Yōkai no Kōgeki

A project report on the production and execution of a new tower defense game

October 4th, 2018

Team Members
Jason Abel, *jabel@wpi.edu*
Adam Moran, *asmoran@wpi.edu*
Will Suriner, *wesuriner@wpi.edu*

WPI Advisors
Gillian Smith, *gmsmith@wpi.edu*

Collaborators
Toyonaka University; Takemura Labs; Cybermedia Department

Worcester Polytechnic Institute
This report is submitted to Worcester Polytechnic Institute faculty in partial fulfillment of the Degree of Bachelor of Science.

Abstract

Yōkai no Kōgeki is a VR tower defense game set in feudal Japan where the player takes on the role of a commander preventing Yōkai from invading. Three objectives were identified for experience goals. Users should feel immersed in the setting. Users should have a strategic challenge while fending off Yōkai. Users should feel like a leader that is in control of his or her territory. This report details the design process of creating a game that achieves these objectives. The report goes over the design and gameplay of the game, the technical implementation, the art direction, and the testing of the game. While the game could be improved, Yōkai no Kōgeki was successful in fulfilling the objectives laid out prior to implementation.

Acknowledgements

We would like to thank Osaka University's Takemura Labs for providing us with a space to work and our peers to help test our game. It is thanks to everyone part of the lab and their input and help that the project expanded to virtual reality.

Specifically, we would like to thank Takemura-sensei for providing WPI with the opportunity to work at Takemura Labs. Additionally, we would like to thank Ae-sensei and Jason-sensei for taking time out of their schedule to help get us acclimated to out new environment in Japan, as well as their overall cordiality.

We would also like to thank our advisor, Professor Gillian Smith, for helping guide the development of this project. We would like to thank the project center director, Professor Jennifer DeWinter; without her forging connections with Osaka University and preparing us for our time in Japan, this project would never have come to fruition.

Lastly, we would like to thank the other WPI students that came to Japan with us for helping us out with our project and playtesting our game. Without their help we would not have the finished product that we have now.

Table of Contents

List of Figures

List of Code Listings

Chapter 1: Introduction

Yōkai no Kōgeki is a virtual reality tower defense game set in feudal Japan where the player takes on the role of a commander in a castle to prevent Yōkai from invading said castle. As the game goes on, the waves of enemies increase in number and become progressively stronger while the map also becomes more complex, with more places for enemies to appear from and alternate paths for them to take. The game features a dual world mechanic, with two separate maps that the enemies can traverse and the player can set defenses in. This dual world mechanic, combined with the inherently limiting view of a first-person perspective on a battlefield, forces the player to always consider where they focus their attention. Due to the time constraints of the project, only three types of towers and three types of enemies were developed; however, this should provide more than enough challenge and strategic possibilities for the player, especially as the map opens up as the game progresses.

Through the decisions to make this tower defense game with a Japanese influence, three objectives were identified for user experience goals. First, users should have a strategic challenge while fighting against Yōkai. Users should have to think about what they are doing and why. Throughout fighting Yōkai, they should experience unexpected changes to the battlefield and enemy movement they may not have foreseen by the Yōkai. Second, users should feel immersed in the setting while at the top of a castle looking at the entire battlefield. They should be able to look out of the castle and see the entirety of the battlefield through which they can control their units. This should get the user immersed in the scene and give the feeling like they are in a castle commanding an army. Lastly, feeling like a commander in a Japanese castle should be the final experience gained. User should fight against Yōkai by controlling their units to fight for them. Additionally, through the strategic challenges provided, the user should get to understand the

difficulties of Japanese commanders who had to decide where to send units and how to prioritize the development of fortifications.

This report serves to explain the design process of the game, including background information, overall design and gameplay, software implementation, art development, the process and results of testing, and a post-mortem. The background section discusses tower defense as a gameplay genre, the Japanese influences that guided our development, and locations in Japan that were visited to further guide our design. The design and gameplay section focuses on explaining the intended player experience, the overview of the gameplay, a list of user stories considered during the development of the game, the paper prototype that was developed to test the game concept before implementation, and the design of the map. The implementation and technology section explains the software used to develop the game, the base model that the game was built off of, implementation of virtual reality, and all the changes made to the base model to fit the intended design, as well as the development of the website to view the art assets.

Following these sections of the report, the art section provides information about the background of the art style used and how it influenced the aesthetics of the game, the reference art and sketching that served as the start of art asset development, the modelling and texture work done for the art assets, and the animation of the assets to bring the game to life, the implementation of art assets into the game engine, and the creation of the game environment. The testing section documents the process of testing the project to ensure that it conforms to the intended player experience, including what the team was looking to get out of the testing, a description of the survey used after the playtesting, the methodology followed during testing, the results of the testing, and the changes made to the project as a direct result of the testing. The

postmortem section concludes the report, providing information on what went right, what went wrong, what the team learned, and future developments for the project.

Chapter 2: Background

Chapter 2 goes into detail about the background of the project. It talks about the background of tower defense games, the influences Japanese culture had on the game and it's background, and lastly, real world locations used as references for the game.

2.1 Tower Defense

In a tower defense game, the player's goal is to defend territory, possession, or destination from waves of enemies by obstructing said enemies with defensive installations that the player places at key strategic locations. The defensive installations can serve many distinct purposes, including blocking an enemy's path, slowing the enemy's progression down a path, or outright attacking enemies; though some games include support installations to boost effectiveness, most installations serve to impede the invading enemies in some way. Tower defense games are considered a subcategory of strategy games. In strategy games, the player commands units on a battlefield to take out an opposing force, be it a computer opponent or another player; the key difference with tower defense is that the player cannot micromanage units and can only build installations that automatically attack incoming enemies. Tower defense games are typically played in real time, though some choose to break up the action by having distinct 'phases' for building towers and letting the invasion unfold.

As described by other reports, tower defense games are seen as their own genre while additionally being under the real-time strategy genre. Players are scored based on how many "waves" the player can last before either the player finishes the round or the enemy "kills" the player. The players are met with a predefined map which enemies travel along a path and towers can be built. In the process, the player must strategize effectively; in order for the towers to

operate at peak performance, they must be placed in appropriate spots on the map. Knowing when and where to place more towers or to upgrade the towers currently on the map is another vital skill within the genre [28, 29].

While tower defense games share these properties, the possibilities presented by the template are surprisingly varied; two games under the tower defense genre can vary dramatically. Both *Plants vs Zombies* and *Orcs Must Die* are considered tower defense games, yet are drastically different in terms of player perspective, map design, mechanics of the defenses, and the power that the player has within the game world. *Plants vs Zombies* has simple one-way maps with multiple lanes with which the player places turrets that shoot forward in some way; the player sees the entire map from a fixed perspective and can only place defenses [11]. *Orcs Must Die* has more complex maps with branching paths and multiple spawn points, with the player placing traps that have specific ranges and specific behaviors, some of which are placed on the ground and others which are placed on the walls or even the ceiling; the player controls a hero in the game world that can also aid in the fight against the enemies though must interact with the world mainly from a third-person perspective of the hero [10]. Structurally, these games are very similar, involving sending stronger and more numerous waves of enemies as the game progresses with the assumption that the player has been fortifying their defenses. Mechanically, however, they are vastly different.

Within the context of this report, mechanics are defined as how the player interacts with the game, including controls and the options the player's avatar has to influence the game world, while structure is defined as how the game or game world is organized to present its challenge. In other words, the structure determines the overall genre that the game fits in regarding progression, while mechanics determines the interface that the player directly commands to

interact with the game which can influence the genre the game is defined in; it is this difference in mechanics that gives *Orcs Must Die* the more specific label of 'action tower defense' as the mechanics are more action-oriented compared to a more typical tower defense game while the structure itself is still that of a tower defense game. As a result, despite being a seemingly specific genre, the definition remains general enough to encompass games that have entirely different mechanics to achieve the same experience goal of defending some point from waves of invading monsters by setting up defenses that cost resources.

Furthermore, many games mix structures and mechanics of different genres, sometimes muddying the lines that define genres and sometimes outright creating new genres. Survival games, usually involving zombies and usually played as a shooter, bear mechanics and structures from tower defense games yet have more action-oriented mechanics and have changed the enemies' destination to the player themselves; due to their mechanics, survival games are considered a subcategory of action games despite having the resource management and defense crafting structure of strategy or tower defense games. There are also genre offshoots that use the structure and mechanics of a preexisting genre to remix it in a novel way. An offshoot of tower defense games exists where the player controls the enemy forces sent to attack defensive installations, usually called 'reverse tower defense', 'tower offense', or 'tower attack', referencing its origins as a tower defense game with inverted objectives, which was considered as a possible asymmetrical multiplayer mode but never received any further focus. Examples of reverse tower defense games include the Anomaly series and Ambush, both available on Android devices [3, 4]. These games swaps the mechanics available to the player and the enemies typical of tower defense games and inverts, yet ultimately maintains, the structure; as a result, while a more specific subgenre, this offshoot is still considered part of the tower defense genre. While

the definition of tower defense as a genre given at the beginning of this section is the one used for this report, it remains a generalization that groups a multitude of games based on their structure while generally ignoring the widely varying mechanics that each use to realize similar goals.

Tower defense games are likely popular because they feed off humans' psychological need to defend what they consider to be 'theirs' from some undefined 'other' or intrinsic fear [43]. Young children are shown to perceive some notion of ownership principles and property rights; furthermore, children between the ages of four and five are shown to value concepts of property more than adults, with experiments showing that children believe that ultimate authority to an object should go to its rightful owner as soon as possible excluding emergency situations [26]. Moreover, similarly aged children have a consistent methodology of determining ownership, primarily based on the principle of first possession, though this can be overridden with higher-level historical reasoning and whether or not an individual has had prior contact with an object in question [39]. This same principle of first possession is carried over in legal systems as well. Modern Western society uses first possession based on precedent set by the Institutes of Justinian in the 6th century. Ancient Jewish law and Inuit decision making regarding ownership of a hunted animal also used first possession to determine ownership. Again, first possession is trumped by higher-level reasoning such as what was necessary for possession (e.g. if person B captures a hunted animal that person A was pursuing, would person A's actions have ensured possession of the animal if person B wasn't there; if yes, then the animal belongs to person A regardless of first possession, if no, then the animal belongs to person B since it was person B's actions that ensured possession) [38]. This suggests that concepts of ownership are ingrained into the human psyche at a fundamental level. Human concepts of ownership and

property can be seen as an extension of common animal behaviors of claiming and defending territory or fending off scavengers from successfully hunted prey. While it is possible that human ownership and property concepts are an extension of animal territorialism, the possibility that this behavior is a result of convergent evolution, creating similar behaviors with similar functions though with different underlying mechanics or origins, cannot be ignored.

Part of what makes possessions so important to individuals is the meaning that the individuals give to their possessions; this meaning extends to how the possession represents or serves as an extension of its owner in some way, as explained by symbolic consumption theory. This theory can be extended to virtual objects, with individuals seeing their virtual representation, or avatar, as either extensions of themselves or idealized versions of themselves. Virtual products, owned by the avatars, are considered to be real entities, and users often render special meaning to them. To many users, the avatar's possessions are seen as more important to who they want to present as in a virtual space than the avatar itself; the avatar serves as a social entity for the user in the virtual world but also as an interface for the user to interact with their virtual possessions [41]. For this project, the player's castle is their possession and serves as an extension of themselves within the virtual space just as much as their own presence in virtual reality. As a result, defending a virtual abstraction of property from some outside force seeking to either claim it or destroy it is a way to simulate the desire to enforce one's natural rights without the need of a real threat to one's real possessions. This project seeks to create such a simulation within a Japanese aesthetic; as such, the player's property is a Japanese castle and the invading force are Yōkai common in Japanese mythology and folklore.

The modern form of the genre started in 1990 with an arcade game by Atari called *Rampart*, which is considered a classic of what is now known as the tower defense genre [32]. In

it, a set of standards was developed for similar games to follow: players place defenses in spots of their choosing that the game permits and must maintain the structural integrity of their territory. *Rampart* split its gameplay up into three 'phases'. In the first phase, the player chooses where their castle shall be positioned on the map and then adds cannons to the walls of the castle. In the second phase, the player has to defend their castle from incoming ships using the cannons that they build. In the third phase, the player has a limited time to repair the damage done to their castle and, if they still have time, build more cannons. While *Rampart* has the structure of a modern tower defense game, it also has mechanics that tower defense, as a genre, has shifted away from, mainly involving manually controlling the defenses; as a result, it is more accurate to describe *Rampart* as a strategy game. Nonetheless, *Rampart* played a pivotal role in setting the foundation for tower defense within the strategy genre. While tower defense games became more mainstream with the widespread adoption of the computer mouse as an input device due to added precision and freedom of movement (a trait that is shared with all strategy games), Adobe Flash, browser-based gaming, and mobile phone app stores caused a considerable boom of tower defense games due to easier development, greater accessibility, and a wider audience [8].

While *Rampart* may have been the game to set the standards for the tower defense genre, the subgenre can trace its roots back to *Space Invaders* in 1978 and *Missile Command* in 1980. Both games involved defending territory from increasingly difficult waves of aliens and missiles respectively. *Missile Command* has an extra strategic edge with resource management as the player's defensive missiles were finite and the number available decreases as each missile silo is lost [1]. Furthermore, both of these games invoke the human desire to defend one's possessions. *Space Invaders* is more broad, as the property to defend is the Earth itself. *Missile Command* opts for a more specific scope for what the player is defending; the cities that the player defends

in *Missile Command* were specifically thought of by developer Dave Theure as cities in his home state of California, but he specifically chose not to explicitly name them as he knew players would imprint their own cities onto the abstractions presented in the game. However, neither are truly tower defense games. Both lack the ability for the player to create their own defensive installations. *Space Invaders* and *Missile Command* are both considered shoot-em-up games, making them not even in the umbrella genre of strategy games. Despite this, *Missile Command* is sometimes cited as the progenitor of tower defense as a subgenre.

Other precursors to the tower defense subgenre include Game & Watch games such as *Vermin* (1980), *Fire Attack* (1982), *Green House* (1982), and *Safebuster* (1988), which all involve the player stopping some hazardous object from reaching and damaging its intended destination, generally in increasingly difficult waves [6, 7, 14, 15, 21]. While there are other Game & Watch games with similar structure and mechanics, these four games were chosen specifically for the context of the mechanics and structure as actively defending something. The Game & Watch series displayed set images on an LCD display, something that designer Gunpei Yokoi specifically pushed for as the technology was widely available and cheap, leading the games developed to focus on mechanics and the consoles to be portable and affordable compared to contemporary devices that used LED displays and required larger batteries [27]. Game & Watch consoles had rudimentary logic to control collision events, step time, and enemy behavior, as well as an internal clock and alarm for general usability [9]. As a result, many games ultimately boiled down to predicting the location and speed of incoming game objects and either intercepting or avoiding it based on the context of the game; defending some area from incoming threats was one such application of this rudimentary logic.

Part of why tower defense is such a popular genre is the simplicity of its formula and the innovations that the individual games can bring to the table. Its base mechanics of automated defenses, enemies that find a path to their destination, and paths that can branch and criss-cross are easily understood and recognizable. Part of what makes tower defense innovative is the wide array of opportunities for the defenses, enemies, and paths to interact with one another. One person can make a tower defense game where there is only one path but the enemies are relentlessly powerful, requiring significant investment to keep the path secure. Another person can make a tower defense game where the map is sprawling and labyrinthine, with weaker enemies that specifically find the path of least resistance, requiring the player to spread themselves thin to cover all of their bases. One more person can make a tower defense game where there are certain enemies that can outright circumvent the paths and take a straight line to the destination. Another person can make a tower defense game where there are multiple territories to defend and enemies can arbitrarily choose which one to head to. The possibilities are seemingly endless and cover a wide spectrum, as shown in the earlier examples of *Plants vs Zombies* and *Orcs Must Die*. Tower defense games can be split up into distinct levels with different maps or made into a single map that the player must constantly defend; perhaps there's a tower defense game that has a larger world that must stay defended but the player can only defend one level at a time, requiring players to think ahead in a local and global sense. The primary gameplay draw of tower defense games is that of strategic challenge. This mainly comes in the form of resource management and thought-out placement of defenses against increasingly numerous hordes of enemies. Further strategic challenge is presented by utilizing a collection of unique enemies with differing movement speed, vitality, and even movement gimmicks such as ignoring barriers.

Looking at some more modern types of tower defense games, sometimes they do not have to be completely virtual. For example, Art of Defense is a type of tower defense game that incorporates both virtual and physical. It is a cooperative tabletop board game, allowing for more social interactions between people allowing form more social play rather than a single player tower defense game. This game also incorporates Augmented Reality to allow for a more interesting experience for the players. This just goes to show that tower defense games are always trying to be unique while still retaining their general forms [28].

As seen from the past examples of tower defense games, there are many ways to define what a tower defense game is in its entirety because of the many variations and uniqueness of each game. Thus, this project decided to keep the general goal of having the player protect their territory from incoming invaders. The player can buy, upgrade, and sell units to prevent the incoming enemies from getting to their desired destination. Where this definition will differ slightly from others is through the lore of the game, influencing the types of enemies and their properties, and the defending units and their abilities. Additionally, since one of the main goals is to provide strategic challenges, some of the game elements will differ from others.

2.2 Japanese Influences

While in Japan, one may notice that the country is home to a very large amount of ancient shrines, castles, and temples dedicated to ancient Shinto gods. Architecture of the historic buildings inspired the unique level design innovations to portray Japanese culture through the towers and enemies. This project attempts to make a different version of this subgenre with a unique spin on the setting, including a unique perspective for gameplay as well as new strategic challenges specific to this new perspective.

Since 300 BCE, the Shinto religion was a common practice in Japan, and many buildings and shrines were built both for worship to the gods, and prevention from the demons, known as Yōkai [33]. This tower defense game is influenced by the Shinto religion, as well as ancient architecture, and has a clear Japanese aesthetic in terms of visual design. The maps, towers, and enemies have a noticeable basis in either Japanese history or mythology. This extends to even the art style of the games, taking inspiration primarily from Ukiyo-e woodblock prints and paintings, which typically emphasized the rising pleasure sectors of the post-Sengoku period Shogunate Japan [19]. Likewise, the games setting is of that time period, either taking place at the end of the Sengoku period or shortly afterwards. Fitting with the time period, a contemporary currency, mon, was used for the in-game money system [35]. The maps themselves take heavy inspiration from Japanese temples and castles, and the towers are styled like the turrets of a Japanese castle. While the setting itself is not a direct copy of a preexisting location, it still takes heavy influence from the temples and castles of Japan.

Furthermore, the enemies that spawn to attack the castle are based off of the Yōkai found in Japanese mythology. This game features three enemy variants: the standard enemy, a fast but fragile enemy, and a slow but durable enemy. To fit the Japanese aesthetic of the game, these three classes of enemies are assigned their own Yōkai to represent them. The oni, a large and brutish demon with horns and fangs that typically represents wrath and destruction, serves as the standard enemy of the game. The Oni is also depicted to have an unusual number of eyes or fingers/toes. Some say that the oni can be of any color, but red and blue seem to be the most common among them. Other sources report that the Oni is the cause of some environmental disasters such as thunderstorms, and earthquakes. It is noted that there are many representations of Oni depending on where one is looking for said information [18, 25, 34, 36]. This project

decided to focus on aspects such as horns, fangs, skin color with red and blue used as the main colors, and a humanoid body for the Oni's appearance. While Oni are typically depicted as the 'slow but durable' type of enemy in other games with a Japanese aesthetic, such as the Red and Blue Ogres (Aka-oni and Ao-oni in Japanese) in *Ōkami* (2006) and the larger oni enemies in *Muramasa: The Demon Blade* (2009), both of which are effectively hard-hitting minibosses when encountered, it was decided that the oni is an easily recognizable Yōkai and that there is a better candidate to serve as the 'slow and durable' type of enemy [5, 13, 44]. The kamaitachi, a magical weasel with scythes for forelimbs that rides in tornadoes, was an obvious choice for a fast yet fragile enemy. However, the kamaitachi may not necessarily ride tornadoes but just strong winds. It is said that since kamaitachi move so fast, a lot of their destruction is blamed on the wind rather than the mythical creature. They are invisible to human perception due to how fast they can move. Some say that kamaitachis use magic to allow it to move incredibly fast with the wind; even without riding a tornado, the kamaitachi is still a weasel, which has a higher top speed to body size ratio than humans [17, 25, 34]. Likewise, a small mammal like a weasel would probably take less to kill than a human, much less an oni. For this project, the kamaitachi is defined similarly to many of the traditional descriptions; it is a weasel with scythes for forelimbs that can move faster than most normal creatures. There is only one slight difference in the context of this project: the kamaitachi uses its magic to ride a cloud rather than the wind to go fast. Ultimately, the nue, the Japanese equivalent to the chimaera with the head of a monkey, the body of a tanuki, the tail of a snake, and the limbs of a tiger, was chosen to be the 'slow and durable' enemy due to the destructive power that it possessed and generally being rarer in Japanese mythology than the oni [25, 34]. Unfortunately, fully implementing the art assets for

the nue was not viable in the time that the team had; its purpose was instead fulfilled by a larger recolored oni.

2.3 On-Site Location Research

While the team was in Japan, three sites were visited to provide more reference for the tower defense game. Osaka Castle was chosen because it is a great depiction of the type of castle and towers the team wanted in the game. Many Temples were visited to help the team get an understanding of traditional Torii gates as well as bell towers. Finally, a visual arts exhibit was visited to provide another potential way to implement the art in the game. Each of these locations gave a different perspective on what the game could look like as well as potential gameplay designs.

Osaka Castle was built in 1583 by Hideyoshi Toyotomi. To this day, Osaka Castle represents the power and wealth obtained by Hideyoshi. In 1615, the castle was destroyed through the Summer War of Osaka during Ieyasu Tokugawa's rein over the country. After this incident, the castle was rebuilt and destroyed multiple times. The final reconstruction of Osaka Castle was built in 1931 where it still stands today [16]. Osaka Castle and its surroundings have had a significant impact to the tower defense game. For starters, it is a perfect representation of the kind of castle the players defend in the game, though the final model was smaller and more simplified. Additionally, Osaka Castle has turrets nearby that were good references for potential tower styles used in the game.


Figure 2.1 (L) and 2.2 (R). Osaka Castle and turret near Osaka Castle respectively

There are many shrines and temples throughout Japan. Unlike the places of worship in the United States, the shrines and temples that dot Japan's otherwise modern cities are mainly outdoors. They are all preceded by a torii gate, either made of wood or stone. They typically have a small purification pool for visitors to cleanse themselves. Then, there is the main building; prayers are typically shown on cards placed on the outside of the main building. Specific shrines and temples had differing structures within them. One of the temples in Kyoto possessed a large bell; a temple near Osaka Castle was guarded by komainu, the eastern equivalent of the gargoyle.

Figure 2.3. A Komainu at a shrine near Osaka Castle

Another temple in Kyoto was primarily indoors and featured a large statue of the Buddha and many candles illuminating the statue. While the designs in the game don't directly take from preexisting locations, they do take heavy inspiration from the shrines and temples in Japan. One such inspiration was a large bell found at a temple. The design of the bell and the structure that held it served as the basis for one of the three towers in the final game. The ringing of the bell served to stun all Yōkai in its range, providing other towers more time to target the given enemy. Other inspiration that was considered but ultimately not implemented included komainu statues that come to life when sufficiently leveled up and purification pools that strengthen all offensive towers within a certain radius. Additionally, the bell tower would have done extra damage to enemies that were susceptible to loud noises, but no such enemy was implemented.

Visual Art exhibits initially might not be considered to be an initial influence in the making of a Japanese style tower defense game. One specific art exhibit is called “Dark Waves” located at the Dojima River Forum in Fukushima, Osaka. The exhibit is a collaboration between Hiroshi Senju and teamLab. The exhibit is inside of the building, covered in mirrors and projectors. The projections are playing a loop of waves crashing in a body of water. There is no light in the exhibit except for the projections and a light blue light with a spiral of sheets near the end of the room. When analysing the exhibit one can see the specific drawing style of the animation. It has thick strokes of color and the planar animation is projected in a way to make the animation absolutely seamless and never ending. This exhibit had an influence on the type of art in the game because the art should reflect the intended setting of the game to offer a sense of flow and connection between all facets of the aesthetics in the game. Finally, the exhibit also got the team immersed in Japanese art styles, helping decide on a type of Japanese art style to implement in the game.

Figure 2.4. Picture of waves inside the exhibit

Chapter 3: Design and Gameplay

Chapter 3 goes into detail about the design of the project as well as the intended gameplay that is trying to be achieved by the project. Specifically, this chapter talks about the intended player experience, a gameplay overview, as well as user stories to give examples of what some users might do while playing the game. Lastly, the paper prototype design of the game as well as the map design for the game is described.

3.1 Intended Player Experience

Players should experience four main things when interacting with the game. First, players should notice that the tower defense game is a unique take on the genre for those familiar with tower defense games. Implementing a second layer to the map that shows another path the enemies can follow is one feature that makes this game unique. This second map provides a new level of difficulty for players when setting up towers or defensive units. Moreover, the second map becomes a novel mechanic to even a completely inexperienced player as they would have had ten waves to get accustomed to just a single map and likely wouldn't be expecting to juggle two separate maps at once. Next, players should recognize the Japanese influence on the game visually via the design of the towers and the enemies. The art style was inspired from different Japanese sources that were research throughout working on the project. The game should also be very easy to learn how to play such that it is simple to understand at its most basic level, but difficult to master. One such example is implementing a tutorial for the game that shows all of the basic instructions on how to play, but the difficulty as the game progresses gets harder to beat. Lastly, the game should be enjoyable to play such that the players are entertained while playing. This is the most important part of our intended player experience. Enjoyability is

heavily related to game flow, which is related to perceived game difficulty. For the game to be fun, it must tread a fine line between overly easy and, therefore, boring and overly difficult and, therefore, frustrating. A difficulty curve allows the early parts of the game to remain easy for inexperienced players to learn the mechanics, while allowing the later parts to escalate the difficulty to provide a challenge for whoever reaches that point. If the game is not fun to play on either end of the difficulty curve, there is no reason for players to keep exploring the game to get the entire intended experience.

3.2 Gameplay Overview

3.2.1 Player Setup

The game consists of defending a castle in the center of the map from invading Yōkai. The player takes the role of a commander within the castle, observing the battlefield and ordering the development of defensive towers. However, the player also takes the perspective of the commander, literally looking down on the rest of the map from the central castle; this forces a limited perspective of the map, turning the player's attention into a resource to be managed.

Figure 3.1. Castle the players are put at the top of

Moreover, while the map itself is labyrinthine with multiple paths that the enemies can take, enemies can also travel to a second map to take alternate routes. Fortunately, the player can also transport themselves to the second map by striking a nearby gong. Again, the player's resources are split up as they must manage two maps at the same time while only having a limited first-person perspective of only one of the maps at any given time. As the commander of the castle, the player can select towers to build and drag them onto the map to build them instantly, in exchange for in-game currency, serving to protect the castle from the invading Yōkai. Furthermore, the towers can be upgraded to make them more powerful or to add extra capabilities, such as greater range, fire, poison, increased fire rate, faster projectile speed, or targeting every enemy in its range. Each enemy wave is larger than the last, and new enemies are occasionally added to the waves to add variety, forcing the player to deal with faster or more durable enemies along with the sheer number of enemies.

Just as the setting and art style are influenced by Japanese culture, so too are the in-game elements that comprise the mechanics of the game. As mentioned prior, the towers are designed like the turrets of a Japanese castle. Moreover, the weapons used by the towers are based on the weapons traditionally used during the Sengoku period. The weapons used by the implemented towers include yumi (Japanese bow), kunai, and a large bell. Other weapons that were considered but not implemented were the katana (Japanese sword), naginata (Japanese equivalent to a lance), and ono (Japanese axe), as well as more exotic weapons, such as the fukiya (blow darts), bohiya (bomb launcher), horokubiya (bombs), shuriken (radial throwing knives), and tetsubishi (caltrops) [42].

3.2.2 Enemies

The game features three types of enemies: the standard oni, the fast but fragile kamaitachi, and the slow but durable boss oni. While more enemy types were considered, three enemy types was seen as a base minimum that must be met. The standard enemy, the oni, is the most common type of enemy. It isn't particularly fast or durable, but they also are not slow or fragile. They are also numerous and the first type of enemy that the player encounters. Every five waves, the game spawns a boss oni, the second enemy type. The boss oni is a larger version of the regular oni. As such, they are more durable, only dying after a minimum of about three times the damage of an unscaled oni. The last enemy introduced, starting much later in stage 4, which will be covered in further detail, is the kamaitachi. Based on a mythological interpretation of the weasel, the kamaitachi is twice as fast as the standard oni. However, they also are much more fragile, unable to take half the damage that the oni can.

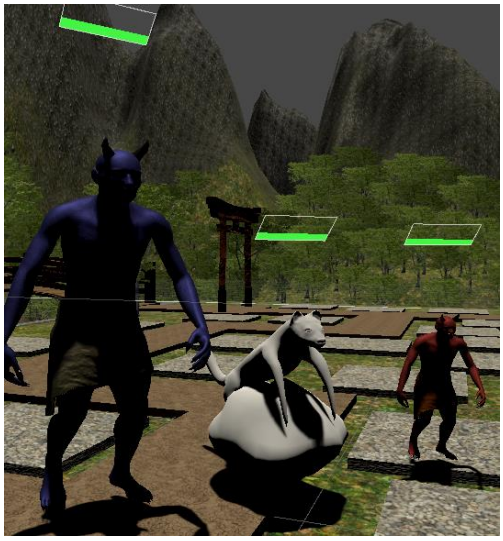


Figure 3.2. The three enemy types of the game: boss Oni, Kamaitachi and Oni respectively

In order to balance the game difficulty, every wave increases the number of enemies that spawn, as well as increases their health. This forces players to continuously improve their defenses, as the enemies also get stronger over time. Every time a new stage is introduced, the

enemies are given a significant increase in health; this continues after the final stage has been introduced. After the final map expansion, enemy scaling significantly increases every five waves to continue increasing the difficulty after the map has reached its maximum complexity. Arguably, this may make the game more difficult as there are no more new nodes introduced for the player to place towers on.

3.2.3 Map Staging

The game world consists of a sprawling maze with many paths for the invading enemies to take. However, this also includes travel between two separate maps, giving enemies a potential shortcut and the player a need to manage both in-game resources and his or her own attention to set up defenses in both maps.

Figures 3.3 (L) and 3.4 (R). Top down view of the overworld and the underworld respectively

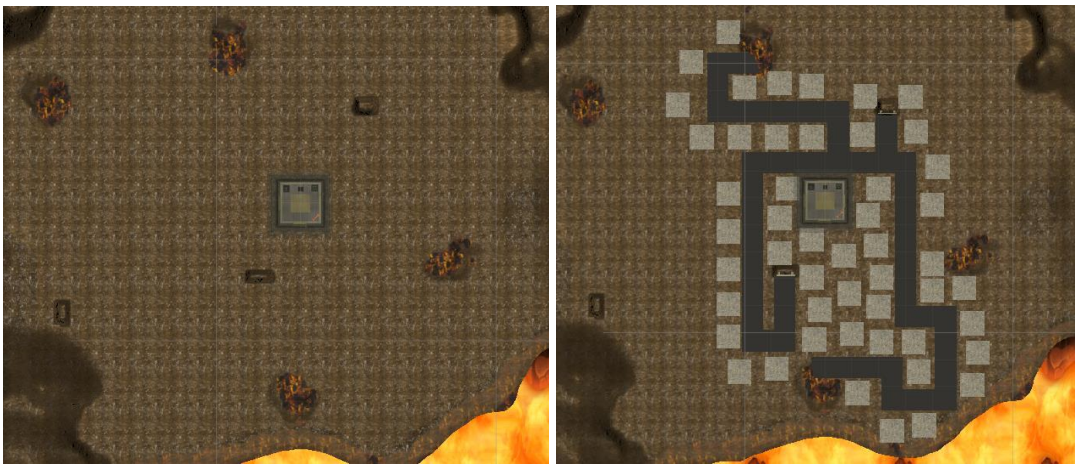
Moreover, since the game is in the perspective of a commander in the enemies' destination, the player can only see a fraction of any given part of the current map. Since there are two spawn points in each map, the player needs to be aware of where the enemies could be coming from at any given time. This provides the game with a level of strategic difficulty via limited perspective, turning the player's own attention into a valuable commodity that must be

used tactically. The dual-world map and the first-person perspective help set the game apart from others in its genre.

Figures 3.5 (L) and 3.6 (R). Limited perspective view and depiction of how much of the map the player actually sees respectively

Naturally, such a map would be overwhelming if the entirety of the map was open from the start. Instead of showing the entire map to the player right from the start, stages were implemented to ease the player into the complex map, with a new stage every five waves of enemies. Stage one is comprised of one spawn point for enemies, two different paths enemies can take, and one end goal. Starting stage two, a second spawn point is added to the game at the opposite side of the first spawn point. Again, this new spawn point has two different paths that enemies can take but converge with the paths of the first spawn point at given intersections. Stage three is where the dual-world system is introduced; pits and ladders are added to the map that allow the enemies to take even more paths to reach the end goal. At this point, the player can switch between levels by hitting a nearby gong. Starting stage four, one additional spawn point is added in the underworld to encourage setting up defenses in the second map. At stage five, only a few more pits are added. Finally in stage six, one more ladder is introduced and another spawn point is added in the underworld. While the map is complete by stage six, enemy vitality is continuously scaled every time a new stage would have become active active, further increasing the difficulty as the game goes on.

Figures 3.7 (L) and 3.8 (R). Overworld map stage 1 and overworld map stage 3 respectively



Figures 3.9 (L) and 3.10 (R). Underworld map stage 1 and underworld map stage 3 respectively

3.2.4 Player Interactions

Furthermore, there are options for how to spend the in-game resources that the player obtains. In its current state, there are three types of towers that the player can build: an archer tower that is weak but has long range, a kunai tower that is strong but has short range, and a Shōrō (bell) tower that deals a small amount of damage and stuns enemies for a brief amount of time. However, beyond that, there is the option to upgrade the towers that are already on the map. As such it is no longer just a matter of placing as many towers as possible; quality is just as important as quantity, especially when further upgrades expand the capabilities of the towers as

opposed to just increased power output. This adds an extra layer of strategy as it is no longer just about having resources; it is also important to know how to spend the resources effectively.

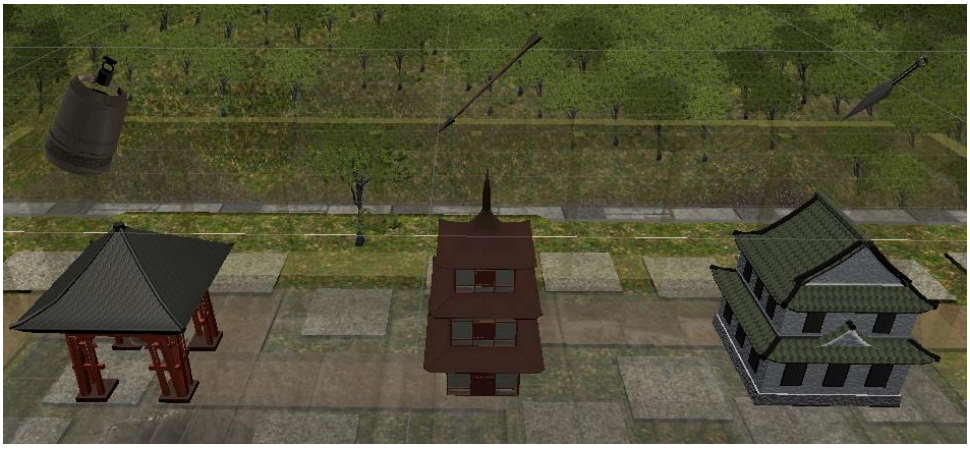


Figure 3.11. Bell towers, archer tower, and kunai tower respectively

To provide additional ways to become more powerful in the game, Skills are provided to the user from the beginning of the game to upgrade, with the game acknowledging this ability at the start of stage two, giving new players a way to ease into the mechanics and experienced players the option to prioritize skills from the beginning. These skills provide a variety of benefits and can be levelled up to improve the benefits. So far, the game has only one skill which gives the player extra money per enemy killed.

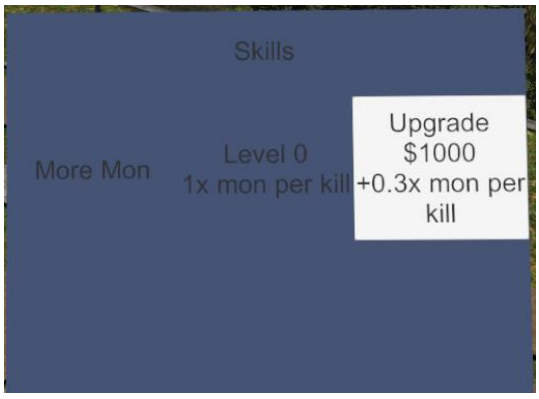


Figure 3.12. Skills menu UI

To help with the early portions of the game, the user interface gives hints about the behaviors of the different towers. Specifically, if the player hovers their pointer over a tower

spawner, it gives information about the initial stats of the tower, including the name, cost, power and range of the tower. When the player picks the tower up to place it somewhere, its range within the map is displayed as it is moved by the player to his or her desired position. This gives the player a clear representation of the tower's defensive screen as opposed to just an abstract number that only serves as a comparative scale between the tower information. This serves to help the player learn the mechanics of the game naturally, making it easier to pick up, which allows the game to progressively get more difficult as the waves of enemies continue their onslaught.

3.2.5 Tower Defense in Virtual Reality

During the planning phase of the game, it was difficult to find unique and novel mechanics to add to the game besides the dual world mechanic. At the end of a game pitch presentation, one of the peers at Takemura Labs suggested that the game should be implemented in virtual reality, followed by others supporting this suggestion. Moreover, the lab had access to virtual reality headsets with which to test our game, with many in Takemura Labs having experience in using VR. Ultimately, the suggestion ended up changing the design on the game by adding the inherent limited view from a first person perspective in the middle of the map as an intended goal of the design.

As a result of this shift in perspective, parts of the design had to be re-thought, primarily with player input. Most tower defense games, being a subset of real-time strategy games, rely on a computer mouse for input due to the precision and freedom of movement offered by it. Furthermore, some such devices have extra buttons that can be assigned functions or macros, giving RTS players more options that are a simple button press. Luckily using the VR controllers

as pointers offers a similar user experience as a computer mouse with only a minor loss in precision due to a lack of surface providing friction.

3.2.6 Sound

While the main element that was considered during the design process was gameplay and aesthetics, sound design is also important in creating context and reinforcing either the gameplay or the aesthetics. Unfortunately, since most of the time was focused on the gameplay, creating original sounds was only briefly considered. It was decided to use freesound.org to collect suitable sound effects for the game. These sound effects included enemy sound effects, tower sound effects, and notification sounds. These serve not only to reinforce the aesthetics of the game but to inform the player of events in the game, even if it's just audio confirmation.

3.3 User Stories

To better serve the experience goals, user stories were developed to guide the planned interactions between the player and the game. For the sake of covering every possible interaction, even the most mundane of interactions was accounted for. The events range from opening and closing the application, player-controlled events such as building towers and backend events hidden to the player such as the effects of the enemy wave number on how enemies spawn and behave. Below are two examples of player personas, and a description of how they might play the tower defense game. The first persona depicts a player who is completely new to tower defense games while the second persona is an experienced tower defense player.

3.3.1 Persona One

Isaac is accustomed to action games and is new to tower defense games but has an interest in VR; upon starting the game and pressing play on the main menu, he is transported to the top of a Japanese castle overlooking a path from a torii gate that splits into two and merges shortly before arriving at the castle's entrance. As he gets his bearings, he notices that the left controller has a text box floating in front of it. It explains the general controls, specifically activating the laser pointer by clicking the right touch pad and selecting the text box with the right trigger as it is being pointed to in order to advance the tutorial. In the next tutorial, the text explains that pointing at objects and selecting them with the trigger is how the majority of the game operates. Clearing this tutorial gives an explanation of the game itself, with a line explaining the premise of the game, what is needed to do to deal with the incoming enemies, and an explanation of where to find the defensive options, as well as what each one does. Feeling confident that he understands the game, he proceeds.

As Isaac clears this tutorial, the countdown starts, giving him the chance to point at the three tower models near him to get information on each via the UI that pops up when a tower model is pointed at that explains the tower's numerical stars, including power, range, and cost. Understanding what each one does, he begins setting up his defenses, prioritizing the archer towers upon seeing their range visualized as a circle on the map as he decides where to build his towers. After setting up the first tower, the game explains that he can upgrade or sell towers by selecting them after they have been placed. Once the countdown reaches zero, the first enemy, a single oni, spawns. The oni is defeated by Isaac's defenses, with the game congratulating him and explaining that every enemy defeated rewards the player with money to be used on further fortifications. The countdown to the next wave starts, giving Isaac fifteen in-game seconds to

strengthen his defenses. Once the next wave starts, three oni spawn; however, not all of them go down the path that Isaac has fortified. While the onis go through the range of one well-placed archer tower, the other towers are barely out of range. As a result, he needs to sell one or more of his towers to set up a tower on the other path. While he defeats two oni, one slipped through his defenses, prompting a message explaining that he has taken damage; if nineteen more enemies slip through his defenses, it is game over. Rethinking his strategy, Isaac begins setting up defenses upon the parts of the path that all the enemies go down: the initial enemy spawn point and the path right before the castle after the paths have merged.

After four waves, the game informs him of a new enemy type: the boss oni, which is slower but more durable than its smaller brethren. As the fifth wave starts, he gets a tutorial explaining that the map has expanded and that enemies will begin moving in from a second spawn point on the opposite side of the current spawn point. The game also explains the skill system; however, Isaac doesn't have the money to invest in skills and instead focuses on building more towers. Isaac has begun setting up defenses near the second spawn point. He is mostly successful in repelling the enemy. After the ninth wave, he gets a new tutorial about how enemies will begin traveling in a second map to get around his defenses, and that he can use the newly spawned gong to travel to the other map to set up defenses there. While marginally successful, he is overwhelmed by the increasing size and power of the enemy waves and having to juggle two separate maps. While he can pause the game to plan things out, he does not have the money to set up the towers that he wants or the in-game time to set them up before the enemies reach the castle. At wave twelve, Isaac loses all of his lives and gets a game over.

3.3.2 Persona Two

Morgan is a fan of tower defense games, but has only played them on IPhone; she wants to try out VR in a genre that she is already familiar with. After she takes some time in the main menu to get used to VR, she starts the game. She reads the tutorials about the game's controls and how to build towers. After reading the tutorials on each tower, she begins setting up her defenses close to the castle because she noticed that the path right before is a chokepoint as all enemies can only get to the castle itself from a single path. She sets up two short-range kunai towers with a bell tower between them to keep the enemies in their range. As the enemies need to make their way to her defenses, she presses the fast forward button to speed up game time to move on to the next waves. Instead of focusing on quantity of towers, she focuses on upgrading the ones that she has since they are on a chokepoint that has limited space for more towers. While she only has three towers set up, she continues upgrading them to make her choke point impenetrable.

While the map expands and a new enemy type is introduced, she isn't worried because they still have to funnel into her choke point; with money to spare, she invests in a skill that increases the amount of money that enemies drop on death. However, things get interesting when she is introduced to the dual-world mechanic, especially when it opens up a path past parts of her chokepoint. She uses the gong to see the second map and sets up a defensive screen at the ladder right before the part that links behind her previously built defenses, making use of the left trigger to hide the game objects in the castle to get a better view of the map and where she's placing her towers. As she stops each wave of enemies and realizes that each wave is more numerous than the last, she decides to hold off on upgrading her towers as they can handle the enemies just fine. She instead invests in her skills, letting her gain much more money on enemy death, which, in

turn, gives her more freedom to set up towers where she wants and upgrading the towers whenever she wants. While the map further expands with spawn points in the second map and a third enemy, the kamaitachi, is introduced, she has already set up powerful fortifications to stop any invaders from entering her castle. With the money to spare, Morgan sets up towers near all of the spawns to soften the enemies before they reach her main defenses. As the game continues past wave 50, she notices that the enemies have become substantially harder to kill despite investing in upgrades. As a result, she once again focuses on upgrading towers, as well as adding bell towers in strategic spots on the map to maximize how long the enemies stay within the range of the weaker towers along the branching paths. With most of the chokepoint towers significantly upgraded, she focuses more on quantity of towers to ensure that the enemies are damaged enough by the time that they reach the chokepoint. Despite her best efforts, enemies still find a way through her defenses. At wave 67, enough enemies made it through her defenses, and she gets a game over.

3.4 Paper Prototype

The paper prototype consisted of two maps drawn on two sheets of graph paper, with towers and enemies represented by square pieces with distinct markings on them. The player is allowed to place towers between enemy phases and the enemy waves are only active when they are specifically allowed to attack. The player is allowed to pause the game and place extra towers if need be, as well as switch to the other map and set up towers there. While the paper prototype served as a great starting point for the development of the full game, even the first digital build of the game varied drastically from the paper prototype. The paper prototype only had two paths that the enemies could travel, with specific points where they can arbitrarily switch paths; the

enemies spawn at one side of either path and their destination is on the other side and is accessible from either path.

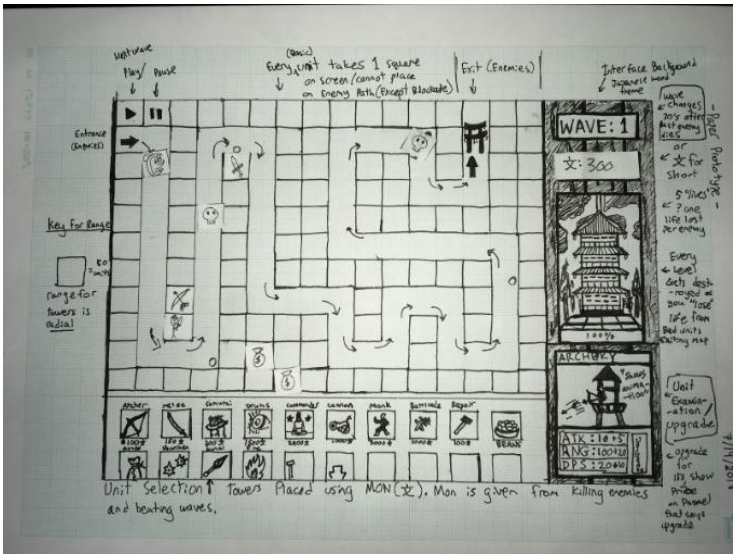


Figure 3.13. Paper prototype

While testing the paper prototype, many things were found that the team wished to change from the initial design. The most noteworthy change was the map. The prototype only had a single path on two maps, with spots on the maps where they can arbitrarily transfer over. While this system was simple, it was decidedly too simple; all that it would take to ensure that the enemy is damaged are two identical towers at the respective spots on each map. The world is, then, effectively, two dimensional, as, excluding radial tower ranges hitting two discrete parts of the path, the path is a straight line with a second line running parallel. Having more than one path on each map justified the two map system. In order to further increase the number of possible paths that enemies could take, the end point of the paths was placed in the center of the map, meaning that the choice of spawn point for the enemies could be at any point on the edge of the map; properly spread out, the spawn points allowed for more paths for the enemies to take. The paper prototype also only had two tower types to place and two enemy types to stop; it was determined that three of each would be the minimum for a complete game. Moreover, testing

enemy movement speed, enemy health, the paths enemies chose, and the damage dealt by each tower was inconsistent as it was all done on the spot. Actually testing in-engine provided greater consistency for more robust testing. However, since the primary point of the paper prototype was to test the underlying nature of the game, with the map being the only specific thing that could be tested, the paper prototype served its purpose successfully.

In the full game, the two paths are expanded into two sets of maze-like maps; each map has two enemy spawners, and the enemies can arbitrarily choose paths from forks in the road. There are still spots on each map that enemies can go to in order to switch over to the other map; however, these are one way, with distinct entrances and exits on each map. The enemies' destination has been moved to the center of the map and can only be entered from a single map, though there is a nearby spot that enemies in the other map can use to pop up incredibly close to their destination. Regardless of the paths that the enemies choose, they always head towards their end goal. The distinct phases were also removed, with a very quick timer separating each phase once the last enemy in a wave has been despawned either by its destruction or from it reaching its destination; towers are placed in real time, with time itself serving as a resource to be managed. The full game is also more controlled than the paper prototype, with the towers launching their projectiles at a fixed rate and the enemies moving at a constant speed and having a clear amount of health to be depleted. Furthermore, the first-person perspective could not be implemented in the paper prototype and can only be experimented with in a digital environment. The specifics of certain towers and enemies were also changed; tweaking of such values is easy to quickly test in a controlled way in a digital environment.

3.5 Map Design

In order to give the feeling of protecting a castle, the map need to be similar to the layout of a castle in Japan. To do this, the player is placed at the center of the map so that they have a 360 degree view of the surrounding area. Furthermore, the roads that lead to a castle entrance were designed to be placed all around the center castle to further add to the layout of the castle.

One of the unique aspects to our map design is a two level map. For this project, one level is considered to be the overworld which is where the castle is. The other level in this project is the underworld which is supposed to represent where the Yōkai come from. The underworld is conceptually placed directly below the overworld so that certain points on both maps lineup so enemies can travel back and forth between both levels. As such, enemies are able to advance towards the end destination through numerous paths due to this ‘layered’ level design. Enemies are able to access the underworld from the overworld by going into pits located at certain spots along the path. On the flip side, enemies can go from the underworld to the overworld through specified location on the map through the use of ladders. This provides the user with a strategic challenge when placing towers since they can place towers in either the overworld or the underworld. In Figure 3.2 below, the initial rough draft of what the map looks like is shown. The solid black lines and spawn boxes represent the overworld, the dotted lines and blue spawn boxes represent the underworld, and the circles on the map represent the pits and ladders where enemies can travel between the overworld and the underworld.

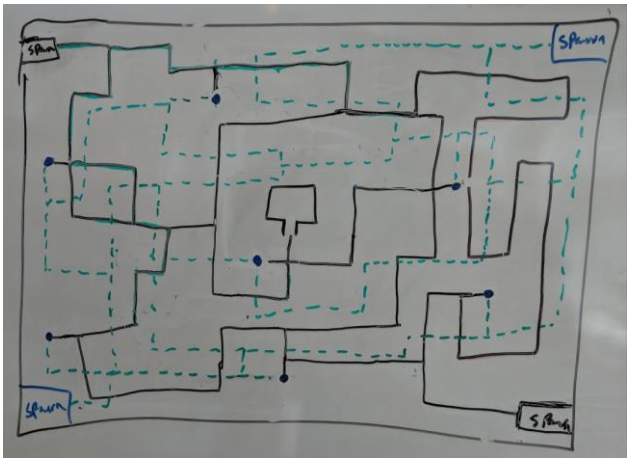


Figure 3.14. Sketch of the two level map (before staging was even considered)

The map is broken up into stages to ease the player into the game as mentioned prior. Once the second map is unlocked at stage three, the player is able to access both levels of the map through a gong located near the player. If the player points the laser pointer at the gong and clicks the trigger, they are teleported to the underworld area where they can place additional towers. This was done by changing the location of the steamVR entity to a specified empty object on the map. In the underworld, the player still had a 360 degree view of a map different from the overworld with the same layout of tower spawners. To get back to the overworld, the player need only click on the gong in the underworld.

Chapter 4: Implementation and Technology

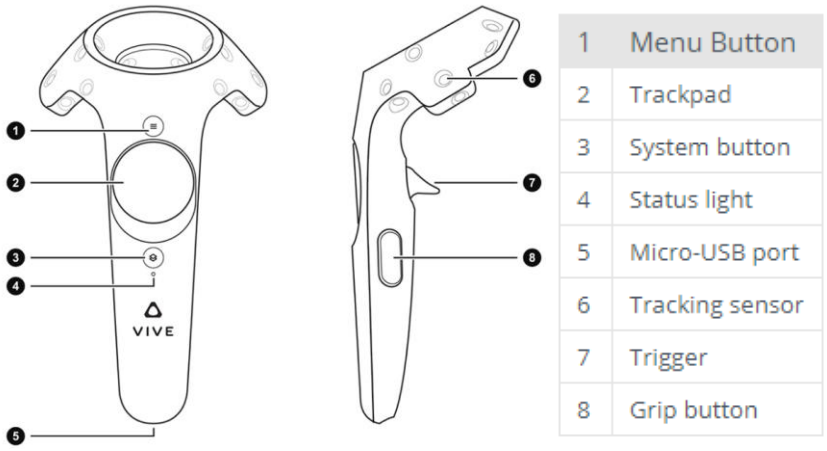
In Chapter 4, implementation of the game is described from a technical perspective. The chapter goes over the software and hardware used for the project, the code structure of the finished project and initial VR implementation. Additionally, the base model used for the initial design and changes made to this simplistic design to achieve the desired product are explained. Finally, the implementation of the website for the models made in the game is described.

4.1 Software and Hardware

There were many game engine software that could have been used to create a tower defense game. However, since this tower defense game was going to be in Virtual Reality (VR), either Unity or Unreal needed to be used. Unity is a content-creation engine for many platforms such as PC and smartphones that provides users with computer graphics tools and resources such as art and design tools, team collaborations, assets, and Virtual and Augmented Reality tools [12, 40]. Just like Unity, Unreal is a creation engine providing much of the same tools and resources that Unity does just provided differently to the user. This project used Unity because of it was better for beginners since there is a large community. It is user friendly when creating a 3D world and very intuitive creating objects and scripts needed to make the game work. Moreover, even though Unreal supports the same CAD software, they have limitations on importing and exporting the models that Unity doesn't have, making it slightly more artist friendly [40]. In terms of team experience, most team members had more experience working in Unity. Finally, Unity provides a much simpler collaboration tool to allow team members to work on the project together.

To allow for easy collaboration between team members, Unity Collaboration was used. Unity Collaboration is similar to Github in that team members can push and pull code. Unity Collaboration is part of the Unity UI as a function for collaborators to use. Since it uses a cloud platform to store the game for collaboration, it is simple to use which was convenient for the team [20]. Unity Collaboration was used over other collaboration services like Github because Unity projects can be difficult to work with when fixing scene conflicts.

The HTC Vive is a VR system allowing users to experience virtual worlds in the first person. The HTC Vive comes with a headset, two controllers, and two base stations. The headset contains 1080x1200 pixel screens for each eye, giving the user a 110 degree field of vision. Additionally, it contains multiple sensors like gyroscope and SteamVR tracking in order for the headset to accurately provided correct vision in the virtual world. The controllers contain six inputs including a trackpad, two grip buttons, a trigger button, a system button, and a menu button. Both the headset and the controllers interact with the base stations in order to give proper feedback to the system. The HTC Vive uses SteamVR in order to give players the ability to enter virtual worlds [22].



1	Menu Button
2	Trackpad
3	System button
4	Status light
5	Micro-USB port
6	Tracking sensor
7	Trigger
8	Grip button

Figure 4.1. HTC Vive Controller

This VR product was used for this project because all of the team members had no experience in VR prior to the project and the HTC Vive is considered to be one of the most user friendly in the VR department. Its community is also vast with a large amount of support when it comes to finding out how to do implement certain features which is helpful for beginners to learn the software quickly [37]. Additionally, the lab where this project was made had a lot of members who were very experience in the HTC Vive and SteamVR so if any problems arose, there were people who could provide advice and assistance.

4.2 Code Structure

4.2.1 General Relationships

Due to the time constraints set on the project, the code structure was not as organized as it could have been. As seen in Figure 4.2, there are many connections between classes that could have been minimized and refactored if there were more time on the project. As such, this section will go into a brief description of the major relations between classes.

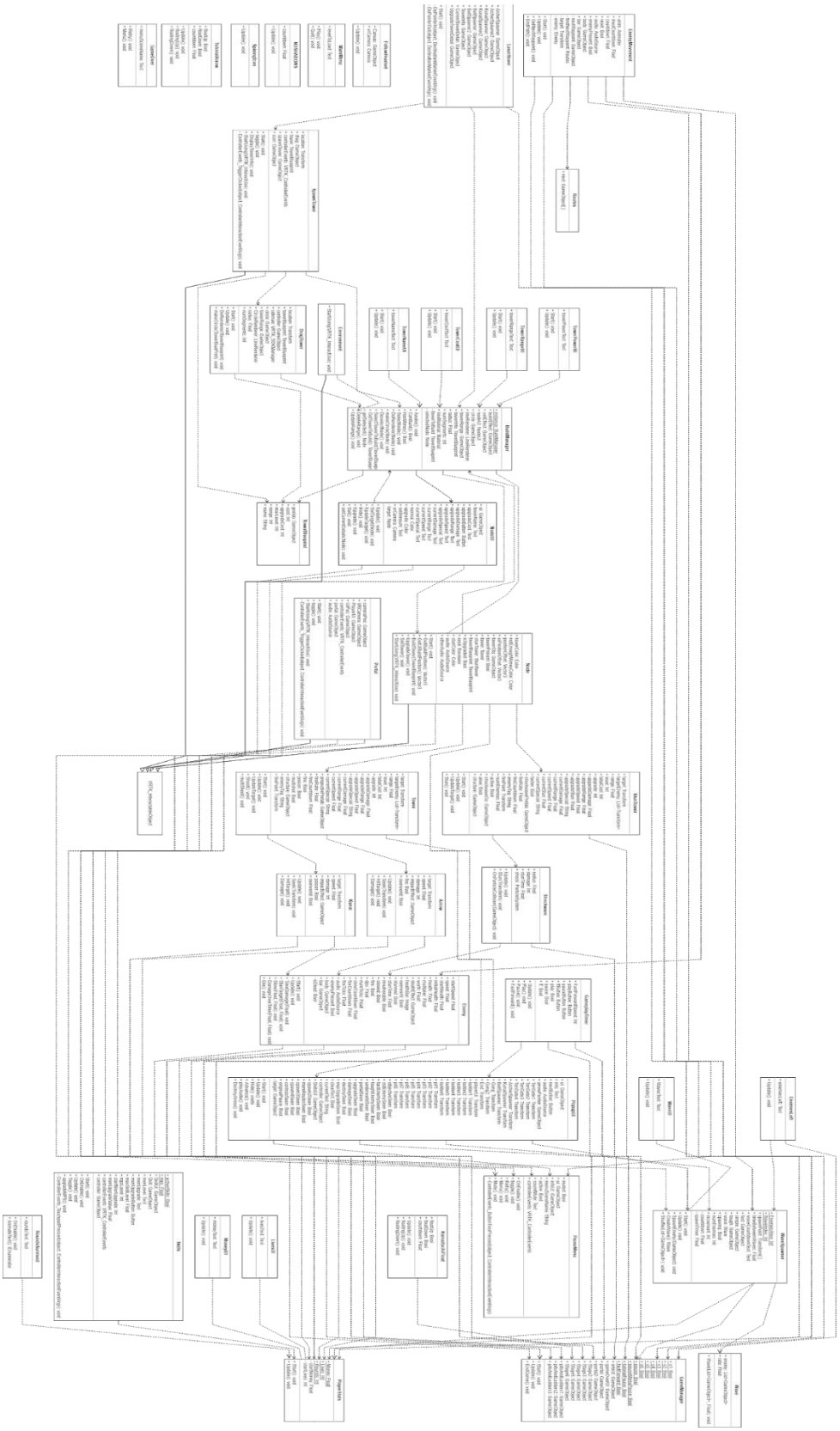


Figure 4.2 Overall relationships between classes

The relationships shown in Figure 4.3 contains the major game mechanics in the game. This ranges from placing towers on the map, towers shooting projectiles at enemies, and enemy movement. The `SpawnTower` script creates drag towers based on the tower blueprints and tells the `BuildManager` script what tower is being built. Then when the player selects a node, the `Node` script gets the information it needs from the `BuildManager` script. Meanwhile, the `NodeUI` script is gathering the information it needs from the `Node` script to display the tower information. Additionally, the `Node` script contains either a `Tower` or a `StunTower`. The `Tower` script creates either an arrow or kunai based on the tower. The `StunTower` script is different from the `Tower` script because it creates shockwaves which are particle systems instead of gameobjects. Finally, the arrow, kunai and shockwave all interact with the `Enemy` script to deal damage and special effects to the enemy. The `Enemy` script then tells the `EnemyMovement` script how to move on the map.

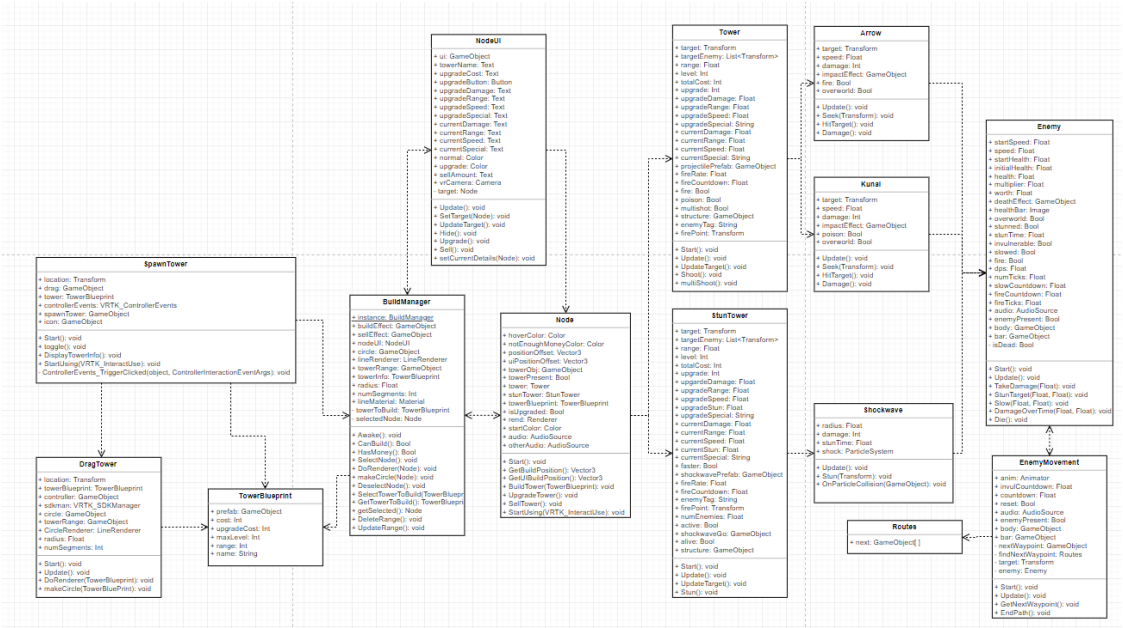


Figure 4.3 Game mechanics relationships

Figure 3.4. below shows the general relationship for spawning enemies. The `WaveSpawner` script generates a wave of enemies based on the current game state in `GameManager`. Those enemies then move throughout the map using the `EnemyMovement` script which uses `Routes` to find where to go to next.

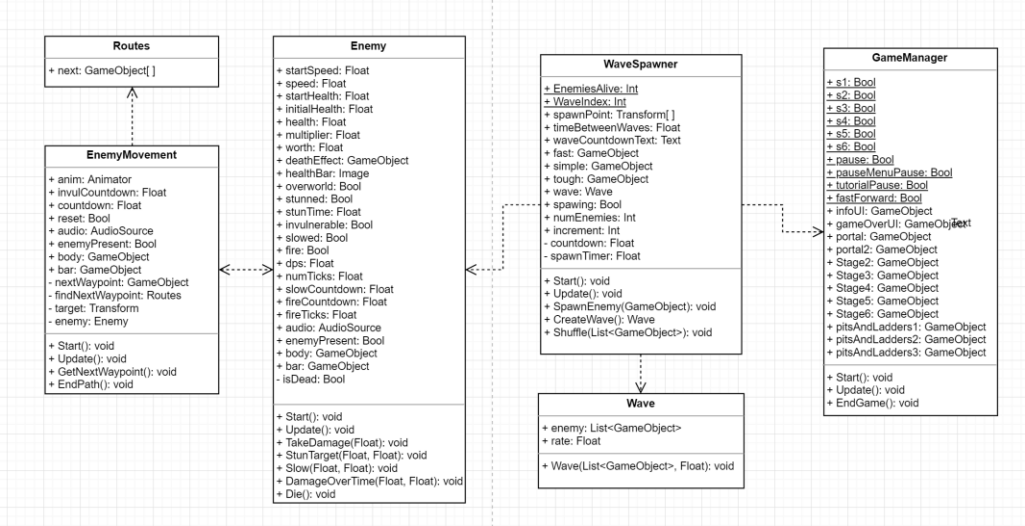


Figure 4.4 Wave spawning relationships

Figure 4.5 shows the general relationship for gameplay speed. Essentially, the `PauseMenu` and `GameplayTimer` scripts tell the `GameManager` script when the game is paused. Then all scripts that rely on movement or have some sort of countdown, base their timing speed off of the current game speed in `GameManager`.

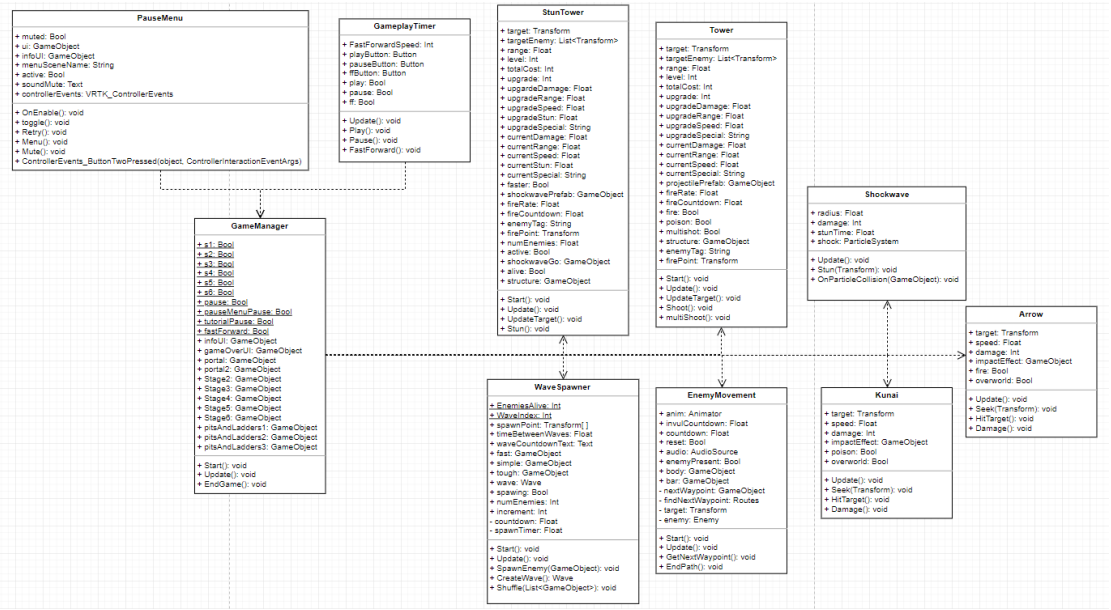


Figure 4.5 Gameplay speed relationships

Figure 4.6 shows all the scripts that inherit `VRTK_InteractableObject`. This `VRTK_InteractableObject` allows gameobjects to interact with the laser pointer. So, gameobjects like nodes that towers are placed on top of, the spawn towers, and the portal are all objects with scripts that needed to be `VRTK_InteractableObject`. Finally the Environment needed to also inherit `VRTK_InteractableObject` so that the player can cancel their tower placement.

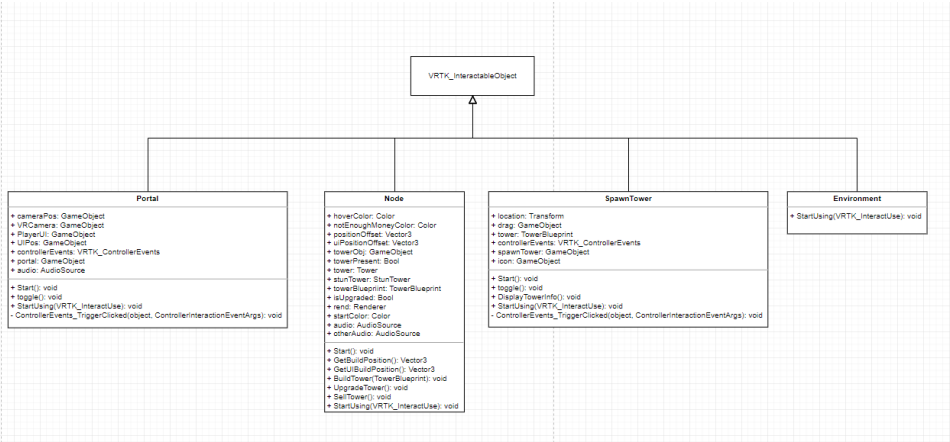


Figure 4.6 Gameobjects that inherit `VRTK_InteractableObject`

4.2.2 Class Descriptions

Below is the list of classes used in the project with brief descriptions about what each class does and what other classes it interacts with.

Arrow.cs - Responsible for the movement of the arrow and contains the stats of the arrow like damage, speed and fire. Moves the arrow towards the desired target as well as determines when to deal damage to the target and when to put an enemy on fire. Interacts with Enemy.cs to find out what world the enemy is in and to deal damage to the enemy and GameManager.cs to know what the gamespeed is.

BuildManager.cs - Singleton that manages selection of nodes for building, selling, and upgrading of towers. Used to determine if a player is building a tower, if the player has enough money for a tower, selecting nodes and deselecting nodes. Interacts with NodeUI.cs to know when to make the UI appear and when to make it disappear, Node.cs to know where the player is is either selecting a tower or placing a tower as well as to pass on information about the tower being built to the node, and TowerBlueprint.cs to know the details of the tower being built.

DragTower.cs - Creates a dummy tower of the tower being built as well as showing the radius of that tower. The tower and radius follows the end laser pointer so that no matter where the laser pointer is pointing on the map, the dummy tower and range are located at that point. This class gets all of its information like the tower to create and the range of the tower from SpawnTower.cs and TowerBlueprint.cs

EnemiesLeft.cs - Purely updates the player UI text field for showing the number of enemies left. Interacts with WaveSpawner.cs to get the number of enemies left on the map.

Enemy.cs - Contains all the stats of an enemy like health, speed, if it's stunned, immune, what world it is in, etc. This class keeps track of the damage the enemy takes, how long the enemy is

slowed for, how long the enemy is on fire for, when the enemy is dead, and when the enemy gets stunned. This class interacts with EnemyMovement.cs to set the speed of the enemy and when the enemy is stunned. It also communicates with GameManager.cs to help determine health and PlayerStats.cs to increase the amount of money when an enemy dies.

EnemyMovement.cs - Attached to enemies to make them progress through the map. Keeps track of when an enemy should be moving, stunned, transferred to the other map, how long an enemy is immune to stuns, and when an enemy has reached the end goal. This class interacts with Enemy.cs for stunning, invulnerability, and speed. Additionally, it interacts with Routes.cs to find out where the enemy must travel next and also GameManager.cs to find out what stage the player is on.

Environment.cs - Attached to all gameobjects and terrain that the player does not interact with. Acts as cancelling building a tower or getting rid of tower UI. Interacts with BuildManager.cs to deselect nodes and towers set the tower to build to null.

FollowHeadset.cs - Attached to gameobjects that should always be facing the player. Simply has the object always look at the main camera.

GameManager.cs - Singleton that keeps track of the staging progression throughout the game. Sets gameobjects to be active and inactive based on the stage the player is on. Interacts with WaveSpawner.cs to find out what wave the player is on to know when to go to the next stage. Additionally, it interacts with PlayerStates.cs to know when the player has lost the game.

GameOver.cs - Manages the interactions made on the gameover menu screen like retry and quit.

GameplayTimer.cs - Keeps track of the game speed, specifically when the game speed is paused, fast forwarded, or played at the normal speed. Interacts with GameManager.cs to set the

game speed variables and WaveSpawner.cs to know when to stop interactions with the player UI pause button.

Kunai.cs - Responsible for the movement of the kunai and contains the stats of the kunai like damage, speed and poison. Moves the kunai towards the desired target as well as determines when to deal damage to the target and when to poison the target. Interacts with Enemy.cs to find out what world the enemy is in and to deal damage to the enemy and GameManager.cs to know what the gamespeed is.

LaserHover.cs - Keeps track of when the laser pointer hovers over an object. The script determines when the laser hovers over a node, the upgrade button on the nodeUI, or a spawner and acts accordingly.

LivesUI.cs - Updates the text for the number of lives the player has left on the player UI. Interacts with PlayerStats.cs to get the number of lives left.

MainMenu.cs - Manages what happens in the main menu scene like when to play the game and when to quit the game.

MoneyUI.cs - Updates the text for the amount of money the player has left on the player UI. Interacts with PlayerStats.cs to get the amount of money left.

Node.cs - Keeps track of the tower currently at that node as well as all functionalities of the tower like building, selling, upgrading, selecting. Specifically for upgrading, it sets the tower's upgrade stats. Interacts with BuildManager.cs to know what tower to build and what tower to select, PlayerStats.cs for managing the money gained and lost, Tower.cs and StunTower.cs to apply upgrades, and TowerBlueprint.cs to get the initial stats of the tower.

NodeUI.cs - Manages the tower UI that displays the tower's stats and upgrade/sell buttons. Keeps track of when to sell a tower and when to upgrade a tower as well as updating the tower

stats when the tower is upgraded. Interacts with Node.cs to sell and upgrade towers and to get tower details.

NOSHADOW.cs - Gets rid of the controller shadows when the scene starts up

PauseMenu.cs - Controls interactions on the pause menu UI. Manages if the game is paused, if the player would like to continue, quit or retry, and sound muting. Interacts with the GameManager for pausing.

PlayerStats.cs - Keeps track of all the player stats like money, lives, and the number of rounds the player has survived.

PopupUI.cs - Attached to the tutorials that pop up on the player's left controller. Contains multiple conditions for when to open a tutorial as well a process for proceeding through the rest of a tutorial. Interacts with GameManager.cs to find out what stage the player is on, PlayerStats.cs to find out how many lives the player has, and WaveSpawner.cs for the wave number the player is currently on.

Portal.cs - Attached to gong objects that transport the player between the Overworld map and Underworld map. Interacts with GameManager.cs as it only shows up once stage 2 is cleared and must be visible and togglable in all subsequent stages.

RoundSurvived.cs - Attached to the Game Over screen to show the number of waves that the player has survived upon game over. Interacts with PlayerStats.cs to get access to the number it displays.

Routes.cs - Creates a list of all possible next waypoints that an enemy can move to and sets the distance between each waypoint and the end point.

Shockwave.cs - Attached to the particle system generated by the bell towers. Interacts with enemies that the particles collide with to stun them for a set time. Interacts with

GameManager.cs to check if the game is paused to stop the particles from continuing on their path.

Skills.cs - Attached to the Skills UI to control the skills' properties, including level, cost, and function. Interacts with PlayerStats.cs for purchase.

SpawnTower.cs - Attached to the tower spawners to control the player's ability to drag a drag tower to nodes on the map. Interacts with BuildManager.cs to deselect nodes, delete ranges, select tower to build, and display tower info. Interacts with GameManager.cs to prevent interaction while the game is paused.

SpinningIcon.cs - Attached to the icons above the tower spawners to have them rotate aesthetically.

StunTower.cs - Attached to the Bell Towers to control their behavior like creating shockwaves and tracking enemies. Interacts with GameManager.cs to change time scale when fast forwarding or paused. Generates shockwaves using Shockwave.cs.

Tower.cs - Attached to the Archer and Kunai Towers to control their behavior like creating projectiles and tracking enemies. Interacts with GameManager.cs to change time scale when fast forwarding or paused. Generates either arrows or kunai using Arrow.cs or Kunai.cs.

TowerBlueprint.cs - Attached to tower spawners to hold their cost, upgrade cost, max level, range, and name. Can use cost to return the sell amount.

TowerCostUI.cs - Attached to the tower UI to display the cost of the given tower. Interacts with BuildManager.cs to get the cost from the tower info.

TowerNameUI.cs - Attached to the tower UI to display the name of the given tower. Interacts with BuildManager.cs to get the name from the tower info.

TowerPowerUI.cs - Attached to the tower UI to display the power of the given tower. Interacts with BuildManager.cs to get the name from the tower info and display the appropriate number.

TowerRangeUI.cs - Attached to the tower UI to display the range of the given tower. Interacts with BuildManager.cs to get the range from the tower info.

TutorialArrow.cs - Attached to the tutorial arrows generated by PopupUI.cs. Makes the tutorial arrow move up and down aesthetically.

Wave.cs - Controls the enemies to be spawned and the rate at which they are spawned. Generated by WaveSpawner.cs after the countdown timer after the last enemy of the previous wave has been defeated.

WaveSpawner.cs - Generates a wave of enemies based on an internal index. Interacts with GameManager.cs to get the game state and what stage is currently active for the sake of spawn points and enemy variety and PlayerStats.cs to increment how many waves they have survived.

WaveUI.cs - Attached to the Player UI to display what wave the player is currently on. Interacts with WaveSpawner.cs to get the wave index.

4.3 Base Model

Since some of the team members were not that experienced in Unity, the first few steps in the project was to learn the game engine as well as finding out how to implement the tower defense game. After thorough research on how to make tower defense games, the team found a commendable tutorial[1] on how to implement a 3D tower defense game in Unity, produced by Brackeys. It was decided to use this tutorial as the base model for the game and change it to fit the new tower defense game.

[1] Website link for tutorial on how to make a 3D tower defense game made by Brackeys: https://www.youtube.com/watch?v=beuoNuK2tbk&list=PLPV2KyIb3jR4u5jX8za5iU1cqnQPmbzG0

The base model included many of the base functionalities that a tower defense game needs. It included a small one-path map where enemies spawn at one point and travel to the end destination by following waypoints placed at intersection points. Each enemy had a health bar to indicate the health of the enemy. Additionally, the base game allowed the user to place three types of towers at nodes on the map: a Standard Turret that shoots bullets dealing single target damage, a Missile Turret that shoots missiles dealing area damage, and a Laser Turret dealing damage over time and slowing the enemies. These placed towers were capable of tracking enemies in their given attack range and would shoot a projectile at the enemy dealing damage until either the enemy died or the enemy was out of range. Each tower was also capable of being upgraded once, increasing damage and range as well as being sold for a portion of their buy cost. Finally a game system was implemented to keep track of basic background information like player lives, money, etc..

Even though this tutorial had a fairly complete version of a simple tower defense game, it was still missing many aspects that were needed for a more developed version. The turrets provided in the base model did not suit the Japanese aesthetic of the project, and the fact that each turret had only a single upgrade and had their properties hardcoded into them only served to restrict strategic (from the user side) and design (from the developer side) options. Some of the biggest missing features in this base model that is need for our game is Virtual Reality, a more complex map and a bunch of game preference changes.

From the code structure mentioned earlier, only some of the scripts were used from the base model without changing while others were changed drastically. For instance, PlayerStats.cs, Wave.cs, Round.cs, MainMenu.cs, and GameOver.cs were either hardly changed or not changed at all. On the other hand, Tower.cs, PauseMenu.cs, NodeUI.cs, Node.cs, Arrow.cs, Kunai.cs,

GameManager.cs, EnemyMovement.cs, Enemy.cs and BuildManager.cs are all scripts that were dramatically changed to fit what was needed for the project. StunTower.cs is a variation on Tower.cs. Arrow.cs and Kunai.cs are variations of Bullet.cs which was removed from the project. All other scripts not mentioned were all original scripts created for other elements of the game.

4.4 Virtual Reality Technical Implementation

Before the implementation of Virtual Reality, the game, like many other tower defense games, had a static bird's eye perspective. This provided the player a full view of the entire map and limited ability to change the default perspective. Furthermore, the player was only able to interact with the game through mouse input. Since the game was going to be VR, these functionalities that the base model had were only good for initial testing of some aspects of the game. Once VR was implemented, these functionalities were replaced with VR-friendly equivalents, including a dynamic first-person perspective and VR controller support.

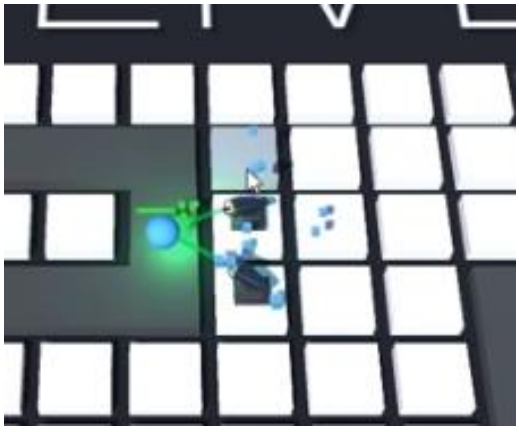
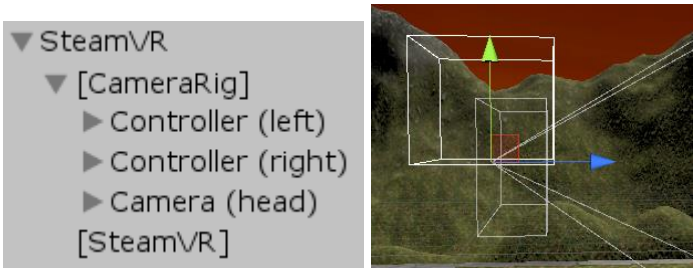


Figure 4.7. Example placing turrets with a mouse

In order to convert the main camera from a 2D screen view to a VR camera view, the SteamVR gameobject needs to be added. This gameobject includes the new main camera as well

as the left and right controllers. When this is put in the Unity scene, a transparent box appears representing the actual real life space that the player is allowed to move in. Since this represents the real world space, this also gives the creator the ability to scale the box to give different perspectives of the game. For instance, if the entire tower defense map was placed inside the box, the map would look tiny. If the box was only one square on the map, the map would look huge to the player.



Figures 4.8 (L) and 4.9 (R). SteamVR gameobject hierarchy and worldspace gameobjects respectively

4.4.1 Virtual Reality Toolkit

Virtual Reality Toolkit (VRTK) is a downloadable asset package that contains a variety of scripts that help with building a VR game. VRTK contains solutions to common VR problems such as interaction with gameobjects, interactions with User Interface (UI) elements, and Software Development Kit (SDK) managers [23]. Additionally, it happens to be one of the three official SDKs for the HTC Vive [2]. One of the useful aspects to VRTK is the VRTK_SDK manager which detects which VR device you are using and appropriately uses the correct gameobjects to allow the game to run. If there are no VR devices connected, then VRTK uses a simulator which allows you to test the game on the computer through the keyboard and mouse.

Figure 4.10. VRTK SDK Manager

Another unique feature of VRTK is the controller functionalities. VRTK allows for easy set up of controller button events through the `VRKT_ControllerEvents`. Though this script, you can set which button on the HTC Vive controller triggers which event right inside the inspector. Additionally, VRTK has a laser pointer script that allows user to interact with the scene using a laser pointer that comes out of the controller. In the inspector, you can control what button on the controller activates the laser pointer, what button triggers events based on the gameobject the laser is pointing at and various other functionalities.

The laser pointer interacts with gameobjects by creating `VRKT_InteractableObject` instead of a `MonoBehaviour`. A `MonoBehaviour` is a type of class specifically used for Unity that allows a gameobject to have specific in game functionalities like having a gameobject constantly be updated and executing an action when the object is initialized. When making the interactable object, many options appear in the inspector to customize how the object can interact with the laser pointer. It allows you to customize grab options, touch options, and use options. In order to make the object interactable with the laser pointer, the *"Is Usable"* checkbox must be clicked. Additionally, the *"Pointer Activates Use Action"* checkbox needs to be clicked to allow the object to know that it is being interacted with and can run a method called `StartUsing()`. Finally, the Laser pointer is able to interact with UI elements by using `VRKT_UIPointer` and `VRTK_UICanvas`. By adding the

`VRTK_UIPointer` to the controllers, it allows the user to decide which buttons on the controller interact with the UI elements. The `VRTK_UICanvas` is applied to any canvas that should interact with the laser pointer. This creates box colliders for buttons so that the laser pointer can interact with them.

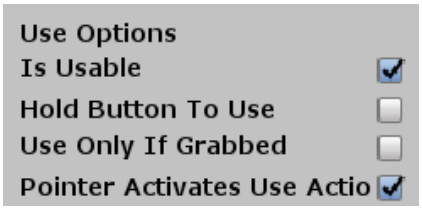


Figure 4.11. VRTK interactable object settings for interaction

The reason why the laser pointer is necessary is because the player is fairly stationary. They do not have to move any more than a few feet and since the map is large relative to the space the user is able to move in, the user needs some way to interact with game elements that are far away. Moreover, since the player is up in a castle looking down at the map, they are not able to walk right up to a spot to place a tower. By using a laser pointer, players can point at the spot instead, giving them an easier time controlling where game elements go and being sure which game elements they are interacting with.

4.5 Tower Defense Design Changes

This section describes the major changes made to the functionality and design of the base model. This mainly includes changes made to scripts and effects as well as additional scripts added to the game to get the intended functionality. Included are topics about enemies, towers, staging, tutorials, and much more.

4.5.1 Enemies

Originally, enemies spawn at one point on the map and follow one path to reach the end goal. Since our map design called for multiple spawn points, multiple paths, and two levels, many changes needed to be made to the enemy scripts.

To solve the first problem, spawning enemies at multiple spawn points, a random number is generated between zero and three. This number indicates which spawn point to instantiate the enemy at. Once that number is decided, it spawns the enemy at the specified transform from the `spawnPoint` list. Variety in enemies also needed to be introduced to the game and was done by having each wave randomly select the order and type of enemies that spawn. As such, every five waves, a new enemy is added to the assortment of enemies that can be spawned. From there, the script randomly chooses an enemy from the variety of possible enemies and add it to the list of enemies for that wave.

```
Creating Waves
This pseudocode  aims to explain how enemy waves are created.
It decides which enemies to create depending on the stage of the game
and randomizes the order they appear.

Create a new list for the order of the enemies

Initialize how many eneimes there will be for the current waves

For the numbers of enemies this waves
    If the game stage is 1, 2, or 3
        Add an Oni to the list of enemies
    Else
        Add a Kamaitachi to the list of enemies

If the current wave index is a multiple of 5
    Then add a Nue (boss) to the list for each multiple of 5
    (ie wave 5 creates 1 Nue, wave 10 creates 2 and so on)

Randomize the order the enemies will spawn

Create a new wave object with the list of enemies and the rate they will spawn

Return the wave
```

Code Listing 4.1. Pseudocode for creating a wave

Due to spawning the enemies in different locations, it was not easy to set the initial node that the enemy has to travel to. To fix this problem, the enemy movement script finds the closest spawn waypoint and sets it as the next waypoint. From there, enemies now need to determine which path to travel in order to reach the end goal. Each waypoint contains a list of waypoints

that the enemy can travel to next. Using that list, enemies randomly choose the path to travel down. When an enemy wishes to travel to the other map, it can take pits or ladders. Once the enemy reaches that pit or ladder, the enemy is moved immediately to the position of the corresponding pit/ladder in the other level.

```
Enemy Movement
This pseudocode aims to explain how enemies move on the map.
This is done in the update function of the enemy movement script


If the game is paused
    Then stop all animations and return immediately

If the enemy got stunned
    Stun the enemy
    Return immediately

If the next waypoint to go to is a pit
    Set the position of the enemy to the next waypoint
    Enemy is now in the underworld
Else if the next waypoint is a ladder
    Set the position of the enemy to the next waypoint
    Enemy is now in the overworld
Else
    Get the direction of the next waypoint
    Translate the enemy towards the next waypoint

If the distance to the next waypoint is small
    Then get the next waypoint

Make the enemy's healthbar look at the player
```

Code Listing 4.2. Pseudocode for moving the enemy from waypoint to waypoint

Furthermore, when the enemies decide which waypoint they are going to next, the enemy rotates its body so that it is facing the next waypoint. This was done using the `LookAt()` function. Additionally, each enemy's health bar is programmed to always face the player using the same function.

One of the game mechanics in the tower defense game is stunning enemies. As such, additional code was needed to make sure that the enemies do not move when they are stunned for the allotted amount of time. Thus, a countdown time was used to make the enemy do nothing while the countdown was running. The enemy movement code would gather the stun information from the enemy and set the countdown timer accordingly. Once the countdown reaches zero, the

enemy is made immune to stuns for half a second by setting a boolean variable to true, making it so that the enemy can't be stunned while that variable is true. Once the countdown for being immune reaches zero, the enemy resumes its original movement.

```
Stunning an Enemy
This pseudocode aims to explain how to stun enemmies
This code is executed in the update function of the enemy
movement script after the pause check but before checking
the waypoints

//Pause check

If the enemy is immune to stun
    If the countdown has reached 0
        Set enemy immune to stun variable to false
        Return immediately

    Decrement the countdown by the game time that has passed since the last update

Else if the enemy got stunned
    Then check if the enemy is already stunned or not
        If it is not stunned, set the countdown to the duration of the stun

    If the countdown has reached 0
        Set the stun duration to 0
        Set the enemy to not being stunned
        Set enemy immune to stun variable to true
        Set immune countdown to 0.5 seconds
        Return immediately

    Decrement the countdown by the game time that has passed since the last update

    Return immediately

//Waypoint check
//Enemy movement
```

Code Listing 4.3. Pseudocode for stunning enemies

The last enemy component that needed to be changed was spawning mechanics with pausing. Due to the way pausing was implemented into the game (explained later in section 4.5.8), a spawning timer was needed to ensure that enemies do not spawn on the map while the game is paused. At first, when the game was paused, all the enemies would keep spawning but would not move. This caused all the enemies to be in the exact same position while walking through the map, giving the player false information about how many enemies there are. To fix this, a countdown timer was implemented to make sure that enemies aren't spawned when the game is paused.

```
Starting a Wave
This pseudocode aims to explain how wave spawning worked
This is put in the update function of the wave spawning script

If the game is pause
    Return immediately

If the game is over
    Return immediately

If the number of enemies alive is greater than 0 and spawning has finished
    Return immediately

If the wave index is 50
    Stop spawning forever

If countdown for spawning the next wave is 0
    If spawning has not started yet this wave
        Increment the wave number

        Create a new wave with enemies to spawning
        Set the number of enemies alive to the number of enemies in the wave
        Set the number of enemies left to spawn to the number of enemies in the wave

        Set the spawn timer to 0 so spawning happens right away
        Set the number of enemies spawned so far to 0
        Set spawning to true

    If the spawning countdown is 0
        Spawn an enemy from the wave list
        Increment the number of enemies spawned so the next enemy can be spawned
        Reset the spawning countdown

    If the number of enemies spawned equals the number of enemies this wave
        Set spawning to false
        Set the countdown to the next wave
        Return

    Decrement the spawn countdown by the game time that has passed since the last update

Decrement the wave countdown by the game time that has passed since the last update

Display the countdown time on the UI to show the player how much time is left
before the next wave
```

Code Listing 4.4. Pseudocode for spawning enemies on the map

4.5.2 Towers

The Archer Tower and Kunai Tower were not that hard to implement because it followed the design of the Standard Turret of the base model. The tower looks for enemies within its range and once it determines the closest enemy, it creates a projectile. Once the projectile is instantiated, the projectile moves towards the target until it is within a certain range. At that point, the projectile is destroyed and deals the appropriate amount of damage the the enemy.

```
Tower Targeting
This pseudocode aims to explain how towers target enemies

If tower doesn't have multishot upgrade yet
    Create a list of enemies filled with Gameobjects with the "Enemy" tag

    Set shortest distance to Infinity to start
    Set nearest enemy to null to start

    For each enemy in the list of enemimes with the "Enemy" tag
        Get distance from the tower to the enemy
        If distance from the tower to the enemy is less than the shortest distance
            Set shortest distance to the new distance
            Set nearest enemy to the new enemy

    If the nearest enemy isn't null and the shortest distance is within the range
    of the tower
        Set the target of the tower to the nearest enemy's transform
    Else
        Set the target to null

Else
    The tower follows multishot targeting
```

Code Listing 4.5. Pseudocode for finding enemies to target

The Bell Tower was fundamentally different from all the other towers. The Bell Tower needed to send a shockwave from the bell to the enemies. Once the shockwave hits the enemy, it stuns the enemy for a brief amount of time. The only similarity with this tower is that it waits for an enemy to be in range before it sends a shockwave. Instead of a projectile prefab, the Bell Tower creates a particle system. Almost all of the particle system settings are predefined except for how long the particle system is active. In order to get the particle system to act like a shockwave, there were two key components to the particle system that were used. The first changing the shape to a circle and the radius of the shape to 1 in the shape settings. This makes it so that the particles expand outwards from the center, making a circle. The second important component is the collisions settings. The collisions settings must be activated with the type set to world. This is the key component that allows the particle system to interact with enemies. Since the particles now have colliders attached to them, the function `OnParticleCollision` activates whenever the particle collider hits another collider. So whenever the particles hit an enemy collider, the enemy gets stunned and takes a small amount of damage.

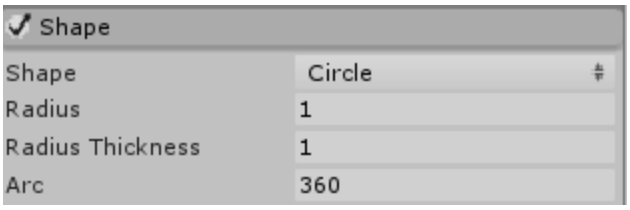


Figure 4.12. Particle system Shape setting

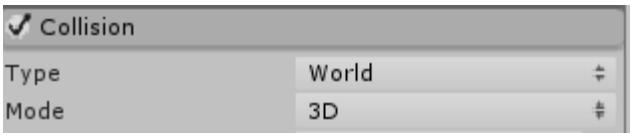


Figure 4.13. Particle system Collision settings

4.5.3 Towers Properties

To make tower upgrading better, more scalable upgrading and displaying the range of the tower were implemented. The idea behind more scalable upgrading is to be able to upgrade the towers higher than just two levels and at certain levels, the tower gains "special abilities" without having to hard code elements and properties in Unity. For example, the max level of all towers is level 10 and at level 5, archer towers get fire arrows dealing damage over time to the enemy. Additionally, the kunai tower gains poison kunai at level 5, which slows enemy movement for a limited time. At level 10, both of these towers start shooting multiple projectiles. In order to accomplish this, each `TowerBlueprint` was given a `maxLevel` variable. Additionally, the upgrade function now accesses the tower information and changes the values based on the tower. Once this code was in place, upgrading towers became more dynamic and can be changed very easily.

```
Upgrading Towers
This pseudocode aims to explain how upgrading towers works
This was put in the node script


If the tower is an Archer tower

    If the player's money is less than the upgrade
        Return and do nothing

    Subtract the upgrade cost from the player's money

    Increment the total cost of the tower by the upgraded amount

    Update the tower's current stats
    Update the tower's upgrade stats

    Check what the tower's level is
        If the tower's level is 4
            Set the tower's upgrade special ability (Fire arrows)
        If the tower's level is 5
            Give the tower the new special ability
        If the tower's level is 9
            Set the tower's upgrade special ability (Multishot)
        If the tower's level is 10
            Give the tower the new special ability

Else if the tower is a kunai tower
    //Same as archer tower except for different special upgrades and scaling

Else if the tower is a bell tower
    //Same as archer tower except for different special upgrades and scaling
```

Code Listing 4.6. Pseudocode for upgrading a tower and altering its stats

To give the player an idea of what each upgrade to the tower does, upgrade details are given in the tower UI. The tower UI provides the amount of damage that is added to the tower, the increase in range, the increase in the speed of projectiles, and any other special effects that may be added to the tower. In the game, there are four special upgrades. Fire arrows deal damage over time by taking a percentage of the damage the archer tower normally deals and dealing that amount of damage to the enemy every second a certain amount of times. Poison kunai slow the enemies by setting their speed to a percentage of their original speed for a certain duration of time. Once the duration is over, the speed is set back to normal. Multishot allows towers to target multiple enemies at once. Instead of getting one target, the tower gets all targets within range and create a projectile for each enemy. Finally, the faster wave upgrade for the bell tower increase the speed at which the shockwave reaches the enemies.

```
Fire and Poison
This pseudocode aims to explain how enemies take damage over time and are slowed
due to fire and poison.
This is put in the update function of the enemy script


If the enemy is poisoned
    If the poison countdown is less than 0
        Set the speed of the enemy to the original speed
        Set slowed to false

    Decrement the poison countdown by the game time that has passed since the last update

If the enemy is "on fire"
    If the fire countdown is less than 0
        Deal damage to the enemy
        Set the fire countdown to 1
        Increment the number of fire ticks

    If the number of fire ticks equals the number of ticks to be dealt
        Set fire to false

    Decrement the fire countdown by the game time that has passed since the last update
```

Code Listing 4.7. Pseudocode for Fire and Poison

```
    Multishot
    This pseudocode aims to explain how towers attack multiple targets
    at the same time. This is put in the same function as the targeting
    enemies function.


    If tower doesn't have multishot upgrade yet
        Target one enemy

    Else
        Create a new list of target enemies

        Create a list of all gameobjects with the "Enemy" Tag

        For each enemy in the list of enemimes with the "Enemy" tag
            Get the distance from the tower to the enemy

            If the distance from the tower to the enemy is inside the
            range of the tower
                Add the enemy transform to the list of target enemies
```

Code Listing 4.8. Pseudocode for Multishot

To allow the user to see the range of the tower, tower range consisted of a line renderer and a translucent circle. The line renderer represented the outer radius of the tower's range while the translucent circle was used to fill in the circle to make it more aesthetically pleasing. When the player clicks on a tower, the tower script creates a circle that is the size of the range of the tower. The translucent circle is also instantiated and then scaled to fit the range of the tower. When the user clicks off the tower or clicks on the tower again, the line renderer disappears and the translucent circle is deleted.

```
Line Renderer
This pseudocode aims to explain how the line renderer is used
to show the range of a tower

Get the Line Renderer component from the gameobject
Create the colors for the line renderer
Set the material of the line renderer
Set the start and end colors of the line renderer
Set the width of the line renderer
Set the number of segments the line renderer will have
Set the line renderer to world space

Get the angle change per line segment using the number of segments
Set theta to 0

For the number of line segments
    Set the x value to the tower's range * Cos(theta)
    Set the z value to the tower's range * Sin(theta)
    Create a new position where x is the tower's x position + x value,
    y is 1, and z is the tower's z position + z value

    Set the position of the line segment to the new position
    Increament theta by the angle change
```

Code Listing 4.9. Pseudocode for creating a line renderer representing the tower range

4.5.4 Placing Towers

Placing towers in the base model was nowhere near what was needed for the VR tower defense game. The base model had the player select the tower to build though an overlay canvas in screen space. Instead of this, towers are placed in world space. There are spawners located near the player that the laser pointer can interact with. When the player clicks on the spawner, a duplicated drag tower appears and notifies the build manager which tower is being built. Additionally, the new tower displays the range of the tower though the same line renderer and translucent circle approach mentioned above.

Next, the newly instantiated tower follows the laser pointer. Wherever the end of the laser pointer is pointing, the tower and its range follows, allowing the user to visually see where the tower are placed as well as a general sense of how much area that tower can cover. This was done by creating a plane normal to the laser pointer and using a ray to return distance along the ray for where to reposition the tower.

```
Drag Tower Laser Pointer Following
This pseudocode aims to explain how the drag tower
follow the laser pointer. This is done in the update function
of the drag tower script


Create a new plane at a certain y value

Create a new ray with the origin set to the controller's position
the direction set to the controller's direction

If the ray intersects the plane
    Set the drag tower's position to the end of the ray
    Set the tower's circle range position to the end of the ray
    Update the line renderer
```

Code Listing 4.10. Pseudocode for making a tower follow the laser pointer

Finally, when the player has decided on a spot to put their tower, the tower that was being dragged around and its range is destroyed. At the same time, the actual tower is spawned at the point the user wanted and immediately starts attacking enemies once they are in range.

4.5.5 UI Interactions

Many of the UI interactions in the base model of the game were done though screen space. Information about the player's lives, money, and wave information was displayed only to the screen and could not be seen in the world. To start, the screen overlay containing the player information was moved to world space so that the player can see the information in VR. Initially, the canvas containing this information followed the player's camera so that it was always at the same spot relative to where the player was looking. To accomplish this, VRTK has a script called `VRTK_ObjectAlias` which makes the canvas a child of the VR camera, thus following the camera wherever it is looking.

After testing, it was determined that showing the UI information like this was not good because it could not always be seen and would go through gameobjects. Instead of having the canvas follow the camera, the canvas was put at a fixed location near the player. The UI canvas would now always face the VR camera instead of follow the camera to provide optimal angles of viewing.

From there, players also needed to be able to interact with the UI elements. Since VR doesn't work the same as a mouse and keyboard, changes needed to be made to the UI canvases. For each UI button that needed to be interacted with, a canvas was needed with `VRTK_UICanvas` attached as a component. This allowed the laser pointer to be able to "press" UI buttons that are a child of the canvas. As a result, UI elements like upgrading and selling towers needed to be altered in order for the player to be able to interact with it thought the laser pointer in VR.

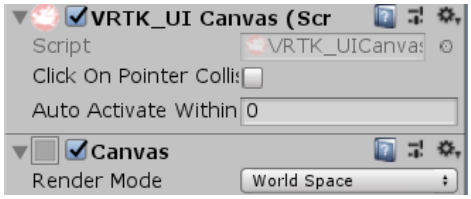


Figure 4.14. Converting the Tower UI to be intractable with the laser pointer

Furthermore, the Pause menu was also altered to allow for laser pointer interaction but it also only appears when the menu button on the controller is pressed. This was done by creating an event that listens for the `buttonTwoPress` on the controller. Once this event is triggered, the Pause menu appears and the game stops.

Additional information was also added to the player information. Originally, there was no information about towers that allowed players to see how much damage a tower does or how much it costs. To do this, whenever the laser pointer hovers over a spawner, the player information UI displays the tower's name, damage, range, and cost. In order to implement this with a laser pointer, the VR controllers needed to be given specific events to listen for. In this case, an event was added to the controllers for whenever the destination marker of the laser pointer either enters or exits a collider. When this happens, the event triggers a function that checks the tag of the object. If the object matches a spawner, it updates the player UI with the correct tower information and position the UI so that it is right above the spawner.

The laser pointer hover events also have one additional functionality. It checks if the laser is hovering over a node. This enables the node to change colors to tell the user if they can place a tower at that node or not. The color is white if nothing is happening, blue if the player has enough money to place the tower at that location, and red if the player doesn't have enough money for that location.

```
Node Color Changing
This pseudocode aims to explain how the nodes change color.
This is done in the laser hover script

When the laser points at a gameobject

    If the gameobject's tag is "node"

        If the build manager is not building a tower
            Return

        If the node has a tower at that location
            Set the color of the node to the original color

        Else if the player has enough money for the tower being build
            Set the node's color to blue

        Else
            Set the node's color to red indicating not enough money
```

Code Listing 4.11. Pseudocode to change the color of the nodes

4.5.6 Skills

To implement skills into the game, a canvas was used to display all the information with a button for each skill, allowing the user to upgrade each skill. The canvas is attached to the left VR controller by having it follow the controller's position and rotation. Additionally, the skill canvas only shows when the left controller's touchpad is pressed. This was done by using event listeners like how the pause menu was implemented. Whenever the touchpad is pressed, the function toggles the canvas to be active.

Finally to implement the Money skill, all that was needed was a variable to hold the money multiplier. This value is set to one initially so that before it is upgraded, enemies gives their normal amount. After the skill is upgraded, whenever an enemy dies, they enemy gives its base money multiplied by the money skill multiplier.

4.5.7 Stage Progression

In order to implement this staging functionality, the game constantly looks at the current wave the player is on. Every five waves starts a new stage by setting the corresponding stage gameobjects to be active so that as soon as the player is getting used to the map, more challenges appear. However, each stage was activated at the end of the wave right before the new stage in order to let the player know that more game elements have been added and allow them time to plan and adjust to the new elements. Finally, some minor changes were made to the enemy movement scripts to make sure that enemies do not go to the underworld before they are allowed to.

4.5.8 Gameplay Speeds

Many tower defense games allow players to pause the game and speed up the game. As such, pausing and fast forwarding the game was added to accommodate this. To increase the game speed, when players click on the fast forward button, a static variable is set to true indicating that the game is speed up. Then, for game elements that rely on time (ie. enemy movement, projectile movement, countdown timers, etc.), everything having to do with either movement or countdowns were multiplied by two to simulate the game is being fast forwarded.

Unfortunately, pausing could not be implemented by setting the timescale to zero. This interfered with other aspects of the game like the tower UI not showing when it's supposed to and the laser pointer not showing sometimes. To get around this, a three static booleans was used to indicate if the game is paused or not. If the pause button is pressed, the pause menu is activated, or the tutorial is open, the respective boolean variable is set to true. Then, in all aspects of the game that relied on time as noted earlier, a small snippet of code is added at the beginning of each update function. This small snippet of code checks to see if any of these booleans are

true. If any are true, the update function for scripts with the snippet of code immediately returns without allowing the gameobject to do anything, thus stopping all of its functions.

```
Pausing
This pseudocode aims to explain how gameobjects that rely
on time are paused in the game. This code is put at the top of
the update function for all gameobjects that rely on time
(ex. enemies, wave spawning, projectiles, etc)


If the Player UI pause button is active or the tutorial is
active or the pause menu is open

    Return immediately
```

Code Listing 4.12. Pseudocode to pause gameobjects

4.5.9 Tutorial

The tutorials were primarily handled with a UI system that pauses the game to give the player a chance to see information. In it, boolean variables are created to store information regarding the tutorials, specifically if a given tutorial has been seen before, if the UI should be cleared on the next interaction, if the player is pointing at tower spawners, if an enemy has been killed, and many more tutorials explaining the game. Almost all of the variables are set to false, meaning that the tutorial has not been seen yet or that the event has not occurred yet. This is to ensure that the flags have the right values to start so that the tutorials activate when the right conditions are met. The conditions vary on what the tutorial is meant to tell the player, with different numbers of conditions to be met for different cases. While `clearText` is set to true, this is irrelevant as `clearText` is set based on the current text.

When a tutorial is opened, its respective boolean variable is set to true, indicating that that tutorial has been seen and making it so that the tutorial isn't continuously opened. Also set to true is the `tutorialOpen` variable, which was created when testing revealed that meeting the conditions to open another tutorial overrode the currently opened tutorial. Two string values are also set to display the information to the player, and to set a tag used for the switch statement in

`Advance()`. Also set is the aforementioned `clearText`; it is false when there is more text to be shown in the given tutorial and is true when the tutorial has reached its last chunk of text. Finally for some of the tutorials, an arrow is created in the scene at the location of certain gameobject in order to show what the tutorial is referring to.

```
Tutorial Conditions
This pseudocode aims to explain  what happens when
a tutorial's condition is met

If condition for a tutorial is met
    Set the tutorial UI gameobject to active
    Play the tutorial audio sound
    Pause the game
    Display the tutorial text on the tutorial UI
    Set up the next text to be displayed
    Set clearText to false
    Create a tutorial arrow at the location of the gameobject being refered to
```

Code Listing 4.13. Pseudocode for tutorial popup conditions

The tutorial UI is actually a button; interacting with it runs the Advance function, which checks if `clearText` is true and a switch statement that lets the tutorial text exist in separate chunks. If `clearText` is true, it sets `tutorialOpen` to false and hides the button by running a function called `Hide()`. In the Hide function, the tutorial UI deactivates, making the UI disappear, and unpauses the game time allowing the game to resume and the next tutorial to open once conditions are met. If `clearText` is false, then the code goes through a switch statement based on the `currentText` tag. The switch statement replaces the current tutorial text with the next chunk of text, as well as setting the `currentText` tag to reflect what is currently displayed. Like the initial tutorial texts, it also sets `clearText` based on whether or not there is still more text to be displayed. If `clearText` is false and `Advance()` is called again, it goes through the switch statement again and, since `currentText` was changed in the previous instance of the Advance function, it goes to the relevant case.

```
Tutorial Button
This pseudocode aims to explain what happens when
the Tutorial button is pressed


If clearText is true
    Resume the game state
    Hide the Tutorial gameobject

Else
    Check what the next text is going to be

    If the next text matches the case
        Set the info text
        Set the next text
        Set ClearText to true or false depending if it is last info for that tutorial
```

Code Listing 4.14. Pseudocode advancing through a tutorial

4.5.10 Sound

In order for the game to feel complete from the user side, sound effects needed to be added as an audible confirmation that something has happened, such as an enemy being defeated, a tower being built or upgraded, selling a tower, transporting between maps, and taking damage. As the game is 3D, objects that make sounds are given an Audio Source component, while the camera is given an Audio Listener component; this system emulates 3D sound by making it seem like sounds come from specific sources within the world when the player hears it. Sound effects were gathered from freesound.org. The sound effects for player damage, skill upgrade, selling a tower, the waterfall, and enemies dying are under the creative commons zero license, meaning that they are completely free to use, no credit is needed, and they can be used in any project as they are in the public domain. The sound effects for the gong, bell, lava, and constructing or upgrading towers are under the attribution license, meaning that they are free to use as long as credit is given, but they can be used in any project. The sound effects for tutorial notifications are under the attribution noncommercial license, which is the same as the attribution license with the exception that it can only be used for noncommercial projects. All sound effects under the attribution license and links to the licenses themselves are included in the appendix section.

While the sound effects gathered were a good start for the project, all of them required editing to be considered usable in the game. Almost all of the sound effects were either too loud

or too long. Some sound effects had empty space before the desired sound played, making it seem like the sound effect played late. As such, all of the sound effects were edited in Audacity to better serve the project. All of the sound effects had their volume lessened. The kamaitachi death sound effect was sampled from a much larger file of an audio recording of a shrieking mustelid; as only one of the sounds in the file was needed, the rest of the file was removed as it did not serve the game. The player damage sound was edited slightly, though the most important change was to the file type so that all of the files were of the same type. The tower construction sound was also shortened as it went on too long an only parts of it were needed; the sound also got progressively louder, so a fade out effect was added to counteract this. The gong was made quieter and repeated application of fade out shortened and softened the sound clip without editing the speed of the sound or abruptly cutting it off. The oni death sound effect, the tower destruction sound effect, and the tutorial notification sound effect were the most useable upon acquisition; the oni death sound simply needed to have the empty space before the desired sound played to be removed, while all of them were made softer. The tutorial notification sound effect was not made as soft as the rest since it needs to get the player's attention as opposed to being background noise to help immerse the player.

Finally to implement into the game, each object that needed sound was given an audio source component with the desired audio. Then in the code, when one of the actions mentioned earlier happens, the audio source is played. For some gameobjects, like enemies, where the enemy needed to be destroyed, the game audio would never play since it was attached to the gameobject that was destroyed. So instead, the entire gameobject is not destroyed at first and only the parts of the gameobject needed to be destroyed were destroyed.

4.5.11 Misc.

To give users the ability to tell what each tower is before hovering over the spawners to find out what the tower is, icons were placed above each spawner. Each icon represented the weapon that the tower uses. For example, the archer tower spawner has an arrow above it. To add additional aesthetics to the icons, all the icons rotate around to give the player nice perspectives of the weapons.

Many of the UI elements in the game are 2D and can be hard to see at certain angles. To relinquish this problem, a small script to have certain UI canvas follow the VR camera was put in place. Inside this script, the given UI canvas transform is set to look at the VR camera using the `LookAt()` function. Thus, no matter where the VR camera is, the UI canvases are always at an ideal angle for the user to see.

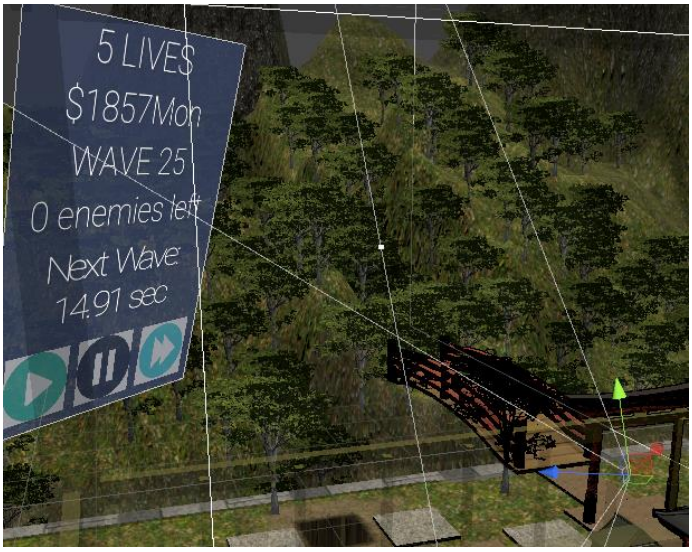


Figure 4.15. UI canvas looking at the VR camera

To allow users a slightly easier time placing towers, the option to hide the spawn towers and the gong was added to the game. This makes it so that the spawn towers and gong do not interfere with the laser pointer since they all have box colliders that the laser pointer can interact with. This was implemented by creating a controller event for the left trigger click. When the

player clicks the left trigger, all the spawn tower and gongs are set to inactive, making them disappear. When clicked again, all game objects reappear.

If users ever got tired of the sound in the game, a mute system was implemented. All this required was adding another button in the pause menu canvas and creating a static variable for muting. When the mute variable is true, the game audio does not play and when it is false, the game audio plays as normal.

4.6 Website

4.6.1 Introduction

The Purpose of the website was to allow users another way to see the enemies, towers, and other game objects closer. This would allow them to see details they could not see while playing the game. For this website, Heroku was used as the platform to make the website live and available to use by anyone. Heroku is cloud platform that allows users to take applications and turn them into live websites [24]. Heroku was used for this website because the team members were familiar with it and it does not cost any money to obtain a unique URL for the website. Heroku works similar to GitHub in the sense that users push their code to a server. This similarity made creating the website through Heroku easy.

Figure 4.16. Example close up 3D model viewing on the website

4.6.2 Server

The server.js file is the file that the Heroku servers run to start the website. The server.js file creates the server that the website runs off of and also maintains all requests made by webpages. For example, the webpage might need to an image, so it makes a request to the server and then server sends the image back to the webpage. So for this project, the server.js file handles all request like going to new webpages, images, filmbox (.fbx) files, cascading style sheets (.css), and other JavaScript(.js) files. Additionally, when working locally, the server.js file allows you to run the website using localhost, which is a temporary network to run the website off of. For production and testing of a website, this is very useful.

4.6.3 Website Structure

The website had three main types of pages. The first is the home page where the user is able to access the web pages with all the models of a specific type. This leads into the second type of webpage where all the models of a specific type are displayed to the user. These models are displayed in a boxed scene with a description of the model off to the side. If the user wants to take a closer look at the model, they can then access web pages with just one model. This is the

last type of web page where there is only one scene and model in the webpage that the user can interact with to control what they want to see specifically. Finally, the player is able to access any of these web pages through the nav bar at the top of the each web page.

4.6.4 Three.js

Three.js is a JavaScript library that allows people to create 2D and 3D computer graphics by using WebGL (another 3D computer graphics library) as its foundation. This library was used for showing 3D models because it had good examples of how to implement the computer graphics and knowing WebGL was not necessary in order to implement the complex 3D applications [30]. In order to get the desired webpage, the loader/fbx example provided by the Three.js library was used as the base code. For the web pages that only needed one model in the scene, two things needed to be done. The first was change the .fbx file that the program loads to the .fbx file of the model to be displayed. Since none of the models used had animations attached to them, the second change made to the file was getting rid of the mixer portions of the script. These mixers is what makes the animation for the model work and since the models didn't have animations, this would cause errors and not render the model. Some of the models also had transparent textures when loaded in so the object's material needed to set to false in order for the model to work as intended.

The controls for these scenes also needed to be changed slightly. The scenes used orbital controls which allows the user to get a 360 degree view of the model. Since some of the models did not have textures on the bottom of the model, the orbital controls also needed to be changed so that the user cannot rotate the camera below the floor. This was done by altering the `maxPolarAngle` in OrbitalControls.js to be *Pi/2*. Finally, the lightings for each model needed to be done individually to make sure the model and its textures are seen well.

```
Getting Rid of Transparent Material
This pseudocode aims to explain how fbx files are added
to the three.js scene. This part of code is located
inside the loader.load() portion of code

Load the fbx file and send the object to a funtion
    Traverse through the object's children
        If the child is a mesh
            Set cast shadows to true
            Set receive shadows to true
            Set transparent material to false

    Scale the object to the desired size

    Add the object to the scene
```

Code Listing 4.15. Pseudocode for loading an .fbx file

For web pages that needed multiple scenes and models, the loader/fbx example was altered to have arrays rather than single variables. For example, the container variable, camera variable, etc. were made into arrays and the function that used them were altered to fit the new arrays as well. Additionally, in all the scenes created in all web pages, the models were very small. To alleviate this, the models were scaled up so they fit nicely in the scene. Finally for the web pages with multiple scenes, the orbital controls were not used. Instead, the cinematic camera was used to still allow the user to see all of the model. The cinematic camera used in one of the examples in the Three.js library had the camera go through the floor to get a view from the bottom of the models. This was not needed, so the cinematic camera was altered to never go past the floor and the speed of the rotation was slowed down.

```
Cinematic Camera Position
This pseudocode aims to explain how the cinematic camera is
prevented from going below the floor

If the Y angle is greater than 190 degrees
    set the Y angle to 10
Else
    Add 0.1 to the Y angle

Set the camera x position to the radius multiplied by the Sin of the X angle
Set the camera z position to the radius multiplied by the Cos of the Z angle

If the Y angle is less than 170
    Set the camera y position to the radius multiplied by the Sin of the Y angle
Else
    Set the camera y position to the radius multiplied by the Sin of 170

Make the camera look at the center of the scene

Update the camera's world matrix
```

Code Listing 4.17. Pseudocode for changes made to the Cinematic Camera

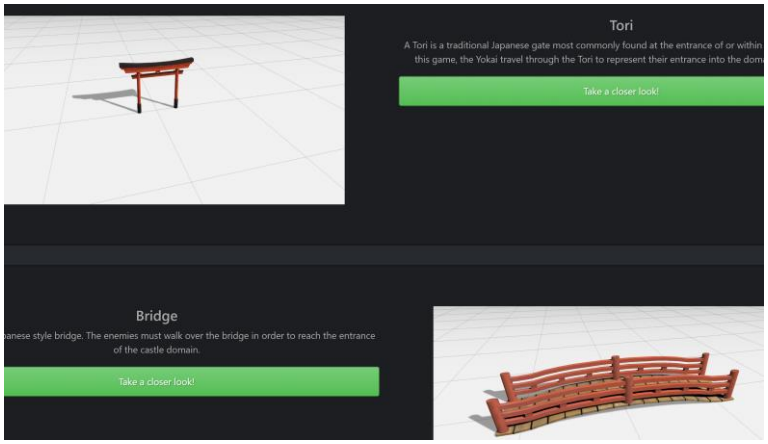


Figure 4.16. Multiple scene on one webpage

Chapter 5: Art

In this chapter, the art section of the project is covered. The chapter is broken up into six sections covering the background research done for the art in the game. This chapter also explains the process that it takes to create the art and implement it into the game engine. Lastly, this chapter covers different Unity engine features that add particle systems, animations, and terrain building for the game.

5.1 Background

Japanese Art has many variants throughout its history. Whether it is their ancient performance art, ceramics, or very recognizable architecture, Japanese Art has evolved over time dramatically and has developed its own unique style. Before the Edo period (1600's), not much hand drawn art was available except for religious prints and scrolls. The medium of painting and drawing on paper wasn't introduced until a Chinese derived art style called Yamato-e was introduced. One of the first novels ever, Tale of Genji, was written and drawn in this art style around the 11th century. Some characteristics of Yamato-e style art include "Aerial perspectives, precise detail, clear outlines, and flat colors." This art style, while similar to a later derivation, was only seen in castles and in the houses of feudal lords until the Edo period when the style evolved from Yamato-e to Ukiyo-e style [19].

Ukiyo-e art was introduced in the Edo period from 1604 until 1868, and continues to be an easily recognizable art style in 2018. Ukiyo-e art, as stated before, is a direct variation of its previous art style, Yamato-e. The differences between these two art styles is that Yamato-e portrayed paintings depicting war, leadership and royalty, while Ukiyo-e portrayed leisurely activities of middle class families, as well as the same subjects Yamato-e were painted about

[31]. Another difference between these two art styles is that Ukiyo-e was much easier to mass produce and was a lot cheaper to purchase because of the use of wood block carvings to make the paintings. These wood block carvings act like a stamp, the artist carves out the wood block and paints the carved area, finally stamping it on the sheet of paper. There are usually multiple blocks with different colors assigned to the areas and different designs to add more detail to the painting. Since these paintings were easy to mass produce, they spread around the world when Japan started to trade with other nations. This is the reason why this style of Japanese art is still known around the world, and is why this project is taking influence from the style [19]. For this project, artistic analysis was performed on Ukiyo-e paintings to find influence to design the game around stylistic aspects of the medium. Like the figure pictured below (figure 5.1), many other Ukiyo-e paintings looked very artistically similar to each other. Some containing more detail or color, but most staying stylistically the same with a clear precise outline around all objects.

Figure 5.1 Example of Ukiyo-e art, taken from UKIYO-E PRINT HISTORY [31]

The implementation of this art style is shown in this game from the textures of the models, the menu screens (shown below in figure 5.2), and the interfaces. The main implementation into the game from Ukiyo-e style was the clear black outline of the objects. The

black lines are important to the game because it provides a unique look to the models without sacrificing any detail from the texture. The shader used was a free Unity script called quick outline. By applying this shader to materials, when the game is loaded, the shader creates a black background on the object so that there is a black outline around the object (shown below in figure 5.3). Most objects were able to have this shader applied to it but some could not have it because the black background sometimes interfered with other object's materials. For example, figure 5.4 shows a bridge cross the river in the game that leads to the grid where the player is in the game. The bridge and torii gate clearly have a black outline to them, but the nodes, rock wall, and pathways do not. The reasoning behind that is because the shader shows through other objects that have the same shader applied to them.

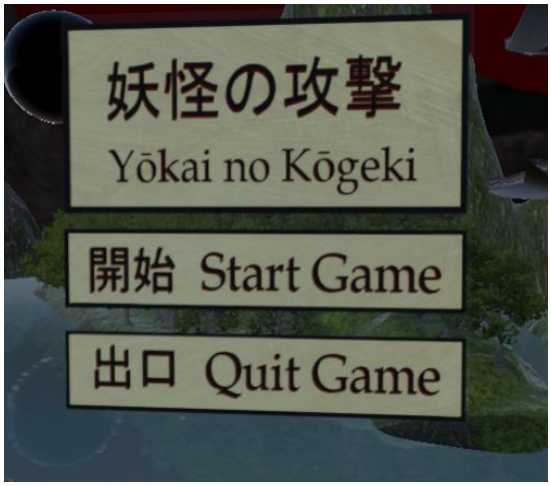


Figure 5.2 In game title / menu screen

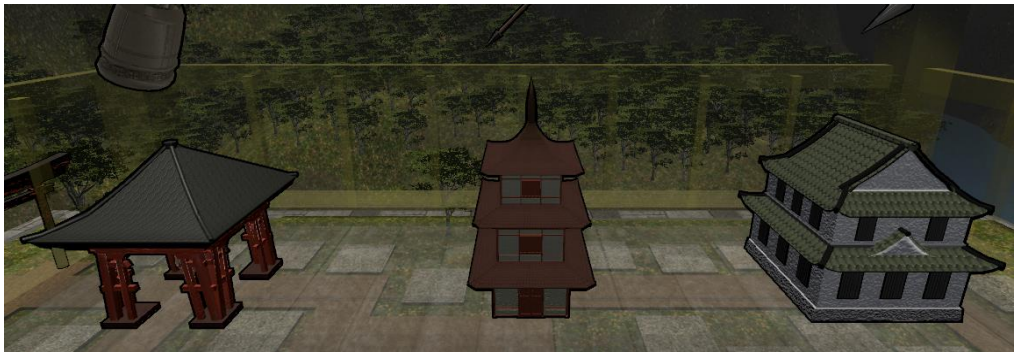


Figure 5.3 All towers featured in game with black outline

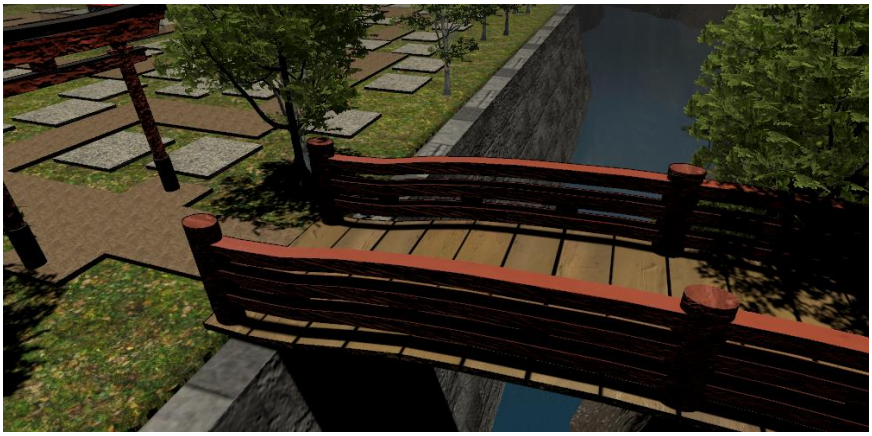


Figure 5.4 Bridge and torii gate models with black outline

5.2 Reference Art and Sketching

To implement this art style into the game through all of the assets that needed to be made, a lot of work and research was needed to find relevant reference art, as well as to create initial sketches for in game assets. When looking for reference art for assets in the beginning of this project, research obtained from the background chapter helped find the ideas of what art references to search for. When the research on the background was finished, all of the initial ideas of what was going to be implemented into the game was established. These ideas of implementation turned into initial assets for the game by the means of brainstorming and research expeditions to local temples, castles, and parks for inspiration. With the brainstorming and inspiration, the art style of the assets was initially drafted. These first drafts came in the form of humans as the placeable towers, and Yōkai (shown below in figures 5.5 and 5.6).

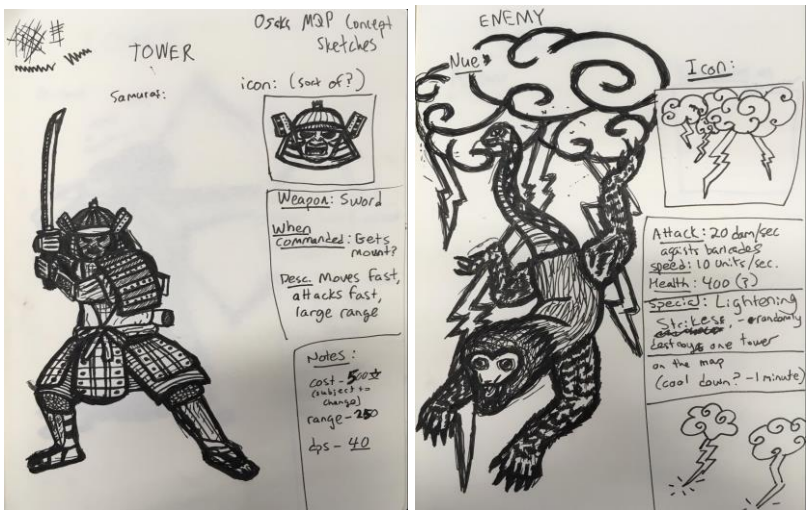


Figure 5.5(L) and 5.6(R) Initial sketch for samurai warrior and Nue enemy respectively

The process of finding exactly what to use for the enemies and towers was to look up Yōkai and Japanese weapons. As already mentioned in chapter 2 section 2 of this report, after getting an initial idea of Yōkai , the group talked about what Yōkai would be good for the game and why.

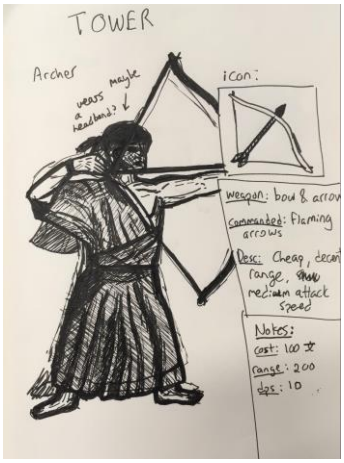
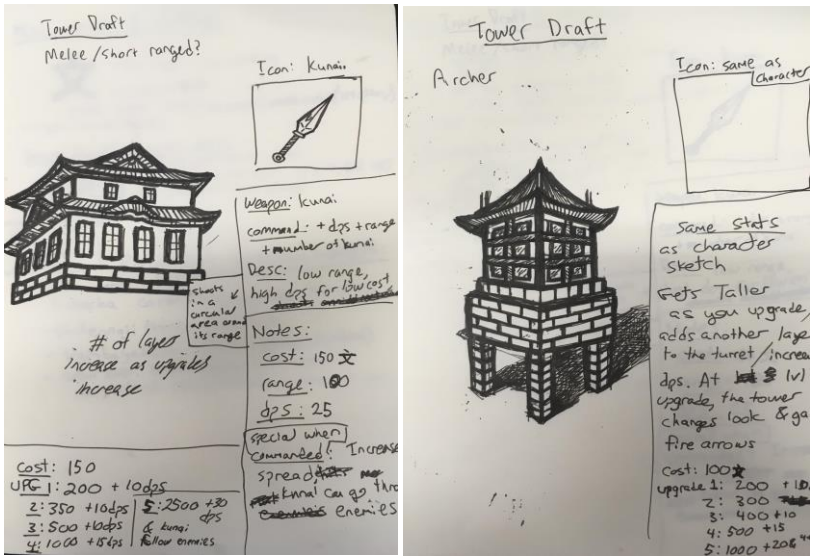


Figure 5.7 Initial sketch of archer tower character

The reason that the game does not have humanoid 3D models, like the one sketched in figure 5.6, was due to time constraints. There was not enough time to make all the details from scratch for all of the assets in the short amount of time to create this project. Since that idea was shortly lived, concepts of the towers themselves were sketched to help create their initial 3D

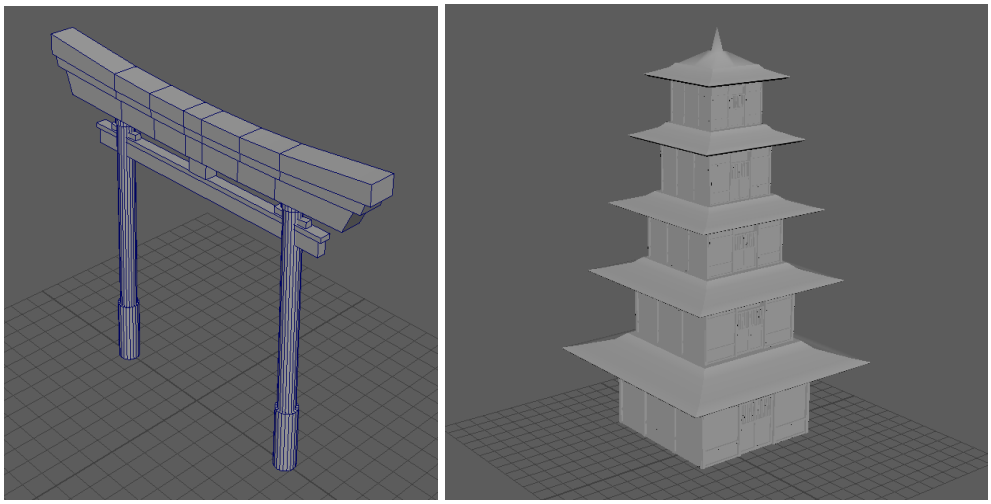
models. The concepts sketched were inspired from personal travels to traditional Japanese towers and turrets across Japan, which are sketched below in figures 5.8 and 5.9.



Figures 5.8 (L) and 5.9 (R). Initial sketch for the kunai tower and archer tower respectively

5.3 Modeling and Texturing

When creating the models for the VR game, a few programs were used for different purposes. This project utilized Autodesk Maya because of the group's familiarity with the program and its easy availability for students. Autodesk Maya was used to make the initial assets for the towers, projectiles, other simple non-organic shaped objects such as torii gates and the player tower as shown below in figures 5.10 and 5.11.



Figures 5.10 (L) and 5.11 (R). Initial 3D models for the torii gate and archer tower respectively

For the organic shapes, a program called Autodesk Mudbox was used. This program was used because it was a free sculpting tool that was fairly easy to pick up and learn. The initial plan was to use Zbrush, but the software is very expensive and wasn't readily available at the lab where the game was worked on. Mudbox does not have nearly as many features as Zbrush, and no one in the group knew how to use this program before this game. However, because it is from the same company as Maya (Autodesk), there were some similarities with the interfaces and wasn't that difficult to understand. This program was used to create objects such as the enemy character models below in Figure 5.12.

Figure 5.12. Drafts for Kamaitachi model made in MudBox

All of the objects made in Maya and Mudbox also have a UV map, which is basically a 2D representation of the 3D object. All of the UV maps are hand made through maya's UV editor and then exported to texture the object. A large amount of time and effort went into making sure that the UVs were the right proportion to the object, as well as trying to hide the seams in the objects. The reason that it is difficult to do this is because the way Maya makes UVs sometimes messes with the proportions of the UVs compared to the object, as seen below in figure 5.13.

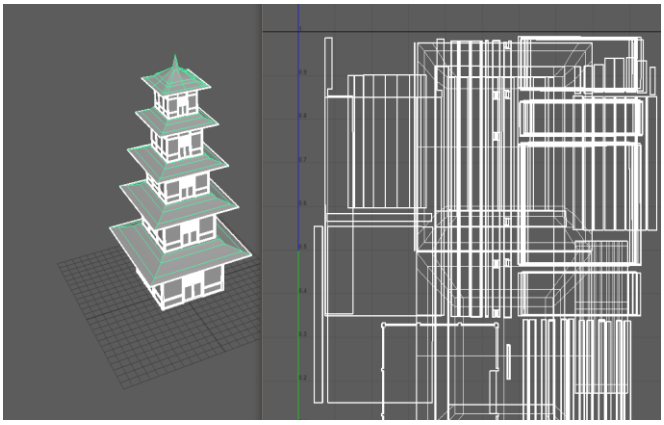


Figure 5.13. Maya's UV editing screen (UVs for 3D object shown on right)

When the UVs were finished, they were exported as .tiff files. The program used to texture these UV maps was Adobe Photoshop. Photoshop was used to texture these files because of familiarity with the program, as well as its versatility with creating textures and attention to detail and polish.

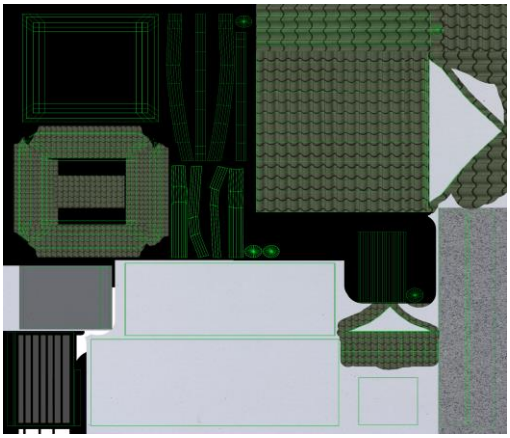


Figure 5.14. Texture for Kunai tower with UVs visible with green outline

The process of making these textures had to do with reference images, and using techniques to mimic the texture. For example, a texture for the Oni model (Figure 5.15 below) utilized the “filter tool” by adding both a Gaussian Blur to the image and then adding a noise filter to mimic a skin texture.

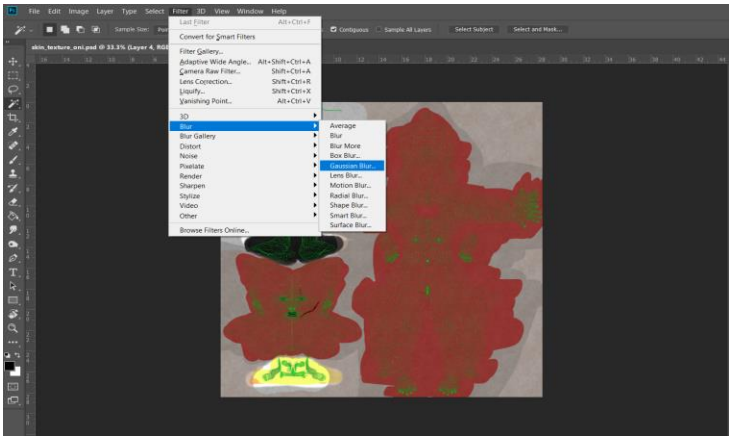


Figure 5.15. Example of filters used to mimic textures like skin

Photoshop was also used because of its simple way of creating a normal map for the object (example shown below in figure 5.16). A normal map is a texture that applies to the UVs that gives the illusion of 3D bumps and textures while not taking up any more polygons than the object has. Normal maps can be applied in both Maya as well as in Unity.

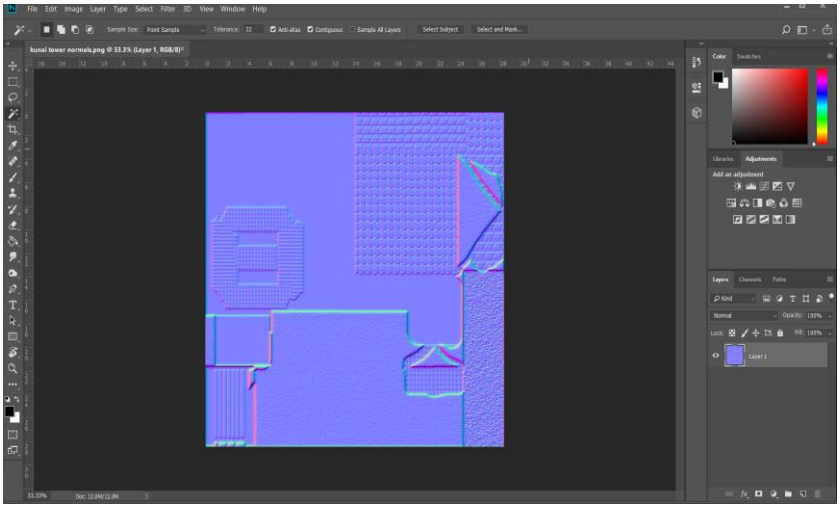


Figure 5.16. Kunai tower texture after the 3D normal map filter is applied

5.4 Animation

Animations are a staple part of any game. Giving objects the ability to move and to be able to interact with the game adds to the realism and immersion into the environment of the game. There are a couple of difference types of animation used in this project, the first one being simple object animations in Unity. This game engine has a built in animation tool that creates animations that you can directly assign to an object. Unity can do complex animations, however, the group is more familiar with rigging in other programs such as Autodesk Maya. Unity's animation tool has an option to transform, rotate, or scale the object for animating.
These simple transformative animations are applied to objects such as the gong, the icons above the towers, and the bell on the bell tower.

For more complex objects, such as the Yōkai models, a few different programs were used. For the Oni model, a website called Mixamo[2] was used that automatically animates humanoid objects was used (Website interface is shown in figure 5.17). The reason that this

[2] Website link for Mixamo: https://www.mixamo.com/#/

website was used was because of the time that was given to finish the project, and because of the group's experience with 3D animation and rigging was minimal. Using this website saved the group days of working to rig and animate the Oni model. As useful as this site was for rigging and animating humanoid objects, it couldn't animate quadrupeds (being that walks on 4 legs).

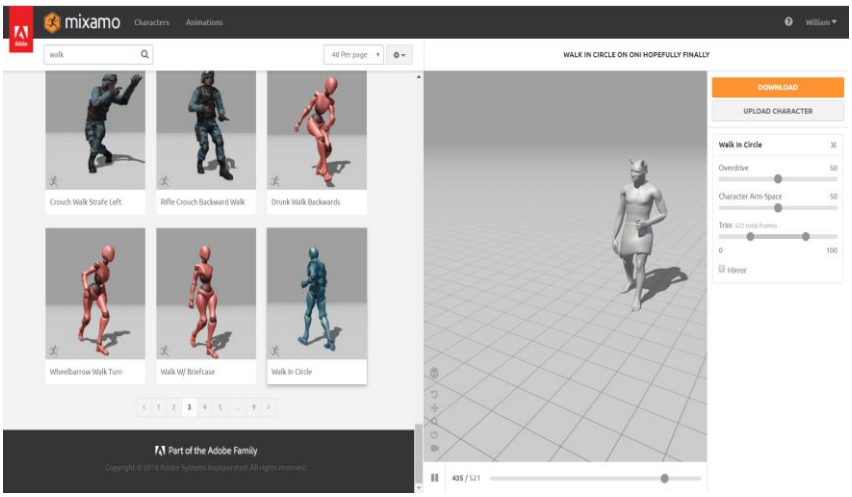


Figure 5.17. Mixamo interface for animations

Since the other two models used as enemies in the game were quadrupeds, they had to be hand rigged and animated. The group has had little experience rigging and animating in 3D, and no one had ever rigged and animated a quadruped. Both of these enemies are quadrupeds so the rigging was very similar from one object to the other. Unfortunately, there was not enough time to fully develop the Nue model and plans for including the Nue had to be scrapped and replaced with a stronger Oni variant. Similarly, there was not enough time for the Kamaitachi model to be rigged and animated. The model was changed to be floating on a cloud because of time restraints, as shown below in figure 5.18. Choosing a cloud was because of the theme of the Kamaitachi, that the Yōkai travels in storms and typhoons. For more information on the enemies, refer back to Chapter 2.2: Japanese Influences.

Figure 5.18. Kamaitachi model riding on top of a cloud

5.5 Implementation of Art Assets into Unity

When the Models, textures, normal maps, and animations were all finished for the modes, they were then imported into Unity. The 3D models made in Maya and Mudbox can be exported in many ways to fit into a Unity scene. For this project, the models were exported as .obj and .fbx files. The models exported as .obj were models that did not have animations attached to them, examples of this type of model would be all of the tower models. Objects were exported as .fbx for two reasons: If it had an animation attached to it, or if the model was being used on the website. The models needed to be exported as .fbx because it was the easiest way to implement into the website. For other imports, such as textures and normals maps, it did not matter what file type it was. For consistency, all of the textures and normal maps are .png files.

Once all of the files are imported into the asset list, the 3D models all need to be assigned a material in order to show the textures and normals maps made for them. Materials can be easily made through Unity and the textures and normal maps can easily be dragged onto the material through the material interface. The material interface in Unity is shown below in figure 5.19.

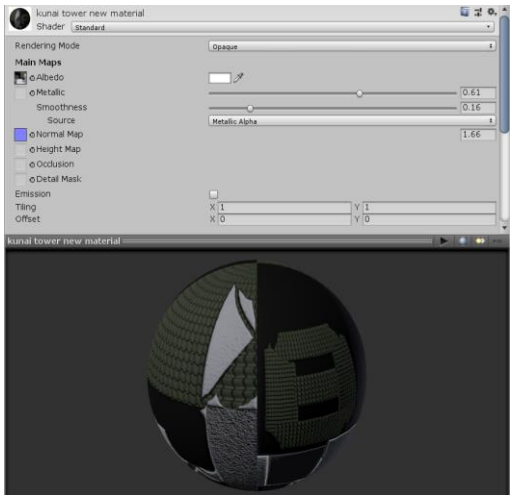


Figure 5.19. Material interface in Unity

Once the material has the normal map, texture, and whatever other file types are attached to the material (specular maps, bump maps), it can now simply be dragged on to the 3D model that uses those textures. Once applied, the settings on the material can be tweaked to change the appearance of it. Settings that can be changed include the intensity of the normal map, and the metallic and smoothness of the texture. Importing models, textures, and normal maps is easy to implement into a Unity scene.

Something a bit more difficult to apply to a Unity scene is animations. Animations in this project were all imported as .fbx files, just for the sake of consistency. Animations when imported need to have a rig be applied to them that already exists in order to work. This rig is the avatar for the animation; it is expected that whatever game object the animation is attached to has a similar rig as the avatar. Naturally, if the game object and the avatar use the same rig, there are no problems. All animations for an entity are placed in an animator, which serves as a state machine that controls when animations play.

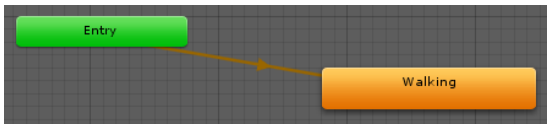


Figure 5.20. Simple animator with only one animation for Oni

Figure 5.14 shows an example of an animator used with the Oni enemies. In this instance, there is only one animation: walking. Walking is transitioned from Entry, which means that Walking is played when the object enters the scene. More animations can be added to the animator and are linked by transitions which use scripts to define the behavior of the transition, such as when the transition should occur (e.g. pressing the jump button to transition the player from a grounded state to the jump animation, which flows into a falling animation when the jump ends, which continues until the player either lands and transitions back to a grounded state or interacts with another object, such as a hangable ledge or an enemy). An animator component is then added to the game object to be animated, using the state machine created as the controller and the rig to be animated as the avatar.

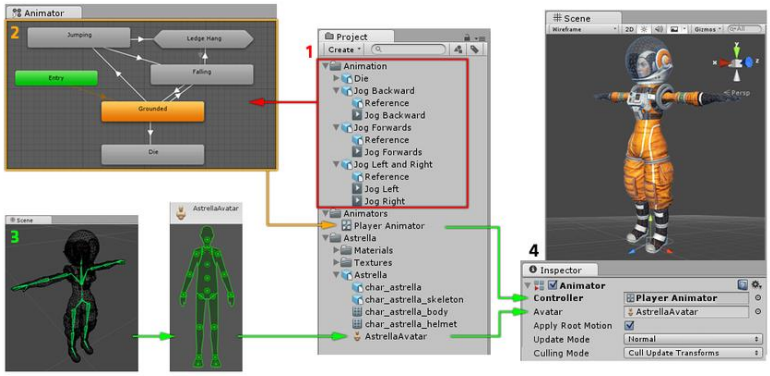


Figure 5.21. Animation implementation using an animator in the Unity Manual

5.6 Creating the Game Environment

5.6.1 Environment Building

As stated before, there were quite a few reasons why Unity was used as the game engine for this project. One of those reasons was also because of Unity's terrain building tools. This tool allows one to build a level for a game with little to no experience in the engine. There are

different brushes to choose from which change the height and color of the map. The right side of figure 5.22 below shows the interface for the terrain builder in Unity.

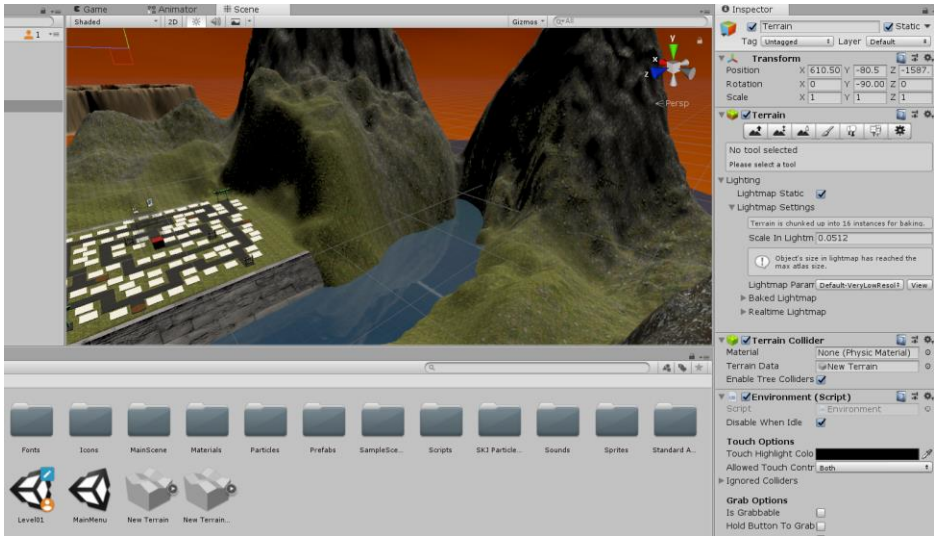


Figure 5.22. Unity's built in terrain interface

The environment for both levels of this game were made with the terrain assets provided. The terrain made was one of the most important parts of the levels when it came to building the environment. To texture the environment, the Unity standard assets, or a free set of game assets provided for free by Unity, were used to texture the surface of the environment.

Another aspect of the game that used Unity standard assets was the trees. Due to time constraints, models provided for free by Unity were used to help build the environment. These models are already textured and are optimized to fit into any scene. Trees in Unity can be applied to the "Tree brush" in the terrain interface. Trees, and essentially any model, can be applied as a brush and can be placed anywhere on the terrain. There are settings, as shown in figure 5.23 below, that this brush that can change the size of the brush, the height of the trees, as well as the spread of them when used on the environment. The tree models automatically are applied to the surface with little or no gap between the tree and the terrain.

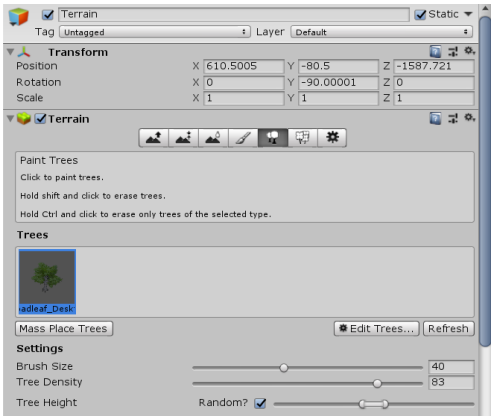


Figure 5.23. Tree brush interface for terrain

5.6.2 Skybox

To create the background of the game, an asset called a skybox is used. A skybox is a cube that surrounds the entire scene and does not move or scale depending on your location. A skybox is composed of six images, all of which cover the inside surface of a cube. A good way to visualize what a skybox would look like is to imagine the game being inside of a very large cube, the skybox being the cube itself. The images for the skybox itself were made an edited in Photoshop and imported onto the skybox. The interface for the skybox is shown below in figure 5.24.

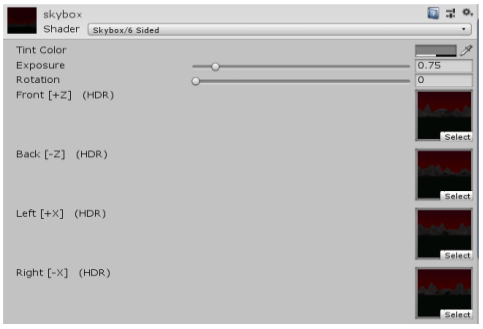


Figure 5.24. Skybox interface

5.6.3 Particle Systems

Particle systems used are tweaked to help them fit into the game and add more depth to the design of the level. For example, one particle system in the level is the sun. There are over 20,000 particles appearing around a sphere to impersonate the way the sun acts. This particle system uses many tools provided by Unity to make it look the way it does, as shown below in figure 5.25. These tool were how long the particle stays on screen (random variation from 5 to 10 seconds) and the speed of the particle (from 0 Units to 1.5 Units). Some particles also randomly rotate, and have different colors and alpha values. These particle system tools allow for a lot of variation in particles the users can make. For example, a few more particle systems used in this level are waterfalls, and lava (shown below in figures 5.26 and 5.27). The waterfall particle system was made with the help of an online tutorial[3] while the lava is a variation off of that tutorial however, many of the values were changed to meet desired visuals. All of these particle systems add a lot to the immersion of the player in the level.

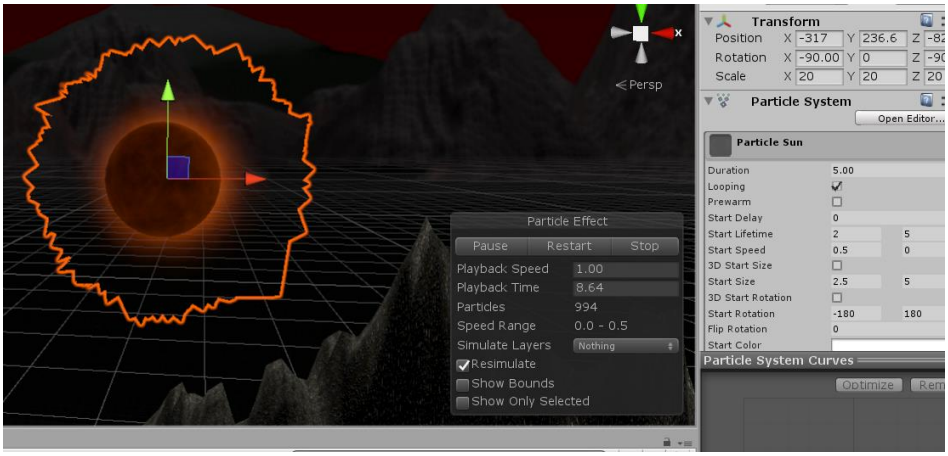
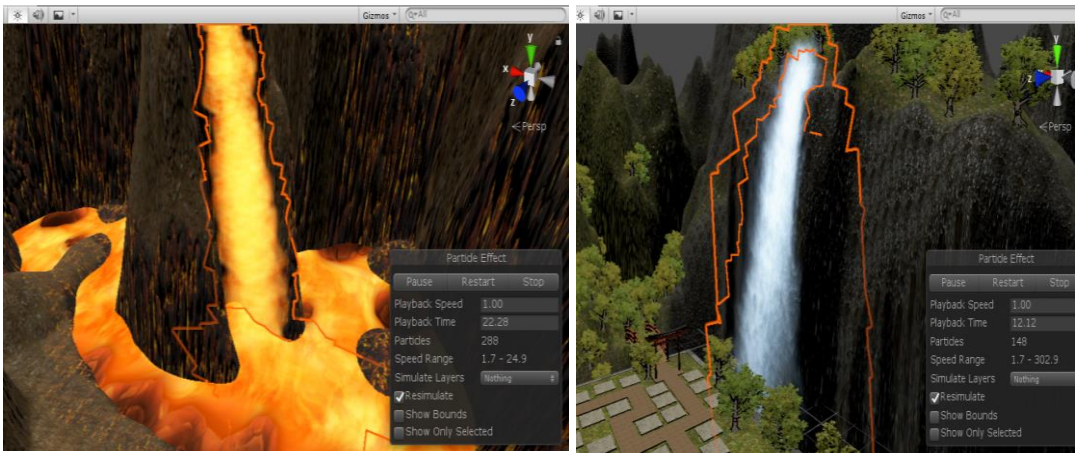


Figure 5.25. Particle system interface in Unity with a particle system set up for the sun

[3] Online particle system tutorial for waterfall and lava particle systems: https://www.youtube.com/watch?v=XhSp8nFLUi4&t=407s



Figures 5.26 (L) and 5.27 (R). Particle systems for lava and waterfalls respectively

Chapter 6: Testing

Chapter 6 talks about the testing process and analysis of the tower defense game. A brief overview of what the goals for testing were are described along with a description of the survey used and the process of how testing was done. Finally, an analysis of the testing is mentioned along with the major changes that came along as a result of the research study.

6.1 Overview

Naturally, all of the effort put into the project would mean nothing if there was no confirmation that our experience goals were met. As such, testing had to be done; having people who are not working on the project to test it and share their perspectives can either confirm or deny that the experience goals were met. It is also important that context behind the individual testers in terms of their experience with similar projects is also considered and recorded. This is important because different experiences or a lack thereof serve as a lens with which the project is seen: an individual with no experience with tower defense have vastly different responses to the project than somebody who is a veteran tower defense player. Information that was deemed pertinent included past experience with tower defense games, as well as how the project compares to them in terms of difficulty and complexity, how well the controls were implemented, the quality of the art, the strategic challenge presented by the game, and the experience generated by the game.

6.2 Survey

In order to gauge player responses to the game after playtesting sessions, a survey was developed. There were twelve questions in the survey addressing the game itself or the tester's

past experience with similar games. The last two questions were open ended questions asking for suggestions, comments, and giving the tester one last chance to ask the researchers questions regarding the game. The first three questions were primarily about tower defense games in general: ‘do you think tower defense games are hard?’, ‘how does this game compare to other tower defenses in terms of difficulty and complexity?’, and ‘what do you find hard about playing tower defense games?’. These questions served to gauge the tester’s own experience with tower defense games and where the game that they just played falls with others in its genre in terms of difficulty and complexity. This lets the researchers know the context of the rest of the data gathered in the survey; it is expected that an individual with little experience with tower defense games and believes them to be incredibly difficult gives vastly different responses to an individual who is a master of tower defense games.

The fourth question asks about the controls themselves, as they are the interface between the player and the game. If the players believe that the controls are intuitive and makes sense, then the game more easily opens up to them regardless of past experience with the mechanics. If the player feels that they are fighting with the controls, then no amount of explaining the mechanics helps as the controls are the foundation from which the mechanics are built. If playtesters respond that the controls are unintuitive, that is a massive warning sign and steps must be made immediately to rectify the problem as bad controls paint the rest of the experience in a negative light regardless of how polished the rest of the game is.

The fifth question asks the tester for how they would describe the art in the game. As the art should create the Feudal Japan-styled backdrop of the game, the tester’s description of the art should line up with the intended setting. Of course, art in this case refers to the models, textures,

and environment design as not every tester was able to play with sound as either sounds weren't implemented yet or the tester did not have headphones to connect to the VR headset.

The sixth and seventh questions were pertinent to the gameplay itself; they both asked the tester to reflect on the strategic challenge presented by the game. The sixth question asked what some of the strategic challenges were while playing the game. This could be anything from the number of enemies to deal with at a time, the management of resources, having to look around to see the rest of the map, or figuring out where to place towers. The seventh question specifically asks by how much the dual world system added to the difficulty and complexity of the game. As the addition of a second map that needs to be managed at the same time as the initial map was a core design element of the game, it should be expected that the dual world system adds to the difficulty and complexity in a meaningful or significant way.

The eighth, ninth, and tenth questions asked the testers to reflect on the experience as a whole. The eighth question asked very simply how the tester felt defending their castle from invading enemies. This provides a raw and unprocessed response from the tester in terms of how they felt interacting with the mechanics of the game. The ninth question focused the experience on one of the experience goals: 'how much does the game make you feel like a leader'. Naturally, as this is one of the experience goals of the game, a higher response means that the game has successfully sculpted a scenario that places the player in command of a castle to be defended from invaders. The tenth question asked the tester to use three words to describe the experience of playing the game, specifically asking to avoid more generic and abstract terms such as fun. Responses to this question are how the tester felt about the experience without any descriptive fluff; they must reflect on their experience and describe it using three words. The

closer that the three words reflect the intended experience, the better a job the game does at conveying systems and mechanics that should elicit such descriptions.

6.3 Testing Methodology

The process of playtesting formally started by sending an invitation to playtest the game via the Takemura Labs mailing list, letting everybody in the lab know that the game was ready for formal testing. Individuals then schedule themselves by filling in cells in a Google Sheets page with their name; which cell they chose determined the date and time of their playtest session.

The session begins by presenting the tester with an informed consent form, stating the names and email address of the investigators, an introduction to the study, the purpose of the study, the procedures that would be followed, risks and benefits from the study, confidentiality, a reaffirmation of the voluntary participation in the study, how to withdraw from the study, and contact information for more information about the project and the participants' rights. Signing the form states that the participant has read and understood the form and wishes to participate in the study and is confirmed by one of the investigators signing the form as well.

At the start of the playtest session, the tester is asked about their experience with tower defense games and VR games which serves as context to the tester's expectations for the game. Afterwards, they put the VR headset on and are given the controllers; once they are ready, the game starts and they are brought to the main menu, with options to either play the game or quit. After pressing play, the tester starts to play the game, starting with tutorials explaining the controls and the base mechanics of setting up towers. The tester plays the game for approximately 5-10 minutes, though they are allowed to continue playing if they choose and

have the time to spare. After the tester has finished playing, they are taken aside to complete the aforementioned survey. The survey ends with questions about demographic information about the individual for more context about the individual tester's experience. Once the survey is completed, the session is complete.

6.4 Results

Since the game was edited throughout the playtesting period to fix the problems encountered, the results of our research study improved as time went on. The analysis for this study was performed in a before phase and an after phase to show how the major changes made to the game affected our intended player experiences. The cutoff between the before and after phases is represents by the game before and after adding in features like sound and finished art, as well as game balancing and rearrangements of staging. There were a total of six playtesters before the game changes and six playtesters after, providing an even split for the testing phases. An example of one of the analyses, the game shouldn't be too easy but it should also not be too hard. At the beginning of the study, the game was noticeably too easy for playtesters. However, after performing the game balances, the game became more difficult, thus giving a hard difficult for those without tower defense experience and a medium difficulty for those with tower defense experience.

When looking at the success of our intended player experience, some of the experiences were more successful than others. The least successful out of the intended player experiences was getting the player to feel like a Japanese commander. Looking at figure 6.1, when playtesters were asked if they felt like a leader on a scale from 1 to 5, there was an even split between whether people felt like a leader or not in the early responses. However, after the addition of

sound and more changes to the environment, more of the later playtesters said to have felt like more of a leader. Some of the playtesters even said they felt tall, powerful, and like a commander when asked how they felt at the top of the castle. Figure 6.2 shows a word cloud of all most of the words people responded with when asked that question.

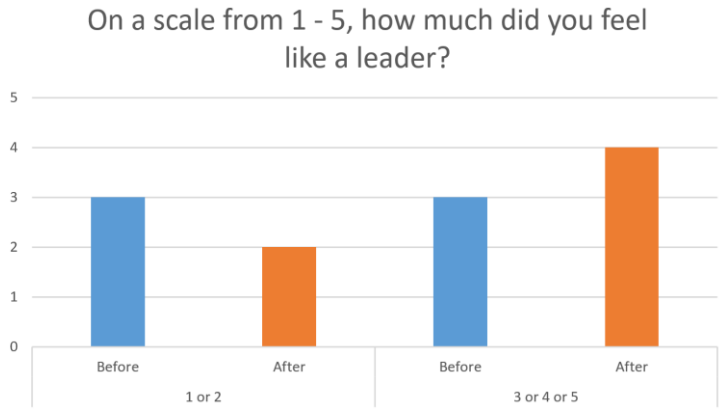


Figure 6.1. Results on how much people felt like a leader

Figure 6.2. Word cloud of how people felt at the top of the castle

For the strategic challenge presented to the players, this was a big success after the first few playtesters. Initially, the stages presented in the game made it so that the player did not have to worry about the underworld until later in the game because they could kill enemies before they took pits to the underworld. After rearranging the stages and introducing an enemy spawn point in the underworld sooner, many of the playtesters felt the two level design provided a nice

strategic challenge. As seen in figure 6.3, more playtesters felt the two level design was more difficult after major changes were made to the game. When asked what some of the strategic challenges they faced while playing, many of the later playtesters mentioned their management between the overworld and the underworld. Additionally, the limited perspective view the players had on the map provided a great strategic challenge for the players. Although it was not asked specifically in the survey, many of the playtesters mentioned that it was hard to get a visualization in their head of the entire map. Playtesters mentioned that they had to look around constantly to make sure they are not missing something on the map and said it was difficult to manage their attention for certain aspects of the game which is exactly what was intended by using VR.

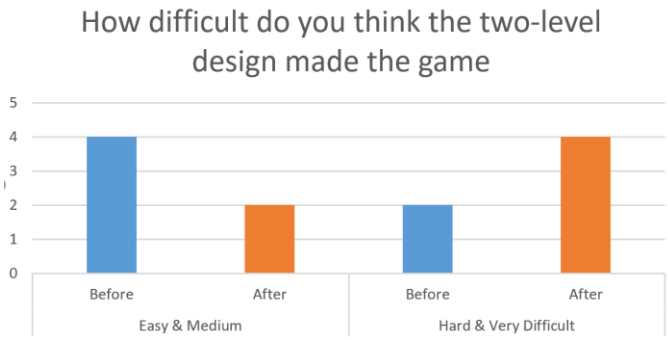


Figure 6.3. Results on the two-level map design

Lastly, getting players to notice the influence of Japanese culture in the game was a huge success all throughout the study. When asked how they would describe the art in the game, many of them mentioned that they liked all the Japanese culture that was present in the game. As shown in the word cloud in figure 6.4, most people wrote words like Japanese, traditional, and beautiful which accurately describes what the intended player experience was aiming for. Additionally, almost all of the playtesters thought that the scenery was very nice and fitting for the game.

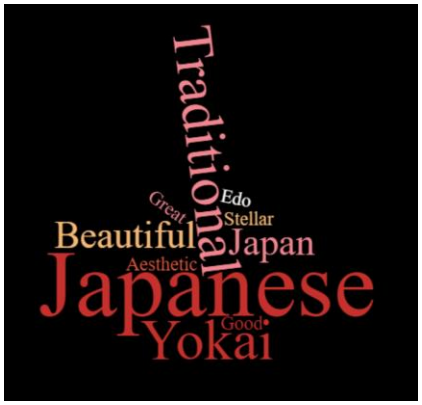


Figure 6.4. Word cloud of how people described the art in the game

Overall, the research study could be considered a success due to the improvement on the intended player experience throughout the study. While the experiences were not as present at the beginning of playtesting, it was more prevalent in later stages. As proof that some of these player experiences were achieved, figure 6.5 shows a word cloud of words people used when players were asked to list three words that described their player experience. Some of the more popular words used by playtesters were: strategic, beautiful, Japanese, busy, and historical. These words can clearly be tied back to our intended player experiences which was a great surprise.

Figure 6.5. Word cloud of how people described their experience

6.5 Major Game Changes

6.5.1 Game Balancing

Throughout playtesting, there were noticeable game balancing issues. Towers would deal too much damage or too little damage. Upgrading towers were too overpowered at one point during the playtesting period. The bell tower's stun could stun enemies infinitely and enemy strength was static at early stages of playtesting. This made it so that players never experienced some aspects of the game and could never lose. However, throughout the playtesting period, many game balancing changes helped fix these issues.

Initially, the kunai tower was far superior to the archer tower. The kunai tower dealt double the damage for half the price compared to the archer tower. To alleviate this, both towers were made to have the same price to build. Additionally, the initial damage of both towers were heavily reduced and had similar values; the kunai tower having slightly more damage than the kunai tower but less range. To compensate for this massive reduction in initial damage, the damage upgrading system was also changed to scale better than it did before. Every level, the damage of each tower would double as well as add an additional base amount. The equation used for this was `currentDamage + constant + (towerLevel * baseDamage)` where `currentDamage` is the tower's current damage, `constant` is some fixed value added to the damage, `towerLevel` is the tower's current level and `baseDamage` is the initial damage of the tower. This, however, made upgrading too powerful and made the game feel too easy for some playtesters. One of the playtesters also noted that when the speed of the tower's increases at level 5 and level 10, it essentially doubles the strength of the tower in addition to already doubling the damage.

Finally, the damage upgrading system was change to add a base amount of damage plus an additional amount equal to the tower's level multiplied by some constant. The equation used for this was `currentDamage + constant + (towerLevel * multiplier)` where `currentDamage` is the tower's current damage, `constant` is a fixed value added to the tower every level, `towerLevel` is the tower's current level, and `multiplier` is a fixed amount of damage for the tower. This made upgrading more reasonable in the large scale of things and made playing difficult for newer players but still gave a slight challenge to experienced players. The increase in speed of the towers at level 5 and level 10 were also reduced to not give a tower too massive of a boost. Even though a decent upgrading system was found, it could still be improved, however due to time constraints, further game balancing was not pursued.

Another problem found during early stages of playtesting had to do with the bell tower. Whenever an enemy got stunned, it another bell tower hit that enemy, it would reset the stun time of the enemy. So, if there are many bell towers near each other, it could stun enemies infinitely as seen from one of the playtesters. To fix this problem, the enemies become immune to stun for a brief duration of time after getting stunned. This makes it so that the enemy has a little bit of time to move before it can get stunned again, allowing progression and preventing players from only creating bell towers.

Finally, the last game balancing done to the game had to do with the enemies. Initially, the enemies had a static health value throughout the entirety of the game. For later stages of the game, players would be able to one-shot enemies which made the game super easy even though the number of enemies per wave increased. Instead of using static health values, each enemy's health now scales based on the current wave the player is on. The equation for determining

health of an enemy was `initialHealth + (initialHealth * (waveNumber * multiplier))` where `initialHealth` is the base health of the enemy, *`waveNumber`* is the current wave the player is on and `multiplier` is the percentage to increase the health by. As the stages went on, the multiplier changed to compensate for the increase in strength the player had.

6.5.2 Gameplay Speed

It was initially thought that the game was going to be very hectic in the mid to late stages of the game. So, pausing was allowed at any point in the game to give players the ability to place, upgrade, and sell towers on the go. Contrary to initial thoughts, many playtesters mentioned that pausing the game at any time made the game easy because they could pause the game and go back and forth between the overworld and underworld to see what was happening to make sure they were safe. Since part of our intended player experience was to use the limited knowledge players have as a strategic challenge, it was decided to only allow the players to pause the game in between waves. This made it so that while enemies are on the map, the player would have to constantly check the overworld and underworld. While in between waves, players could take a break and set up new towers to prepare for the new game elements added.

6.5.3 Sound

While users were playtesting, it was noted that playtesters did not always know when they had a new tutorial or when they lost a life. Due to this, sound was implemented to help notify the player that certain events are happening in the game. For example, tutorial notification sound, enemy death sound, player damage sound, building tower sound, selling tower sound, and more were added.

6.5.4 Stages

Some of the playtesters early on mentioned that the underworld was underwhelming since they could just kill enemies before they get to the pits. Therefore, the stages were introduced in a different order. As seen in the figure 6.6. below, the third torii gate is introduced one wave earlier right after in order to make sure that the player has to do something in the underworld.

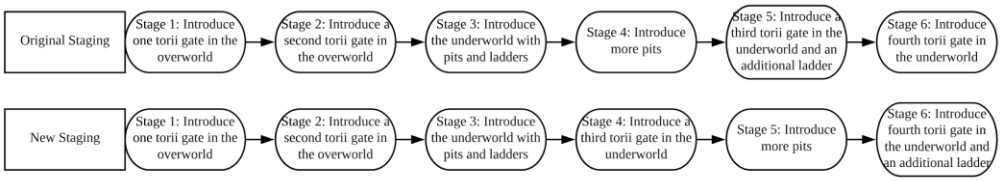


Figure 6.6. Staging flow chart before and after testing

6.5.5 Tutorials

While some of our playtesters were not as fluent in english as other playtesters, there was still a lack in communication for the tutorials. Players would get confused easily about when the game was starting and did not have a sense of where things were on the map. To help with this issue, a tutorial arrow was added to the tutorials. This arrow would point to the relevant gameobjects in the scene that the player is currently reading. For example, when the tutorial is talking about the Yōkai coming towards the torii gate, the tutorial arrow appears above the torii gate to clearly show what the tutorial is talking about. Additionally, some of the content of the tutorials were changed to get rid of confusion about certain parts of the game. For instance, at one point the tutorial says the game starts after one of the messages, however this is not true and there were more tutorials to go through. This was changed to mention the game is starting later on in the tutorial.

6.5.6 Art/Environment

Throughout the research study, the art changed drastically. At the beginning, the bell tower, kunai tower, and the player tower were not completely finished. However, after the first few playtests, all three of these towers were finished with their final design and textures added to the scene. This significantly helped the Japanese cultural experience goal because these were major aspects to the Japanese culture design. Additionally, the environment was changed a lot to make it feel a little more realistic. Particle systems were added to simulate waterfalls and lava. The skybox was fixed to be seamless; trees and fog were added to the terrain to get nature to be more prevalent in the scene. Textures were added to the tower spots and pathways. Finally, a bridge was added to the environment so that enemies have to cross the bridge to reach the torii game.

Chapter 7: Post Mortem

7.1 What Went Right

Ultimately, this project succeeded in what it set out to do. A tower defense game was successfully implemented in virtual reality, and the three tower types and three enemy types were also implemented successfully. The dual-world mechanic succeeded to provide an extra layer of complexity and strategy to the game, and the game was given a steady difficulty curve with a slowly expanding world with increasingly powerful enemies. Measures were made to create balance in the game so that any strategy is viable when executed skillfully yet no single strategy overruled the rest. Finally, the art implemented successfully represented Japanese culture and the art style the project aimed to achieve.

The team was also able to operate efficiently together, with no real disagreements flaring up between individual members. Clear roles and tasks were assigned to maintain overall efficiency, with a dedicated lead programmer, secondary programmer, and artist. When a task was given to any individual member, the individual saw to it that the task was completed on time. When the technical implication was nearing completion, the team was able to assist the artist in developing and polishing art assets.

7.2 What went Wrong

While the project as a whole was a success, there were some flaws in the project that the team would have liked to work on or polish further. The team was only able to work on the project for about 80 days; the first week was spent on coming up with a game idea to work on while the second week was spent planning out the development of the chosen game idea. As a

result, the team lacked the time to animate the kamaitachi model or to even develop a unique model and animation for the third enemy type. There was also only time to develop one skill for the player to unlock and upgrade. Time constraints were further pressed by the fact that one of the team members was also taking another class remotely to satisfy diploma requirements and, therefore, could not entirely focus on the project.

7.3 What Was Learned

The main thing that the team learned during the project was first-hand experience with game development as a small team, specifically while working in Unity, a mainstream game engine. More specifically, the team learned about developing a virtual reality game using Virtual Reality Tool Kit and the unique development challenges inherent to virtual reality. Additionally, the team learned about the website library Three.js to help display 3D models on a website. Lastly, the team further refined their skill in basic Unity developments.

The development process was also informed by what the team had learned while immersed in the Japanese culture and the lab culture at Takemura Labs. The team also learned about the development process of making a game of this scale and how to take responsibility for the task given to them. Finally, the team learned what components go into handling team dynamics and how to work well together.

The team also learned a lot about how to implement and work with art assets in Unity on a large scale. The artist on the team increased his knowledge on the interfaces of all of the programs used in this project, such as: Photoshop, Unity, Maya, and Mudbox. The artist also learned how to understand the code behind the game and what it does.

7.4 Future Developments

Due to time constraints, many ideas created during the planning phase of development had to be scrapped. As mentioned in what was intended to be implemented but wasn't, there was the Nue. While the functionality for the Nue was complete, the art assets were unable to be implemented. Having a separate model to immediately differentiate between types of enemies would serve the player by the two enemy types looking completely different and would serve the aesthetic of the game by providing more enemy variety.

Furthermore, there were ideas that had to be scrapped early due to scoping issues. This includes multiple enemy types, towers having health and having the ability to be destroyed by enemies, and ground units to directly fight enemies and repair damaged towers. Ground units would cost less than towers but would also require food to heal damage from Yōkai; food production would likely be a constant, with a skill to increase production. There are many Yōkai in Japanese mythology with very specific behaviors. Before the project became VR, there was an idea to have the map illuminated by lanterns and an enemy type that specifically extinguishes the lanterns to lower the player's visibility. This enemy would have been based on the Abura Akago Yōkai, which takes the form of a small child and licks up the oil from oil lamps. To counter food production for the ground units, there would be a passive enemy based on Hiderigami which causes droughts. A certain enemy type, based on Bakeneko, Nekomata, and Kitsune (ordered in increasing threat level), would be able to disguise themselves as ground units and require the player to manually target them. The presence of the Kitsune enemy would spawn Kitsunebi, weak fireball enemies that attack towers but also alert the player to the Kitsune on the map.

More tower types were initially planned before settling on the three used in the project. One tower would behave differently from the others and would be placed on the path the Yōkai

travel on. The barricade prevents Yōkai from continuing down the path, requiring them to break the barricade to proceed. This ‘tower’ would have been incredibly cheap to offset the fact that it is destructible and does not damage enemies. However, the ultimate function of having an option that impedes the Yōkai from moving was already handled by the bell tower. Moreover, on major upgrades, the tower models would change to reflect the increased and functionality.

One of the ideas thrown around early in development was to make the game multiplayer. As this option was considered after the project shifted into virtual reality, multiplayer functionality would have been asymmetrical; the player using the VR headset would play the game as normal while the second player would command the invading Yōkai, choosing where they spawn, what order they spawn in, and which paths to prioritize. Ultimately, it was found that balancing the game for a single player was difficult enough on its own and that a solid single player experience was more desirable than a messy multiplayer experience.

Finally, if given more time, animations would have been implemented to give more life to the kamaitachi. Instead of having the kamaitachi float on a cloud, the original intention was for it to have a walking animation. But since this was not possible with the time constraints, the cloud was put in and the whole enemy was moved up and down as it moves through the map.

Chapter 8: Conclusion

Over the course of three months, this group made a VR tower defense game in Unity game engine. The game had three main experience goals: have the player feel like a commander, provide strategic challenges to the player throughout the game, and portray the Japanese influence through the art and style of the game. In order to accomplish these player experiences, the game was designed in such a way to accomplish this in the best way possible. The two level design and large amount of routes to the castle made for a great strategic challenge. Putting the player at the top the castle looking down at the towers and enemies helped make the player feel like a commander. Finally, the Ukiyo-e art style and the traditional Japanese architecture and mythological creatures helped portray the Japanese influence on the game.

The game started out as a simplistic tower defense design based off of a tutorial found online. Since this design was far from the complexity required for this game, many changes were made. First the game was changed so that it is playable in VR as well as enabling the user to interact with towers. Additionally, a two level design allowing enemies to find new routes to the castle was implemented. Finally, many more minor additions were added to the game in order to polish the functionality of the game. In terms of art, it started off with sketches based on background research of possible enemies and towers in the game. From those sketches, 3D models were made, textured, and brought into Unity to implement the art assets. Once the art assets were in Unity, other aspects were added to the scene like the environment, particle systems, and shaders to add to the detail of the art in the game.

After implementing the game in Unity and testing over the course of the later half of development, the game had sufficiently met the experience goals. Early testing showed that the experience goals initially seemed unmet. A more complete version of the game with game

balancing, sound, improved environment, shaders, and models implemented, as well as other technical tweaking, showed a noticeable increase in the success of the intended player experience. As a result, the project succeeded in delivering the product that it had aimed to deliver.

References

[1] Alex Rubens. 2013. The creation of Missile Command and the haunting of its creator, Dave Theurer. (August 2013). Retrieved October 9, 2018 from https://www.polygon.com/features/2013/8/15/4528228/missile-command-dave-theure

[2] Anon. 2018. Best virtual reality SDK for VR development in 2018. (July 2018). Retrieved October 9, 2018 from https://thinkmobiles.com/blog/best-vr-sdk/

[3] Anon. Ambush! - Tower Offense - Apps on Google Play. Retrieved October 9, 2018 from https://play.google.com/store/apps/details?id=air.funfactory.ambush&hl=en

[4] Anon. Anomaly games – Official site of strategy games franchise created by 11 bit studios. Retrieved October 9, 2018 from http://www.anomalythegame.com/

[5] Anon. Blue Ogre. Retrieved October 9, 2018 from http://okami.wikia.com/wiki/Blue_Ogre

[6] Anon. Fire Attack. Retrieved October 9, 2018 from http://www.intheattic.co.uk/fire_attack.htm

[7] Anon. Greenhouse. Retrieved October 9, 2018 from http://www.intheattic.co.uk/greenhouse.htm

[8] Anon. Kiwi games industry booms. Retrieved October 9, 2018 from http://www.gameplanet.co.nz/news/i138234/Kiwi-games-industry-booms/

[9] Anon. Main board of the original Game & Watch (Ball). Retrieved October 9, 2018 from https://retrocomputing.stackexchange.com/questions/1613/main-board-of-the-original-game-watch-ball

[10] Anon. Orcs Must Die! Unchained. Retrieved October 12, 2018 from https://orcsmustdie.com/#!/en

[11] Anon. Plants vs Zombies Video Games - PopCap Studios - Official EA Site. Retrieved October 12, 2018 from https://www.ea.com/studios/popcap/plants-vs-zombies

[12] Anon. Products. Retrieved October 9, 2018 from https://unity3d.com/unity?_ga=2.260461582.2094885780.1534904523-773745064.1532594833

[13] Anon. Red Ogre. Retrieved October 9, 2018 from http://okami.wikia.com/wiki/Red_Ogre

[14] Anon. Safebuster. Retrieved October 9, 2018 from http://www.intheattic.co.uk/safebuster.htm

[15] Anon. The great Nintendo Handheld Games from the 80's ...! Retrieved October 9, 2018 from https://www.gameandwatch.ch/en/game-watch-information/all-60-games.html

[16] Anon. The Symbol of Osaka. Retrieved October 9, 2018 from https://www.osakacastle.net/english/history/index.html

[17] Anon. The Ultimate Yōkai Guide. Retrieved October 27, 2018 from https://www.wattpad.com/357572877-the-ultimate-yōkai-guide-kamaitachi

[18] Anon. The Ultimate Yōkai Guide. Retrieved October 27, 2018 from https://www.wattpad.com/340886513-the-ultimate-yōkai-guide-oni

[19] Anon. Ukiyo-e Movement, Artists and Major Works. Retrieved October 9, 2018 from https://www.theartstory.org/movement-ukiyo-e-japanese-woodblock-prints.htm

[20] Anon. Unity Services - Collaborate. Retrieved October 1, 2018 from https://unity3d.com/unity/features/collaborate

[21] Anon.Vermin. Retrieved October 9, 2018 from http://www.intheattic.co.uk/vermin.htm

[22] Anon. VIVE Virtual Reality System. Retrieved October 9, 2018 from https://www.vive.com/us/product/vive-virtual-reality-system/

[23] Anon. Welcome to VRTK · VRTK - Virtual Reality Toolkit. Retrieved October 9, 2018 from https://vrtoolkit.readme.io/docs

[24] Anon. What is Heroku | Heroku. Retrieved October 9, 2018 from https://www.heroku.com/what

[25] Anon. Yokai.com. Retrieved October 9, 2018 from http://yokai.com/

[26] Bruce Bower. 2011. Humans: Kids perceive ownership principles: Concept of property rights may come naturally to preschoolers. *Science News*179, 13 (July 2011), 17–17. DOI:http://dx.doi.org/10.1002/scin.5591791317

[27] Damien McFerran. 2010. Feature: The History of the Nintendo Game & Watch. (February 2010). Retrieved October 9, 2018 from http://www.nintendolife.com/news/2010/02/feature_the_history_of_the_nintendo_game_and_watch

[28] Duy-Nguyen Ta Huynh, Karthik Raveendran, Yan Xu, Kimberly Spreen, and Blair Macintyre. 2009. Art of defense. *Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games - Sandbox 09*(2009). DOI:http://dx.doi.org/10.1145/1581073.1581095

[29] Ereny Bassilious et al.2011. Power defense: A video game for improving diabetes numeracy. *2011 IEEE International Games Innovation Conference (IGIC)*(2011). DOI:http://dx.doi.org/10.1109/igic.2011.6115113

[30] Jos Dirksen. 2015. *Learning Three.js - the JavaScript 3D Library for WebGL: create stunning 3D graphics in your browser using the Three.js JavaScript library*, Birmingham ; Mumbai: Packt Publishing.

[31] Lawrence Bickford. 1993. UKIYO-E PRINT HISTORY. (July 1993). Retrieved October 9, 2018 from https://www.jstor.org/stable/42597774

[32] Luke Mitchell. 2008. Tower Defense: Bringing the genre back. (June 2008). Retrieved October 9, 2018 from https://web.archive.org/web/20140203062250/http://palgn.com.au/11898/tower-defense-bringing-the-genre-back/

[33] Mark Cartwright. 2017. Shinto. (April 2017). Retrieved October 9, 2018 from https://www.ancient.eu/Shinto/

[34] Michael Dylan Foster and Kijin Shinonome. 2015. *The book of yokai: mysterious creatures of Japanese folklore*, Berkeley: University of California Press.

[35] Noriko Fujii. The history of Japanese copper coins: Illustrated from the collection of the Currency Museum of the Bank of Japan. *Journal of the Oriental Society of Australia, The Volume 45 (2013)*, 77–92.

[36] Noriko Reider. 2013. *Japanese demon lore: oni from ancient times to the present*, Place of publication not identified: Utah State Univ Press.

[37] Onix-Systems. 2017. What VR platforms are best for game development? – Onix-Systems – Medium. (July 2017). Retrieved October 9, 2018 from https://medium.com/@onix_systems/what-vr-platforms-are-best-for-game-development-b5b65084a2c2

[38] Ori Friedman. 2010. Necessary for Possession: How People Reason About the Acquisition of Ownership. (July 2010). Retrieved October 25, 2018 from http://journals.sagepub.com/doi/abs/10.1177/0146167210378513

[39] Ori Friedman, Julia W. Van de Vondervoort, Margaret A. Defeyter, and Karen R. Neary. 2013. First Possession, History, and Young Children's Ownership Judgments. (March 2013). Retrieved October 25, 2018 from https://onlinelibrary.wiley.com/doi/full/10.1111/cdev.12080

[40] Panagiotis Petridis, Ian Dunwell, Sara De Freitas, and David Panzoli. 2010. An Engine Selection Methodology for High Fidelity Serious Games. *2010 Second International Conference on Games and Virtual Worlds for Serious Applications*(2010). DOI:http://dx.doi.org/10.1109/vs-games.2010.26

[41] Peter Nagy and Bernadett Koles. 2014. "My Avatar and Her Beloved Possession": Characteristics of Attachment to Virtual Objects. (October 2014). Retrieved October 25, 2018 from https://onlinelibrary.wiley.com/doi/full/10.1002/mar.20759

[42] Rich. 2015. The Unique Weapons of Ancient Japan. (August 2015). Retrieved October 9, 2018 from https://www.tofugu.com/japan/ancient-japanese-weapons/

[43] Ryan Clements. 2012. Why We Love Tower Defense - IGN. (September 2012). Retrieved October 9, 2018 from http://www.ign.com/articles/2012/09/24/why-we-love-tower-defense

[44] Spencer and Ishaan. 2013. How Muramasa: The Demon Blade Was Made To Feel Convincing. (March 2013). Retrieved October 9, 2018 from http://www.siliconera.com/2013/03/27/how-muramasa-the-demon-blade-was-made-to-feel-convincing/

Appendices

Appendix A: Informed Consent Agreement Form

Informed Consent Agreement

Investigators: Jason Abel, jabel@wpi.edu; Adam Moran, asmoran@wpi.edu; Will Suriner, wesuriner@wpi.edu

Introduction: You are being asked to participate in this research study on Toyonaka Tower Defence. Before we begin with the study, we must make sure you are fully informed about the purpose of the study, the procedures to be followed, and any benefits, risks or discomfort that you may experience as a result of your participation before you agree to participate. This informed consent agreement form provides all of this information so that you may make a fully informed decision regarding your participation.

Purpose of the study: This study investigates gameplay of a Toyonaka Tower Defense (name subject to change) by researching the different player experiences we hope to achieve through the tower defense game such as: 1) immersion through Virtual Reality; 2) strategic challenges through various level design choices; 3) Japanese culture through the story, art, and gameplay. Obtaining this data through this study is needed in order to understand the experiences of users without the influence of the creators. The data gathered from this study will help in revising the gameplay experience of the user and to further improve the immersion and strategic challenges.

Procedures to be followed: You will first be asked about your experience with tower defense games and VR games. Then you will play a game that combines VR and tower defense for approximately 5-10 minutes. Finally, you will be asked questions regarding your thoughts around the gameplay of the tower defense game as well as basic demographic information.

Risks to study participants: Long exposure to computer screen, potential motion sickness from playing in VR. If at any point during the study you do not feel comfortable, please let an administrator know and they will end the session.

Benefits to research participants and others: There are no direct benefits to participating in this study besides your enjoyment of a videogame. The results of this study may enable us to refine and polish our game.

Confidentiality: The information that you give will be handled anonymously and confidentially. Your information will be assigned a ID number; however, your name will not be linked with your participant number. Your name or identifying qualities will not be used in any publication or presentation.

Voluntary participation: Your participation in this study is completely voluntary.

How to withdraw from the study: If you want to withdraw from the study, please tell the researcher and leave the room. Your data will be destroyed. If you would like to withdraw after your materials have been submitted, please contact us at jabel@wpi.edu

For more information about this research, contact:
Jason Abel, Adam Moran, Will Suriner. Email: jabel@wpi.edu

For more information about the rights of research participants contact:
WPI IRB Chair: Professor Kent Rissmiller, Tel. 508- 831-5019, Email: kjr@wpi.edu
WPI's Human Protections Administrator: Gabriel Johnson, Tel. 508-831-4989, Email: gjohnson@wpi.edi
WPI Faculty Advisor: Gillian Smith, Tel. 508-831-6986, Email: gmsmith@wpi.edu

By signing below, you acknowledge that you have been informed about and consent to be a participant in the study described above.

_____ Date: _____
Study Participant Signature

_____
Study Participant Name (Printed)

_____ Date: _____
Administrator Signature

Appendix B: Playtest Survey

Participant ID Number: _____

Thank you for playing our game. We would now like to ask you some questions about your experience while playing the game.

1) In general, do you think Tower Defense games are hard?
    a) Extremely hard
    b) Hard
    c) Medium
    d) Easy
    e) Don't know/No experience

2) If you have played other tower defense games before, how would you compare them to this game in terms of difficulty, complexity, etc?

---

3) What do you find hard about playing tower defense games?

---

4) Did the controls seem intuitive?
    a) Intuitive
    b) Easy after a couple minutes
    c) A little confusing
    d) I never got the hang of it

5) How would you describe the art shown in the game?

---

6) What were some of the strategic challenges you face while playing the tower defense game?

_____

_____

7) How difficult do you think the two-level design made the game?
    a) Very Difficult
    b) Hard
    c) Medium
    d) Easy, no problem at all

8) How did you feel fighting against the Yōkai at the top of the castle?

_____

_____

9) On a scale from 1 - 5, how much did you feel like a leader?
    a) 5 (I felt like a leader)
    b) 4
    c) 3
    d) 2
    e) 1 (I didn't feel like a leader at all)

10) What are 3 words that describe you experience playing the tower defense game? (Try not to use general words like "Fun")
    a) _____
    b) _____
    c) _____

11) Do you have any suggestions on how to make the game better?

_____

_____

12) Do you have any questions?

_____

_____

Demographic Questions

1) What gender do you identify with, if any?
    a) _____
    b) Prefer not to disclose

2) What is your age?
    a) 18 - 29
    b) 30 -39
    c) 40 - 49
    d) 50+

3) Please note your ethnicity. Check all that apply.
    a) Japanese
    b) Chinese
    c) Vietnamese
    d) Korean
    e) Hispanic
    f) African
    g) Indian
    h) Caucasian
    i) Other please specify _____
    j) Prefer not to disclose

4) Please note your current occupation.
    a) _____
    b) Prefer not to disclose

Thank you participating in our research. You have now finished participating. If you have any questions after you leave or would like us to remove you survey data, please email us at jabel@wpi.edu

Appendix C: Sound Effects and Licensing

Attribution License:

https://creativecommons.org/licenses/by/3.0/legalcode

Hammering Nails, Close, A by InspectorJ

https://freesound.org/people/InspectorJ/sounds/406048/

Gong Hit by GowlerMusic

https://freesound.org/people/GowlerMusic/sounds/266566/

Lava Loop by Audionautics

https://freesound.org/people/Audionautics/sounds/133901/

Budda_large bell by Taira Komori

https://freesound.org/people/Taira%20Komori/sounds/212057/

Attribution Noncommercial License:

https://creativecommons.org/licenses/by-nc/3.0/legalcode

FlyffNotif by grey24

https://freesound.org/people/grey24/sounds/316798/