



The Conduit's Garden

A Fourth-Dimensional Shinto Garden

A Major Qualifying Project Report

Submitted to the Faculty of

Worcester Polytechnic Institute

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

On October 17th, 2016

Submitted by: Alex Guerra, John Nelson, Paul Orvis

Advised by: Professor Jennifer deWinter and Professor Charles Rich

Abstract

This report summarizes the development process of the Interactive Media and Game Development and Computer Science Major Qualifying Project titled *The Conduit's Garden*. This game was completed over a three month period at the Osaka University Toyonaka Campus in Osaka, Japan. *The Conduit's Garden* is a fourth-dimensional Japanese gardening game developed in Unity3D to work with the HTC Vive virtual reality platform. Players are given the role of a Shinto spiritual conduit tasked with caring for the bonsai trees, Koi ponds, and gravel pits in a fourth-dimensional Japanese garden to appease local nature spirits. This report details the designs, implementations and playtesting that went into developing this game as well as aspirations for future development.

Acknowledgements

The team members of *The Conduit's Garden* would like to thank the following people for assisting us throughout the development of this project:

Thank you to Professors Jennifer deWinter and Charles Rich for the support you gave us during this project. Thank you for the constructive criticism on the game and the paper as well as the advice on having an enjoyable experience in Japan.

Thank you to Professor Kiyoshi Kiyokawa of Osaka University for being a great on-site advisor. We couldn't have made this game without the resources and technical advice you provided us throughout the project.

Thank you to our playtesters for taking the time to play our game and provide meaningful feedback on areas that needed improvement.

Thank you to Takemura Lab for graciously hosting us during our stay in Osaka while we worked on our project.

Table of Contents

Abstract.....	2
Acknowledgements.....	3
List of Figures.....	8
List of Tables.....	11
1 Introduction.....	12
2 The Fourth Dimension.....	18
2.1 Design Decisions.....	19
2.2 Fourth-Dimensional Objects.....	22
2.2.1 Hyper-Creature Code.....	22
2.2.2 Hyper-Object Code.....	23
2.2.3 Hyper-Collider Code.....	25
2.3 Fourth-Dimensional Shaders.....	26
2.3.1 Design Requirements.....	26
2.3.2 Programming Challenges.....	28
2.3.3 Implementation.....	33
3 Core Game Zones.....	42
3.1 Bonsai Trees.....	42
3.1.1 Bonsai Design.....	43

3.1.2	Interacting with the Bonsai	46
3.1.3	Bonsai Implementation	48
3.2	Koi Fish	53
3.2.1	Fish Design	54
3.2.2	Interacting with the Fish	56
3.2.3	Fish Implementation	59
3.3	Gravel Pit.....	61
3.3.1	Gravel Pit Design	62
3.3.2	Interacting with the Gravel Pit.....	64
3.3.3	Gravel Pit Implementation.....	65
3.3.4	Pattern Recognition.....	66
4	Additional Game Content	75
4.1	Spirit Shrines	75
4.1.1	Design	76
4.1.2	Implementation	76
4.2	Kami (Nature Spirits).....	79
4.2.1	Design	80
4.2.2	Implementation	80
4.3	Wash Basin.....	82
4.3.1	Design	82

4.3.2	Implementation	83
4.4	Scrolls.....	84
4.4.1	Design	84
4.4.2	Implementation	88
4.5	Clock	89
4.5.1	Design	89
4.5.2	Implementation	90
5	Tools and Specifications	91
5.1	HTC Vive.....	91
5.1.1	Controller	91
5.1.2	Movement	91
5.1.3	User Interface.....	92
5.2	Unity3D.....	93
5.2.1	HTC Vive Support	93
5.2.2	Cg/HLSL Shader Programming Support	94
5.3	Art Tools	95
5.3.1	Blender	95
5.3.2	Gimp	95
6	Playtesting.....	96
6.1	Control Mapping Test	96

6.2	Four-Dimensional Playthrough	98
6.3	Three-Dimensional Playthrough	102
7	Postmortem	106
7.1	What Went Right.....	106
7.1.1	Design	106
7.1.2	Teamwork	107
7.1.3	Above and Beyond Work.....	108
7.1.4	Positive Changes	109
7.2	What Went Wrong	111
7.3	What We Would Do Differently	113
	References.....	115
	Appendix A: Playtesting Resources.....	118
A.1	Outline of Procedure	118
A.2	Interview Questions.....	119
	Appendix B: List of Game Content	120
	Appendix C: Additional Interactive Media Class	122
C.1	Alex and Paul's Interactive Project: Bubble Keyboard	122
C.2	John and Yuheng's Interactive Project: Rope Skipping Simulator.....	124

List of Figures

Figure 1: Four-dimensional Cubes (Hypercubes) Viewed from Three Different W Positions	14
Figure 2: Three-Dimensional Cross-Sections along the Fourth Dimension.....	18
Figure 3: Four-Dimensional Scene View	20
Figure 4: Vision along the Fourth Dimension	21
Figure 5: Moving a Hypercube from W Position zero to W Position One.....	25
Figure 6: Unity Rendering Pipeline (<i>ShaderLab: Blending</i>)	29
Figure 7: Phong Reflection Model.....	30
Figure 8: Opaque Shader on Game Objects.....	33
Figure 9: Opaque Shader Properties	34
Figure 10: Opaque Shader Pass 1 Vertex Function	35
Figure 11: Opaque Shader Pass 1 Fragment Function.....	36
Figure 12: Opaque Shader Pass 2 Vertex Function	37
Figure 13: Opaque Shader Pass 2 Fragment Function.....	37
Figure 14: Transparent Shader on Game Objects	38
Figure 15: Transparent Shader Properties.....	38
Figure 16: Transparent Shader Pass 1	39
Figure 17: Transparent Shader Pass 2.....	40
Figure 18: A Healthy Bonsai Tree	43
Figure 19: Bonsai Tree Components	44
Figure 20: Bonsai Components on Different Points of the Fourth Dimension.....	44
Figure 21: Stages of Bonsai Tree Infestation.....	45
Figure 22: Trimming a Dead Bonsai Leaf with Shears	47

Figure 23: Removing a Bonsai Infestation with Insecticide	47
Figure 24: Bonsai Growth Cycle	48
Figure 25: Infestation Decision Tree	49
Figure 26: Leaf Death Decision Tree.....	50
Figure 27: Koi Swimming in Fish Pool	54
Figure 28: A Hungry Fish (left) and a Happy Fish (right).....	55
Figure 29: Grabbing a Fish in the Reservoir Pool	56
Figure 30: Fish in the Reservoir Pool	56
Figure 31: Pouring Fish Food into a Fish Pool.....	57
Figure 32: Koi Feeding Behavior Reference	58
Figure 33: Fish Feeding Cycle	59
Figure 34: RequestTarget Sequence	60
Figure 35: Player Designed Gravel Pit	61
Figure 36: Gravel Pit.....	62
Figure 37: In-Game Pattern Lines.....	63
Figure 38: Wooden Rock Garden Rake Reference.....	63
Figure 39: Rake.....	63
Figure 40: Tine Drawing by Pixel Boxes	66
Figure 41: Comparison between Original and Player Made Pattern	69
Figure 42: The Player in the Process of Raking, Trying to Match the Pattern Lines	71
Figure 43: Pattern Recognition Pseudocode	72
Figure 44: Wooden Spirit Shrine Reference	75
Figure 45: Gravel Shrine.....	77

Figure 46: Fish Shrine.....	78
Figure 47: Bonsai Shrine	79
Figure 48: Kami Reference.....	79
Figure 49: Kami above the Garden.....	80
Figure 50: Traditional Stone Basin for Washing Hands Reference.....	82
Figure 51: Wash Basin and Ladle.....	83
Figure 52: Contract Scrolls	84
Figure 53: Tokyo Contract Scroll Text.....	85
Figure 54: Controls Scroll Text	86
Figure 55: How to Play Scroll Text	86
Figure 56: Game Concept Scroll Text	87
Figure 57: Information Scrolls.....	87
Figure 58: Garden Clock.....	90
Figure 59: Control Mapping Test Scene.....	97
Figure 60: Qualitative Results of Control Scheme Test	98
Figure 61: Fourth-dimensional Playtest Tokyo Garden.....	99
Figure 62: Testers' Opinions on Game Difficulty	100
Figure 63: Testers' Opinions on Individual Game Zone Difficulty	100
Figure 64: Three-dimensional Playtest Tokyo Garden.....	103
Figure 65: Testers' Opinions of Game Difficulty after Removing 4D.....	104
Figure 66: Testers' Preferred Game Types.....	104
Figure 67: Mockup of the Bubble Keyboard Design.....	123
Figure 68: The Group Working with Bubbles to Measure if they would Conduct Current	124

List of Tables

Table 1: Normalized RGB Color Value per 4th Axis Position.....	27
Table 2: Completed Content per Person	121
Table 3: Incomplete and Cut Game Content.....	121

1 Introduction

While studying abroad at Osaka University in Osaka, Japan from July to October of 2016, three WPI students - Alex Guerra, John Nelson, and Paul Orvis - developed *The Conduit's Garden* for their Major Qualifying Project. *The Conduit's Garden* is a fourth-dimensional Japanese gardening game developed to work with the HTC Vive virtual reality platform. In the game, players take on the role of a Shinto spiritual conduit who specializes in restoring four-dimensional Japanese Shinto gardens.

As a spiritual conduit, players must work to create a stable balance between civilization and nature to appease the ancient nature spirits residing in the heart of Japan. In our game, a former temple worshipper who recently moved to Tokyo asks the conduit to visit his new home. By having the conduit tend to his rooftop garden, he hopes to fill his home with the energy of the spirits, or Kami, that thrived in the gardens of the conduit's temple. Keeping the garden's bonsai trees free of decay and pests, the koi ponds full of fat and happy fish, and the gravel pits filled with flowing designs will bring more Kami to the garden within the short time allowed by the client and fulfill the wishes an old friend to the temple.

We designed *The Conduit's Garden* to give players a sense of pride and satisfaction in creating something that is enjoyed and praised by others. We also wanted to capture the feeling of falling in love with a project, but eventually having to accept the consequences of deadlines. In designing the experience goal of our game, we wanted to parallel the exact feelings we have when creating a game that players will both appreciate and criticize. This experience is realized by tasking players to create enjoyable environments for the nature spirits of Japan. We want

players to work hard on these projects so they can bring joy and glee to the spirits of the garden and let their imagination manifest in the way they choose to grow the fourth-dimensional garden.

The designs of our game were shaped by the constraints that guided our development. For our team, constraints turned our project into a learning experience where we had to research and learn new concepts and methods that we could then use outside of this project in the future. This project is the result of adhering to our own defined constraints along with those set by our advisors. We wanted the game to be set in a fourth-dimensional environment, playable using the HTC Vive and incorporate elements of Japanese culture.

The world of virtual reality in video games is constantly growing. Being able to work in a lab with access to many of these technologies is a great opportunity to use and learn about them. Virtual reality games are also very unique as more considerations need to be made during development in an attempt to use the full potential of having VR technology. Real world devices, unfortunately, have a limited range of space where the device can be used. This will impact how we design our virtual environment and how we want players to interact with our game.

Our team had the privilege of completing our Major Qualifying Project in a technologically advanced lab located in Japan. This project is not about grinding out a game; it is about exploring a new part of the world and allowing this experience to influence how we design our game. Having this constraint required us to explore Japan and incorporate our experiences into our designs.

Early before we began development on this game, we set a priority in creating a game that explored fourth-dimensional game mechanics. In the real world people cannot see or knowingly interact with the fourth dimension or four-dimensional objects. Such objects, however, can be constructed and simulated in a virtual environment. We believed that

successfully simulating the fourth dimension in our game would set our project apart from the ones completed before us.

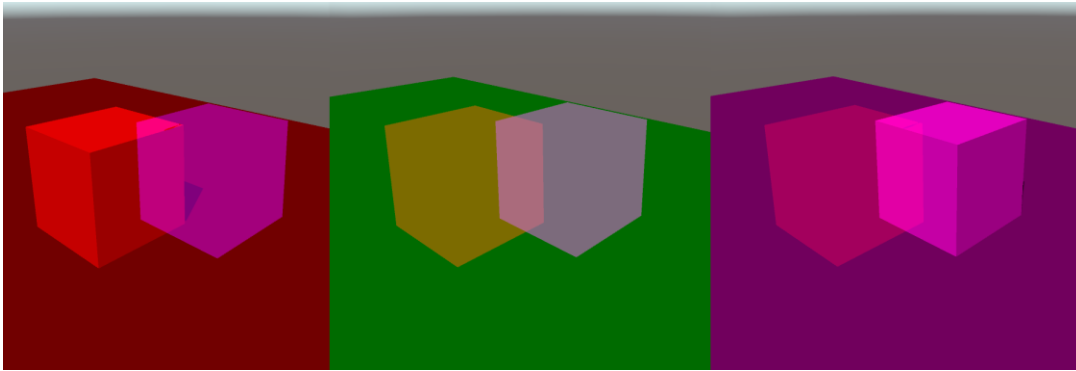


Figure 1: Four-dimensional Cubes (Hypercubes) Viewed from Three Different W Positions

In our game we use colors to represent the fourth dimension; specifically five of the seven colors of the rainbow. Fourth-dimensional objects, called ‘Hyper-Objects,’ have x , y and z coordinates like three-dimensional objects, but also have a w coordinate for their position on the w axis. Objects at a w value of zero appear red while objects at a w value of four appear magenta. What this does is create five similar spaces that are overlapped in three-dimensional space. Figure 1 depicts two hypercubes placed next to each other in a fourth-dimensional world. The red cube has a w value of zero and the magenta cube has a w value of four. While the cubes appear to intersect in three-dimensional space, they do not because of four-dimensional space. The magenta cube only takes up space on the fourth w point, but can be seen from the other points on the w axis as transparent. The same goes for the red cube, as it only takes up space on the first w point; therefore the two cubes are not physically colliding. More details about the fourth dimension and how we use it in our game are explained in Chapter 2.

We have two primary motives for including the fourth dimension in our game. Our first motive is due to the spatial limitations of the HTC Vive playing area. With our range of the fourth dimension we can expand the relatively small, two meter by two meter, playing space of

the Vive by five. In a garden like ours, where many objects must be within the player's reach, this extra room is invaluable. With the fourth dimension we can layer each of our important game features on top of each other while limiting player collisions to make the garden more spacious than it is in reality. We can also limit player vision to not see objects on other points on the fourth dimension. This makes the game world less cluttered and feel more spacious.

Our second motive is to introduce our players to what we believe is a fascinating and exciting concept. Most people never imagine what the fourth dimension could be in their lifetime and, to our knowledge, no one has ever experienced existing on four dimensions of space. Giving our players a space with which to explore and interact as a fourth-dimensional being will be a truly unique experience.

The environments in our game are populated with these four-dimensional objects whose properties and physics are modified for a new level of interaction for players. The player character also has this presence and the ability to move along the fourth dimension. The original idea for our game was to have a garden grow along four dimensions of space. In the fourth-dimensional Shinto garden we ended up creating, players interact with this new environment to the point where they learn to become four-dimensional beings themselves.

Our implementation of the fourth dimension is very basic to allow our players to receive an initial understanding of fourth dimension through our game. There are many more physical interactions between fourth-dimensional objects and methods of moving in four-dimensional space that we did not implement into our final product. We felt that many of these actions and behaviors would confuse a player who is unfamiliar with the fourth dimension. Designing a tutorial to familiarize players with these additional features would expand our scope to unmanageable proportions. Additionally, a gardening game is a poor choice of genre to convey

all the features of the fourth dimension. We felt it was best to simplify the fourth dimension to enhance the normal conventions of gardening rather than overcomplicating a familiar topic with the fourth dimension.

Our game consists of three core game zones that concentrate the majority of gameplay. These core game zones are fully discussed in Chapter 3. We designed these game zones to align with the primary tasks required of gardeners caring for true Japanese Shinto gardens. These three core game zones are the bonsai trees, fish pools, and gravel pits. The bonsai trees grow along the fourth dimension creating this multi-dimensional plant that requires moving along the fourth dimension to prune and shape it. The fish pools can be filled with fish that both live together and completely apart. The fish have their own presence on the fourth dimension, and seeing them collide and interact according to fourth dimensional physics is both life-like and fantastical. The gravel pits are a combining force for the different points on the fourth dimension bringing separate planes of reality together to create flowing and beautiful designs.

This paper discusses in detail the development process of our game *The Conduit's Garden*. Chapter 2 details the concept of the fourth dimension and the designs and implementations of scripts and shaders required of our fourth-dimensional game mechanic. Chapter 3 details the three core game zones in our game: the bonsai tree, fish pools, and gravel pits. Chapter 4 details the additional game content created to support the core game zones and structure the game around our mechanics and puzzles. Our additional game content includes the spirit shrines that provide feedback on the core game zones, the wash basin that provides full 4D vision, the Kami representing the score, the information scrolls outlining controls and gameplay, and the garden clock displaying remaining game time. In each section of Chapters 3 & 4, we detail our designs behind the game mechanic and the method by which we implemented it.

Chapter 5 details the results of our playtesting and how those results guided the changes made in our final product. A postmortem on the positives and negatives we experienced during our project can be found in Chapter 6. The Appendices of this paper provide resources for our playtesting, a list of game content created by each teammate, and a short excerpt on the additional interactive media class we participated in while working at Osaka University.

2 The Fourth Dimension

The fourth dimension is another axis of space in addition to the three we accept as the structure of our universe. Instead of just existing within three-dimensional space (on the x, y, and z axis) our game world exists within four-dimensional space (on the x, y, z, and w axis). The best way to visualize the fourth dimension is to move from the zeroth dimension up to the fourth dimension. At the zeroth dimension we have a point, at the first dimension we extrude the point to a line, at the second dimension we extrude the line to a square, at the third dimension we extrude the square to a cube, and at the fourth dimension we extrude a cube to a hypercube. A hypercube can be considered a collection of three-dimensional cubes at many points along the w axis (Figure 2).

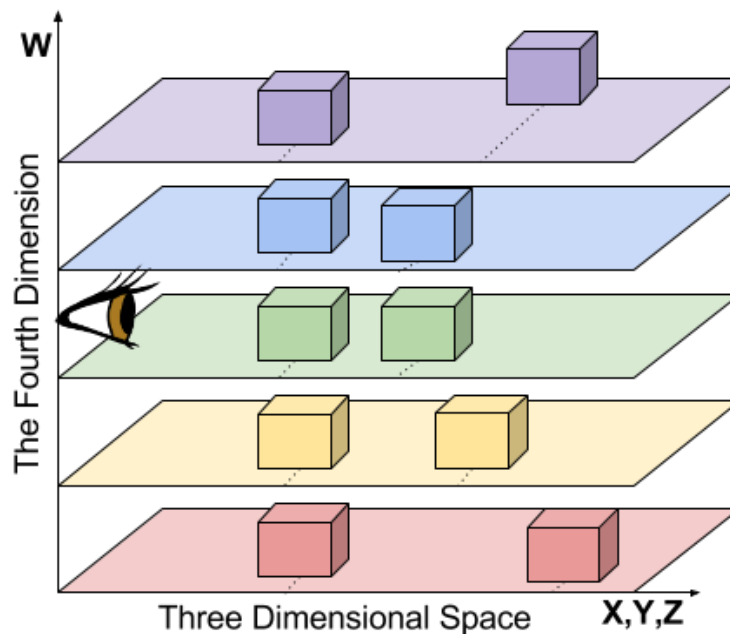


Figure 2: Three-Dimensional Cross-Sections along the Fourth Dimension

2.1 Design Decisions

One of the key mechanics of our game is placing players in a fourth-dimensional world and giving them a fourth-dimensional body to control. We implement this in our game by placing players in a garden that grows along four dimensions. We wanted to introduce this conceptually complex mechanic in a way that didn't overwhelm new players with too much new information. Incorporating this mechanic into a garden would provide them with a visually pleasing environment in which to experience the fourth dimension. Our goal with this fourth-dimensional mechanic is to get players to understand the theory of the fourth dimension through interaction and experimentation.

By fourth-dimensional, we mean four dimensions of space (x, y, z, w) instead of our usual three (x, y, z) as explained previously. This means that an object can have different positions on the w axis, which defines what objects that can and cannot interact with it. This also means that objects can have a depth along the w axis. The w depth of an object is how many more points on the w axis on which this object exists. A cube with a depth of zero on the w axis and a position of $(0, 0, 0, 0)$ could only be seen or touched by players and objects on the zero point of the w axis. A cube with a depth of four, however, could be seen and touched anywhere along our range of the w axis.

Vision along the w axis will be similar to how we look within three-dimensional space, but limited to decrease the complexity of the implementation and controlled to reduce the overwhelming nature of the experience. Customarily, we view things by pointing our eyes, or a camera, along a singular direction on an axis. For example, we point a camera towards the positive direction on the x axis. When doing so, we also have peripheral vision within a limited range along the y and z axis in both the positive and negative directions. We plan to give players

peripheral vision on the w axis as well as the y and z when looking down the x axis. The same goes for when players look along the y or z axis. This means that players will be able to see all objects at a set distance along the w axis at the same time.

We will do this by merging the objects found at each point of the w axis into the three-dimensional Unity scene inhabited by players (Figure 3). We are essentially taking three-dimensional cross-sections of the fourth dimension at five points and overlaying these cross-sections into one scene to allow players to obtain a semblance of fourth-dimensional vision. Objects in the scene will be rendered to replicate their fourth-dimensional characteristics. Objects on the same point of the fourth dimension as the player character will be solid as they are in three-dimensional space. Objects on different points of the fourth dimension will be transparent so that players can see them all at the same time as a true fourth-dimensional being would. These transparent objects will be visible to players even if the transparent objects are found behind other solid objects in three-dimensional space.

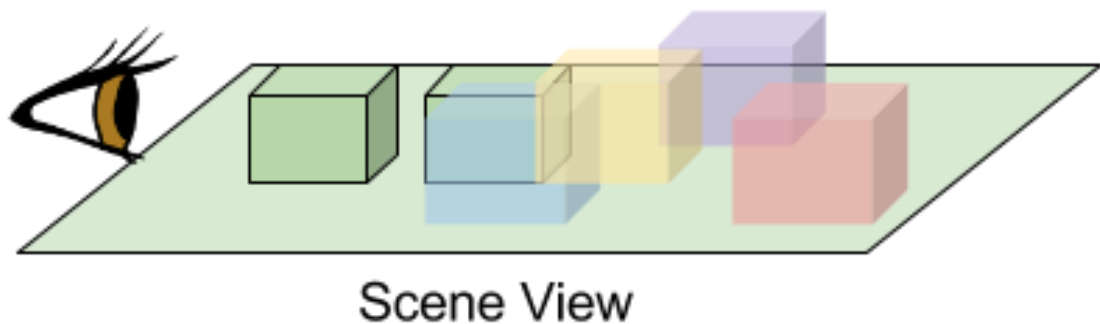


Figure 3: Four-Dimensional Scene View

Figure 4 demonstrates how the player camera will treat objects on different points of the fourth dimension. The camera is located at the origin (0, 0, 0, 0) and views objects at each point of the fourth dimension. As is shown, the camera has vision of every object in this scene except

for square B since the camera's vision of square B is occluded by the geometry of square A. Even though the green squares have the same x, y and z position as the red squares, they are still fully visible because the camera can view them along the empty space along the w axis. The camera is also able to see the square E even though it has a further x, y and z position than square B because there is nothing occluding it along the w axis. This means that when we condense all of these objects into the same three-dimensional scene, square E will still have to be visible. Finally, squares B, C, D, and G may all share the same position in three-dimensional space, but since they are all on different points along the fourth dimension, they will all be visible to the camera.

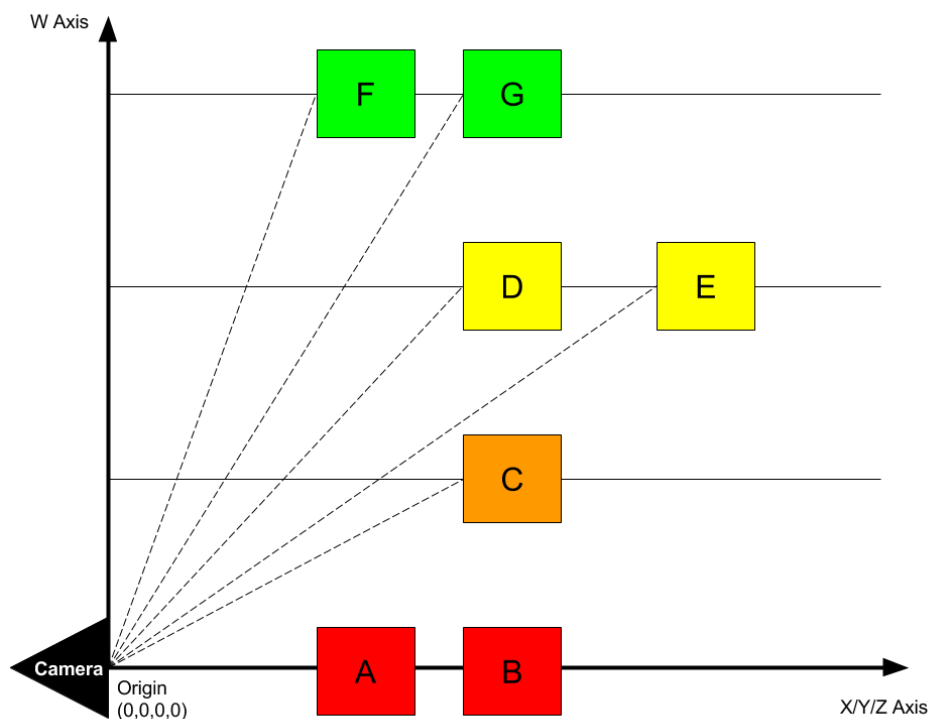


Figure 4: Vision along the Fourth Dimension

For many people, this will be their first time hearing of the fourth dimension and we hope that by showing fourth-dimensional movement and presence through a familiar topic such as gardening, our players will be able to more easily understand the fourth dimension. By giving

players a hands-on experience with the fourth dimension, they can overcome the challenging initial stages of perceiving the fourth dimension and have a far easier time envisioning how things work in fourth-dimensional spaces.

2.2 Fourth-Dimensional Objects

In the real world, we cannot easily visualize or experience fourth-dimensional objects and environments. In the virtual world, however, using code to give objects a fourth coordinate can be a way to simulate a four-dimensional environment. The code we wrote for the fourth dimension is broken up into three scripts:

- The Hyper-Creature Script: allowing players to move on the fourth dimension
- The Hyper-Object Script: simulating the visual appearance of four-dimensional objects
- The Hyper-Collider Script: simulating the collisions between four-dimensional objects

The following sections will discuss the implementation of each of these scripts and their role in simulating the fourth dimension.

2.2.1 Hyper-Creature Code

The Hyper-Creature script holds all the information about the player character's position on the w axis along with the value of the player's peripheral vision of the w axis. The script is placed on the camera for the eyes of the Vive camera rig prefab. Since there are many objects that need to have a reference to the hyper-creature, it has been declared as a singleton. This makes it easier for other objects to get the reference to the hyper-creature through a static variable rather than using the built-in 'find' methods that traverse every object in the scene.

The two main functions found in this script are `WMove` and `WMoveAllHyperObjects`. The `WMove` function is what the controllers use to change the `w` position of the player character. Although the `w` value of the player character is public, this method contains the calculations to ensure that the `w` value does not go below zero or above four. `WMoveAllHyperObjects` is a public function that allows any object to force all hyper-objects in the scene to update their visuals. Objects that change players' peripheral vision along the `w` axis, like the wash basin, will use this function. The variable for players' `w` peripheral vision is public, but hyper-objects cannot detect when this value changes, so instead `WMoveAllHyperObjects` should be called. This function iterates through all the hyper-objects in the scene and calls their `WMove` method. This function is infrequently called as to not impact the performance of the game.

The hyper-creature script also holds the functions that cause the camera to fade in and out between scenes. These function will return true if the camera is done fading in or out. Other objects that control when the game starts and end can use these functions to postpone changing levels until the camera finished fading out or in.

2.2.2 Hyper-Object Code

The Hyper-Object script acts like a plugin for game objects to give them a `w` coordinate and depth along the `w` axis. This script controls the visuals of hyper-objects to draw them to simulate how they would appear in a fourth-dimensional environment. The texture of the object is also loaded here as our custom shaders are controlled through this script.

The main function that controls the visuals for fourth-dimensional objects is called `WMove`. This function runs calculations on the object's position on the `w` axis and compares it to the player character's position on the `w` axis. The player's fourth-dimensional peripheral vision and the object's depth on the `w` axis are also considered in these calculations. The code will use

all the variables to determine if the object is on, or passes through, the player character's w position. If this is true, then the object should appear solid, if not then it should be invisible or transparent. A coroutine is then started that will smoothly change the color and transparency of the object to simulate players moving between two points on the w axis. During this process, the shader for the object is changed depending if the object is becoming transparent or solid. Objects on the w point that players are moving to will increase in opacity and use our custom opaque shader, while objects players are moving away from will decrease in opacity and use our custom transparent shader. If players have peripheral vision along the w axis, objects that players are moving away from will have 20% opacity and still be visible to them. If they do not have any peripheral vision, instead those same objects will fully decrease in opacity and become invisible.

WMove can be called by the Hyper-Creature script using WMoveAllHyperObjects, as mentioned in the previous section. Since our game will contain many hyper-objects, calling a function that first finds, then loops, through all instances of hyper-objects would be inefficient and have a negative impact on in game performance. The better approach that we implemented is to have each hyper-object hold a reference to the controller manager and wait for a variable in one of the controllers to be true. When the controller registers a trigger being pressed it will set a public variable to true so that the hyper-objects will know to run the WMove method. The controller will set the variable near the end of the update step and then reset it to false at the start of the next update step. The hyper-object checks the variable in its late update step so that it is reading the variable in between the time it is set by the controller and the time it resets back to false.

2.2.3 Hyper-Collider Code

Physical interactions between hyper-objects are controlled through the Hyper-Collider Manager script. The important function in this script is the SetCollisions function. It operates similar to the WMove function; the object running the method is checking its own variables against all other objects with hyper-collider scripts to determine if this object would collide with those objects. If an object determines that it cannot collide with another object because the two objects are not on the same w position or they do not extend into similar w points, then the collider on the object will ignore all collision events containing that object.

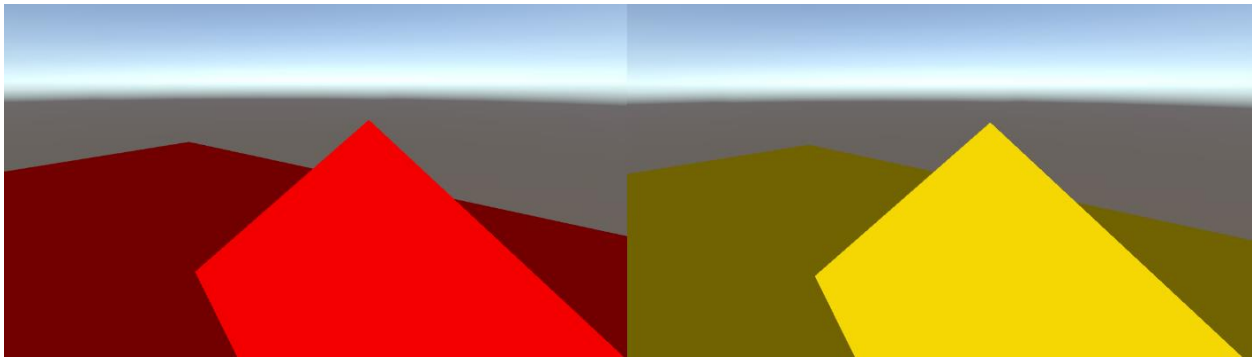


Figure 5: Moving a Hypercube from W Position zero to W Position One

Players can move objects to different points on the w axis by holding them as they move along the w axis (Figure 5). The SlideW method in the hyper-collider manager script checks to ensure that the object can be moved to the w position players are moving towards. If an object's depth would cause it to extend past the minimum or maximum w value (in our game this is zero and four respectively) then the object will not be moved on the w axis, but players will still be able to hold the object. If the object can move along the w axis, then the object will reset its collisions to recalculate what objects it can now collide and not collide with.

Hyper objects may be made up of a collection of children with any combination of hyper-object and hyper-collider manager scripts attached to them. The hyper-object and hyper-collider

code have recursive functions that check for children with these scripts and use them to determine visuals and collision. The SlideW function will check that all the children would be able to move before performing the move. During recursion, any hyper-object scripts would also have their SlideW method called as this would update the color of the object to show it has moved to a different w position.

2.3 Fourth-Dimensional Shaders

The key technical challenge in implementing our fourth-dimensional environment was developing a method by which we could render objects according to their position within fourth-dimensional space. Unity3D is built to render objects in three-dimensional environments by testing the distance between the camera and each object and drawing them over each other accordingly. We had to build upon Unity's current rendering process to account for an object's position along the fourth axis of space and provide us with a representation of a fourth-dimensional environment. In order to achieve this effect, we had to build shaders and scripts that satisfied our technical requirements.

2.3.1 Design Requirements

According to our design, we chose to constrain our fourth-dimensional environment to five points along the fourth dimension. We originally designed the game to work with seven points along the fourth dimension, but our playtesting results caused us to move to five points to improve player comprehension. These points would be found at whole numbers from zero to four and players would only be able to move by snapping their position to one of these points. We did not account for positions with decimal values between two whole numbers.

These five points had to be noticeably different to players, so we rendered all objects at a specific point on the w axis with the same base color (Table 1). The five colors we chose were red, yellow, green, cyan, and magenta. To allow for variation between objects with the same base color, we altered the color value of individual objects to make them darker or lighter.

W Position	Red	Green	Blue
0	1.0	0.0	0.0
1	1.0	1.0	0.0
2	0.0	1.0	0.0
3	0.0	1.0	1.0
4	1.0	0.0	1.0

Table 1: Normalized RGB Color Value per 4th Axis Position

In order to render a fourth-dimensional environment, we designed our game to overlay three-dimensional cross sections of points along the fourth dimension onto our three-dimensional Unity environment. According to our designs outlined in Chapter 2.1, objects within three-dimensional cross-sections at the same point on the w axis as the player character had to be rendered as opaque, while objects within cross-sections on different points would be rendered as transparent. Not only did objects have to be transparent when on different points of the w axis from the player character, but they also had to be visible at all times by not being culled by opaque geometry in the scene. These transparent objects had to be drawn over all other geometry even if they were physically behind another object.

With our peripheral vision mechanic, only objects within the range of the player's peripheral vision would be drawn to the screen. For example, a peripheral vision of zero would only draw objects on the same point of the w axis as the player character while a peripheral vision of two would draw objects on every point within our range of the w axis independent of

the player character's w position. This range of peripheral vision would change as players activated more shrines in the garden or drank from the wash basin.

2.3.2 Programming Challenges

The initial challenge in implementing these fourth-dimensional shaders was learning how to program custom shaders for Unity. After a bit of research, we discovered a good collection of tutorials on how to program shaders in Unity. The best tutorials on Unity shader programming were found at the Cg Programming in Unity Wiki page (*Cg Programming/Unity*) which taught us how to program shaders for basic materials, transparent surfaces, basic lighting, basic texturing, three-dimensional texturing, environment mapping, and complex lighting. The tutorials that taught us the techniques we used in programming the fourth-dimensional shaders were on Diffuse Reflection, Order-Independent-Transparency, and Textured Spheres.

The Unity Documentation Manual's Shader Reference (*Shader Reference*) provided a set of tutorials on surface shaders, vertex and fragment shaders, and fixed-function shaders. These tutorials were especially helpful because they taught step-by-step guides in building complex shaders from scratch. This manual is useful for building an initial level of shader programming knowledge for newcomers.

After completing the tutorials on shader programming found at the listed websites, we began programming the fourth-dimensional shaders. Due to the design of our gardens, there is a large mix of transparent and opaque objects within the game scene. At a given time, most stationary scenery and interactive game objects need to be rendered as solid because they are present on the player character's position along the fourth dimension. For example, scenery objects like rocks will have an origin at zero and a depth of four, making them present at every point within our range of the fourth dimension. We could have rendered these objects using the

transparent shader and simply set their alpha to the max, 1.0, but that would require the GPU to render them with the same performance as any other transparent object. Unity has a sizable performance difference when rendering opaque objects compared to transparent objects due to its rendering pipeline.

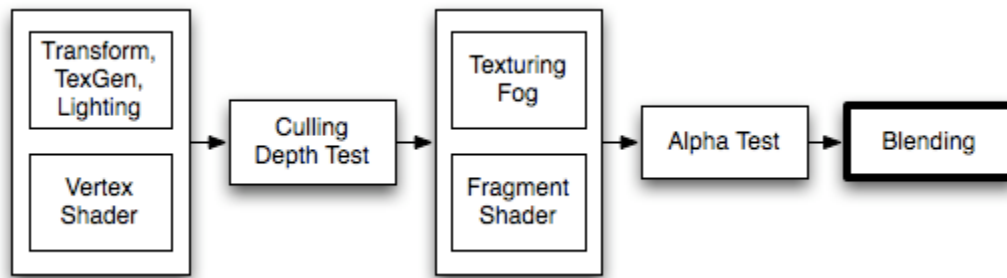


Figure 6: Unity Rendering Pipeline (*ShaderLab: Blending*)

In Figure 6, the GPU finishes drawing opaque objects at the fragment shader, while transparent shaders have to go through the alpha test and blending stages. Opaque objects also have the benefit of always culling their back faces, so that only a portion of the vertices in an opaque mesh are drawn. Also, transparent shaders often have multiple render passes to achieve their desired effect which multiplies the effort required of the GPU. This massive difference in performance requirement takes its toll on the GPU, which for our game, is already busy rendering images for the display of the HTC Vive. We had to make sure that our shaders could achieve our desired visual effect without causing a significant drop in our game's average frame-rate. Our solution was to swap the transparent shader used by an object's material with an opaque shader whenever the object was on the same position on the fourth dimension as the player character.

The next challenge we had to overcome was getting transparent objects drawn over every other object in the scene independent of their position in three-dimensional space. This effect was simply achieved by turning off the ZWrite for our transparent objects and setting the ZTest

to NotEqual. This ZWrite tag is a Unity ShaderLab property that determines if pixels drawn to by an object are written to the depth buffer (*Culling & Depth Testing*). By turning this off, we can draw transparent objects in Unity. The ZTest tag is a Unity ShaderLab property that determines how depth testing is performed on the pixels drawn to by an object. The depth test is performed between the vertex and fragment shader functions to determine if a pixel, not a vertex, is occluded by objects within the scene. By setting this to NotEqual, an object is drawn at all times unless it is at the exact same position as another object. We chose to use the NotEqual value because Unity has a hard time deciding which pixel to draw when a transparent and opaque object occupy the same position, so it flickers between the two desired colors every other frame.

Our opaque shaders are a combination of methods found on diffuse lighting of textured surfaces and point lighting from multiple light sources. The diffuse lighting we use in our opaque shader applies the Phong reflection model (Figure 7, *Lighting Textured Surfaces*) using texture color and the given color constant as parameters for the material constant k_{diffuse} .

$$I_{\text{diffuse}} = I_{\text{incoming}} k_{\text{diffuse}} \max(0.1, \mathbf{N} \cdot \mathbf{L})$$

Figure 7: Phong Reflection Model

Since we don't want any object being colored black by the diffuse reflection, we set the minimum possible value of the dot product between the vertex normal direction and the incoming light direction to 0.1 (scale of 0.0 – 1.0). Multiplying the fragment texture color and the color constant provides us with a material constant that allows for the effect of diffuse lighting on our textured objects.

This diffuse reflection model is expanded further to account for up to four lights present in our game scenes (*Multiple Lights*). First we have to alter the material constant to account for the attenuation of the light source if it is a point light rather than a directional light. The

attenuation of the point light tracks the distance between the fragment and light position in world space. Since directional lights are constant, the attenuation should be constant with no alteration to the reflection value. Then, we compute the aggregate lighting value of each of the four possible lights for each vertex. This aggregate value is then added to the diffuse reflection value when computing fragment colors. We chose to limit the use of point lights in our scenes because this lighting method quickly overloads fragment colors to the maximum value of white when many lights are placed next to an object.

Researching possible implementations for transparent shaders, we came across a method of performing a two pass render on objects for a more photogenic and clear transparent effect. This method combines multiplicative and additive blending to create an order-independent effect that works without having to consider the rendering order of triangles within object geometry (*Order-Independent Transparency*). The first rendering pass of this order-independent transparency runs a multiplicative blending on the pixels rendered by the background with the alpha value of the transparent object. This is performed on the “contents of the framebuffer before the triangles are rasterized” to improve the additive blending of triangles as they are pixelated during the rasterization process. The second rendering pass multiplies the alpha value of the object to the color data and adds the value to the filtered pixels of the multiplicative pass.

After creating our shaders and testing them on our game scenes, we noticed that the scene skybox was occluding transparent objects. This occurs because the Unity skybox is drawn over objects that do not perform a depth test on themselves. Therefore, the transparent objects could be seen only if it were being drawn over an opaque object. We found a solution on the Unity answers forum written by user Matou who proposed turning off the Unity Skybox, creating a custom skybox object using a provided skybox shader, attaching the skybox object to the game

camera, and attaching a script to the skybox object that freezes its rotation on the x, y, and z axis (*Transparent shader in background queue*). We placed the skybox on the in-game HTC Vive camera rig camera and we were able to see the transparent objects over the skybox. We also noticed that the skybox object had to fit within the limits of the Vive camera's rendering distance. This meant that our game scenes had to be limited in size to fit within the skybox area and the camera's rendering distance. We chose not to increase the Vive camera's rendering distance because it would heavily decrease the game frame-rate. This limited size had no significant impact on our design of the Tokyo garden or home level since the HTC Vive already limited player movement.

The final step in implementing the shaders was integrating them with the existing hyper-object code. Hyper-objects are designed to smoothly transition the RGBA value on their material as they move in relation to the player character's position on the fourth dimension. Once a transitioning transparent object's alpha nears 1.0, the hyper-object script assigns the opaque shader to the object material and carries over the color and texture data. When an opaque object begins its transition to transparent, the hyper-object script assigns the transparent shader to the object material and carries over the color and texture data. Unity provides functionality within C# or JavaScript scripts to assign a material's shader and assign the shader's property values.

With the shader functional and implemented, we tried different shading styles for the opaque fourth-dimensional shader that would provide the clearest and most visually appealing experience for players. At first, we tried applying cel-shading to our objects. The cel-shading applied a single shading value of about 25% the material color to all vertices where the dot product between the directional light direction and the normal direction were less than zero. For complex objects with many faces, the effect was visually appealing, but it was hard to tell the

difference between objects in the scene since they were all made up of two similar color values. Instead of going with the stylistic cel-shading, we resorted to a basic diffuse shading that allowed for more variation in shadow values on objects. However, we didn't want an object to appear black at any point, so we set a minimum shading value that the diffuse shading could not pass. At the most, an object's diffuse shadow could reach 10% the value of the material color.

2.3.3 Implementation

The fourth-dimensional visuals of our game were created by switching objects between two shaders as their position in relation to the player camera changed. The first shader rendered objects as opaque with diffuse lighting, texturing and shadow casting and receiving. The second shader rendered objects as transparent with texturing. The hyper-object script attached to all game objects in our game scenes smoothly switched between the two shaders and altered their displaying color values as the player camera moved along the w axis.



Figure 8: Opaque Shader on Game Objects

While active, the opaque shader renders objects without any transparency and with lighting effects (Figure 8). Diffuse lighting dependent on the direction of the directional light in

the game scene is used to draw light levels on the object's vertices. The shader also makes use of Unity's ShaderLab functions for calculating incoming shadows from other objects and projected shadows on distant objects.

The opaque shader takes in a color value that is applied to the material surface and a texture image that is wrapped to the surface geometry of the object if it has UV wrapping data (Figure 9). In the first pass, the shader uses the ForwardBase tag to calculate the shading required by the directional light and all incoming shadows from other objects in the scene. The second pass makes use of the ForwardAdd tag to calculate incoming light from point lights found in the scene. This shader is run at the render queue order of 2000 along with all other opaque geometry. The shader itself has two passes each with their own vertex and fragment functions.

Opaque Shader Properties	
1	Properties
2	_Color - normalized RGBA color value
3	_MainTex - Texture to be wrapped onto the object's surface
4	Pass 1 Tags
5	LightMode is ForwardBase for directional lighting
6	Pass 2 Tags
7	LightMode is ForwardAdd for point lighting

Figure 9: Opaque Shader Properties

Figure 10 outlines the steps taken by the vertex function in the first pass of the opaque shader. The vertex function must pass on the vertex position in both model space and world space, the vertex normal direction, the vertex texture data, and the diffuse shading required of the directional light and 4 point lights. A majority of this data is calculated using pre-built and on-compile unity functions. Note the for-loop that reads in the data on the point lights. This shader allows up to 4 point lights in the scene to apply lighting to the object using this shader. In cases like our home level, point lights allow us to maintain the ability to cast shadows and illuminate

objects without a direct line of sight to the scene directional light. Without the use of point lights, objects like the ceiling would cast a blanket shadow over all indoor objects.

Opaque Shader Pass 1 Vertex Function	
1	Output position = product of the model-view-projection matrix and the vertex position
2	Output position in world space = product of the model matrix and the vertex position
3	Output normal direction = normalized product of the vertex normal and the inverse of the model matrix
4	Output texture = vertex texture coordinate position
5	For four lights
6	Vertex position to light source = difference between the light position and the vertex position in world space
7	Light direction = normalized vertex position to light source
8	Squared distance = dot product of the vertex position to light source against itself
9	Attenuation = 1 divided by the sum of 1 and the product of the light source unity encoded attenuation value and the squared distance
10	Lambertian value = max between 0 and the dot product of the normal direction and light direction
11	Diffuse reflection = sum of the attenuation, light color, _Color and lambertian value
12	Output vertex lighting = sum of the diffuse reflection and the previous value of the vertex lighting
13	End for loop
14	Call Unity's TRANSFER_VERTEX_TO_FRAGMENT function on the output
15	Return the output

Figure 10: Opaque Shader Pass 1 Vertex Function

The fragment shader in the first pass (Figure 11) converts the vertex data into per-pixel fragment color data. Most of the math for this pass is performed in the vertex function to decrease the work required of the GPU and increase game performance. As performed in the for loop of the vertex function, we must calculate the diffuse reflection of the directional light at this fragment position and apply that value to the texture color and point light diffuse reflection. The if statement in the above pseudo code checks whether the light data being added on this pass is a

directional or point light to account for the correct attenuation and light direction. Directional lights have an assumed world space position at the origin, so there is no need to calculate the direction between the fragment and light.

Opaque Shader Pass 1 Fragment Function	
1	If directional light
2	Attenuation = 1
3	Light direction = normalized directional light position
4	End if
5	If point or spot light
6	Attenuation = 1 divided by distance between fragment and light source
7	Light direction = normalized difference between the light source and vertex position
8	End if
9	Lambertian value = max between 0.1 and the dot product of the normal and light directions
10	Diffuse reflection = product of the attenuation, light color, color property, and lambertian value
11	Texture color = pixel value of the texture at the fragment position on the mesh
12	Fragment color = product of the texture color, the max between 0.1 and the unity encoded shadow value of the vertex, and the sum of the vertex lighting and the diffuse reflection
13	Return fragment color

Figure 11: Opaque Shader Pass 1 Fragment Function

Since the math behind calculating the point light data within the first pass has already been done, the second pass of the opaque shader (Figure 12) simply blends the light data over the previously rendered fragments of the last pass. The second pass vertex function simply passes on the positional, normal, and texture data of the current vertex.

Opaque Shader Pass 2 Vertex Function	
1	Output position = product of the model-view-projection matrix and the vertex position
2	Output position in world space = product of the model matrix and the vertex position
3	Output normal direction = normalized product of the vertex normal and the inverse of the model matrix
4	Output texture = vertex texture coordinate position
5	Return output

Figure 12: Opaque Shader Pass 2 Vertex Function

The second pass fragment function (Figure 13) performs the same process as the first pass fragment function with the exception of the vertex lighting data of the first pass. After running through these shader passes for each scene light, we achieve the visual effect presented in Figure 8 at the top of this section.

Opaque Shader Pass 2 Fragment Function	
1	If directional light
2	Attenuation = 1
3	Light direction = normalized directional light position
4	End if
5	If point or spot light
6	Attenuation = 1 divided by distance between fragment and light source
7	Light direction = normalized difference between the light source and vertex position
8	End if
9	Lambertian value = max between 0.1 and the dot product of the normal and light directions
10	Diffuse reflection = product of the attenuation, light color, color property, and lambertian value
11	Texture color = pixel value of the texture at the fragment position on the mesh
12	Fragment color = product of the texture color and the diffuse reflection
13	Return fragment color

Figure 13: Opaque Shader Pass 2 Fragment Function

While active, the transparent shader renders objects as transparent and draws them over opaque objects in the game scene (Figure 14). Before blending the object colors to the render

image, the shader pulls the object out of testing for distance to the camera and draws to the render image at all times after other solid geometry is drawn. To achieve the transparent effect, the transparent shader performs two passes when rendering an object's geometry by first running multiplicative blending of the object's alpha value to the background render image and then running additive blending of the object's color to the render image. This all results in objects on other points of the fourth dimension from the player character always being drawn and allowing for vision of distant 4D objects.



Figure 14: Transparent Shader on Game Objects

To set up the two pass blending for our transparent shader we need the color and texture properties (Figure 15) of the given object and we need to set the RenderType tag to transparent to allow the object to be drawn with other transparent geometry. The transparent render type sets the render queue order of this object to 3000 and allows for running alpha test and z test calls.

Transparent Shader Properties	
1	Properties
2	_Color - normalized RGBA color value
3	_MainTex - Texture to be wrapped onto the object's surface
4	Tags
5	RenderType = Transparent (RenderQueue order 3000 and AlphaTest calls)

Figure 15: Transparent Shader Properties

The first pass of the transparent shader performs a multiplicative blending of the background pixels to allow the second additive pass to draw to a more uniform base (Figure 16). This pass inevitably dulls pixels drawn to by this object by a factor of 20%. Of note in this pass is the culling of back faces which halves the GPU render time of the object. Also the ZWrite tag is set to off so that depth testing is not performed and transparency can be added to the render frame. ZTest is set to NotEqual so that this object is drawn to the render frame even if it is behind solid geometry. The only time the object will not be drawn is if it occupies the exact same space as another object since Unity can't decide which object should be drawn and the result flickers between the two objects. The Blend tag is set so that the existing pixel color on the render frame is multiplied by one minus the alpha of the fragment color. At all times, the fragment alpha is 0.2.

Transparent Shader Pass 1	
1	Cull Back Faces
2	ZWrite is Off
3	ZTest tests for NotEqual
4	Blend Zero OneMinusSrcAlpha
5	Start CG Program
6	Vertex Function
7	Output position = product of the model-view-projection matrix and the vertex position
8	Output texture = vertex texture coordinate position
9	Return output
10	Fragment Function
11	Texture color = color value of the input xy position on _MainTex
12	Fragment color = product of _Color and the texture color
13	Return fragment color
14	End CG Program

Figure 16: Transparent Shader Pass 1

The second pass of the transparent shader performs an additive blending of the object's geometry to the currently drawn render frame (Figure 17). The tags for face culling, ZWrite, and ZTest are the same as the previous pass. The Blend is set to multiply the fragment color value by the fragment alpha and add the result to the existing pixel color on the render frame. This results in an additive effect where the object geometry appears to have a transparency by which other objects are visible through it.

Transparent Shader Pass 2	
1	Cull Back Faces
2	ZWrite = Off
3	ZTest = NotEqual
4	Blend SrcAlpha One
5	Start CG Program
6	Vertex Function
7	Output position = product of the model-view-projection matrix and the vertex position
8	Output texture = vertex texture coordinate position
9	Return output
10	Fragment Function
11	Texture color = color value of the input xy position on _MainTex
12	Fragment color = product of _Color and the texture color
13	Return fragment color
14	End CG Program

Figure 17: Transparent Shader Pass 2

The hyper-object script updates the shader used by the materials of game objects in the game scene whenever players move their position along the fourth dimension. Once players move, the hyper-object script determines if the object is within their range of peripheral vision along the fourth dimension and if so, checks if the shader needs to be swapped for the transparent or opaque one.

If an object must transition from transparent to opaque, then a linear interpolation over time is applied to the alpha of the transparent shader and once that alpha is close to 1.0, the opaque shader is applied to the object material and the shader information is carried over.

If an object must transition from opaque to transparent, on the visual update call the transparent shader is swapped into the material and the shader values are carried over. Once the switch has been made, the alpha values linearly interpolate over time to the constant transparency alpha value.

If an object has a depth greater than zero, it is visible on multiple points of the fourth dimension, so the color value of the opaque shader must be linearly interpolated over time from one color to the next dependent on the player character's position. If an object is present at w value two and has a depth of one, then it will be visibly solid on w points two and three, green and cyan. If players are at w position two, the object will appear as transparent green. If players are at w position four, the object will appear as transparent cyan.

If an object is outside of the player's peripheral vision along the fourth dimension, the object's material will use the transparent shader with an alpha value of zero to make it seemingly invisible. When players move along the fourth dimension so that an object moves outside the peripheral vision range, then the color alpha value of the transparent shader is linearly interpolated over time from the transparency constant to a value of zero.

3 Core Game Zones

Our game incorporates a lot of real world objects, such as bonsai trees and Koi fish. The behaviors these objects have must be modified, and in some cases removed, to work with the fourth-dimensional mechanic while also making the gameplay interesting. The following sections will focus on the three core game zones - the bonsai trees, the fish pond and the gravel pit - and how we modified their real world behaviors to work in our fourth-dimensional game environment. These modifications should help these objects retain some real world traits while also helping achieve our experience goal. We want players to enjoy how objects move in the fourth dimension while also working on tending to the garden to appease the nature spirits.

3.1 Bonsai Trees

The bonsai trees (Figure 18) are one of the three main mechanics in the game. Shinto gardens will have bonsai trees of a variety of sizes and styles. “The ultimate goal of growing a Bonsai is to create a miniaturized but realistic representation of nature in the form of a tree” (*Definition and Meaning*). For our Tokyo garden, the bonsai trees will represent the Tokyo Tower and the Tokyo Sky Tree. The bonsai trees also add an artistic element to the game along with providing interesting gameplay. In the following sections we will discuss:

- Designing the Bonsai: Components and Health of the Trees
- Interacting with the Bonsai: Pruning and Disinfecting the Trees
- Implementing the Tree: The Steps of a Growth Cycle

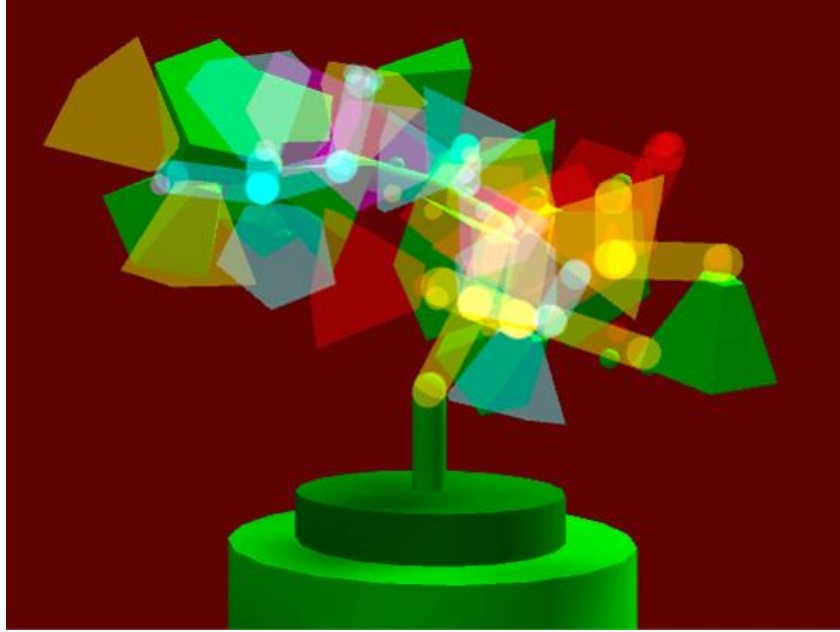


Figure 18: A Healthy Bonsai Tree

3.1.1 Bonsai Design

The bonsai trees are made up of a collection of branches, buds and leaves (Figure 19). Buds grow on branches and will grow into a new branch or leaves. Branches that grow from buds will have buds of their own already on them. Branches at the ends of the tree without any buds on them will grow new buds during the next growth cycle. Buds are what allow the tree to grow onto multiple points of the w axis (Figure 20). This creates areas where segments of the tree may appear to be floating or areas where the different colors of the branches blend. This provides both an artistic effect and provides an interesting challenge for players to try and tend to the trees in their own unique ways.

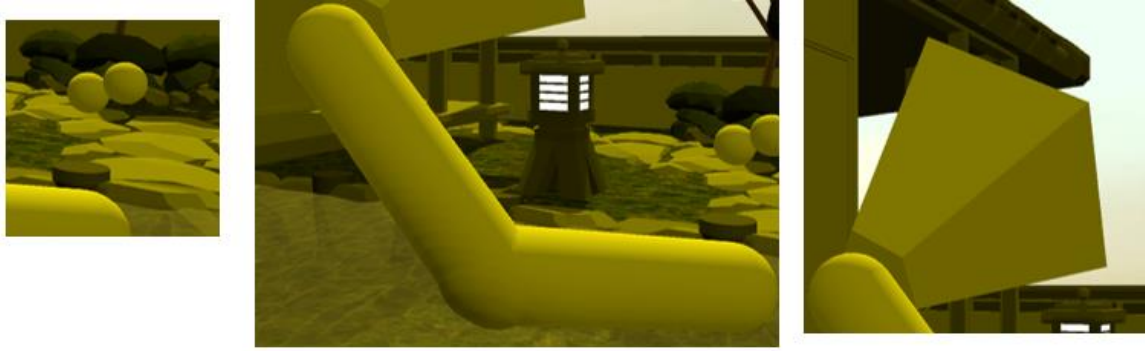
**Buds****Branches****Leaf**

Figure 19: Bonsai Tree Components



Figure 20: Bonsai Components on Different Points of the Fourth Dimension

At the beginning of the game, the trees will have many branches and leaves already on them. The bonsai trees grow exponentially, so having each tree initialize as a single branch would keep the tree problem free for the entire first half of the level. Initializing the trees with three long branches will generate more leaves and branches after the first two growth cycles and make the bonsai trees more closely resemble bonsai trees in the real world.

A growth cycle is when the tree grows new components and processes the status of each of its already existing components. The time between growth cycles is two minutes. Since the bonsai tree is the most complex game zone, we decided to make the time between cycles long to allow players to explore the other game zones without becoming overwhelmed by the bonsai.

A typical bonsai tree will grow forever until the day it dies. In our game, a single tree can have up to 60 leaves and 45 branches on it before it stops growing. This is to prevent the tree from growing to fill the entire game world or extend past the HTC Vive play space. This also limits the number of game object the tree will make to prevent it from overwhelming the system and decreasing performance.

Trees naturally try to grow taller and bigger, a trait known as “apical dominance” (*Pruning Bonsai*). This means that trees tend to focus growth on their higher and outer leaves and branches rather than the inner and lower ones. This eventually causes “the tree’s inner and lower branches will eventually die, while top branches grow out of proportion; two effects not desirable for the design of Bonsai trees” (*Pruning Bonsai*). We recreate this behavior in our game by having leaves detect if they would not be getting enough sunlight. This will eventually lead to leaves on the tree dying (Figure 21, Stage 2) as the game progresses. Dead leaves are the main antagonists in the bonsai game zone and what call players to interact with the trees.

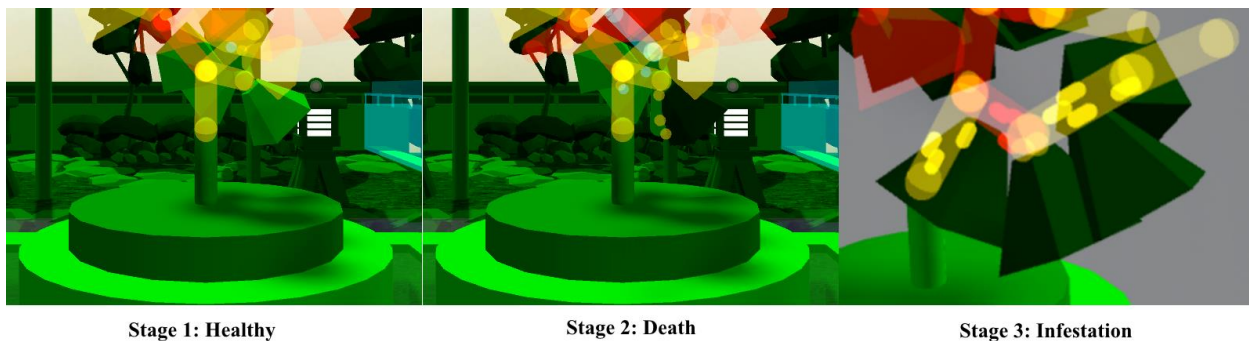


Figure 21: Stages of Bonsai Tree Infestation

If all the leaves on a branch are dead, the branch may become infested with pests (Figure 21, Stage 3) that will slowly spread to and kill other branches on the bonsai. Usually, bonsai will become infested due to poor watering and fertilization habits that cause the health of the tree to deteriorate. To prevent the bonsai from being too complicated, we define the health of the tree by

the number of dead leaves on branches. We wanted to force the infestation behavior into our tree as we wanted there to be repercussions if players neglected the trees. Before the infestation mechanic was implemented, we found that play testers would leave trees with dead leaves, as there were no negative drawbacks by doing so. They would then come to the trees last and cut off all the dead leaves to easily activate the shrine. By implementing a possibility of infestation, neglecting the tree will start to kill the branches, forcing players to cut them off and undo their progress up to that point.

3.1.2 Interacting with the Bonsai

There are many ways to care for a real bonsai tree - there are different fertilizers that can be used, ways you can wire the tree to control its growth, and you can even repot the tree. To keep the bonsai mechanic from becoming too complex, we decided to only implement trimming and removing infestations. These two behaviors are ideal as they provide interesting scenarios and gameplay and the tools that players use for these actions are similar in function and use.

Players trim the tree through the use of the shears tool (Figure 22). Holding the shears with the grip buttons and pressing the touchpad with the controller holding the shears will perform the snipping action. Any branch or leaf in between the blades of the shears during the snipping action will be cut from the tree and disappear. The only exception to this is the branch at the base of the tree as destroying it would delete the tree and ruin the game. In our initial design, only branches and leaves at the edges of the tree could be cut, however this would simply extend the time required to remove an unwanted branch. If a tree is overgrown with many edge branches, this process becomes very tedious and frustrating, so we allowed players to snip any branch on the tree.

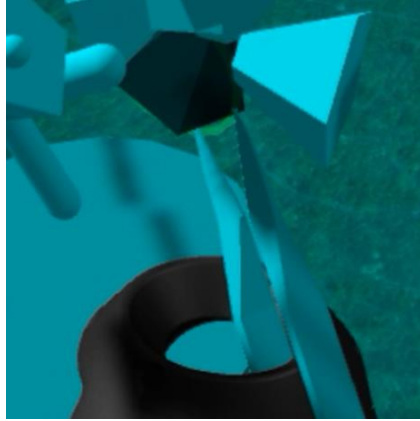


Figure 22: Trimming a Dead Bonsai Leaf with Shears

Players can remove an infestation from a tree by using the insecticide (Figure 23).

Similar to the shears, the can of insecticide can be held using the grip buttons and pressing the touchpad will cause the can emit particles from the nozzle at the top. The spray area is large, but only affects branches on the same w point as the insecticide. We wanted the process of disinfecting the tree to be simple while not being convenient. The purpose of the infestation mechanic was to punish players who neglected their tree by slowly killing the branches. Disinfecting the tree should not be an additional punishment, but also should not be so easy that it undermines the purpose of the infestation mechanic.

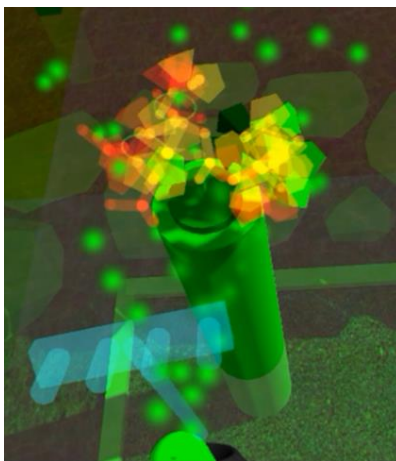


Figure 23: Removing a Bonsai Infestation with Insecticide

3.1.3 Bonsai Implementation

The bonsai tree is implemented in the game by connecting the branches, buds, and leaves in a hierarchal tree structure, sending growth signals up and down the tree, and registering tree data within the bonsai manager. The tree is implemented so that newly grown branches, buds and leaves are instantiated as children of their parent branches. The bonsai manager maintains a reference to the base parent branch of the entire tree and initiating its growth process every two minutes.

Once the tree's base branch receives a signal to process its growth cycle, it follows the growth sequence outlined in Figure 24. Each step of the growth sequence is performed on a separate frame to reduce the load on the game while a tree grows. The step labeled "Process Branches" is a recursive call to the branch growth cycle of each child branch. This results in a depth first growth cycle splitting each step of the process onto separate frame updates. The growth cycle cannot be initiated again until the tree has run through every step of the growth cycle for each component on the tree.

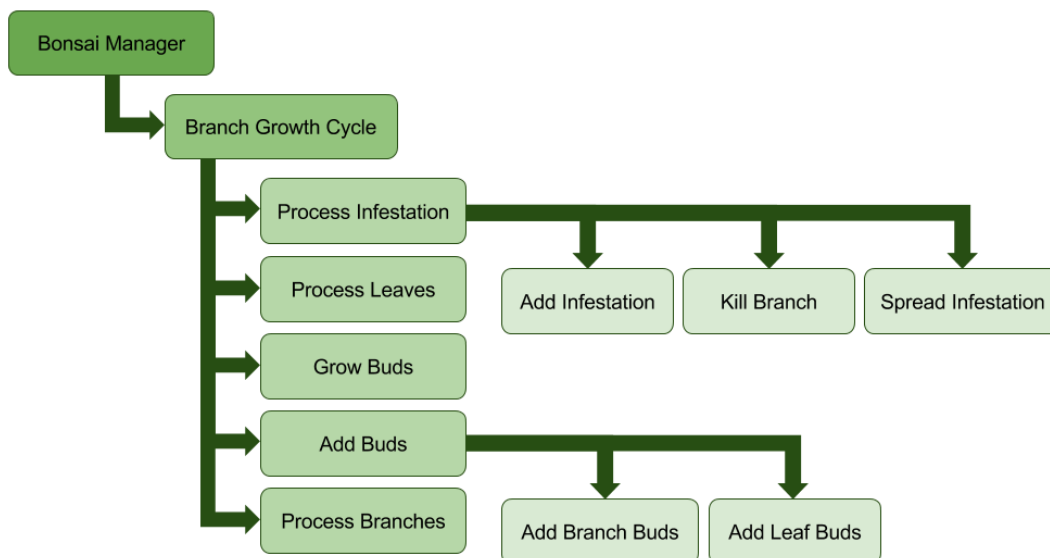


Figure 24: Bonsai Growth Cycle

The first step of the growth cycle processes the newest state of the infestation on a branch (Figure 25). At any time, the branch can be either clean, infested or dead. If the branch is clean, it checks its children leaves to see if they are all dead. If they are, the branch then initiates the timer for adding infestation to the branch. If the timer for reaching infestation has run out, infestation is added to the branch by instantiating the bonsai bug object on the infested branch. It will then attempt to spread the infestation to adjacent branches. Additionally, it checks to see if the timer till death has run out and kills the branch and its living children leaves if it has. Once dead and infested, a new timer is initiated to when the infestation is removed from the branch. Once the timer has run out, the infestation disappears from the branch, and a cooldown timer is initiated before the branch can be infected again. This cooldown timer ensures that the infestation does not perpetually spread to dead branches and consume the tree forever.

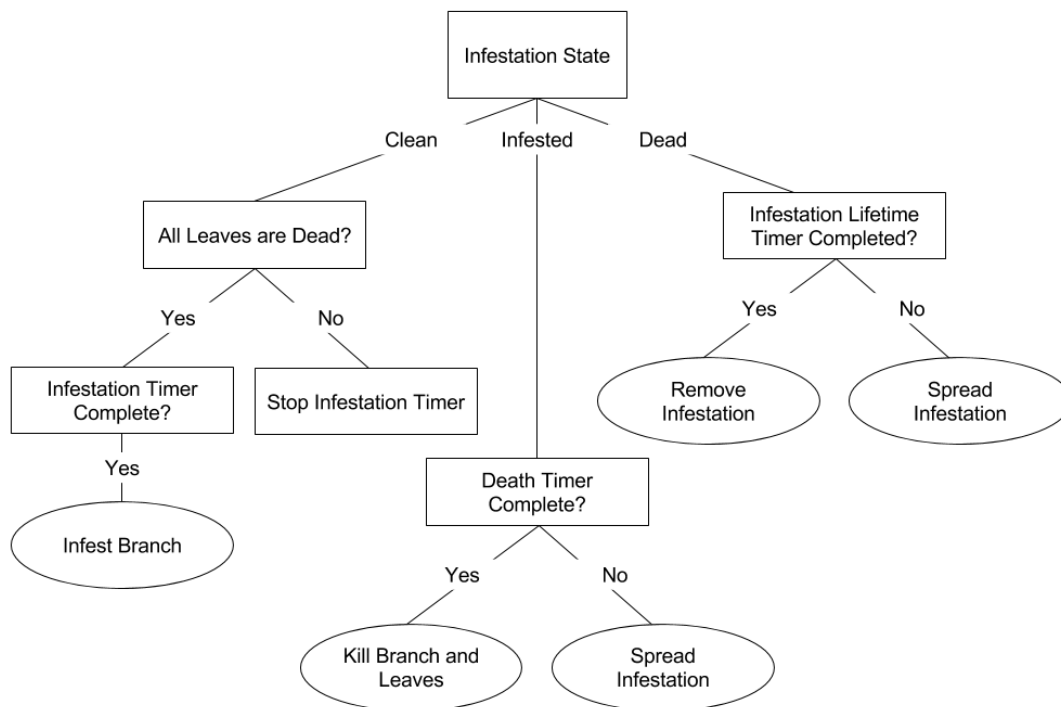


Figure 25: Infestation Decision Tree

The second step of the growth cycle process all of the branch's children leaves (Figure 26). A leaf will remain alive for the current growth cycle if it satisfies one of the following requirements:

1. It is on a branch with no children branches.
2. a) It is on a branch that is higher than its children branches.
b) It has less than two leaves above it.
3. a) It is facing above the horizon.
b) It has less than two leaves above it.

Requirements 2 and 3 require both a and b to be true. If one of these requirements is not met, the leaf will die and dull in color. On its death, the leaf will signal the bonsai manager so that the number of dead leaves on the tree is tracked.

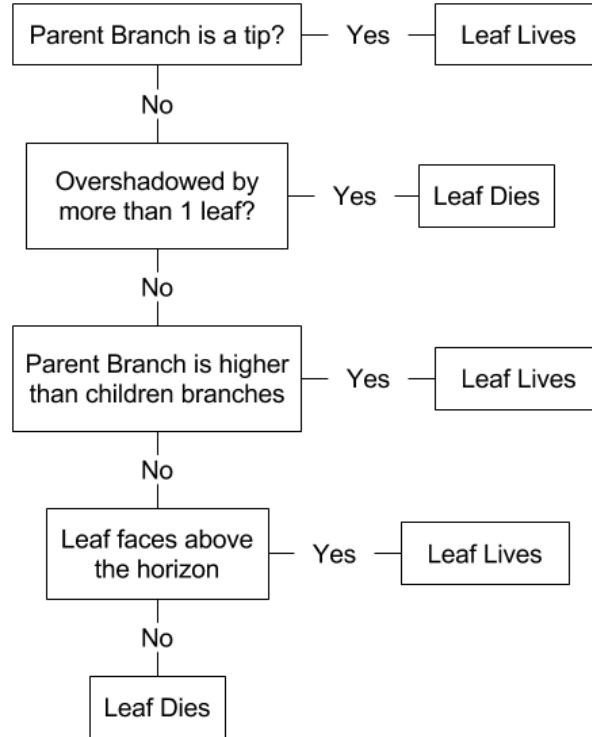


Figure 26: Leaf Death Decision Tree

The third step of the growth cycle processes the growth cycle of all existing children buds. The children buds are assigned their type (branch bud or leaf bud) on instantiation. They simply create new tree components, attach them to their parent branch and pass them a reference to the bonsai manager and its position on the w axis. When a new component has been created, a coroutine responsible for extending the object from its initial position is called. New branches extend outward from the center of their parent's tip, new buds expand from a point on the surface of the parent branch, and new leaves expand from a point on the surface of the parent branch.

The fourth step of the growth cycle adds new branch and leaf buds. New children branch buds can be grown as long as the following conditions are met:

1. There are less than 45 existing branches on the tree.
2. The branch does not currently have three children branches.
3. Any of these three conditions are met:
 - a. The branch's depth is two or greater (the base branch being at a depth of zero) and the modulo of the branch age and branch bud growth cycle is equal to zero
 - b. The branch's depth is less than two and the modulo of the branch age and the branch bud growth cycle is equal to zero
 - c. The branch's depth is less than two and all of its children branches were removed between the last growth cycle and the current growth cycle

The requirements listed in 3 account for special cases that ensure the tree grows as expected. Case A ensures that if the current branch is at or beyond the minimum depth at which leaves can grow, new branch buds will be grown on the growth cycle after new leaf buds have grown. Case B ensures that if the current branch is below the minimum depth for growing leaves, then new branch buds will grow on the same cycle that the current branch has grown. In practice,

this results in new branches already populated by branch buds. Case C ensures that if players snip the tree right above the base or below the minimum leaf line, then the tree will immediately start growing at the next growth cycle. This is necessary so that new players do not ruin their playthrough by accidentally snipping the tree at the base. Additionally, the branch bud growth cycle is relative to the depth of the branch, meaning that branches at lower depths will grow new branch buds at a lower rate than branches at further depths.

If a new branch bud is to be grown on a growth cycle, then it is instantiated as a child of the branch, placed at the center of the tip sphere of the branch, assigned a random rotation within 90 degrees of the vector pointing out from the tip and altered to have the visual of the bud moved forward to make it appear on the surface of the branch. All of this is done to give the appearance that a new bud has grown on the branch and ensure that when the bud transforms into a branch it is grown in the right direction and position.

New children leaf buds are grown if all of the following conditions are met:

1. There are less than 60 existing leaves on the tree.
2. There are less than five existing leaves on the current branch.
3. The depth of the current branch is two or greater (the base branch having a depth of zero).
4. The modulo of the branch age and the leaf bud growth cycle is equal to zero.

If all of these conditions are met, a new leaf bud will be instantiated, parented to the branch, and placed in the correct position and rotation. Determining the position of the leaf bud is a bit more complicated since it can be grown anywhere on the surface of the central cylinder or tip sphere of the branch visual. The algorithm for placing the leaf bud first picks a random y position between the central cylinder's base to the tip of the branch. If this y position is on the cylinder a random point is selected on the circumference of the circular cross section of the

cylinder at that y position. If this y position is on the surface of the tip sphere, then a random point is selected on the circumference of the chosen circular cross-section of the sphere.

This location is then compared to the existing positions and rotations of other leaves to determine if the new leaf will have an angle greater than 45 degrees between its facing direction and that of older child leaves. If the offset is not great enough, a new point is randomly chosen until a maximum number of 15 attempts is made in which case the bud is discarded. The position is then stored for future checks and the bud is spawned at the position with a rotation pointing away from the center of the cylinder or sphere.

The final, fifth, step recursively calls the function for processing the current growth cycle in each of the current branch's children branches. Once every child branch has signaled it has completed its growth cycle, the branch will send a completion signal to its parent.

3.2 Koi Fish

The koi fish (Figure 27) are another of the three core game zones. The addition of fish and pools of water is what classifies our garden as a Shinto garden and not a Zen garden. Zen gardens are a type of dry garden where bodies of water are represented in gravel or sand pits (*Japanese Gardens*). In the Tokyo contract, we have four fish pools. Three of the pools represent Tokyo Bay, the Arakawa River, and the Sumida River. The fish represent the people and vehicles that move around Tokyo. The 3D model for the Koi fish was acquired for free from ShareCG.com (*Brownson, B. 2010*). In the following sections we will discuss:

- Designing the Koi Fish: Hunger States and Acquiring Fish
- Interacting with the Koi Fish: Relocating and Feeding
- Implementing the Koi Fish: State Behaviors, the Feed Cycle and Finding a Target



Figure 27: Koi Swimming in Fish Pool

3.2.1 Fish Design

The fish in our game have two main behaviors: wandering and hunting. These behaviors depend on the state of the fish. The states of a fish (happy, hungry, hunting and dead) are discussed in more detail in Section 3.2.3. It takes the fish a set amount of time before it changes states. When a happy fish gets hungry, it will dull in color and move slower (Figure 28). Hungry fish are notably different from happy fish, so players should be able to identify and feed the fish that are hungry. When fish are hungry, they will wait a period of time before eating nearby fish. This buffer time gives players the opportunity to feed the fish before they eat one another, as players may be focused on another section of the garden during the state change. However, if the fish are not fed, they will eat the smallest fish closest to them, prioritizing size over distance. Usually, “Koi carp are unlikely to eat other large fish” (*Do Koi Eat*); however, if a hunting fish

cannot find another fish smaller than its own size it will eat the closest fish of equal size. Without this modification to normal Koi behavior, players would not have to pay attention to the fish pools during the game. If a hunting fish cannot find a suitable target before the next hunger cycle, it will die and vanish from the garden.



Figure 28: A Hungry Fish (left) and a Happy Fish (right)

Fish spawn into the garden in the happy state. Once a fish has spawned, it will initialize its feeding cycle clock. The time it takes for the fish to become hungry for the first time will have an offset to make feeding cycles less predictable. If the feeding cycle had the fish get hungry around the same time, feeding the fish would be less challenging. This offset adds a level of variance to the fish feeding mechanic and leaves room for skilled players to align all of the fish in the garden on the same feeding cycle. Only the initial feeding cycle offset is random; the time between feeding cycles is a fixed rate for every fish. With every revolution of the feeding cycle, a fish will deteriorate in hunger until it dies or eats. After play testing we set the time between feeding cycles to be 60 seconds. More detail about what happens during a feeding cycle can be found in Section 3.2.3.

3.2.2 Interacting with the Fish

Players can grab and move fish by pressing the grip buttons on a Vive controller while the controller is touching a fish (Figure 29). Being able to move the fish is important as the fish do not enter the garden already in a fish pool. Instead, the fish are spawned in a special pool known as the reservoir (Figure 30) separate from the real fish pools. We felt that this method was the best way to introduce the fish mechanic to players over having fish already in the garden.

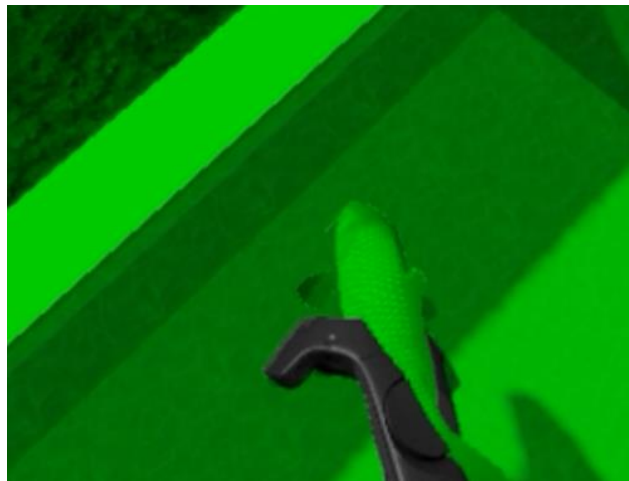


Figure 29: Grabbing a Fish in the Reservoir Pool

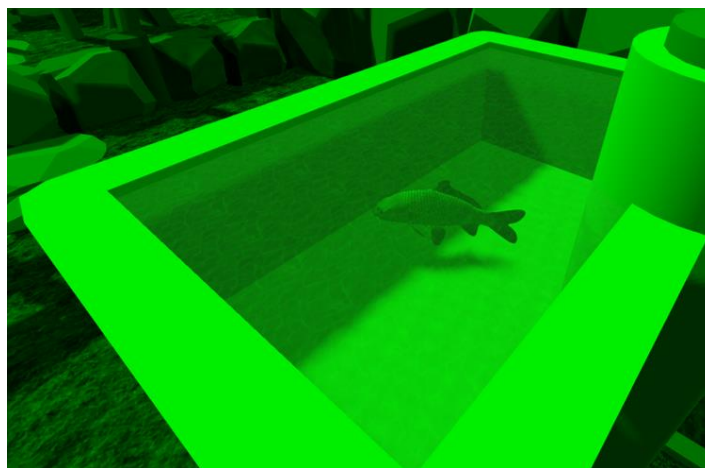


Figure 30: Fish in the Reservoir Pool

The reservoir pool controls the rate at which fish can enter the garden. The reservoir will continue spawning fish until the maximum limit of 40 fish have spawned. This causes players to build up the garden fish pools over the course of the contract time instead of immediately at the start of the game. The reservoir will spawn a new fish 15 seconds after it becomes empty again.

Fish can be fed using the fish food can tool. Players can grab and move the can by pressing and holding the grip buttons on the Vive controller while the controller is colliding with the can. Turning the can upside down will cause fish food pellets to spawn from the top of the can and fall to the ground (Figure 31). Food will spawn a maximum of 30 food pellets on a single w point while held and upside down. Food can be poured on any point of the fourth dimension, but can only be poured on one point at a time. We debated allowing players to pour food on multiple points of the fourth dimension as a reward for activating more shrines, but decided against the idea as we wanted to constantly keep players moving along the fourth dimension.

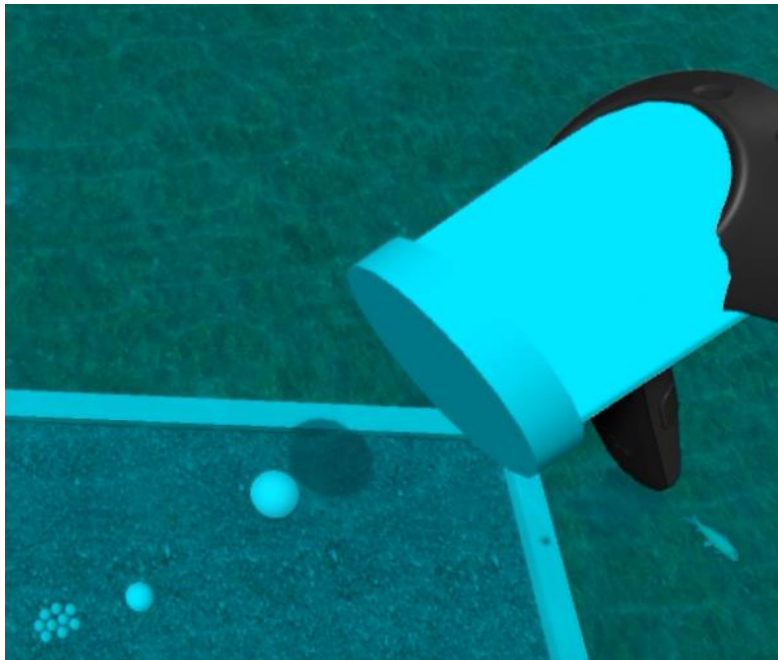


Figure 31: Pouring Fish Food into a Fish Pool

Fish can only be fed when they are in either the hungry or hunting state. Although happy Koi could eat food, having to strategically place food in the fish pool would take time away from tending to the other parts of the garden. When fish detect that food has entered the water, they will cautiously move towards the closest pellet. Once they get close enough to the pellet, they will move swiftly towards it and eat it. From our observation of Koi on Awaji Island, we found that Koi will clump in an area where food is present (Figure 32). The Koi would also eat more than one pellet of food and still linger in the area after eating. In an early implementation we found that fish on the edge of the clump could never reach the food, so we modified these behaviors to prevent the fish from clumping together in one spot of the pool when the food was added. Now a fish will be able to quickly eat up a food pellet before the other fish get in range. Food will disappear after 30 seconds if not eaten by a fish. This prevents players from pouring as much food as possible into a pool and neglecting it for the rest of the game.

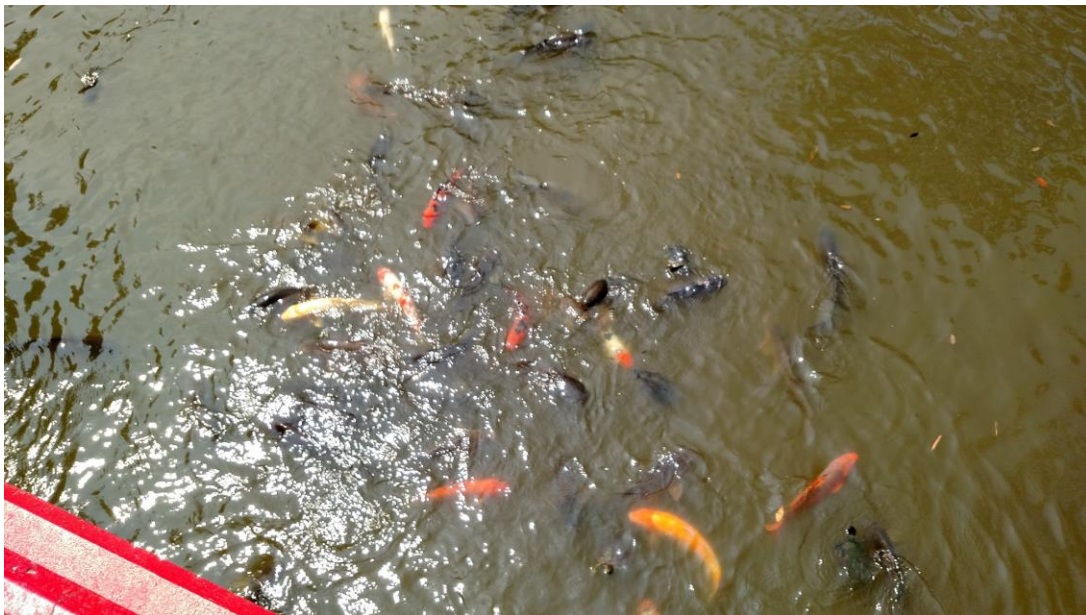


Figure 32: Koi Feeding Behavior Reference

3.2.3 Fish Implementation

Fish run on a clock called the feeding cycle (Figure 33). Every feeding cycle the hunger state of the fish changes. Fish have four hunger states:

- Happy
- Hungry
- Hunting
- Dead

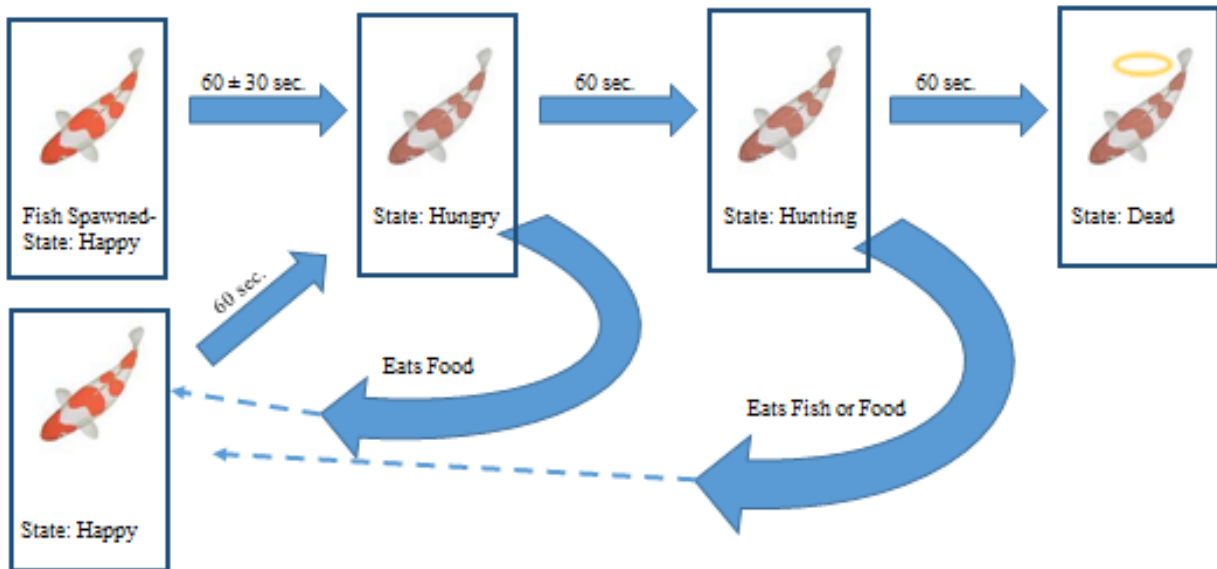


Figure 33: Fish Feeding Cycle

Each hunger state alters the behaviors of a fish: how fast it moves, where it moves, what alerts it, and how vibrant its color is. If the fish is taken out of water, then it will not take on any behaviors. When the fish are in either the happy or hungry states, they will rotate to face a random location in the pool and move forward at a constant speed. Fish in the hungry or hunting states will change their target if there are food pellets on the same w point as them. Hunting fish that are already moving towards a fish they can eat will not change their target if food is added to the pool. A fish in the dead state will be removed from the game by the fish manager. This state

is used as a buffer between the frame a fish dies and the next frame where it is removed from the scene.

The fish manager contains the list of all fish and food pellets in the game world. When the reservoir spawns new fish or a fish asks to be removed after death, it must receive permission from the fish manager. When a fish enters the hunting state it requests a target fish to eat from the fish manager using the RequestTarget function (Figure 34). This method can also be used to target food for hungry and hunting fish.

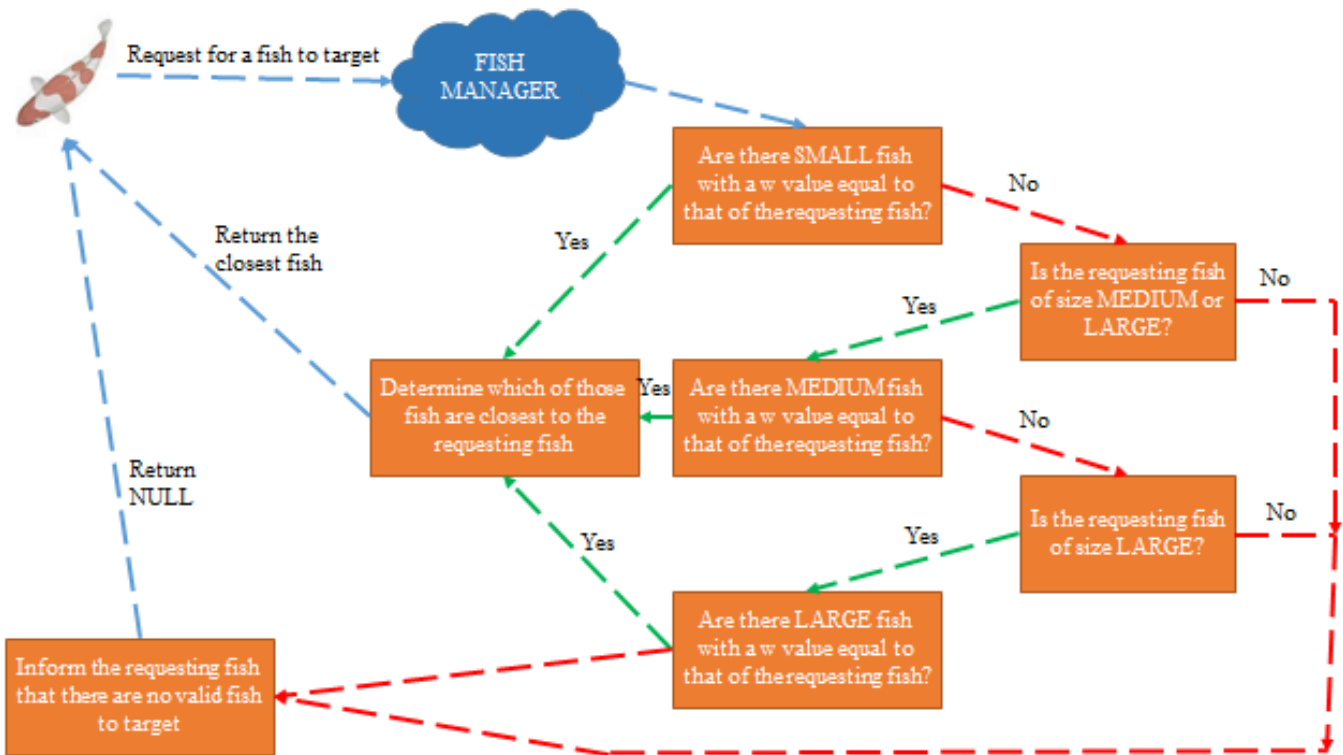


Figure 34: RequestTarget Sequence

The fish manager will return either a game object or a value of null to the requesting fish. The game object may be either another fish or food depending on how the function was called in the fish manager. Fish will move towards the given target until they collide with it or the target becomes invalid. This can happen if the target disappears, is moved to another w position or changes to a size larger than the fish targeting it. If a hunting fish manages to reach its target it

will ask the fish manager for permission to consume it. The fish manager will make sure the action is valid and, in the case of a tie between fish, have the fish that has been in the game the longest eat the other fish. Eating sends a request to the fish manager to remove the target and increases the number of food the fish has eaten. After eating two objects (any combination of fish or food) the fish will increase one size unless it is already the large size.

3.3 Gravel Pit

The gravel pit is the last of the three main mechanics in the game we designed. This feature was important to include from a cultural standpoint, because in Shinto gardens, “large stones are worshipped as Kami” and gravel was “used to designate sacred ground” (*Garden Elements*). In addition, as players rake more and more designs into each of the gravel pits, the pits will give them a visual effect that is reminiscent of the streets and roads that run through the city of Tokyo. This supplements other game objects like the fish, which remind players of vehicles and people moving around the city.

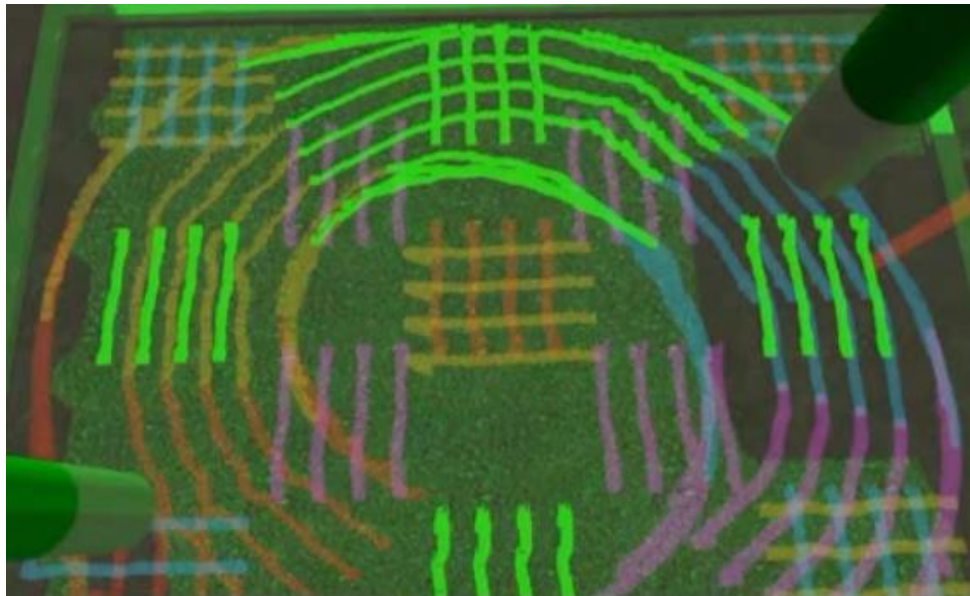


Figure 35: Player Designed Gravel Pit

3.3.1 Gravel Pit Design

The gravel pit is made up of one plane to hold the gravel texture (Figure 36), and then five “alpha” planes (Figure 35) on which players will rake patterns. These are transparent and separate from the gravel texture plane. When players switch w dimensions, the gravel pit can change color, and they can draw to the corresponding alpha plane as opposed to drawing everything on the same gravel plane.

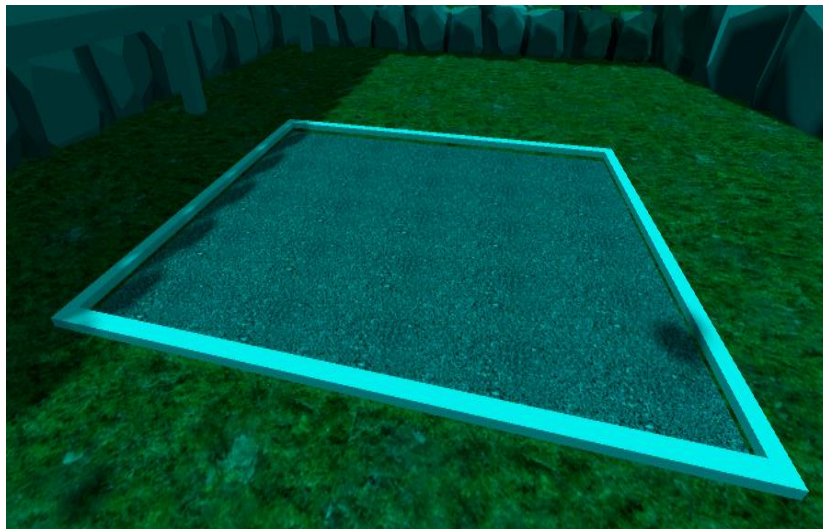


Figure 36: Gravel Pit

There are also game objects known as pattern lines (Figure 37), which match the color of the current gravel pit and appear on the surface of each pit while players are holding the rake. For each gravel pit, these serve as a visual representation of a pattern that players will have to rake in that pit in order to activate the gravel pit shrine. To make these patterns, players can match the pattern lines by raking over them in the pits. Invisible game objects known as nodes are also present. These mark the start and end of the pattern lines, and are also placed at the beginning and end of lines made by players with the rake. Both the pattern line and node objects are used in pattern recognition, and as such will be discussed later in Section 3.3.4.

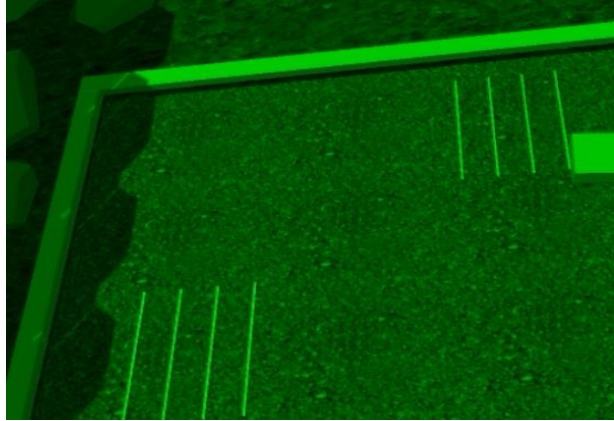


Figure 37: In-Game Pattern Lines

The rake tool is modeled after the style of rake shown in Figure 38. Similar to a real-life rake, players will hold the rake by the tool's handle, and they will push and drag each of the four tines at the end of the rake (Figure 39) around to draw their patterns in the pit.



Figure 38: Wooden Rock Garden Rake Reference

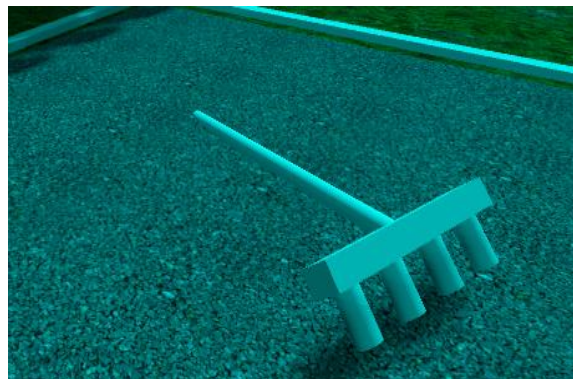


Figure 39: Rake

3.3.2 Interacting with the Gravel Pit

Players interact with each of the gravel pits using the Vive controller to pick up the rake tool provided to them, which originates in the cyan dimension. To “draw” lines and design patterns in a gravel pit, they would then just need to place the tines of the rake tool into the current pit they are on and maneuver the tool in whichever way they desire using the Vive controller. To finish a line or a design that they were making in a pit, players would then move the rake tool up, making sure that the tines of the tool were not touching the pit anymore.

When players work on matching the patterns in the gravel pits, they are able to start raking from either end of a given pattern line. The only factor players need to worry about is the orientation of the lines, so they will need to either rake straight horizontally or vertically along the pattern lines. Walking in the gravel pits – which does not leave footprints – also does not affect what is raked inside, so players are free to walk around it without destroying the patterns they raked.

In our original design, we wanted to implement a wind feature for the gravel pits. When the wind blew, it would clear out or reset what was raked by players in a third of a random gravel pit, which could possibly deactivate the gravel pit shrine. This feature would keep players on their toes, as they would have to check the gravel pits occasionally to make sure their patterns and rakings were not blown away and that the gravel pit shrine stayed active. This feature was eventually taken out from the game after testing due to some negative effects it had on the game's performance.

3.3.3 Gravel Pit Implementation

In the gravel pit, players will be able to use the rake tool given to them to make designs and “draw” lines, circle, and zigzags among other things through raking motions. In order to accomplish this task, two types of planes are used to construct the pit - one plane to hold the pit's gravel texture, and another transparent alpha plane directly on top of this gravel texture for each w dimension that players are raking and making designs on.

When players have the tines of the rake tool come in contact and collide with the current gravel pit – and more specifically, the current alpha plane based on what w dimension players are on – an 11x11 box of pixels with color corresponding to the current w dimension will be drawn at the location of any of the tines that are in contact with the pit. This means that if the rake is parallel to the pit when the collision occurs, all four of the rake's tines will draw to the transparent alpha plane. On the other hand, if the rake is rotated to the side so that only one tine is touching the pit, a single line will be drawn on the plane. This feature allows players to have more control over what they are capable of drawing within the pit, and can choose to rake with all of the tines or just one or two if they wish. As players move the rake tool along the gravel pit while the tines are colliding, every update frame a new 11x11 box will be drawn at the tines' location, and only drawing to the parts of the box that have not been raked on yet.

An example of how this would work is shown in Figure 40. If on Update Frame 1 an 11x11 box of pixels is drawn on the current alpha plane, and on Update Frame 2 the rake has been moved forward 9 pixels, then the new box drawn would only draw the first nine rows from the top of the box to the plane (shown by the black-bordered green squares). This occurs because the bottom two rows of the new box (shown by the green-bordered red boxes) were already raked and drawn on Update Frame 1, and so those rows do not need to be raked and drawn again

on Update Frame 2 with the new box. The blue “X” marks the location of center of the given tine for each Update Frame.

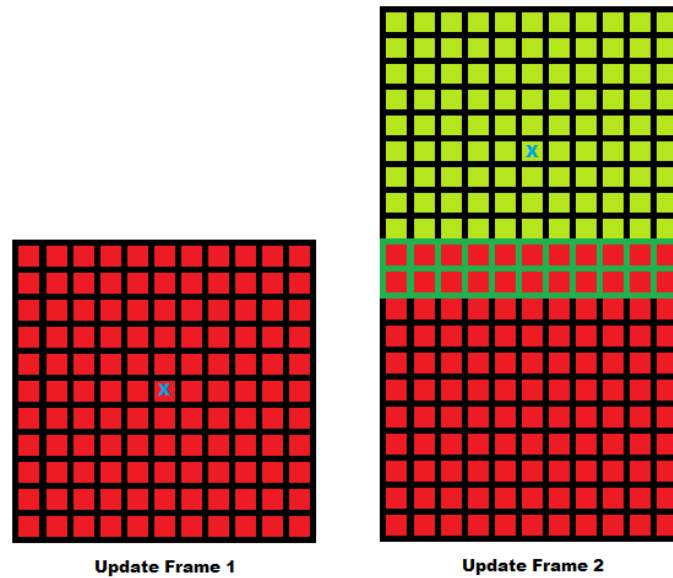


Figure 40: Tine Drawing by Pixel Boxes

3.3.4 Pattern Recognition

Our other technical challenge with making *The Conduit's Garden* game was getting over the obstacle of comparing pattern lines – made for players to match by raking – with what players actually raked inside the gravel pit. To overcome this challenge, we needed to design an algorithm that recognized what patterns were being made by players as they drew lines on the gravel pits using the rake tool.

3.3.4.1 Design Requirements

In order to match our design for what we intended to be able to do in the gravel pit, we needed to implement a method of pattern recognition that met a few specific requirements. First of all, the pattern recognition method needed to keep track of what lines were a part of the pre-

designed patterns for the Tokyo contract, and the shape and position of each line made by players while raking. This was done through the use of “nodes,” which will be discussed later in Section 3.3.4.3. The other part to this pattern recognition that we wanted implemented was allowing players to have some freedom to rake what they wished. This was made possible through our implementation method, as all lines made by players are matched to the pre-made pattern for the contract. As such, as long as those pre-made lines are matched, any excess lines made by players would not affect the pattern recognition and the activation of the gravel shrine.

3.3.4.2 Previous Research

Prior to coding our final algorithm for pattern recognition in the gravel pit, some research was done to see if others had tried solutions to solve our problem of pattern recognition or a similar issue. While we did not find any exact solutions to our version of pattern recognition, we did find some people who had discovered methods of constructing code that ended up helping us with an alternative pattern recognition that we did not use – this will be discussed later in Section 3.3.4.3 on alternative approaches – or guided us towards our final recognition implementation method. For instance, there were a number of posts on the Unity3D Answers forum that we found help. One user posted about comparing two bitmaps, where he was looking to “trace a line that’s drawn on a bitmap” and needed to “check how good you did this” (*Compare 2 Bitmaps*) at the end of his game. His code for accomplishing this task was helpful in implementing our first alternate approach to pattern recognition of comparing the pixels of the images of the gravel pit and what players have raked. Another user was looking to make a Tetris-style game, and was trying to figure out how to check if a pattern was reached when “certain colored balls touched each other in a specific pattern” (*Pattern Detection*). Someone replied to this and mentioned that “For any given cell (i, j) , you can determine if there's a corresponding L shape by checking the

colors of cells $(i, j-1)$, $(i, j-2)$, and $(i+1, j-2)$ ” (*Pattern Detection*). This comment inspired us to try implementing in our “node” approach having previous and/or next nodes to keep track of what nodes were in a specific line in a pattern and their order. One last user posted on the Unity3D Answers forum about having a player draw runic symbols to cast spells, and wanting to know “the most effective way to work out if the user has drawn a square / swirl / triangle” (*Gesture Recognition, Determining Shapes*). Another user commented that the poster could “store the angles you end up with in-order in an array or list (instead of the vertex positions - although you can start with those and work out where and what your angles are later), then compare with pre-stored lists of angles that define your shapes (with some margin for error).” (*Gesture Recognition, Determining Shapes*). This was extremely helpful, as we modified this suggestion in our final pattern recognition implementation to use vectors between “nodes” as part of a way of recognition. We will go into more detail in the following section on how it worked, but we also looked into EyeOpen, an “image processing library in C#” (*Similar Image Finder*) as a possible option for recognizing patterns.

3.3.4.3 Alternative Approaches

Before deciding on and implementing our method of recognition, we went through a couple of different approaches to deal with this challenge. The first approach we tried to use was by comparing the pixels of two different images – the original pattern we designed, and the current state of the gravel pit. We would go through every pixel in one image and check if they matched the color of the same pixel in the other image. While this did technically work and produced a percent similarity value to determine if the pattern players have drawn on the pit were right, the issue we had with this implementation method was its accuracy.

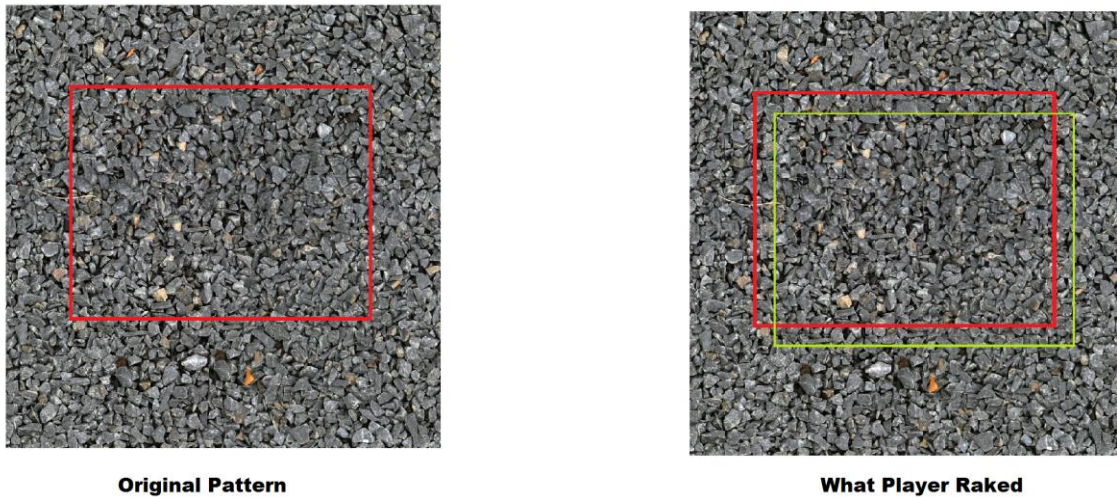


Figure 41: Comparison between Original and Player Made Pattern

An example of this accuracy issue is shown in Figure 41, where players might be asked to rake a design into the pit like the red box in the Original Pattern picture. However, if players actually raked the design 20 pixels to the right of the red box and 20 pixels below where they were asked to rake (see the green box in the What Player Raked picture), the algorithm used in this approach would be way off because many of the pixels would not match. This was an issue for us because players would draw something very similar to what was asked but would get penalized for it because it was not in the correct position.

The second approach we tried was using the pre-existing EyeOpen library. To implement this method, the original design image and the current image of the gravel pit would be taken in and put into a pre-made function from the library to compare the two images “using RGB projections (horizontal and vertical)” (*Similar Image Finder*). This would then determine the percentage of similarity between the two images and return that value. While this seemed promising and would have most likely worked, the issue with this implementation method was actually its compatibility with the Unity3D engine – some of the file types used in the code for

the comparison were not supported by Unity3D, therefore, the library could not be used for this project.

Our last alternate approach, which our actual implementation method is essentially an extension of, was to use “nodes” to keep track of the patterns that players have made with the rake, as well as the pre-designed pattern. On the pre-made pattern, lines and curves would be broken up into sections, with empty “node” game objects placed at intervals along them. Similarly, when players begin raking on the gravel pit, nodes would be placed at specific intervals at the current location of the rake’s tines. On certain update frames, the game would go through each node from the pre-designed pattern and see if there was a node made by players raking that was within a small radius of the node currently being checked. If there was a node made by players in that radius, a similarity score would increase. We discuss the issues with this design later in Section 3.3.4.5, when we talk about the challenges in programming our current pattern recognition method.

3.3.4.4 Implementation

We decided to implement pattern recognition for our gravel pit through the use of “nodes.” When players make the tines of the rake tool collide with the gravel pit and starts drawing a pattern with said rake, a node will be placed at the start of each tine collision. Then when players are done raking a line or pattern and lifts the tines out of the gravel pit, a final end node will be placed at the location where each tine left the pit to mark the end of the line or pattern. Making up the original design players are trying to replicate while raking – shown to players through the pattern lines mentioned in Section 3.3.1 and displayed in Figure 37 and Figure 42 – each line in the pre-made pattern will be marked by a start node and an end node placed at the line’s start and end points. When players are finished raking a line (thus making an

end node), the current nodes made by players from raking will be checked against the pre-made nodes from the original design.

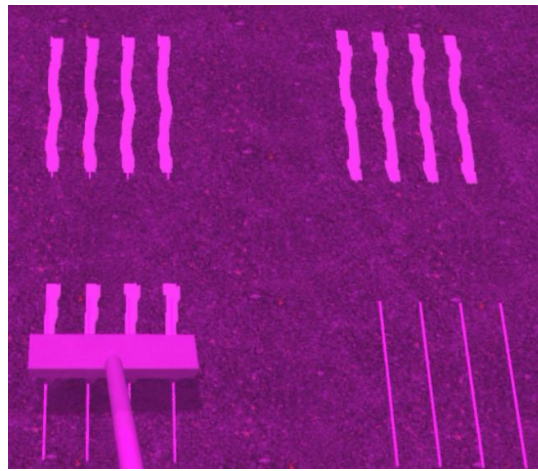


Figure 42: The Player in the Process of Raking, Trying to Match the Pattern Lines

The pattern recognition algorithm determines the vectors between the start and end node for each line in the original design as well as the start and end nodes of each line and pattern raked by players. The algorithm then iterates through the group of lines in the original design (Figure 43, line 3) and finds the best line from the group made by players which matches it. This is done by checking if the x and z position of the player-made line's start node is within range of the x and z positions of the start or end node from the current pattern line (Figure 43, lines 10 and 13). If these components are within range and the player-made line's x and z vectors are close enough to the pattern line's x and z vectors (Figure 43, lines 11 and 14), then that line is given a score of one (Figure 43, lines 12 and 15). This sequence will occur for each line drawn by players, and the best scoring player-made line will add its score to a total for the entire pattern (Figure 43, line 18). At this point, after going through every line in the pre-made design, the total score for the pattern will be divided by the number of lines drawn by players (Figure 43, line 19). If after going through all of this, the player-made lines total score is 1 (meaning their lines are similar enough to the original pattern design), then the gravel pit's shrine will partially activate

for that dimension (fully activating only if players have also raked a tenth or more of the pit's area).

Pattern Recognition Pseudocode	
1	patternLinesOnDimension = number of pattern lines on the current dimension
2	similarity = 0.0f
3	For each line in the original designed pattern
4	If the current pattern line's w dimension is the same as the current alpha plane's
5	highestCheckValue = 0.0f
6	For each line drawn by the player
7	thisCheckValue = 0.0f
8	If the current player line's w dimension is the same as the current alpha plane's
9	rangeCheck = 0.1f
10	If drawnStartX is between patternStartX +/- rangeCheck and drawnStartZ is between pattternStartZ +/- rangeCheck
11	If difference between patternVectorX and drawnVectorX <= rangeCheck and difference between patternVectorZ and drawnVectorZ <= rangeCheck
12	Increment thisCheckValue by 1.0f
13	If drawnStartX is between patternEndX +/- rangeCheck and drawnStartZ is between pattternEndZ +/- rangeCheck
14	If difference between patternVectorX and drawnVectorX <= rangeCheck and difference between patternVectorZ and drawnVectorZ <= rangeCheck
15	Increment thisCheckValue by 1.0f
16	If thisCheckValue > highestCheckValue
17	highestCheckValue = thisCheckValue
18	Increment similarity by highestCheckValue
19	percent = similarity/patternLinesOnDimension
20	Return percent

Figure 43: Pattern Recognition Pseudocode

This implementation method is convenient and works well for our project over other ways of implementing pattern recognition for a number of reasons. One of the main reasons is because using these “nodes” simplifies the process of recognizing patterns; instead of having to spend more CPU processing power and time going through every pixel in two images and comparing them, for instance, we only have to check a significantly smaller number of nodes and

see if they make a similar pattern to the one that was originally designed. In terms of physically designing the original pattern to compare against what players made, it is also easier because we don't have to worry about making sure every line we draw on the pattern matches the size of the lines drawn by the rake. Instead, we just have to make sure to place enough nodes in the original design so that there is enough skill and time needed raking in the pit to get a similar pattern to the original. The final reason why it works well for our project and with the gravel pit is because it is more lenient regarding mistakes made by players. Theoretically, players should be taking their time with raking so that they can have a better match to what they were being asked to rake, but we also had to factor into the equation the sensitivity of the rake and players' control of the rake with their hands. When trying to match the pattern given to them while raking, players could, for example, accidentally start the line too early if their controller jittered. Normally, this might be an issue with most other implementation methods, but with ours, making such a mistake does not drastically affect the score of players' lines as much. This is because part of the algorithm depends on the x and z position of the nodes in the lines made by players. If those nodes are within a close range of x or z value of the line from the original design pattern, then players will still get credit for matching the pattern.

3.3.4.5 Programming Challenges

Most of the issues with programming our pattern recognition algorithm was dealing with edge cases in players raking designs. For example, with our previous "node" implementation, the pattern recognition was solely based on the position of the nodes that players made through raking and comparing those positions to the positions of the nodes in our pre-designed pattern. The issue with only working with positions is that players could theoretically just place their rake at the positions of the nodes in the pre-made design and still get credit for similarity even though

they didn't draw any lines. We did not test this issue with any players, but we felt that this would be fairly hard for players to accomplish, since players don't know where the pre-made nodes are. Despite the fact that it may not happen often, it would be bad programming design to let this glitch persist in the game, and players might be confused if they raked a pattern with a line and got the same credit as placing the rake only on the nodes.

Another challenge with programming the pattern recognition was an issue with making circles as part of pattern recognition. While this would have been interesting to have as part of the designs for the gravel pit, it proved to be fairly difficult to implement in theory with our current design. The issue with circles was that players could start at any point along the circle when attempting to draw it. This conflicts with our current implementation, as each pre-made line needs a start and an end point to make a vector from and compare to the player-made lines - if players started at the opposite side of the circle, they wouldn't get credit for making it even though it matched the design.

3.3.4.6 Evaluation

In terms of evaluating the pattern recognition algorithm as a whole, the issue with it is that the accuracy of the algorithm is dependent on how players rake their lines in the gravel pit. To clarify, the algorithm works as intended and successfully when players makes a continuous line from the start node to the end node of a pattern line from one of the pre-made patterns. The flaw in this implementation is if players split the line they raked up into two sections - say they raked halfway through a line, picked up the rake, then placed it back down and finishes the rest of the line. Unfortunately, the algorithm does not account for a pattern line being matched by using two lines. As such, players would not get credit for matching that pattern line, even though to them, it would look like one line was made.

4 Additional Game Content

Outside of the three core game zones, we have many other objects that are important to our game. Spirit shrines are the main source of feedback that shows players what areas of the garden need more tending to and what areas are already satisfied. The Kami act as a score and will encourage players to tend to the game zones. The wash basin cleanses players, granting them to full peripheral vision of the w axis. The scrolls serve as a quick manual for players to learn about the game zones and back story before playing the level.

4.1 Spirit Shrines

Japanese Shinto temples and gardens contain structures called spirit shrines (Figure 44) that are meant to house the Shinto gods, Kami. Shinto practitioners often give prayers and offerings to the spirits that reside within these shrines. We wanted to incorporate these shrines into our Shinto gardens for their visual, cultural, and functional qualities.



Figure 44: Wooden Spirit Shrine Reference

4.1.1 Design

Since Shinto gardens are meant to create harmony with nature in one's home and appease the nature spirits that reside within the land, we wanted these spirit shrines to physically house the nature spirits that arrive within our garden. Within our garden is a shrine dedicated to each of our three core game zones - the bonsai trees, fish pools, and gravel pits – that provide feedback to players on their performance. As players improve these sections of the garden, the lights within the shrines will grow stronger signaling that more Kami can enter the garden and players are performing well. Each time players perform a correct action, such as adding a fish to a pool or removing a harmful part of the bonsai tree, the shrine will give additional temporary feedback through particle effects. Whenever players need to check their performance in the garden they can observe the level of activation shown on the shrines.

4.1.2 Implementation

Each of the three shrines have different requirements for activation, but they share a few general mechanics. In the center of the shrine is the core lantern that shows the current activation of the shrine. Lights in the core lantern turn on and off depending on changes made to the garden. When a positive change has been made to a core game zone, the side lanterns on the left and right of the core lantern will briefly emit fire particles. This fire effect will last for a couple seconds and then turn off. Once the shrine has been fully activated, the side lanterns will continuously emit fire particles.

Each shrine will also signal the Kami Manager to create new Kami and bring them into the garden once a certain number of shrine points have been achieved. Each shrine will spawn a single Kami for every third of the maximum possible shrine points that have been met. Once a

shrine has been fully activated, a timer will begin that spawns new Kami every 60 seconds. Once a shrine has created a maximum of 10 Kami, no new Kami will be spawned by that shrine. This ensures that there is a maximum possible score of 30 Kami for the entire game. If the shrine loses points and falls below a milestone, Kami will be made sad and begin to leave the garden every 60 seconds.

The gravel shrine (Figure 45) measures how much effort players are putting into raking the pit and their attention to detail. In order to fully activate the gravel shrine, players will need to rake at least a tenth of the total area of each pit along the fourth dimension. In addition to raking a tenth of each pit, players will also have to rake in particular patterns in order activate the gravel shrine. To do this, players will be given colored pattern lines in the pit that they need to match with their design in the gravel. The shrine will fully activate only when a design players raked is 100% similar to these lines given to them.

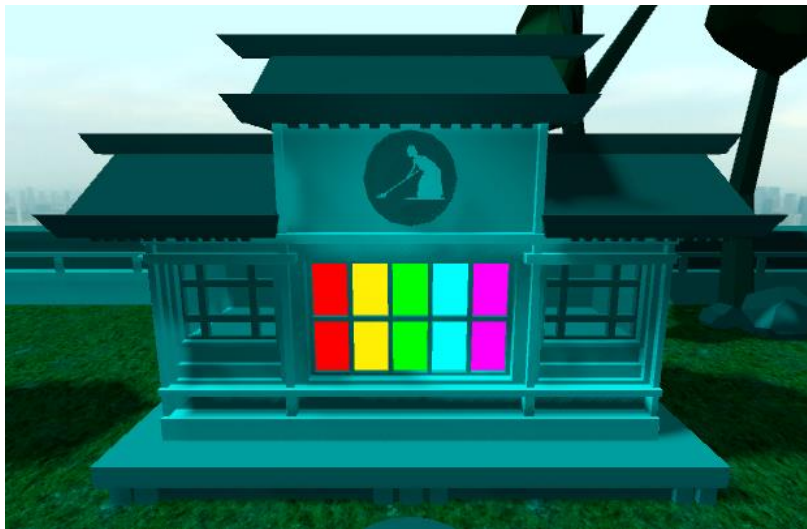


Figure 45: Gravel Shrine

The fish shrine (Figure 46) measures the care given to the population of fish present in the fish pool encircling the gravel pit. In order for the shrine to fully activate, there must be at least three large fish in each fish pool located in the garden. With each new large fish in a fish

pool, a slot on the shrine associated with the fish pool color will activate. If the large fish die of hunger or being eaten, then the shrine will lose a point of activation.

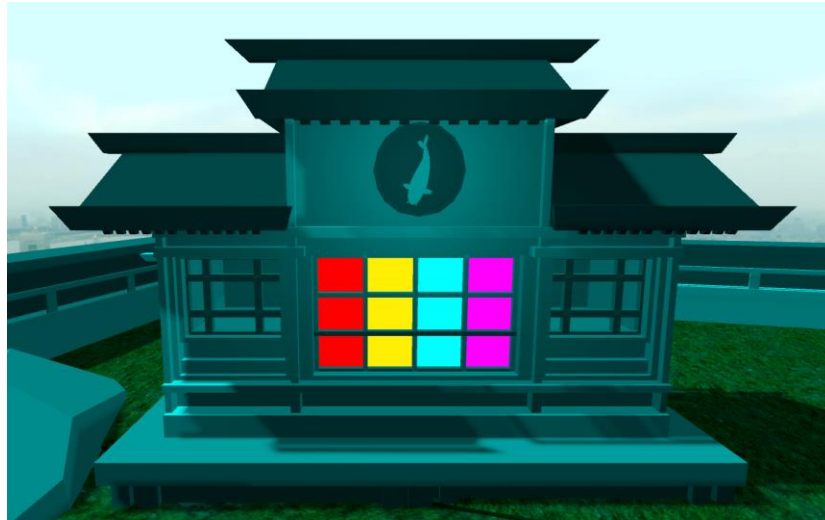


Figure 46: Fish Shrine

The bonsai shrine (Figure 47) denotes the care placed into the garden's bonsai trees. While the fish shrine counts how many things are right with the fish, the bonsai shrine counts how many things are wrong with the bonsai trees. At the start of a level, the two bonsai trees will be grown into a general shape and it is the goal of players to ensure that these trees remain healthy and grown beyond a certain threshold. Bonsai trees are seen as being in poor condition when any of these conditions are met:

1. There are dead leaves or branches present on the tree.
2. A portion of the tree is infested by bugs.
3. The maximum number of leaves or branches has grown on the tree.
4. There exists no branch made of four segments from the tree base to branch tip.

Bonsai trees also have to reach a minimum number of four branches and five leaves before activating the shrine. If this minimum number has not been reached yet, the shrine will completely deactivate.



Figure 47: Bonsai Shrine

4.2 Kami (Nature Spirits)

The Kami are nature spirits (Figure 48) that inhabit Shinto gardens. The players' goal in the game is to call as many Kami possible back into the garden. The purpose of the Kami is to provide incentive for players to tend to the garden.



Figure 48: Kami Reference

4.2.1 Design

Kami will fly above the play area and emit small particles. Kami also function as a score for players at the end of the game, as well as visual feedback for players to show them how well they are taking care of the garden while in the contract level. Once the time limit for the level has been reached, the number of Kami in the garden is counted. In the home scene, there are Kami houses along one of the walls. Each house with a Kami in it represents one point. This is effectively a high score board in the game. By adding this quantitative feedback, players may feel that they can do better in another playthrough of the game – this was evident in our playtesting, as players would often ask us how their final score ranked among scores from other testers. This feedback and scoring makes the game re-playable and shows players their skill in caring for the garden.

4.2.2 Implementation

The Kami (Figure 49) act similarly to the fish; they have behaviors that control how and where they move. Objects like the shrines and clocks will change the behavior of the Kami to respond to different events that occur in the game.



Figure 49: Kami above the Garden

At any one time, a Kami will have one of the following behaviors:

- Happy
- Sad
- Flee
- Ending

Kami spawn in the **happy-state** by default. While happy, the Kami will fly at a rapid pace and turn to face a random point in the area above the garden. We had planned to have more detailed artificial intelligence in the Kami to interact with objects in the garden. For example, the Kami might circle around a gravel pit if a pattern was matched, or a fish pool if it had the correct number of healthy fish in it. However, this was cut due to scope issues.

The **sad-state** is used only in the home garden. Kami in the sad state will not move, which allow us to create the high score table in the home garden. This state was also originally planned to have the Kami try and hover near objects that needed tending to, but this was also cut due to scope.

The **flee-state** causes the Kami to target a location high above the garden. Shrines set some of the Kami to flee if they decrease in activation when a player fails to tend to that area of the garden. Once the Kami reaches a height of 20 units, it will have the Kami manager remove it from the game.

At the end of the game, the spirits that remain in the garden will start to spin around the player. The spirits will continue to spin around the player as they begin to move upwards towards the sky, triggering the end of the game.

4.3 Wash Basin

In Shinto tradition, the wash basin is used in a purification ritual during which shrine visitors must dip a ladle into the water and clean their hands and mouths of impurities. These wash basins (Figure 50) are usually made from stone and found at the entrance to Shinto temples.



Figure 50: Traditional Stone Basin for Washing Hands Reference

4.3.1 Design

There is a wash basin in the garden that allows players to receive a buff to their fourth-dimensional vision for a short period of time when they drink from its water. This buff will allow players to see the full spectrum of the fourth dimension no matter how many shrines are activated. Players receive this buff by performing the purity ritual, where they dip the ladle into the basin water and drink from it.

4.3.2 Implementation

When players enter a level, they will find the wash basin close by within their play area and on the yellow point of the fourth dimension (Figure 51). A ladle can be found on top of the wash basin. The ladle can be grabbed by players when pressing the grip buttons while touching the controller to the ladle handle. When players dip the ladle into the basin, the ladle cup fills with water. They can then lift the ladle cup to their face and tip it over to drink from it. Drinking from the ladle empties the cup and sets a player's peripheral vision along the w axis to its maximum for 60 seconds. If players tip the ladle cup over while out of range from their face, the cup will empty, and the drinking effect will not happen. The ladle is determined to be tipped over when the angle between the world up vector and the up vector pointing from the top of the cup is greater than 90 degrees along the x, y, or z axis.

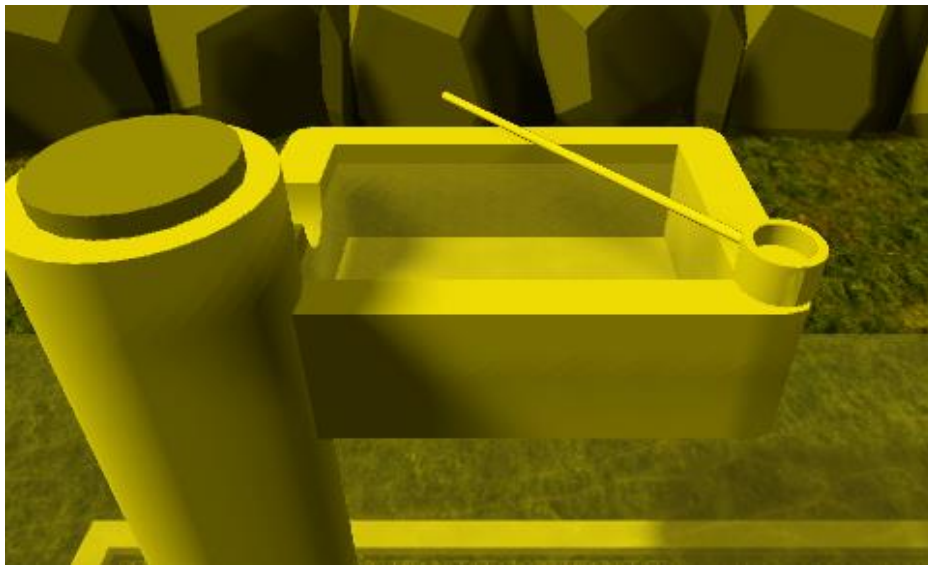


Figure 51: Wash Basin and Ladle

4.4 Scrolls

We needed a method to convey information to players about controls, game goal, and contract requirements that functionally fit into the limited game space offered by the HTC Vive.

4.4.1 Design

We decided to create a scroll object that could expand and contract to meet the limited space requirements of the Vive and display information in bite sized chunks to players. We felt that scrolls would match the more traditional Japanese cultural presence that we were trying to make use of in our game.

Our scroll object eventually performed two major functions within the Home level of our game. First, the contract scroll managed as a go between for the home level and the Tokyo contract level. The contract scroll hung from a wall (Figure 52) and displayed a letter shown in Figure 53 from a contractor asking the spiritual conduit to come tend to the garden at his home. When players grab the rope handle at the bottom of the scroll and yank it, the scroll closes and players are moved into the Tokyo level.

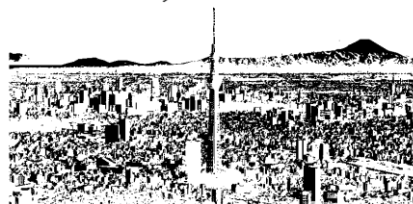


Figure 52: Contract Scrolls

Conduit Contract

Tokyo

15 minutes



My work has pulled me from my lovely home in Kyoto to the heart of Tokyo. No longer can I stroll through the beautiful gardens of your temple to shed the troubles of my day. I ask that you kindly visit my rooftop terrace in the Chiyoda ward of Tokyo and connect me once more with the nature that I dearly miss.

Best Wishes,
Kiyokawa Kiyoshi

Close the Scroll to Begin

Figure 53: Tokyo Contract Scroll Text

Second, the tutorial scrolls sat on a desk on the adjacent wall of the contract scrolls and displayed important game information to players. The three information scrolls covered game controls, game zone rules, and the story concept.

As seen in Figure 54, the controls information scroll informed players how to use the HTC Vive controllers to interact in the garden. Below the controllers graphic is a note informing players that they can open the other information scrolls for more information on the game.

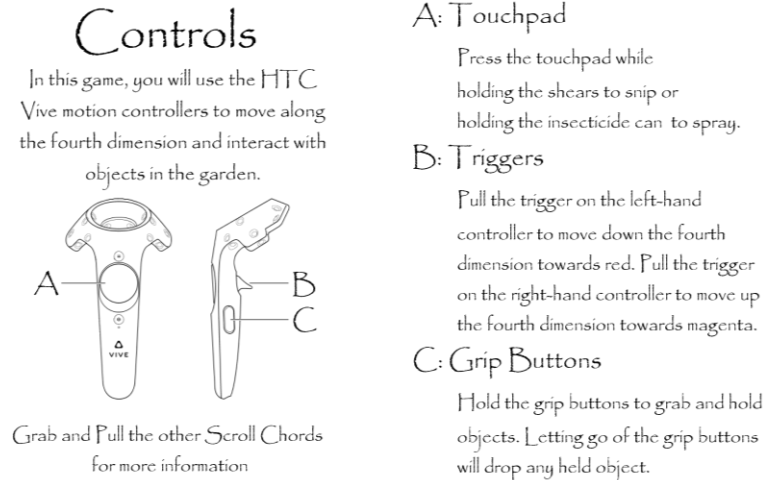




Figure 54: Controls Scroll Text

As seen in Figure 55, the how to play information scroll informed players how to care for the core game zones in the garden. There is also a section on the wash basin so that players know they can gain full four-dimensional vision by drinking the water. We did not have time to implement a tutorial teaching players how to interact with the garden game mechanics, so we settled for including an in-game manual.



Figure 55: How to Play Scroll Text

As seen in Figure 56, the game concept information scroll informed players about the general goal of the game and their role in its world.


The Spiritual Conduit


You are a spiritual conduit. It is your job to connect with the spirits and gods of the Shinto religion. You do so by tending to the temples, shrines, and gardens dedicated to worshipping the Shinto gods, or Kami.

In this game, you must accept contracts given to you by people all throughout Japan and improve their gardens. You will have a limited amount of time to care for the Bonsai Trees, Gravel Pits, and Koi Ponds before you must return home.

Your goal is to attract as many Kami as possible to the garden before this time runs out. Kami will move into the garden as the surrounding spirit shrines become more activated through your gardening efforts.

Figure 56: Game Concept Scroll Text

These three scrolls (Figure 57) sit on top of each other with the game zone rules and story concept scrolls initially closed. The game-controls scroll asks players to open the other two scrolls for more information. Stacking these scrolls saved a large amount of space and put the scrolls close enough to players that the text was readable.



Figure 57: Information Scrolls

4.4.2 Implementation

First of all, we had implement a method by which players could grab the handle tied to the bottom of the scroll page and pull it to open or close the scroll. This was done by checking if the handle ring was grabbed and following the position of the player's hand. The ring was considered grabbed if the player's hand was colliding with the handle ring and the controller grip buttons were pressed. When grabbed, the scroll would check its defined orientation axis and follow the position of the player's controller along that axis. So long as the controller did not exceed a predetermined range of movement along the axis of movement, the scroll would follow the controller's x, y, or z position. If the scroll was initially closed when grabbed, then it would expand to its open position once players let go of the handle ring. If the scroll was initially opened when grabbed, then it would contract to its closed position once players let go of the handle ring.

In order to implement the visuals of the scrolls, we actually had to write two new shaders that made use of Unity's stencil buffer: the Visibility Volume and Limited Visibility Object shaders. The reason we needed new shaders was because we needed the plane displaying the scroll text to disappear when the scroll was closing or closed. Normally, we would just decrease the scale of the plane along a single axis and move it towards the scroll top to give it the appearance of closing, but this method would shrink the texture on the plane. We needed the texture to remain the same size during the opening and closing process to give the appearance that the scroll was actually unraveling from the base.

The Limited Visibility Object shader was written to take a texture and color value and display it on the surface of the object. The scroll page shader had two passes with the first drawing a single color to the plane surface and the second drawing the semi-transparent scroll

text texture to the plane surface. However, the shader would only draw to the plane surface if the reference value on the stencil buffer was equal to one. This ensured that any section of the plane outside of the required render zone would not be drawn to the render frame. When the scroll closed, the page would seemingly roll back into the scroll base.

The Visibility Volume shader was written to create a render zone on some given shape. We placed the material shader on a thin cube object that fit within the space required by a fully extended scroll page. When the camera looked at the scroll page plane through this render zone it would render the page, otherwise the page would be invisible. This effect was achieved by writing the value 1 to the stencil buffer reference occupied by the render zone. This value was checked by the other shader and would allow the scroll page to be drawn.

4.5 Clock

In the original design of our game, we intended to have multiple contract levels with varying game times. In the end, we had to cut back on having multiple levels due to scope issues. For our game, we wanted to show a proof of concept by implementing a single contract level, and gave that level a time limit of 15 minutes. Players have to activate all of the shrines and accumulate as many Kami as possible within these 15 minutes.

4.5.1 Design

To convey how much time was left for the contract level, we added a clock into the garden (Figure 58). In the Tokyo level, the clock sits on the outer wall of the house facing the center of the play area. There is a single hand on the clock that performs a single rotation during the play session. Players can check how much time is generally left for the level by looking at

the clock. The outer edge of the clock has markers denoting the game time in eighths. Once 15 minutes have passed, the clock hand will return to the top marker, and players are moved back to the home garden.



Figure 58: Garden Clock

4.5.2 Implementation

The clock works by taking the game time set within the contract level and rotating the clock hand by a proportion of the current game time against the time limit. At a time of 0 seconds the clock-hand will rest facing upwards. At half the time limit, the clock-hand will face completely downwards. At the end of the time limit the clock-hand will return to face upwards. In our implementation, the clock object is in charge of ending the game once the time limit has been reached. Once the time limit has been reached, it tells the Kami in the garden to spin around the player and then changes game scenes after the Kami reach a certain height above the player.

5 Tools and Specifications

5.1 HTC Vive

5.1.1 Controller

This game was developed for use with the HTC Vive, so players control the player character using the Vive Head Mounted Display (HMD) and wireless controllers. The HMD controls the direction and position of the player camera in the game world. The controllers link the player's real hands to the player character's hands in the game world.

5.1.2 Movement

Due to the features of the HTC Vive, players move around the game world by physically walking within the HTC Vive's defined play space. This play space is set up before the game starts when players calibrate their HTC Vive. The play space is determined by the distance between the two tracking cameras used by the Vive. The cameras can be a maximum of 5 meters apart from each other, making the maximum area 3.5 x 3.5 meters. During our development, we used a HTC Vive work station with an area of 2.5 x 2.5 meters by placing the cameras 3.6 meters apart from each other.

Since we also have a fourth-dimensional game mechanic, players can move along the fourth dimension just as they do in three-dimensional space. Players' movement along the fourth dimension is limited to five points signified by five colors. When moving to another point on the

4th dimension, the origin of the player character's body is rooted at the new position along the 4th dimension.

5.1.3 User Interface

We have the HTC Vive's wireless controllers represent a player's hands within the game world. As a player moves and rotates the controllers around, the player character's hands follow the same path. Players can use the buttons on the controller to interact with objects as if they were grabbing or using them in the real world. The controller buttons perform the following functions:

- Grip buttons
 - Grab and hold objects in the player's hands
- Trigger
 - The trigger on the left controller will move the player in the negative w direction, meaning that the player will move towards red
 - The trigger on the right controller will move the player in the positive w direction, meaning that the player will move towards violet
- Trackpad
 - Clicking the pad while holding the shears will perform the snip action
 - Clicking the pad while holding the insecticide will perform the spray action

Once players have moved to a new position on the 4th dimension, they can interact with objects around them of the same color. This means that they can pick up items, rake gravel, snip tree branches, touch fish, and change chimes that reside on their current w value. Objects at a different position on the fourth dimension will not react to the players' touch. Players can also move objects along the 4th dimension by picking them up and moving their body's position to a new w value.

5.2 Unity3D

When choosing the correct game engine for develop our game, we had to consider four major requirements: the engine had to support three-dimensional graphics, have a small overhead for learning engine tools, provide in-engine support for the HTC Vive, and allow us to write our own custom shaders. The two engines we had to choose between were Unity3D and the Unreal Engine. Of the two engines, only Unity3D allowed developers to write their own custom shaders using their framework with the Cg/HLSL programming language. Fortunately, two of us had previous experience working with Unity3D so we chose to use it for the development of this project.

5.2.1 HTC Vive Support

Between the two engines, Unity has the quicker and simpler Vive set up process. To get the head mount display working in Unreal, a game mode has to be created, a couple of blueprints need to be wired, and input settings need to be adjusted. With Unity, downloading the SteamVR asset from the asset store and dragging the camera rig into a scene instantly sets up the HMD, the controllers, animations for the components of the controller and the chaperone. To avoid any errors or bugs from trying to set up the HTC Vive, Unity was the best option as this was done by simply importing the asset into the editor.

The Vive input system does not come fully optimized after importing it into the project. This means that we would have to edit the code for the Vive to make the input work in our game. This would be harder to do in the Unreal engine as the setup uses blueprints to run Vive functions and doesn't directly import Vive scripts into the editor. When importing the asset into Unity, a new folder is created in the project hierarchy that holds all the models and scripts that

are directly used by the game. The scripts can be edited and the changes will immediately update in the inspector of objects using Vive scripts.

The SteamVR asset folder for Unity also comes with an example scene. Since this would be our first time using and coding the Vive, this example scene would help us understand Vive functions and give us insight on its capabilities with Unity.

5.2.2 Cg/HLSL Shader Programming Support

According to the Unity Documentation Manual, Unity has a wrapper for all Cg/HLSL programs called ShaderLab. ShaderLab code provides fallback shaders when an older computer cannot run your written shaders, properties that allow developers to alter shader performance within the editor inspector and scripts and packages that provide fixed-functions that simplify the process of rendering Unity scenes.

Within the Unity3D engine, developers can choose between creating surface shaders or vertex and fragment shaders. Surface shaders rely heavily on Unity's ShaderLab as they provide a "higher level of abstraction for interaction with Unity's lighting pipeline" (*Writing Shaders*) in order to cut down on the work required by shader programmers to account for directional, point, and spotlights placed into a scene by level designers. Using this abstraction, developers only need to write a small amount of Cg/HLSL code to create the effects they desire and the rest is auto-generated during the compilation of the shader.

Vertex and fragment shaders provide developers with the largest amount of control in how a shader renders objects in a game scene. These shaders are written using vertex and fragment functions which are the fundamental components of any shader. The vertex function is passed the vertex point data of a mesh within a game scene, and then performs the fragment

function on every screen pixel taken up by that vertex. This allows developers to write a larger amount of Cg/HLSL code with more control over how game meshes are rendered to the screen.

For our development, we focused on writing vertex and fragment shaders that were fed information by scripts on objects within our game scene. This information allowed us to render our fourth-dimensional environments depending on each object's position in relation to the player's position along our 4th axis of space.

5.3 Art Tools

5.3.1 Blender

The primary three-dimensional modeling program used for creating three-dimensional assets in our game was Blender version 2.77a. Blender is a “free and open source three-dimensional creation suite” (*Blender.org*) that provides a free alternative to Autodesk's 3DS Max or Maya. Although we could have used 3DS Max or Maya under a free student account, we had more experience working with Blender than the Autodesk products. Blender was used to create all three-dimensional models in our game that were not downloaded from websites providing free models.

5.3.2 Gimp

The primary image editing tool used for editing and creating 2D images for use in our game was Gimp version 2.8. Gimp is a “free and open-source GNU Image Manipulation Program” (*Gimp.org*) that provides a cheap alternative to Adobe Photoshop. Gimp was used to create the textures, skybox, and supplementary art assets for this project.

6 Playtesting

The formal playtesting session was made up of three parts: a control mapping test, a four-dimensional playthrough of the game, and a three-dimensional playthrough of the game. After each part, there was a short interview session to gather information about the features in the individual parts. The full IRB approved¹ procedure and list of interview questions can be found in Appendix A. Our formal playtesting session gave us insight on the preferred control scheme for the game and on the difficulty of each of the game zones. The playthrough without the fourth-dimensional mechanic also informed us if the mechanic had any significant impact on the gameplay and the level of engagement players could maintain while playing the game.

6.1 Control Mapping Test

Deciding on how players were going to move along the w axis was a difficult design decision as it is an action unfamiliar to human beings. We decided that players could use either the triggers or specific touchpad input to move along the w axis. The purpose of the control mapping test was to evaluate the most preferred control scheme for moving along the w axis. Figure 59 depicts the game environment used for this part of the playtest. During the time of this playtesting session, our game had seven points on the w axis. In the scene, there are seven cubes, one on each of the seven points of the w axis. Players start on position zero on the w axis, so initially only the red cube is visible. Players were then asked to move to different positions on the w axis using four different control mappings.

¹ IRB number: 16-204

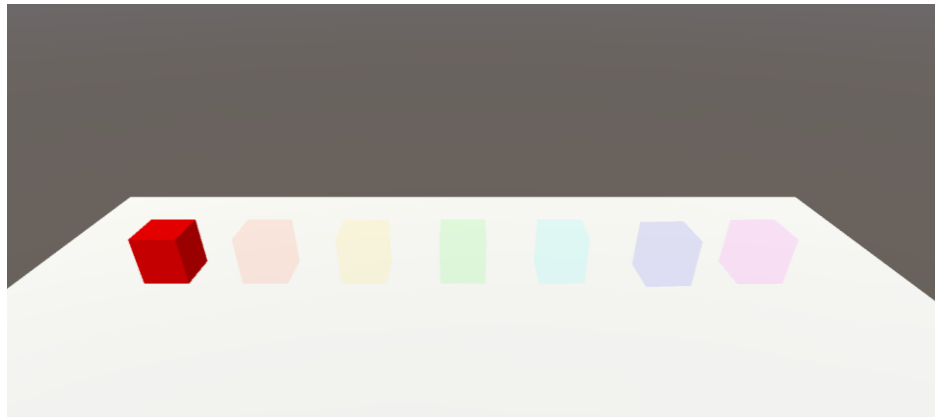


Figure 59: Control Mapping Test Scene

Control Mappings:

1. Trigger: The trigger on the left controller decreases the player's position on the w axis and the trigger on the right controller increases the player's position.
2. Vertical Slider: The touchpad is divided up vertically into seven sections. Touching the section that corresponded to the current position on the w axis and sliding up or down would change the position. Sliding to the very top would move to the maximum w position, and sliding to the bottom would move to the minimum w position.
3. Up/Down Buttons: Touching the upper half of a touch pad increases the player's position on the w axis, and touching the lower half would decrease the player's position.
4. Scrolling: Touching a touch pad and sliding in a clockwise motion increases the position on the w axis, and sliding counter-clockwise decreases the position.

During the interview session following the control mapping test, the majority of the participants preferred the trigger control scheme over the others (Figure 60). Most participants found that the control schemes that involved sliding a finger on the touchpad were too sensitive and inaccurate. Participants were more likely to pass the target w position using these controls

than using the trigger or up/down buttons schemes. Only one of the participants had a negative opinion on the up/down buttons control scheme. The tester found it to be very tedious to move from the maximum w position (at the time, this was six) to the minimum w position. The two participants who preferred the vertical slider control scheme both mentioned how they could move on the w axis the fastest with this control scheme. The general consensus, however, was that the trigger control scheme was the most accurate and had no sensitivity issues.

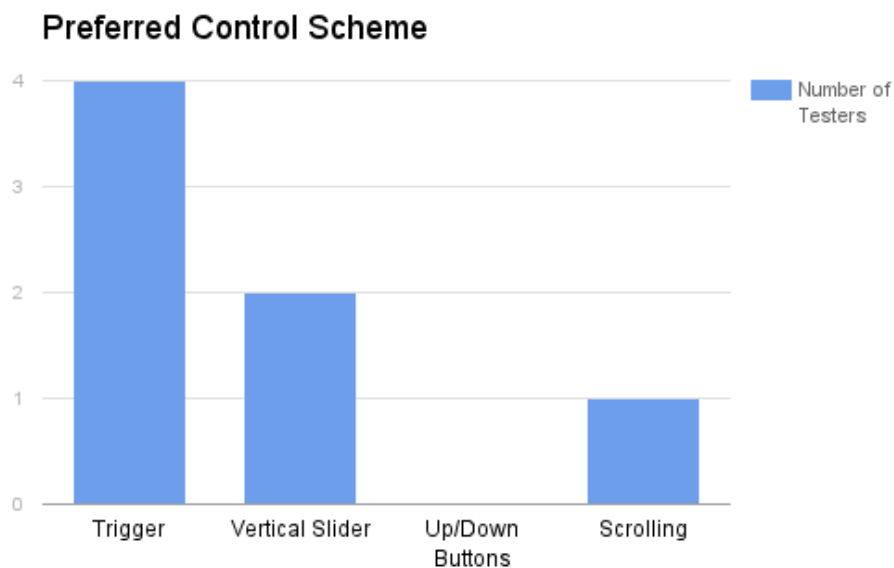


Figure 60: Qualitative Results of Control Scheme Test

6.2 Four-Dimensional Playthrough

After the interview for the control mapping test, participants were asked to play our game with the fourth dimension enabled. At the time of the playtest, the fish and bonsai game zones were completely implemented, the bonsai had required zones where the tree had to grow to activate the shrine, and there were seven points on the w axis. Since pattern recognition for the gravel pit had not been completed, the gravel shrine could be activated after raking a tenth of

each gravel pit in the scene. The scene used for the playtest (Figure 61) also used the first models for the shrines, the orientation of the gravel pit to the shrines was different compared to the current scene, and the Tokyo level only lasted ten minutes.

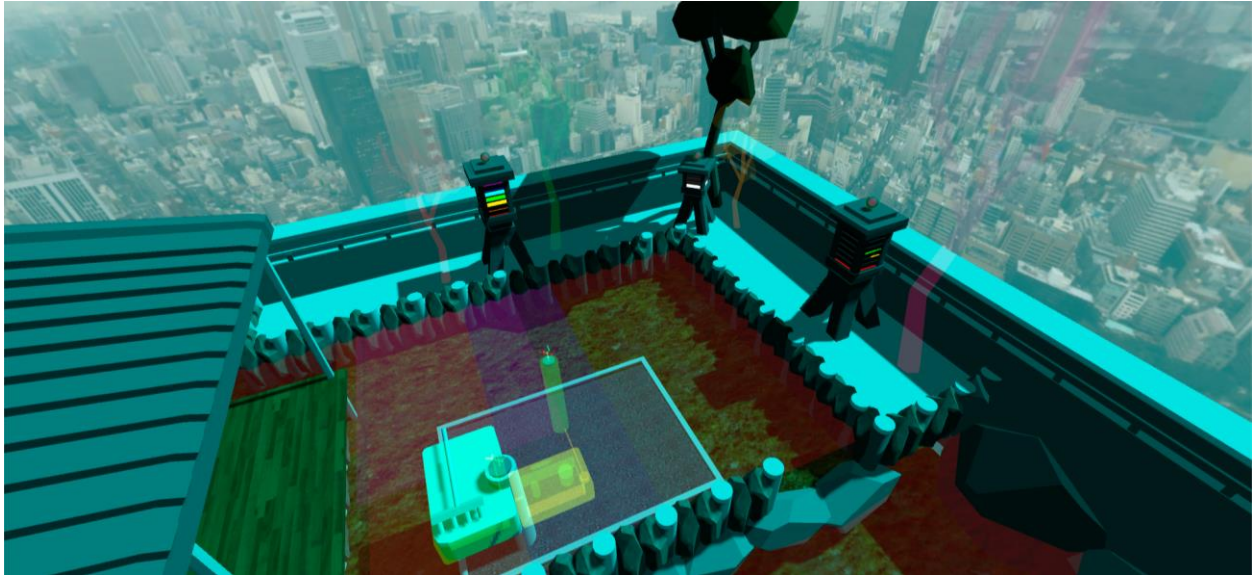


Figure 61: Fourth-dimensional Playtest Tokyo Garden

The data provided from the second interview provided us with valuable insight on how to adjust the balancing of the game zones. The main findings were:

- The bonsai trees required too much time to care for
- The fish pools are too dark and far apart

The data presented in Figure 62 reveals that the majority of the testers found the game to be difficult. This result was expected, as our testers would most likely be experiencing and manipulating four-dimensional objects for the first time.

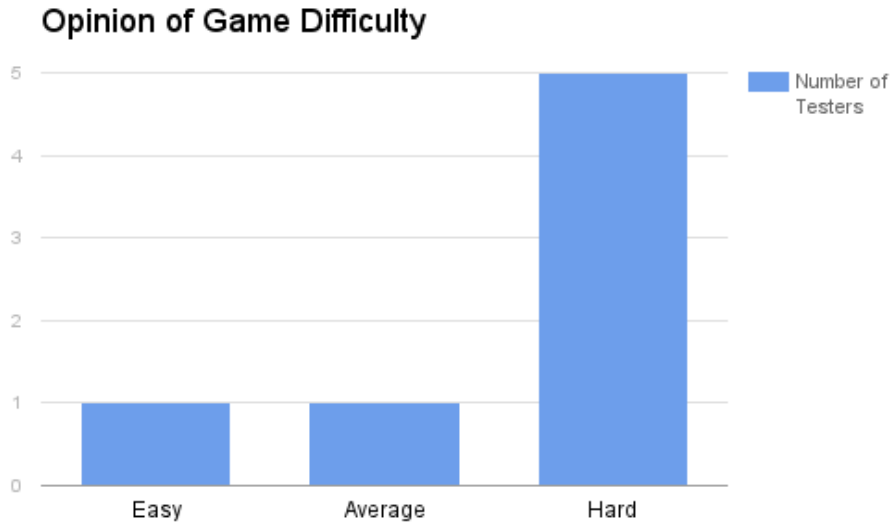


Figure 62: Testers' Opinions on Game Difficulty

The important results can be seen in the data in Figure 63. The number of testers who found the bonsai trees to be the most difficult game zone is equal to the number of testers who had no opinion on which game zone was the least difficult. Many of the testers found the bonsai trees to be the most interesting game zone, but the majority of the playthrough was them trying to keep up with the growth of the bonsai.

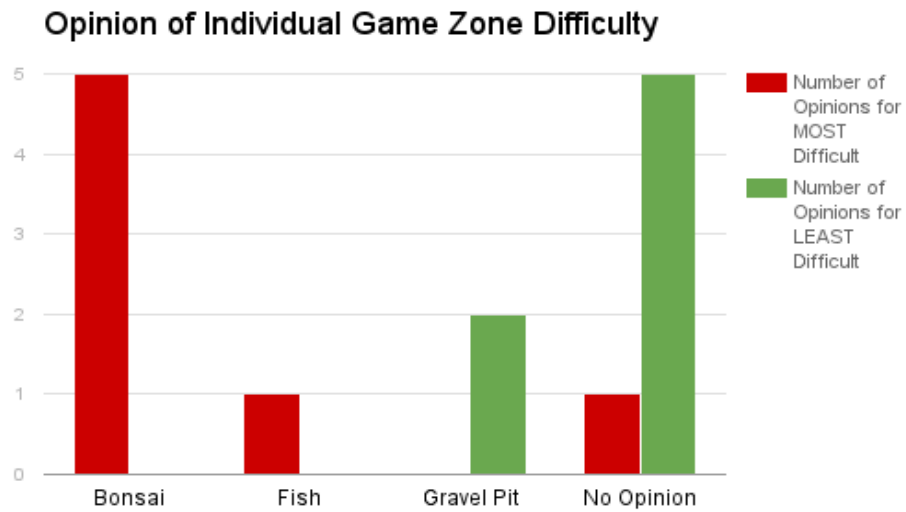


Figure 63: Testers' Opinions on Individual Game Zone Difficulty

When observing the playthroughs, it was noted that once players realized there were dead leaves on a tree, they would spend the remainder of the time trimming dead leaves unless they accidentally cut off a large section of the tree. The bonsai took up a lot of time, as the most common occurrence in all the playthroughs was a player cutting the last dead leaf, only to have the tree grow again. Any fish players may have had eventually died, and the gravel pit was only used by two testers.

The fish zone proved to be more of an inconvenience rather than a challenge. The water texture was very thick and dark, which made it difficult to see the fish in the water. The fish pool locations were also inconvenient to get to, as they were on the edges of the w axis and the furthest from the tool table. This caused most players to only put fish into one or two of the fish pools to make it easier to feed them.

Using the results from the first two sections of the playtest, we were able to make these changes to the game:

- The time between growth cycles was doubled
- The tools were positioned closer to the shrines
- The water texture for the fish pools were brightened
- The model for the shrines was updated
- The range of the w axis was reduced to five from seven, removing the colors orange and blue.

The first change was to double the length of the bonsai tree growth cycle from one minute to two minutes. This gave players more time to adjust to their surroundings and familiarize themselves with the other game zones in the garden.

The next change we made was to orientate the pedestal with the tools on it in a way so that players can look up to see the shrines instead of turning around. This would help players notice the shrines since they provide important feedback on their progress.

We then made some changes to the texture of the water and the intensity of colors on objects in the scene to make the fish and tools easier to see.

The shrine models were then updated to give feedback specifically for this one level since we would not make any more contracts for this game.

The last, and biggest, change we made was to remove the orange and dark blue points from the w axis. With w peripheral vision enabled, players were often confused about their position on the w axis. This was caused by the orange color being very similar to red, and the dark blue color mixing and darkening other colors, making them difficult to distinguish. These two colors made the game world inconvenient to navigate as they were extra points that players would have to move to in order to get to the fish pools near the ends of the w axis. These changes made the game easier to play for first time players while still providing a challenge to frequent players.

6.3 Three-Dimensional Playthrough

The last part of the playtest was another playthrough of the game, but the fourth dimension had been removed from the game. All the game objects were set on the fourth w point (cyan) (Figure 64), and the ability to move on the w axis had been removed. We added this to the formal playtest since we wanted to analyze the effect that the fourth dimension has on our game. More specifically, we wanted to know if the game was any more enjoyable, easier, and/or interesting by adding the fourth dimension to it.

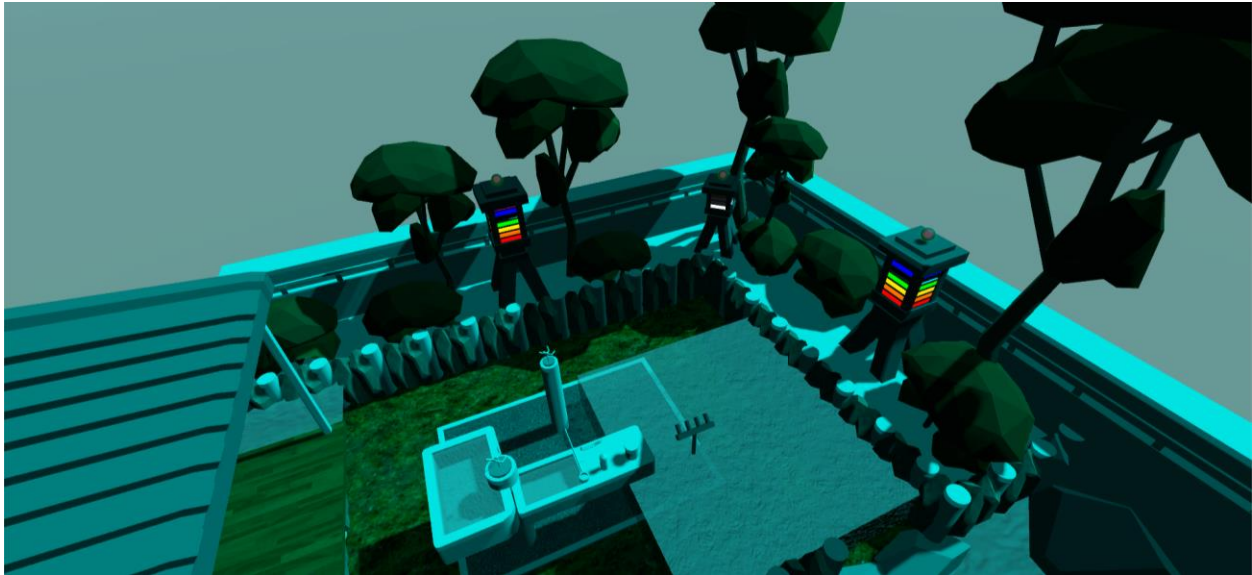


Figure 64: Three-dimensional Playtest Tokyo Garden

From the last playthrough, we noticed that testers found that the fourth-dimensional playthrough was very complicated and difficult (Figure 65), but enjoyed it more than the three-dimensional playthrough (Figure 66). Many of the testers found the fourth dimension to be interesting and enjoyed moving to different w positions and interacting with fourth-dimensional objects. Some noted that having objects spread out on to other points on the w axis prevented from being distracted by them, especially when tending to the bonsai trees. Some found the fourth dimension to be more of a distraction and an inconvenience. Not only would they have to move in three dimensional space but also manage their position on the fourth dimension as well.

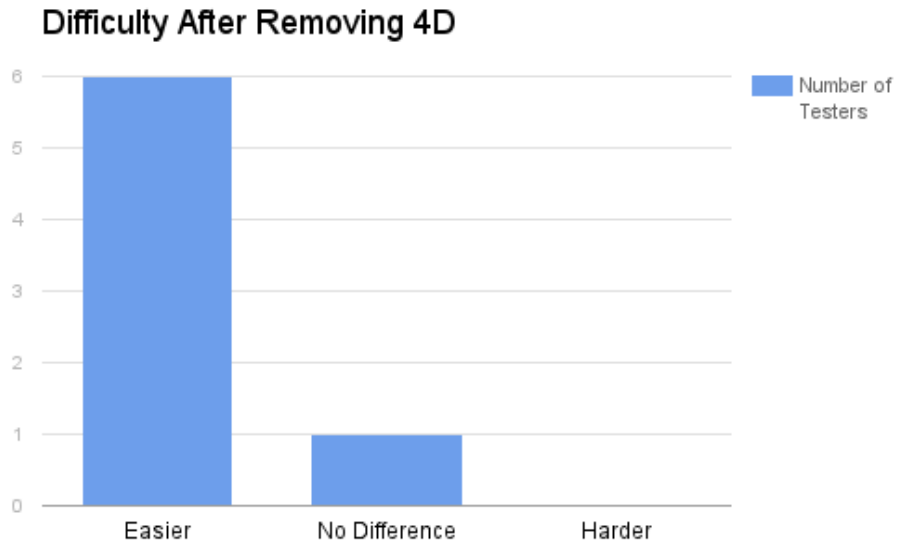


Figure 65: Testers' Opinions of Game Difficulty after Removing 4D

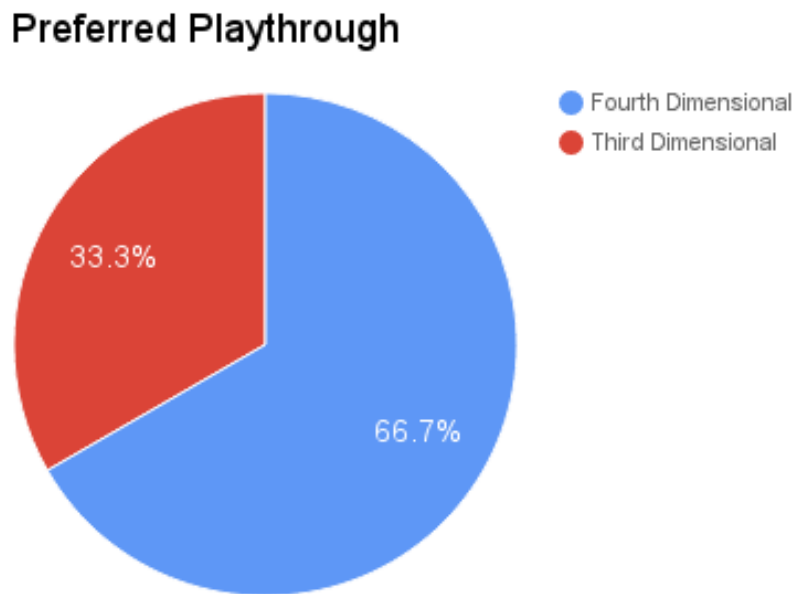


Figure 66: Testers' Preferred Game Types

This result may have been biased since this was all of the testers' first time moving in a fourth-dimensional environment. If they had more experience and practice moving in our game as we did when making the game, they would find the fourth dimension to be more useful. There

were also some comments about how the environment looked prettier in the fourth-dimensional playthrough compared to the third-dimensional one. This may be another bias since we chose not to suppress the color changing in the third-dimensional playthrough, which would have caused objects to keep their original, more natural colors. This may have impacted some of the testers' opinions on the game, especially those who prefer both equally. In the future, it might be better to put more effort in making the third-dimensional playthrough look realistic to reduce the quality of visuals from impacting the preference of testers.

7 Postmortem

7.1 What Went Right

7.1.1 Design

Our team came into this project committed to making a fourth-dimensional game for a virtual reality platform. Although unsure at first, our advisors allowed us to explore our ideas so long as we incorporated Japanese culture into the game. For the first three weeks of this project we focused on designing a game that had players care for a fourth-dimensional garden with the immersive controls provided by the HTC Vive. We ended up designing a largely original gardening game that explored some very new and experimental mechanics. Fortunately, we believe that we succeeded in bringing our designs to fruition from conception and meeting our experience goal. We had our fair share of development hurdles and scope changes, but in the end we overcame our problems and produced a working, polished game.

Our greatest success from our designs came in how well the virtual reality and fourth-dimensional mechanics worked to support each other. Our idea for a fourth-dimensional game mechanic was very experimental in the first place, and although our implementation is a limited version of the complete concept, seeing our players quickly adapt to moving on four dimensions of space is incredibly rewarding. From our playtests, we noticed that players found the game more interesting and challenging when the garden was fourth dimensional. The fourth-dimensional game mechanic had a clear impact on improving the game, and if removed, would harm the game.

We felt that the use of the HTC Vive was paramount in making the fourth-dimensional game mechanic work. If players had to use a mouse and keyboard or gamepad to interact with the game, it would most likely be unplayable. Even changing the control scheme of the Vive controller increased the difficulty in navigating the garden. Players were meant to have a large amount of finesse in interacting with core game zones and moving about the garden. We also found that we had unconsciously solved the problem of the HTC Vive's limited play area space. The fourth-dimensional mechanic took the 9 square meters of play space and multiplied it by 5, one for each color. Even though we packed a good amount of objects into the play space, players are never cramped or wanting for space.

7.1.2 Teamwork

The three strengths of our team were our ability to efficiently divide work between ourselves, take responsibility for our features and tasks, and maintain good communication. Our team consisted of three tech-track game design students, two of which were computer science double majors. Without a dedicated artist on the team, we had to come up with enough features to meet our degree requirements and keep us working throughout the timeline of our project. We chose to divide the three core game zones among the three of us and dole out additional features to whomever had time or interest. Alex focused on the fish pools, John focused on the bonsai trees, and Paul focused on the gravel pits. Doing this ensured each of us were invested in the game and responsible for our features that were essential to the game. This also meant that we had to constantly keep up-to-date on how the game worked and how our features worked with the rest of the game. Also, by linking our core game zones to the shrines and Kami, we were able to connect each other's work in some way.

We were able to stay connected by using a set of useful tools without which we would never have been able to complete the game. Our primary platform for sending text chat and progress updates was the messaging app LINE. We used LINE because it was the most popular messaging app at the time in Japan. We could use the app to message each other, our advisors, family, and lab mates for free over Wi-Fi and data. Whenever we had a problem, idea, or update, we could easily inform each other at all times. Early in development we decided to work in separate rooms, which gave us access to the resources we needed and space to focus. Alex worked on the 5th floor in the VR experiment room and Paul and John worked on the 6th floor in the communal lab room. LINE allowed us to keep this space and call each other over for testing and meetings.

We used GitHub as our means of storing a single online repository of our project code and merging individual code. Luckily, Unity3D has a built-in feature for converting metadata to a text format and safely storing it on GitHub. We kept a methodology of creating individual branches for each new feature or bug fix. Once a branch was completed, it would be reviewed and pulled into the master branch. We kept each other updated about which files we were changing in LINE and required code reviews before merging pull requests.

7.1.3 Above and Beyond Work

There were a few areas during development where we went above and beyond the expectations placed on us to improve our game when possible. We originally designed our game to be rather large, and had to eventually cut our scope, but during development we ended up creating some additional features that were cut from the final game. Having this surplus of features allowed us to take a hard look at which of our designs were good and bad. Rather than stretching to meet our minimum goals, we were able to cut extraneous features and focus on core

game zones. Our final product has a lot of things to keep our players busy and flesh out the game world, and we were lucky to reach that point in development.

During the last week of development, we made a decision as a team to take on a crunch week and polish the game as much as possible with the time we had left before our presentation. We ended up pulling at least 10-hour work days including weekends that week making sure that everything we wanted to polish was completed. We could have scoped better during the project and worked more consistently so that a crunch week wasn't needed, but in the end, most of our core features were completed by this time. The additional time we put into this week improved performance, art, game flow and level design. From what we have seen, this work squashed almost every bug remaining in the game.

In the final week, Alex spear-headed our playtesting taking full responsibility for scheduling and performing tests. His commitment to conducting and reviewing the playtests allowed us to continue developing other important features instead of halting development for testing. Another major additions was the improved art for the shrines, home level, and Tokyo garden. After fixing any remaining bugs and changes to his features, John spent a few days of the crunch week devoted to improving the game art. The improved art significantly improved the clarity of game goals and flow of play, while also hopefully helping our game visually stand out even though we had no dedicated artists.

7.1.4 Positive Changes

We made some changes in the design and implementation of our game in the middle of development that resulted in improving the game experience. First, we decreased the number of points on the fourth dimension from 7 to 5. We found through playtesting that 7 points were too many for players to manage and some colors were too similar. The common complaint was that

red and orange were too similar, so we moved to 5 colors. This simplified our development of the garden and the experience for players.

Second, we decided a month before the project end date to focus on making only one contract garden. We originally planned for 3-9 contract levels, but reduced our scope to one level to meet our requirements for a minimum viable product. Focusing on one contract level alleviated the issues we were having with finding time to get all of our features completed. It also allowed us to devote weeks of development time to polishing the contract garden.

Third, we decided in the last week of development to ditch the constraining theme of Tokyo from our contract garden. To fit the Tokyo theme, we designed a restrictive requirement on the bonsai tree and shaped the fish pools in a way that confused players. The bonsai tree requirement confined bonsai growth to a small area and made the bonsais near unwinnable if a branch passing through a requirement ring was snipped. The fish pools representing the Arakawa and Sumida rivers were far too thin. Shifting focus away from thematic integrity towards making sure that the core game zones were structured to make the most of their creative freedom definitely improved the final product.

Finally, we changed the implementation of the shrines to better convey game progress and fit the true Japanese cultural image. During early development our shrines were actually lanterns that lit up in sectors with each positive action. Later in development, we changed the shrines to actually look like spirit shrines with internal lanterns providing feedback for progress and positive actions. We also altered the shrines to show specific progress for the Tokyo level rather than generic progress for any possible contract level. For example, the fish shrine kept track of the 4 Tokyo fish pools rather than a possible maximum of 5 fish pools. We could make this change since we pivoted to only create one garden in the game.

7.2 What Went Wrong

Although we are content with the final product of our game, there are certainly a number of things that went wrong during the development process. These events either hindered our progress or could have been improved to make our overall experience with designing and producing the game easier.

The first issue we had with our project was having too large of a scope when designing the game. This led to over designing the game and many objects were created that would eventually not make it into the final product. The time used for creating those objects could have been used to polish already existing features in the game. This may have caused us to have to work more often in the final weeks of the project to produce our final product. On a similar vein, initially our project was scoped under the assumption that each team member had experience with the Unity game engine. This was not the case, as Paul had not worked with the engine before. As a result, milestones that we had planned to accomplish in a week instead took around another two to three weeks to reach.

This lack of experience directly affected work on the gravel pit, which Paul was tasked with completing. Paul spent the first month researching the Unity engine and adaptive help systems; his initial technical challenge. Development of the gravel pit started mid-August and lasted the remainder of the project duration. This affected our report of progress during our weekly meetings with our advisors, as we believed that the gravel pit would take less time than it did. As a result, milestones were not reached and the minimal viable product was not implemented as early as was desired. The lack of a minimal viable product caused John and Alex to stop working on some features, as they needed the gravel pit to be finished before they could start building contracts. This was an issue because during that time of limbo, they could have

been working on other features that did not necessarily need the gravel pit, such as the tutorial levels which we had to cut from the game because of scope. Also, when it came to coding the gravel pit, Paul did not initially consider some cases that might/did occur when merging the pit into the game, such as the pit possibly needing to be scaled down or up in size. As a result, when the gravel pit was initially merged into the game, it didn't work properly and needed to have bugs fixed before it could be merged into the game again. Similarly, when implementing Paul's pattern recognition into the game, he and Alex spent a large amount of time fixing bugs with the code while using the Vive. Although this was helpful working together to problem solve and figure out the issues with the pattern recognition and was possibly faster, Paul could have done this by himself with better debugging, which would have allowed Alex to work on other features or polish existing ones. The lack of the minimal viable product is what eventually caused us to cut our game down to one contract level with the three game zones.

We mentioned earlier that we overdesigned parts of our game. This was partially due to the small amount of time John and Paul were each asked to play the game in its current state. If they had played the game more often and seen what kind of progress had been made and what needed to be done still, the overall design could have been scoped better and sooner.

Communication issues with game progress and how game features worked in the Vive setting was also a hindrance to how quickly game features were implemented and if they needed to be fixed before re-merging them into the game. Finally, the amount of features produced and the input on the original design of the game by each member was uneven. There was the occasional loss of motivation and some irregular work schedules with some members detracted from our work production. Perhaps the last week crunch time could have been avoided if the time we worked was more organized and consistent.

7.3 What We Would Do Differently

From this project, we learned a lot about choosing to add more depth to a game versus polishing already existing features. After the formal playtesting session, we realized how interactive and interesting the bonsai game zone was. Most testers claimed to enjoy the game even though they never touched the gravel pit and had all their fish die. We feel that another version of this game could have been made with just the bonsai game zone implemented, but with more depth focusing on the other way to tend to bonsai trees. Instead of dividing the work by having each team member work on a different game zone we could have each of us implement a tool and behavior to the tree to make it more complex and interesting. This way we could have one core game zone that we add some depth to and then have more time to polish it to make it look and run superbly. We could even incorporate the fourth dimension as there were testers who claimed that the bonsai tree was easier to tend to in the four-dimensional playthrough.

With the version of the game we did complete, we felt that the people who played our game did not experience the game in the way we had planned. This may have been caused by cutting the Kami having unique artificial intelligence and behaviors that would make players feel appreciated for their work in the garden. We would have liked to have the Kami emit sounds that convey their feelings and interact with the objects in the garden and players more to provide more feedback on how well players are tending to the garden. This would also help achieve the experience of working to please an audience which was part of our experience goal of working on a project.

While our game was four dimensional, there were many aspects of the fourth dimension and four-dimensional physics we did not implement. There are many aspects of our four-

dimensional environment that are forced or fixed that, in a true four-dimensional world, players would have more control over. Some of these actions include: looking down the w axis, reaching along the w axis, four-dimensional velocity, collisions when moving between w points and scaling objects depending on their shape and depth. While the theme for our game may not fully support all these features, perhaps a puzzle game would be the best setting to allow players to learn and exploit the fourth dimensions.

References

- Aitchison, A. (2014, Dec 30). *Adding Shadows to a Unity Vertex/Fragment Shader in 7 Easy Steps*. Retrieved from <https://alastaira.wordpress.com/2014/12/30/adding-shadows-to-a-unity-vertexfragment-shader-in-7-easy-steps/>
- Blender.org*. Retrieved from <https://www.blender.org/>
- Brokenvector. (2016, July 18). *Free-Low-Poly-Pack*. Retrieved from <https://www.cgtrader.com/free-3d-models/plant-tree/leaf-tree/free-low-poly-pack>
- Brownson, B; Miyazawa, T. (2010). *Koi Character by Toru Miyazawa*. Retrieved from <http://www.sharecg.com/v/42205/related/>
- Cg Programming/Unity*. (2016, August 28). Retrieved October 13, 2016 from https://en.wikibooks.org/wiki/Cg_Programming/Unity
- Definition and Meaning of Bonsai*. In *Bonsai Empire (What is Bonsai)*. Retrieved from <http://www.bonsaiempire.com/origin/what-is-bonsai>
- Garden Elements*. (2011, August 27). Retrieved October 13, 2016 from http://www.japan-guide.com/e/e2099_elements.html
- Gimp.org*. Retrieved from <https://www.gimp.org/>
- Harrison, A. *Traditional Stone Basin for Washing Hands*. Retrieved from https://usercontent2.hubstatic.com/12192339_f1024.jpg
- Japanese Gardens*. Retrieved from <http://www.aboutjapanesegardens.org/index.htm>
- Lancheres, E. (2016, July 20). *How HTC Vive is Outpacing Oculus Rift*. Retrieved from <https://www.engadget.com/2016/07/20/how-htc-vive-is-outpacing-oculus-rift/>

Lighting Textured Surfaces. (2015, August 26). Retrieved October 13, 2016 from

https://en.wikibooks.org/wiki/Cg_Programming/Unity/Lighting_Textured_Surfaces

Matou. (2014, March 29). *Transparent shader in background queue*. Retrieved from

<http://answers.unity3d.com/questions/494760/transparent-shader-in-background-queue.html>

Multiple Lights. (2016, August 21). Retrieved October 13, 2016 from

https://en.wikibooks.org/wiki/Cg_Programming/Unity/Multiple_Lights

Nixor. (2012). *Katana 3d model*. Retrieved from <http://tf3dm.com/3d-model/katana-75134.html>

Order-Independent Transparency. (2016, August 18). Retrieved October 13, 2016 from

https://en.wikibooks.org/wiki/Cg_Programming/Unity/Order-Independent_Transparency

Persona. (2010, September 21). *Pattern Detection*. Retrieved from

<http://answers.unity3d.com/questions/27831/pattern-detection.html>

Pruning Bonsai, Cutting Branches to Shape the Tree. In *Bonsai Empire*

(How-To, Bonsai Styling). Retrieved from

<http://www.bonsaiempire.com/basics/styling/pruning>

Rolo. (2011, March 1). *Comparing 2 Bitmaps*. Retrieved from

<http://answers.unity3d.com/questions/49935/compare-2-bitmaps.html>

Shader Reference. Retrieved from <https://docs.unity3d.com/Manual/SL-Reference.html>

ShaderLab: Blending. Retrieved from <https://docs.unity3d.com/Manual/SL-Blend.html>

ShaderLab: Culling & Depth Testing. Retrieved from <https://docs.unity3d.com/Manual/SL->

[CullAndDepth.html](https://docs.unity3d.com/Manual/SL-CullAndDepth.html)

Teoarch. (2011, April 8). *Similar Image Finder - .NET Image Processing in C# and RGB*

Projections. Retrieved October 13, 2016 from

<http://similarimagesfinder.codeplex.com/>

TheNinjassin. (2015, March 2). *Gesture Recogniton, Determining Shapes*. Retrieved from

<http://answers.unity3d.com/questions/914115/gesture-recognition-determining-shapes.html>

Vertex and fragment shader examples. Retrieved from [https://docs.unity3d.com/Manual/SL-](https://docs.unity3d.com/Manual/SL-VertexFragmentShaderExamples.html)

[VertexFragmentShaderExamples.html](https://docs.unity3d.com/Manual/SL-VertexFragmentShaderExamples.html)

Wooden Rock Garden Rake. Retrieved from [https://s-media-cache-](https://s-media-cache-ak0.pinimg.com/564x/27/68/9f/27689f2948cf745daaced81bc0e72e10.jpg)

[ak0.pinimg.com/564x/27/68/9f/27689f2948cf745daaced81bc0e72e10.jpg](https://s-media-cache-ak0.pinimg.com/564x/27/68/9f/27689f2948cf745daaced81bc0e72e10.jpg)

Writing Shaders. Retrieved from <https://docs.unity3d.com/Manual/ShaderOverview.html>

YourKoiPond. *Do Koi Eat Other Fish?* In Your Koi Pond (Home, Learn). Retrieved from

<http://yourkoipond.com/koi-eat-fish/>

Zen Wood. Retrieved from <http://zenwood.com/woodwork.html>

Appendix A: Playtesting Resources

A.1 Outline of Procedure

1. Have the participant put on the Vive headset and hand them the controllers
2. Ask the participant to move along the w axis using the triggers
3. Ask the participant to move to four specific colors, ensuring that he/she both try moving up and down as well as pausing between directions to ensure he/she stays on the target w point
4. Repeat steps 2 and 3 for the other three control schemes
5. Have the participant take off the Vive headset, then conduct a short interview using the questions from Figure 1 of the interview questions (Section A.2)
6. Inform the participant that he/she will now play a prototype of the game and give them detailed explanations of the game zones and how to activate the shrines, answering any questions he/she may have.
7. Have the participant put on the Vive headset with the game loaded to the home garden scene, the contract scroll set to load the Tokyo scene and the controllers mapped to the participants desired control scheme
8. Once the playthrough ends, have the participant remove the Vive headset, then conduct a short interview using the questions from Figure 2 of the interview questions (Section A.2)

9. Repeat steps 7 and 8, but with the contract scroll set to load the TokyoSolidColor scene instead. Make sure to inform the tester of the change from three dimensions to four and use the interview questions in Figure 3 of the interview questions (Section A.2)

A.2 Interview Questions

Figure 1: Experiment 1 Interview Questions

1. Which of the control schemes did you find the most comfortable or easy to use and why?
2. What about the other control schemes did you find uncomfortable or difficult to use?

Figure 2: Experiment 2 Interview Questions

1. Did you find the objects in the game easy to tend to, normal to tend to or hard to tend to?
2. What was the easiest/hardest objects to tend to throughout the game?
3. What changes would you suggest to make the game easier/harder to play?
4. Any additional comments or suggestions?

Figure 3: No 4D

1. Did you find this version of the game any more or less difficult to play?
2. Did you enjoy this version of the game any more or less than the previous version?
3. Any additional suggestions or comments on the game in general

Appendix B: List of Game Content

Listed below are the features and tasks completed by each person on the team throughout the duration of development.

	Completed Content	
Alex	John	Paul
Playtesting Schedule	4D Opaque Shader	Gravel Pit
Playtesting Tests	4D Transparent Shader	Pattern Recognition
Playtesting Report	Bonsai Tree	Rake
Setup VR in Unity	Bonsai Shrine	Gravel Pit Shrine
Setup VR Input for Grabbing Tools	Insecticide	Found Background Music
Setup VR Input for Insecticide	Shears	Added Background Music
Setup VR Input for Shears	Katana	Found Gravel Texture
Improve VR Performance	Scrolls	
Different Control Schemes	Garden Clock	
Hyper Object Code	Wash Basin	
Hyper Object Collision Code	Water Ladle	
Hyper Creature Code	Tintbox	
Fish Pool	Tokyo Skybox	
Fish Object	Home Level	
Fish Food	Tokyo Garden Level	
Reservoir Pool	Created Shears 3D Model	
Fish Shrine	Created Insecticide 3D Model	
Fish Manager	Created Wash Basin 3D Model	
Kami Object	Created Water Ladle 3D Model	
Kami Manager	Created Shrine 3D Model	
Tool Reset Box	Created Fish Seal 3D Model	
Resource Preloader	Created Bonsai Seal 3D Model	
Created Water Texture	Created Gravel Pit Seal 3D Model	
Animated Koi 3D Model	Created Kami House 3D Model	
	Created Tokyo House 3D Model	
	Created Tokyo Contract Scroll Texture	
	Created Controls Scroll Texture	
	Created How-To-Play Scroll Texture	
	Created Concept Scroll Texture	
	Created Executable Icons	

	Found Katana 3D Model	
	Found Koi 3D Model	
	Found Low-Poly Tree 3D Model	
	Found Low-Poly Rock 3D Model	

Table 2: Completed Content per Person

Listed below is the game content that was either cut from the final product or never reached completion during development.

Incomplete Game Content	Cut Content
Adaptive Help	Bonsai Contract Requirement
Tutorial Levels	Gravel Pit Wind Clearing
Kami Personalities	Menu Candles
Gravel Pit Rocks	Lighter
Gravel Pit Rocks 3D Model	Candle 3D Model
Rake 3D Model	Lighter 3D Model
Fish Food 3D Model	Wind Chimes
Colorblind Settings	Wind Chime hooks
Improved Tokyo Skybox	Wind Chime 3D Model
Blowing Leaves particle effect	Wind Chime Hook 3D Model
Ambient Sound Effects	
Game Sound Effects	

Table 3: Incomplete and Cut Game Content

Appendix C: Additional Interactive Media Class

A second part of our Major Qualifying Project was to work in groups with Japanese students on an interactive media project the students originally designed, and then meet with them every week in order to complete it. To do this, the group of WPI students split into pairs - John worked with our classmate Yuheng Huo, while Alex and Paul worked on their project together.

C.1 Alex and Paul's Interactive Project: Bubble Keyboard

The project that Alex and Paul's group initially suggested we work on was making a human-shaped interactive bubble. According to the original design, bubbles would be made to fill a human mold, then stood up. An image of a person would then be projected onto it, and the user would then be able to interact with it in some way. After discussing the logistics of making this human bubble, the design was overhauled so that the final product would be a bubble keyboard instead. As shown in Figure 67, how this would work is that a large plastic container would hold a bubble mixture and have a mesh net covering the top of it. A hole would then be cut in the side of the container and sealed with PVC piping, with a mini-leaf blower attached at the other end. Once the leaf blower was turned on, it would then blow air into the mixture, forming bubbles and pushing them out of the top through the mesh. From there, the user would put on a glove with two wires attached to one of the fingertips and connected to an Arduino.

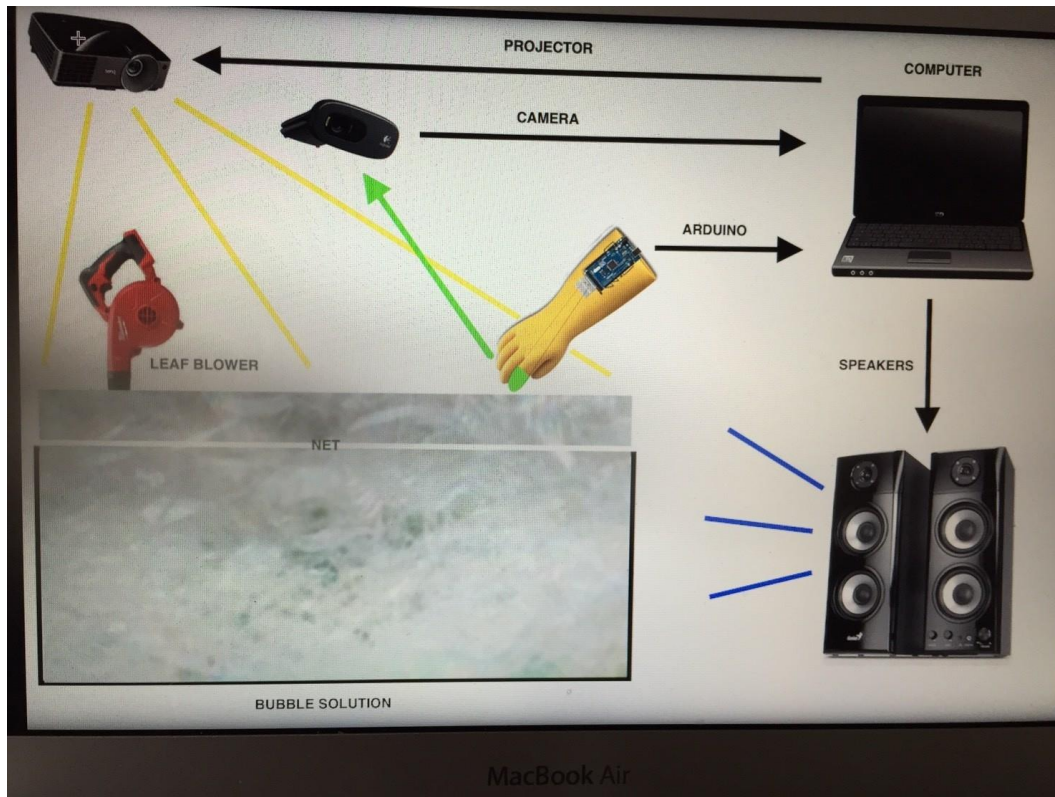


Figure 67: Mockup of the Bubble Keyboard Design

With this set-up, the user could touch the bubbles coming out of the container with the two wires. The bubbles would complete the circuit between the two wires, which would then send a signal to the Arduino. The Arduino, in turn, would then send a signal to the computer to play a note, which would then be output through a speaker. A camera would be stationed above the keyboard device to track where along the keyboard the user was touching, while a projector would then show an image on top of the mesh so the camera could track the glove. Based on the position of the user's finger, the note she would be playing from the keyboard would change.

After designing how the product would work and purchasing the necessary materials, the group went to work on constructing the keyboard. As seen in the above Figure 68, Alex, Paul, and two other group members first conducted an experiment with a voltmeter borrowed from Takemura Lab, and from this determined that bubbles carry a current. This would therefore mean

that the bubbles would be able to complete the circuit with the two wires on the glove. Alex was then able to get the Arduino we purchased to send signals to the computer. If more time was allowed, Paul would have worked on making the speakers project the keyboard's sounds from the computer. Unfortunately, due to busy work schedules outside of this project, the aforementioned work was the only progress that the group was able to make towards constructing the bubble keyboard, so we did not get to see it to its completion.

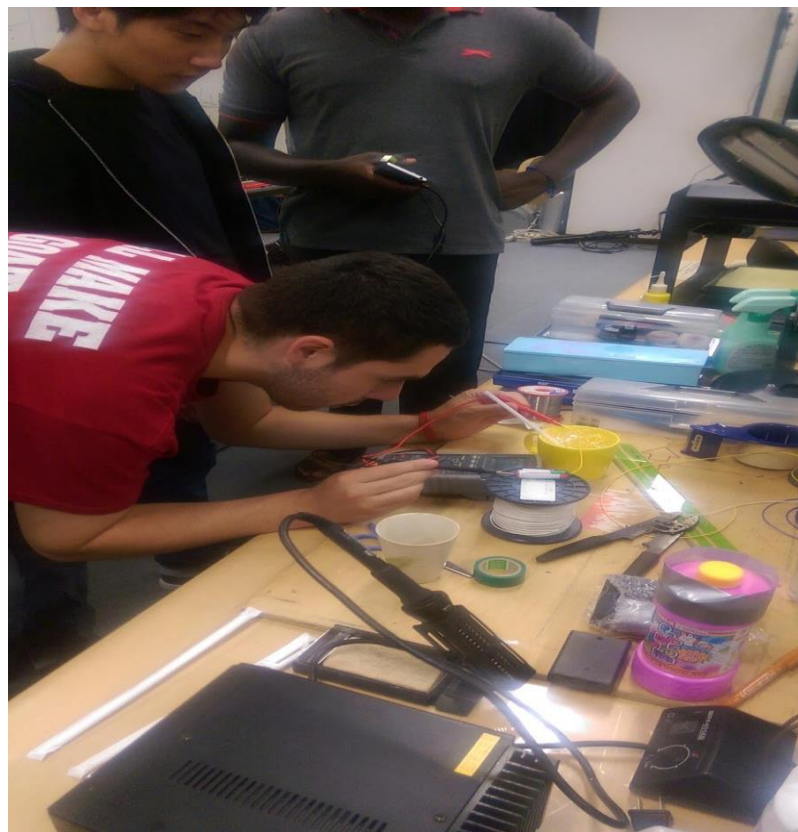


Figure 68: The Group Working with Bubbles to Measure if they would Conduct Current

C.2 John and Yuheng's Interactive Project: Rope Skipping Simulator

John worked on his additional interactive project with another WPI student working in Japan, Yuheng Huo. They worked with a team of Japanese students to make a device that simulated skipping a rope at various speeds. The idea was to replicate the act of spinning a rope

multiple times during a single jump without the danger of tripping on the rope. The device consisted of a laser line that would move down a sheet to give visual feedback of the 'rope' and hand devices with internal motors to provide haptic feedback. There would also be speakers playing the audio of a spinning rope.

John was tasked with creating the audio files meant to simulate the sound of a rope spinning multiple times in one jump. He used his iPhone to capture the audio of one of the Japanese students performing these multi-spins. He then imported the audio into Reaper and created sounds suitable for the device. The audio files were given over to the Japanese students once completed. Yuheng made a logo image to be used for the device.

John and Yuheng were limited in their ability to assist their Japanese partners because they could not communicate or actively participate in the project. They did not speak Japanese and the Japanese students did not speak English. The two parties had to speak each other's languages in broken language and use Google Translate to facilitate the process. On meeting days, they would take a bus to the Suita campus and walk to the students' lab to work till 7 PM. At the beginning of September, Japanese schools went on summer vacation and the buses stopped running. There was no way for them to reach the Suita campus. After providing the needed audio and image files, John and Yuheng no longer participated in the project.