COMP 4801 Final Year Project

VR Gamification of Open World Games

Final Report

The University of Hong Kong

Department of Computer Science

Student: ZHI Weichen (3035448550)

Supervisor: Dr. Choi Yi King

I

Abstract

The synergy between the development of Virtual Reality hardware and the interest in VR from the general public has been rising during the past decade. This project tries to bring out an immersive VR open world game with attractive game logic and immersive visual effects by first giving a literature review on the most welcomed and state-of-the-art virtual reality games and the potential to adapt it to open world games. A detailed set of project objectives will then be discussed, followed by the methodology this project applies to achieve each of the planned goals. Later, a demo representing the development progress will be shown as the result of the project, and a discussion of the limitations of the project will be given in the end.

II

Acknowledgement

We would like to express our deepest and most sincere appreciation to Dr. Choi Yi King, department of computer science, for her dedicated supervision of this project with her profound knowledge in computer geographics and project management. Throughout the entire project period has she been carrying out regular meetings for an agile and healthy progress of this project.

Furthermore, we would like to extend our thanks to the Department of Computer Science for their enabling the opportunity for students to carry out self-proposed projects as an incentive for the final year students to be motivated in their research.

Contents

Table of Figures

Abbreviations

The list of abbreviations appeared in this report is as follows:

ABBREVIATION	MEANING
VR	Virtual Reality
AR	Augmented Reality
XR	VR + AR
UE	Unreal Engine
FSM	Finite State Machine

1. Introduction

The bloom in the development of Virtual Reality hardware recent years have been an ecstasy to me of a virtual world with similarities with the world we live in, but with infinite possibilities for my exploration. Among all the application of VR, gaming is definitely one way with the most potential to experience the feeling of immersion and fantasy. According to a study by William J. et al (2017), VR games have gone superior above their traditional counterparts with respect to gamer satisfactory level.

2. Literature Review

As VR gaming being the focus of this project, this section will give a thorough analysis on some of the representative games that are largely welcomed in the recent decades. We will try to discuss the unique feature regarding the gameplay for each game. Specifically, we will conduct our discussion on famous games from two perspectives: 1) Open World games, and 2) VR games. Last, we would try to throw a quick insight into the popular gaming engines required to build a game in 2024.

2.1. Open World Games

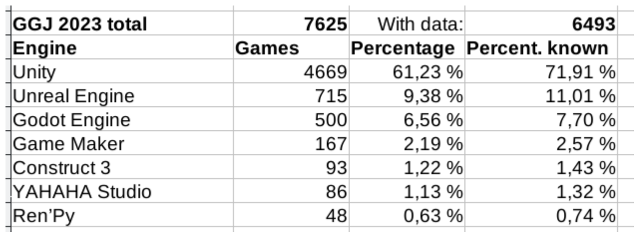
With a non-linear gameplay and storytelling, open world games have gone viral ever since its appearance (*Computer and Video Games*, 2008). Such games are designated to bring the player to a world where they can explore freely with or without objectives. The bottom-up system-level designed game logic allows players to exploit their imagination to interact with the world. Some of the most well-known open world games includes *Red Dead Redemption 2*, *Legend of Zelda Series* and *Grand Theft Auto Series*.

2.2. Virtual Reality Games

Never had the world imagined such possibilities of the interaction VR could offer until Valve brought out its iconic first-person VR-based dystopian *Half Life: Alyx* in 2020. As the winner of "The Best VR/AR" at Game Awards 2020, the successor of the famous franchise perfectly combined combat, puzzle solving and exploration with an utterly immersive atmosphere. The success of *Half Life: Alyx* seems to have started the era of VR AAA gaming, and has set up a pole for the late comers such as Resident Evil 4 Remake (2023) and Assassin's Creed Nexus (2023).

2.3. Game Engines

Game engines is the framework used when designing video games. It typically includes relevant development environments, necessary libraries and additional tools that are helpful in the game development process. According to the statistics from Global Game Jam (2023), the most popular game engines are Unity and Unreal Engine which contributes 71% and 11% of the total games delivered. As for the scope of this project, we will introduce the most famous game engine, Unity and Unreal Engine, and try to justify our choice of Unity in the later Section 4.1 Framework.



GGJ 2023 total	7625	With data:	6493
Engine	**Games**	**Percentage**	**Percent. known**
Unity	4669	61,23 %	71,91 %
Unreal Engine	715	9,38 %	11,01 %
Godot Engine	500	6,56 %	7,70 %
Game Maker	167	2,19 %	2,57 %
Construct 3	93	1,22 %	1,43 %
YAHAHA Studio	86	1,13 %	1,32 %
Ren'Py	48	0,63 %	0,74 %

Figure 1 Game Engine Popularity in 2023

2.3.1. Unity

Unity is first announced in 2005 as a game engine designed for MacOS. It has then been extended to the development of applications compatible with PC, mobile, console and XR platforms. This gradual extension of enables Unity's nature of cross-platform support. In fact, Unity SDK has been a choice for a monopolistic 70% of mobile game developers according to the data from data.ai (2023). Besides, Unity introduces a repertoire of out-of-the-box components that eases the accessibility to game developers.

2.3.2. Unreal Engine

For those advocates of AAA games, the name of Unreal Engine must have proceeded. UE is also a popular game engine developed by Epic Games since 1998. In fact, UE has contributed many of the most phenomenal games in recent years such as Cyberpunk 2077 (2020), Final Fantasy VII Remake (2020), Star Wars Jedi: Fallen Order (2019) and Fortnite (2017). In 2014, Guinness World Records even named UE as “the world's most successful video game engine”. However, there has been debate on this famous engine for its steep learning curve until the latest release of Unreal Engine 5 in April

2022. In general, UE offers less graphical programming tools and is hard to master when compared to its young counterpart, Unity.

3. Objective

This project aims to bring out an open world game that supports Virtual Reality. Considering our discussion on the manifestations of gameplay that make the game outperform its counterparts in Section 1, we further split the objective into sub-objectives. Specifically, we want the final product to outstand itself with features in two major aspects: 1) gameplay, and 2) visual effect.

3.1. Features: Gameplay

This part introduces the goal of general game logic of the final product, i.e., player inputs and character control, system-level interaction and a combat system.

3.1.1. Input and Character Control

As specified previously, the support for VR is one of the major targets of the project. Hence, we expect the game enables the player to look and move around in the game world we established with a 6-DoF VR headset and controllers. Specifically, the difference between 3-DoF and 6-DoF control is that the latter one supports movements along the x-y-z axes, as shown in the chart below:

Head Movement	Roll	Yaw	Pitch	x-axis movement	Y-axis movement	z-axis movement
6-DoF	✔	✔	✔	✔	✔	✔
3-DoF	✔	✔	✔			

Table 1 Input Checklist of Different DoF

Also, the player should be able to use a pair of controllers for hand tracking and therefore to achieve more functionalities involving interactive game logic.

3.1.2. Interactions

On top of the player inputs, we can now carefully design the interactions between the player and the world, that is, what the player can do in the world. Though we can apply a lot of abstractions with respect to the models and textures of the game objects, we still want the interactions to be as realistic as possible, which corresponds to the immersive nature of VR as discussed in introduction part.

The first notable interaction between the player and the world is the physical effects which could be experienced when moving around and attacking enemies in the open world. Therefore, the goal of this part is to implement basic physics such as gravity and collision detection.

It is also worth noting that in all welcomed VR games, grabbing object is a must. In fact, *Half Life: Alyx* has implemented an addictive 'gravity gloves' which let the pointed object to fly into the hands of the player with a shake of wrist. This idea saves the effort for the player to actually bend down and pick up objects in person, yet still remains the interaction with the world. We also want to give our player an easy access to pick up objects on the ground.

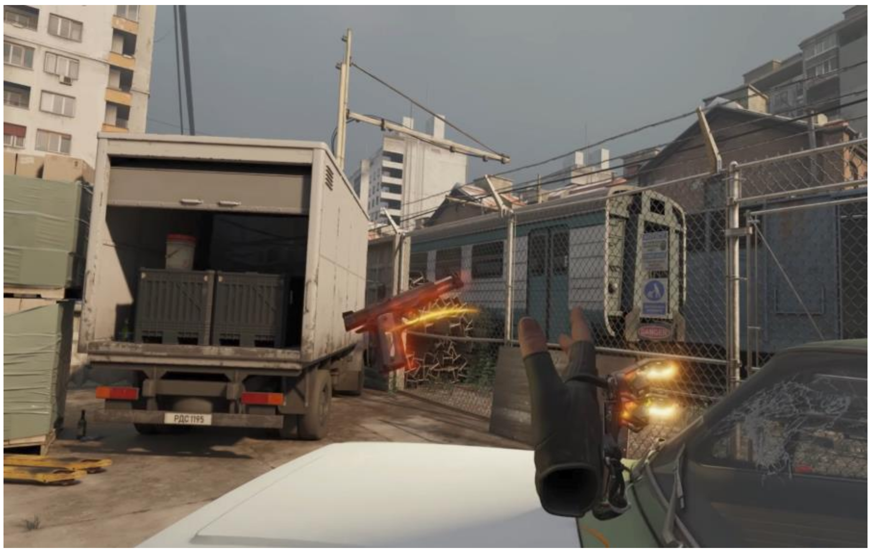


Figure 2 Gravity Gloves from Half Life: Alyx

3.1.3. Combat System

An open world game shouldn't adopt limitations on how the player attacks the enemy. This project received insight from *Legend of Zelda: Breath of the Wild*, where the player can attack the enemies with weapons just picked up from the ground.

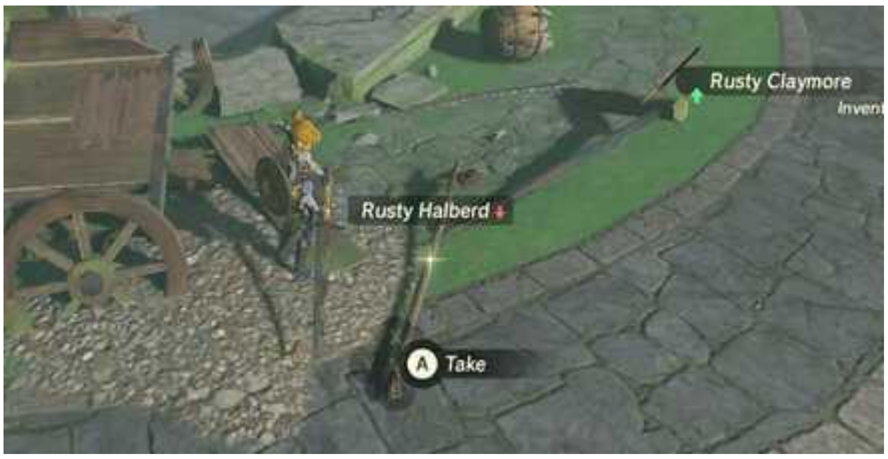


Figure 3 Various Pickable Weapons in Legend of Zelda

Also, the means of attack should be weapon-based, so that the player gets the incentive to try out all the weapons he or she could possibly have access to. For example, there should be pickable melee weapons such as swords and glaives, there should also be long-range weapons such as bows, and magic wand for AOE attacks. Furthermore, the enemies should have health bars which helps diversify different means of attack by applying different damage statistics.

3.2. Features: Visual Effects

After talking about the game logic, this section starts the discussion on visual delivery of the game.

3.2.1. Spherical Bending Effect

The *Animal Crossing: New Horizons* went viral following the outbreak of Covid-19 pandemic when most people are spending their quarantine time at home. The game portrays a cute tiny island where the player can live with his or her animal friends. This project got insight from the special cylindrical, scroll-like bending effect of the world. This effect provides the players a top-down view yet remains their sight to the sky and the objects in the distance, just as the quote of *New Horizons* as its name.

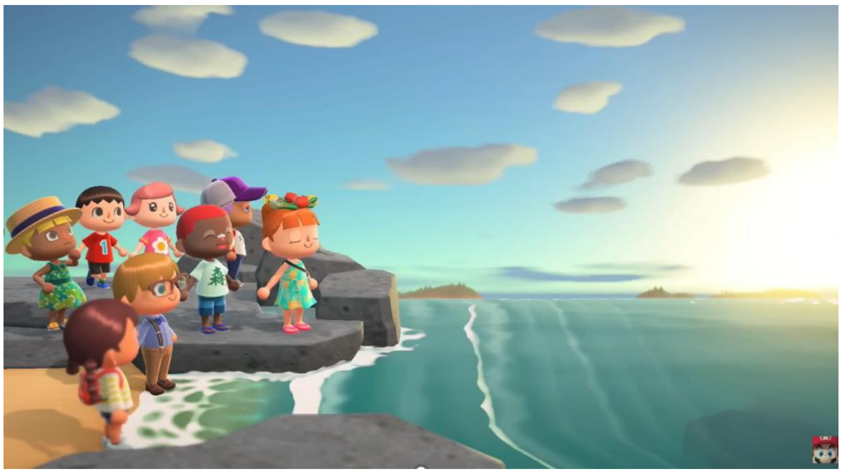


Figure 4 Scroll-Bending Effect of Animal Crossing: New Horizons

Based on this bending effect, we would like to further extend the scroll-bending effect into a spherical bending effect, in view that the players' view in our game is not a fixed top-down view but a movable one controlled by head movement.

3.2.2. Ocean Water Simulation

The simulation of water has been an ever-evolving question in the area of computer graphics. Not only does it include a physical change of shape for ocean waves, but the simulation also needs to consider the reflection of light, the obscuration of objects whose parts immersed in water, and potential foam effect caused by waves' impact. This project also tries to make the greatest effort on simulate the water as realistic as possible.

4. Methodology

This section aims to address the methods used in the development of the project.

4.1. Framework

This project applies the bottom-up approach when deciding the framework to be implemented. Recalling our background research on game engines in Section 2.3, we will now justify our choice of Unity. Then, the structure of the application will be discussed.

Unity is written in C++ for runtime and in C# for its scripts. It exploits the component-orientation capability of C# by providing a bunch of built-in Unity components for easier access and faster development.

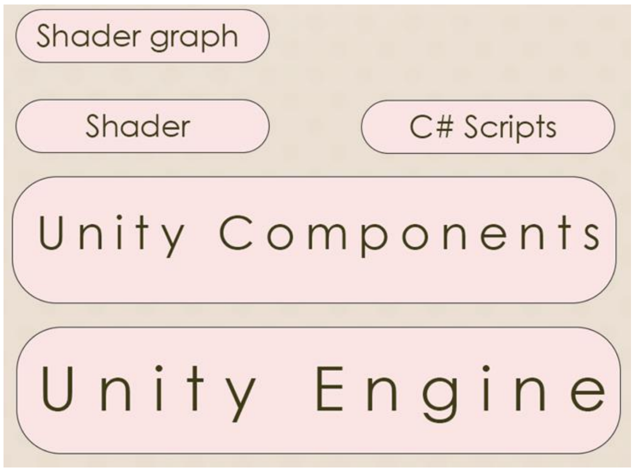
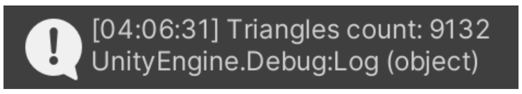


Figure 5 Unity Project Structure

However, when considering the performance of Unity compared to its UE counterpart, since C# has done the trivial yet necessary steps such as memory management and garbage collection, the performance of it, is compromised due to the infeasibility of program-level customed optimization for the agnostic developers. It's now worth discussing whether this project would need additional optimization to achieve the required performance.

We note that Unity meshes are consists of mere triangles. This project then executes an original triangle counting script which constantly retrieve the number of triangles from the *sharedMesh* attribute of <MeshFilter> component from every gameObject to be rendered in the scene. As the result, we achieved a total count of around 9k triangles in the whole scene. The number is of minor performance bottleneck compared to a normal Unity project with millions of triangles. The reason for that is we were using simple models for abstraction of entities. Therefore, the performance of Unity is good enough for the scope of this project.



```
[04:06:31] Triangles count: 9132
UnityEngine.Debug:Log (object)
```

Figure 6 Triangle Count

Last but not least, thanks to the cross-platform support of Unity as mentioned before, we can easily establish the bridge between the software and the VR hardware inputs. Unity natively offers an *XR Interaction Toolkit* based on its action-based *Input System*.

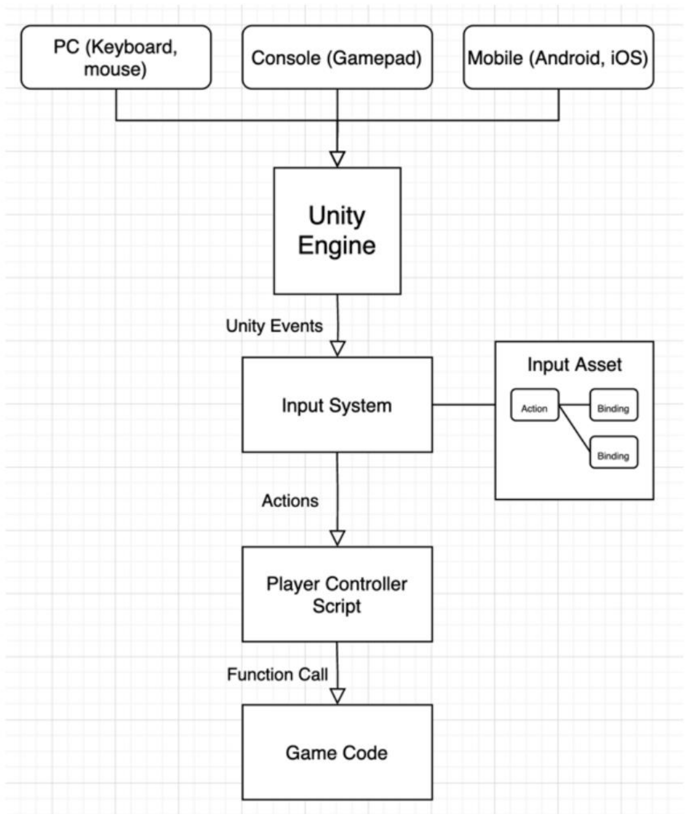
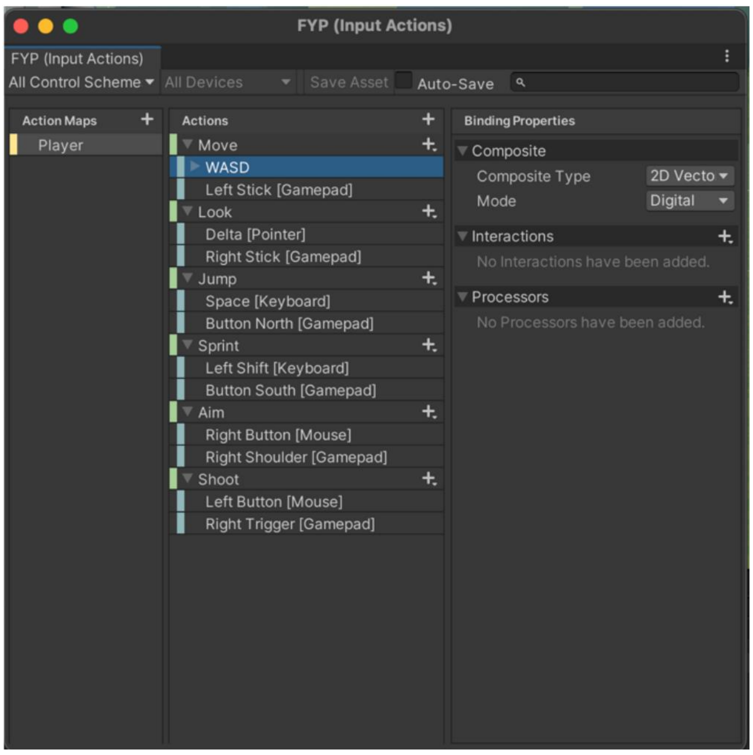


Figure 7 Unity Input System Workflow



FYP (Input Actions)

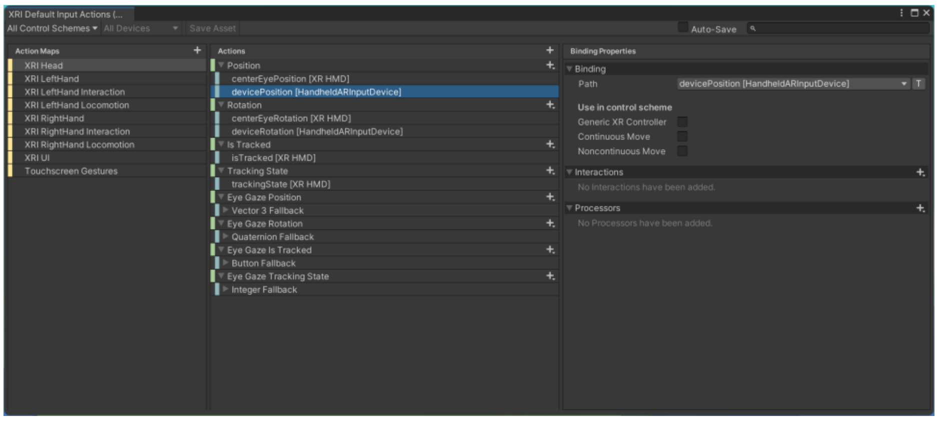
FYP (Input Actions)
All Control Scheme | All Devices | Save Asset | Auto-Save

Action Maps
- Player

Actions
- Move
  - WASD
  - Left Stick [Gamepad]
- Look
  - Delta [Pointer]
  - Right Stick [Gamepad]
- Jump
  - Space [Keyboard]
  - Button North [Gamepad]
- Sprint
  - Left Shift [Keyboard]
  - Button South [Gamepad]
- Aim
  - Right Button [Mouse]
  - Right Shoulder [Gamepad]
- Shoot
  - Left Button [Mouse]
  - Right Trigger [Gamepad]

Binding Properties
- Composite
  - Composite Type: 2D Vecto
  - Mode: Digital
- Interactions
  - No Interactions have been added.
- Processors
  - No Processors have been added.

Figure 8 Example of Action Map



Action Maps
- XRI Head
- XRI LeftHand
- XRI LeftHand Interaction
- XRI LeftHand Locomotion
- XRI RightHand
- XRI RightHand Interaction
- XRI RightHand Locomotion
- XRI UI
- Touchscreen Gestures

Actions
- Position
  - centerEyePosition [XR HMD]
  - devicePosition [HandheldARInputDevice]
- Rotation
  - centerEyeRotation [XR HMD]
  - deviceRotation [HandheldARInputDevice]
- Is Tracked
  - isTracked [XR HMD]
- Tracking State
  - trackingState [XR HMD]
- Eye Gaze Position
  - Vector 3 Fallback
- Eye Gaze Rotation
  - Quaternion Fallback
- Eye Gaze Is Tracked
  - Button Fallback
- Eye Gaze Tracking State
  - Integer Fallback

Binding Properties
- Binding
  - Path: devicePosition [HandheldARInputDevice]
  - Use in control scheme
    - Generic XR Controller
    - Continuous Move
    - Noncontinuous Move
- Interactions
  - No Interactions have been added.
- Processors
  - No Processors have been added.

Figure 9 Action Map for XR Toolkit

As shown in the topology above, the *XR Interaction Toolkit* does the job of the *Input System* to detect hardware input in the form of Unity Events and call the action scripts according to the action map. The remaining job is scaled down to define the triggered actions in C# scripts such as moving the character and attacking the enemy, after which a player is able to move around and look around with tracked hands in the virtual reality world.

4.2. Implementation: Gameplay

4.2.1. Interaction

Thanks to Unity's abundant arsenal of components for physics, the realization of interaction part is done with minimal detour. For the general physics such as gravity and collision between rigid bodies including player, environment and enemy, Unity's *Rigidbody* and *Collider* component quickly tackles the question by simply designating parameters required for the calculation of locomotion.

As for non-rigid bodies' collision, after setting the *isKinematic* parameter according to the regulations of Unity, we can then successfully detect whether an object is in some certain area, which helps us in the implementation of guarding enemy and AOE attack determination. Specifically, after an enumeration for object colliding with different parameters, we summarize the behavior of rigid body collision detection as concluded in the following chart:

Object 1				Object 2				Result
Collider?	**Rigid**	**Kinematic**	**Trigger?**	**Collider?**	**Rigid**	**Kinematic**	**Trigger?**	
✔				✔				Nothing happens
✔				✔	✔			Both calls onCollision Method. Object 2 bounces back, Object 1 not move
✔	✔			✔	✔			Both calls onCollision Method. Both bounces back.
✔	✔	✔		✔	✔			Both calls onCollision Method. Object 2 bounces back, Object 1 not move.

✔			✔	✔				Nothing happens
✔			✔	✔	✔			No bounces back, Both calls onTrigger None calls onCollision
✔	✔		✔	✔	✔			No bounces back, Both calls onTrigger None calls onCollision
✔	✔		✔	✔	✔		✔	No bounces back, Both calls onTrigger None calls onCollision

Table 2 Rigid Body Collision Behavior

Then, for grabbing objects in our application, we first overload the *XR Grab Interactable* component by specifying the holding position and direction after attaching the component to the grabbable model such as guns, we then define a grab action in the *XR Input Map* as mentioned in Section 4.1. We are hence able to grab an object from ground by pressing the hold button which is designed at the middle finger position on the controllers.

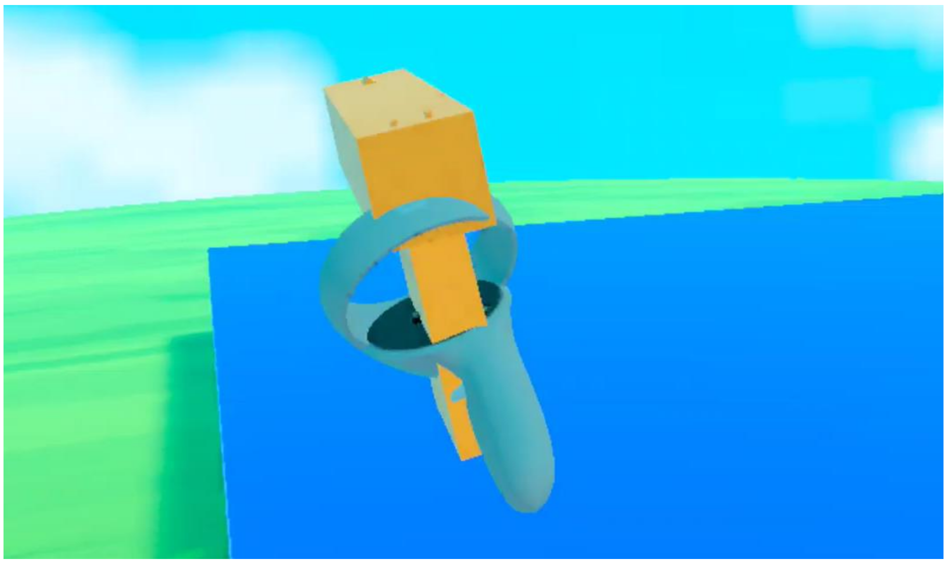


Figure 10 Grabbing a Gun in Our Application

4.2.2. Combat System

Now that we can grab up weapons from ground, it’s time for the actual design of a practical combat system. This project put a great emphasis on modeling the behavior of the hostile characters. From the review of the success of Legend of Zelda in Section 2.1, we conclude that a successful depiction of the enemy’s behavior largely bolsters the immersion of the open world since the enemies are actually alive characters with flesh and blood. This project introduces the method of Finite State Machine (FSM) to simulate the behavior of enemies.

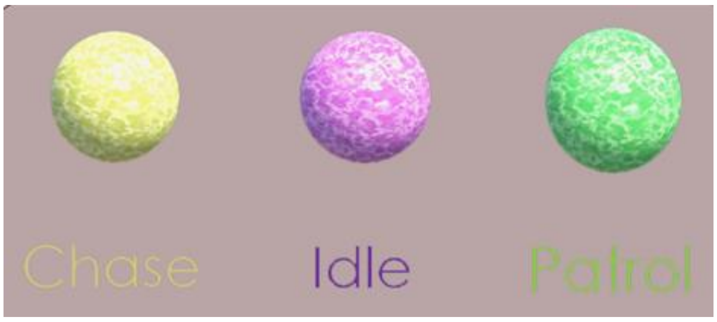
The key mechanism of FSM is to make abstraction from the characters’ behavior and implement a set of pre-defined action states with another set of state transition conditions. After considering the scope of this project, the basic enemy’s state set of our game consists of four different action states: 1) Idle, 2) Patrol, 3) Chase and 4) Hit.

Also, this topology defines the set of transition conditions among the states, which is listed below:

No.	Condition	State Transition
1	Enemy not moving for 1 second	Idle => Patrol
2	Enemy reaching one of the patrol points	Patrol => Idle
3	Player entering enemy's guard range (sight)	Patrol/Idle => Chase
4	Player gets out the guard range of the enemy	Chase => Idle
5	Enemy being hit by player	Idle/Patrol/Chase => Hit

Table 3 Transition Condition for FSM

Specifically, the manifestation of the enemy with different state is by using a moving sphere in respective color. We believe that different color is intuitive and straightforward when representing the change in enemy behaviors.



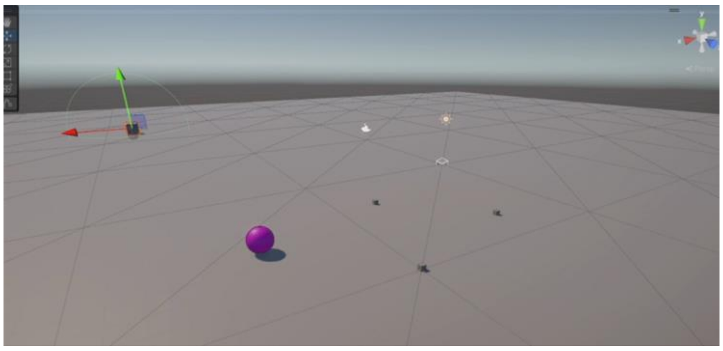


Figure 11 FSM Demo Scene with Stated Enemy

4.3. Implementation: Visual Effects

After demonstrating the implementation of gameplay related features, this section now describes the effort to achieve the objectives of visual effects as stated in Section 3.2. Specifically, the two visual effects in question are achieved with techniques of shaders. Shaders are the program that directly instruct the graphics card on rendering of objects including the position of vertices (vertex shader) and color of each "fragment[1]" (fragment shader).

4.3.1. Spherical Bending Effect

Recall the objective of implementing a spherical world in Section 3.2.1, the reason we choose shaders instead of actually build a spherical model for the world is from two aspects:

[1] The unit of work for rendering at most one pixel, product of rasterization by converting vector shapes into rasterized ones.

1) By creating an actual spherical model where all other game objects reside on, we need to overwrite all the interaction part including the physics since Unity only supports the necessary physical effects for a flat surface. The work to re-write a physics system is bulk and non-robust.
2) Since the model for a world should be relatively larger than most of the objects reside on it, the resolution of the spherical model is restricted to such high a level that being unaffordable to mobile platforms including the all-in-one VR headsets.

Considering the two factors above, we decided to turn for shaders to solve the tricky question. After some careful design we found that shaders, when applied to render a shape, are able to conduct a conditional mapping from the actual position of a vertex to a tweaked one. Therefore, the principle of mapping is shown in the diagram below:

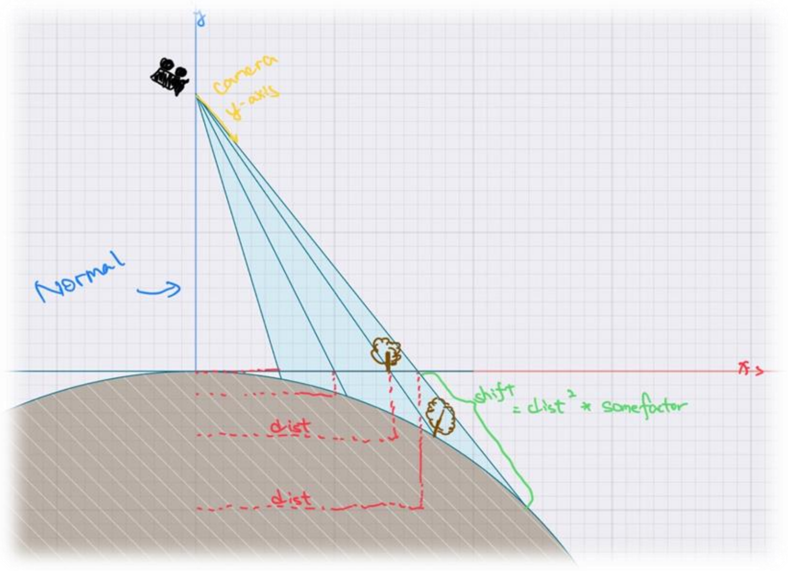


Figure 12 Principle of the Spherical Bending Vertices Mapping

The principle of this vertex displacement mapping is adding an additional in-line SHIFT value to the original camera-object distance for the graphics card to render. The SHIFT value is designed to be proportional to the power of horizontal distance (DIST) between the camera and the object. That is to say, when we look top-down on the camera, the SHIFT value will increase in a circle with the camera as the center. In another word, all objects will be moved further from the view of the camera with respect to their horizontal distance to the camera.

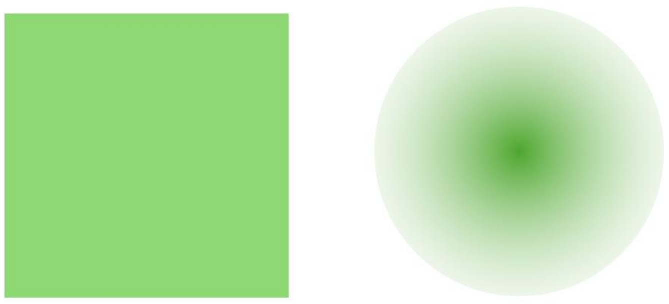


Figure 13 Top-Down View Before/After Vertex Mapping

As visualized in Figure 13, we treat the left square as the set of distance between each vertex to our eyes when looking top-down to the center of the square, the plain color suggests how we perceive this set of distance without a vertex displacement (direction: inward the paper). After applying a displacement to every vertex inward the paper, we perceive the new distance to any vertex as ever further as the increased distance from the center, and we now perceive a round shaped gradient color which suggests the further the original horizontal distance of a vertex is from the center, the further it would be sent inward the paper, and a spherical effect is achieved without actually manipulating the fundamentals of the scene, which therefore saves the demo from overwriting physics and the problem of unaffordable resolutions.

For actual implementation, we went for the native Shader Graph from Unity, which is basically a graphical programming tool that allows the developers to represent elements in the algorithm with graphical nodes such as input/output, operands and functions, and then connect the nodes to establish a flow of operation to represent the entire algorithm.

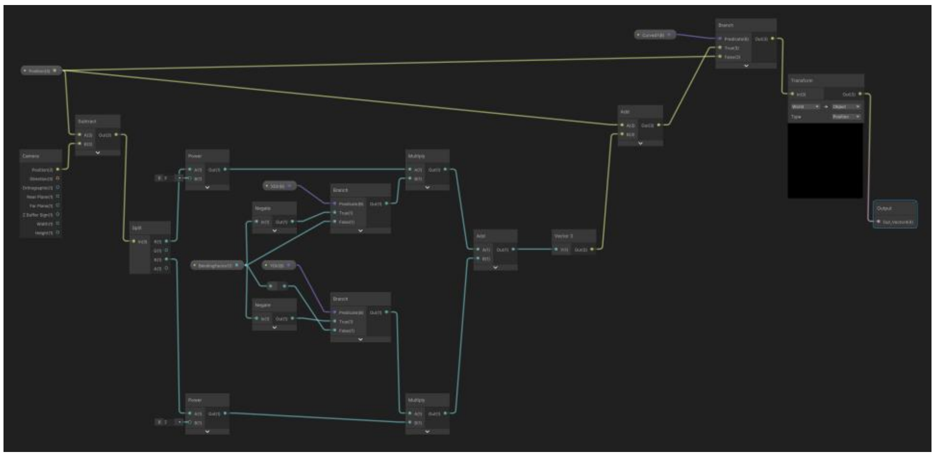
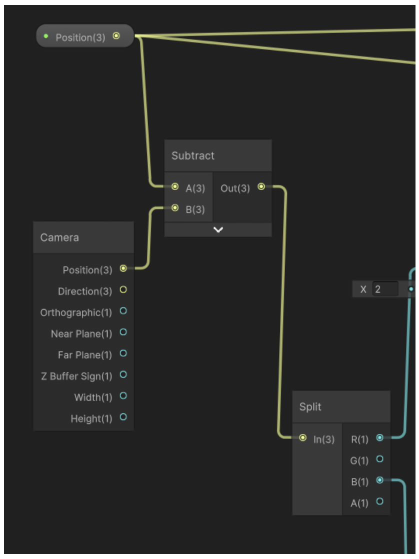
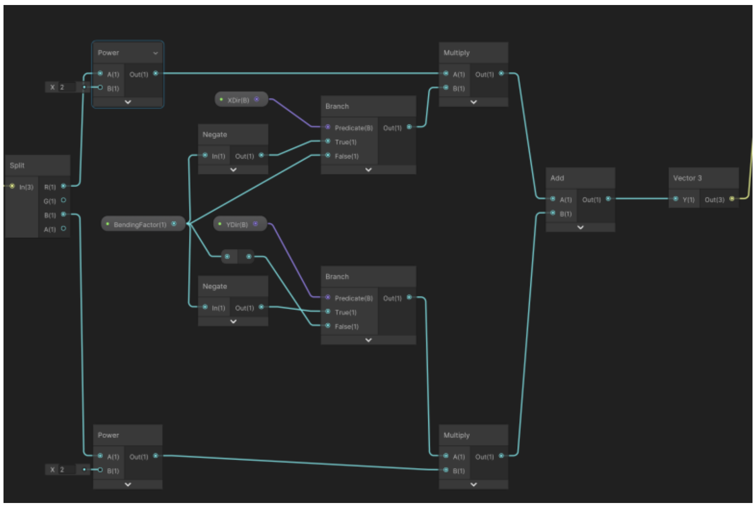


Figure 14 The Shader Graph for Spherical Bending (Subgraph)

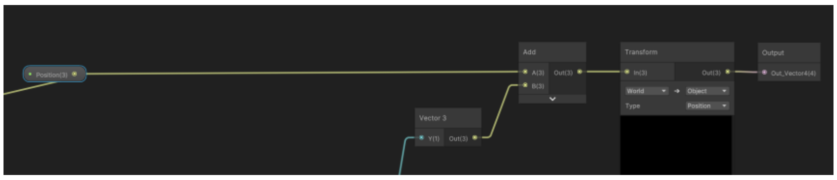
As shown in figure 14, the execution flow of the graphical shader still follows the principle discussed previously: adding a shifted value to camera-object distance proportional to the horizontal distance of camera and object. To further clarify the logic, we will try to illustrate the graph in pieces as the last part of this implementation.



This part first gets the original *Position* of each vertex and the camera as the input and then do the subtraction to get a vector pointing from the camera to the vertex. Last, we split the x-y-z axes and only taking x and z axes (horizontal plane) to calculate the square of horizontal distance with Pythagorean Theorem.



After the calculations of horizontal distance, we now apply a custom *BendingFactor* as a parameter to control the scale of bending.



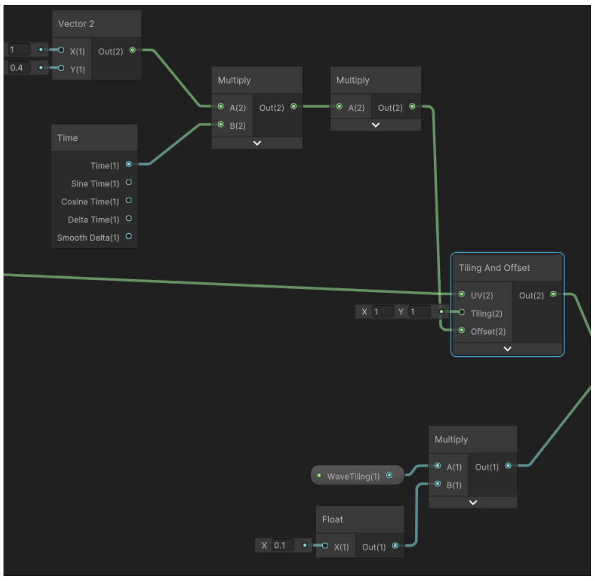
Finally, we add the additional SHIFT value in-line with the camera-object distance to the original position of the vertex, and a displacement of the position is completed.

4.3.2. Ocean Water Simulation

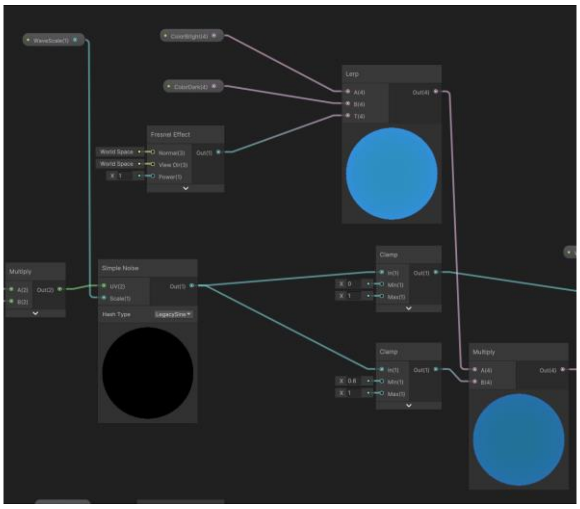
After the successful implementation of vertex displacement shader for the spherical bending effect, we would like to further exploit the potential of shader to simulate the ocean water. After considering the scope of this project, we abstracted the behavior of ocean water into two aspects: 1) simulation of waves and 2) reflection of water.

● Simulation of Waves

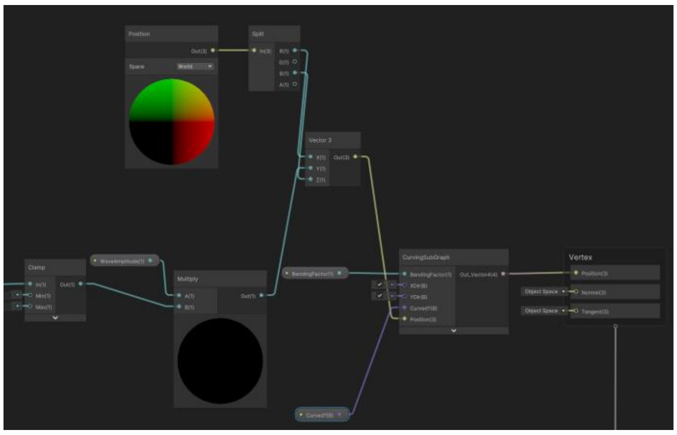
The simulation of waves' movement is somewhat similar to the approach used in bending the world, that is, vertex displacement in vertical axis. However, the displacement this time is no longer corresponding to the camera. On the contrary, the displacement of every vertex of a water plane is decided by a periodic noise multiplied by its local position, which allows the wave to be ever raging without the movement of camera.



The first part of the shader here shows the step to create time-driven periodic parameters, which utilizes the C# Time library and will later be fed into the generation of simple noises.



The next part first shows the use of Fresnel Effect to illustrate the gradient of strength of reflection corresponding to the time-driven noises to create the color difference between peak and trough of the waves.



Last but not least for the vertex displacement, we retrieve the position of every vertex in a water plane and apply the time-driven noise multiplied by the a certain vertical shift (along y-axis), and a wave with vertical movement and different color is then accomplished.

- **Reflection of Water**

The reflection of water is calculated by a symmetry transpose of the incident light along the normal vector of the surface. Objects in real world merely have a complete smooth surface with all normal vectors being vertical to the general surface. Hence, we apply a technique called normal map to the project for a more realistic and more diffused reflection.

The key to a normal map is a "normal texture" that being a picture, instead of storing colors, it keeps a set of angle deviations for all normal vectors along the surface. The fragment shader can therefore conduct the calculation of reflection light according to the normal angle at every point stored in the normal texture.

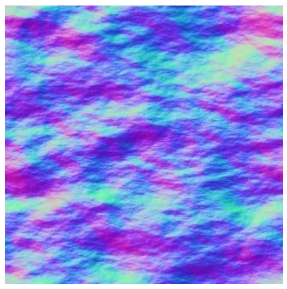


Figure 15 Normal Texture Applied in the Project

The implementation of normal map is also done in the form of shader graph. What's worth noting is that we actually applied two normal maps in orthogonal direction with blending for a more unpredictable manner of reflection. Yet the principle remains the same, that is, to use the information of deviation of normal vectors stored in normal texture to achieve a more diffused calculation of reflection light.

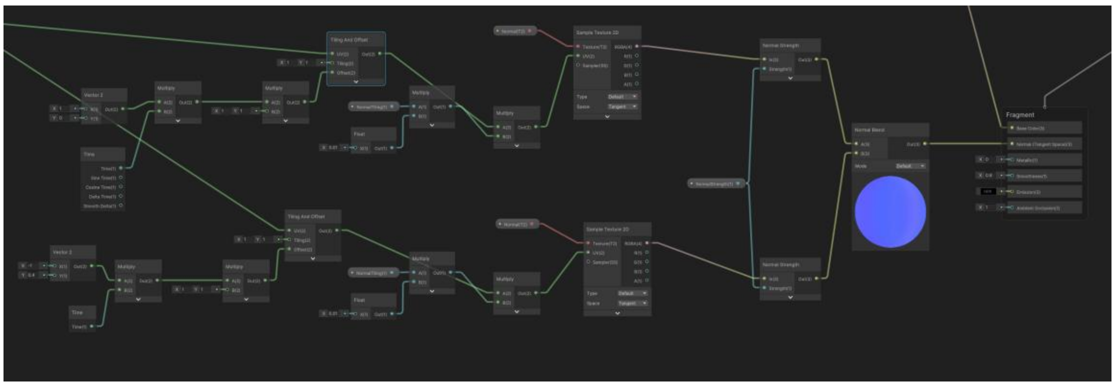


Figure 16 2-Normal-Map Blending

To wrap up the part of water simulation, we include in the following part the editor-view of the same water plane but in different forms: 1) plain water plane, 2) single normal map with Fresnel Effect and 3) Normal Blend with wave simulation.

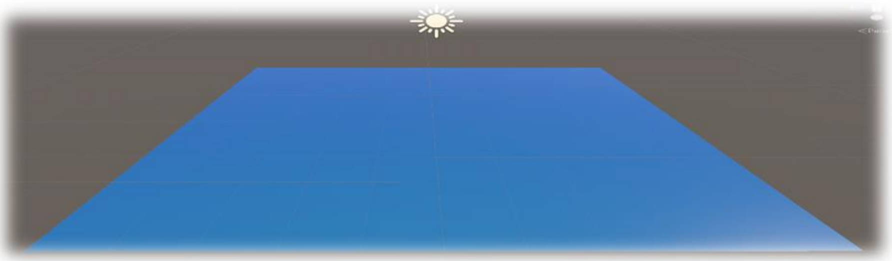


Figure 17 Plain Water Plane

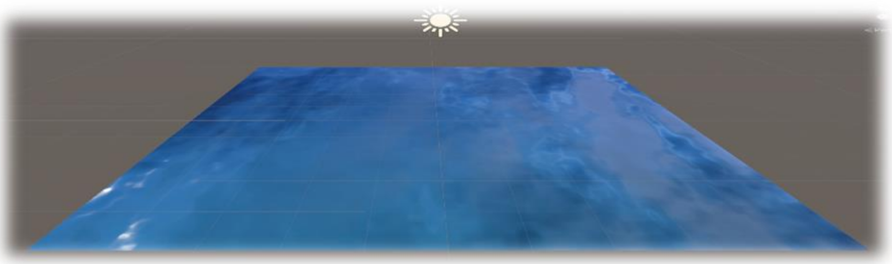


Figure 18 Water with Single Normal Map and Fresnel Effect

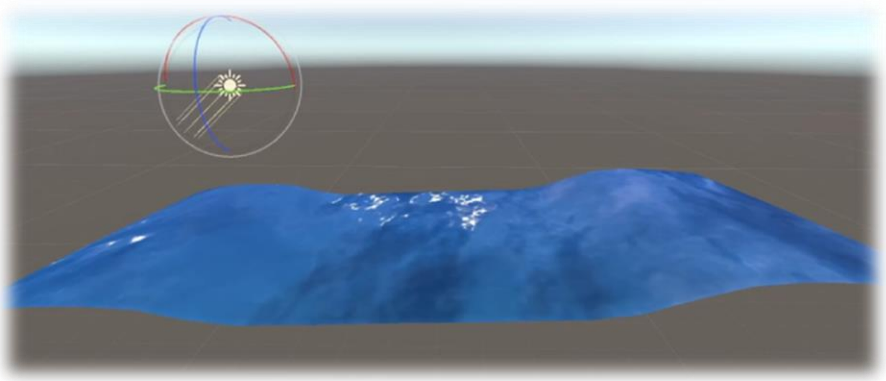


Figure 19 Final Water Effect with Normal Blend and Wave Simulation

5. Results

After the two-semester development period, we are happy to introduce the final product. This section identifies the features of the product with in-game screenshots, which correspond to the methodology used for the implementation.

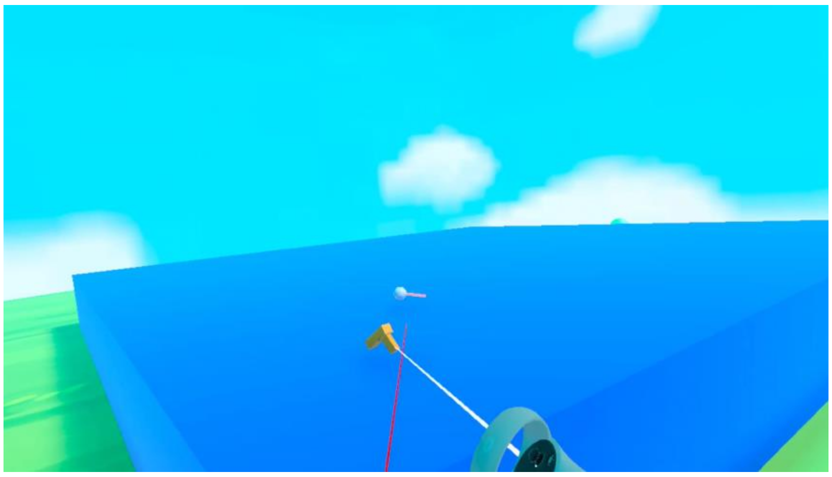


Figure 20 In-game Player Control Interface

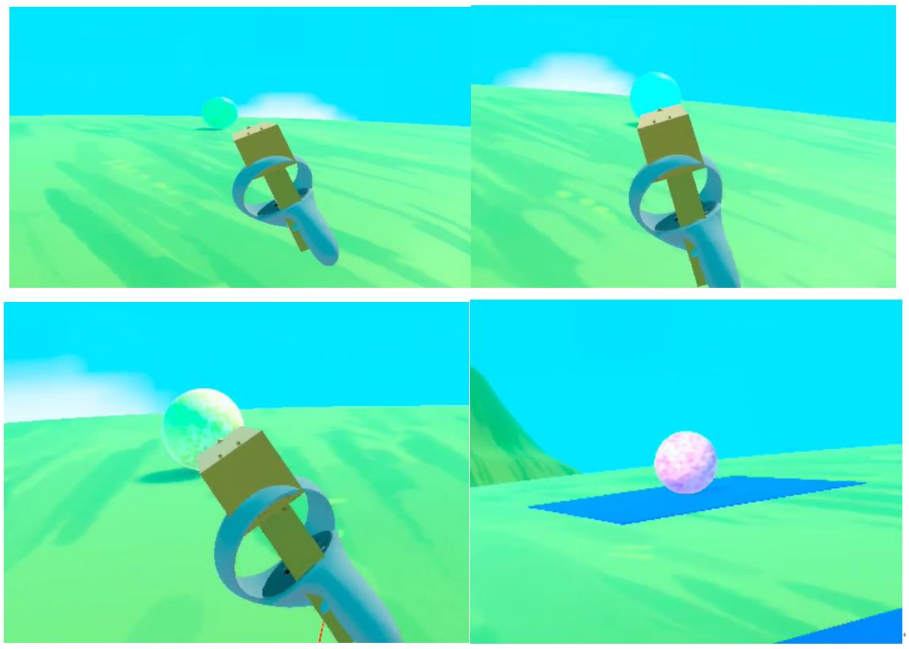


Figure 21 FSM with Patrol, Hit, Chase and Idle State

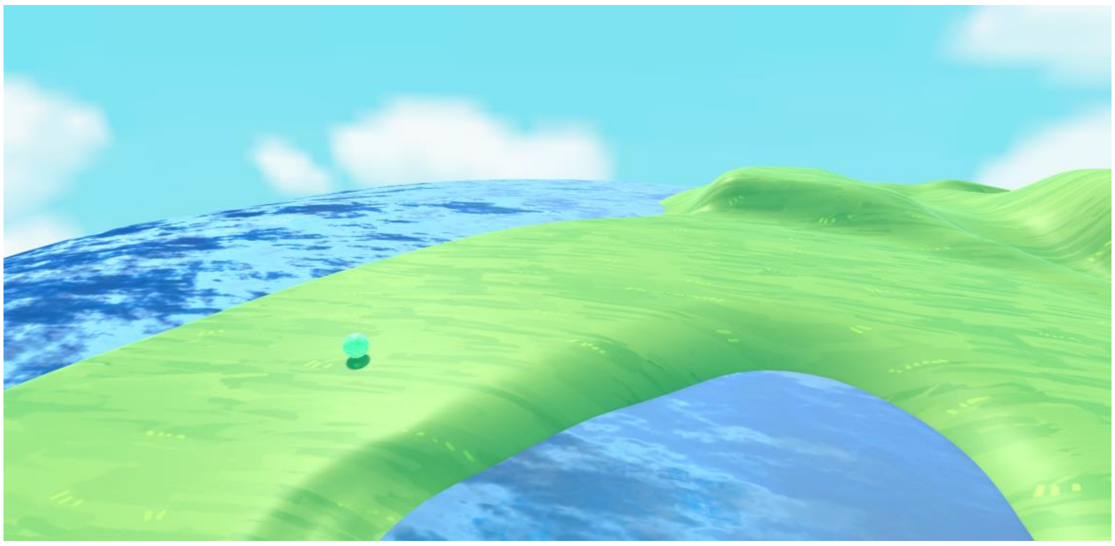


Figure 22 Spherical World with Vertex Displacement Shader

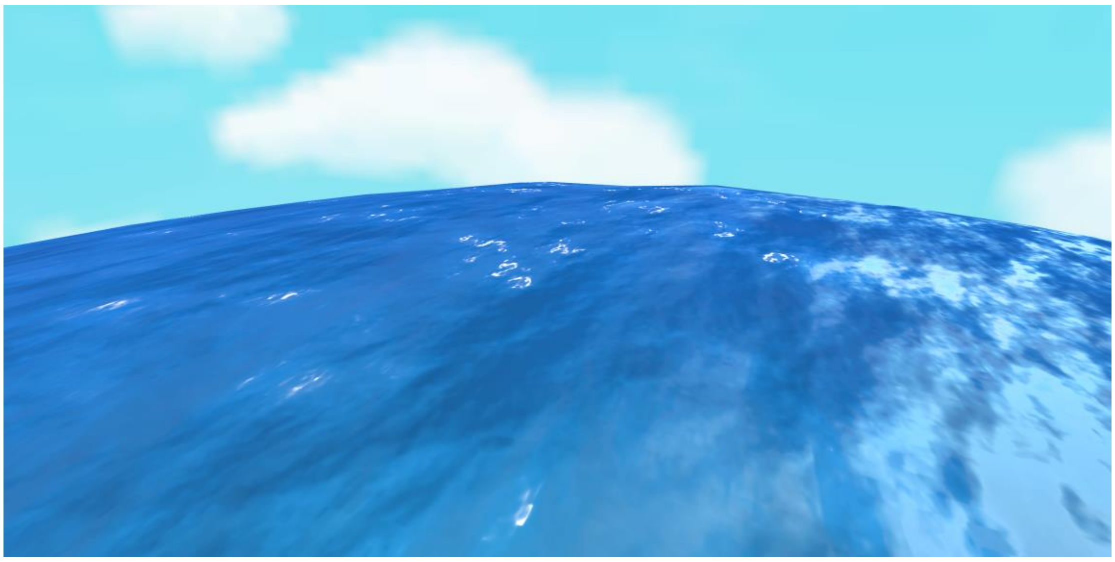


Figure 23 Water Simulation with Vertex and Fragment Shader

6. Limitations and Future Works

Despite the maximized effort in development and management of the project, a delivery to perfection can still not be guaranteed. This section tries to give a discussion on the limitations of the project due to limitations in technology stack and human resources. Then a virtual roadmap will be shown under the assumption that the project would carry on in the future.

6.1. Model and Animation

As the project put more emphasis on game logic feasibility, the manifestation of enemy and interactable objects are largely abstracted. For example, the enemies are represented by spheres with different colors to represent its current state, also, the equipment (pistols and magic wands) are represented by a simple combination of cubes and spheres.

To create a set of more decent original models, it would require much more time to study modeling application such as blender 3d. However, since Unity also provides animation components that supports the smooth transition between postures of the model, the realization of animation would be the next goal for the project to be more attractive and user friendly.

6.2. Light Rendering

Dealing with light is another important technique in visual effects for video games. Lighting plays a vital role in delivering the atmosphere. This project only applies a limited processing of light in the water reflection part as discussed in Section 4.3, yet not being sufficient without a global illumination such as a baked lighting, where the lightings for static objects are pre-calculated and stored according to the light probes distributed in the scene.

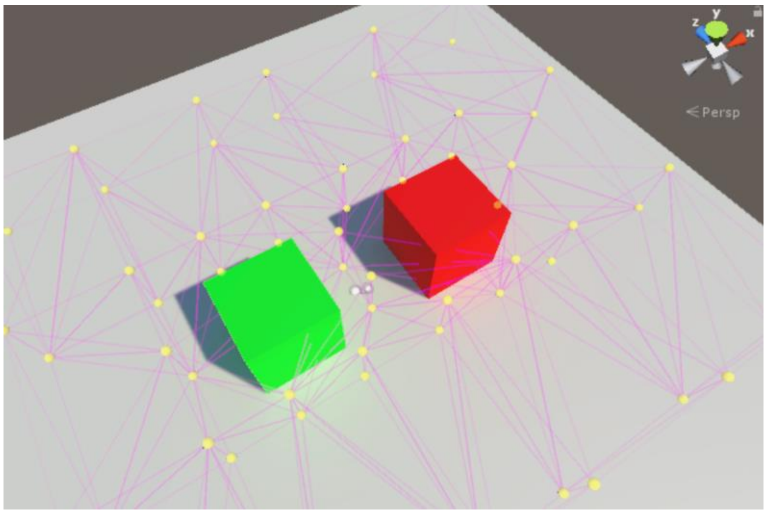


Figure 24 Example of Baked Lighting

7. Conclusion

This final year project serves as my first solely responsible attempt for a relatively large-scale project. Despite the limitations mentioned in section 6, the final output of this project basically satisfies the project objective. This section will quickly recapitulate the project objective and the respective degree of completion. Last, the schedule and milestone throughout the project period will be restated.

7.1. Objective Degree of Completion: Gameplay

This project identifies the objectives regarding gameplay and has accomplished features of character control with VR headset, interaction system and combat system. Despite the reluctance in model animation and fineness, all objectives have been proved to be feasible and delivered in the final product.

7.2. Objective Degree of Completion: Visual Effect

On the other hand, this project has achieved the objectives regarding visual effects by implementing the spherical bending effect and a simulation of water. The completion of this part is also satisfactory.

7.3. Recap: Project Schedule and Milestone

The development period of this project has been a mixed journey of joy, frustration, tilt and sense of accomplishment. When looking back to the original schedule of development when the project emerged, it’s encouraging to find that most objectives have been accomplished in time. This project will then wrap up with an actual timeline and milestone throughout the development process of the project.

Date	Content
8 Oct 2023	First Deliverable – Project Plan
Oct 2023	R&D session: refining the game framework, conducting feasibility research on proposed game systems (i.e., interaction, combat), completing local environment
Nov – Dec 2023	Implementation: Realization of the spherical world effect with a player control system for ordinary 3d game
1-7 Jan 2024	Preparation for the first presentation
Late Jan 2024	First presentation and second deliverable: Interim Report
Feb – Mar 2024	Implementation: Compatibility transplant to Oculus Quest 2, combat system and water simulation effect
Late Apr	Final presentation, final deliverable: final report, project exhibition

Table 4 Actual Schedule and Milestone

8. References

[1] *Creating a Vertex Displacement Shader - Unity Learn*. (n.d.). Unity Learn. https://learn.unity.com/tutorial/creating-a-vertex-displacement-shader

[2] Enright, D., Marschner, S., & Fedkiw, R. (2002, July). Animation and rendering of complex water surfaces. *ACM Transactions on Graphics*, *21*(3), 736–744. https://doi.org/10.1145/566654.566645

[3] Hagen, T., Hjelmervik, J., Lie, K. A., Natvig, J., & Ofstad Henriksen, M. (2005, November). Visual simulation of shallow-water waves. *Simulation Modelling Practice and Theory*, *13*(8), 716–726. https://doi.org/10.1016/j.simpat.2005.08.006

[4] Hasu, J. (2018). LUT University. *Fundamentals of shaders with modern game engines*.

[5] Hohstadt, T. (2013, August 17). The Age of Virtual Reality. Lulu.com. http://books.google.ie/books?id=iA5VAwAAQBAJ&printsec=frontcover&dq=in+the+age+of+virtual+reality&hl=&cd=2&source=gbs_api

[6] Hocking, J. (2022, March 1). Unity in Action, Third Edition. Simon and Schuster. http://books.google.ie/books?id=jXBYEAAAQBAJ&printsec=frontcover&dq=unity+in+action&hl=&cd=2&source=gbs_api

[7] Isidoro J et. al. (2002). Direct3d ShaderX, Vertex and Pixel Shader Tips and Tricks. *Rendering Ocean Water*, 347-357.

[8] Nintendo of America. (2019, June 11). Animal Crossing: New Horizons - Nintendo Switch Trailer - Nintendo E3 2019 [Video]. YouTube. https://www.youtube.com/watch?v=_3YNL0OWio0

[9] Pallavicini, F., Pepe, A., & Minissi, M. E. (2019). Gaming in Virtual Reality: What Changes in Terms of Usability, Emotional Response and Sense of Presence Compared to Non-Immersive Video Games? Simulation & Gaming, 50(2), 136-159. https://doi.org/10.1177/1046878119831420

[10] Ralf Habel and Michael Wimmer. 2010. Efficient irradiance normal mapping. In Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games (I3D '10). Association for Computing Machinery, New York, NY, USA, 189–195. https://doi.org/10.1145/1730804.1730835

[11] Technologies, U. (n.d.). Unity - Manual: Baked lighting. https://docs.unity3d.com/2019.1/Documentation/Manual/LightMode-Baked.html#:~:text=Baked%20Lights%20are%20Light%20components,any%20run%2Dtime%20lighting%20calculations.

[12] Technologies, U. (n.d.-a). Unity - Manual: Input system. https://docs.unity3d.com/Manual/com.unity.inputsystem.html

[13] Technologies, U. (n.d.). *Unity - Manual: Shaders*. https://docs.unity3d.com/Manual/Shaders.html

[14] Technologies, U. (n.d.). *Unity - Manual: Using Shader Graph*. https://docs.unity3d.com/Manual/shader-graph.html

[15] Yong He, Tim Foley, Teguh Hofstee, Haomin Long, and Kayvon Fatahalian. 2017. Shader components: modular and high performance shader development. ACM Trans. Graph. 36, 4, Article 100 (August 2017), 11 pages. https://doi.org/10.1145/3072959.3073648

[16] You, M., Park, J., Choi, B., Noh, J. (2009). Cartoon Animation Style Rendering of Water. In: Bebis, G., et al. Advances in Visual Computing. ISVC 2009. Lecture Notes in Computer Science, vol 5875. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-10331-5_7