said inputs affect gameplay. This is what we used to implement the ability to change game settings in the play menu. Below are pictures of the main and play menus.

[Figure 35: Pictures of our Main and Play menus.]

Controls Menu

In *Maze of Rebirth*, the player can walk forward, backward, left, or right, jump (in the non-VR version), sprint, draw using chalk and throw talismans. Each of these controls is set to a default button on the keyboard or controllers depending on what the player is using to play the game. To provide players the opportunity to play in whatever way made them feel the most comfortable, we added a controls menu that allowed players to change the controls to whatever variation he wanted. This was done by creating objects with text fields attached that were associated with specific actions. When the object is selected, the player can choose whatever input he wanted to, and the object would take that input and set whatever action was associated with the object to the input given.

The following pseudocode is for whenever a player selects an object associated with a specific action on the controls menu

```
objectSelected() {
	On next input {
		object.associatedActionInput = input
	}
}
```

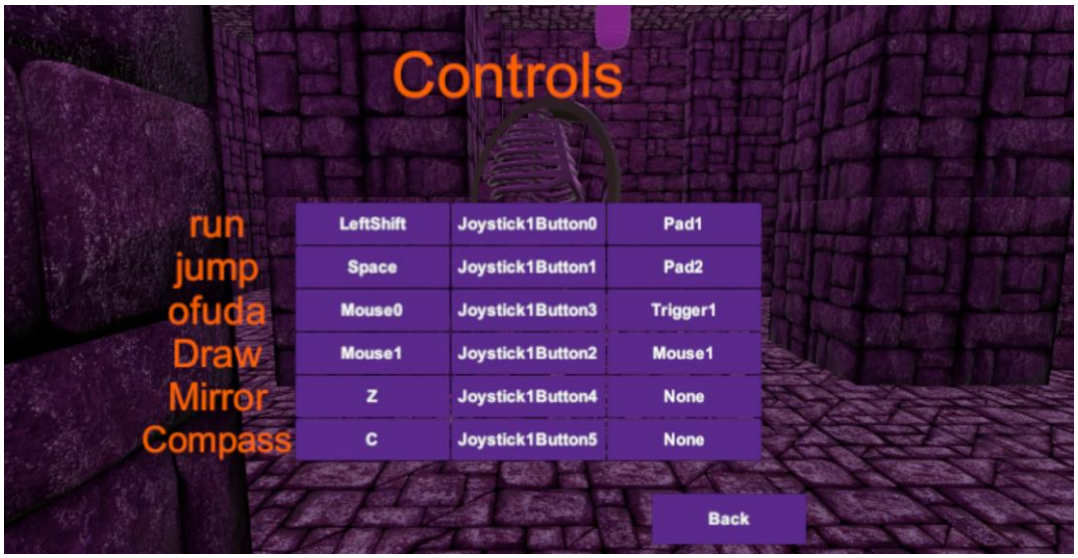
[Code Listing 26: Pseudo-code for mapping controls to inputs]

The following pseudocode is for whenever a player presses a button while in gameplay

```
onInput(input) {
	Foreach possible action {
		if(action.associatedObject.associatedActionInput == input)
			Do action;
	}
}
```

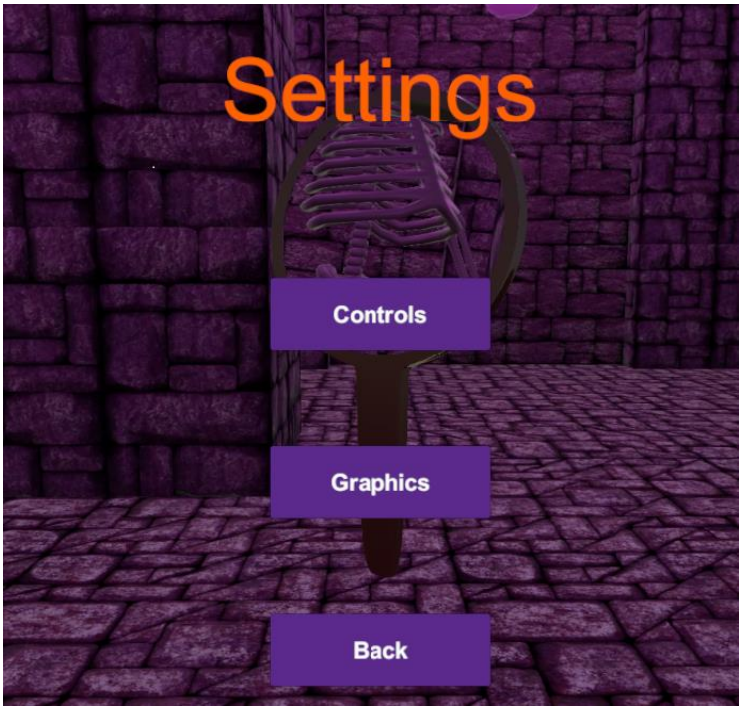
[Code Listing 27: Pseudo-code for mapping controls to inputs]

The one exception to this ability for the player to change controls is that he/she cannot set an action to be activated by the clicking of the left or right mouse (assuming he is playing on a computer with a standard keyboard setup).



[Figure 36: Picture of our Controls menu.]

It should be noted that the controls menu is reached through the settings menu, shown below.



[Figure 37: Picture of Settings Menu]

Unfinished Menus

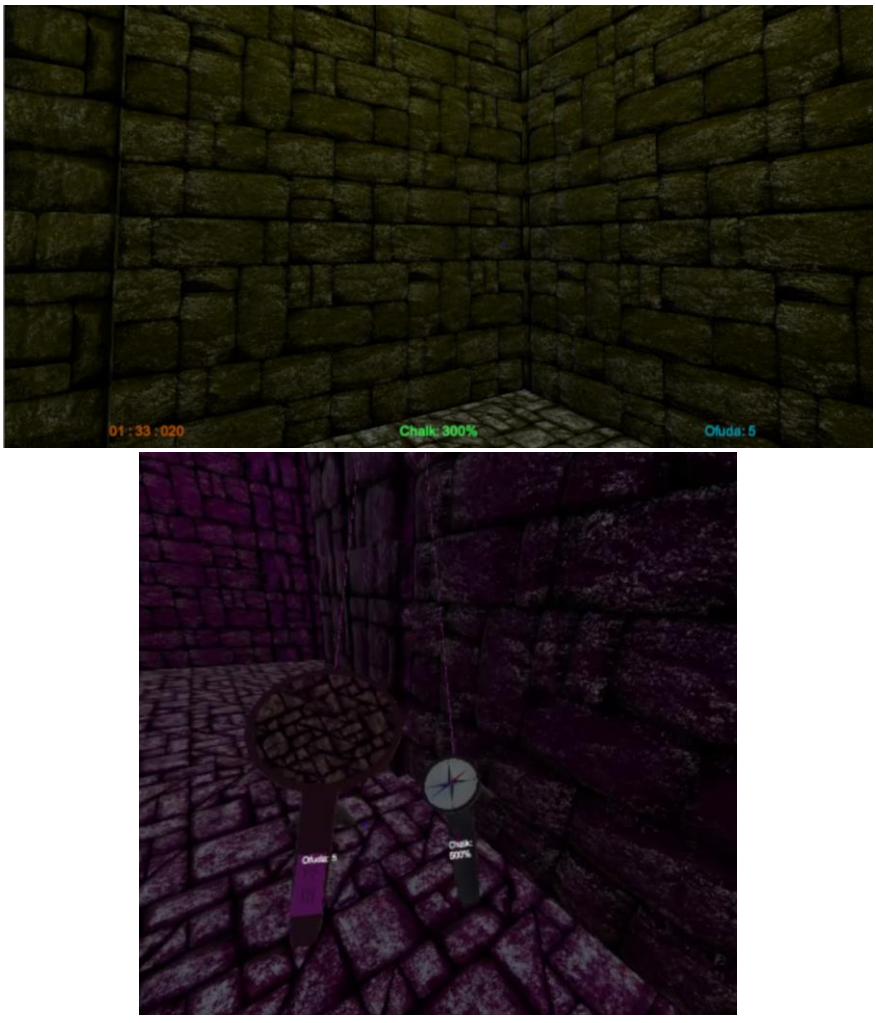
The options for "graphics" in the above figure, as well as those for "help" and "achievements" shown in figure 35, were all meant to take the player to new menu screens. Unfortunately, we did not finish those elements of the game, and so said menu screens are empty, which is why no figures of them appear in this report.

VR Menu

One major aspect of the game we wished to complete was VR specific menus that the user can see with the VR headset and interact with using the VR controllers. The menu system we implemented requires interaction occur using keyboard and mouse instead of the VR setup. We did not have the time to adapt the menu system as described above in order to not cause VR sickness. The VR sickness would be caused by the static images remaining in front of the players' eyes even if they turn their head. Our plan to correct for this was to create a level with a physical environment to stand in for menus. Inside this environment the player could find objects that would represent the various choices a player could make in our more traditional menus. Once the player holds an option related object he or she could interact with said object in lieu of pressing a button. This would allow the player to interact with options without having to deal with sickness induced by a static screen in front of his or her face.

Player Feedback

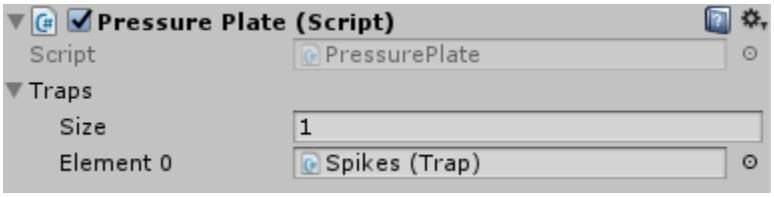
In order for any game to be played properly, the player must be able to perceive and understand the results of his or her actions. Accordingly, we implemented two distinctive user interfaces to allow for player feedback, one for Virtual Reality controls and one for keyboard and mouse. The design for the feedback: in a keyboard and mouse-controlled environment is very straightforward. In this version at the bottom of the screen, the player has several indicators with each of said indicators being a different color and clearly informing the player of what they are tracking. The information provided to the player in this case includes the state of his or her inventory and how much time has passed in game as seen in the top part of figure 38. The feedback for the Virtual Reality version is significantly different. Due to the traditional feedback designs, like what we used for mouse and keyboard, causing VR sickness in many people, we knew we had to make use of a more unconventional design. Instead of the information being attached to the screen, we instead attached the information to the player's hands as seen in the bottom part of figure 38. This allows for the player to quickly and easily access the information without it being intrusive.



[Figure 38: The non-VR interface (top) and VR player interface (bottom)

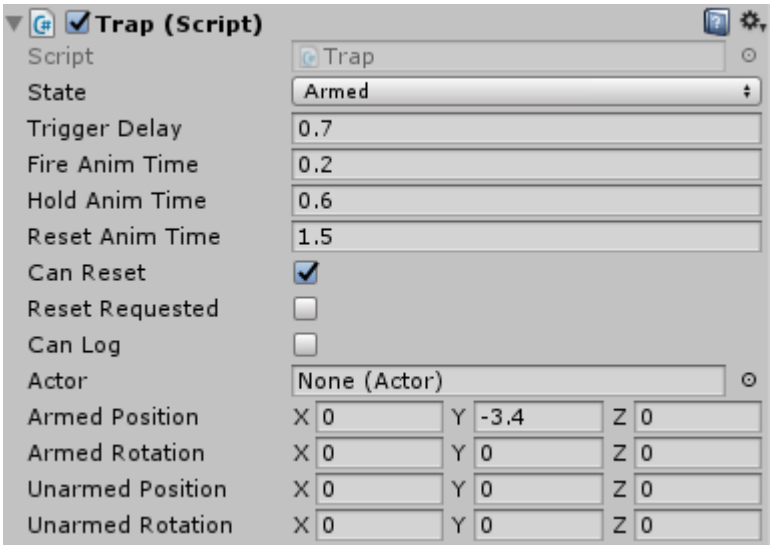
3.7 Traps

Each trap is made up of one or more moving objects with the trap component and a set of pressure plates that trigger them.



[Figure 39: Pressure Plate Component]

A pressure plate will trigger all of its connected trap components when a new object collides with it (See Figure 39). The pressure plate will stay down as long as there is an object still colliding with it, which it does by tracking each of the game objects that has entered its trigger collider. An object is removed from the tracking if the object exits the collider or is destroyed. When the pressure plate is released, it will request it's connected traps to reset.



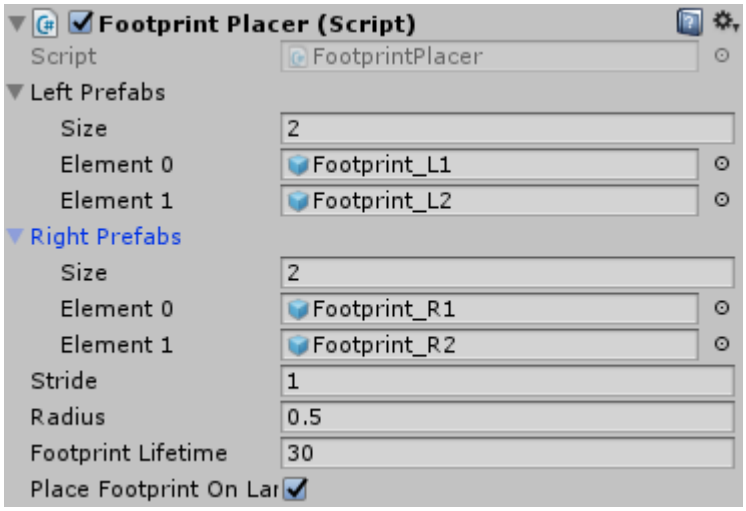
[Figure 40: Trap Component]

The trap component has 5 states that it progresses through. An armed trap can be triggered by a pressure plate. Upon triggering, it waits a period of time before playing the fire animation. The firing animation linearly interpolates the object from the armed position to the unarmed position. After the fire animation has finished, the trap will stay in the unarmed position until a reset has been requested by the pressure plates. The reset is requested when a pressure plate is released. Once a reset has been requested and trap is unarmed, it will begin the reset animation and rearm the trap. Each of the animation times and the trigger delay can be set in the

components properties which can be seen in the figure above. The trap also takes relative positions and rotations for the armed and unarmed states (See Figure 40). A trap object will kill an actor that triggers its collider if it is not resetting. This allows a player to walk against a resetting trap without dying instantly. The trap component works for all trap varieties.

3.8 Footprints

Each character in our game places footprints as they move throughout the maze, which is done using our footprint placer script (See Figure 41). The script tracks distance the player has traveled on the ground, and when the tracked distance reaches the stride distance, a footprint is placed on the ground below the character. The stride distance can be set in the Unity editor for each individual character. The script requires the CharacterController to determine if the character is grounded or in the air. This is used to prevent tracking distance in the air and to place footprints when the character has landed on the ground.



[Figure 41: Footprint placer script]

To give variance within the path of footprints, the script alternates between left and right footprints. Each character has a list of left and right footprints. The script randomly chooses one from the list when placing a new footprint (See Code Listing 28). The distance between the left and right footprints can be controlled in editor for each character using the radius property.

In order to help the enemies' artificial intelligence, a doubly linked list of footprints is maintained. This allows an A.I. to get the next and previous footprints in the path, given a footprint. In addition, each of the player's footprints are added to a spatial hash. The spatial hashing is used to efficiently search for footprints given a location. Spatial hashing is a three-dimensional representation of a hashmap. This is done by converting the 3-dimensional location into the cells row, column, and floor. This information is then condensed into a single integer

which is used as a key for the hashmap. This allows an entity to supply a location in a cell and get a list of all footprints within the cell.

```
private void placeFootprint() {
	Vector3 footPosition = transform.position + transform.right *
		Radius / 2f * (rightFootLast? 1 : -1);
	Ray ray = new Ray(footPosition, Vector3.down);
	RaycastHit rayHit;

	if(Physics.Raycast(ray, out rayHit, controller.height, mask))
	{
		//New footprint position
		Vector3 position = rayHit.point + rayHit.normal * .001f;

		//Select next footprint prefab
		GameObject prefab = NextPrefab();

		//update previous
		previousFootprint = currentFootprint;

		//spawn
		currentFootprint = Instantiate(prefab, position, transform.rotation,
			footPrintParent.transform).GetComponent<FootprintList>();
		currentFootprint.getComponent<FootprintDecay>().
			SetLifeTime(FootprintLifetime);

		//update Linked list
		if(previousFootprint != null)
			previousFootprint.setNext(currentFootprint);
		currentFootprint.setPrevious(previousFootprint);

		//Prep for next footprint
		distanceTraveled = 0f;
		rightFoolLast = !rightFootLast;

		//Attach to moving objects
		if(rayHit.collider.gameObject.layer == dynamicObjectLayer)
		{
			//attach footprint to dynamic objects
			AttachTo a = currentFootprint.gameObject.
					addComponent<AttachTo>();
			a.To = rayHit.collider.gameObject;
			a.UseWorldSpace = true;
		}

		//Add to spatial hash
		if(gameObject.CompareTag("Player"))
		{
			GameManager.Instance.AddFootprint(currentFootprint,
			footPosition);
		}
	}
}
```

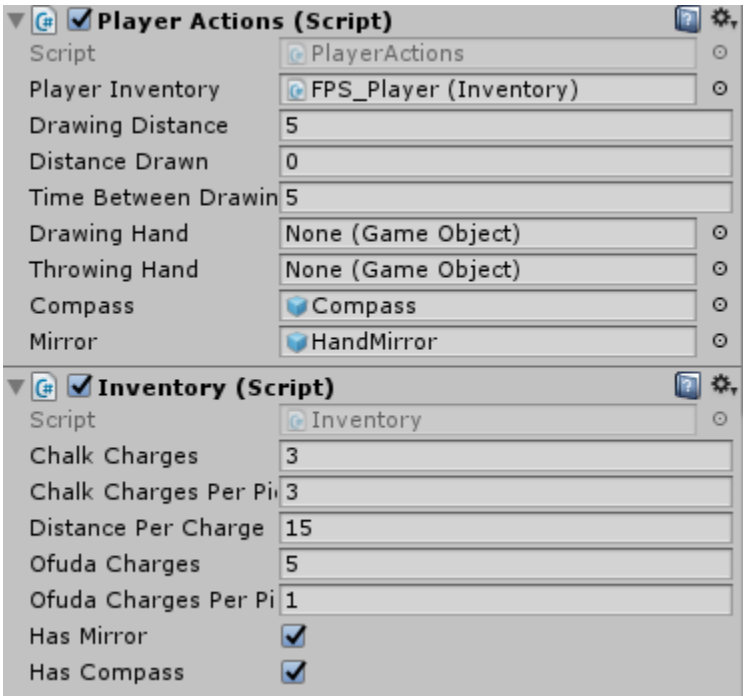
[Code Listing 28: PlaceFootprint method]

Each footprint decays over time. During the decay, the footprint will slowly become transparent and eventually disappear. This is done using both a script and a shader. The shader has an additional parameter for transparency. Over time, the footprint's transparency is controlled using the ratio of current time alive divided by the lifetime of the footprint. The footprint becomes completely transparent as it reaches the end of its lifetime and destroys itself. The reason a custom shader was written for this is, the Unity StandardSpecular shader does not take transparency into account for specular lighting. The footprints use specular highlights to increase its visibility in the dark maze (See Environment Design Section).

3.9 Items

Inventory

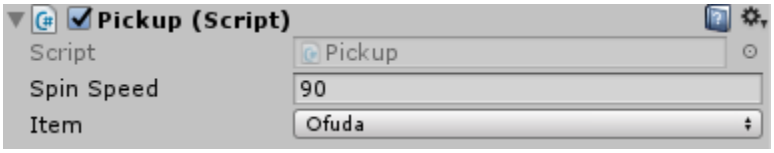
The player's items are tracked with the inventory script. The script allows the designer to set how many charges of chalk and ofuda the player starts with (See Figure 42). Additionally, the designer can control whether the player starts with the mirror and or compass. The inventory script controls how many charges a chalk and ofuda pickup gives.



[Figure 42: Player Actions script, Inventory script]

Pickups

Players may find and collect items throughout the maze. Each item prefab has the pickup script component, which determines what type of item the pickup is. When the player collides with the pickup, the pickup will add the item to the player's inventory and destroy itself. To draw attention to the item pickup, the object rotates over time. The rotation rate can be controlled for each item prefab via a property in the Unity editor (See Figure 43). The script has been written to work with any item type without having to be rewritten if we decide to add more items in the future.

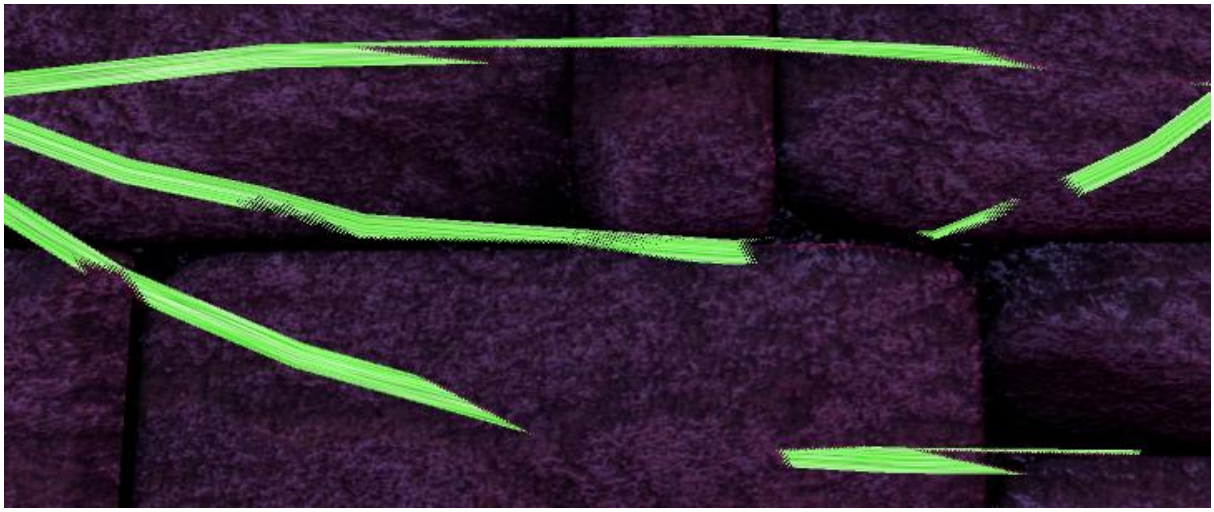


[Figure 43: Pickup script]

Chalk

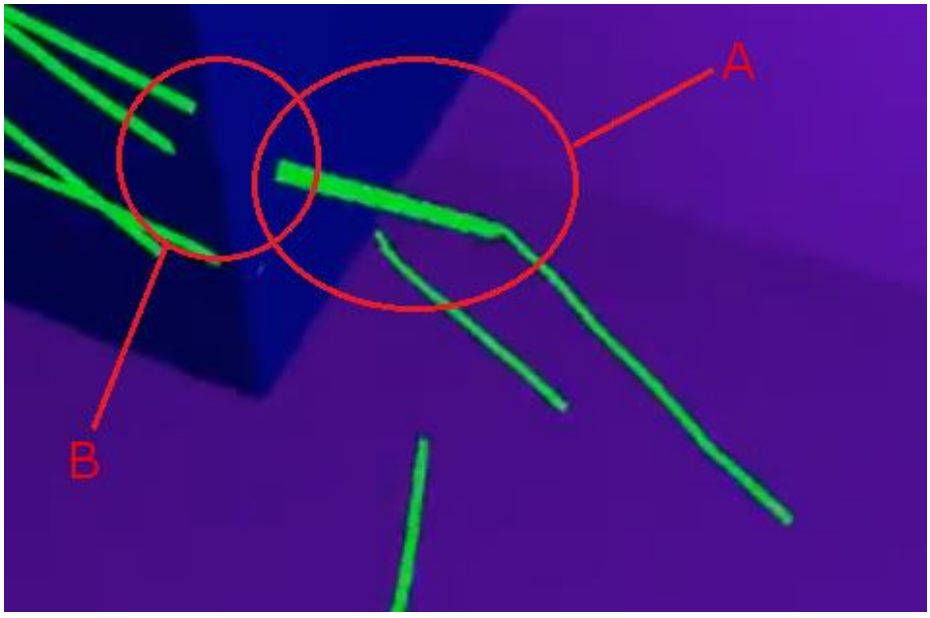
One of the primary ways the player can interact with the world is by drawing chalk marks. This works similarly for both the virtual reality and the traditional first-person controls. In the VR version of the game, the right-hand controller is used to point where the player would like to draw; in the traditional version of the game, the center of the player's camera view is used to point where they would like to draw.

When the player wishes to begin drawing, the PlayerActions script checks with the inventory to see if they have chalk. If they have chalk, the PlayerActions script will spawn a new LineRenderer. A raycast is shot at the location they are pointing. If the raycast hits a game object on the Level Layer, the ray hit position is added to the LineRenderer's line. In order to prevent the LineRenderer's line from overlapping with the environment and causing z-fighting (See Figure 44), a small fraction of the face normal of the surface the raycast hit is added to the position of each point. As the player changes what they are pointing at, new positions are added to the LineRenderer's line. As new positions are added, the distance between each point is accumulated. This distance is used to determine if the player has used up a charge of chalk. The distance a chalk charge lasts can be set in the inventory script properties (See Figure 42).



[Figure 44: chalk z-fighting with wall]

The current implementation correctly draws on walls, though it has one problem. When the player is facing a corner and draws from the corner to the floor, a line will be drawn from the corner to the floor (See Figure 45). This line is not attached to the wall or the floor and would not work for chalk as the player cannot draw chalk in mid-air (chalk should be limited only to surfaces). The other problem associated with this is when drawing between two wall surfaces at a corner quickly, a line may be drawn that passes through the corner (See Figure 45). The solution to this problem is using the surface's face normal vector. The face normal is a vector that points out of a surface. When the vector changes, a new line should be started.



[Figure 45: A - Line from wall to floor, B - Line through corner]

Another problem that occurs for both chalk and footprints is handling moving objects. The player may decide to draw on the doors of the floor trap. These doors open when the trap is triggered. Both the chalk marks and footprints on these doors should move with the doors rather than float in midair. This is solved by putting moving objects on a separate layer we called the DynamicObject layer. In order to make the chalk drawings and footprints follow the moving objects, a script called AttachTo was created. The script allows an object to follow another object's position, without parenting them together. The script saves the starting position of the game object it is following and uses it to calculate an offset. The script then applies this offset to its own position every frame, causing it to follow the other object.

Each chalk drawing is saved by the analytics system. Since a drawing can be made up of many lines, we needed a way to associate many lines back to a single drawing. An easy way to differentiate the drawings is using time. When the player finishes a line, a timer is started. If the timer finishes before the player begins a new line, we assume the drawing has been completed. The completed drawing is then sent to the analytics to be recorded. The time between each drawing can be modified through a property in the Player Actions script (See Figure 42).

Compass

In our game, the player's compass is a physical object rather than a UI element. In the VR version of our game, the compass is attached to the right-hand controller and is always active. For the traditional first-person controls version, the compass is shown in front of the player while they hold down a button. The compass is made up of both a frame and a pin. The pin should always be pointing north. North in our game is the positive z axis. The compass pin should only rotate on the local Y axis as any rotation on the other axes would cause the pin to intersect with the frame itself. The pin's local yaw rotation is set by reversing the frame's world yaw rotation (See Code Listing 29). This results in the pin always pointing towards the positive z axis while locked to the local xz plane.

```
//Update is called once per frame
void Update() {
       transform.localRotation = Quaternion.Euler(0, 360 -
              transform.parent.rotation.eulerAngles.y, 0);
}
```

[Code Listing 29: Compass Pin rotation towards north]

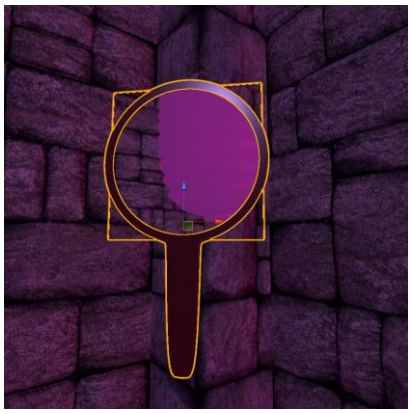
Mirror

In the VR version of our game, the mirror is always active and is attached to the player's left-hand controller. The traditional first-person controls use three buttons to activate it. One button orients the mirror to look behind the player. The other two buttons allow the player to

look left or right by placing the mirror at a 45-degree angle in front of the player's camera (See Figure 46).

[Figure 46: VR and traditional controls mirrors]

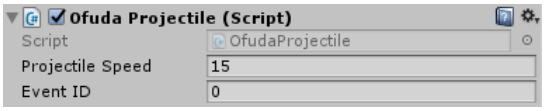
The mirror itself is made up of a shader and a mirror reflection component that can be found on the Unity3D wiki. The script is a modification of the water script from Unity's Pro Standard Assets. The modification was done by Aras Pranckevicius. Pranckevicius removed the refraction parts of the script that was used for water (Pranckevicius, 2015). It was created to work with Unity's plane game object. The mirror model used in our game has a circle rather than a plane for the reflective surface. In order to create a circular reflective surface, the shader was modified to use transparency. The fragment shader discards each fragment where the texture is not completely opaque. A texture with an opaque circle and transparent edges can then be used to create a reflective surface in the shape of a circle. The plane can then be placed inside the mirror frame, however only the circle inside the mirror is rendered (See Figure 47).



[Figure 47: Mirror with reflection plane highlighted]

Ofuda

Ofuda are used to stun enemies. In the VR version of our game, the ofuda are shot from the left-hand controller, whilst the traditional first-person controls use the camera center. When the player pushes a button, an ofuda is spawned. The ofuda has a projectile script which uses the RigidBody component. The projectile script sets the RigidBody velocity and adds force in the forward direction. The projectile's speed can be set in the editor (See Figure 48). This causes the ofuda to move forward in the direction it was thrown and while still applying physics. A collider is attached to the ofuda. When the collider hits an enemy, it sends a message to stun them. Alternatively, if the ofuda collides with a player, the player picks the ofuda back up.



[Figure 48: Ofuda Projectile script]

4. Testing

4.1 Introduction

In this section of the report we describe our methodology and results for testing and refining our game. Section 4.2 describes how we began with a paper prototype to materialize the underlying concepts of our game. Sections 4.3 and 4.4 describe the components we needed to consider for out testing. Section 4.5 details our methodology of how we went about testing. Section 4.6 is a summary of the results of our testing. Lastly section 4.7 is our conclusions drawn from our testing.

4.2 Paper Prototype

Before we even started coding, we initially tested out our ideas with paper prototypes. Paper prototyping is a common method of making sure the general design decisions of a game are viable by making the game in micro scale using simple materials. In these prototypes we drew out individual mazes, and had players navigate through them, by describing their surroundings, including possible directions to go, whether there are enemies nearby, and so on, and then asking them how they wanted to act in these situations.

We did four iterations of paper prototyping. With each iteration we refined our ideas and added more to the gameplay we represented in our prototype. From our first iteration we learned to lessen the mental load on the players by providing flash cards that summarized situations and made the game easier for the player to understand by changing the prototype to be done in the first person perspective. From the second iteration we learned that the maze size should be reduced and the player should be given a compass to allow them to know their orientation. From the the third iteration we learned that the players needed an area to slow down and think through their situation. For the final iteration we introduced puzzles and learned that the players did not want to do puzzles. From all of these iterations we were able to largely refine our design decisions and begin to know what sort of issues players would run into while playing our game.

4.3 Study Design

From the outset of our project, we wanted to ensure that the game that we created would accomplish our experience goals like many great game designers before us. That desired experience is a combination of anxiety and dread from trying to avoid enemies mixed with bouts of relief from finding ladders. We knew that we would need our game to be playtested to confirm if we were either accomplishing our goals or if we were at least on the right track.

4.4 Experiment Design

In order to run our study, we had three critical components to take into account:

- Our game
- The technology we were using
- Our ability to observe the players

The first and most critical component was our game. In order to meet our experience goals, we needed to design a game around those goals. As we were aiming to create a game that combines the feelings of horror, being lost, and relief, we decided to create a maze game modeled after traditional Japanese horror. The main aspects of Japanese horror we included were powerful and nigh-unstoppable beings, slow encroaching death, enemies creepily being aware of the player and turning towards them slowly, and a tense, low-lit atmosphere. This was accomplished through a mix of aesthetic and gameplay design.

The second component of our study was the technology we chose to implement it with. Virtual Reality (VR) is a recent development that dramatically increases the ability for game designers to create perceptual immersion into the games they have produced. This is caused by how VR surrounds the player's vision with the game he or she is playing and having the audio feed directly into the player's ears. VR is not without its challenges, which leads to it being suited to several types of games of which horror games are among the best matches.

The third component of our study is our ability to observe the players. In order to ascertain if we have met our goals, a method to discern the player's experience of playing the game is required. Our solution to this problem was to use a combination of surveys and careful observation of an individual as he or she plays the game. This solution allows us to make use of both our impressions of how the player reacted as well as how the player believes he or she reacted.

4.5 Methodology

Our project JapanActionGo is dedicated towards making a Virtual Reality (VR) survival maze game. The player will have to escape a maze containing demons by using traps and talismans. There will be two versions of the game, one using Virtual Reality controllers and one using either mouse and keyboard or a console gamepad controller. Our testing will be used to improve the quality of our game. The focus of our testing will be on the player's opinion of our game and how he or she felt playing our game. The testing data will be used to evaluate how effective our motion sickness mitigations are. The tests will consist of a pre-test survey, a gameplay session, and a post-test survey.

Ensuring Safety

We will ensure the wellbeing of our tester by informing them of all the issues that could possibly occur while testing, this includes sections that could cause motion sickness and other symptoms such as claustrophobia, as well as making sure the tester is aware of the possibility of bumping into real objects while playing and that we will be warning them about when he or she is in danger of doing so. We will clean the Virtual Reality equipment between play sessions with anti-bacterial in order to prevent any illness issues, this does not guarantee that the user will not get encounter bacteria and get sick and we will inform the player as such. We will also allow the user to opt out of testing at any time with the option of having us delete any and all data related to that particular user. The testers may also request the deletion of the data concerning after their tests have concluded. The tester will be informed of what type of data we are collecting and be informed that the data will only be used for the purposes of developing this game. In order to begin testing we had to receive certification from Worcester Polytechnic Institute's Institutional Review Board (IRB) that we were adequately ensuring the safety of our testers. As such we filled out the appropriate IRB forms with the information above and received certification.

Gameplay Session

Our gameplay session will consist of a 3D game made from the game creation software Unity with the player making use of a Virtual Reality headset and controls. The game will play in real time and will provide feedback of the current status of the player and what in-game items he or she has. The gameplay will consist of navigating a maze while avoiding three types of enemies and making use of what the player finds in the maze.

Data Collection Methods

Overall, we employed three methods of collecting player data from testing. The player may take an optional pre-test survey concerning their history with VR games so we may contextualize their results to see if our game was particularly egregious. We will then have the tester play the game for anywhere between five and thirty minutes, depending on the choice of the tester. During the playthrough of the game, we will make use of observation and in-game analytics. After the playthrough has ended, the player may take an optional post-test survey to provide additional information that may not have been conveyed otherwise. We will also allow the user to opt out of testing at any time with the option of having us delete any and all data related to that particular user. The testers may also request the deletion of the data concerning after their tests have concluded. The tester will be informed of what type of data we are collecting and be informed that the data will only be used for the purposes of developing this game.

In-Game Analytics

We will be using two methods to record the player's actions during the game. Our first recording method utilizes various functions within the game to record data on what the player does, and stores it in a local database. The data is anonymized by having a unique number for each player without any personal information. The number is given to the player at the end of the session. This number can be given back to us for deletion of their game data if he or she so wishes. We will not maintain a list of names to go with the numbers to ensure anonymity. When analyzing, we will mostly focus on what the player actor did but we will use the data from the other actors to contextualize those player actions. Sample analytics include when actors were in specific locations, timestamps for item use, and when an actor dies who killed it.

Observation

For our second recording method, we will carefully watch and record what the player does in person. We will be engaging the player in dialogue and transcribing the tester's thoughts on the game. The unique session number will be recorded and associated with the in-game analytics data as well as the pre-test and post-test surveys should the player choose to take them.

Surveys

In addition to recording concrete data on the player's actions, we will also be getting feedback from the player regarding how he or she felt playing the game in Virtual Reality (VR), and what elements made them feel sick, if any. This feedback will be obtained by having the player fill out surveys before and after their play session, as well as direct observation while he or she plays. The pre-test focuses on the user's experiences with getting motion sickness as well as their history of playing games. The post-test survey focuses on what the user experienced during their playthrough of the game. These surveys will not ask for any names. The surveys will have a unique session ID number, which will be connected to the gameplay session ID numbers to create a cohesive set of data that will allow the player to remain anonymous.

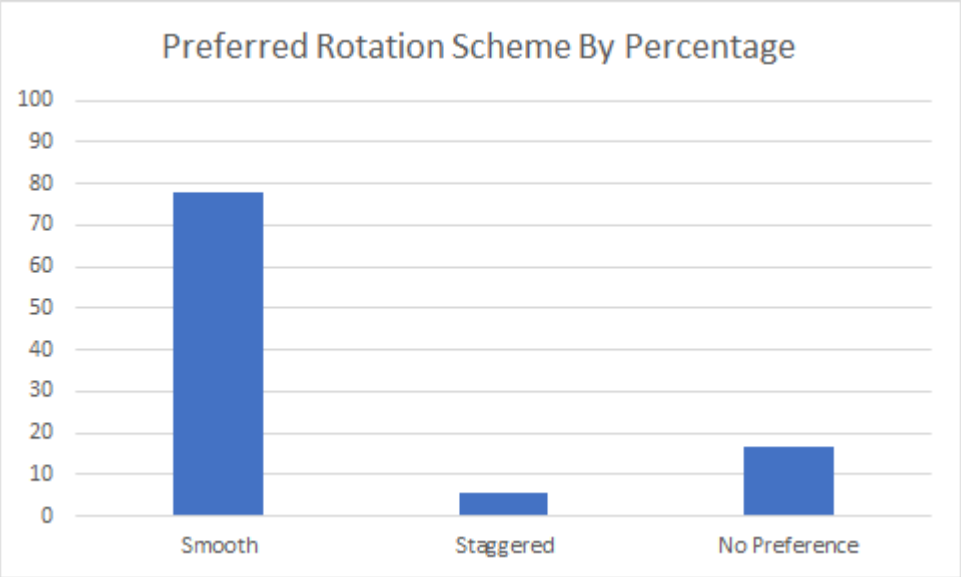
Analysis

The recording of concrete data will allow us to find ways to balance the game, such as tweaking the algorithm for enemy placement, increasing or decreasing enemy speed, and changing the number of enemies, traps, and items in any given level. The direct player feedback, on the other hand, will allow us to tweak our approach to the Virtual Reality component of our game, so that we can make a game that minimizes motion sickness.

4.6 Results

After extensive playtesting, we have gathered data from a total of 36 different players of which 28 chose to take our surveys. After having organized this data, graphs have been created to help show the correlation between types of data.

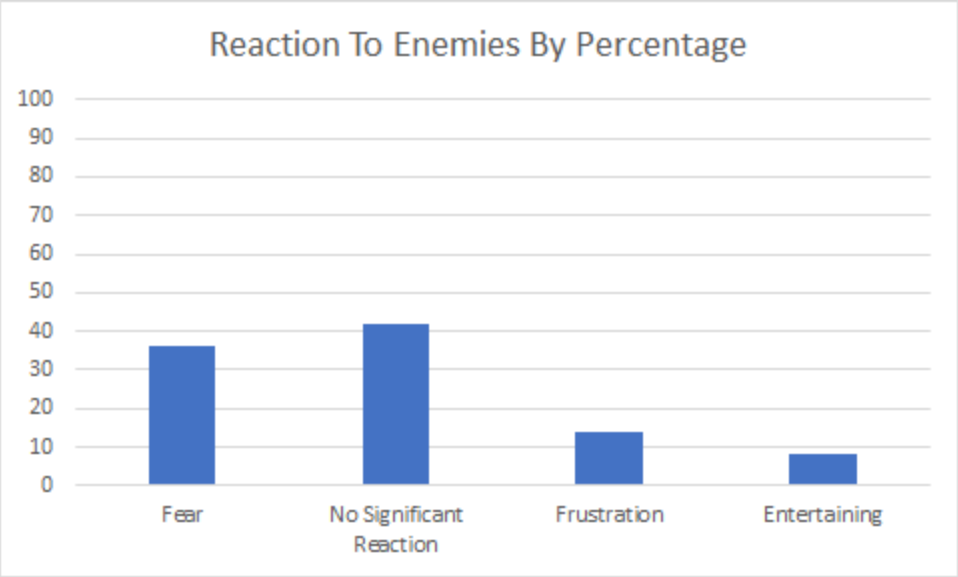
From our observations, we found out several insights into our mechanics. One thing we were testing was *staggered* versus *smooth* turning as we found some games such as Minecraft implemented staggered turning in order to attempt to reduce VR sickness. Our findings were that the overwhelming majority preferred smooth turning. Figure 49 is a bar chart showing the results of the players being asked what turning scheme they preferred.



[Figure 49: Graph of user turn preferences.]

These results indicated to us that the familiarity of smooth turning is valued more by the players than our specialized staggered turning. The staggered turning was implemented with the idea of potentially reducing VR sickness by preventing excess movement in peripheral vision. From user commentary we found that one reason for this is that several players felt the staggered turning was too similar to lag to be comfortable.

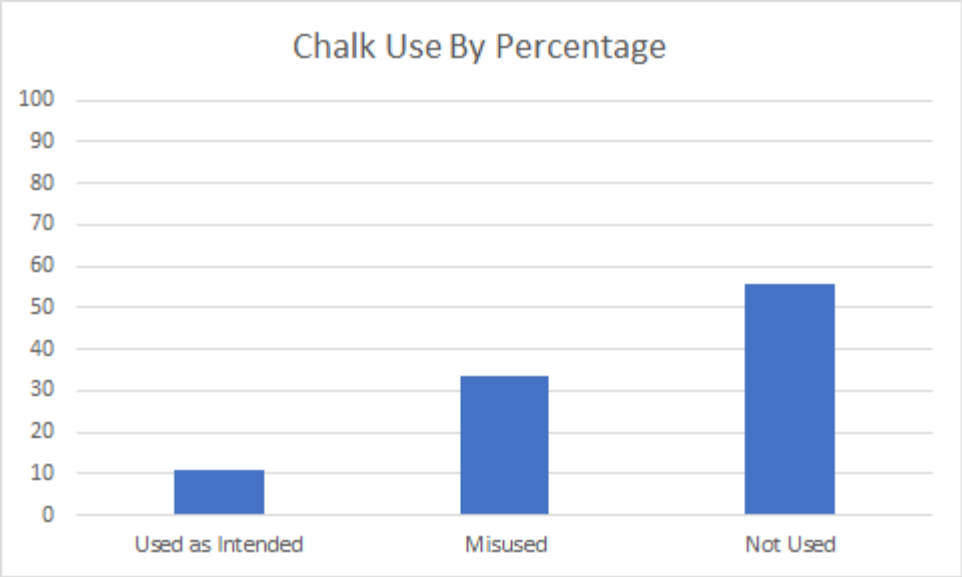
One of the main aspects we were investigating was the reaction to enemies that the players had. Figure 50 is a representation of the responses that were given to us by the players. Of those that had a significant reaction the majority had experienced some measure of fear of the enemies as we intended. Unfortunately, the portion of players did not express or indicate a significant reaction to enemies was larger than we would have liked.



[Figure 50: Graph of user reactions to enemies.]

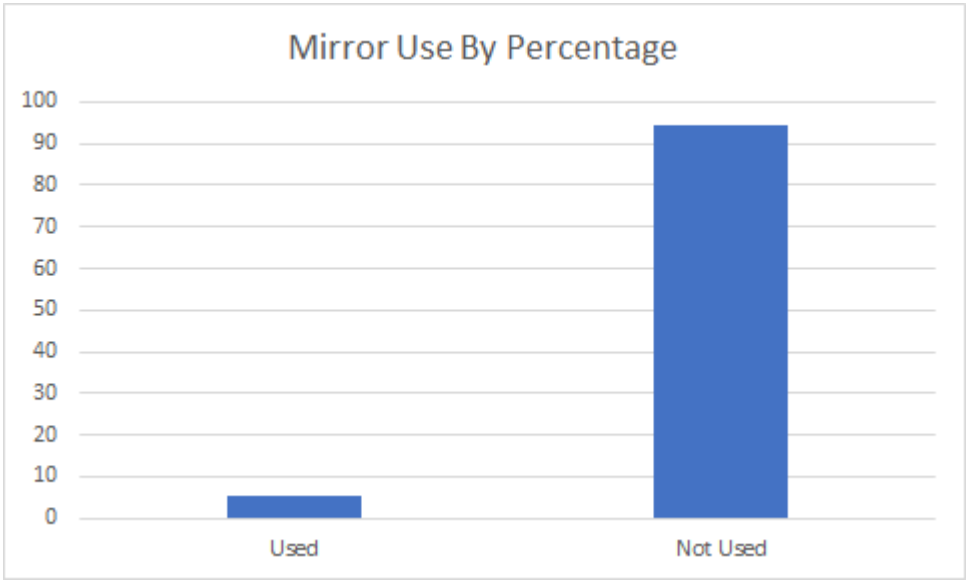
These results tell us while we have not completely reached our goals we are on the correct course to reach them with further development and implementation of our game. Since the main section of players who indicated an emotion indicated that it was fear we believe that our gameplay should remain largely the same and the main aspect we need to implement to completely reach our goal is sound, as sound is a very prominent feature in horror games.

We had four main mechanics of how the player could interact with the world around them beyond simply moving. These were chalk, a mirror, a compass, and ofuda. Figures 51 through 54 correspond to how the players used our items in game. We wanted to know if the players were using our items as intended and our results were not encouraging. For chalk we observed that the majority of people either did not use chalk or used it to simply amuse themselves. We believe this largely to be due to the short amount of time the players had access to the game as well the smaller size of the mazes they were playing.



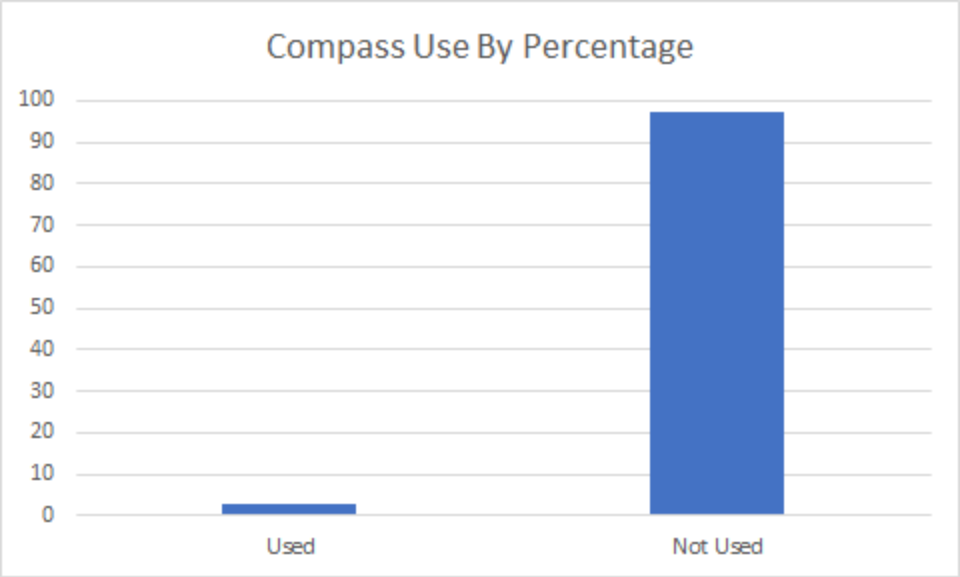
[Figure 51: Graph of user chalk usage.]

For the mirror the overwhelming majority of the testers forgot the mirror existed and thus never used it. We are unsure why the testers forgot the mirror existed as we pointed out the mirror's existence at the beginning and it was attached to their hand, one possible fix we brainstormed was to potentially make the mirror glow.



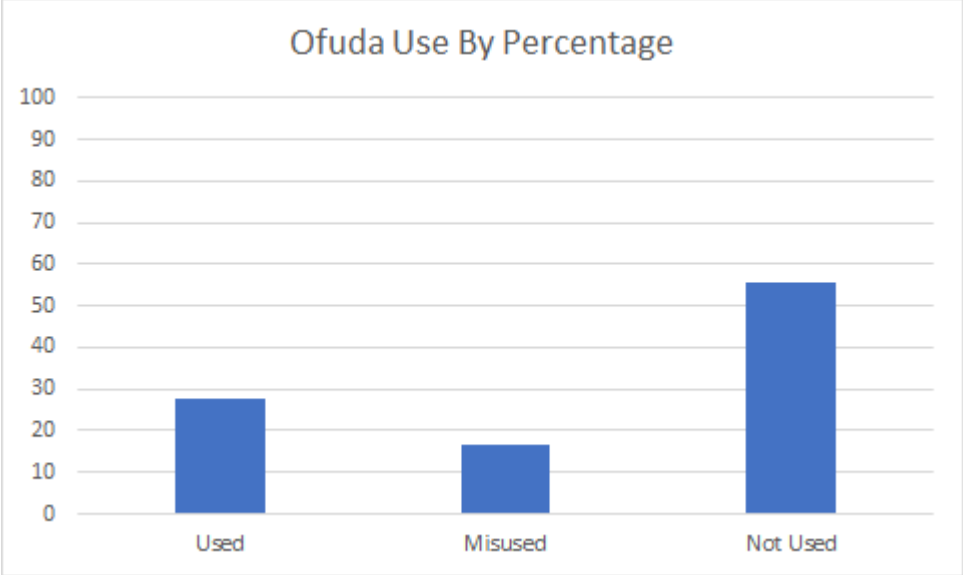
[Figure 52: Graph of user mirror usage.]

The compass was slightly worse in this regard as it was also forgotten but by slightly more of the testers and we considered the same potential solution as the mirror.



[Figure 53: Graph of user compass usage.]

The ofuda had the best results but were still not ideal. A significant amount of the testers remembered the ofuda existed but more still forgot that the ofuda existed. Several testers also wasted all of their ofuda and simply threw them on the floor and didn't pick them back up.

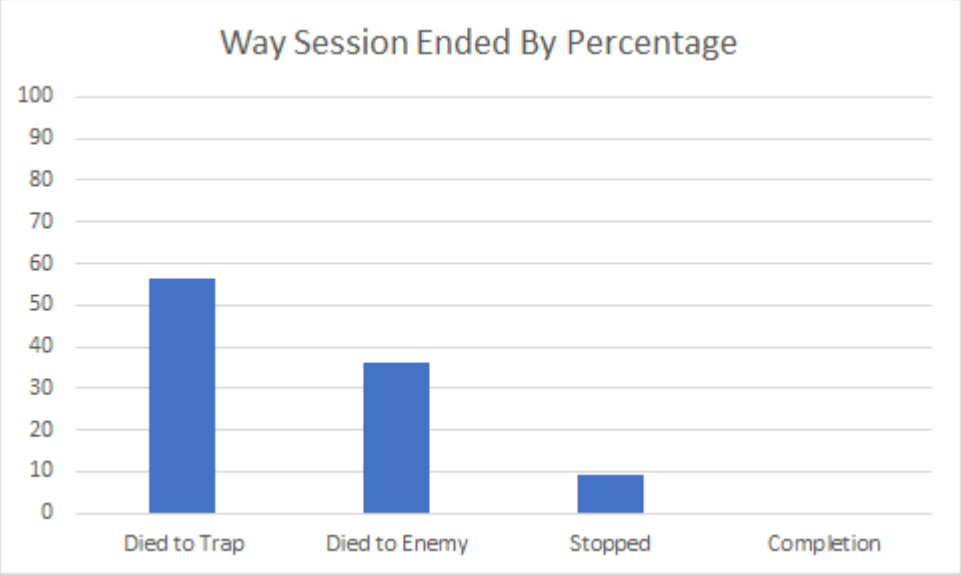


[Figure 54: Graph of user ofuda usage.]

These results tell us that our tutorial needs to be more explicit in informing the player of their options and when they should be used. The players largely did not use items and when the players did use the items it was common for them to misuse the items. From this we can conclude that the players did not know how to use the items properly. The purpose of our tutorial

is to teach the player how to play the game so the tutorial needs to be refined with these results in mind. Additionally, these results tell us that the players may have believed that they did not need to use the items in order to navigate the maze. As a team we came up with the possible solution of having instructions for the player left as chalk markings on the walls of the tutorial. This solution allows us to make use of an aspect of the game we were already working on and allows us to be explicit in game without having to be there in person. This in combination with making items more noticeable should fix the problem of the players not knowing how to use their items.

The last metric from our observations was how each individual session ended. We tracked this to see how the players interacted with the game and if they felt they needed to stop. Figure 55 is a representation of the how each of the gameplay sessions ended.



[Figure 55: Graph of sessions ended.]

The majority of the time the players died from running into a trap and a significant amount of the remaining players eventually got killed by an enemy. Only a few players got VR sick to the point where they felt the need to prematurely end the session, and no one managed to complete the game in the short time frame that they played. Of those who stopped on account of getting VR sick several cited lag spikes as a primary reason, which can be solved with further optimizations. Two of the players sabotaged themselves with wide sweeping motions of their heads, causing an extreme dissonance with what their body was experiencing and what they were seeing, and one just said they get VR sick very easily. It should be noted that gender was not tracked during our testing so no conclusions on the gender disparity of VR sickness can be drawn.

In addition to our observations, we asked our participants to fill out two surveys. From those who agreed to fill out the surveys, we compiled the results for analysis, the results of which were composed into graphs that are shown below.

In the table below, the labels on the X axis relate to the questions asked on the survey. The correlations are as follows:

Question 1: “I get motion sick easily”
Question 2: “I get motion sick often”
Question 3: “My motion sickness is severe”
Question 4: “The controls were simple and easy to use”
Question 5: “I got motion sick while playing the game”

	Question 1	Question 2	Question 3	Question 4	Question 5
Mean	3.75	2.57	2.61	7.46	3.96
Standard Deviation	2.27	1.62	1.79	2.19	2.81

[Table 5: Showing the standard deviation and mean of answers for five questions. From this point on red indicates a lower number and green indicates higher.]

The above tables display the results for some of the questions asked in both our pre and post-game surveys, as well as the mean and standard deviation of these values for each question. While it is true that question five is worded as “I got motion sick while playing the game,” such a question can only have two answers: “yes” or “no.” As such, the number between 1 and 10 which the testers used to answer this question is implied to measure the severity of the motion sickness experienced. In this case 1 indicates they did not get sick whereas 10 indicates they got very sick. In addition, it should be noted that, while VR and motion sickness are different, they are similar in the sense that both types of sickness are related to people perceiving themselves as moving, even though their bodies are not doing anything. As such, it makes sense to look at a person’s history of motion sickness when also looking into their experiences with VR sickness.

With both of these facts in mind, the average value for the severity of motion sickness that our players tend to have outside of playing our game was lower than the average level of sickness experienced within our game (comparing questions 1 and 5), which could indicate that our game has a tendency to make people more motion sick than they normally would in other scenarios. The level of motion sickness experienced while playing our game also had the largest standard deviation compared to the values of the other questions, meaning that there was greater variation in regards to the players feeling sick. This would suggest that, even though our game

does not consistently cause severe motion sickness, it has the potential to make people more motion sick than other causes, such as cars and boats. This suggests that overall, we failed to develop any means of mitigating VR sickness more than the average VR game, like we had hoped.

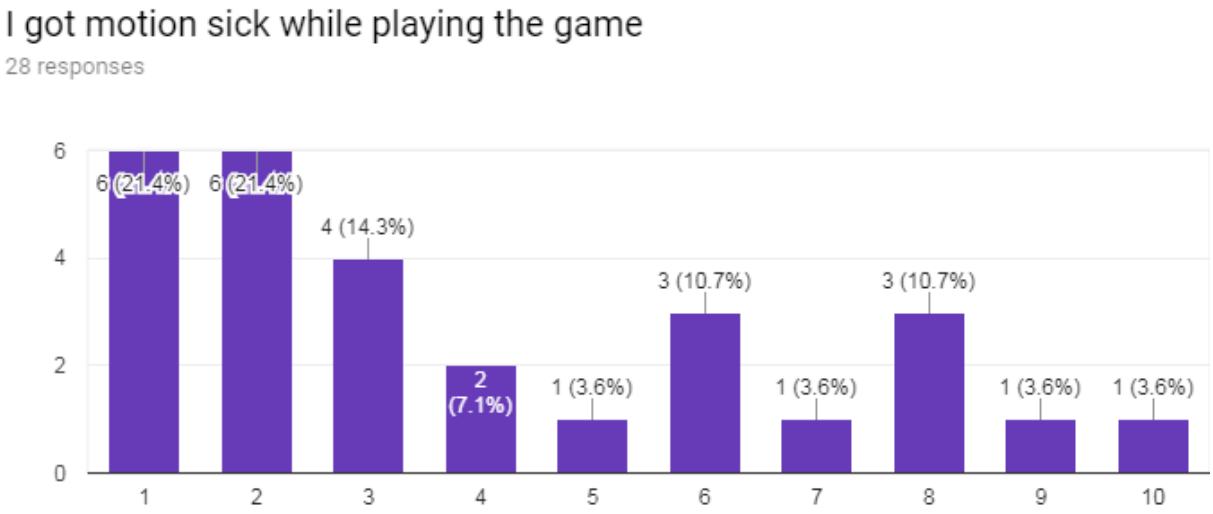
In the table below, Question 1 - Question 5 refer to the same questions mentioned in the previous table.

	Question 1	Question 2	Question 3	Question 4	Question 5
Question 1	1				
Question 2	0.664	1			
Question 3	0.639	0.386	1		
Question 4	0.114	0.006	0.342	1	
Question 5	0.359	0.241	0.292	0.238	1

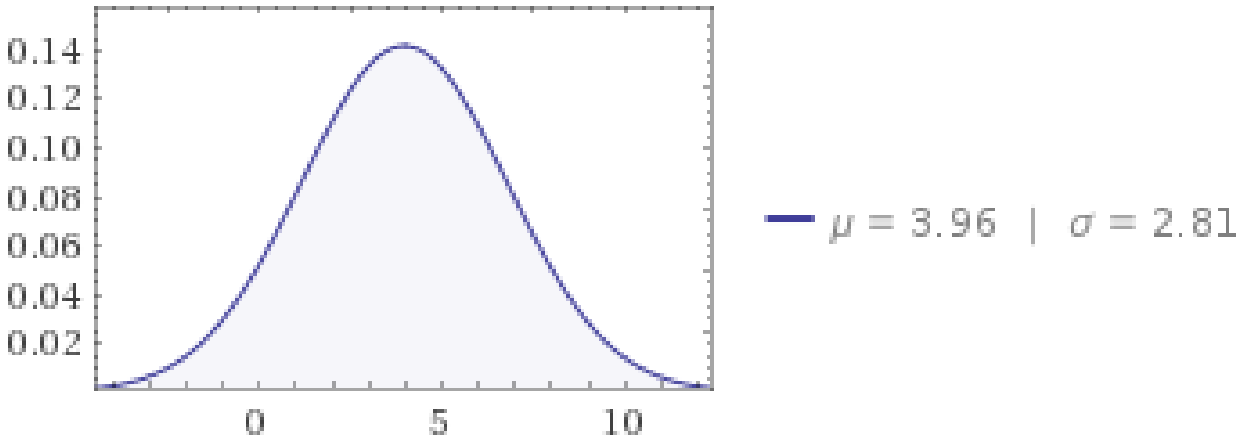
[Table 6: Correlation matrix for survey results.]

The above correlation matrix shows the relationship between the different variables portrayed in the table shown in Appendix A. The only variables with a correlation to each other that is greater than .5 are how often a player gets motion sick outside of our game, the severity of that sickness, and how easily a person gets motion sick in general, with the correlation between whether or not a player actually got sick playing our game, and any of the variables tested on never exceeding 0.36. This low correlation value seems to indicate that how sick someone gets playing our VR game is not necessarily determined by the player's past history of motion sickness.

Having looked at the possible significance of the overall values of the survey results, as well as the strength of the correlation between the questions of these answers, it is also important to look at the results of the individual questions. Below, there is a bar graph showing how testers answered question 5, as well as a bell curve created from the results.



[Figure 56: Bar graph showing Player's response regarding motion sickness]



[Figure 57: Bell Curve based on above results]

The bar graph above shows that more than half of the players rated their VR sickness as being either a three or lower, and while there were still some who rated there sickness as being fairly high, the average and standard deviation of this value were both low. This implies that if we were to test on a larger scale, the number people who experienced severe motion sickness would be small. This is made visible in the bell curve above: comparing the area under the curve for values eight and higher to the area of the rest of the curve helps give an idea of how small the proportion of people who would experience serious motion sickness from our game would be.

4.7 Conclusions and Moving Forward

From our results we can construct a clear plan of how further development of the game would progress. This plan would incorporate pre-existing ideas of what should be done with the problems revealed through our play testing sessions. There are six key areas we believe we need to improve upon:

1. Making items more attention grabbing.
2. Fully implementing text on the walls and using it to improve the tutorial (help players know how to use items)
3. Improve VR controls (mitigate VR sickness)
4. Creating and implementing VR specific menus.
5. Finish implementing session reports
6. Acquisition and implementation of additional art and sound assets.

The first two points are directly affected by our results from testing. Through adding a glowing effect to the items in the game we would make it easier for the player to find items and remember that he or she is holding the mirror and compass. We hope that by easing the mental load in this area we will allow for the players to make better use of the items in the game. One aspect of the game we did not manage to fully implement was pre-existing chalk drawings on walls. The original idea was to have them be messages from "people" who tried the maze prior to the player. Once implemented it would be trivial to adjust these messages to add clear instructions to the tutorial.

While making changes based on player feedback is important to our goals, we also have our own ideas for what to do in order help improve our game. Moving forward, we would like to continue to experiment with our VR controls. An idea we had was dimming the sides of the screen during quick movement in order to reduce the movement in the peripheral vision. Rubin describes this as a "tunneling approach where you restrict the viewpoint to get rid of the discomfort during movement but then broaden out to get all the benefits of VR when you're still" (PC Gamer 2016). One such aspect we wished to finish was VR specific menus, as described in section 3.6, that would generally improve the experience of playing our game with a VR setup. Furthermore, we would like to add the session reports functionality mentioned earlier in the paper. This would allow us to further understand what design choices helped improve the game experience and what choices did not, and make further changes accordingly. The last major aspect of our game that we wished to finish was the full implementation of art and sound assets. Sound design in particular is a major component of inducing emotions in players and as such implementing sound into our game would greatly help in achieving our experience goals.

5. Post-Mortem

5.1 What Went Right

After figuring out what our game idea was going to be, one of the first tasks we set out to do was find a way to randomly generate a maze. We worked together to look at various algorithms that could be used to achieve this goal. We found working algorithms relatively quickly, and as one of our team members already had experience in building randomly-generated mazes, the time it took to implement our algorithm of choice took little time. After that, time was spent figuring out how to correctly divide the maze up into sections, as many attempts would often create two massive sections, with one section that was only one square large. These problems were fixed relatively quickly, however, and after that, developing a method for spawning enemies across the maze was finished quickly as well. The maze-like design of the levels served its process, as players often got lost. And after some tweaking, the enemies would only spawn in places where players would have enough time to get their bearings before being attacked.

Once the maze and all of its components were generated, the next step was to find a way to have only the section the player was currently in spawn. This would cut down on the strain put on the computer, and as such help make the game run faster. A good deal of time was spent refactoring our code, but once that was done, implementing the spawning feature became easy, with the only real challenge coming from having to deal with the player teleporting from one floor to the next, which was somewhat difficult due to the floor they were teleporting to not yet existing. This problem, however, was also solved relatively quickly.

While many players did not use the chalk, the players who did, used it extensively. Players seemed to enjoy writing on the walls and treated it like a drawing simulator. Some players drew intricate pictures on the walls of the maze. Despite the many technical challenges of implementing drawing chalk on surfaces, the end result works well and feels natural.

Working with VR comes with its own set of challenges. The common approaches did not fit our game well. Even with our locomotion design limitations, our controls were well received. The majority of players found the controls easy to use and quickly grasped them. On average players did not experience high levels of motion sickness.

Once of the best things went right was that the majority of play testers indicated that they enjoyed our game. As one of the main objectives of games is creating enjoyment in the user, this result is exactly what we wanted to get. That our game was rated so highly while in an unfinished state is a good indication that we were designing a good game. Alongside this we found that a significant amount of our playerbase felt the emotions we intended them to with the

A.I. The Oni was found to be scary and threatening whereas the Taka Nyudo was found to be creepy in those that managed to reach the portion of the game where Taka Nyudo started appearing.

From the outset of our project we knew it was a good idea to split up the work that needed to be done. This was done in order to make sure we had a realistic chance of creating a working product in the time period we had available. We split up the work according to our experiences and skills. Corey Dixon as our most experienced software engineer handled many of the technical aspects of our game such as the Virtual Reality aspect as well as creating our game's visual style. Patrick Malone as our most prominent mathematician handled the creation of the maze generation algorithms and assisted with other aspects of the game. William Kelley as our Artificial Intelligence expert handled researching into Japanese lore as well as the creation and implementation of the Artificial Intelligence in our game.

5.2 What Went Wrong

One of the things that went wrong with our project is that until the later half of the project we were too optimistic about getting the project done on time. As a result of this we spent too much time thinking about adding additional features and trying to implement them when the main aspects of the game were not done at the time. Had we focused harder on the main aspects of the game we could have completed the project in a timelier fashion.

Continuing on the thought of time spent poorly, our group came to the unanimous conclusion that the side project we were asked to do while at the project site was poorly planned. Said side project was considerably time consuming and added very little to the project beyond providing more opportunities to interact with local students. Unfortunately, there were major communication issues as a result of language barriers which greatly diminished any benefits the side project could have had.

One significant aspect of our game that did not have good results was the implementation of items. The first issue was that the majority of the playerbase did not remember the existence of the mirror and compass items after having both explicitly pointed out to them. The second issue is that even when the players knew about the items they had, the players had a tendency to waste or misuse them. This highlighted the issue that we did not properly teach the players about their items and how to use them. The solution we agreed upon was to retool the appearance of items and improve the tutorial.

Arguably one of the biggest problems we ran into during the project occurred during our second round of playtesting. We had a system set up to record important analytics data within the game, as mentioned earlier. Unfortunately, upon finishing our testing, we saw that most of the

data that should have been recorded was nowhere to be found. This meant we could not perform any sort of data analysis on our telemetry data. Considering the amount of time we put into creating the tools to record the data and testing so many people, this was a huge blow to our project.

A significant issue that went wrong during our project was debugging. The three major issues were excessive amounts of debug messages, nigh-endless loops of discovering and fixing bugs, and excessive queries that bogged down our game. The first issue, excessive debug messages, was the result of not knowing precisely were issues would arise leading to a clutter of unnecessary checks which drowned out the actually important messages. Over the course of the project this issue was refined and made less of an issue. The second issue, loops of discovering and fixing bugs, was a result of a targeted individual bug at a time approach to debugging instead of a holistic approach. Due to this approach individual bugs were discovered and fixed. After fixing that bug the individual would check specifically to see if that bug was fixed and then reported the game to be working only for another member of the team to discover another bug. That teammate was approached by the other two and a system was put in place to mitigate the issue. The third issue, excessive queries, arose due to a bug causing our A.I. to change states every single game loop. These state changes would then be logged to our database which significantly slowed down our game. The logging of state changes was an intentional design decision but was made with the idea that state changes would occur on a much rarer basis. In order to fix this, we created a queue to handle non-time sensitive queries to our database that would be dump the queries into a separate file if the queue became too long. The queue would allow non-time sensitive queries to be processed when the game was paused or finished and the file would allow us to perform the analytics at a later time in the case where the processing would cause lag.

A less significant issue that arose during the project was that the main software we were using, Unity, updated the version available. One of the members of our team decided to update the version we were using for the game which resulted in instabilities that had to be fixed. The main issue was that nothing in the patch notes for the update indicated what could have caused said instabilities which lead to time being spent to restore the game to working order.

The last major issue we had concerned our attempts to reduce VR sickness in our players. In order to make controlling the player in VR more comfortable we researched possible methods of doing so and came across segmented turning as a possible option. As a result of this, we implemented segmented turning to be our default turning method. The issue arose when during testing it was revealed that not only did the segmented turning not help, it actively discomfited the players. As segmented turning was one of our main design choice to reduce VR sickness, this was severely disheartening.

5.3 What Did We Learn

The first thing we learned is that we needed to start player testing as opposed to bug testing sooner. This would have revealed the issue we had with our items even if the game had not been completely working when the players tested it. If we had discovered this earlier we could have compensated for the issue.

One incredibly important lesson we learned was to always check to make sure that whatever data needs to be recorded is being recorded. If we had checked to make sure our data existed during testing, we may have found the problem early and found some way to fix it, which means we could have had our data intact.

The third thing we learned is how to properly distinguish what sets of information are important when debugging. Learning how to refine which debug messages are needed allowed for the whole debugging process to go much smoother and faster.

The fourth thing we learned is that the player needs to be explicitly taught how to interact with the game world if we expect them to learn how to do so in a short time frame. The two main reasons we had issues with players using items correctly were that we never explicitly pointed out when and where to use items and that the players did not have enough time with the game to experiment with the items. As such if we want players to be capable of using items in a shorter time frame we needed to improve the tutorial.

The last thing we learned is that in order to improve a given player's VR experience we had to tailor it to specifically what they are doing. This lead to decisions like not implementing teleporting like most VR games due to the maze aspect of ours.

References

3dhaupt: "Wolf Rigged and Game Ready 3d model." Wolf Rigged and Game Ready - 3d model - .3ds, .obj, .dae, .blend, .fbx. December 13, 2015. Accessed December 8, 2017. https://free3d.com/3d-model/wolf-rigged-and-game-ready-42808.html.

"A UX designer's guide to combat VR sickness." RealityShift. March 23, 2016. Accessed December 12, 2017. http://realityshift.io/blog/a-ux-designers-guide-to-combat-vr-sickness.

Blain, Louise. "How the battle to stop VR sickness will change game development forever." Gamesradar. October 14, 2016. Accessed December 12, 2017. http://www.gamesradar.com/how-the-battle-to-stop-vr-sickness-will-change-game-development-forever/.

Chamberlain, B. H. *The Kojiki: Records of ancient matters*. Rutland, Vt: C.E. Tuttle Co, 1982

D, John: Ofuda (talisman). July 30, 2011 Accessed December 12, 2017. http://www.greenshinto.com/wp/2011/07/30/ofuda/.

“Google Cardboard – Google Store”. Google. Accessed January 12, 2018. https://store.google.com/product/google_cardboard

Hastings, Erin J., Jaruwan Mesit, and Ratan K. Guha. "Optimization of large-scale, real-time simulations by spatial hashing." In *Proc. 2005 Summer Computer Simulation Conference*, vol. 37, no. 4, pp. 9-17. 2005.

Kumamoto, Hiromu, writer. *Yami Shibai*. Directed by Tomoya Takashima. Produced by Naoko Kunisada and Nobuyuki Hosoya. TV Tokyo, AT-X. July 14, 2013.

Mason, Betsy. "Virtual reality has a motion sickness problem." Science News. March 08, 2017. Accessed December 12, 2017. https://www.sciencenews.org/article/virtual-reality-has-motion-sickness-problem.

Mazza, Laurie: Created Oni model April 2017.

Meyer, Matthew: "Yokai.com." Yokai.com. 2013. Accessed December 12, 2017. http://yokai.com/.

Melvinsalcedo169 : "Old man 3d model." Old man - 3d model - .obj. August 19, 2016.

Accessed December 8, 2017. https://free3d.com/3d-model/old-man-50058.html.

Morita, Shuhei, writer. *Kakurenbo: Hide & Seek*. March 31, 2005.

"Oculus Rift | Oculus". Oculus Rift, accessed January 12, 2018. https://www.oculus.com/rift/

PCGamer. "The quest to solve VR's biggest problem: walking around | PC Gamer". PC Gamer. Accessed December 11, 2017. http://www.pcgamer.com/the-quest-to-solve-vrs-biggest-problem-walking-around/

Pranckevicius, Aras. "MirrorReflection4 – Unity Community Wiki". Unity 3D. Accessed December 11, 2017. http://wiki.unity3d.com/index.php/MirrorReflection4

Pullen, Walter D. "Think Labyrinth: Maze Algorithms." astrolog.org. Last edited Nov 20, 2015. http://www.astrolog.org/labyrnth/algrithm.htm#perfect.

Riviere, Alex: "How Do You Design Your VR Game Around Motion Sickness Constraints." Gamasutra. Article. June 20, 2017. Accessed December 12, 2017. https://www.gamasutra.com/blogs/AlexRiviere/20170620/300275/How_Do_You_Design_Your_VR_Game_Around_Motion_Sickness_Constraints.php.

Sebastian, Rose. Visual Music Systems. Accessed December 12, 2017. http://www.visualmusicsystems.com/blog/motioninvr.htm.

stym. "Game Assets - Modular Dungeon [Part 1]". Youtube video, 24:02. Posted August, 2014. https://www.youtube.com/watch?v=2s0mANzeuUk

"The Nihon Shoki." The Nihon Shoki Wiki - Home - shoki. Compiled by Prince Toneri in 720 A.D. Page last edited July 12, 2012. Accessed December 12, 2017. http://nihonshoki.wikidot.com/.

"Unity – Movement in VR". Unity 3D. Accessed December 11, 2017 https://unity3d.com/learn/tutorials/topics/virtual-reality/movement-vr

"VIVE | VIVE Virtual Reality System". Vive. Accessed January 12, 2018. https://www.vive.com/us/product/vive-virtual-reality-system/

Appendices

Appenix A: Survey results

	Motion Sick Easily	Motion Sick Often	Motion Sickness Severity	Ease of Controls	Got Motion Sick
Player 1	8	6	6	8	10
Player 2	5	1	2	5	8
Player 3	4	2	3	8	8
Player 4	4	2	6	8	4
Player 5	5	2	3	8	7
Player 6	3	3	2	7	5
Player 7	4	2	3	10	8
Player 8	5	2	3	8	2
Player 9	6	5	3	10	6
Player 10	7	1	4	8	1
Player 11	1	1	1	8	6
Player 12	1	4	1	7	3
Player 13	2	2	1	4	2
Player 14	3	1	2	5	3

Player 15	5	5	0	1	3
Player 16	2	2	4	7	3
Player 17	1	1	1	8	2
Player 18	7	5	7	9	2
Player 19	3	3	3	8	2
Player 20	1	1	1	10	6
Player 21	6	4	4	10	9
Player 22	2	2	2	5	1
Player 23	2	2	1	10	1
Player 24	7	5	5	7	4
Player 25	2	1	2	10	1
Player 26	1	1	1	4	1
Player 27	1	1	1	8	1
Player 28	7	5	1	8	2

[Figure 58: Survey Results.]