November 2017

Shiki: A Seasonal Virtual Reality Game

Andrew Thomas Walter  
*Worcester Polytechnic Institute*

Joseph Salvatore Crimi  
*Worcester Polytechnic Institute*

Larisa I. Motova  
*Worcester Polytechnic Institute*

Rachel Louise Plante  
*Worcester Polytechnic Institute*

Follow this and additional works at: [https://digitalcommons.wpi.edu/mqp-all](https://digitalcommons.wpi.edu/mqp-all)

**Repository Citation**


This Unrestricted is brought to you for free and open access by the Major Qualifying Projects at Digital WPI. It has been accepted for inclusion in Major Qualifying Projects (All Years) by an authorized administrator of Digital WPI. For more information, please contact digitalwpi@wpi.edu.
Shiki 四季
A Seasonal Virtual Reality Game

A Major Qualifying Project Report
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for graduation by:

Joseph Crimi
Larisa Motova
Rachel Plante
Andrew Walter

Project Advisors:
Professor Jennifer deWinter
Professor Clifford Lindsay
Abstract

*Shiki* is an immersive, first-person experience-oriented game designed for the HTC Vive using Unity 2017. In *Shiki*, players must complete holiday-related quests and subtasks in order to restore the life and balance that a village and its surrounding landscape had previously lost. This project was developed during the months of July to October 2017 in Takemura Laboratory at Osaka University to satisfy the Computer Science Major Qualifying Project requirement at Worcester Polytechnic Institute.
Acknowledgements

Our team would like to extend our most sincere gratitude to the following people who helped make our project possible:

First, we would like to thank our advisors, Professor Jennifer deWinter and Professor Clifford Lindsay, for their continued support, advice, and guidance throughout our project.

We would also like to thank Professor Photchara Ratsamee and the members of Takemura Laboratory at Osaka University for sponsoring our project, providing us the lab space and equipment, and ensuring our stay in Japan was safe, educational, and memorable. We would also like to recognize those members of Takemura Laboratory who playtested our prototypes, as their feedback was greatly helpful in the development of our project.

Finally, we would like to thank John Guerra, WPI ‘16 for providing us custom 3D art assets for use in our project.
# Table of Contents

Abstract 2

Acknowledgements 3

Table of Contents 4

List of Figures 7

1. Introduction 9

2. Background 12
   2.1 Edo Period 12
      2.1.1 Social Structure and Layout 12
      2.1.2 Village Life 13
   2.2 Japanese Mythology 13
   2.3 Cultural Significance of Seasons in Japan 14
   2.4 VR Overview 15
      2.4.1 Motion Sickness and VR 16
      2.4.2 VR Interfaces 17
      2.4.3 HTC Vive 18

3. Design 20
   3.1 Artistic Design 20
      3.1.1 2D Art Design 20
      3.1.2 3D Art Design 24
   3.2 Technical Design 25
      3.2.1 Component Based Software Engineering 25
   3.3 Sound Design 30
   3.4 Story Design 31
   3.5 Game Mechanic Design 33
      3.5.1 Season Mechanic 33
      3.5.2 Crafting Mechanic 34

4. Implementation 37
   4.1 Environment Setup 37
      4.1.1 Hardware 37
      4.1.2 Development Environment 38
         4.1.2.1 Unity 38
         4.1.2.2 Git 38
         4.1.2.3 Software Testing and Bug Tracking 39
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.3</td>
<td>External Libraries</td>
<td></td>
</tr>
<tr>
<td>4.1.3.1</td>
<td>TOML and Nett</td>
<td>40</td>
</tr>
<tr>
<td>4.1.3.2</td>
<td>Style Blending with Neural Networks</td>
<td>41</td>
</tr>
<tr>
<td>4.2</td>
<td>Game Mechanics</td>
<td>44</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Season Mechanic</td>
<td>45</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Crafting Mechanic</td>
<td>50</td>
</tr>
<tr>
<td>4.3</td>
<td>Game Managers</td>
<td>51</td>
</tr>
<tr>
<td>4.3.1</td>
<td>EventManager</td>
<td>51</td>
</tr>
<tr>
<td>4.3.2</td>
<td>QuestManager</td>
<td>56</td>
</tr>
<tr>
<td>4.3.2.1</td>
<td>SaveDataManager</td>
<td>57</td>
</tr>
<tr>
<td>4.3.2.2</td>
<td>Triggers and OnCompletes</td>
<td>59</td>
</tr>
<tr>
<td>4.3.2.3</td>
<td>QuestManager and InteractionEvents</td>
<td>60</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Inventory System</td>
<td>61</td>
</tr>
<tr>
<td>4.4</td>
<td>Art and Assets</td>
<td>64</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Prologue Art</td>
<td>64</td>
</tr>
<tr>
<td>4.4.2</td>
<td>3D Design</td>
<td>67</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Sound Assets</td>
<td>70</td>
</tr>
<tr>
<td>5.</td>
<td>Playtesting</td>
<td>72</td>
</tr>
<tr>
<td>5.1</td>
<td>Playtesting Procedures</td>
<td>72</td>
</tr>
<tr>
<td>5.2</td>
<td>Alpha Playtesting</td>
<td>72</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Feedback and Results</td>
<td>73</td>
</tr>
<tr>
<td>5.3</td>
<td>Beta 1.0 Playtesting</td>
<td>74</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Feedback and Results</td>
<td>74</td>
</tr>
<tr>
<td>6.</td>
<td>Postmortem</td>
<td>76</td>
</tr>
<tr>
<td>6.1</td>
<td>What Went Right?</td>
<td>76</td>
</tr>
<tr>
<td>6.2</td>
<td>What Went Wrong?</td>
<td>77</td>
</tr>
<tr>
<td>6.3</td>
<td>What Could Be Different?</td>
<td>78</td>
</tr>
<tr>
<td>Appendix A</td>
<td>Game Design Document</td>
<td>80</td>
</tr>
<tr>
<td>Appendix B</td>
<td>List of Assets</td>
<td>90</td>
</tr>
<tr>
<td>Appendix C</td>
<td>Playtesting Procedures</td>
<td>92</td>
</tr>
<tr>
<td>Appendix D</td>
<td>Playtesting Feedback</td>
<td>98</td>
</tr>
<tr>
<td></td>
<td>Alpha Notes</td>
<td>98</td>
</tr>
<tr>
<td></td>
<td>Beta 1.0 Notes</td>
<td>100</td>
</tr>
<tr>
<td>Appendix E</td>
<td>Known Issues and Bugs</td>
<td>102</td>
</tr>
<tr>
<td>Appendix F</td>
<td>Task List</td>
<td>104</td>
</tr>
</tbody>
</table>
List of Figures

Figure 1: Hanging Scroll 21
Figure 2: Kiyomizudera 22
Figure 3: A Chinese Port 22
Figure 4: Early storyboard images 24
Figure 5: Code duplicated between two classes 26
Figure 6: Code shared using inheritance 26
Figure 7: Example of implementation where Cat inherits from Dog 27
Figure 8: Simple solution by creating a superclass 28
Figure 9: More complex class structure 28
Figure 10: Shared navigation code 29
Figure 11: Initial class hierarchy design 30
Figure 12: Comparison of JSON and TOML 41
Figure 13: Example of neural network poem 41
Figure 14: Bamboo texture 43
Figure 15: Door and lantern texture 43
Figure 16: View of village 44
Figure 17: Diagram of Shiki’s world representation 45
Figure 18: Diagram showing how season-relative coordinates are calculated 46
Figure 19: Equations for translating between world and season-relative coordinates 47
Figure 20: Example entry in the variant database file 48
Figure 21: Diagram of control flow 49
Figure 22: ToolFunctions.cs script 50
Figure 23: ObjectPickedUpEvent definition 53
Figure 24: Empty DropTargetBehavior component script 55
Figure 25: DeleteableBehavior component script 55
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>Table of component scripts</td>
<td>56</td>
</tr>
<tr>
<td>27</td>
<td>Script to convert a flat list of tasks into a list of task trees</td>
<td>59</td>
</tr>
<tr>
<td>28</td>
<td>Script to check if tasks have been triggered</td>
<td>61</td>
</tr>
<tr>
<td>29</td>
<td>Examples of VR inventory styles</td>
<td>62</td>
</tr>
<tr>
<td>30</td>
<td>Screenshot of inventory interface</td>
<td>63</td>
</tr>
<tr>
<td>31</td>
<td>Finished village scene</td>
<td>66</td>
</tr>
<tr>
<td>32</td>
<td>Same village scene with color saturation reduced to 0</td>
<td>67</td>
</tr>
<tr>
<td>33</td>
<td>In-game landscapes</td>
<td>70</td>
</tr>
</tbody>
</table>
1. Introduction

*Shiki* is an immersive, first-person experience-oriented game in which players must solve seasonal holiday-related quests to restore life and balance to the village in which the game takes place. *Shiki* was developed by our team at the Takemura Laboratory at Osaka University during the months of July through October 2017, for the HTC Vive using Unity 2017.

Our game takes place in an Edo Period farming village. This village had experienced several years of bountiful harvests that the villagers were initially thankful for, but they had come to take it for granted. The villagers quickly used all available resources without regard to preserving the surrounding nature and landscape, which angered the seasons’ gods. The seasons’ gods stripped the village of life and vitality and split the surrounding landscape into four identical quadrants, each representing a season of the year, as punishment for the villagers’ greed. However, there were no villagers left to worship the gods, and the gods’ power soon declined. The player, a wandering spirit, must complete tasks related to seasonal celebrations in order to restore power to the gods and life to the village.

Having the opportunity to complete our project in Japan, our team wanted to design the game so that it was reflective of the environment around us and the technology and expertise we had available to us. We initially discussed several ideas but ultimately decided on a game about the seasons, as seasons are an integral part of Japanese culture and folklore. We chose to develop our game for the HTC Vive for a few reasons. Takemura Laboratory specializes in research and projects involving immersive user interfaces, HCI applications, and a combination of these fields for use in education and communication platforms. The strengths of Takemura Laboratory lend well to VR development, as the resources and knowledge of laboratory members were readily available to us. Additionally, none of us had worked with VR technology before and saw this as an excellent opportunity to learn about development for a new platform and hardware arrangement. VR’s immersiveness was a great asset in helping us mold the player’s experience in playing the game.

*Shiki* was originally intended to be a casual puzzle game that used the simultaneous existence of all four seasons as a key gameplay mechanic: objects placed in one season would be manipulated in some way in another season. However, our game had a large shift in focus and design about four weeks into the project. We identified several characteristics and goals that we wanted our game to exhibit and defined our experience design as “relaxing, ethereal, and pensive.” First, we wanted our game to have a casual feel so that players would not be stressed out or feel a sense of pressure. Next, we wanted our players to complete a series of small tasks that would make use of our seasons and crafting game mechanics and follow our experience design. We wanted these tasks to be interesting to the player while being stress-free. Finally, we wanted to convey feelings of *hikarakuyō* (飛花落葉; “Blossoms fall and leaves scatter”; the
The evanescence of worldly things), which resemble emotions conveyed in Japanese literature, so that our project could be as reflective of our environment as possible.

Our project implementation was structured around three main prototype releases: Alpha, Beta 1.0, and Beta 2.0. The Alpha prototype contained a simple game world filled mostly with placeholder assets and the basic functionality needed for player movement and the seasons mechanic. The Beta 1.0 prototype contained the same game world with fewer placeholder assets and included the functionality needed for the crafting mechanic. This prototype also implemented the tutorial quest, which is designed to teach players how to play the game. The Beta 2.0 prototype was the final prototype in our design plan, and consisted of the finished game world with the addition of the Chrysanthemum Day quest. The Alpha and Beta 1.0 prototypes were playtested by members of Takemura Laboratory. Beta 2.0 was the final product of the feedback we received and our design plans; however, it was not playtested due to time constraints.

By the nature of Unity’s own design, the code for our project followed a component-based design pattern. Rather than having complex hierarchies of game objects that inherit behaviors and functionality, a component-based design pattern allows game objects to acquire behaviors and characteristics by storing these values in a single component, and attaching that component to the game objects in the Unity project. Components in Unity can be C# scripts, or characteristics of Unity’s GameObjects, such as a Material, RigidBody, or Collider. Much of the functionality of our EventManager, QuestManager, Inventory System, and seasons and crafting mechanics relied on components. Objects that should change based on their surrounding season had a seasonal effect C# script as a component to communicate that these objects should change appearance appropriately. Objects that had specific behaviors or attributes in regards to crafting and storing in the inventory had C# scripts as components that defined these behaviors. Attaching these behaviors to GameObjects as scripts allowed us to individually define the characteristics for each object in our game without having to worry how it relates to other GameObjects.

Over the course of our project, our team faced a number of challenges, some relating to the software and hardware we were using and some relating to our own knowledge and communication, that influenced the design and development of our project. None of us had ever worked with Unity or the HTC Vive before for a large project, but the learning curve for both was not significant. Issues arose with Unity as our project grew larger: Unity frequently crashed and ran too slowly on our own machines, and we were slowly forced to complete most development on Takemura Laboratory’s machines, which hindered how much we could accomplish at one time. In order to minimize the performance impact of our game, and in attempts to reduce the number of times Unity crashed, we optimized portions of our code and adjusted rendering properties of our game world. While this somewhat fixed our issues with Unity, the changes to rendering properties had a slight impact on how players saw and experienced the game world. Aside from issues relating to technology, we faced challenges in
the effectiveness of our communication. Towards the beginning of our project we frequently had to revisit design decisions after their first implementations as it became apparent we had unknowingly agreed to different ideas. To fix our communication issues, we organized more discussions and used several means of communicating our ideas, from words, to images, to demonstrations.

This paper discusses the development process of Shiki, the resources and techniques we used to build it, and the outcomes of repeated testing and evaluation. Chapter 2 discusses a brief history of the Edo Period, Japanese mythology, and the cultural importance of seasons in Japan. It also discusses a brief history of virtual reality hardware, its various forms and challenges with development, and the HTC Vive. Chapter 3 discusses the design of our artistic assets, background story, and project structure. Chapter 4 discusses the implementation of each technical and artistic component of our game, including the challenges faced and adaptations made to the original design. Chapter 5 discusses the procedures we followed and the results and feedback we received from playtesting sessions of our two prototypes. Finally, Chapter 6 discusses the post-mortem analysis of our project, including what went well, what did not go well, and what could have gone differently.
2. Background

One of our priorities throughout the development of Shiki was that we would, as best as we could, ensure the accuracy of the Japanese culture and history included in our game. To this end, we researched the setting of our game (an Edo Period village in Japan), as well as Japanese mythology. Since much of the team had little experience in virtual reality game development, we also researched the history and unique challenges of that medium.

2.1 Edo Period

The beginning of Edo Period is generally regarded as the moment that Japan became one nation, rather than a collection of warring factions, and is characterized by relative political stability. Ieyasu Tokugawa formed the beginnings of the Tokugawa Shogunate in 1603, which marked the transition from a society ruled by emperors to a society controlled by military leaders, or shoguns (Oono, 2004). It is during this period that Japanese art and culture flourished and infrastructure expanded to create centralized living areas (“The Edo Period in Japanese History,” 2016). This new organization of city layouts and social constructs played an important role in both the lives of Edo Period citizens and in Japan’s governmental and societal structure today.

2.1.1 Social Structure and Layout

As the new political leader of Japan, Tokugawa instituted a strict social hierarchy to maintain political stability and control over the people of Japan (Oono, 2004). This hierarchy consisted of four social classes and was reinforced by the structure of each region’s capital city and surrounding settlements. Those citizens of the highest social classes lived in the center of the city (“Castle Towns,” 2017; Oono, 2004). This city center was known as the jōkamachi (城下町; castle town), and often marked the region’s center of control. Because of the jōkamachi’s importance to maintaining the stability of the Tokugawa Shogunate, these city centers were often walled cities or surrounded by moats (“Castle Towns,” 2017).

Outside the jōkamachi were the chōninmachi (町人町; the town of the townspeople), or “commoner towns.” Those who lived in the chōninmachi were middle class citizens, consisting of farmers, skilled craftsmen, and the ashigaru (足軽; common foot soldier), who were less-educated foot soldiers that acted as an extra line of defense for the region (“Castle Towns,” 2017; Oono, 2004). On the outer edges of the chōninmachi were the lowest class citizens, often consisting of merchants and social outcasts (Beasley, 1972; Oono, 2004). While in some Western societies merchants were considered middle class or higher due to their wealth, merchants in Edo
Period Japan were part of the lower classes of society as they did not contribute unique and useful skills coveted by society (Beasley, 1972).

2.1.2 Village Life

During the Edo Period, villages themselves were considered the basic unit of society, rather than individual families, which were considered the basic unit of economic production (Oono, 2004). Villages paid taxes to the shogunate as one unit; as such, all village members contributed equally to fulfilling this responsibility (Varner, 1977). While there was no strict class hierarchy within each village, the takamochi (高持; land-owning farmers), were loosely considered to be leaders of the village. The takamochi employed mizunomi (水飲み, short for 水飲み百姓; poor peasant or farmer), to help harvest rice to pay the taxes. In exchange for their labor, mizunomi were often granted small portions of farming land for personal use (Kurosu, Takahashi, & Dong, 2017).

With the exception of farming lands and residential areas, there existed a communally-owned piece of land called the iriaiyama (入会山; common land) The iriaiyama was designated as the place to obtain the resources used for creating village infrastructure and other supplies needed by the village (Varner, 1977). Each village member had the right to use this land but also carried the responsibility to do so within reason, as without it the village as a whole would not be able to sustain itself.

2.2 Japanese Mythology

Throughout the history of Japan, many aspects of folk, Shinto, and Buddhist mythologies have been major influences on Japanese culture. Similar to the mythologies of other cultures, Japanese mythology describes natural events through the actions of Kami, the deities of Japan. In Japan, anything that inspired awe and ephemeral beauty would be considered a Kami (Daisaburō, 2014). Life and death, for instance, are explained through the marital conflicts between the two deities of the creation of the Japanese islands: Izanagi-no-Okami and Izanami-no-Okami.

One special aspect that Japanese mythology holds over other mythologies is the idea that deities are not all-powerful, and rely on humans to believe in them in order to survive. The more faith a deity can gather, the more powerful they become; the fewer people that believe or worship a Kami, the weaker that Kami becomes. If a Kami loses all, or even most, of their believers, that Kami can die (Daisaburō, 2014).

Even though the Shinto religion has ties to Buddhism, the two religions still have many differences. In Buddhism, a Buddha is an unmarried human male who has reached enlightenment. While in this state, Buddhas are not reincarnated when they die, and instead cease to exist. Kami are not human, but many are considered the ancestors of humans and can live and die as regular humans. Kami are not exclusively male, and many marry. Finally, Buddhist
worshippers can create images and statues of Buddhas, but Shinto worshipers do not make images of Kami. One similarity between the two religions is that neither Buddhas nor Kami live in the temple or shrine in which they are worshipped (Daisaburō, 2014).

2.3 Cultural Significance of Seasons in Japan

From early times, seasons have been of great importance to Japanese culture. Japanese people believe strongly that Japan is unique in having four distinct seasons, and as a result, the Japanese lifestyle has always been heavily influenced by the change of the seasons. Diets change as new foods come into season throughout the year, and events change with the weather and seasons. The seasons have been integrated so thoroughly in Japanese culture that the Japanese have developed several different calendars over the centuries to more accurately divide the year, such as nijūyon sekki (二十四節氣; 24 divisions of the solar year), or the even more specific shichijūnigumi kō (七十二候; 72 climates of the solar year). In modern times, the globalization of Japan has begun to slowly reduce the importance of being synchronized with the changes of the year, but many of the ancient traditions still take place. In the Edo time period in particular, the seasons would dictate daily activities, events, customs, foods, and many new seasonal traditions came into circulation that are still practiced today.

In the Edo time period, the year would begin with shōgatsu (正月; Japanese New Year), which occurs from the 1st of January to the 3rd, during which Winter foods, such as the Japanese citrus fruit mikan, would be consumed. Shortly following shōgatsu would be the first of five sekku (節句; seasonal festivals), taking place on the 7th of January. These seasonal festivals occur once every two months - excluding November - on 1/7, 3/3, 5/5, 7/7 and 9/9. Each festival celebrates the recent harvest and holds unique activities and events, dependent on the time of year.

The festival for which our quests were created and designed was the kiku no sekku (菊の節句; Chrysanthemum Festival), occurring on the 9th of September. This festival is also known as Chouyou (重陽; lit. “Heavy Sun”), because under the Yin-Yang school of thought, this day in particular has too much “Yang,” which can result in bad luck and misfortune if not dealt with. The notion of drinking Chrysanthemum Sake - sake with chrysanthemum petals floating in it - became one of many ways to balance the yin and yang of this day, as well as promote longevity and good health (菊の節句とは？その日に食べるものは？どんなことをするの？2015). The process of creating Chrysanthemum Sake that was used in the Edo Time Period is said to be similar to the process of creating Ume Shu (梅酒; plum sake), where one takes Japanese sugar rocks and chrysanthemum petals, and pickles them in shochu for a short while (酒の辞典一き行.).
2.4 VR Overview

Virtual reality (VR) has a long and storied past, dating back to the 1950s. In 1957, the filmmaker Morton Heilig invented the Sensorama, a booth which incorporated a 3D display, stereo sound, a scent dispersal machine, seat vibrations, and a fan (Inventor in the field of virtual reality, n.d.). This device was developed alongside a projector and a 3D camera which enabled the filming of scenes to be later presented in the Sensorama. Morton Heilig also filed for a patent for the Telesphere Mask in 1957, the world’s first head-mounted display (HMD). In 1965, Ivan Sutherland gave a speech titled “The Ultimate Display”, in which he describes many different interfaces that might be used to enable realistic virtual experiences. This speech prophesied many of the developments that have lead to modern virtual reality and augmented reality technology - eye tracking, head and body tracking, gestural interfaces, and haptic feedback (Sterling, 2009). Note that at this point in time, even the modern computer mouse was still in development, along with the modern computer interface. (The first demonstration of a modern computer system - now known as The Mother of All Demos - took place in 1968. (Edwards, 2008)) Then, in 1968, Sutherland built a prototype head-tracked 3D stereoscopic HMD, nicknamed the Sword of Damocles (Ivan Sutherland’s experimental 3-D display, n.d.). The VR company VPL, founded in 1985, developed the DataGlove, a glove containing sensors and actuators allowing for a system to track the movement of a user’s hands, as well as provide force feedback. VPL would go on to develop commercial VR systems before collapsing in 1993. In the 1990s, VR began attracting artists and creators, who created VR art pieces like Char Davies’ Osmose. This, in turn, led to a wider public recognition of VR. Despite its commercial promise, VR was eclipsed by the spread of the Internet in the mid 1990s. Ultimately, the technology of the 1990s was unable to fulfill the promise of immersive, seamless VR, and so the industry retooled and began to work mostly on military projects. The U.S. wars in Afghanistan and Iraq gave the military a greater need for better training, as well as better treatments for illnesses like PTSD.

The modern era of VR began in 2012, when Palmer Luckey created a Kickstarter crowdfunding campaign for a $300 VR headset called the Oculus Rift (Oculus Rift: Step Into the Game, 2012). Thanks to the rise of the smartphone industry, small and high-resolution displays were commercially available. And thanks to the pace of technological growth, computers were now powerful enough to provide high-quality graphics with a system latency below 20ms (a latency above this will give users motion sickness) (Carmack, 2013). One of Luckey’s main innovations was the use of software to correct optical errors that had traditionally been corrected in hardware, allowing the Oculus Rift to use much cheaper lenses than existing VR headsets (Clark, 2014).

While modern VR headsets have solved many of the hardware problems that have prevented truly immersive VR experiences from existing until now, VR still has many issues. Though latency has been reduced, users can still get motion sickness from games that do not
follow the best practices with regard to movement. Additionally, the desire for a more immersive VR experience has encouraged the development of new kinds of user interfaces, as well as research to determine which kinds of interfaces work well. The hardware requirements for VR are still high - it requires a powerful graphics card to achieve the high frame rate and low latency required to reduce motion sickness.

2.4.1 Motion Sickness and VR

The two main causes of virtual reality motion sickness both have to do with discrepancies between signals from the vestibular system - the inner ear - and signals from the visual system - the eye. The first type of motion sickness is often referred to asvection and occurs when the eye sees movement, but the inner ear does not sense any movement. An example ofvection that can occur in everyday life is when a person stands on a stationary train, looks out of the window, and sees another very long train move by. This can trick the eye into believing that the person is moving, but the inner ear does not think that there is movement (Illusions of self-motion<br>2017). The second type of motion sickness can often take the form of seasickness in everyday life; the eye does not see movement, but the inner ear feels movement. Regardless of the cause, the brain logically concludes that one has been poisoned, and thus needs to remove the poison by inducing vomiting (Treisman, 1977).

Beyond these two general causes of motion sickness, there are various other factors that can cause discomfort. One such factor is any appearance of motion that the player is not initiating themselves by traditional locomotion (Simulator sickness.). The inner ear is particularly sensitive to accelerations, so if one must accelerate, it should not be sudden high levels of acceleration (Simulator sickness.); if a player walks in real life, they should not appear to accelerate faster than in real life (Motion.). On a similar vein, taking any amount of control away from the player and visualizing anything that the player has not done themselves directly can cause strong discomfort (Simulator sickness.); any head-bobbing or additional movement not created by the player must be avoided (Motion.). In regard to more static images and overall background or visual effects, stereoscopic images should be avoided as a whole, as some people find them uncomfortable (Simulator sickness.). Any flickering images and lag or latency should be avoided at all costs as well (Simulator sickness.). Additionally, the player should not be subject to excessive amounts of data on the screen at any one time (Simulator sickness.). Too much on the screen at once can cause great discomfort.

While many issues with movement in VR can be solved by only allowing the user to move in accordance with their headset (i.e. turning their head will cause the view to turn; accidental, natural movements also cause the view to change slightly), the issue of how to implement large-scale movement without causing motion sickness still stands. One popular solution to this, using the HTC Vive, is to implement a teleportation mechanic; the user can perform a series of actions in order to teleport their current position in the game to a different one
(Francis, 2015). If a player has hit a wall in reality, or they cannot move around to begin with, teleportation still allows them to explore a large world. The downside to teleportation, however, is that it can break immersion, and still cause motion sickness if not implemented correctly. Teleportation can cause motion sickness if the player is witness to their transition from their original position to the teleportation destination. This is due to the appearance of movement on the screen that the player has not caused directly. However, simply moving the player directly to the location in question can cause disorientation; therefore, some developers choose to fade the screen to black. But this effect can break immersion, and feel unnatural. To combat this, studies have shown that fading to black from the top and bottom of the screen at the same speed that the average person blinks can give the effect of fading to black, but will be automatically filtered out by the brain (Oculus, 2014). In our implementation, we used SteamVR’s teleportation plugin, which fades the screen to black quickly each time the player teleports.

2.4.2 VR Interfaces

Interacting with an immersive virtual reality (VR) environment is substantially different than interacting with a traditional computer game, and consequently many of the input methods that work well for traditional video games do not work as well in VR. Due to the motion sickness issues described above, traditional joystick-based movement controls are not viable when using immersive VR displays. The consumer VR systems available today use wand-style, tracked controllers that users can position in the virtual world, with buttons to initiate actions. Steam is currently working on a new controller - code-named “Knuckles” - that attaches to the user’s hand and tracks how curled the user’s fingers are (Yang, 2017).

Current research seems to be focusing on controller-free interfaces, mainly using devices such as the Leap Motion to track hand movements, and interfaces which provide haptic feedback. Additionally, there is a large amount of research being done on eye tracking and its applications in VR: see Tanriverdi and Jacob (2000), Jacob and Stellmach (2016), Pai et al. (2016).

VR also requires different kinds of visual user interfaces (UI) when compared to traditional games. Traditional games tend to make heavy use of HUD interfaces, in which information is shown overlaid on top of the user's view. This kind of interface is typically positioned near the edge of the screen, to prevent it from obscuring the view of the rest of the game. This approach is not well suited to VR; in order to be in front of the rest of the game view, the interface would have to appear to be very close to the user’s eyes (User interfaces for vr, n.d.). Additionally, Oculus recommends that the UI fit inside the middle third of the user’s field of vision, to avoid forcing the user to move their eyes in order to see it (“Introduction to best practices,” n.d.).
2.4.3 HTC Vive

Valve Software, the creators of the Steam PC video game market, as well as many critically renowned video games, met with HTC, who had been developing a VR headset since 2013. They decided to work together on a VR headset combining Valve’s positional tracking technology with HTC’s experience in hardware development (Souppouris, 2016). However, this was not Valve’s first attempt at entering the VR market. Valve had been working on positional tracking using AprilTags (2D barcodes, similar to QR codes) since 2012, and Oculus, still a young company, was interested in their technology. However, after Facebook acquired Oculus in 2014, Valve and Oculus parted ways, allowing Valve to collaborate with HTC.

The HTC Vive, alongside the Oculus Rift, was one of the two “first generation” high-end, modern VR headsets to be released to consumers in 2016. The Vive system consists of a headset, two “Lighthouse” base stations, and two controllers. The headset and controllers both have sensors inset that detect the base stations, and are thus able to determine where they are located relative to the base stations. Before the headset is used for the first time, the user must perform a calibration routine to determine the necessary offsets so that the system knows where the ground is relative to the base stations, as well as where any play boundaries may be (Setting up Vive for the first time, 2017). The Vive also includes a front-facing camera, which can be used to show the player an image of the real world in front of them, or potentially enable augmented reality experiences.

The major difference between the HTC Vive and other VR systems on the market is the use of Valve’s Lighthouse 3D tracking system. Valve’s system enables the Vive system to track the positions of the VR headset and the two controllers in a space with a maximum diagonal distance of up to 5 meters, using only two Lighthouse units. Each Lighthouse unit contains a row of infrared LEDs, and two perpendicular spinning units which each emit a line of laser light. The infrared LED row emits a pulse of light - a timing pulse - 120 times per second, and one of the spinning laser units sweeps a laser line over a 120 degree arc. This process repeats, with another timing pulse and another sweep, using the other spinning unit. The sensors inside the tracked object detect the timing pulse, and then time how long it takes for the laser line to hit the sensors (Buckley, 2015). The position of the tracked object can then be calculated using trigonometry. To augment the data provided by the laser timing system, tracked objects also include an inertial measurement unit to detect orientation and acceleration. Valve claims that this system enables “sub-millimeter accuracy” (SteamVR tracking, 2017), but one study has found that its accuracy is roughly 2mm (Kreylos, 2016).

The HTC Vive controllers are “wands” that each contain several infrared sensors, a touchpad with a button, a trigger button, and two “squeeze” buttons. The squeeze buttons are wired together, with one on each side of the controller so that the controllers are ambidextrous.
The controllers communicate with the host computer wirelessly, and are able to determine their position in space by detecting signals emitted by the two Lighthouse beacons.

Like the Oculus Rift, the HTC Vive uses two 1080x1200 displays, with a 90Hz refresh rate. Both headsets also offer a 110 degree field of view (VIVE virtual reality system, 2017). The HTC Vive also has an adjustable IPD (interpupillary distance), and lens-to-eye distance for each eye (Vive PRE user guide, 2016).
3. Design

Shiki possesses several artistic and technical aspects that work in conjunction to create a final playable game. These design areas include artistic assets, sound assets, and story and plot design, as well as several technical components of coding architecture and project structure. This chapter outlines in detail the design plans for each area of Shiki.

3.1 Artistic Design

The art assets for Shiki draw upon traditional Edo Period screen paintings and village architecture in order to convey the settings of an Edo Period village. The following section discusses the history and creation of paintings in the Edo Period and how these paintings influence our art design for the introductory prologue scene in Shiki. Additionally, this section discusses the influences and design plans of the textures and models used for 3D elements in the game.

3.1.1 2D Art Design

In deciding what 2D art style to use for the prologue of our game, our team primarily considered our artistic abilities to determine how to proceed with creating 2D art. Since Shiki takes place in the Edo Period, we chose to base the art style for all 2D art off of Edo Period paintings. The Edo Period lasted for over 200 years, and as such, featured varying art styles that developed from different sources.

At the beginning of the Edo Period, the Tokugawa Shogunate gave patronage to the Kanō family of artists to create works for the homes of the shogun, samurai, and other elite members of society (The kano school of painting 2003). The Kanō family of artists was an emerging group of artists from the Momoyama Period that originally drew inspiration from traditional Chinese greyscale screen paintings but began to develop a more vibrant style using colored inks. These paintings were often created using ink on gold-leaf paper screens, typically for 6-panel folding screen doors.
The Kanō School of Painting expanded on the traditional Chinese greyscale style by emphasizing purposeful outlines and bright colors that were sparingly grouped to dictate both shapes of objects in the scene and depth in the environment (The kano school of painting 2003).
Figure 2: Kiyomizu-dera from Famous Places in the Capital and Legendary Figures, Kano Shōei, ca. late-16th century. Museum of Fine Art Boston.

Figure 3: A Chinese Port, artist unknown, ca. early-17th century. Museum of Fine Art Boston.
As the Edo Period progressed and generations of Kanō artists developed their own unique styles, the Kanō School of Painting branched off into different sub-styles that would eventually become the prominent styles of the Meiji Period (The kano school of painting 2003).

It is due to the simplicity of the early Kanō School of Painting’s style that our team chose to create the 2D art for Shiki’s prologue in their style of screen paintings. Since Shiki is a first-person perspective in a VR environment, our art implementation had to be conscious of the risks of VR motion sickness. Animations or movement within a VR environment that are not the direct result of the player’s body movements are likely to increase the chances of developing motion sickness. As such, the animation style for the 2D art follows a storyboard-like style.

(a) Village and farming scenes
3.1.2 3D Art Design

As one of the most important aspects of any game, we wanted to ensure that Shiki’s 3D art complemented the game’s story, background, and experience design. Unfortunately, no one on our team has major experience in 3D-modeling or art, so we had to rely mostly on importing assets from the Unity Asset Store. Also important to the quality of Shiki was the world design. To truly capture our game-feel, we needed to blend natural seasonal beauty into an unnatural environment while still allowing players to navigate our world without difficulty.

As explained earlier in this paper, Shiki takes place during the Edo period, so we needed to find asset packs that reflected the design of buildings during that era. Since Shiki is a game
that focuses on the four seasons, the assets we used needed to express the beauty of nature. This included not only trees and flowers, but also particle effects that could be used in each season.

One of the more challenging aspects of our design was the creation of the world. For a game centered around a relaxing user experience, we had to make sure that our world conveyed Shiki’s story cleanly; no part of the world should betray the player’s expectations. The core of our game’s story is that the four local Kami of the seasons, in response to the actions of the villagers, removed their corresponding season from the village and formed their own versions of the landscape outside the village. To complement the seasons mechanic, we made each quadrant have identical terrain but have each represented during a different season.

3.2 Technical Design

The structure of our code for this game was developed with component-based software architecture in mind. The following section begins with a description of a related architecture, object-oriented programming, where component-based software engineering differs from it, and transitions into describing our team’s transition from object-oriented to component-based, as well as our reasoning behind it.

3.2.1 Component Based Software Engineering

When video games were first developed, they were small and simple enough that a procedural programming style was sufficient. However, as video games became more and more complex, and as larger development teams became the norm, this simplistic programming style was no longer feasible.

Perhaps the most natural architecture for the majority of modern computer science students is the object-oriented paradigm. This is commonly taught to students in some of their earliest classes, often using Java. In the object-oriented architecture, code is shared through inheritance - if some code is duplicated between two classes (see figure 5), a new class should be created to hold all of the code common between the two classes, and the two classes should inherit from the new class (see figure 6). This works reasonably well for many different applications, and as such, is commonly used in practice.
However, the object-oriented paradigm is not perfect. An issue that often comes up when designing video games is that of unneeded inherited functionality. Imagine that a programmer has a Dog class, containing complicated code for navigating through some environment. Imagine then that the programmer is asked to create a Cat class, which also needs to be able to navigate through the environment. One possible approach to take would be to make Cat a subclass of Dog. Now, Cat has inherited the complex navigation code from Dog. However, Dog might have methods that are specialized for dogs - for example, maybe the Dog class contains a wagTail() method, or a bark() method. Cat also inherits those methods - which is likely not desired. To fix this, the programmer might override these methods in Cat and make them throw exceptions, or
otherwise indicate that the methods should not be called on Cats. However, this is highly imperfect, and has the potential to cause many issues further on in the development process.

```java
class Dog {
    void bark() {...}
    void wagTail() {...}
    void doNavigationThings() {...}
}

class Cat : Dog {
    // We need to do this for every method in Dog
    // that we don't want to be called on Cat!
    void bark() {
        throw new Exception("Can't call this on Cat objects!");
    }
    void wagTail() {
        throw new Exception("Can't call this on Cat objects!");
    }
    // These methods are specific to Cat
    void meow() {...}
    void walkOnKeyboard() {...}
}
```

Figure 7: Example of an implementation where Cat inherits from Dog

In this simple example, a better solution is evident: make a superclass NavigableAnimal that contains the shared navigation code, and make Cat and Dog both inherit from NavigableAnimal (see figure 8). However, in more complicated systems (see figure 9), such a solution is not always available. Dog might already have a superclass that Cat should not be a subclass of - maybe Dog is a subclass of Canid, which contains code common to Dogs and Wolves. Many programming languages only allow a class to inherit from a single class, so there is not a good solution if Wolf should not have the navigation code, but Cat and Dog should.
The core issue that leads to these inheritance problems is that classes often become specialized to enable all necessary functionality. This makes inheriting from that class without sharing unneeded functionality difficult. A more ideal solution would be to have modules that each only enable a single bit of functionality, and that can be added to classes/objects in any combination.
This is exactly what the component-based architecture aims to provide: small, self-contained, modules (components) that can be attached to objects in any combination. Now, our cat-and-dog example might look something like figure 10:

![Diagram](image)

**Figure 10**: Sharing navigation code between Cat and Dog using a component. Both Cat and Dog have a NavigationComponent attached to them that they can use to call `doNavigationThings` as needed.

Dog and Cat can now share the navigation code cleanly, without having to give other classes unneeded functionality.

When we started developing *Shiki*, we almost unconsciously gravitated towards a classic object-oriented inheritance architecture. This is likely due to the focus that is given to such architectures in computer science classes, and the fact that the team members had more experience with such architectures. Our initial design looked something like figure 11.
However, once we started working with Unity, we found that is heavily designed around a component-based architecture (Porter, 2013). Rather than try to shoehorn an object-oriented inheritance-based architecture into a game engine that is not designed to work with it, we decided to rethink our design.

Though we were unfamiliar with component-based architecture, Unity’s design helped steer us in the right direction when writing our code. The combination of our custom event system and component-based design helped us write highly modular components that can easily be combined to give objects the functionality they need. A list of the components developed for Shiki is available at the end of Section 4.3.1.

3.3 Sound Design

The experience that we hope to invoke in Shiki’s players is best described as “relaxing, ethereal, and pensive.” While imagery and gameplay are major contributors to player experience, sound also plays an often overlooked but important role in rounding out a player’s experience.
*Shiki* takes place in a disrupted visual environment, in the sense that the player can distinctly see that life and energy exist in the season quadrants but stop short at the entrance to the village. Environmental sounds are necessary to fill in the gaps of the intended experience that visuals and in-game events cannot convey alone. In order to reinforce the sense that the season quadrants represent their respective seasons, the sounds used in the seasons are reflective of the sounds of nature heard during each of these seasons.

Spring is associated with rebirth and growth. The sounds one might hear during Spring are increased animal activity (such as birds chirping), and water melting. Summer is associated with full life and activity. The unmistakable sound of cicadas and other insects can be heard anywhere one may go. Autumn is associated with winding down and preparations for darker times. Leaves begin to fall from trees and the environment begins to dry out. Weather patterns change and wind becomes more prominent. One might hear sounds such as rustling leaves and grass, gentle wind, and a gently flowing river. Winter is associated with the end of the year and an absence of life. The environment is cold and exposed. Wind sounds more hollow as there are very few plants to interrupt its path.

The village represents the area of the world that is missing its energy and life. Aside from the player, there is nothing in the village capable of acting or displaying energy. A total absence of sound would likely ruin the experience design for the player as it would feel like something was unintentionally missing. To still convey a sense of emptiness with an ambient sound, the village might contain a reverberant white noise.

### 3.4 Story Design

In a small rural village during the Edo period, the Kami were held as the highest figures of worship. Throughout the year, the Kami would grant the influence of their seasons upon the villagers in return for the villagers’ faith. Just outside the northern border of the village sat a shrine in which the Kami would be enshrined on rotation throughout the year. During periods of *dōyō* (土用: the eighteen days before the beginning of each season where almost no seasonal influence exists), the village’s people would hold an enshrinement ceremony to remove the current Kami from the shrine and enshrine the next. This allowed the previous Kami to rest for the duration of the other three seasons and regain their energy. For years this cycle was successful, the Kami enjoyed the power that they gained from gathering the faith of the villagers and the villagers enjoyed the seasons that the Kami provided. During each season, the villagers would also hold festivals to celebrate the grandeur of the current Kami.

These traditions went smoothly until one year, the Kami of the Autumn season rewarded the villagers’ deep loyalty with an especially bountiful harvest season. Wheat grew in immense quantities and there was enough food grown that the bounty from this one season could last the village an entire year. The Kami of Autumn expected a larger amount of faith back from the villagers, but was insulted as the villagers became greedy and took more than what they needed.
Large banquets were held and the villagers simply lived a life of absolute wealth, not considering that the harvest was a gift from the Kami. Near the end of Autumn, just before the beginning of doyō, the villagers pulled the last remaining crop, and after using it for one last meal, had too little resources to hold the enshrinement ceremony for the Kami of Winter.

Angered beyond belief, the Kami punished the villagers. At the end of the following doyō, the four Kami held a strike. As all four deities took residence in the shrine, the land at the entrance to the village that houses the shrine replicated, surrounding the village with four identical copies of the land around the shrine, one Kami to a copy. The Kami then rescinded their seasonal influences from the village, letting it remain in doyō until they deemed that the villagers learned their lesson. This never happened, however; the villagers, unknowing of the Kami’s plot, instead lost their faith in the Kami, believing that they had fled. As the seasonal liveliness in the village subdued under the power of doyō, the villagers began to petrify into lifeless stone. Eventually an eternal doyō flooded across the village, and once the final villager became petrified, there was nothing the villagers could do.

Believing that they had successfully punished the villagers, the Kami attempted to restore the village back to its previous state. Without a lively soul in the village to gather faith from, however, the Kami did not have enough energy to pull off this feat of restoration. The Kami soon realized that they were in danger, as Kami can only live if there is a sentient creature to believe in them. With the remnants of their power, the Kami summoned into existence a wandering spirit. Before entering a state of hibernation to conserve their remaining faith, the Kami explained that the spirit must bring liveliness back to the village, or they and the villagers will perish forever.

Shiki’s backstory is heavily inspired from a mixture of Edo Period Japanese culture and general Japanese mythology. During the first few weeks of our project, the story of Shiki changed often. We wanted to make sure that our all of our game mechanics and game tasks could be explained through the story rather than have arbitrary mechanics with no connection to the story. It took about a week for our group to decide that we wanted to make a game about the seasons. While it is a cultural importance that Japan has four distinct seasons, it was difficult to create a story that made sense regarding Japanese culture and mythology, and was easy to understand from the view of a player who knows nothing about these concepts.

Originally, we were leaning towards creating a game that brings out the themes of waiting and meditation and beauty from transience. While we were able to partially create a story from these themes, we could not connect this story to the early versions of our game mechanics. One version of the game’s story was told over the course of twelve in-game years where certain events would only happen during certain seasons, certain years, or a combination of the two. The player would have to visit and complete each event for the game to be finished, but we could not think of a thematic reason as to why the player must complete these events in the first place.

We later developed a world similar to the final version of Shiki’s world. Like the current Shiki, each season would be represented as identical quadrant of terrain surrounding the village.
This time, however, the player character would be a villager, specifically the villager who pulled the last bit of crop from the land and angered the Kami. This design led to problems in determining the role of the player’s character in the village, as well as the motivation behind some of the quests. Additionally, at that point in time we had not yet determined why the villagers petrify in the first place. After some research we discovered an arcane season called doyō, which encompasses the four boundaries between seasons throughout the year where the weather does not quite represent the current season or the next. In modern Japanese contexts, doyō is akin to the Western idea of the dog days of Summer, as opposed to the transition between each season. The liveliness of the environment fades in times of doyō, and there is less vitality. Combining the idea of a lifeless doyō with the idea that a Kami cannot exist if forgotten led to Shiki’s final story.

3.5 Game Mechanic Design

This section discusses the two main game mechanics we created for Shiki: the Season mechanic and the Crafting mechanic.

3.5.1 Season Mechanic

Shiki is designed around the use of a seasons mechanic. With this mechanic, players are able to place certain, “seasonal” objects down in any of the quadrants in the game world and see how they change throughout the year by checking the relative location in another quadrant. Since the beginning of our project, this mechanic has seen several different implementations leading up to where it is now.

The very first idea our team came up with was to give the player the explicit ability to change the current season to solve puzzles. Each season would apply different effects to the environment. Winter would freeze water and Spring would allow plants to grow, for instance, while Summer would thaw frozen objects and Autumn would wither away plant life. Based on this idea, there were several different implementations that the team conceived, such as only being able to change the season in a small area around the player or a designated object. Our team further experimented with ideas involving the player being able to change the season for both the global environment and within a small area around an object.

Very quickly our team determined that by giving the player direct control of some aspect of the environment, we violated one of our design goals by breaking with Japanese literary themes. Our second main idea for the seasons mechanic involved the idea that “transience is beautiful”. With this idea, Shiki would take place in a small world over the course of a single zodiac cycle, that is, 12 years split into 4 seasons each. Events would occur during certain years, during certain seasons, or a combination of the two. Players would need to hunt down these events and complete all of them to finish Shiki. Unlike the previous set of ideas, this 12-year idea would not give the player the ability to control time or the seasons; players would only have to
wait for the seasons to change. While this idea would work in terms of story, the team would not have enough time to fully create this game. At a minimum, the game would take an hour to complete, resulting in each year taking place over 5 minutes.

The team devised a world where each season is represented at the same time in four different segments of the world during a brainstorming session after scrapping the previous idea. With a bit of tweaking, this idea became the basis for what the seasons mechanic of *Shiki* is now. *Shiki* takes place in a world where the Kami of the seasons pulled their influences away from a central village and into four completely identical landscapes that now surround the village. Each landscape is linked by time, so a player can place an object in one season, travel to another quadrant, and observe that object as it would appear in that season.

### 3.5.2 Crafting Mechanic

After the shift in our design from a puzzle game to an experience game, we developed a crafting mechanic as a way to increase players’ immersion in the game. Each of the quests in our game require the player to create objects to be used in festivals for the village. Players must perform chopping, digging, and cutting motions on craftable objects while wielding a tool to produce a desired result. The overarching design of the crafting mechanic is simple in that it employs one script, `ToolFunction.cs`, which defines the tool and determines which action should be taken on a craftable object. This script works in conjunction with our project’s `EventManager` to send an event to be recognized by objects that possess `HittableBehavior`. The crafting mechanic underwent several iterations of design that ultimately increased the previous iterations’ complexity.

The very first iteration of the crafting mechanic used `ToolFunction.cs` and `ObjectScript.cs` to detect if there was a collision between a tool and a craftable object. If a collision was detected, the object’s `changeObject()` function would be called and the object would change its material to a new material defined in the editor. This worked well as a basic implementation; however it did not allow for immersion or contribute to the experience design of the game. It is unrealistic to expect that the player can craft an object instantly without any thought or careful movements required. Thus the next iteration employed collision velocity in an attempt to make the crafting mechanic more immersive.

In the next iteration of the crafting mechanic, the velocity magnitude of the collision between the tool and the craftable object was used to determine how much the object should be affected. Per Unity’s `Collision` class, each collision contains a `Vector3 velocity` that is the velocity of the colliding object at the point of the collision. The magnitude of this velocity tells us the “force” of the collision, or rather, the speed of the colliding object right before the collision occurred. `ObjectScript.cs` was updated so that each craftable object had a customizable hitpoints field, `hp`. Now, in order for the player to completely craft an object, they needed to hit the object more than once, until the object’s `hp == 0`. The addition of object `hp` slightly
increased the immersion and was more in line with our experience design; however, through playtesting our players discovered that the way with which they hit craftable objects did not matter. A tree could be hit with a stabbing motion rather than a chopping motion, for instance, and it would still be cut down after enough collisions occurred. We again needed another solution for the crafting mechanic that would allow players to have a more realistic crafting experience.

The final iteration of the crafting mechanic attempted to use a combination of number of collisions, time between collisions, and the position of each collision to classify crafting behavior before using velocity to change a craftable object. The number of collisions, time, and position values were all required to be within certain thresholds in order to determine if classification should occur and what behavior was happening. Collisions were tracked according to how many collisions occurred in a certain time period and compared against a required amount needed to detect if an action was occurring. This process was used to narrow down when behavior classification should occur, because it would be expensive optimization-wise if we were to attempt to classify every single collision when some of these collisions might be one-time occurrences. If the script determined a behavior might be happening, it then called `classify()`, which used position data to determine what motion might be happening.

Position data from collisions were tracked along the x-, y-, and z-axes and the difference between collision positions were checked between min and max range values for each axis. If the change in position along an axis was within that axis’s min and max values, this meant that the position of the collision along this axis did not change much, and it could be assumed the player was intentionally trying to hit the same relative location more than once, as expected if they were trying to perform a specific crafting motion. Through informal experimenting, we discovered that the x-axis in game is always inline with the direction of the headset where the player is looking. We used this knowledge to attempt to classify different movements. We hypothesized the following movements:

- Pounding motions, such as hitting something with a hammer, would involve a small positional change along the z-axis with virtually zero change along the x- and y- axes
- Drawing motions, such as using a paintbrush on a vertical surface, would involve positional changes along the y- and z- axes but virtually zero change along the x-axis
- Chopping motions, such as swinging an axe to cut down a tree, would involve a small positional change along the y-axis with virtually zero change along the x- and z- axes

If a behavior was detected and met one of these classifications, it was then safe to assume that the player was indeed performing that crafting motion, and a change should be affected onto the craftable object if the crafting motion was appropriate for the task at hand. By classifying
behaviors and defining objects to have an expected crafting behavior, this allowed us to achieve an even greater level of immersion and follow our experience design more closely.

The final version of the crafting mechanic implemented in the game was based on our second iteration of the design. The third iteration of the design, while the most immersive, was not entirely accurate as it would classify all behaviors instead of just one. Due to time constraints with our time at Osaka University, we reverted to the second iteration.
4. Implementation

In implementing our game, we used a large variety of resources and strategies to guide us in creating what we had imagined. Our environment setup consisted of using the HTC Vive in Unity, with Git as our version control, and GitHub’s Issue Tracker to track bugs and test software. Additionally, we used two external libraries: Nett and neural-style. With these tools in our repertoire, we developed the season and crafting mechanics, as well as the EventManager, QuestManager, and Inventory System. Finally, we used a variety of techniques and external resources to acquire artistic assets for Shiki. The following sections discuss each of these in more detail.

4.1 Environment Setup

Before we started to implement Shiki, we had to set up the necessary development software on the team members’ laptops, and get access to the hardware needed to run our game. Takemura Lab graciously gave us access to their VR hardware, including two HTC Vive systems, and a dedicated desktop computer for our team. The following sections describe in detail the hardware and software used to develop Shiki.

4.1.1 Hardware

At Takemura Lab, we had access to two HTC Vive systems, and were provided with a VR-capable desktop computer to use for development. The first Vive system was located in Takemura Lab’s “Experiment Room”, and had a large amount of open floor space for a room-scale VR setup. This system had a dedicated desktop connected to it, but because it was open for use by other lab members, we could not always access it. The second Vive system was located in an office room, and did not have sufficient open floor space for a room-scale VR setup, so we set it up for standing-room VR.

Providing a comfortable VR experience currently requires a powerful GPU, since the latency must be kept below the 20ms threshold and each eye’s view must be rendered separately. Of the laptops the group members brought to Japan, not a single one was capable of running VR, so most of the development process took place without the ability to easily test changes in a VR environment. This led to some issues where scenes that looked good in non-VR testing were found to look worse, or even nauseating, when they were viewed in VR.

Generally, to test changes, team members had to create a new development branch in the git repository if one did not already exist, commit any changes made, push those changes, relocate to the VR-dedicated lab desktop, switch to the development branch, pull the changes, and then open Unity. While this is relatively painless in principle, this process was often hampered by issues with Git LFS tracking, very slow checkouts using Github Desktop on
Windows, and various issues with Unity. This was compounded by the fact that debugging issues in our game often took a significant amount of time, blocking other team members from using the VR desktop. This presented a significant bottleneck in the late stages of our development.

4.1.2 Development Environment

Developing a game for the Unity Engine necessitates the use of the Unity Editor development environment. This program provides the ability to create Unity scenes and add objects to them, as well as to attach scripts to objects and perform basic 3D modelling. The Unity Editor does not contain a dedicated code editor, so the Unity installer also installs Visual Studio on Windows, and a customized version of MonoDevelop on macOS.

4.1.2.1 Unity

One of the first big technical decisions that we had to make was the choice of game engine in which to develop our game. There are a vast array of freely available game engines online, from proprietary industry heavyweights (Unity, Unreal Engine, CryEngine) to fledgling royalty-free and open-source ones (Godot, OGRE, BansheeEngine). Given the time restrictions of our MQP, we decided to use Unity (version 2017.1), since it has an established history of VR support, and since a few members of our team were familiar with it.

Unity officially supports two programming languages: C# and JavaScript. We chose to write our code exclusively in C#, given that our team was more familiar with C# overall, and since C#’s static type system makes it easier to catch issues originating from accidentally misusing an API written by another team member. Unity uses .NET for its C# runtime on Windows, and Mono on macOS and Linux operating systems. This led to some issues later on, as one of our team members used a macOS laptop.

Unity is able to support C# code on non-Windows operating systems by using the Mono framework, an open-source implementation of the C# runtime and C# standard library (Mono.2017). This provides a replacement for Microsoft’s .NET framework that is available on Windows machines. However, Unity only supports specific versions of Mono and .NET - and the versions that it supports are quite old. This means that it is difficult to use C# libraries which are available online, since many of them require a more modern version of C#. In Unity 2017.1, it is now possible to use .NET version 4.6 (and the equivalent version of Mono) within Unity, an improvement over the previously supported version of 3.5 (released in 2007).

4.1.2.2 Git

Version control was a necessity for us - it allowed for all of us to work on different features at the same time on several different computers, and not have to worry about keeping

---

track of which files we changed. We chose Git as our version control system (VCS), since it was the VCS which our team was most familiar with. We hosted our repository on Github, and set its visibility to private for the duration of our MQP. This allowed us to host asset files on the repository which we could not have hosted on a public repository due to licensing issues.

We made use of the Git LFS extension, which is designed to make storing large files in Git repositories easier. We used the example .gitattributes file provided by Rich Reilly (2017) to set up which files would be tracked using Git LFS. This worked out well, but we ran into a few issues involving Github’s LFS bandwidth cap, which we exceeded when the team all ran the initial clone of the repository. We ended up purchasing a bandwidth cap increase to get us through the first month.

For the members of our team who were more comfortable with interacting with Git through a GUI, we used the beta version of Github Desktop (GitHub desktop2017). This tool allows users to create branches, stage files, and commit through a fairly simple interface. We used the open-source Meld merge tool (Meld2016) to assist us in resolving merge conflicts. We had a few issues pop up over the course of our MQP relating to some of our team’s computers not correctly tracking files in LFS, but we managed to solve the issues using online resources.

4.1.2.3 Software Testing and Bug Tracking

During the development of our game, we chose to utilize integration testing as our primary source of software testing. We were frequently testing the full game, and we also ran two playtests of the game with potential users. In both the playtests as well as us experiencing the game itself, we ran into bugs and other issues along the way. For bug and issue tracking, we had decided to use Github’s issue tracker. When a bug or some other task that needed to be added was discovered, the discoverer would create a new issue on Github with the appropriate labels. Issues would then be assigned, either by the discoverer or another member, sometimes commented on, dealt with, and then closed when deemed complete.

We used Github’s issue tracker to keep track of known bugs and who was responsible for fixing each bug. To help organize the issues, we used the github-standard-labels tool (Wuyts, 2017) to add a standard set of labels to Github’s issue tracker.

4.1.3 External Libraries

During the development process, we researched and utilized a few external libraries to guide our game more towards the image we had envisioned. For information that we wanted the game to access outside of the source code, we used Nett - the C# library supporting the TOML file format - to store necessary quest/game states. For the overall look of the game, we used Justin Johnson’s style-transfer neural network called “neural-style” (Johnson, 2015).
4.1.3.1 TOML and Nett

TOML, which stands for “Tom’s Obvious, Minimal Language,” is a minimal configuration file format that is designed to be easily readable by humans due to its “obvious semantics” (Presten-Werner, 2015). Despite having implementations in many different programming languages, TOML is still considered an unstable work in progress by its creator, Tom Preston-Werner, currently at version 0.4.0 (Presten-Werner, 2015). TOML works primarily through key-value pairs, meant to be implemented as dictionary hashes.

Our team came across TOML when we were searching for a method of loading quest information into the game. We had wanted to keep quest information separate from the code itself to make it easier to make changes if necessary and not bloat up the code. In addition, we had wanted a system that would give us the functionality we needed, but not require future developers to work with something so complex that they could not learn it quickly. These concerns, along with a hope to find a format that would be easy to both read and write, eliminated common formats such as JSON from consideration. In our search for the perfect format, we discovered TOML, as well as an implementation for it in C# called Nett.

When the game begins running, a series of files are read in from outside of the code. One file contains tables that hold quest information, while the other contains an individual player’s save data. The quest information file is organized in a series of TOML tables, where each entry in the table contains the information for a single task. Tasks are the fundamental element that make up quests; they have names, child tasks (which must be completed before the parent task can be completed), completion requirements (which are known as triggers), and post-completion events (which are known as onCompletes). These are then read in and handled by the QuestManager, which is discussed in more detail in the QuestManager section.

<table>
<thead>
<tr>
<th>JSON</th>
<th>TOML</th>
</tr>
</thead>
<tbody>
<tr>
<td>{&lt;br&gt;  &quot;tasks&quot;: [&lt;br&gt;   {&lt;br&gt;     &quot;Name&quot;: &quot;AddIngredients&quot;,&lt;br&gt;     &quot;SubTask&quot;: &quot;AddShochu AddSugar AddPetals&quot;,&lt;br&gt;     &quot;OnComplete&quot;: &quot;Item SakeBowl Become Item UnfermentedSake&quot;&lt;br&gt;   },&lt;br&gt;   {&lt;br&gt;     &quot;Name&quot;: &quot;AddShochu&quot;&lt;br&gt;   }]&lt;br&gt;}</td>
<td>[[task]]&lt;br&gt;  Name = &quot;AddIngredients&quot;&lt;br&gt;  SubTask = &quot;AddShochu AddSugar AddPetals&quot;&lt;br&gt;  OnComplete = &quot;Item SakeBowl Become Item UnfermentedSake&quot;</td>
</tr>
<tr>
<td></td>
<td>[[task]]&lt;br&gt;  Name = &quot;AddShochu&quot;&lt;br&gt;  Trigger = &quot;Player Drop Item Shochu On Item SakeBowl&quot;</td>
</tr>
</tbody>
</table>
"Trigger": "Player Drop Item Shochu On Item SakeBowl"
}
]
}

Figure 12: Comparison of JSON and TOML file formats for some example content

The other file that is read in early on is the player’s save data. When a player loads their
save data, their respective save file is read in, and parsed by the SaveManager. We used the
TOML format to easily state which tasks the user had completed. When the player saves the
game, this file is edited appropriately. The SaveManager itself is discussed in more detail in the
QuestManager section.

4.1.3.2 Style Blending with Neural Networks

Since the beginning of the project, the team was interested in implementing some kind of
neural network into Shiki. During the early stages of brainstorming, we thought of creating a
system that, after analyzing and learning from a large selection of Japanese poetry, would create
its own poetry in real-time. This idea did not work as intended, as a lot of the source material we
were working with would use grammar that Japanese poetry does not use today. Some of the
sources even used the Hiragana symbol わ (wi), which had been ruled obsolete in 1946 in favor
of い (i). When limiting sources to more modern-day poetry, the results did not show much
improvement. Grammar was still off and nearly unreadable, and sometimes translations would
show that a “poem” was indistinguishable from utter nonsense - see figure 13.

![Japanese poem example](image)

Figure 13: Example of neural-network generated Japanese “poem”.

After reworking our game’s theme and mechanics, our team started looking for new ways
to integrate a neural network into our project. Our team reached out to Professor Lindsay about
halfway through the project’s duration, and received an idea about a new digital-artistic
 technique called style transfer (sometimes called style blending). Currently used primarily by
 apps like Snapchat that revolve around taking and editing photos, the style transfer technique is
 used to recreate a base image in the style of a reference image. Knowing that Shiki, as an
 experience game, must be art-heavy to convey story to the player, we thought that style blending
 would be appropriate for our game.

 Our team did not have enough time or skill to fully implement a style transfer neural
 network from scratch, so the implementation “neural-style” by Justin Johnson (Johnson, 2015)
 was used. Running on an Amazon Web Services g2.2xlarge instance (8 virtual cores, 15 GiB of
 RAM, and a Nvidia K520 graphics card with 8GiB of graphics memory), this neural network
 used a process called convolution to break down the reference image and learn the rules around
 how different bodies of color were laid out, and then through content replacement, would recraft
 a given base image with these new rules. Convolution is used in any neural network that has any
 sort of image recognition or analysis. Instead of looking at a picture as a whole, a convolutional
 neural network analyzes every single block of a certain size of the image, finding unique patterns
 in each of those blocks, and then determining which blocks contain the most useful information.
 In a style transfer neural network, like “neural-style,” the computer looks for the most drastic
 color changes in each convolution block, and after each reading of the style image, outputs a
 version of the base image using the content replacement rules it currently has. By comparing the
 output with the input reference image, an error can be calculated, and the program will make
 another attempt using an updated ruling system, up until the maximum number of iterations
 given by the user.

 As we were already creating art for the prologue inspired by the Kanō School of Art, we
decided to maintain this theme and use Kanō paintings as reference images for objects within the
village. After playing around with different runtime options provided by “neural-style,” we were
able to replace the textures of the objects from the asset packs we imported with versions that,
while still having the same basic details of the original assets, were recolored to look like they
were cut from a painting from the Kanō School of Art. We found that using a learning rate of
1e-1 produced the best results. While generally using more iterations also produced better results,
using more than 1000 or 1500 iterations caused the resulting image to have too many artifacts.
Alone, these new textures looked very off-putting on the models in the village, but once every
object was style transferred, the village looked better. The dull and muted, but still somewhat
vibrant, colors of the reference images mixed into the textures of the asset packs we imported
created a village that looked devoid of vitality, matching our intended experience design.
Figure 14: Our bamboo texture. From left to right: Untouched, the Kanō-style painting we used to style transfer with, at 50 iterations, at 250 iterations, and at 1000 iterations of our neural network.

Figure 15: Textures from the door and lantern models before (left) and after (right) style transfer (GabrielMGuimaraes, 2016a)

---

2 Ran with settings -gpu 0 -num_iterations 1000 -optimizer lbfgs -learning_rate 1e1 -original_colors 1 -backend cudnn -cudnn_autotune

3 Ran with settings -gpu 0 -num_iterations 1500 -optimizer lbfgs -learning_rate 1e1 -original_colors 0 -backend cudnn -cudnn_autotune
Figure 16: View of the village before (top) and after (bottom) applying style transferred textures

4.2 Game Mechanics

Shiki implements two game mechanics. The first mechanic is the seasons mechanic that allows players to change objects by placing them in seasonal areas. The second mechanic is the crafting mechanic that allows players to modify objects and create new ones by using tools found throughout the world.
4.2.1 Season Mechanic

The way in which Shiki’s world was implemented in Unity (see figure 17) resulted in some challenges when it came time to begin implementing seasons. We decided to model the ground of each season using Unity’s terrain tool as it is significantly easier to use than a full 3D modelling program. Unfortunately, Unity terrain objects cannot be rotated, and they must be rectangular, so the season terrains ended up overlapping the village ground due to our world blueprint. We solved this by making the season ground drop off sharply where it meets the village, so the village terrain is on top of the season terrain where they intersect.

![Diagram of Shiki’s world representation](image17a.png)

Figure 17: From left to right, then down: (a) Diagram of Shiki’s world representation, (b) Close-up view of a single quadrant of the world map, (c) Heightmap of Shiki’s world, (d) The world as it appears in Shiki

Another solution we considered was placing colliders to cover every possible area where a player could move from one season to another, or between the village and a season. By placing two colliders at each of these entrances/exits - one on one side of the entrance/exit, and one on the other side - it is possible to determine what area the player is in by remembering the last collider that they exited. This solution also poses some problems - it is more fragile than other solutions and is more affected by terrain changes than the solution we ended up choosing.
Our final solution uses a raycast to check if the object is above a special season area object. Another special object exists above the season area object, to prevent a season from incorrectly reporting that an object that is inside the village is inside the season. In retrospect, it would have been more efficient to mathematically determine whether a coordinate exists within a season. The layout of the world is sufficiently complicated that the math for determining what season a coordinate is in is not obvious, so during development we decided to simply implement the raycasting approach as a “quick and dirty” solution. This approach has the added benefit that the season bounds are visible inside the Unity editor, making it easier to check that the bounds are synchronized with the seasons.

When a seasonal object is placed inside a season, its other variants are moved to be in the same relative position inside their seasons. Determining the season-relative position of an object is done by taking the arctangent of the object’s position (in the ground plane, XZ) and adjusting it so that it is between 0 and 90 degrees. The distance of the object’s XZ position from the center is also saved. To convert from the season-relative position to world coordinates for a particular season, the “start angle” of the season is needed. This is the angle at which the season’s quadrant begins. This angle is added to the season-relative position angle, and the world coordinates can be obtained by taking the cosine/sine of this angle and multiplying that value by the radius. Refer to figure 18 and figure 19 for more information on converting between season-relative and world coordinates.

![Diagram showing how season-relative coordinates are calculated](image)

**Figure 18:** Diagram showing how season-relative coordinates are calculated
The logic for the seasonal effect mechanic is spread throughout three different components: `GrabObject`, `SeasonInfo`, and `SeasonalEffect`. `GrabObject` deals with the player grabbing and letting go of objects, and is only involved in the seasonal effect logic to the extent that it notifies `SeasonInfo` and `SeasonalEffect` if the player picks up or places an object. Each season has a `SeasonInfo` script on an empty object inside of it, and it is mainly used to determine whether an object lies within that season, as well as to spawn extra copies of a seasonal object when it is placed for the first time. The `SeasonInfo` script also stores the start angle of the season it is inside.

`SeasonalEffect` is attached to each variant of a seasonal object, and handles keeping track of the object’s season, and changing the object’s model as needed. The object’s model is changed by loading a new object from a prefab, copying the `SeasonalEffect` to it, and then deleting the original object. The name of the appropriate prefab for an object in a given season is specified in a TOML file (`VariantDatabase`, see figure 20 for an example entry), enabling some flexibility in prefab file names, and making it easy to set a “default” prefab and override it for only a single season.

---

4 Atan2(y, x) = θ is defined as the following (Math.atan2 method (double, double) (system).)
- If y is 0 and x is not negative, θ = 0.
- If y is 0 and x is negative, θ = π.
- If y is positive and x is 0, θ = π/2.
- If y is negative and x is 0, θ = -π/2.
- If y is 0 and x is 0, θ = 0.
name = "Shiroyama"
prefab = "Shiroyama_DefaultPrefab"
[item.variant.Spring]
prefab = "Shiroyama_SpringPrefab"
[item.variant.Fall]
prefab = "Shiroyama_FallPrefab"

Figure 20: Example entry in the VariantDatabase file
Figure 21: Diagram of control flow after an object is placed.
4.2.2 Crafting Mechanic

One of the major focuses of Shiki’s gameplay is the crafting mechanic. This mechanic helps improve immersion and add an element of mastery to the game. The crafting mechanic is implemented mainly in a single script: ToolFunction.

ToolFunction is responsible for detecting collisions between the tool and other objects. The tool is intended to only cause an effect on craftable objects, thus collisions detected by the tool that are not with craftable objects are ignored. ToolFunction calls a craftable object’s hit() function, if a collision is detected, with the velocity magnitude information present in Unity’s Collision class as a parameter.

```csharp
public class ToolFunction : MonoBehaviour {
    public float strengthThreshold = 0.25f; // strength at which collision of object produces a result

    void OnCollisionEnter(Collision col) {
        if (col.rigidbody != null && col.rigidbody.gameObject.GetComponent<HittableBehavior>() != null) {
            var velocity = col.relativeVelocity.magnitude;
            var material = col.gameObject; // object player is colliding tool with
            hit(material, velocity);
        }
    }

    void hit(GameObject material, float velocity) {
        if (velocity >= strengthThreshold) // if player collides tool with object hard enough
            EventManager.FireEvent(new ObjectHitEvent(material, this.gameObject));
    }
}
```

Figure 22: ToolFunction.cs script, showing how collision detection works.

This implementation, while elegant, does not fully align with our experience design and goals. A player can cause any craftable object to change despite the physical movements they perform. This creates a disconnect with reality - one would not expect to chop down a tree by
performing a stabbing motion, yet this implementation allows this to happen. The later iterations of the crafting mechanic attempted to use a combination of number of hits, when they occurred, and where they occurred on an object to classify the motion the player was performing. However, through preliminary testing, it was determined that the thresholds and position data used for this implementation were too broad to classify a behavior. The function used to classify movement determined that all behaviors were occurring no matter what motion was performed. Furthermore, the assumptions based on changes in positions along each axis limited our scope of possible crafting behaviors beyond what we had planned for our quests. It is likely that using the velocity and position of the tool as it travels from the beginning of the player’s motion to the collision to the end of the player’s motion would have been a more accurate and representative way of classifying crafting behaviors. If more time were available to fix and further refine this classification method, this iteration of the crafting mechanic would have been the ideal solution to improving the player’s experience with crafting. Due to the limited time remaining during our implementation, we reverted to the second iteration of our crafting mechanic that required players to hit craftable objects more than once to cause a change to these objects.

4.3 Game Managers

In order to implement certain systems, such as quests and events, implement features, such as the inventory, and keep our code organized, we developed a set of game managers. These game managers consist of the EventManager, the QuestManager, the SaveDataManager, and the Inventory System. The following sections discuss each one in more detail.

4.3.1 EventManager

From the beginning of the project, we knew that we had to be careful to keep the code from different parts of our game from being too tightly coupled (to enforce separation of concerns). The importance of loosely coupled code had been instilled in us in many of the computer science courses we had previously taken at WPI.

There are many tools and design patterns that can be used to help decouple parts of a program. Some common examples of this are the Model-View-Component pattern, dependency injection frameworks, and messaging systems (communication between parts of a program using some “external” protocol). During the development of our game, we considered using a dependency injection framework, but decided against doing so due to the unknown amount of effort it would take to integrate it with Unity’s C# project setup.

In the end, we decided to use an event system for communication between different parts of our game. In this context, an event system is a system that allows code to register an interest in some particular event, and allows code to fire events. When an event is fired, any code that had registered interest in that kind of event will be notified of the event’s firing.
Events are a commonly used approach for decoupling video game code. As a result, Unity offers two built-in event systems, in addition to C#’s built-in event system. We evaluated each of these systems before we began developing our project, and determined that we could improve on them with our own event system implementation.

The first of Unity’s built-in event systems is called the “Messaging System” (Unity - manual: Messaging system.2017). This system is tightly integrated with Unity, and is used by Unity’s Standalone Input Module and Touch Input Module. To create a new message type, the user must create a new interface that extends the IEventSystemHandler interface. The functions that the user adds to this new interface are the functions that can be triggered by a message. The user can then implement their new interface in a class that also inherits from MonoBehavior to allow objects of that class to receive messages.

Due to the way this system is implemented, it is highly integrated with Unity GameObjects, and depends heavily on the hierarchy of GameObjects within a game. A message must be targeted at a GameObject – there is no way to raise an event globally. This is because a message is only propagated to the children of a GameObject that receives a message. Additionally, the SendMessage function is significantly slower than using C# events (Dunstan, 2016).

Unity also has another, relatively new, event system called UnityEvent. This system is almost undocumented, with the only official documentation surrounding it being a few sparse pages in the Unity Scripting API documentation (Unity - scripting API: UnityEvent.2017). This system is quite similar to C#’s built-in event system, with the added features of additional integration with the Unity Editor, the ability to fire events from classes outside of the class where the event was defined, and the ability to access persistent (added in the Editor) listeners. However, one limitation of UnityEvents that C# events improve on is that UnityEvent handlers can have at most 3 parameters, whereas C# event handlers can have at most 16 (this is likely due to Unity being based on .NET 3.5, where many generic types, including Func and Action, only had implementations for up to 4 parameters).

Listeners can be added to a UnityEvent by calling its AddListener method, and removed using the RemoveListener method. A listener can also be added in the Editor, by dragging a GameObject to a box in the Inspector window of the object containing the UnityEvent, and then selecting a method on it or on one of its components (MonoBehaviors) from a dropdown menu. This means that listeners cannot be added across scenes - which makes UnityEvents almost useless for our game, since we need to send messages across scene boundaries.

Finally, we have C#’s built-in Event implementation. Since this is built into C#, it is how most non-Unity C# code exposes and uses events. To create a new event, the programmer has to first create a delegate type for the event’s handler. Basically, this is just writing the function signature that handlers for the event must have. Then, the programmer can create an event field inside of a class. Using this event field, the programmer can attach and detach listeners for the
event, and fire the events. Note that C# events can only be fired from the class in which the event field resides - though in practice this isn’t a huge issue, since one could just create a public method that simply fires the event.

The major downside with using C# events is the large amount of boilerplate code that it requires. In our case, where we need to be able to fire events from outside of the class that their event field is in, that means that adding a new event to the system requires defining three things:

- A new class to store any information that the event’s handlers need to access (generally derived from the EventArgs class)
- A new event delegate type
- A function that allows outside code to fire the event

(note that the EventArgs subclass is not required, since it is possible to add arbitrary parameters to the delegate type. However, the convention used heavily in C# is to pass any extra information in an EventArgs subclass.)

After investigating these three event systems, we determined that it would be worth it for us to develop a custom event system implementation. Though one might see this work as being an unnecessary reinvention of the wheel, we have devised a system that is more convenient for our use case. Though it may not have as many features as some available 3rd party event system libraries, it is sufficient for our needs.

Our system is called EventManager, and it is based around C# delegates (though not the full C# event system). An event in our EventManager system is simply a class that implements the IGameEvent interface. This interface is empty: its sole purpose is to mark which classes are events and which are not. This adds an additional level of type-safety that helps catch more errors at compile-time. The programmer can add arbitrarily many fields and methods to the event class, to store whatever data needs to be passed on to the event handlers. For example, here is the definition of one of the events used in our game (some code omitted for brevity):

```csharp
/// Fired by: GrabObjects
/// Fires: When an object is picked up
public class ObjectPickedUpEvent : IGameEvent {
    /// The GameObject that was picked up.
    public GameObject pickedUpObject;
    ...
    public ObjectPickedUpEvent(GameObject go) {...}
    ...
}
```

Figure 23: ObjectPickedUpEvent definition
To attach a handler to an event, the programmer can simply call the
EventManager.AttachDelegate method, as shown below.

```csharp
EventManager.AttachDelegate<ObjectPickedUpEvent>(this.OnObjectPickedUp);
```

While some might believe that having to explicitly state the generic type parameter (the event type) is verbose, I believe that it is good to have to state it explicitly, to make it more clear what event the method is being attached to. The compiler will also ensure that the method being attached has the correct signature – it must take in a single argument, with the argument’s type being the event type that the function will be attached to.

To fire an event, the syntax is even simpler:

```csharp
EventManager.FireEvent(new ObjectPickedUpEvent(pickedUpObject));
```

In this case, it is fine to omit the generic type parameter, since the event type is explicitly stated inside the call.

This design for EventManager helped shape the design of the rest of our codebase. Broadly speaking, we ended up writing 3 different kinds of components (MonoBehavior subclasses): empty components that simply served as tags, small components that listened for one or two events and performed simple tasks, and large components that contained a lot of logic.

The “empty components” were mainly used as a workaround for limitations of Unity’s label/tag system. We needed a way to mark certain GameObjects as being valid targets for certain kinds of interactions (i.e. some GameObjects would react to being hit, some GameObjects were able to be merged into other GameObjects, etc). Unity does have a “tag” system which allows each GameObject to have a string attached to it, but it only supports a single tag per GameObject. This was insufficient for us, since a GameObject might be a valid target for multiple kinds of interactions.

An inheritance-style approach to this problem might be to make a subclass of GameObject which includes a boolean field for each type of interaction that a GameObject could be a target of. Then, this field could be set appropriately for each type of interaction the object supports.

To keep with Unity’s component design, we instead decided to create a few components that exist solely for the purpose of signalling to other components whether a GameObject is a valid target for a particular interaction. These components are empty classes that subclass MonoBehavior (see figure 24).
Signals that the GameObject is a valid target for another object to be dropped onto
public class DropTargetBehavior : MonoBehaviour {}
<table>
<thead>
<tr>
<th>Component</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeleteableBehavior</td>
<td>Object can be deleted</td>
</tr>
<tr>
<td>DeleteOnMergeBehavior</td>
<td>Object is deleted when it is merged with another object</td>
</tr>
<tr>
<td>DroppableBehavior</td>
<td>Object can be dropped on top of another object</td>
</tr>
<tr>
<td>DropTargetBehavior</td>
<td>Object can detect if another object is dropped on top of it</td>
</tr>
<tr>
<td>HittableBehavior</td>
<td>Object can be targeted by a tool</td>
</tr>
<tr>
<td>InventoryItemBehavior</td>
<td>Object can be stored in the inventory</td>
</tr>
<tr>
<td>LeaveableItemBehavior</td>
<td>Object can detect if it was left to wait for a number of seasons</td>
</tr>
<tr>
<td>MergeableMasterBehavior</td>
<td>Object can have another object merged into it</td>
</tr>
<tr>
<td>MergeableSlaveBehavior</td>
<td>Object can be merged into another object</td>
</tr>
<tr>
<td>ReplaceableBehavior</td>
<td>Object can have its model replaced with another one</td>
</tr>
<tr>
<td>SeasonalEffect</td>
<td>Object can have different model based on the season it's in, and creates copies of itself in other seasons</td>
</tr>
<tr>
<td>ShowHideChildBehavior</td>
<td>Object can show and hide its child object</td>
</tr>
<tr>
<td>ToolFunction</td>
<td>Object can be used as a tool on other objects</td>
</tr>
</tbody>
</table>

Figure 26: Table of components used in Shiki

4.3.2 QuestManager

The QuestManager was born from the need to keep track of the tasks given to the player throughout the game. The player would complete these tasks - which when grouped together are referred to as quests - and the game would need to respond appropriately. The main challenges in designing this system arose from our need for detailed and varied tasks, a complex way to structure those tasks, and a method for the quests to interact with the game and the player.

Tasks are the fundamental constituents of a quest. They have names, fulfillment requirements (known as triggers), completion sequences (known as onComplettes), and an order. A task can have children, known as subtasks, which are other tasks that must be completed before their parent task can be completed. Triggers are the actions the player must perform in order to complete the task, which include entering specific areas, picking up certain objects, using tools on objects, using parts of the UI, and more. OnComplettes are the actions that occur as a result of a task being completed. OnComplettes can affect the UI as well as objects in the game.
4.3.2.1 SaveDataManager

When the game is loaded, all of the tasks are read into the QuestManager class from TOML formatted files by the SaveDataManager. The SaveDataManager was created in order to provide the player with the ability to leave the game and return without losing progress. This would mean saving object locations, inventory state, and task state. Loading tasks was written in such a way that the state of the tasks were read in first, followed by the task information itself. If no task state file existed, only the tasks would be read in.

We designed our SaveDataManager to be agnostic of the format used to store data in the state files. We achieved this by creating interfaces for reading and writing data to the quest state file, and reading data from the quest task file. These interfaces were called the IQuestStateReader, IQuestStateWriter, and the IQuestReader. We then implemented these interfaces in concrete classes that used Nett to read and store TOML data in the state files.

If the task state file exists, the SaveDataManager reads it in using the LoadQuestState function, which returns a Dictionary\<string, bool\> containing the tasks found in the file and their state. Our concrete implementation of the IQuestStateReader interface is the TomlQuestStateReader class, which reads TOML files containing the task states. The task state file contains key-value pairs of users and the names of the tasks they have completed, separated by spaces in a string. We did not complete the implementation of multiple user support, but this design allows for it.

Once the task states have been read in, the SaveDataManager then reads in the complete task tree using IQuestReader’s LoadQuestTasks function, which returns all of the tasks in the form of a list of quest trees. Much like the IQuestStateReader, we chose to implement a TOML file reader for the task file called TomlQuestReader. The task file contains each task in the form of a TomlTable, where each task has the following key-value pairs: Name, SubTask, Trigger, and OnComplete; all values are in the form of a string, and the only required key is the Name key. Multiple SubTasks are written within the same value string, separated by spaces. The TemporaryTask class was designed to parallel this format, and as such contains the fields string Name, string SubTask, string Trigger, and string OnComplete. This enables the file to be read more easily. The task information is read from the TOML file using the Toml.ReadStream function, which returns a List of TemporaryTasks.

With a List of TemporaryTasks, the next step taken by the TomlQuestReader is to convert this list to a collection of TaskNode trees. The TaskNode class contains three fields: List\<TaskNode\> children, TaskNode parent, and Task associatedTask, where the first two fields describe the relationship between a single TaskNode and the other TaskNodes in the tree, while the third contains the instance of the Task class that the TaskNode itself is associated with. The Task class contains similar fields to the TemporaryTask, however slightly fewer strings are involved: string name, string[] subTasks, bool isComplete, Predicate\<InteractionEvent\> trigger, and Action\<InteractionEvent\>.
onComplete. InteractionEvents are discussed in more detail in Appendix H. An additional static helper class called TemporaryTaskConverter was created to facilitate the conversion of a TemporaryTask list to a TaskNode tree, and back. The function that is called in the TomlQuestReader is the TempTaskToTaskTree function, which returns a list containing all TaskNode trees.

The first step in the TempTaskToTaskTree function is to convert the list of TemporaryTasks to a list of Tasks. This begins with passing each TemporaryTask, along with its state, to the TempTaskToTask function. In this function, the value of the Name field in the TemporaryTask object is assigned to the name field in the Task object; the string SubTask is split up by spaces and assigned to the string[] subTasks field of the Task object; and bool isComplete field of the Task object is set to the state of the task (complete or not complete), which is passed in as the parameter bool isComplete. At the start, the Predicate<InteractionEvent> trigger and Action<InteractionEvent> onComplete are set to null, as they are optional. The process of creating triggers and onCompletes is discussed in more detail in section 4.3.2.2: Triggers and OnComplete.

Once the list of all Tasks has been created, the next step is converting the list into a collection of trees. The reason why only a single tree is not returned is because each tree represents a single quest. We had wanted to give players the ability to work on any set of tasks that they had met the prerequisites for in any order of their choosing, and as such multiple quests were needed. The code for transforming a flat list of tasks into a list of trees of tasks, heavily referencing a solution from Stack Overflow, is shown in figure 27 (Afshari & Preite, 2013).

```csharp
public static IEnumerable<TaskNode> TaskListsToTaskTrees(List<Task> tasks) {
    var lookup = new Dictionary<string, TaskNode>();
    tasks.ForEach(t =>
        lookup.Add(t.name, new TaskNode { AssociatedTask = t });
    );

    foreach(var item in lookup.Values) {
        var proposedChild;
        foreach(string st in item.AssociatedTask.subTasks) {
            if(lookup.TryGetValue(st, out proposedChild)) {
                proposedChild.Parent = item;
                item.Children.Add(proposedChild);
            } else if(!String.IsNullOrEmpty(st)) {
                // could not find child
            }
        }
    }
    return tasks.Select(t =>
        var proposedChild;
        foreach(string st in t.ChildrenNames) {
            if(lookup.TryGetValue(st, out proposedChild)) {
                proposedChild.Parent = item;
                item.Children.Add(proposedChild);
            } else if(!String.IsNullOrEmpty(st)) {
                // could not find child
            }
        }
    );
```
Functions for writing save data were created; however, they were not used in the final game. In its current state, only the status of the tasks are written. Much like reading saves, writing saves is also done with a TOML implementation of IQuestStateWriter. A single key-value pair per user is used, where the key stores the user’s name, and the value is a space-separated string containing all of the names of the quests that user has completed.

4.3.2.2 Triggers and OnCompletes

As our game was designed to focus on experience, the ability to add detailed and unique triggers and onCompletes was required. For each task with a trigger or onComplete, a string, written in a custom language, is passed in, and then parsed by the ParseString function. This function returns an object of type ParsingResult, which has various information involving the trigger/onCompletes. Many of the fields are optional, and their usage is dependent on the type of the action, which is stored in an enum known as InteractionKind. An InteractionKind is a keyword used directly in the language to define the type of interaction between objects; a full example of a trigger is “Player Hit Item SteamedRice With Item Hammer.” In this sentence, the word “Hit” is the action: it describes what must happen between the item “SteamedRice” and the item “Hammer.” The word “With” also plays a key role here: it specifies the relationship between the two objects involved in the action (i.e. one would not normally try to hit a hammer with rice in order to pound rice).

ParsingResults are used to create the appropriate trigger Predicate or onComplete Action. In both cases, the first portion of the ParsingResult that is checked is the type of InteractionKind. From the InteractionKind, the necessary additional information is taken (i.e. objects involved, location, etc). Each InteractionKind has its own specific set of information that must be included in order to be parsed and transformed into a Predicate or Action correctly. In the case of triggers, the InteractionKind is passed in, and the Predicate<InteractionEvent> that is returned is a check of whether the key information in the trigger matches the information that is being passed to it. For example, for the case of the InteractionKind PickUp, the predicate checks whether the name of the picked up object matches the name specified in the trigger sentence. In

```csharp
} } return lookup.Values.Where(x => x.Parent == null);
```
the case of onCompletes, the InteractionKind is also passed in, and the
Action<InteractionEvent> will fire the appropriate IGameEvent once its attributes have
been set. For example, in the case of InteractionKind Become, the fired event is a
ReplaceObjectEvent, which contains the object to replace, and the object it should be
replaced with. Additional information about triggers and onCompletes and the language they are
written in can be seen in Appendix G.

4.3.2.3 QuestManager and InteractionEvents

Whenever an InteractionEvent is fired, the QuestManager calls the UpdateTasks
function. This function goes through each individual tree in the collection of trees and calls
UpdateTasksBranch (see figure 28) on each tree, working recursively on each TreeNode until
all of the nodes in each tree have been checked. The doAll parameter was created enable the
possibility of re-completing individual tasks even after they have been completed; however, this
functionality was not used in the end. Tasks are checked by checking that the trigger, when
passed the InteractionEvent, returns true. If the trigger returns true, or does not exist, the
onComplete is then run and the game updates itself as appropriate.

```csharp
public bool UpdateTaskBranch(InteractionEvent evt, TaskNode tn, bool doAll) {
    bool allChildrenComplete = true;

    foreach(TaskNode tchild in tn.Children) {
        allChildrenComplete &= (tchild.AssociatedTask.isComplete ||
                                UpdateTaskBranch(evt, tchild, doAll));
        // if child not complete, and updating child doesn't complete
        // the child, allChildrenComplete = false
    }

    if((doAll || !tn.AssociatedTask.isComplete) &&
       (allChildrenComplete && (tn.AssociatedTask.trigger == null ||
                                tn.AssociatedTask.trigger(evt)))) {
        tn.AssociatedTask.isComplete = true;
        tn.AssociatedTask.onComplete(evt);
        return true;
    }

    return false;
}
```
4.3.3 Inventory System

We decided early on in the design of the game that it would be beneficial to implement an inventory system. Having an inventory allows the player to conveniently bring items with them, without having to hold down the “hold item” button on the controller. Additionally, we ran into a few issues where sometimes a held object would disconnect from the player’s hand due to certain kinds of collisions with other objects. Giving the player an inventory helps mitigate this issue, since the player can place objects in the inventory instead of holding them for a long time, and makes for more convenient gameplay.

The traditional overlay or sidebar inventory interfaces used in many non-VR video games do not translate well to a VR environment. To design our inventory interface, we looked at many existing implementations of VR inventory interfaces for inspiration. We found a few general styles of interface: a “tray” that items can be dropped on and picked up from, a grid of boxes that can each hold an item, a backpack that has “storage points” on it, and a more traditional menu interface that is controlled using the user’s hands - see figure 29 for examples of each of these styles of interface.
We first considered using a backpack-style interface (similar to figure 29 (d)), since it does not break immersion like many of the other interface styles do. During our background research, we found that “backpack chests” were used by Japanese merchants during the Edo Period (Japanese edo period backpack chest – edo arts.; Edo period merchant,s backpack chest – edo arts.). These backpack chests are effectively a normal chest of drawers with straps to allow it to be carried on one’s back. We decided against such an interface because we believed it would
be inconvenient for the player to use: both of the backpack style interfaces that we looked at required the user to have one free hand to open and close the inventory. Additionally, the backpack-style interfaces can be more difficult to use, since the item storage areas are spread across the backpack.

Since we had decided not to use the Leap Motion sensor, which would have allowed us to track the player’s hands, our choice came down to either a tray-style interface, or a grid-style interface. We decided to use the grid style interface, because we believed it would be easier to implement. See figure 30 for a screenshot of our implementation.

![Screenshot of the inventory interface](image_url)

Figure 30: Screenshot of the inventory interface

When the inventory is closed and the player presses the one of the Vive controllers’ side buttons, the inventory positions itself 0.5 Unity units in front of the headset’s reported position, and faces the headset. Then, the inventory makes itself visible. We found that this puts the inventory interface roughly an arm’s length away from the player, and allows the player to see the entire inventory when its capacity is 4 items.

The inventory system is set up by the InventoryManager script when the game is loaded. In the Unity Editor, the number of inventory slots is exposed as a customizable number. When the InventoryManager is created, it first creates a backend with the desired number of inventory slots, and then generates and positions that same number of inventory target prefabs (this prefab is set in the editor). Currently, the targets are positioned in a single row, and their size is set by a few private variables inside the InventoryManager script. If desired, these variables could be made public, and thus accessible for modification in the Unity Editor.

“Inventory target” is our term for an area that is part of the inventory in which a single item can be stored or removed. The inventory target prefab contains a cube with a special shader on it that shades an outline of the cube’s polygons. This makes the cube mostly transparent,
allowing the player to see the item inside of it, if any. The cube also has a script on it - InventoryTarget. This class takes care of handling the detection of collisions between inventory items and inventory targets, and it keeps track of the inventory target’s contents.

Most of the logic for the inventory system is contained inside the InventoryItemBehavior script. This class is attached to any GameObject that should be able to be placed in the inventory. To function properly, the InventoryItemBehavior requires that the GameObject have both a RigidBody and a MeshFilter. The majority of the code inside this class is responsible for scaling down and positioning the item inside the inventory target. This class also is responsible for disabling gravity’s effects on the item when it is placed inside an inventory target, and reenabling them when the item is removed from an inventory target.

Most of the inventory logic exists inside the InventoryItemBehavior class. This class listens for events emitted from InventoryTargets that notify the InventoryItemBehavior instance when an object with InventoryItemBehavior attached to it has entered or exited an InventoryTarget. The InventoryItemBehavior instance uses this information to keep track of which InventoryTargets it is currently colliding with, of which there may be multiple depending on how close the inventory targets are, and how large the item is. It also listens for ObjectPlacedOnInventoryTargetEvents, which are fired by the script that handles picking up and placing items (GrabObjects).

4.4 Art and Assets

The following section details our artistic techniques for the implementation of the ideas listed in the Design chapter. The 2D prologue, 3D world, and sound and art assets are described here.

4.4.1 Prologue Art

To tell the story of Shiki, our game employs a cutscene at the beginning featuring hand-drawn images that represent the story. The prologue cutscene consists of five unique physical panels. These panels were created using watercolor pencils on watercolor paper with inspiration from the style of the early Kanō School of Painting. The panels were scanned at various points during their creation, and modified in Adobe® Photoshop®, to result in eleven distinct panels for the final cutscene.

When determining what styles and techniques we would use to create this cutscene, we considered our skills as artists and the feelings we were trying to achieve. None of us are dedicated artists, so the style and medium we chose had to be something that would allow us to produce decent quality art in a reasonable amount of time. We chose to follow the style of the Kanō School of Painting as it was a style that heavily relied on small groups of color to dictate both shape and depth. This style did not require us to paint intricate details or require complex blending of colors, so it was appropriate for our skill level. We also chose the Kanō School of
Painting because we wanted to reinforce to the player the time period in which the game takes place. There are other styles and schools of art throughout Japanese art history that we could have mimicked due to their simplicity; however, these art styles did not originate during the Edo Period, and we felt it would not be true to our game’s background to choose an unrelated art style. Finally, we chose to use the watercolor medium as it was something easily available to us for a low cost and something we had worked with before.

Our early design consisted of a rather basic storyboard of eight frames, which included a village scene, a farming scene, scenes denoting the iraiyama, and a shrine scene. This storyboard acted as the outline for our design. When we drew out these scenes using the watercolors, we realized that our scenes did not convey the entire story we wanted our players to see. We had expanded the number of iraiyama scenes to better illustrate the transition of plentiful resources to no resources, but realized through explaining the story in playtesting, before we implemented the cutscene, that we needed an additional scene to complete the story. We added a scene to represent the four seasonal quadrants surrounding the village to show its current state.

Each watercolor scene was constructed using the following process. Our watercolor paper was completely white, but Kanō style paintings were traditionally done on gold-leaf paper screens. We first painted each sheet of paper a golden-brown color to mimic the gold-leaf paper material. Once this was completed, we sketched outlines for the designs and then filled them in using the watercolor pencils. Finally, a brush was used to even out and intensify colors so that it appeared the images were created using ink, rather than watercolors, as ink was the traditional medium used in Kanō style paintings. An example of this technique with the village scene is shown in figure 31.
The *iriaiyama* and shrine scenes were created in a reverse order from their playback order so that we could create multiple, nearly identical scenes in a short period of time. The *iriaiyama* scene was painted in five stages, which were scanned when completed before moving on to the next scene. The shrine scene was created in a similar fashion, except with only two scenes. Scenes that lacked color, represented by the grayscale images, were modified using Adobe® Photoshop®. We wanted to remove the color from the scenes without removing the base coloring we added to the paper, so that we could convey to the player that these paintings were also following the Kanō style medium. Figure 32 illustrates the final product after color was removed. These scenes were created by placing their colored counterparts into Photoshop, removing all color saturation, and then using the History brush tool to draw over places on the image where we wanted to restore color, which was in the image background.
4.4.2 3D Design

Shiki, as an experience game, needed a visual world that would complement the story. As no one on the team was skilled with 3D modeling to the extent that Shiki would look professional, the team had to rely on asset packs from the Unity Asset Store and outside help. Asset packs used include the Modular Japanese House pack (GabrielMGuimaraes, 2016b), the Japanese Scenery Set pack (JD Creative Machine, 2013), the Edo Japanese Town Props pack (Tiny Utopia, 2013), the Stylized Environment Effects pack (VFXCraft, 2016), and a set of custom human-statues created by WPI graduate John Guerra (WPI ‘16). With these packs, the process of actually creating our game world was as simple as loading up the Unity editor, dragging in individual objects, and positioning everything correctly.

With very few difficulties, we were able to create a game world that we were happy with. The Japanese Scenery Set provided most of the seasonal design, while the Modular Japanese House pack and Edo Props provided most of the design for the village. We used the Stylized Environment Effects particle effects throughout the game world to give each season its own
unique feeling. John Guerra’s human-statues helped solidify the uneasy feeling we wanted to get out of the village.

The main difficulty we faced in creating our game world was the inability to use Unity’s snap tool. In most cases, this tool allows users to align objects with the global grid on the closest interval that the user sets. While it would have been useful, the snap tool does not account for assets of different sizes or assets where the origin is not at the corner of the object. Because of this, each and every object had to be moved manually to verify that walls lined up nicely and corners were connected correctly. Although time-consuming, the fact that the village was the only place in our world that needed these fine details made the process go rather smoothly.

Another issue that we ran into involved Unity-generated Trees. Although helpful, there are a few issues with Unity trees that required the use of third-party tools to bypass. One issue involved the fact that trees are all prefabs. Even though a user can load in two of the same tree and edit the transform components of the trees at will, if the user tries to edit the actual tree structure (including the randomization seed), all trees referencing that prefab will be modified. Even if the user opens up the asset files and duplicates the prefab within the file explorer, there are a few unexpected problems that can appear. In the end, Broken Vector’s Tree Randomizer (Broken Vector, 2017) was incredibly helpful in generating a lot of trees from the same prefab. This tool took in a single tree as an input and simply created a user-defined number of trees that not only could be added to the world, but individually edited. This tool allowed our team to confirm that every tree in a seasonal quadrant had a perfectly identical (besides the leaf color) tree in the same location in every seasonal quadrant.

The final issue, while very minor, caused the team some trouble in working around it. Unity has a native terrain object that can have height painted over it to create realistic-looking hills and rivers. Unlike any other native object in Unity, however, terrain is specifically locked to the XZ-plane, and does not support rotation of any kind. This meant that in order to create four identical terrains for our season-quadrants, we had to export the heightmap of the first quadrant, rotate the file in an external program, and then reimport the rotated files as new terrains. While it was a manageable fix, this process discouraged the team from making small adjustments to the seasonal terrain after the rotated files were created.
4.4.3 Sound Assets

All sounds incorporated into Shiki were incorporated as ambient and locational sounds. Ambient soundtracks were created for each season to reinforce the concept that the player is in one season or another. These ambient soundtracks were created using a collage method from nature sounds recorded in and around the Hyogo, Osaka, and Kyoto Prefectures and from online sources with unrestricted use. Sounds were edited and combined using Adobe® Audition®, and generally followed a basic process of frequency isolation and amplification, cutting and rearranging clips, and finally compressing all clips so that all sounds in the game remained at a consistent volume level so as not to ruin the relaxing aspects of our experience design. All ambient soundtracks were inserted into the Unity project as 2D music tracks in their appropriate scenes so that the audio listener could prioritize these tracks over other sounds that might be later added to the game.

The Summer and Autumn seasons’ ambient soundtracks were assembled using a base of ambient recordings taken at Kabutoyama Forest Park in Hyogo Prefecture during early August. These base recordings primarily feature the sounds of cicadas and other tree insects but also subtly include sounds of native bird species. These ambient recordings served as the perfect base for Summer, as the sound of cicadas is a key aspect of Summer in Japan. The ambient track for Autumn was created using an ambient recording taken during the brief moments cicadas ceased making sounds.

While these recordings contained many characteristic sounds of Summer and Autumn, they also contained artifacts of sounds from hikers, vehicles, and a bell at a nearby shrine. These
sounds were filtered out by applying a band-pass equalization filter in Adobe® Audition® that heightened the amplitude of sounds occurring in the frequency ranges of 200Hz-800Hz and 1200Hz to 20KHz. The sounds of hikers and vehicles mainly fell below 200Hz and the bell between 780Hz and 1150Hz, so these sounds were mostly filtered out by the band-pass filter. The recordings for Summer and Autumn were further modified to create seamlessly-looping tracks by isolating “clean” areas of the recordings that did not contain remnants of filtered-out artifacts. These areas were then further split into smaller clips and had their order inverted. Crossfades were applied at the clip boundaries to ensure that sounds were sequential and level in their volume and pitch. Finally, these ambient tracks were compressed so that their volume level would not peak past -6dB so that audio distortion and clipping would not happen no matter what audio system would eventually be used to play these tracks in-game.

The ambient soundtrack for Winter was created using different sound sources. Since the recordings our team obtained were collected during the Summer months, the sounds contained in them were too characteristic of Summer and contained little to no sounds characteristic of Spring and Winter. The Winter soundtrack was assembled over a base recording taken during Typhoon Noru, which made landfall on the Kansai Region on August 7th, 2017. This recording featured strong wind sounds, some of which peaked past -6dB and caused clipping in the original recording. First, “clean” clip-free portions of the recording were isolated, split, and rearranged, following the same process for the Summer and Autumn soundtracks. However, equalization filters were applied after clipping and rearranging instead of before. A combination of a Scientific Function band-pass equalization filter and standard equalization filter were applied to the recording, respectively to create a hollow-sounding wind. The volume of this clip was then dramatically reduced before applying a compressor to ensure sound peaks did not exceed -6dB. The resulting soundtrack for Winter mimics a calm but hollow wind.

The ambient soundtrack for Spring was created in a similar fashion to the Summer and Autumn soundtracks. A recording taken at a lake on Osaka University’s campus just before dusk was used as the base sound for the Spring soundtrack. This recording featured gentle sounds of water, wind rustling tree leaves, and a few insects but was mostly silent. To reinforce the concepts of spring, recordings of various birds found in the Kansai region during Spring were inserted over the ambient track to form a sound collage representing Spring. These bird recordings were found on the website of NPO Bird Research Organization. As with the other ambient season soundtracks, the Spring soundtrack was compressed so that volume level did not exceed -6dB.
5. Playtesting

Our project involved an iterative design process, in which we collected feedback and testing data on earlier prototypes, so that future prototypes of *Shiki* could use this data to make improvements on our planned designs. We completed two rounds of playtesting at Osaka University. These playtesting sessions tested the Alpha and Beta 1.0 prototypes of *Shiki*. This chapter discusses the prototypes, playtesting environment, and procedures followed for each testing session, as well as the feedback and data collected from these sessions.

5.1 Playtesting Procedures

Our study was conducted in two phases, consisting of two separate digital prototypes of our game. These prototypes were denoted as Alpha and Beta. Each phase of our study was conducted using the HTC Vive VR headset and handheld controllers.

The first phase of our study was intended to test the Alpha version of our prototype. The second phase of our study was intended to test the Beta version of our prototype, which included changes made to the Alpha prototype as a result of Alpha testing. As a result of Beta testing, we developed a final prototype which included changes made to the Beta prototype as a result of the previous playtesting sessions. However, we did not formally test this last prototype.

5.2 Alpha Playtesting

The purpose of this playtesting phase was to test the functionality of the seasons mechanic and controls in our first digital prototype of the game. The game world consisted of the basic world map (a central village surrounded by four identical-landscape quadrants) and some placeholder game objects as they were intended to be placed in the game world for our final version. The objective of this playtesting session was to ensure that the seasons mechanic worked, that players could easily understand the concept of the season mechanic, and to test the controls related to teleportation and picking up object throughout the game world.

At the beginning of each playtesting session, we explained to the participant what tasks they would be completing and assisted them with putting on the headset and controllers. After allowing the playtester to become acclimated to the VR environment, we instructed them to complete the following tasks:

- Using the handheld controller, teleport to another location in the game world
- Locate the white cube in the center village and place it in one of the seasons quadrants. Observe the color change. Teleport to the other 3 quadrants and observe the cube in the same position, with its different seasonal colors.
- Locate the cylinder in the Summer quadrant. Pick it up and place it in the Winter quadrant. Observe the color change from green to blue. Teleport to the 3 other quadrants and observe the 3 new cylinders, all with seasonal color changes
• With the time remaining, for as much time as you like, explore the environment and interact with it as you might do naturally.

5.2.1 Feedback and Results

Each playtesting session and accompanying feedback section were recorded using an audio recorder, and the screen of the computer was recorded while the participant was in-game. After completing the playtesting portion of the experiment, we assisted the participants in taking off the headset and controllers and gave them time to re-acclimate to the surrounding environment before asking them the following questions to collect feedback.

• For each task (teleporting, picking up objects, understanding the seasons mechanic), did you find it was easy or difficult to complete?
• What aspects of the game controls did you like/dislike?
• Would you change how any of the game’s controls function?
• Did you experience motion sickness or any physical discomfort at any point during the experiment?

All of our playtesters said that the tasks themselves were easy to complete. And while for the most part, our playtesters found the controls to be intuitive and easy to pick up, one comment overwhelmingly received was to change the controls for picking up objects. The Vive has a set of side buttons on the left and right sides of each controller, and we had originally used these side grip-like buttons to pick up objects, thinking that it would more closely mimic picking up objects in real life. However, it seemed this was not the case, as many of our playtesters recommended we change picking up objects to the trigger buttons, located on the back of the Vive controller. None of the players felt any motion sickness in general; however, one player felt that the flowers and trees were rather jarring. Overall, our landscape and backgrounds were well-received, but many of the flowers and tree leaves were set to billboard (i.e. the flowers and leaves would rotate to face the player as the player rotated their head). Most playtesters did not notice this, however - or if they did, chose not to comment - and a portion of our team did not realize this as well, but once someone does notice the billboard effect, it can become very distracting.

In addition to the questions we asked, we received other feedback as well. One playtester noticed that something was off in regards to throwing objects; at some point in time, objects thrown using the controller began to head in a different direction from that which they were thrown in. Some players also expressed slight annoyance at needing to physically fully bend down to pick up objects, and requested that we extend the reach of the controllers. Another also expressed dislike of how it was possible to get unrealistically close to walls; there’s no real sensation of having a hitbox or body. Many players appreciated having a method of teleportation, but one suggested we add a segment of the game that requires players to physically walk from one place to another. Other feedback on elements we should add include: brightness settings,
minimap, a clock with time spent or time remaining, allowing players to teleport and hold an object with the same controller at the same time, and adding a sword somewhere.

5.3 Beta 1.0 Playtesting

The purpose of this playtesting phase was to test the functionality of the QuestManager and EventManager operations in the context of the tutorial quest, to test the complexity of the tutorial quest, to test the basic implementation of the crafting mechanic, and to incorporate any suggestions and improvements made on issues found during Alpha playtesting. In this prototype, the game world consisted of a completed village and seasons quadrants environment. Most placeholder objects were removed and additional artistic design details were incorporated. The objectives of this playtesting session were to ensure that the QuestManager and EventManager worked properly with each other in the context of a quest, that the players could easily understand the concept of crafting mechanic and test its functionality, and that players could understand the objectives of and complete the tutorial quest.

At the beginning of each playtesting session, we explained to the participant what tasks they would be completing and assisted them with putting on the headset and controllers. After allowing the playtester to become acclimated to the VR environment, we gave them the following verbal guidance to help them get started with completing the quest:

- Walk to the start of the quest location as indicated on the screen
- Following the instructions and clues given in game, attempt to solve the puzzle
- The investigators will answer any questions you may have about the game controls but will not give you clues or hints on how to solve the puzzle

5.3.1 Feedback and Results

Each playtesting session and accompanying feedback section were recorded using an audio recorder, and the screen of the computer was recorded while the participant was in-game. After completing the playtesting portion of the experiment, we assisted the participants in taking off the headset and controllers and gave them time to re-acclimate to the surrounding environment before asking them the following questions to collect feedback.

- Did you find the tutorial quest was easy or difficult to complete?
- What aspects of the tutorial quest did you like/dislike?
- Were you confused or unsure of how to complete the tutorial quest at any point?
- Were there any assumptions you had to make about some aspect in the game in order to complete the tutorial quest? If so, what assumptions did you make?
- Did you find the crafting mechanic easy or confusing to implement?
- Did you find the QuestManager UI elements easy or difficult to understand?
- Did you encounter any event in the game that you did not expect?
• Did you experience motion sickness or any physical discomfort at any point during the experiment?

All of the playtesters for the Beta 1.0 version of Shiki said that there was insufficient visual feedback as to whether or not quests had been completed, and additionally that it was almost impossible to determine what their next task was. All of the playtesters also liked the updated controls, with the trigger button on the Vive controllers used to pick up objects. However, several of the playtesters had trouble finding the Vive controllers’ side buttons, which are used to open the inventory. Some of the playtesters also were not aware of the river, since it is not visible from the entrance to the seasons.

Generally, the playtesters liked the season mechanic, and thought it was relatively intuitive to understand. They also thought the teleportation system worked well, and that there were no real unexpected results.

None of the playtesters experienced motion sickness or physical discomfort at any point during the playtest.
6. Postmortem

As our time at Osaka University came to an end, we sat down to evaluate what we accomplished and how we progressed on our project. We focused broadly on three areas of discussion: what went right during our project?, what went wrong during our project?, and what could we have done differently should we have the opportunity to complete a similar project in the future?

6.1 What Went Right?

Our team identified three overall positive aspects of our project. We had the opportunity to learn about and work with new software and hardware, we were able to obtain and produce high quality assets, and we were able to produce a working game prototype within our limited time.

Prior to this project, our team had some experience using Github to manage semi-large projects, had very little experience working with C# and had even less experience working with Unity. Most of us had ever worked with VR technologies. Despite our limited experience using these new software and hardware, we were able to create a Beta-level prototype for our game’s concept within the timeframe of our stay at Osaka University. A majority of our learning was completed while we were trying to implement different features of our game. Normally, learning as you complete a task tends to reduce the speed with which you can complete this task; however, we were fortunate to pick up on the programming paradigms of Unity for VR and C# rather quickly so that our progress was not hindered too much by our split focus.

Another aspect of our project that went well was asset production. Because our game was designed to be an experience game, the visual representation of our game world was extremely important in determining the feelings and experience our players would receive from playing our game. It is certainly easy to find assets on the Unity Asset Store and other online 3D model asset stores, but most did not come free. While searching for assets, our team expected to spend a decent amount of money to obtain assets that were in line with our vision for the world design. Luckily, we found a number of asset packs for a relatively cheap price, which let us design our game world in a timely fashion without having to spend a lot of money. In order to customize these assets for our game, we used convolutional neural network-based style blending on asset textures. This process was surprisingly easy, as we had anticipated needing to spend several hours training the neural network to produce desirable results. We were able to obtain realistic, “Kanō-ized” textures with only a few executions of the neural network.

About four weeks into our stay at Osaka University, we had a sudden, large shift in our project design and goals. Our game transitioned from a puzzle game about the four seasons to an experience game involving seasonal-related holiday tasks. We, essentially, started our project
from the beginning. We had to redefine our experience design, game objectives, game mechanics, and implementation practices, among other things. This late change in direction caused us to fall behind on our projected progress for the remaining time at Osaka University. Despite this late setback, we were able to produce and test more than one prototype that included two quests for the player to complete. There are still some pieces missing from our game that would enhance player experience; however, we have a game prototype that a player can play from start to finish without assistance.

6.2 What Went Wrong?

Our team identified five aspects of our work that went poorly. These aspects include our group communication skills, identifying project goals and design in a timely fashion, following our project outlines and plans, regularly documenting our work, and issues with Unity.

Perhaps our biggest problem in completing our project was clear communication. Often during brainstorming sessions, we would introduce our ideas on how to design a feature but these ideas were not adequately explained. Instead of asking for clarifications, we often made assumptions based on our own perceptions of what was said. It seemed at the time that we all understood and agreed on topics and approaches, but when it came time to implement them, we realized that we all assumed different things. This unclear communication led to misunderstandings and arguments that could have easily been avoided had we clearly expressed our ideas and asked each other for clarification when we did not fully understand each other.

Our second shortcoming was related to identifying our project goals and design. Each time our project shifted focus, we did not immediately identify crucial design aspects, particularly our experience design and target audience, slowing down our progress and implementation. Without a clear design goal, it was difficult to narrow down a specific design, and we ended up making our designs too general in the beginning. This problem was exasperated by our lack of effective communication, as mentioned earlier, which also prevented us from identifying the specific aspects and goals we hoped to achieve. As a result, we were not able to implement all of the holiday quests we had initially planned.

After realizing we needed to be more specific in defining and outlining our goals for our project, we sat down and created priority lists for tasks that needed to be completed. We sorted these lists by “musts,” “shoulds,” and “wants” and categorized them by which prototype we wanted them completed for. This organization strategy helped us clearly identify what work we needed to complete, and we integrated these lists into our Trello page. In the beginning we were diligent with following the Trello page to organize our task completion, but as the project went on we were not as strict with meeting our personal deadlines and organizing our lists in Trello as things had been completed. We did not strictly hold ourselves to the deadlines we set, and we hadn’t been regularly updating the Trello page, so when it came time to update it, we discovered tasks we had aimed to complete and ended up forgetting about. We fell behind in our projected
progress making up for these tasks and because we were not strict with deadlines, which was an issue that was easily avoidable had we been more diligent with our task organization.

In the beginning of our project, we were advised to keep a developers blog to keep track of our daily and weekly progress on our project. This blog had the benefit of documenting our project work in small portions so that we could see how we progressed over time and could easily review our progress towards the end of our project. We started out updating this blog regularly, but as time passed it fell to the side. We quickly went from posting several updates a week to one update every one to two weeks. While we could see the changes in our code for the project in our Github history, we could not see the changes in thought process and design of our game without regular updates to the developers blog. By not keeping up with the developers blog, it was hard to identify if we were truly meeting our smaller goals since we had no baseline for comparison.

The final drawbacks we experienced with our project were related to Unity. We chose to use Unity as some of our team members had minimal experience using it and we collectively had no experience using any other game engine. Unity also has a rather simple interface, which makes it a good game engine for first time game developers. We did find Unity very easy to learn and were able to create small tutorial projects of our own fairly quickly. As we added more to our project, Unity became difficult to work with on our machines. We ran into countless crashes, freezes, and errors, and the Unity documentation was not very helpful in resolving our issues. For our teammates using Unity on macOS, debugging was nearly impossible due to issues with compatibility with C#/MonoDevelop and Windows. These software-related issues took up a large portion of our working time and ultimately contributed to our project running behind our proposed schedule.

6.3 What Could Be Different?

Based on our project’s successes and shortcomings, we were able to identify practices that we would implement given another opportunity to work on this project. Our changes are related to communication, discipline in following goals and timelines, and taking initiative in learning and overcoming new material and challenges.

Before we began our project, we discussed broadly some strategies we wanted to implement in our project work. We discussed having quick daily meetings, much like used in Agile development, to identify our goals for the day and who would be responsible for completing which tasks. We did meet quite often in the beginning of our project and briefly discussed goals and assignments, but our meetings were not very structured and organized. We had identified, classified, and organized our priorities and project tasks, but were not strict on following up with these goals. In the future, we would implement these daily meetings with clear agendas and task assignments, which would ideally allow us to follow our priorities lists more strictly and hopefully improve our communication by increasing its frequency.
At the start of our project, we had scheduled on our calendar to do a weekly writing review. During these review meetings, we would read over parts of our final report and complete group edits on sections to make sure sections were accurate, up-to-date, and all sections followed the same styling and voicing. We did not end up having weekly writing review sessions, which would have greatly improved the quality of our writing earlier on. If given another chance to complete this project, we would certainly have regularly scheduled writing reviews. In addition to writing review sessions, we also did not keep up with writing posts to our developers blog. This hindered us towards the end of our project as it was harder to recall and track our progress throughout the project without regular, consistent documentation. We found that towards the end of our project, we were too immersed in our work to remember to write blog posts regularly. In the future, we would set calendar reminders to post regular updates two to three times a week per person.

Our final change that we would implement if we could work on the project in a formal setting again would be to take more structured time at the beginning of our project to learn how to use new software and hardware and become familiar with C#. We had completed a few Unity tutorials for the general software and VR applications, but had done these tutorials casually during our working time. If we were to repeat our project process, we would choose to spend a few days at the beginning of our project to thoroughly learn Unity, VR development, and C# inside and out so that we could better handle challenges and issues that arose later in our project with debugging software and hardware errors.
Appendix A: Game Design Document

Osaka University MQP Game Design Document

Name: 四季 (Shiki) - The Four Seasons

Tagline: A Seasonal Virtual Reality Game

Story:
Introduction - A certain small Edo village has experienced bountiful harvests in recent years. The villagers have been grateful for the prosperity but have begun to take it for granted. Instead of preserving the environment for its natural beauty and maintaining sustainability, the villagers take more than what they need and live greedily.

Plot - This village is known for having spectacularly vibrant seasons, as the village enshrines the four Kami of the seasons on rotation every quarter-year. In this fashion, the Kami are able to use their power to their full extent during their season, while resting and regaining that energy during the other 3 quarters of the year. During the seasons of doyō, or the 18 days before the start of each season when the vitality and life of nature is at its lowest, the villagers prepare the enshrinement ceremony to move the current Kami out and the new Kami in on the first day of the season.

The four seasonal Kami could not affect the world in the ways they did without the village, as each relied on the faith and belief of the villagers to exist. The more faith gathered from the villagers, the more vibrant the Kami could make the current season. To support this need of faith, the villagers would hold many celebrations throughout each season with the most focus on the double days, such as March Third, May Fifth, July Seventh, or September Ninth.

Everything in the village is fine until one year, the harvest and fertility Kami Ugajin decided to reward the villagers for their good faith in the Kami with a massively bountiful harvest. Contrary to what the Kami believed the villagers would do, the villagers become greedy and begin taking too much crop for what they need. With the sudden new bounty, the villagers lose track of the date, and during the doyō before Winter, forget to prepare the ceremony to move Kuraokami into its shrine. This forgetfulness angers the four seasonal Kami, and each take residence in a copy of the village’s shrine, keeping their seasonal influence out of the village and leaving it in a constant state of doyō.

The villagers do not seem to notice the unnatural halting of the seasons, and carry on their daily lives as if nothing has changed. As doyō becomes stronger in the village, the vitality and life energy of each individual villager begins to dwindle. Eventually, each individual villager
runs out of their life energy and vitality, freezing in place as a lifeless husk in whatever pose they were just in. This scares the Kami, as their protest has led the village to become lifeless, and without the faith from the villagers, they cannot spread their influence enough to undo their punishment. In a desperate plea for life, the Kami call in the player, a wandering spirit. To bring life back into the village, the player must gather supplies from the landscape around the village to hold the celebrations for the Double Day Festivals. The celebrations will hopefully restore enough vitality to the village to have the villagers begin celebrating and sending their faith to the Kami.

**Game Mechanics:**
Most puzzles in *Shiki* involve the changing of nature over the four seasons. The area around the village is composed of the same slice of terrain copied and rotated 90 degrees four times. Each slice is represented in a different season, however, and changes made to the environment in one slice affect the other three slices. This opens up many opportunities for keeping puzzles interesting, as most puzzles will require the player to travel to a different season to solve the puzzle.

An object may have multiple variants, with each variant mapping to a season. When an object in variant A is placed in season B, the object will remain in variant A. However, if the player travels to season C, the object there will be in variant C. This way, the player can always go back to the season in which they placed an object if they want the object in the same variant it was in when they dropped it. We decided to go with this mechanic, as opposed to one where an object immediately changes to the variant associated with the season it’s in as soon as it’s dropped, as we felt that having objects immediately change would feel unnatural. Another mechanic that we rejected is one in which a dropped object would change its variant only when the player left the season that the object was dropped in. We rejected this mechanic because we thought that it added tedium to the gameplay, and detracted from the idea that objects change because of the change between seasons.

The game will save automatically as the player solves puzzles. It was decided that a manual save feature would not be necessary due to the lack of a lose condition.

The player can interact with the world by…
- Picking up items and storing them in his “bag”
- Retrieving items from his “bag” and placing them somewhere
- Praying at shrines (asking Kami for help?)
- Reading notes or scrolls

**Puzzles:**
The game’s primary quest involves 5 Japanese holidays throughout the year:
Jan 7: Combination of New Years (Jan 1 - Jan 3) and the Festival of Seven Herbs (Nanakusa no sekku)
Mar 3: Girl’s Day (hina matsuri)
May 5: Children’s Day (kodomo no hi)
Jul 7: Star Festival (Tanabata)
Aug 9: Chrysanthemum Day

**Character Design:**
This game takes place in an Edo Period village. All characters will be wearing dress reflective of their roles in the village as was traditional for the Edo Period. Below is a picture of typical fabric patterns/designs worn by commoners during the Edo Period.

![Figure 34: Edo Period clothing patterns](image)

[Source: Edo Museum, photographed by Larisa]
Main Character - The main character (currently ambiguous gender) is a wandering spirit that was called in by the Kami with the remnants of their energy. Because the game is played in a 1st-person perspective, there will not be a full main character model. The main character may be reflected in surfaces in the game, such as a mirror, shiny metal, or water.

Villagers - The villagers are background characters, and as such, don’t have the same importance as the main character. The villagers will wear clothes reflective of different working roles in the village as shown below.

![Villagers](image)

Figure 35: Various commoner dress (a) Commoner in Kosode of the early Edo Period (b) Commer in kamiko haori (c) Commoner wearing a hikimawashi-kappa (d) Young married commoner in kosode (Costume history in Japan).

Due to the limitations to our team in regards to art creation, we plan to use 4 to 5 character models and apply minor adjustments to each model to simulate a larger number of villagers.

Ugajin - Ugajin is the Kami of the harvest and fertility and is represented as both male and female in Buddhist mythology. In this game, Ugajin will be represented in its female form.
Ugajin is typically displayed as a snake with the head of a human, as shown below in a sculpture. In this game Ugajin will preside over the Autumn portion of the world map.

Figure 36: Ugajin’s feminine form (Ugajin’s feminine form).

Konohana Sakuya-hime - The Princess of Spring and volcanoes, Konohana Sakuya-hime is the goddess that represents everything about the springtime in Japan, particularly the blooming of sakura trees. She is regarded as highly beautiful. As such, Konohana Sakuya-hime will preside over the Spring portion of the world map. Picture below is a woodblock print made in the Edo Period by Katsushika Hokusai.

Figure 37: Goddess Konohana Sakuyahime (Katsushika Hokusai, 1834).
Kuraokami - Kuraokami is the Kami of Snow, Rain, and Darkness. It is often represented as a white or blue dragon born from the blood/body of the fire deity Kagu-tsuchi. It typically resides in mountainous or valley areas. In this game, Kuraokami will preside over the Winter region.

Susanoo - Susanoo is the Trickster God of the Summer, Sea, and Storms. He is often represented as a human-like warrior with a long sword. In the game, Susanoo will preside over the Summer region.

Figure 38: Dragon rising up to Heaven (Ogata Gekko, 1897).
**Art Design:**
At the beginning of the game, the sector of land under each Kami’s control is highly desaturated, reflecting the Kami’s punishment of the main character. As the player solves puzzles in a particular sector, that sector begins to regain some of its color. Additionally, solving puzzles causes the rest of the village to regain some of its color, excluding the frozen villagers.

The world consists of a village, surrounded by wilderness. The map is separated into 5 zones, one zone for each season and a “neutral” area in the center of the village. Each of the season areas has a small shrine, at which that season's Kami resides. Each season extends into a portion of the village.

The “cutscenes” at the start and end of the game will be designed with a 2D “stop-motion” animation in mind, similar to the visual style of the [Borderlands 2 introduction video](http://www.example.com). These cutscenes will be shown on a flat surface, such as a scroll or Japanese folding screens (屏風, byoubu).

**Audio Design:**
Each season area has a different set of environmental sounds associated with that season in Japan. At the beginning of the game, when the player has not solved a season’s particular puzzle, the season area will be most silent. As the player completes the puzzles in a season, ambient sounds begin to return to that season. Independent of each season area, there is ambient background music. This music is inspired by and will take characteristics from the “Chimes of Kyoto.”

Audio recordings will be created and edited using sound editing software, including - but not limited to - Reaper, Noteflight, and Adobe® Audition®. Audio will be scripted to game objects, actions, and events using Unity’s built-in audio scripting tools and Wwise audio scripting software.

**Examples of Seasonal/Locational Sounds**

**Village (Limbo) -** Looping of extended ethereal, ambient wind noises mixed with “Chimes of Kyoto” sounds, and minimalistic introduction of Edo Period Japanese instrument sounds (Koto, Shakuhachi, Tsuzumi, Kokyuu, etc…)

**Winter -** Sound of crunching snow when player walks on ground, hollow wind sounds, Wintertime birds (Tundra Swan, Red-neck Grebe, etc…)

**Spring -** Babbling brook sounds, Springtime birds (Pacific Golden Plover, Japanese Sparrowhawk, etc…)

**Summer -** Cicadas, Summertime birds (Oriental cuckoo, Jungle Nightjar, etc…)

**Autumn -** Rustling leaves, Autumn birds (White-throated needletail, Pacific Swift, etc…)

**Technological Design/UI:**

The game is designed to be played through the HTC Vive VR headset, with the user sitting or standing, but not moving their feet. Due to the Vive’s system requirements, the game will be targeted towards desktop computers.

Typical interactions with the environment might be picking up and moving in-game objects, triggering dialog with NPCs, and opening scrolls for reading.

We will test whether or not allowing movement using both the Vive controllers and the Vive’s Room Scale VR system at the same time works well. We believe that the Room Scale VR system allows for more natural interactions with the virtual environment. If our tests show that doing so does not cause motion sickness and feels natural, we would prefer to use both the touchpad on one of the Vive controllers, and the Room Scale VR system, to allow the user to both move large distances, while allowing for more natural movement at shorter range.
The game will be developed using the Unity 5 game engine. This engine was chosen since it has good support for the HTC Vive, and since our team is more familiar with it than other game engines we considered (Unreal Engine 4, CryEngine, …). Additionally, Unity is currently beta-testing a built-in collaboration tool that our team will be able to use for free for the duration of this project.

It is expected that many of the visual effects (i.e. desaturation of color in an area depending on the number of puzzles solved in that area, as well as effects depending on the season) will be implemented using custom shader programs. These shaders will have to be highly optimized to ensure that overall render performance is sufficient to ensure motion-sickness-free gameplay.

**Responsibilities:**

**Drew -**
- Shader programming
- 3D modeling
- Character modeling
- Repo master (+ bug/task tracking)

**Joe -**
- Dialogue/script writing
- Cut-scene writing
- Event programming
- Character modeling

**Larisa -**
- 2D art
- General object behavior programming
- Landscape design
- Quality assurance/find bugs

**Rachel -**
- Audio asset creation
- Audio scripting
- Physics programming
- 2D art
## Timeline:

<table>
<thead>
<tr>
<th>August</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Sunday</td>
<td>Monday</td>
<td>Tuesday</td>
<td>Wednesday</td>
<td>Thursday</td>
<td>Friday</td>
<td>Saturday</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Create Unity project, Begin world, audio, and character asset creation</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>Begin writing controller testbed and structuring</td>
<td>Final paper, finish puzzle development</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>Begin basic audio scripting</td>
<td>10</td>
<td>11</td>
<td>Finish controller testbed and run tests</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>Finish audio asset creation</td>
<td>15</td>
<td>Weekly Final Paper Review</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>begin custom shader work and alpha playtesting, finish environment audio scripting</td>
<td>22</td>
<td>Weekly Final Paper Review</td>
<td>23</td>
<td>Finish world modeling</td>
</tr>
<tr>
<td>27</td>
<td>28</td>
<td>Finish 2D art asset creations</td>
<td>29</td>
<td>Weekly Final Paper Review</td>
<td>30</td>
<td>1 (September)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Finish custom shader work, Beta phase of game completed Methodology Section of Paper Completed</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>September</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Sunday</td>
<td>Monday</td>
<td>Tuesday</td>
<td>Wednesday</td>
<td>Thursday</td>
<td>Friday</td>
<td>Saturday</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>Weekly Final Paper Review</td>
<td>6</td>
<td>7</td>
<td>Finish Beta playtesting</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>Weekly Final Paper Review</td>
<td>13</td>
<td>14</td>
<td>Beta 2.0 phase of game completed THEORETICAL FINISH</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>19</td>
<td>Weekly Final Paper Review</td>
<td>20</td>
<td>21</td>
<td>Finish Beta 2.0 playtesting</td>
</tr>
<tr>
<td>1 (October)</td>
<td>2</td>
<td>3</td>
<td>Weekly Final Paper Review</td>
<td>4</td>
<td>5</td>
<td>Weekly Final Paper Review</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Submit eCDR Return to America</td>
</tr>
</tbody>
</table>
# Appendix B: List of Assets

## Asset store

<table>
<thead>
<tr>
<th>Asset Name</th>
<th>Author</th>
<th>Type</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free Bucket</td>
<td>Game-Ready</td>
<td>Models</td>
<td><a href="https://assetstore.unity.com/packages/3d/free-bucket-24611">https://assetstore.unity.com/packages/3d/free-bucket-24611</a></td>
</tr>
<tr>
<td>Garden Realistic Tools</td>
<td>GBAndrewGB</td>
<td>Models</td>
<td><a href="https://www.assetstore.unity3d.com/en#!/content/68960">https://www.assetstore.unity3d.com/en#!/content/68960</a></td>
</tr>
</tbody>
</table>

## Outside of asset store

<table>
<thead>
<tr>
<th>Asset Name</th>
<th>Author</th>
<th>Type</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unity-VRInputModule</td>
<td>wacki</td>
<td>Scripts</td>
<td><a href="https://github.com/wacki/Unity-VRInputModule">https://github.com/wacki/Unity-VRInputModule</a></td>
</tr>
<tr>
<td>Unity Wireframe Shaders</td>
<td>Chaser324</td>
<td>Shaders</td>
<td><a href="https://github.com/Chaser324/unity-wireframe">https://github.com/Chaser324/unity-wireframe</a></td>
</tr>
<tr>
<td>Nett</td>
<td>paiden</td>
<td>Scripts</td>
<td><a href="https://github.com/paiden/Nett">https://github.com/paiden/Nett</a></td>
</tr>
<tr>
<td>--------------</td>
<td>--------</td>
<td>---------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>Neural-Style Style Transfer</td>
<td>Justin Johnson</td>
<td>External Script</td>
<td><a href="https://github.com/jcjohnson/neural-style">https://github.com/jcjohnson/neural-style</a></td>
</tr>
</tbody>
</table>
Appendix C: Playtesting Procedures

The following playtesting procedures are as followed from our IRB application. Instructions and questions that are underlined indicate the steps we ended up using during our playtesting sessions. Those steps that are not underlined were omitted from testing.

Study Protocol
Our study will be conducted in three phases, consisting of three separate digital prototypes of our game. These prototypes are denoted as Alpha, Beta 1.0, and Beta 2.0. Each phase of our study will be conducted using the HTC Vive VR headset and handheld controllers.

The first phase of our study is intended to test the Alpha version of our prototype. The second phase of our study is intended to test the Beta 1.0 version of our prototype, which will include changes made to the Alpha prototype as a result of Alpha testing. The final phase of our study is intended to test the Beta 2.0 version of our prototype, which will include changes made to the Beta 1.0 prototype as a result of Beta 1.0 testing.

Prototype Testing Phase 1: Alpha
The purpose of this phase is to test the functionality of our game mechanics and controls in a digital environment built using the Unity 5 game engine for the HTC Vive Virtual Reality platform. This digital environment consists of the basic game world map and placeholder game objects as they are intended to be placed in the game world for our final version. These placeholder game objects will be represented by basic geometric solids and will be labeled appropriately so that participants can identify what each object is supposed to be.

The game mechanics and controls to be tested in this prototype are player movement throughout the game world, the player’s ability to interact with objects in the game (picking up objects, placing objects, opening/closing doors), and the seasonal masking effect on game objects.

Procedure:

1. Introduce the participant to the purpose of this playtesting experiment. Explain the goals and objectives of the tasks they will complete if they choose to participate. Introduce the platform on which they will be testing the game, and explain any and all risks associated with using this platform. Inform the participant that their voice will be recorded during testing and during feedback collection after testing, and that their actions in the game will be recorded using screen recording software on the testing computer. Obtain written informed consent using the Informed Consent Form before beginning the testing portion of the procedure. (Approx. 10 minutes)

2. Instruct the participant on how to use the HTC Vive Virtual Reality headset and handheld controllers. Introduce the tasks that the participant will attempt to complete during the testing. (Approx. 2 minutes)
3. Assist the participant in putting on the HTC Vive VR headset and handheld controllers and ensure that the participant stays within the confines of the HTC Vive Play Room. Investigators should position themselves on the borders of the Play Room to help safely guide the participant in case the participant: steps outside of the Play Room boundaries, is at risk of tripping or falling, and/or experiences discomfort or disorientation and needs to terminate the testing procedure. Allow the participant as much time as needed to acclimate to using the HTC Vive equipment. *(Estimated 5 minutes)*

4. When the participant is ready, instruct the participant to complete the following tasks:
   - Using the handheld controller, teleport to another location in the game world
   - Locate the white cube; pick up the cube, then navigate to a “Seasons” quadrant and place the item on the ground; observe the change in appearance to this item; navigate to the other “Seasons” quadrants and observe the changes in appearance to the corresponding items
   - Teleport to the “Summer” quadrant. Pick up the green cube and place it in another “Seasons” quadrant; observe the change in appearance to this item

The allotted time for these tasks is 10 minutes. Participants may end up using less time to complete the tasks, but no participant will be allowed to continue playtesting after 10 minutes. This is to minimize the risk of physical discomfort or injury while using the HTC Vive. *(Approx. 10 minutes)*

5. Assist the participant in taking off the HTC Vive VR headset and handheld controllers and provide them a chair to sit in while they re-acclimate to the surrounding environment. Allow the participant as much time as needed to re-acclimate. Ensure that the participant is ready and willing to move onto the next portion of the testing procedure before continuing. *(Estimated 5 minutes)*

6. Ask the participant the following questions:
   a. On a scale of 1 to 5 with 1 being “Very Difficult” and 5 being “Very Easy,” how easy was it to complete each task?
   b. What aspects of the game controls did you like/dislike?
   c. Were you confused or unsure of how to complete a task at any point?
   d. Would you change how any of the game’s controls function?
   e. Did you experience motion sickness or any physical discomfort at any point during the experiment?

The participant’s responses to these questions will be recorded by audio and transcribed later for analysis. Thank the participant for their cooperation and ensure that they are not experiencing any residual negative physical effects, if at all, associated with using the HTC Vive. *(Approx. 10 minutes)*

**Prototype Testing Phase 2: Beta 1.0**
The purpose of this phase is to test the functionality of QuestManager operations and function of the puzzle in the digital game environment, as well as any improvements on issues found during Alpha playtesting. This digital environment consists of the finished world map and game objects as they are intended to be placed in the game world for our final version. Most of the game objects placeholders found in the Alpha prototype will have been replaced by their final version counterparts.

The functions of the QuestManager code to be tested in this prototype include initiating a quest, completing steps necessary to solve a puzzle, and verifying that a quest has been complete.

Procedure:

1. Introduce the participant to the purpose of this playtesting experiment. Explain the goals and objectives of the tasks they will complete if they choose to participate. Introduce the platform on which they will be testing the game, and explain any and all risks associated with using this platform. Inform the participant that their voice will be recorded during testing and during feedback collection after testing, and that their actions in the game will be recorded using screen recording software on the testing computer. Obtain written informed consent using the Informed Consent Form before beginning the testing portion of the procedure. *(Approx. 10 minutes)*

2. Instruct the participant on how to use the HTC Vive Virtual Reality headset and handheld controllers. Introduce the tasks that the participant will attempt to complete during the testing. *(Approx. 2 minutes)*

3. Assist the participant in putting on the HTC Vive VR headset and handheld controllers and ensure that the participant stays within the confines of the HTC Vive Play Room. Investigators should position themselves on the borders of the Play Room to help safely guide the participant in case the participant: steps outside of the Play Room boundaries, is at risk of tripping or falling, and/or experiences discomfort or disorientation and needs to terminate the testing procedure. Allow the participant as much time as needed to acclimate to using the HTC Vive equipment. *(Estimated 5 minutes)*

4. When the participant is ready, instruct the participant to complete the following tasks:
   - Walk to the start of the quest location as indicated on the screen
   - Following the instructions and clues given in game, attempt to solve the puzzle
   - The investigators will answer any questions you may have about the game controls but will not give you clues or hints on how to solve the puzzle

   The allotted time for these tasks is 10 minutes. Participants may end up using less time to complete the tasks, but no participant will be allowed to continue playtesting after 10 minutes. This is to minimize the risk of physical discomfort or injury while using the HTC Vive. *(Approx. 10 minutes)*

5. Assist the participant in taking off the HTC Vive VR headset and handheld controllers and provide them a chair to sit in while they re-acclimate to the surrounding environment.
Allow the participant as much time as needed to re-acclimate. Ensure that the participant is ready and willing to move onto the next portion of the testing procedure before continuing. *(Estimated 5 minutes)*

6. Ask the participant the following questions:
   a. On a scale of 1 to 5 with 1 being “Very Difficult” and 5 being “Very Easy,” how easy was it to solve the puzzle?
   b. What aspects of the puzzle did you like/dislike?
   c. Were you confused or unsure of how to proceed solving the puzzle at any point?
   d. Were there any assumptions you had to make about some aspect in the game in order to solve the puzzle? If so, what assumptions did you make?
   e. Did you find the QuestManager interface easy or confusing to interpret?
   f. Did you encounter any event in the game that you did not expect?
   g. Did you experience motion sickness or any physical discomfort at any point during the experiment?

The participant’s responses to these questions will be recorded by audio and transcribed later for analysis. Thank the participant for their cooperation and ensure that they are not experiencing any residual negative physical effects, if at all, associated with using the HTC Vive. *(Approx. 10 minutes)*

**Prototype Testing Phase 3: Beta 2.0**

The purpose of this phase is to test the user experience design of the game and to test any improvements on issues found during the Alpha and Beta 1.0 playtesting sessions. The game is intended to model an experience that is “relaxing, ethereal, and pensive” and should evoke emotions relating to these characteristics in the player. The digital environment of this prototype will consist of the final layout of the game world map, game objects, and all puzzles written for the game. All remaining placeholder objects in the game will be replaced with their final versions.

Procedure:

1. Introduce the participant to the purpose of this playtesting experiment. Explain the goals and objectives of the tasks they will complete if they choose to participate. Introduce the platform on which they will be testing the game, and explain any and all risks associated with using this platform. Inform the participant that their voice will be recorded during testing and during feedback collection after testing, and that their actions in the game will be recorded using screen recording software on the testing computer. Obtain written informed consent using the Informed Consent Form before beginning the testing portion of the procedure. *(Approx. 10 minutes)*

2. Instruct the participant on how to use the HTC Vive Virtual Reality headset and handheld controllers. Introduce the tasks that the participant will attempt to complete during the testing. *(Approx. 2 minutes)*
3. Assist the participant in putting on the HTC Vive VR headset and handheld controllers and ensure that the participant stays within the confines of the HTC Vive Play Room. Investigators should position themselves on the borders of the Play Room to help safely guide the participant in case the participant: steps outside of the Play Room boundaries, is at risk of tripping or falling, and/or experiences discomfort or disorientation and needs to terminate the testing procedure. Allow the participant as much time as needed to acclimate to using the HTC Vive equipment. *(Estimated 5 minutes)*

4. When the participant is ready, instruct the participant to complete the following tasks:
   - Walk to the start of the quest location as indicated on the screen
   - Following the instructions and clues given in game, attempt to solve the puzzle
   - The investigators will answer any questions you may have about the game controls but will not give you clues or hints on how to solve the puzzle

   The allotted time for these tasks is 10 minutes. Participants may end up using less time to complete the tasks, but no participant will be allowed to continue playtesting after 10 minutes. This is to minimize the risk of physical discomfort or injury while using the HTC Vive. *(Approx. 10 minutes)*

5. Assist the participant in taking off the HTC Vive VR headset and handheld controllers and provide them a chair to sit in while they re-acclimate to the surrounding environment. Allow the participant as much time as needed to re-acclimate. Ensure that the participant is ready and willing to move onto the next portion of the testing procedure before continuing. *(Estimated 5 minutes)*

6. Ask the participant the following questions:
   a. On a scale of 1 to 5 with 1 being “Very Difficult” and 5 being “Very Easy,” how easy was it to solve the puzzle?
   b. What aspects of the puzzle did you like/dislike?
   c. Were you confused or unsure of how to proceed solving the puzzle at any point?
   d. Were there any assumptions you had to make about some aspect in the game in order to solve the puzzle? If so, what assumptions did you make?
   e. Did you encounter any event in the game that you did not expect?
   f. Did you experience motion sickness or any physical discomfort at any point during the experiment?
   g. What emotions did the following aspects of the game evoke in you?
      i. Game atmosphere
      ii. Soundscape
      iii. Visuals
      iv. Objectives
      v. Story
   h. What experience do you think the game was trying to make you feel?
The participant’s responses to these questions will be recorded by audio and transcribed later for analysis. Thank the participant for their cooperation and ensure that they are not experiencing any residual negative physical effects, if at all, associated with using the HTC Vive. (Approx. 15 minutes)
Appendix D: Playtesting Feedback

Alpha Notes

Questions:
1. Did you find it easy or difficult to complete each instruction (teleporting, picking up object)?
2. What aspects of the game controls did you like/dislike?
3. Would you change how any of the game’s controls function?
4. Did you experience motion sickness or any physical discomfort at any point during the experiment?

Person 1
1. picking up objects - choose another button
   liked the trigger button for picking up things
2. liked being able to see the summer winter etc
   if there were some animals or birds walking around that’d be better
   things moving
3. none in particular
4. no motion sickness but not really getting motion sick anyways

Person 2:
1. thought trigger button should’ve been the thing to pick up things instead of the side buttons.
2. didn’t dislike it but the picking up thing again ^
3. nothing in particular
4. nothing in particular

Person 3:
1. thought it was very intuitive and easy to understand
2. thought it was veyr fun, even though they only did it for a little bit. Didn’t hate anything yay
3. they didn’t like the atari-han hitting the walls and buildings and stuff
4. didn’t feel any.

Person 4:
1. the side buttons for picking up things is a bit difficult
   being able to choose the teleportation distance was nice
2. the flowers were pretty and mountains were fun to climb
3. none in particular
4. nothing

Person 5:
1. thought it was quite easy
2. grabbing things was difficult: had to go deep down inside the object
   rabbit rescue(?) was easy to grab the rabbits once you brought it close enough
   trigger button was more intuitive for grabbing
   touch pad was easy for teleporting
3. picking up things —> trigger button
4. not at all
   thought it was fun

Person 6:
1. pretty easy
2. teleporting with both hands was good; preferred picking up with the trigger; didn’t like having to bend down with the controllers (add more reach)
   extend the controller model + collider
3. other than trigger: fix trajectory on throwing; add reach to grab; add some interaction that requires people to move around (since they can just go around teleporting only)
   add some sort of control thing on their arm that they don’t see when walking but if they move the controller in front of them they’ll see it
   minimap; time remaining; current quest
   also explain to people about the cord (controller in one hand, cord in another)
   right-handed people will turn clockwise naturally
4. slight motion sickness while standing in the middle of flowers (or with a tree) and moving back and forth because it looked like it was moving with the display - shouldn’t be happening
   maybe a bit too bright-colored (allow them to adjust brightness before playing maybe?)

thought it was jarring that the flowers and stuff move with your head
recommended moving them at a fraction of the head speed
   additionally stop the wind (or maybe make it weaker?)
maybe add a mini map to one of the controllers
wants a sword somewhere in the game (katana)

**for summer**: turn up the light/reflections (it’s very sunny)
  - add heat effects (looking at concrete makes it wavy)
  - shadows should be stronger
  - could just write summer on the gate
  - add wisteria
  - fish in the river..?

get rid of the wind in winter if we want it to be relaxing
  - also make the trees stop moving as much (particularly the branches)
  - add more snowfall

change the structure of the gate for each season
  - particularly the floor
  - but yeah no change everything slightly

To Persons 2-3: forgot to explain to them that the cube/items disappeared

**THINGS TO DO:**
make it so we can teleport while holding things with the same hand
load things like cubes and stuff after the ground forms
  - we had no cube at the start…

**Beta 1.0 Notes**

Beta 1.0 Playtesting Notes

Participant #1

- Easy, but not sure what the task was each time
- Give prompts in game
- Explain the controls beforehand
- Liked teleportation, felt intuitive
- Didn’t know what to expect, no previous expectations or experience
- Didn’t experience motion sickness
- Didn’t think the tree looked bad
- Did not see the river
Participant #2

- Computer crashed mid-test, had to restart
- Medium difficulty
- Liked season concept, change in one changes in others
- It would be nice to do things out of seasons orders
  - If you didn’t take care of the tree during blight, it would have a different final characteristics
  - Then that would give someone a hint that they need to do things to stop the blight
- Unsure of if events were completed, couldn’t tell that the tree was water
  - More feedback
- Not really unexpected results
- No motion sickness
- No physical discomfort looking at tree
- Hint to don’t forget to care for the tree
- Didn’t know the river existed when first entering spring, needed to teleport

Participant #3

- Branch looks like original sapling
- “Where is my minimap that I asked for?”
- Pretty easy
- Changing of the tree in the seasons is cool, like how it grows every time
- Would be nice to have more information - didn’t feel like I was chopping, maybe haptic feedback
- Waypoint system or some sort of navigation system rather than “there’s a blue thing I’ll walk over to it”
- Didn’t really feel like it was a puzzle
- In lieu of explanation, put hints, like a picture of someone chopping down a tree
- Not confused after explanation of of goal
- Did not expect stone statues of children, thought there would be interaction, seems like there’re frozen in time
- Another way to give hints - people can’t fail, they just can’t succeed if you don’t do it right
- Children could be different sizes - give a hint with the tutorial quest
- Very slight eye fatigue, much less than last time
- Didn’t expect to see the tree be black/void like that but it wasn’t a negative physical experience
Appendix E: Known Issues and Bugs

<table>
<thead>
<tr>
<th>Title</th>
<th>Priority</th>
<th>Labels</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add in dialog/monolog/some sort of hint to players on what to do</td>
<td>Critical</td>
<td>Status: Available</td>
<td>Type: Enhancement</td>
</tr>
<tr>
<td>Implement Chrysanthemum Day Quest</td>
<td>High</td>
<td>Status: Available</td>
<td>Type: Enhancement</td>
</tr>
<tr>
<td>Hard to see growing yorishiro tree in spring</td>
<td>Medium</td>
<td>Playability Improvements</td>
<td>Status: Available</td>
</tr>
<tr>
<td>Show better that the hole in winter is not big enough to plant seeds into</td>
<td>Medium</td>
<td>Status: Available</td>
<td>Type: Enhancement</td>
</tr>
<tr>
<td>Enable arbitrary strings with Play Dialog</td>
<td>Medium</td>
<td>Type: Enhancement</td>
<td></td>
</tr>
<tr>
<td>Make small changes to branch as it is being merged with tools</td>
<td>Medium</td>
<td>Status: Available</td>
<td>Type: Enhancement</td>
</tr>
<tr>
<td>Add smoke to all tool objects after picking up the branch</td>
<td>Medium</td>
<td>Status: Available</td>
<td>Type: Enhancement</td>
</tr>
<tr>
<td>Do something about objects player is not ready to interact with</td>
<td>Medium</td>
<td>Status: Available</td>
<td>Type: Enhancement</td>
</tr>
<tr>
<td>Mechanism to prevent player from leaving village at very start</td>
<td>Medium</td>
<td>Status: Available</td>
<td>Type: Enhancement</td>
</tr>
<tr>
<td>Objects affected by gravity in the village jump when player teleports</td>
<td>Medium</td>
<td>Component: Teleportation</td>
<td>Status: Available</td>
</tr>
<tr>
<td>Allow teleportation while holding object</td>
<td>Medium</td>
<td>Status: Available</td>
<td>Type: Enhancement</td>
</tr>
<tr>
<td>Make it possible to replace an object that is being held without dropping it</td>
<td>Low</td>
<td>Playability Improvements</td>
<td>Status: Available</td>
</tr>
<tr>
<td>Inventory targets block teleport</td>
<td>Low</td>
<td>Status:</td>
<td>Type: Bug</td>
</tr>
<tr>
<td>beam</td>
<td>Available</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-----------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Make it easier to chain OnCompletes</td>
<td>Low</td>
<td>Type: Enhancement</td>
<td></td>
</tr>
<tr>
<td>Make filling up bucket more accurate to real life</td>
<td>Low</td>
<td>Status: Available</td>
<td>Type: Enhancement</td>
</tr>
</tbody>
</table>
### Appendix F: Task List

At the beginning of the MQP, after we developed our basic idea, we developed a list of features that could be added to the game, and assigned them rankings of “Must”, “Should”, and “Want”. “Must” items are features essential to the game functioning at all, “Should” items are nonessential features that would add a lot to the game and make it more presentable, and “Want” items are features that are even less essential than “Should”. This list of features helped guide us throughout the project. (The list is available below.)

To keep track of the tasks that needed to be completed for each of our release stages, and to keep track of what each team member was working on, we used the Trello webapp. Trello is based off of lists, which can contain cards. Cards can be assigned due dates, be assigned to team members, and contain checklists.

![Trello screenshot](https://example.com/trello-screenshot.png)
<table>
<thead>
<tr>
<th>Must</th>
<th>Should</th>
<th>Want</th>
</tr>
</thead>
<tbody>
<tr>
<td>(we fail without these)</td>
<td>(a little polish)</td>
<td>(would be really cool if we have time)</td>
</tr>
<tr>
<td>Comfortable VR experience (low potential for VR sickness)</td>
<td>Log-tool that players can keep track of puzzles with</td>
<td>An epilogue</td>
</tr>
<tr>
<td>Well designed, modular software architecture</td>
<td>Players naturally run into puzzles (no guided &quot;god arrow&quot; telling them where to go)</td>
<td>Accurate sun position based on season</td>
</tr>
<tr>
<td>Research Non-teleportation movement</td>
<td>Players see current season explicitly</td>
<td>Novel user interface</td>
</tr>
<tr>
<td>2 fully functional puzzles</td>
<td>Visual mixing at boundaries between seasons</td>
<td>Some sort of neural-network or machine-learning system.</td>
</tr>
<tr>
<td>All puzzles have 2 or more solutions</td>
<td>Particle effects</td>
<td>Day/night system</td>
</tr>
<tr>
<td>Functional interface</td>
<td>Nice shaders</td>
<td>Visually appealing landscaping in each season</td>
</tr>
<tr>
<td>Tutorial level</td>
<td>Checkpoint system (automatic saving)</td>
<td>Method to travel between opposite seasons without going through the center</td>
</tr>
<tr>
<td>Way to represent objects existing in different seasons with different states (gameobject family and relative coordinate system)</td>
<td>Game world doesn't feel small to the player</td>
<td>Winter at Nighttime</td>
</tr>
<tr>
<td>Players should have a way of knowing the current season implicitly</td>
<td>High quality art</td>
<td>Fully modeled deities</td>
</tr>
<tr>
<td>Central village</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Possible to get to seasons from the center</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Possible to travel directly between seasons</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Know which direction is towards the center</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pause method / manual save point</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Puzzles that involve all 4/5 sections of the world</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiple puzzles can be done at the same time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tasks inside of a quest can be completed without knowledge of the quest beforehand</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bag/Inventory system</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Stuff to keep in mind</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No game mechanic can exist without some story element behind it</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interaction with gods is consistent with mythology</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Areas that are fenced off to players make sense to be fenced off</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-teleportation movement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Well written story</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prologue/introduction</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix G: Language Specifications

The following text comes from the “Trigger and OnComplete Language.txt” file, describing the specifications in writing trigger and onComplete strings.

The language follows English sentence order, however the words must be limited to:
actions: Get, Enter, Drop, Cut, Hit, Dig, Grind, Merge, Become, PickUp, Store, Retrieve, Leave, Play, Delete, Show, Hide
prepositions: On, With, And
object types: Player, Item __, Location __

Note: Get and Become are both actions, but they are to be used in onComplete functions.
(However, Get can be used in trigger functions.)
The language can be edited to add more features.

At the moment, a sentence can have up to two objects (other than the player) or a location, and must contain an action.
Objects include all named objects in the game, and must follow the form “Item ObjectName” where ObjectName is the name of the object.

Actions fall into three categories: ones where two objects must be involved (with a word describing their interaction order); ones involving only one action; or one with a location only.
The current list of actions are “Get, Enter, Drop, Cut, Hit, Dig, Merge, Become, Open, Play”.

- Get - Player receives/picks up an item
  [Not Implemented] can be used to spawn an item in another location
  Ex: “Player Get Item PoundedRice”

- Enter - Player enters a particular area (must be specified by name)
  Ex: “Player Enter Winter”

- Drop - Player must cause an item to fall onto another item
  Ex: “Player Drop Item RiceInBowl On Item ActiveFire”

- Cut - Player must cause a cutting motion with one object on another
  Ex: “Player Cut Item Bread With Item Knife”

- Hit - Player must cause a hitting motion (e.g. with a hammer)
  Ex: "Player Hit Item SteamedRice With Item Hammer"

- Dig - Player must cause a digging motion (e.g. with shovel)
  Ex: “Player Dig Location RiceField With Item Shovel”
Merge - Player must bring two items close together, while holding both of them
   Ex: “Player Merge Item MultiTool And Item Shovel”
Become - One item transforms into another (to be used in OnComplete)
   Ex: “Item Rice Become Item CookedRice”
Open - Player opens something (at the moment primarily meant for inventory)
   Ex: "Player Open Item Inventory"
Play - UI plays an action (to be used in OnComplete)
   Ex: "UI Play Sound SoundName" or "UI Play Dialog DialogName"
Leave - Allows the player to leave an item in one season, to pick it up X seasons later
   Ex: "Player Leave Item ItemName For 2 Seasons"
Delete - An item deletes itself.
   Ex: “Delete Item ItemName”
Show - An item becomes visible and can be interacted with.
   Ex: “Show Item ItemName”
Hide - An item becomes invisible and can no longer be interacted with.
   Ex: “Hide Item ItemName”
PickUp - Player grabs an object

Note: Tasks are not required to have Trigger functions, as such, in the event that one would like to combine multiple OnComplete functions, simply create new tasks without triggers, and set their subtasks to be one of the tasks that contains the other OnComplete event you want. Children cannot have multiple parents. There is no check for this at the moment, so try to avoid this.

Objects are typically separated by prepositions; these describe the order in which they are interacting in.
The current list of prepositions are “On, With, And”
   With - Interact an object WITH a tool
   target WITH source
   EX: “Cut Item Tree With Item Axe”
   On - Interact a tool ON an object
   source ON target
   EX: “Drop Item Water On Item ActiveFire”
   And - Interact two objects equally
   object AND object
   EX: “Merge Item Stick And Item Tape”
The term object1 will often refer to source objects, AKA tools. They are not necessarily the first object in a sentence.
The term object2 will refer to the target object (which are NOT tools). They are not necessarily the second object in a sentence.
Appendix H: List of Events

Broadly, the IGameEvent implementors (“events” in this section) fall into three categories: internal communication between UI-side objects, messages from UI-side objects to the QuestManager, and messages from the QuestManager to UI-side objects.

<table>
<thead>
<tr>
<th>Event</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ObjectPlacedEvent</td>
<td></td>
</tr>
<tr>
<td>ObjectPlacedInSeasonStartEvent</td>
<td></td>
</tr>
<tr>
<td>SeasonalObjectPlacedForFirstTime</td>
<td></td>
</tr>
<tr>
<td>ObjectPlacedInSeasonFinishedEvent</td>
<td>Seasons game mechanic</td>
</tr>
<tr>
<td>ObjectPlacedOnInventoryTargetEvent</td>
<td></td>
</tr>
<tr>
<td>ObjectRemovedFromInventoryTargetEvent</td>
<td></td>
</tr>
<tr>
<td>ObjectAcceptedByInventoryTargetEvent</td>
<td></td>
</tr>
<tr>
<td>ObjectEjectedByInventoryTargetEvent</td>
<td>Adding/removing objects from the inventory</td>
</tr>
<tr>
<td>AllScenesLoadedEvent</td>
<td>Allows objects to wait until the game is fully loaded</td>
</tr>
</tbody>
</table>

Internal events listing

<table>
<thead>
<tr>
<th>Event</th>
<th>Related trigger action</th>
</tr>
</thead>
<tbody>
<tr>
<td>ObjectPickedUpEvent</td>
<td>Player PickUp Item X</td>
</tr>
<tr>
<td>ObjectStoredEvent</td>
<td>Player Get Item X</td>
</tr>
<tr>
<td>ObjectRetrievedEvent</td>
<td>Player Drop Item X</td>
</tr>
<tr>
<td>PlayerOpenedInventoryEvent</td>
<td>Player Open</td>
</tr>
<tr>
<td>ObjectHitEvent</td>
<td>Player Hit Item X With Item Y</td>
</tr>
<tr>
<td>ObjectDroppedOntoDropTargetEvent</td>
<td>Player Drop Item X On Item Y</td>
</tr>
<tr>
<td>ObjectMergeEvent</td>
<td>Player Merge Item X With Item Y</td>
</tr>
<tr>
<td>ItemWaitedEvent</td>
<td>Player Leave Item X For 2 Seasons</td>
</tr>
<tr>
<td>Event</td>
<td>Related OnComplete action</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>PlayerRecieveObjectEvent</td>
<td>Player Get Item X</td>
</tr>
<tr>
<td>ReplaceObjectEvent</td>
<td>Item X Become Item Y</td>
</tr>
<tr>
<td>PlaySoundEvent</td>
<td>Play Sound X</td>
</tr>
<tr>
<td>ShowTextEvent</td>
<td>Play Dialog X</td>
</tr>
<tr>
<td>DeleteObjectEvent</td>
<td>Delete Item X</td>
</tr>
<tr>
<td>HideObjectEvent</td>
<td>Hide Item X</td>
</tr>
<tr>
<td>ShowObjectEvent</td>
<td>Show Item X</td>
</tr>
</tbody>
</table>

**QuestManager to UI-side events listing**
References


GitHub desktop (2017).


Japanese edo period backpack chest – edo arts. Retrieved from


*The kano school of painting* (2003).


*Meld* (2016).


doi:10.1126/science.301659

https://docs.unity3d.com/Manual/MessagingSystem.html

https://docs.unity3d.com/ScriptReference/Events.UnityEvent.html


https://www.assetstore.unity3d.com/en/#!/content/62551

菊の節句とは？その日に食べるものは？どんなことをするの？. (2015). Retrieved from
http://pinez15.com/259.html
