COMP4801 FINAL YEAR PROJECT

Final Report

Topic: A multi-platform multiplayer game

Chan Ting Lok, Leon (3035574763)

Supervisor: Dr. Chim, T.W.

Submitted on 18/4/2022

Abstract

With the advancement of Virtual Reality (VR) technology, it is forecasted that the VR market size in 2024 will triple that of 2019. This project aims at developing a game that supports VR, mobile, and computer concurrently with a single codebase. With broader hardware coverage, not only more players can enjoy the game, but also the game revenue will increase drastically.

The final deliverable of this project is Dual X - a cross-platform multiplayer cooperative game that supports VR Devices (Oculus Quest 2), computers (Windows machines), and mobile phones (Android). Unity is chosen as the game engine for the project because it can deploy the game on more than 20 platforms and Photon is the chosen multiplayer engine because of its compatibility with Unity. Currently, two puzzle levels and one boss fight level have been implemented, supporting PC (both controller and keyboard and mouse), VR device (Oculus Quest 2), and mobile (Android). With limited budget and time, it is impossible to support all kinds of gaming consoles/hardware and produce a standard-length, polished game. A demo-size game should be expected instead.

Acknowledgement

Firstly, I would like to express my deepest thanks to the Project Supervisor, Dr. T.W. Chim, Lecturer, Department for Computer Science, The University of Hong Kong, for giving me opinion and comments while I am working on this final year project topic. When I have any inquiries, he always replies to my email swiftly and provides me with professional support. Secondly, I am grateful to Dr. Locky Law, Assistant Lecturer, Centre of Applied English Studies, The University of Hong Kong, for helping me in the aspect of technical English. He is always willing to lend a helping hand to me when I need help with the preparation of project documents/presentations.

Table of Content

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Meaning
AR	Augmented Reality
VR	Virtual Reality
MR	Mixed Reality
PC	Personal Computer
USD	United States Dollar
FYP	Final Year Project
GPU	Graphics Processing Unit
PUN	Photon Unity Networking
CCU	Concurrent Users
3D	Three Dimensional
RPC	Remote Procedure Calls
UI	User Interface

1. Introduction

1.1. Overview

The gaming industry is advancing. The market will eventually be saturated with the traditional way of gaming. The game developers are, therefore, trying to integrate and develop new technology in their games; for instance, motion tracking, augmented reality (AR), virtual reality (VR), mixed reality (MR), etc. Consequently, gaming is no longer constrained to gaming consoles, there are more and more devices for gaming, for example, personal computer (PC), mobile, and VR headset.

There are a few games that offer cross-platform multiplayer. However, we cannot find a game that is compatible with PC, mobile, and VR headsets concurrently. For a game that can support more platforms, not only more players can enjoy it, but the revenue for the game will also be drastically increased (refer to section 2). The goal of this project is to investigate whether it is possible to develop a game that supports multiplatform to provide players with a similar gaming experience. Therefore, a multiplatform, multiplayer game is the targeted deliverable of this project.

1.2. Motives

This section will cover the benefits for different parties concerned by this project. In section 1.2.1, the benefit for game developers will be explained. In section 1.2.2, the benefit for players will be discussed.

1.2.1. For Game Developer

From the game developer's perspective, the market size is positively related to the revenue. The broader the hardware coverage, the more profitable the game will be. But at the same time, covering more platforms with multiple separate codebases will drastically increase the developmental time and cost. Apart from that, if there are any additional updates for the game, the developers have to implement them in multiple codebases, it will be extremely hard to maintain. Supporting a cross-platform game with a single codebase will make the game more maintainable and economical.

1.2.2. For Player

Different players have different gaming preferences, for example, some prefer the convenience and choose the mobile platform, while others prefer immersive gaming experience and choose virtual reality. Also, the success of Nintendo Switch indicates that there is demand in that market for games that can be played under different scenarios. A cross-platform game can provide players with a similar gaming experience with different types of hardware. For example, when they are at home, they can play with their desktop computer/VR device and when they are outdoor, they can play using their mobile phone.

In terms of game design, the game is designed in a way that one single player cannot do all the work by himself, the player must learn to rely on another partner, and cooperation is the only way to ace the game. On the cooperative aspect, players will learn how to cooperate with their partners and communicate efficiently to exchange information; The puzzle-solving levels can improve players' creativity, logical thinking (for example learning by observing patterns), and problem-solving skills. The boss fight level can require players to be observant, they have to observe the boss's attack pattern and act corresponding to avoid damage.

1.3. Outline

This project plan is organised in the following way. In section 2, the background of the project will be introduced. Followed by section 0, the objective and scope will be brought out. After confirming the scope, in section 4, detailed methodology and relevant development tools will be laid down. In section 5, the implementation result will be discussed. Then, difficulty and limitation will be shared in section 6. After that, the future development will be covered in section 7. And finally, a summary of the main idea will be given in section 8.

2. Background

An overall picture of the traditional gaming market and VR gaming market will be shown in this section. The revenue data of the traditional gaming market and virtual reality gaming market will be analysed in section 2.1 and section 2.2 respectively. In section 2.3, a board picture of the cross-platform games in the current market will be given.

2.1. Traditional Gaming Market

Mobile, Console (e.g., PlayStations, Nintendo Switch, Xbox, etc), and PC are still the major mediums people use to play video games. It is observed that mobile platform is dominating the gaming market (generating more than 90 billion USD in 2021), followed by console and PC (generating 49.2 billion USD and 35.9 billion USD respectively in 2019) (see Figure 1) [2]. It is obvious that a game will generate much more revenue if it supports mobile, console, and PC concurrently. It gives grounds for why the game should be supporting both keyboard and mouse, game controller, and touchable monitor as input.

2.2. VR Gaming Market

Virtual Reality (VR) is defined as “the use of computer modeling and simulation that enables a person to interact with an artificial three-dimensional (3-D) visual or other sensory environments” [1]. It is a new trend in gaming. VR gaming is becoming more and more popular because of its intuitive control and immersive experience. It is forecasted that the VR market size (2.4 billion USD) in 2024 will triple that (0.8 billion USD) in 2019 (see Figure 2) [3]. The VR gaming market is expanding rapidly. In the not-too-distant future, VR will become a popular platform for gaming. This growing trend justified why it is important to support VR devices in this project.

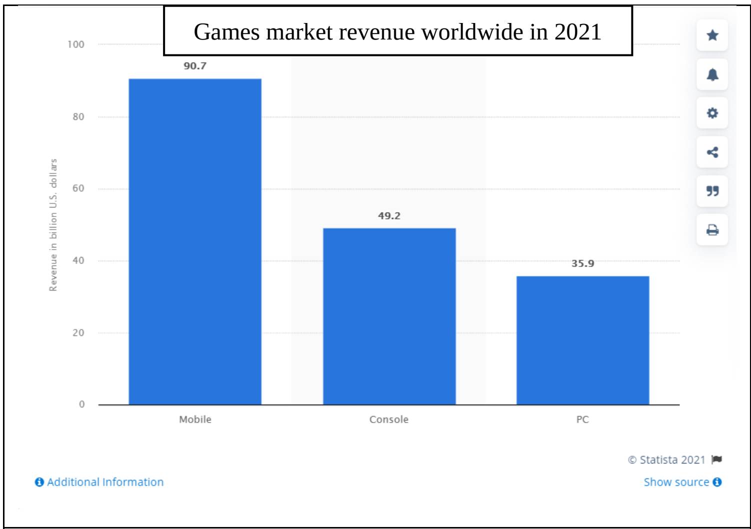


Figure 1: Games market revenue worldwide in 2021 [2]. It is concluded that mobile, console, and PC are still the major platforms for gamers to play games by comparing the revenue with Figure 2.

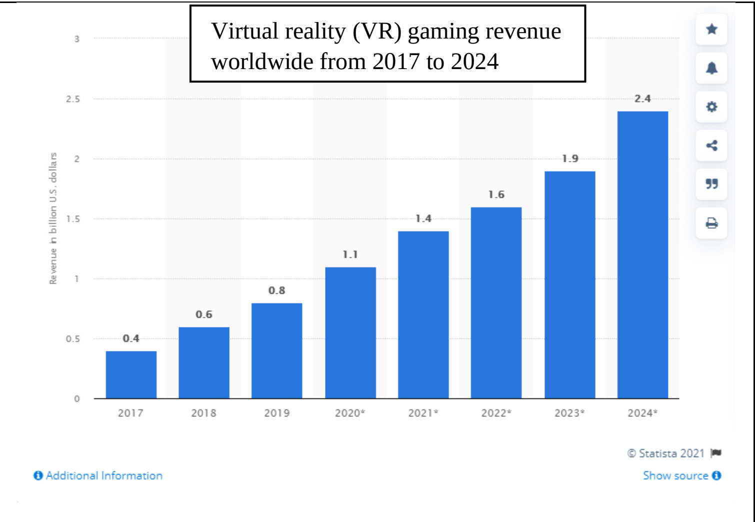


Figure 2: Virtual reality (VR) gaming revenue worldwide from 2017 to 2024 [3]. It is observed that the VR gaming revenue is growing drastically over the years.

2.3. Cross-platform games

There are more and more games that support cross-platform in recent years. However, we can observe that most of the cross-platform games support gaming consoles and PC only [4]. It is not hard to understand why the game developers just support gaming consoles and PC but do not support VR devices and mobile phones. (Marketing and economic reasons are not discussed here.) Both gaming consoles and PC support game controllers, making it easier to integrate. To support keyboards and mouses on top of gaming controllers may not be hard, because these two types of input mostly consist of physical buttons. For VR devices, the input is drastically different. Apart from physical buttons, VR controls heavily rely on VR controllers' (i.e. player’s hand) and the headset’s (i.e. player’s head) position and rotation. For mobile devices, the input becomes a touchable monitor. It is quite different from physical buttons. On top of that, the graphics processing unit (GPU) of mobile devices is relatively weaker than gaming consoles and PC, a game can hardly run smoothly on mobile devices without drastic graphics down-grading and optimisation.

3. Objective and Scope

In this section, the objective will be explained in section 3.1; Followed by a big picture of what will eventually be delivered in section 3.2.

3.1. Objective

This project aims at investigating the possibility to support multiple traditional gaming platforms and VR gaming with a single codebase. Using a single codebase instead of three sets of codebases to support three platforms can consequentially reduce the developmental cost and time.

3.2. Scope and Deliverable

A cross-platform, multiplayer game will be delivered at the end of the project. The game is called ***Dual X***.
Dual stands for 2 players because it is a compulsory 2-player cooperative game. It contains three levels - two puzzle-solving levels, and one boss fight level. To solve the puzzle, players need to figure out how to solve the puzzle and perform certain actions at the right timing, communication is thus very important. In the boss fight level, players will be given different abilities. To beat the boss, both abilities are needed, cooperation thus plays a vital role in this level. To facilitate communication and cooperation, a built-in voice chat system will be available in the game.

The X is a cross, which stands for cross-platform. Players can choose to play the game on various platforms. PC (Windows), Mobile (Android), and VR (Oculus Quest 2) are supported. In order to show that it can also support gaming consoles without further complexing the hardware requirement, for pc platform, both the game controller and keyboard and mouse will be supported.

4. Methodology

This section will be divided into three parts. The hardware setup and targeted platform will be detailed in section 4.1. The software aspect will be covered in section 4.2, justification will be provided for the development tools chosen. The programming design philosophy will be followed in section 4.3.

4.1. Hardware and targeted platform

Different VR headsets offer different functions, for example, some of them support hand tracking while others do not. Some of them may even run different operating systems. Thus, a single set of code will not be compatible with all VR headsets. Oculus device is chosen for this project because it is the most popular VR headset player chosen. As of January 2021, around half of the gamers on Steam (A popular video game digital distributing platform) choose Oculus devices as their VR device [5]. I choose Oculus Quest 2 (a model of Oculus device) as a result of budget concerns. For the mobile platform, I decided to support Android because of its popularity and operating system openness. In 2021 more than 70% of mobile devices are Android devices [6]. As mentioned in the part of the motives (section 1.2.1), market size is one of my concerns, Android is still the most dominating the mobile market. In addition, building the game on an iOS device requires a Mac Machine, which will further complex the hardware requirement. Not supporting iOS devices will not make a massive negative impact on the project result, because the input of Android and iOS devices are similar (touchable monitor). For the PC platform, windows will be supported. macOS is seldom a choice for gaming, it is, therefore, removed from the scope of this project.

4.2. Software

The following part will be covering two essential development tools for a multiplayer game – game engine (section 4.2.1) and multiplayer engine (section 4.2.2). To explain analogically, if the final game is a dish, the game engine is the basic set of cooking utilities to cook the dish. Some dishes (games) may have some special features like baked (multiplayer). We need an oven to cook it (multiplayer engine).

4.2.1. Game Engine

Considering the project aims at supporting multi-platforms, ***Unity*** will be the best choice. With ***Unity***, a single code base can be deployed on more than 20 platforms [7]. Apart from that, ***Unity*** offers excellent integration with VR technology. In addition, the developer community is large, and its code is comprehensively documented; In case of facing any technical difficulties, there are more resources available on the Internet. Windows is selected as the development platform for consistency and better optimization (see section 4.1 for justification). For programming language, C# is the only language that is supported by ***Unity***. It is an object-oriented scripting language.

4.2.2. Multiplayer and built-in voice chat system

To implement networking, there are two options: using the native unity networking technology or third-party technology. There are a few first-party solutions. UNet is a deprecated official solution [8]. It is not a wise choice to use a networking solution that will not work soon. There are two types of new networking solutions: GameObjects networking and ECS-based networking [8]. However, these two official solutions are still in an experimental state. In another word, there is currently no favorable first-party option.

Among tons of third-party solutions, ***Photon Engine***, which includes ***Photon Unity Networking 2*** (for multiplayer) and ***Photon Voice 2*** (for the built-in voice chat system), is chosen. Its excellent integration with ***Unity*** can smoothen the development and prevent optimization problems [9]. Apart from that, Photon is free of charge for 20 concurrent users (CCU), which is good enough for testing a small-scale multiplayer game like this [9]. In addition, Photon can host the server for the game, extra time, and cost on the maintenance of the server computer can be saved. To explain it with an analogy, to offer services (multiplayer services) for some customers (players' computers), you will need a customer service officer (host computer). Normally, you will need to hire one by yourself; With ***Photon***, the customer service officer will be hired and managed by ***Photon***.

4.3. Programming Design

The main difficulty of the implementation comes from supporting different types of hardware. For one action, different types of input need to be handled and interpreted. Take picking up an object as an example, for VR players, they may directly use their hand (controller) to grab the object; for PC players, they may use their keyboard and press a certain key to act; for mobile players, maybe they will use their finger to tap a virtual button on the touchable screen. The main programming design philosophy is to reuse code as much as possible. I will try to achieve it by creating some generic functions, like in this example PickUpObject(Object) and this will be called by PC, VR, and mobile devices. I will avoid creating three sets of code for three kinds of input.

In addition, considering the three-level scope of the game, in this project, I heavily used inheritance and prefab variant. Prefab can be understood as the template of the game object and prefab variant can be understood as an inheritance in the game object level. For example, common movements will be put in a base player class, and for level-specific action (like shooting) in the boss fight will be put in a shooter player class, which is a child class of the base player class.

5. Discussion of Result

This section will detailly discuss different aspects of the final deliverable – Dual X. The discussion will include game design, implementation, and technical findings of different parts of the game.

5.1. General

5.1.1. Menu

In the menu scene (see Figure 3), the player can choose which level to go to. In addition, the button in the bottom right corner is for the player to decide whether they join the voice chat or not.

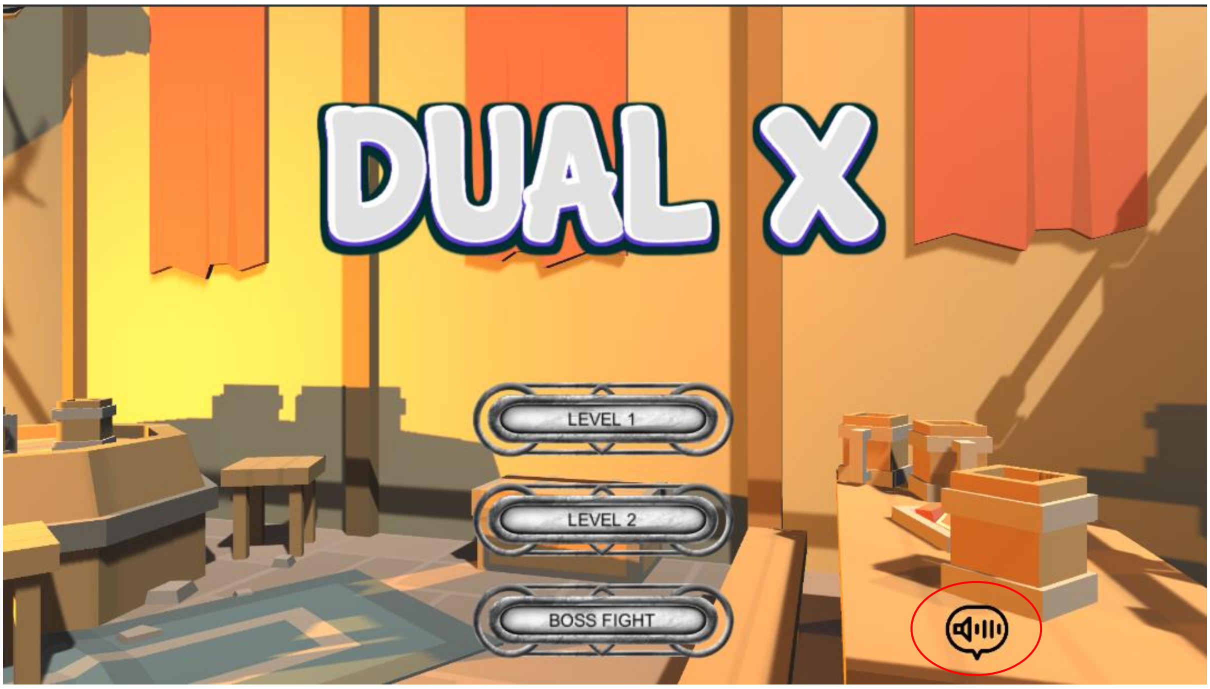


Figure 3: Screenshot of the menu of dual X. The button in the red circle is the voice chat button.

5.1.2. General Movement for different input.

For PC players with a keyboard and mouse, the player can use the mouse to look around, and WASD keys to move around, they will use the space bar as the jump button.

For PC players with controllers, the player can use the left joystick (number 3 in Figure 4) to move, the right joystick (number 1 in Figure 4) and use button A to jump.

For VR player, who use oculus quest controller (see Figure 5). They can use the left joystick to move and the button A to jump; if they want to look around, they can directly move their head.

For mobile players (see Figure 6), similar to PC players with controllers, they will use the left virtual joystick to move, and the right virtual joystick to look around. A virtual jump and interact button will be given.

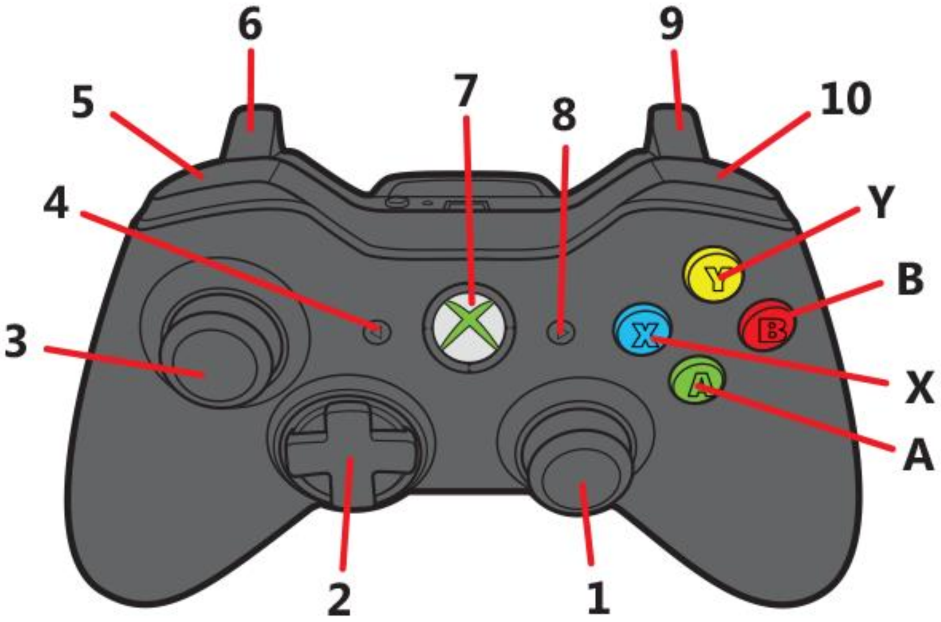


Figure 4: Xbox controller with label.

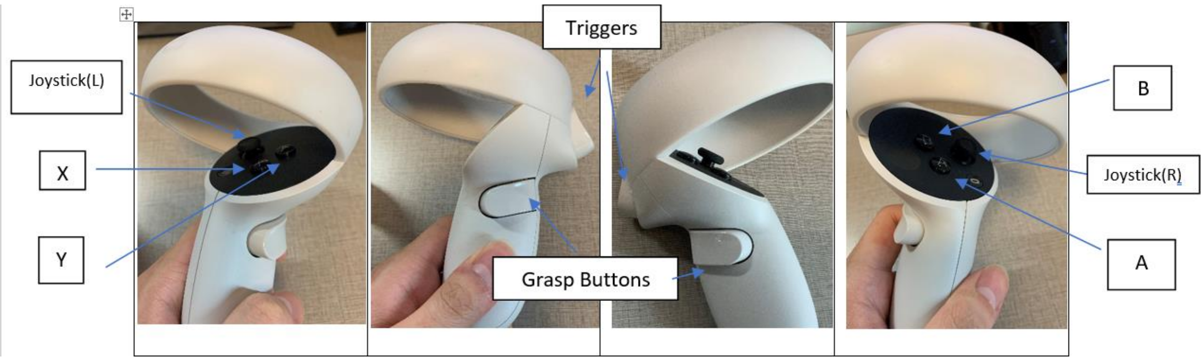


Figure 5: Oculus Quest 2 controller with label

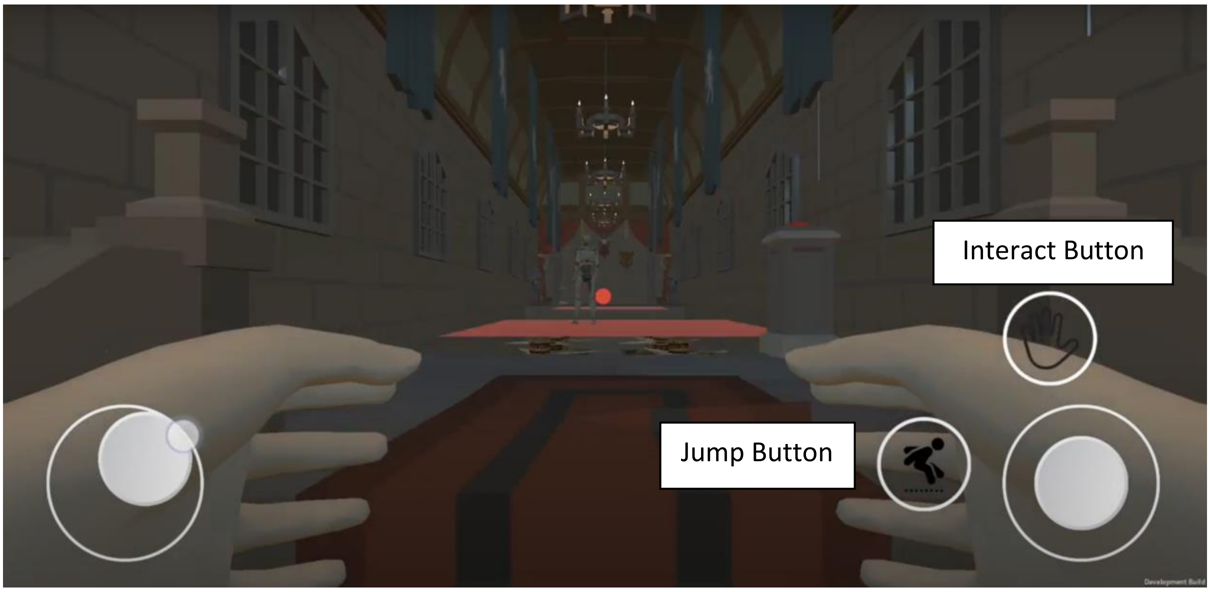


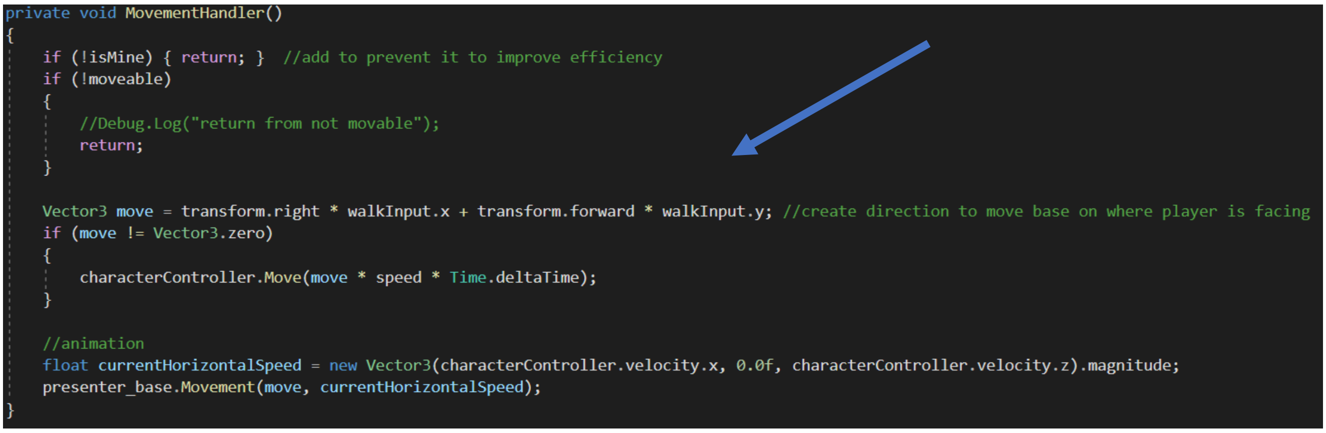
Figure 6: Screenshot showing mobile UI

To implement the control, there are mainly two ways to do so, first way is to use the character controller component, which provides many useful functions like Move() without dealing with a rigidbody. The second option is using a rigidbody component, which is a physics-based solution, that can provide more accurate physical simulation. A rule of thumb to determine which one to use is that if you need accurate physics simulation or customised moveset, go for rigidbody approach; else, go for character controller approach. That is because, as suggested by its name, character controller implemented general character control that should suit most cases.

The character controller approach is chosen for two reasons, firstly, accurate physics is not required for this project. Secondly, for VR control, using accurate physics may lead to self-colliding (e.g. the player's hand stuck in the body). Therefore, character controller is a more suitable approach in this use case.

To implement the player's movement, we need to take the player's input. The walkInput (see the blue arrow in Figure 7) and combine it with the player's current rotation (the direction the player is facing) to form a new move Vector. After that we will pass the move Vector to the character controller, multiplying it with the player’s movement speed (a float) and Time.DeltaTime (it is the time passed since the previous call of this function) to ensure a smooth motion.

The isMine bool is to check whether the player game object belongs to the player using the computer because the game scene will also contain a copy of the partner player’s game object. After all, the player in this computer will not be controlling the player game object of the partner player, an early return can enhance code efficiency. The moveable bool is to check whether the player is allowed to move at that moment. Under some situations, the player will not be allowed to move, for example, when the player received an electric shock (refer to section 5.5.1).



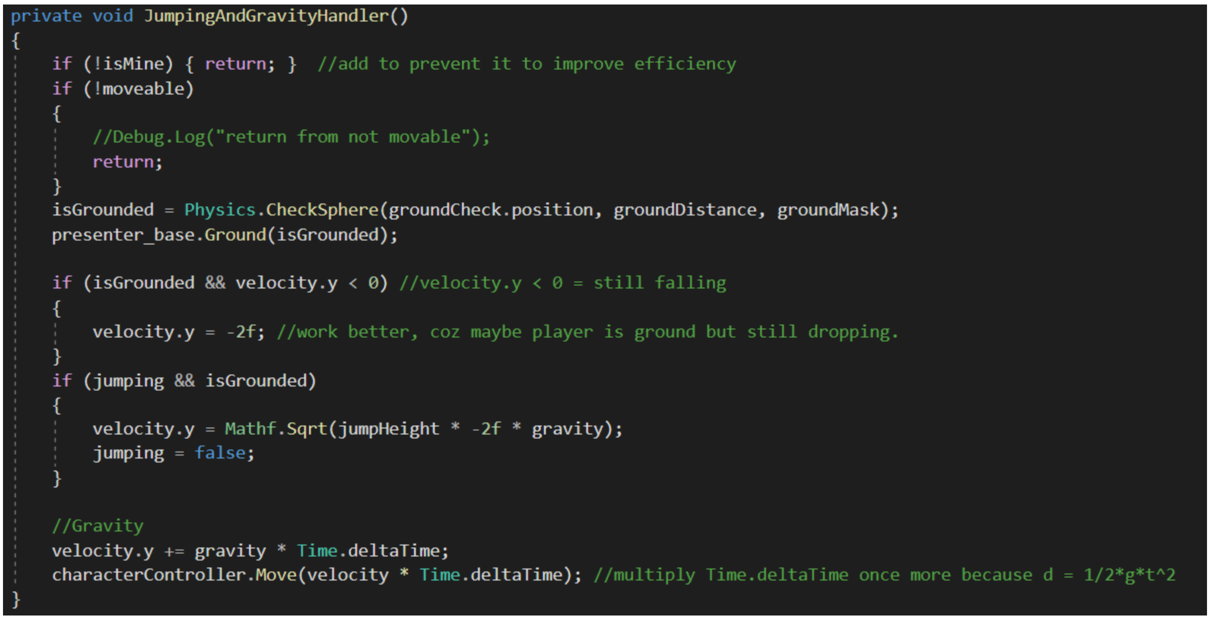
```
private void MovementHandler()
{
    if (!isMine) { return; }  //add to prevent it to improve efficiency
    if (!moveable)
    {
        //Debug.Log("return from not movable");
        return;
    }

    Vector3 move = transform.right * walkInput.x + transform.forward * walkInput.y; //create direction to move base on where player is facing
    if (move != Vector3.zero)
    {
        characterController.Move(move * speed * Time.deltaTime);
    }

    //animation
    float currentHorizontalSpeed = new Vector3(characterController.velocity.x, 0.0f, characterController.velocity.z).magnitude;
    presenter_base.Movement(move, currentHorizontalSpeed);
}
```

Figure 7: function to handle player movement

To implement jumping, ground checking is needed. Jumping will only be allowed when the player is landed on the ground. If players are allowed to jump in the mid-air, they can jump past all the obstacles and break the game. Physics.CheckSphere() is a function that returns a bool value. The function takes three parameters: The first parameter is centre of the sphere. I inputted the groundCheck.position, ground Check is an empty game object that I place in the bottom part of the player game object. The second parameter specifies the radius of the sphere. I inputted the groundDistance (the maximum distance between player and ground such that he will be considered grounded). The third parameter specifies the layer mask of the game object that wanted to be detected. Layer mask can be understood as a label on a game object. Here ground mask means the game object labelled as ground.

That’s how the function work (see Figure 9), it creates a sphere based on the centre (parameter 1) and radius (parameter 2) inputted to see whether any game object with a specific label (parameter 3) overlaps the sphere.

Similar to how movement is handled, after calculating the overall effect of jump force and gravity, the vector representing the overall effect is passed to character controller.Move() to handle the player's position (see Figure 8).



```
private void JumpingAndGravityHandler()
{
    if (!isMine) { return; }  //add to prevent it to improve efficiency
    if (!moveable)
    {
        //Debug.Log("return from not movable");
        return;
    }
    isGrounded = Physics.CheckSphere(groundCheck.position, groundDistance, groundMask);
    presenter_base.Ground(isGrounded);

    if (isGrounded && velocity.y < 0) //velocity.y < 0 = still falling
    {
        velocity.y = -2f; //work better, coz maybe player is ground but still dropping.
    }
    if (jumping && isGrounded)
    {
        velocity.y = Mathf.Sqrt(jumpHeight * -2f * gravity);
        jumping = false;
    }

    //Gravity
    velocity.y += gravity * Time.deltaTime;
    characterController.Move(velocity * Time.deltaTime); //multiply Time.deltaTime once more because d = 1/2*g*t^2
}
```

Figure 8: function that handles player's jump and gravity

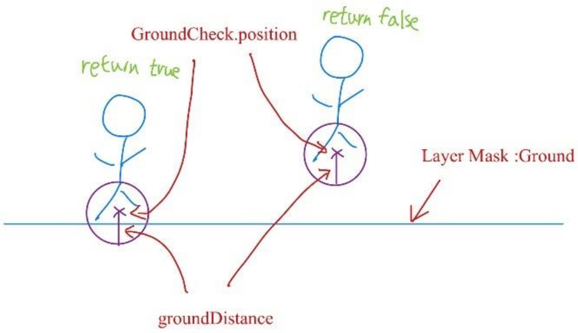


Figure 9: Graphical illustration for Physics.CheckSphere() function

To handle looking around, we will multiply the rotation input with the rotational sensitivity and Time.deltaTime. X and Y rotation Input are handled separately because the camera and player game object are needed to be handled separately (see Figure 10). For the camera, we need to rotate it according to both horizontal and vertical rotation, because the player may look up and down. For the player's body's rotation, when the player looks at the right side, I want the player to turn his whole body towards the right (That is the way for the player to turn around). However, when the player looks up, I don't want to rotate the body vertically. That's why transform.Rotate() [transform here mean player's body transform] only take X rotation into account (see Figure 11).

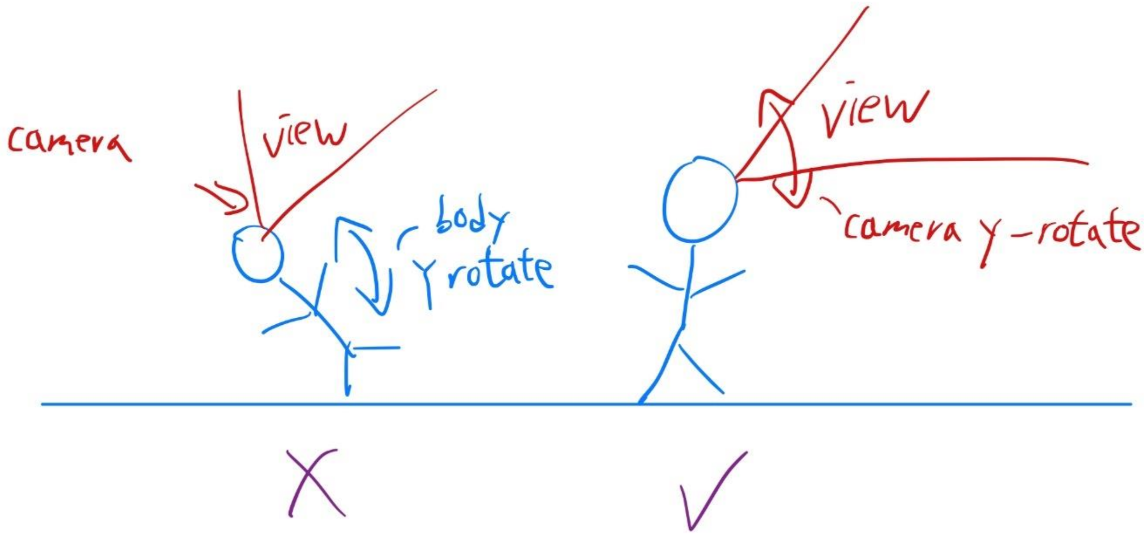
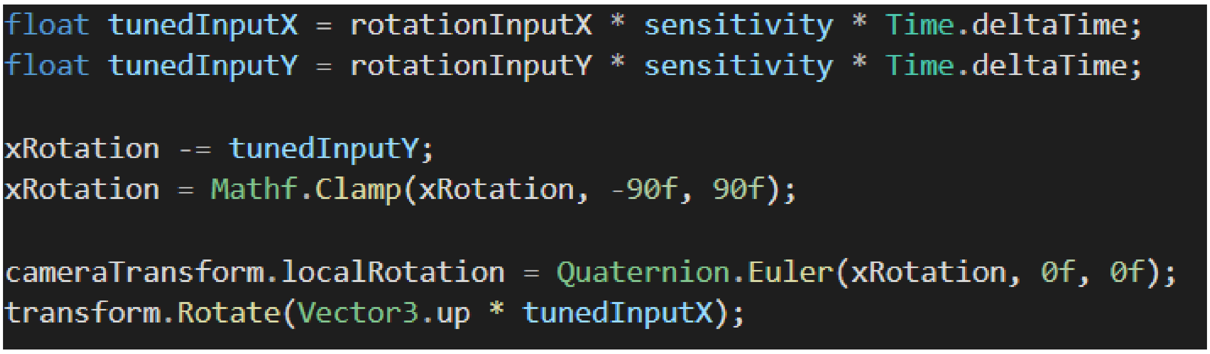


Figure 10: graphical illustration of the difference between correct and wrong y rotation. Left: shows the whole player's body is rotated; Right: shows only the player's view (camera) rotated



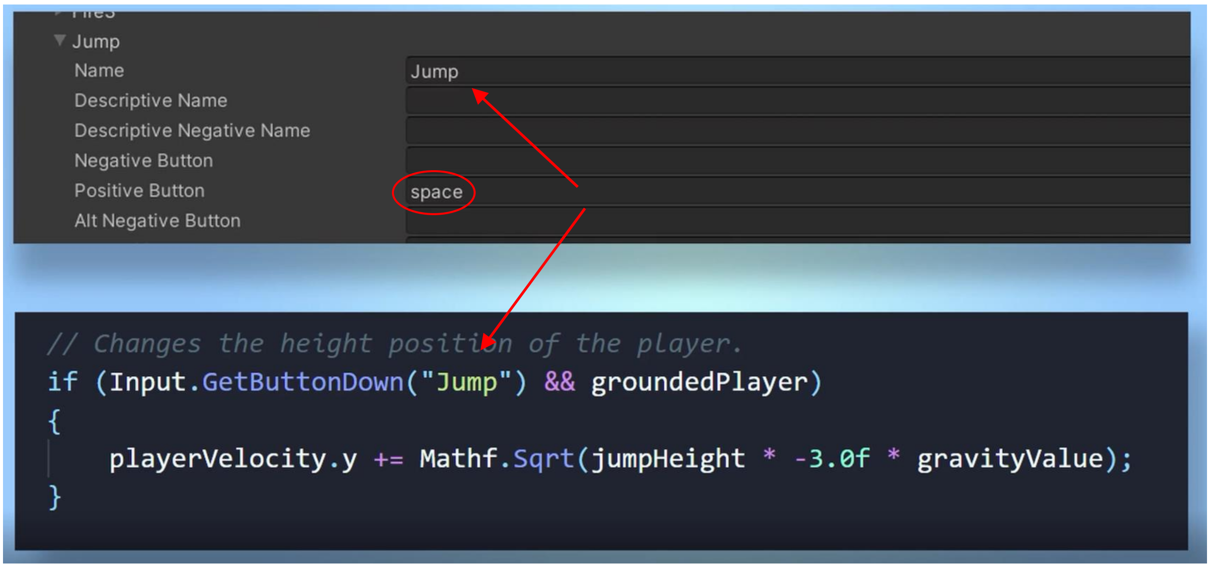
```
float tunedInputX = rotationInputX * sensitivity * Time.deltaTime;
float tunedInputY = rotationInputY * sensitivity * Time.deltaTime;

xRotation -= tunedInputY;
xRotation = Mathf.Clamp(xRotation, -90f, 90f);

cameraTransform.localRotation = Quaternion.Euler(xRotation, 0f, 0f);
transform.Rotate(Vector3.up * tunedInputX);
```

Figure 11: code segment for look handling

5.2. Unity New Input System - Supporting Different Types of Input devices

In recent years, Unity introduced a new input system. In this section, a brief comparison between the two systems will be given and justifications of why the new input system is employed will be talked about.

In the old input system (see Figure 12), developers have to check the player's input in the update function (it is a function that Unity will call in each frame) if a certain key is pressed, perform a certain action. There are some problems with this approach. Firstly, unity will check player input every frame which is inefficient and wastes computing power. [10] Secondly, players usually have many input actions, making the update function massive. [10] As a result, it will be hard to maintain and debug. Thirdly, the device compatibility is low. You have to manually add more conditions to check for additional devices input [10].

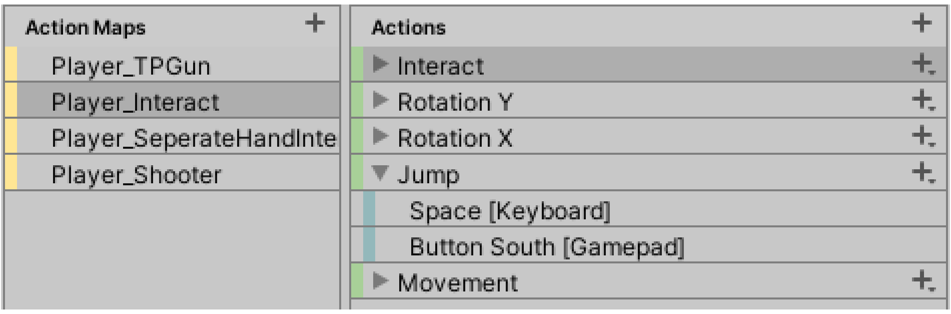


▼ Jump
Name: Jump
Descriptive Name:
Descriptive Negative Name:
Negative Button:
Positive Button: space
Alt Negative Button:

```
// Changes the height position of the player.
if (Input.GetButtonDown("Jump") && groundedPlayer)
{
    playerVelocity.y += Mathf.Sqrt(jumpHeight * -3.0f * gravityValue);
}
```

Figure 12: demonstration of unity old input system. [10]

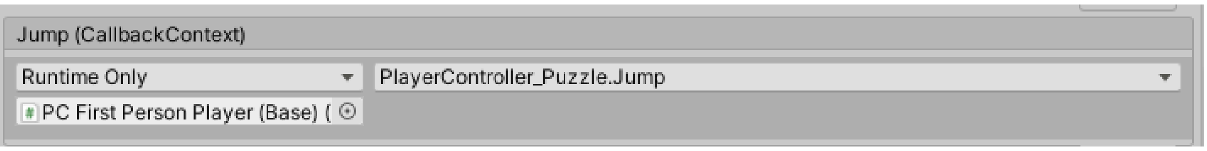
The new input system fixed most of the problems mentioned above. Unity introduced a type of asset called input actions (see Figure 13Figure 13). Action maps can be thought of as a container of actions. Different actions can be mapped to an infinite number of buttons/input keys. For example, the “Jump” action in Figure 13, can be triggered by the space bar on the keyboard or the button south on a gamepad.



Action Maps	Actions
Player_TPGun	▶ Interact
Player_Interact	▶ Rotation Y
Player_SeperateHandInte	▶ Rotation X
Player_Shooter	▼ Jump
	Space [Keyboard]
	Button South [Gamepad]
	▶ Movement

Figure 13: Input actions in Unity. Showing different actions inside Player_Interact action map.

And when jump action is triggered, “Jump” function in “PC First Person Player (Base)” script will be fired (see Figure 14). The new input system increases the code maintainability by separating device and input. Also, it promotes device compatibility because developers can map an action with an infinite number of input keys coming from different input devices.



Jump (CallbackContext)
Runtime Only ▾ | PlayerController_Puzzle.Jump ▾
PC First Person Player (Base) (

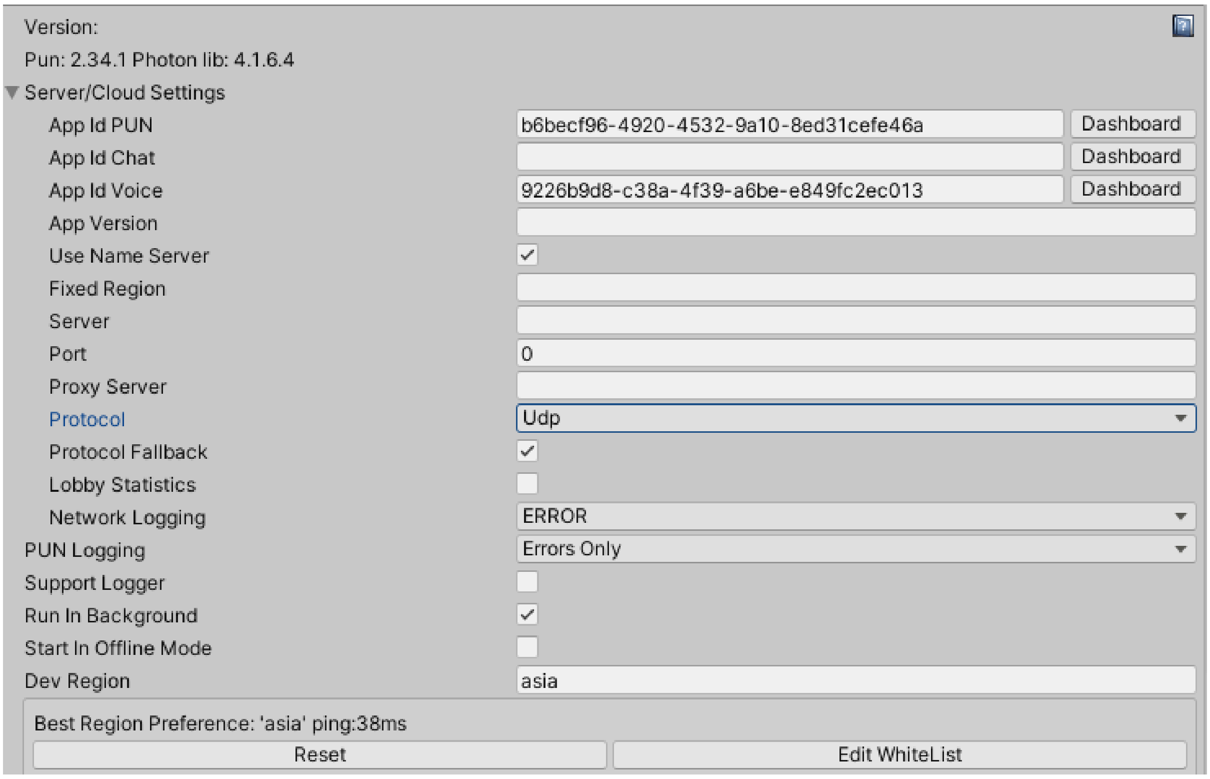
Figure 14: Player Input component in Unity, showing jump function is triggered when Jump action is performed.

5.3. Multiplayer Implementation – Photon

This section will cover two most important topics in Photon. How to establish connection and how to synchronise will be discussed in section 5.3.1 and section 5.3.2 respectively.

5.3.1. Establish Connection

To create a multiplayer environment, there are three steps we have to do. Before we implement anything, we have to set up the photon server setting. Some important fields (see Figure 15) will be introduced here. Unchecking Use Name Server means to host photon myself. I am checking it because I want my game to be hosted by photon's server instead. Fixed Region means whether the game can only be run by server in a specific region. If no, then photon will choose the best region for you. Here, I leave it empty.



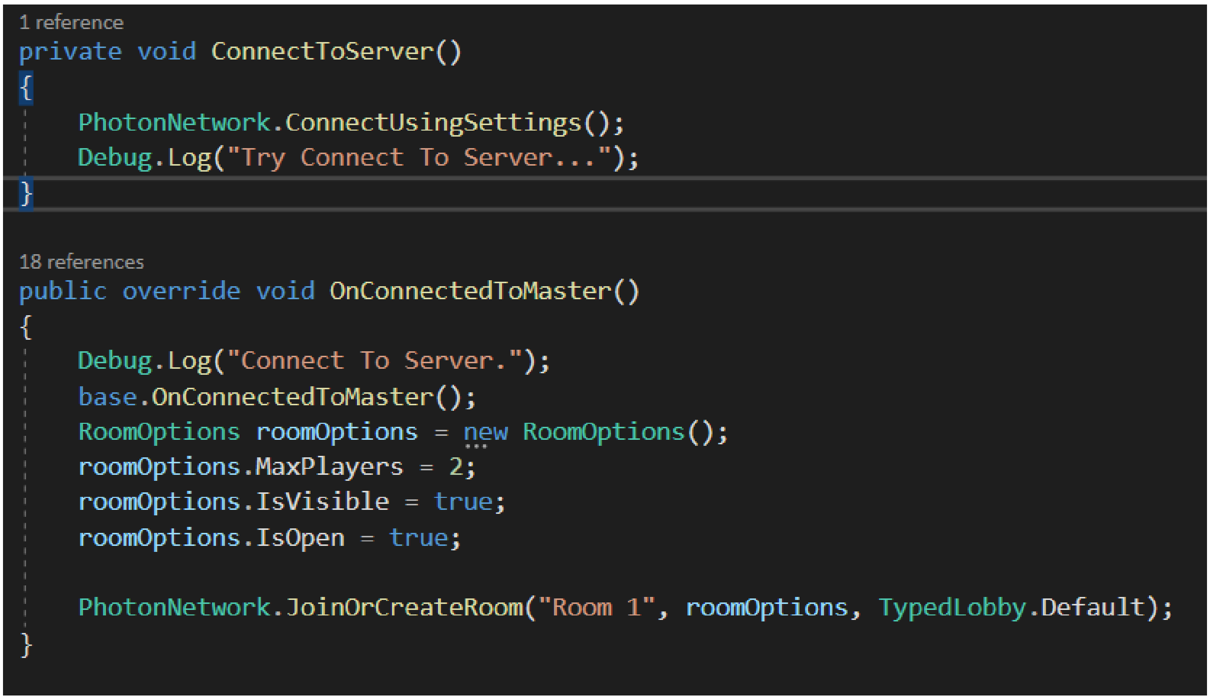
Version:
Pun: 2.34.1 Photon lib: 4.1.6.4
▼ Server/Cloud Settings

App Id PUN	b6becf96-4920-4532-9a10-8ed31cefe46a	Dashboard
App Id Chat		Dashboard
App Id Voice	9226b9d8-c38a-4f39-a6be-e849fc2ec013	Dashboard
App Version		
Use Name Server	☑	
Fixed Region		
Server		
Port	0	
Proxy Server		
Protocol	Udp ▾	
Protocol Fallback	☑	
Lobby Statistics	□	
Network Logging	ERROR ▾	
PUN Logging	Errors Only ▾	
Support Logger	□	
Run In Background	☑	
Start In Offline Mode	□	
Dev Region	asia	

Best Region Preference: 'asia' ping:38ms

Figure 15: Showing the settings available for a photon server

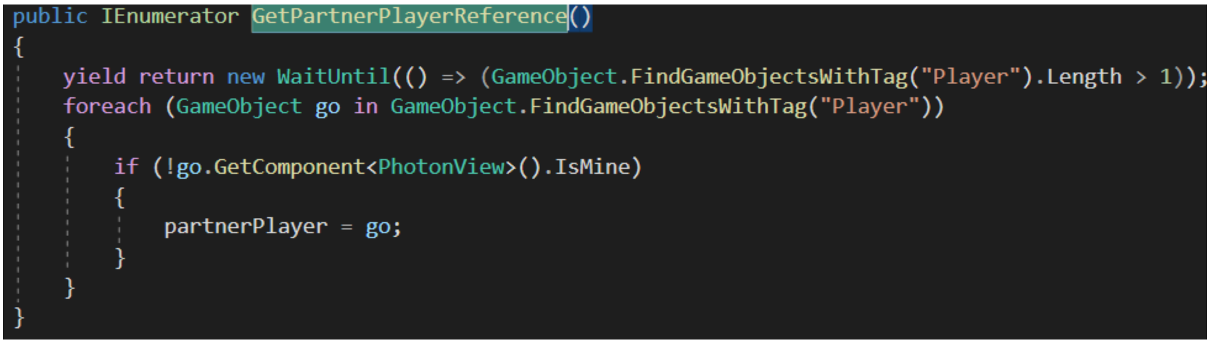
Firstly, we have to connect to Photon's server by PhotonNetwork.ConnectUsingSettings(). OnConnectedToMaster() will be called once the client has successfully connected to the master server (see Figure 16). After that, we have to join (or create if not exist) a room. Here I set the maximum player to 2 (because this is a 2-player game), and the room is open and visible to other players. The connection is done here (see Figure 16). And the client who created the room will act as the master client (can be understood as the room master). Once the master client is disconnected, the room ownership will be transferred to other clients. In other words, another client will become the master client instead.



```
1 reference
private void ConnectToServer()
{
    PhotonNetwork.ConnectUsingSettings();
    Debug.Log("Try Connect To Server...");
}

18 references
public override void OnConnectedToMaster()
{
    Debug.Log("Connect To Server.");
    base.OnConnectedToMaster();
    RoomOptions roomOptions = new RoomOptions();
    roomOptions.MaxPlayers = 2;
    roomOptions.IsVisible = true;
    roomOptions.IsOpen = true;

    PhotonNetwork.JoinOrCreateRoom("Room 1", roomOptions, TypedLobby.Default);
}
```

Figure 16: figure showing ConnectToServer() function (created by me) and OnConnectedToMaster() (a callback function)

What tasks left is to config the game correctly using other callback functions like OnJoinedRoom(), OnPlayerEnteredRoom(), etc. For example, to get a reference of the partner's player object in level manager, GetPartnerPlayerReference() (see Figure 17) is called in two places. For Master Client, when a new player enters the room, OnPlayerEnteredRoom() callback function will be triggered, so it is a good place for the master client to get a reference of the partner player. For normal client, OnJoinedRoom() is a good place to call GetPartnerPlayerReference(). This IEnumerator will wait until the number of game objects with the tag "Player" is more than 1, then it will loop throw all the player game objects. If the IsMine bool in PhotonView is false, that means it is the partner player.



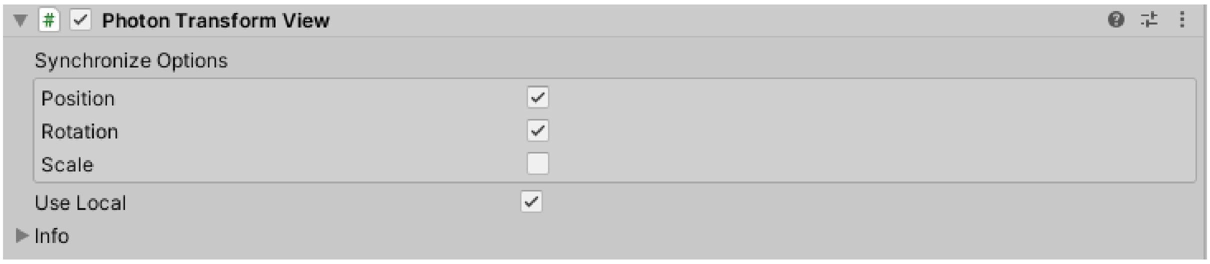
```
public IEnumerator GetPartnerPlayerReference()
{
    yield return new WaitUntil(() => (GameObject.FindGameObjectsWithTag("Player").Length > 1));
    foreach (GameObject go in GameObject.FindGameObjectsWithTag("Player"))
    {
        if (!go.GetComponent<PhotonView>().IsMine)
        {
            partnerPlayer = go;
        }
    }
}
```

Figure 17: showing GetPartnerPlayerReference(), a function that gets the reference of the partner player.

5.3.2. Synchronization

Photon provides two types of options for synchronization. In this section, both of them will be introduced, followed by how I implemented synchronization in this project.

The first option is photon view, it is a type of continuous synchronization. One of the examples is photon transform view (see Figure 18). With photon transform view put in the game object, photon will sync the checked parameters (position and rotation in Figure 18) continuously [11]. Photon transform view is just one example, there are also other photon views that sync other types of parameters. Developers can also define their own parameters to sync. The advantage of this option is that it is easy to implement and maintain (all troubles are left to photon to handle for us). However, because of the continuous data transfer between peer and master, it will create a great burden on the network if it is used for everything.



☑ Photon Transform View

Synchronize Options

Position ☑

Rotation ☑

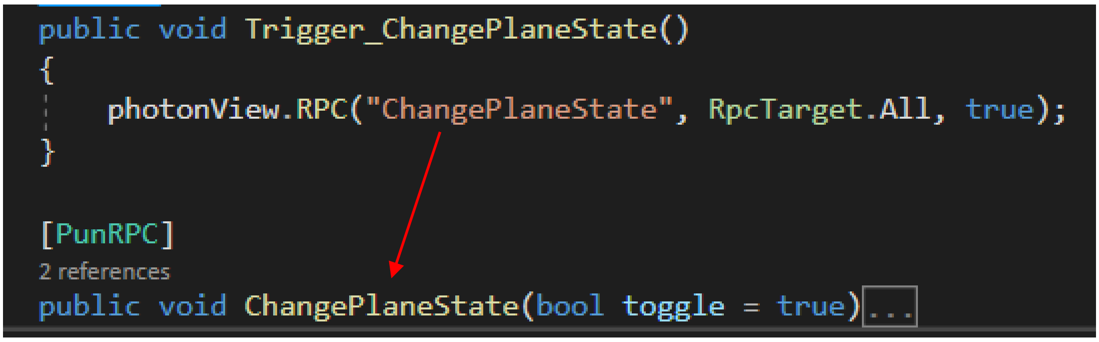
Scale □

Use Local ☑

▶ Info

Figure 18: Photon Transform View component in Unity.

The second option is Remote Procedure Call (RPC). RPC allows developers to fire functions in their peers [12]. Take how I synchronize the plane state in puzzle level 1 – toggle plane (refer to section 5.4.1) as an example (see Figure 19). "photonView.RPC()" takes three parameters. The first parameter (ChangePlaneState) is the name of the function to call in other peers; The second parameter is the target of the function call, RpcTarget.All means to call this function in all targets (including itself), there are also other options like RpcTarget.others (call the function in all targets except itself). The third parameter (true) is the parameter input for the function call [12]. In this case, I pass a true to all the targets in the ChangePlaneState() function call. This downside of this method is that relative to photon view it is harder to implement (because the logic is more complicated to manage). However, thanks to the on-demand usage of network resources, it is more network efficient.



```
public void Trigger_ChangePlaneState()
{
    photonView.RPC("ChangePlaneState", RpcTarget.All, true);
}

[PunRPC]
2 references
public void ChangePlaneState(bool toggle = true)...
```

Figure 19: code segment for synchronization of plane state in puzzle level 1 – toggle plane

To enhance network efficiency, my rationale is to use continuous synchronization as little as possible. I will use it only if I need to synchronize something continuously. I use photon transform view to synchronize players' positions. And for other things in the scene (like plane state and conductor gap state), I use RPC to synchronise.

5.4. Puzzle Level 1 - Toggle Plane

5.4.1. Game Design

The game design of the toggle plane level is fairly simple. It will act as a tutorial level for players. The floor plan is attached below (see Figure 20). The goal of this level is to get to platform B from platform A (starting point). In the floor plan, red and blue rectangles are used to represent the corresponding toggleable planes. On each platform, there is a toggle button that can toggle the plane's state. If the planes are in the red state, only red planes will be shown and vice versa (see Figure 21 and Figure 22). Please go to http://i7.cs.hku.hk/~tlchan/FYP21057/Source/index.html for video demonstrations.

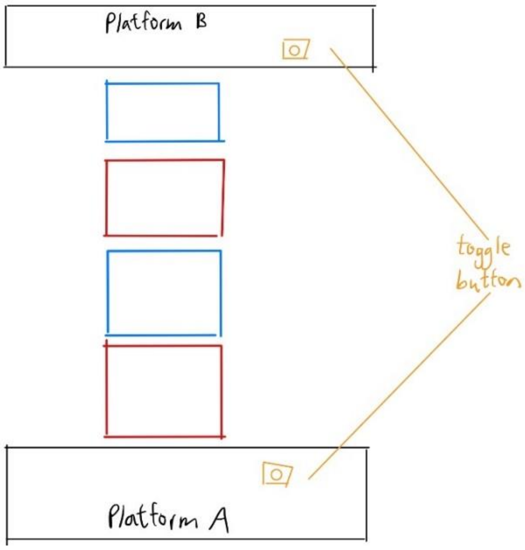
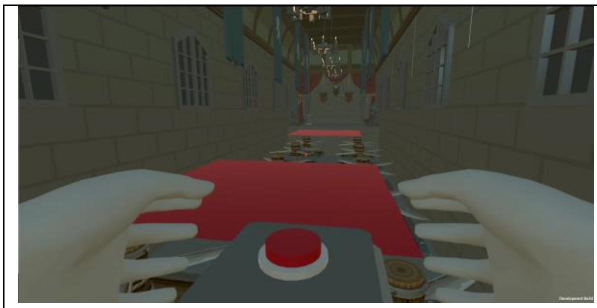
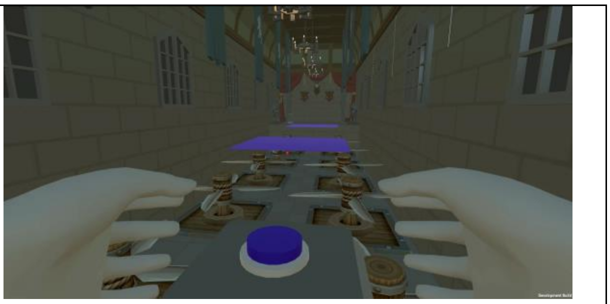


Figure 20: Floor plan of puzzle level 1 – toggle plane

 *Figure 21: Player screenshot showing the red state of the toggle planes.*	 *Figure 22: Player screenshot showing the blue state of the toggle planes.*

The solution for this level is that player A controls the toggle button, and toggles the planes at the right time (click the button when player B jumped and is in the mid-air between planes) to help player B to get to the platform B. And then player B uses the toggle button on platform B to do the same thing for player A.

5.4.2. Different controls and implementation

For PC with keyboard and mouse, they will have to use a sight (by moving their mouse) to aim at the object they want to interact with and click it (see Figure 23).

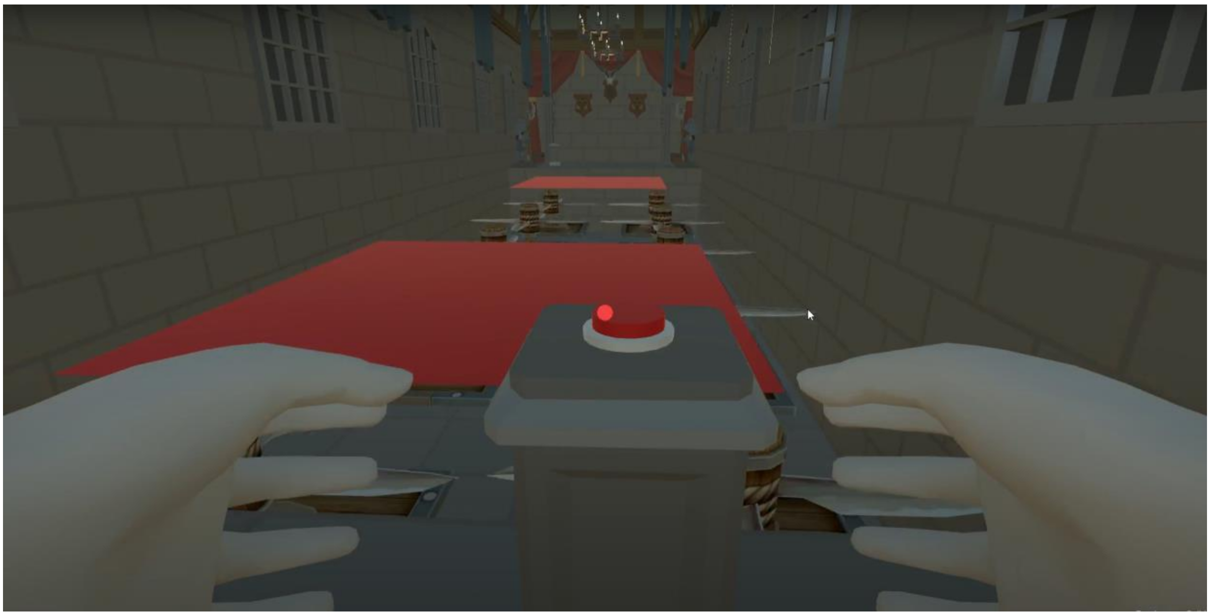
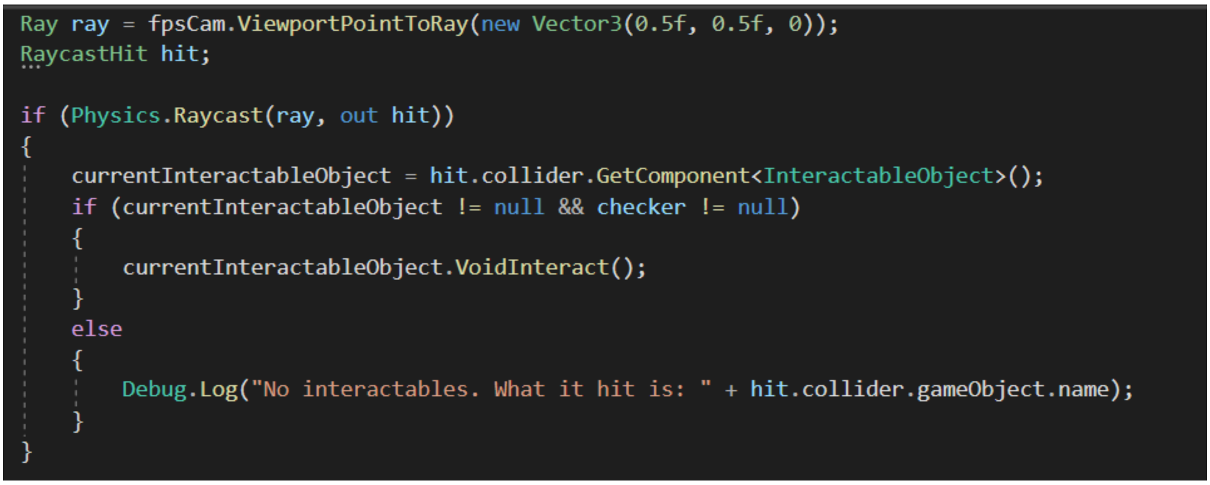


Figure 23: Screenshot of puzzle level 1 PC player with keyboard and mouse

The implementation of the sight aiming system made use of raycast. Raycast can be imagined as shooting an invisible beam from a certain point, and the beam (ray) can gather information about the object it hit without actually affecting the object physically. see Figure 24 for detail implementation.

fpsCam is the short form of the first-person camera of the player. ViewportPointToRay() function take a Vector 3 (a point) as parameter. This function will return a ray going from the first-person camera through the input point. Physics.Raycast() will take a ray, and then return what is hit by an out object in the function below, I give the out object with the variable name “hit”. If the ray does hit something with a collider, I will then check if the object it hit contains the component InteractableObject, which is a class I create for all interactable objects in the game. InteractableObject is the parent class of all interactable objects in two puzzle levels. Different child classes can override its VoidInteract() method to act according to their own specific behaviour.



```
Ray ray = fpsCam.ViewportPointToRay(new Vector3(0.5f, 0.5f, 0));
RaycastHit hit;

if (Physics.Raycast(ray, out hit))
{
    currentInteractableObject = hit.collider.GetComponent<InteractableObject>();
    if (currentInteractableObject != null && checker != null)
    {
        currentInteractableObject.VoidInteract();
    }
    else
    {
        Debug.Log("No interactables. What it hit is: " + hit.collider.gameObject.name);
    }
}
```

Figure 24: code segment for raycast

The PC player with a controller will interact by pressing the east button (see Figure 26), note that it works not only for the Xbox controller I show below, but it also works for all kinds of typical controllers (e.g. PS4/PS5 controller, Nintendo Switch Pro controller). Sight is no longer needed, so when the game recognised the controller is plugged into the computer during the game, the sight will be dismissed (see Figure 25).

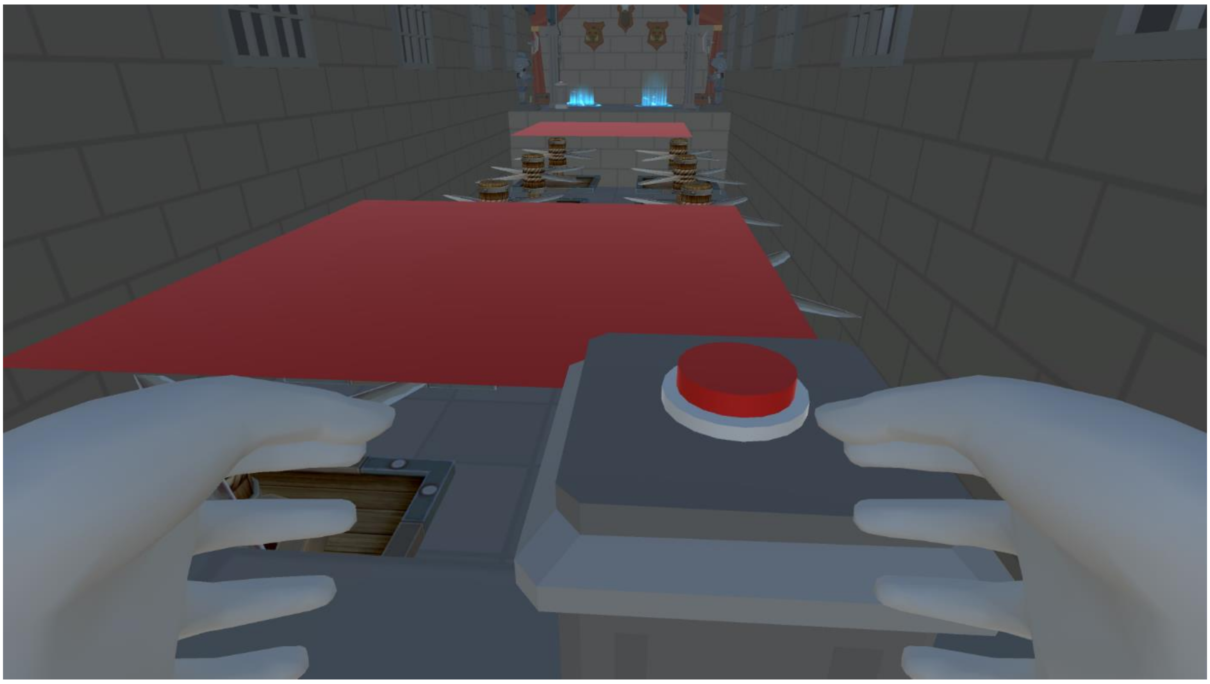


Figure 25: screenshot for pc player using controller

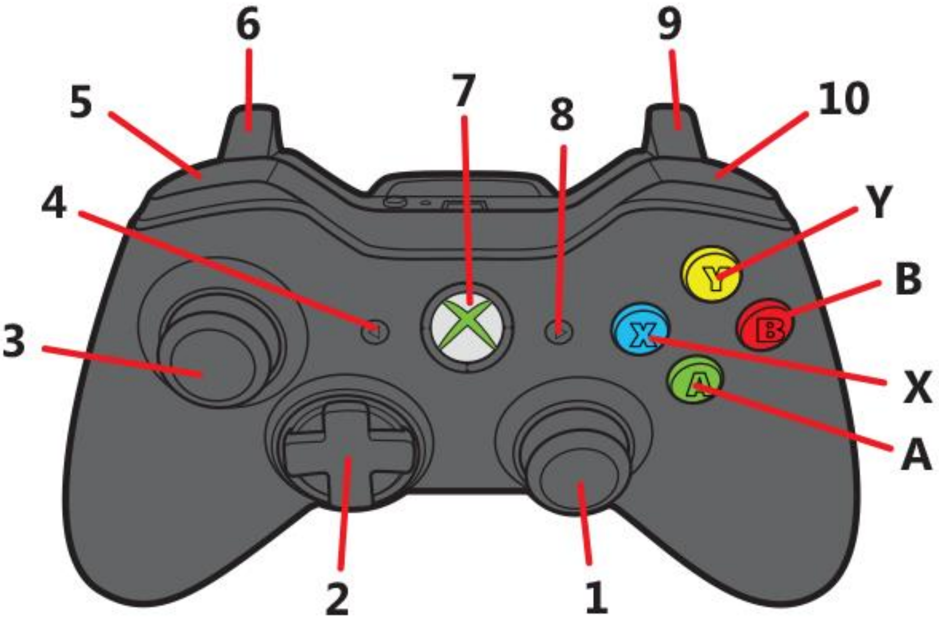


Figure 26: Xbox controller with label.

For the controller, I need to implement a way such that the player can interact with the game without aiming because aiming is too hard using the right joystick. For all interactable objects, it contains a big collider (see the big green box in Figure 27) for players to check whether their surroundings contain any interactable object. When the player collides with something, OnTriggerEnter() will be called (see Figure 28). It will then check whether the object it collides with contains a range checker component (a class I created for checking if the player is in range). And store a reference of the collided game object in currentTriggerCollisionGO (GO here stands for game object). And this reference will be set to null when the player leaves the range check zone. So, when the player presses the interact button (button east), it will check whether there is a reference to the collided game object. If there is, it will trigger the VoidInteract() function in the interactable object, the same as how we trigger that in raycast approach.

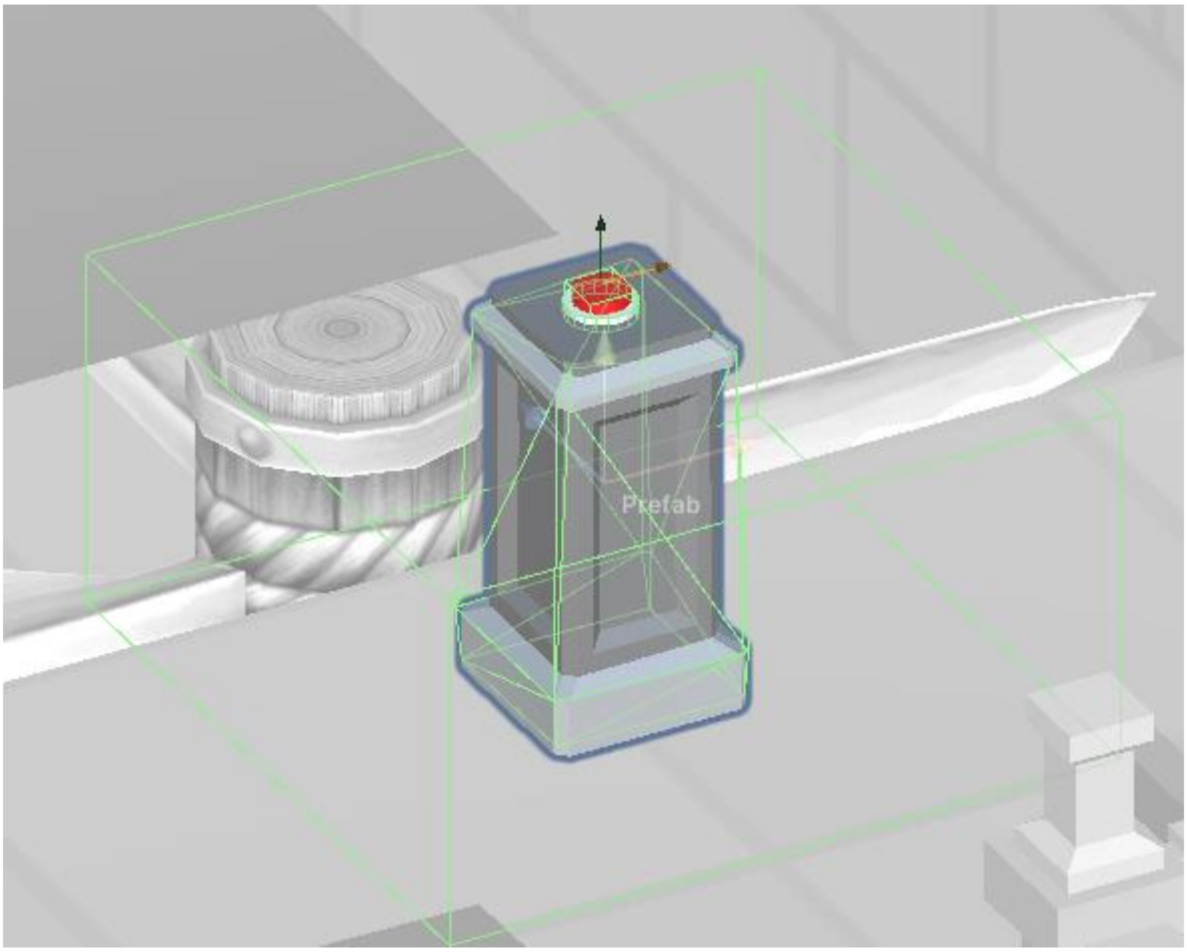
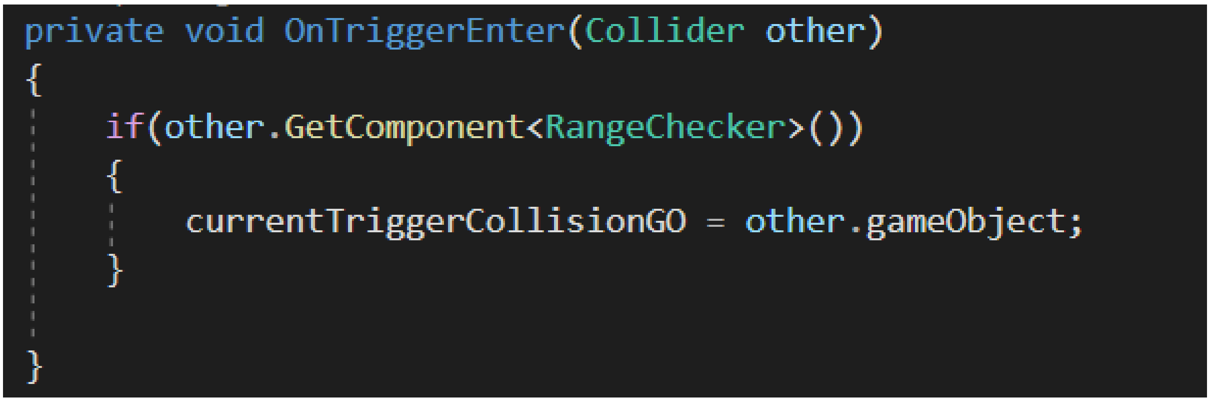


Figure 27: Button to toggle plane for puzzle level 1



```
private void OnTriggerEnter(Collider other)
{
    if(other.GetComponent<RangeChecker>())
    {
        currentTriggerCollisionGO = other.gameObject;
    }

}
```

Figure 28: code segment to inform user it is collide with trigger collider

For VR players (see Figure 29), to provide the most intuitive control and user experience, they can press it directly with their hands (VR controllers).

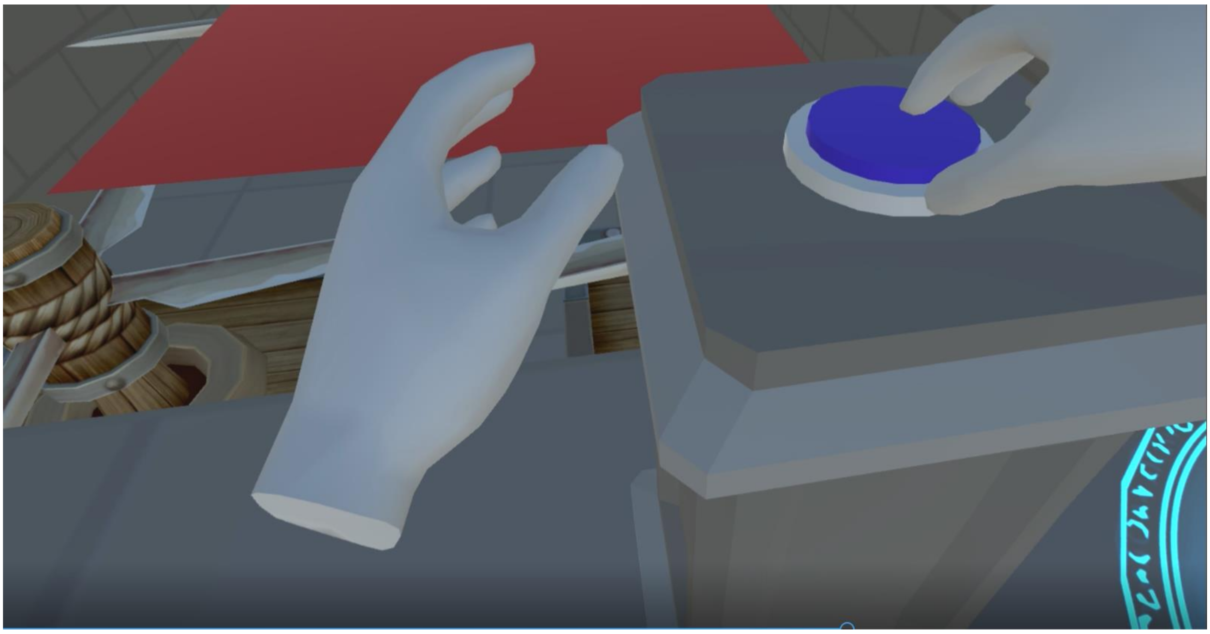
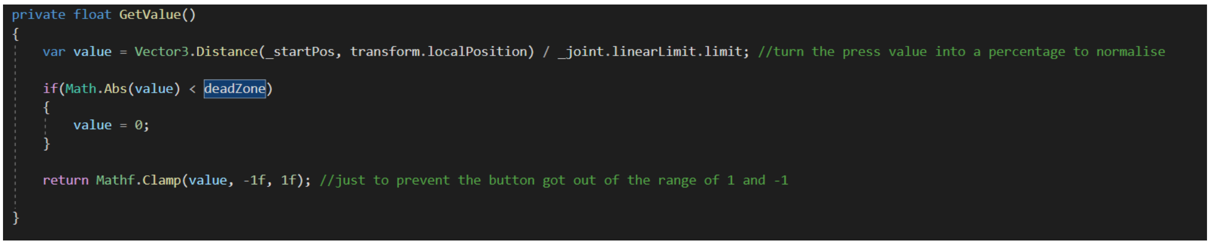


Figure 29: screenshot of VR player pressing toggle plane button

To implement a physics button that can be triggered. We need to get the value of how deep the clicker (the moveable part of the button) is pressed (see Figure 30). First, we need to get the delta distance of the clicker by Vector3.Distance() function, it returns the distance between the clicker’s startPos (pos stands for position) and current position. And divide it by the maximum distance of how low the button can be pressed to normalise the press value. Then I add an if statement to ensure the player really presses the button down for a certain amount, to prevent miss-pressing. deadZone here is a small float, if the value does not even exceed the dead zone of the button, then probably the player just miss-clicked, so we set the value to zero. And finally, we confine the range of values between 1 and -1. So as to prevent unnecessary error, it is possible to exceed the linear limit, because this is physics-based, so if the player smashes the button hard and quick, the clicker's position can be very low (the value will exceed the range).

After I get the value, the remaining is simple, if it exceeds a certain threshold, trigger the voidInteract() function of the interactable button like above.



```
private float GetValue()
{
    var value = Vector3.Distance(_startPos, transform.localPosition) / _joint.linearLimit.limit; //turn the press value into a percentage to normalise

    if(Math.Abs(value) < deadZone)
    {
        value = 0;
    }

    return Mathf.Clamp(value, -1f, 1f); //just to prevent the button got out of the range of 1 and -1

}
```

Figure 30: The function that calculate how deep a button is pressed.

For mobile players, they just have to get close to the interactable object and press the interact button (see Figure 31). The control logic is similar to a PC player using a controller, just that the controller is now virtualised.

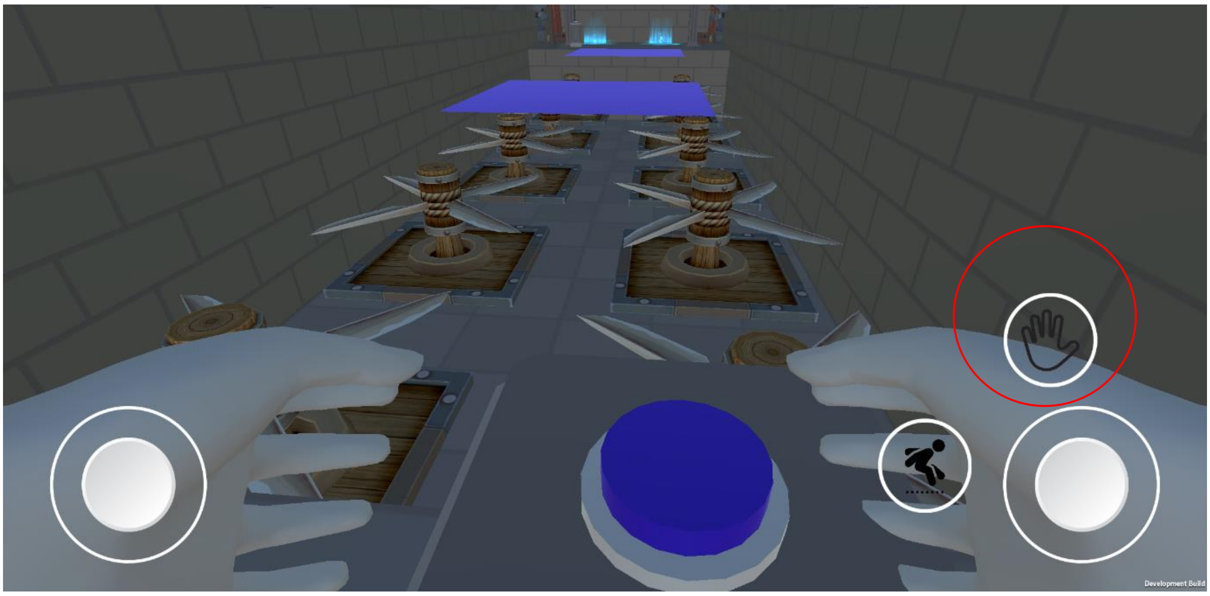
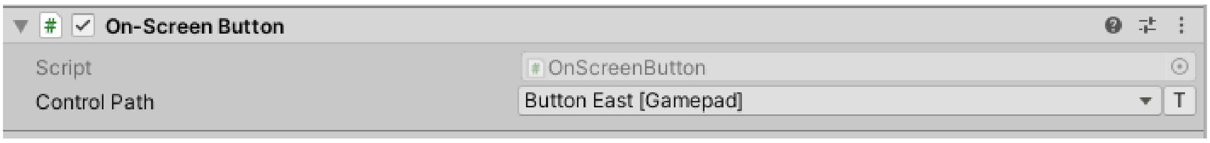


Figure 31: screenshot of mobile platform to show interact button

In fact, after investigation, I found that one way to implement mobile control with the new input system is to map the virtual buttons on the mobile canvas with other buttons. For example, I map the virtual interact button with the button east of the gamepad (the button for players using a controller to interact) (see Figure 32). With such an approach, clicking the virtual button is logically equivalent to clicking the physical button, so I can reuse most of my code for the PC controller player (see Figure 33) in mobile player control integration, enhancing maintainability.

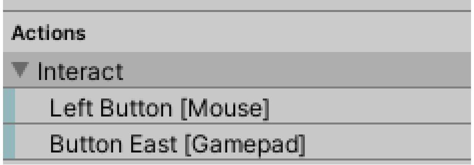


On-Screen Button

Script: OnScreenButton

Control Path: Button East [Gamepad]

Figure 32:On-Screen button component showing that is mapped to button east



Actions

- Interact
  - Left Button [Mouse]
  - Button East [Gamepad]

Figure 33:Action asset showing button east is mapped with interact action

5.5. Puzzle Level 2 – Conductor

5.5.1. Game Design

The floor plan is attached below for reference (see Figure 34). The goal of this level is to transmit a stream of magical energy from the start point to the endpoint. When the magical energy reached the endpoint, the secret door will be opened (see Figure 37) and that marks the end of the level. The orange line is used to represent conductors and there are gaps between conductors (labelled by number), please refer to Figure 35 for the actual setting of the game. When the player steps on the pressure sensor, a stream of magical energy will be transmitted from the start point. And when the energy reaches any unhandled gap, the energy will vanish. Therefore, players need to grip both ends of the conductor to close the gap of the conductor and after the energy pass through the player's body, the player will be paralysed (i.e., they cannot move for a couple of seconds) (see Figure 36). Therefore, if a player closed gap 1, it is impossible for that player to reach gap 2 on time. The solution of this level is to handle gaps and pressure sensor alternately by two players.

Please refer to http://i7.cs.hku.hk/~tlchan/FYP21057/Source/index.html for video demonstrations.

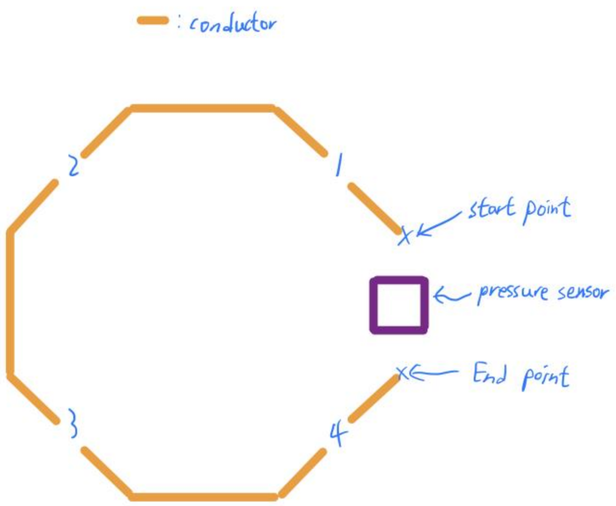


Figure 34: Floor plan of puzzle level 2 – conductor.

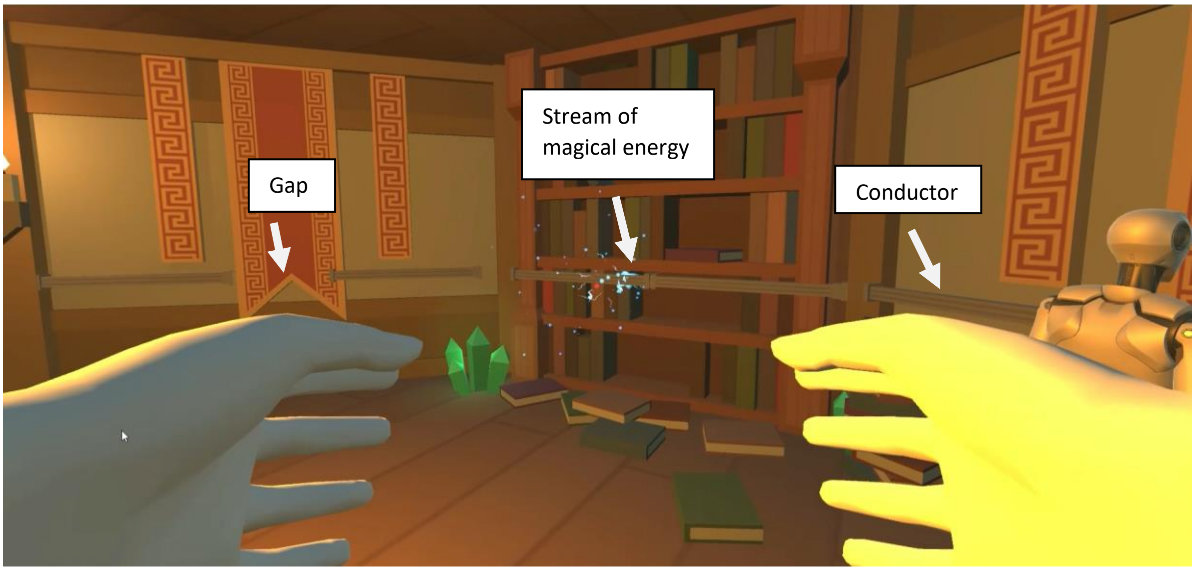


Figure 35: Screenshot demonstrating the setting of puzzle level 2 – conductor

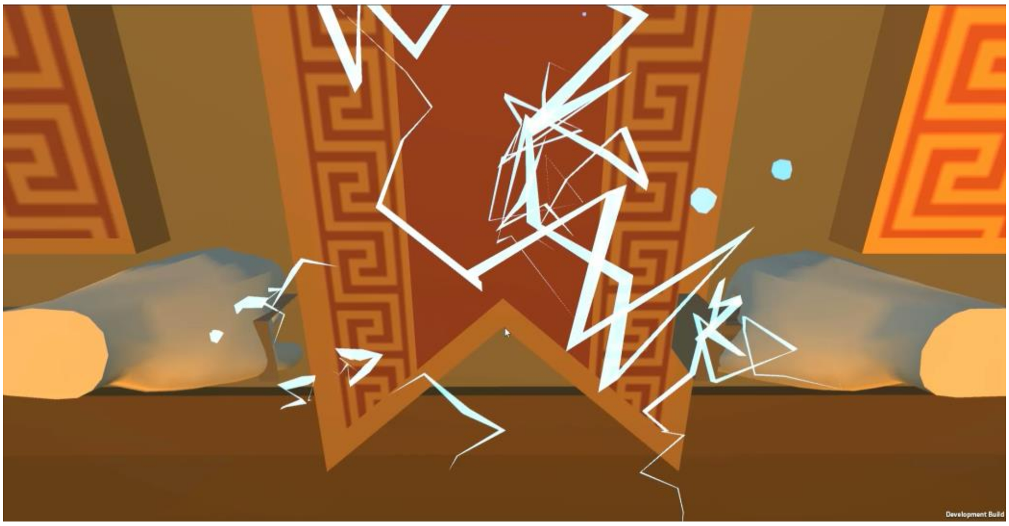


Figure 36: Screenshot demonstrating player close the gap by gripping both ends of the conductors. The lightning particle in the centre represents the player is paralysed.

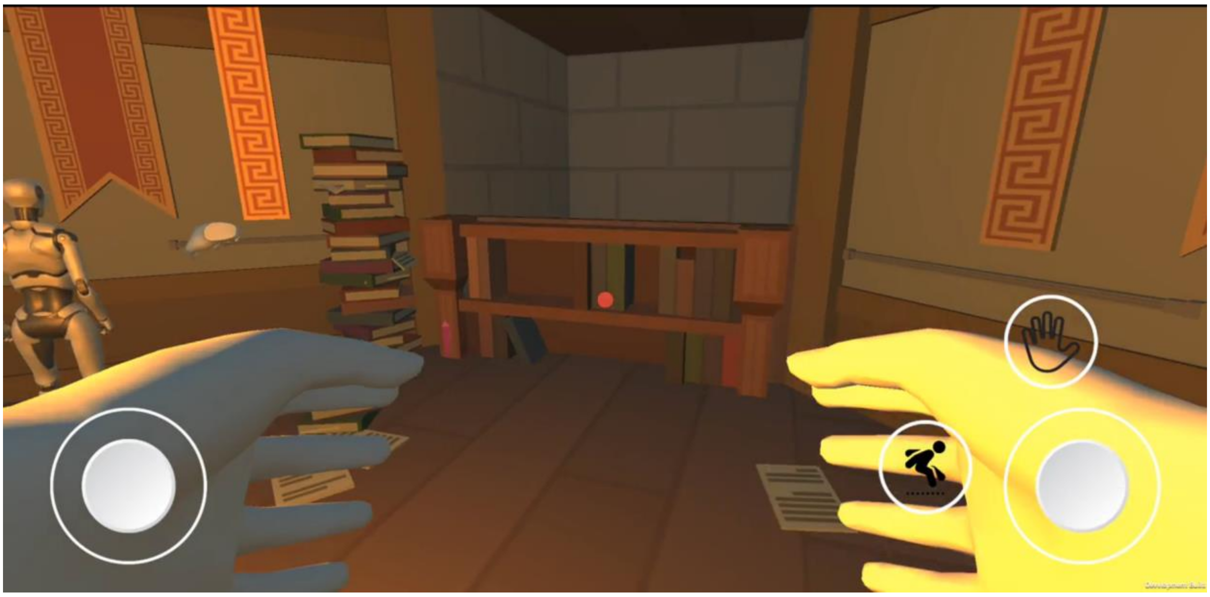


Figure 37: screenshot showing the secret door is opened when the stream of magical energy reached the other end.

5.5.2. Different Controls and implementation

For pc players with a mouse and keyboard, they will have to aim at the gap and press (and hold) the left and right-click of the mouse. Requiring the player to hold the button is a UX decision, this can give the player a sense that they are gripping something. The implementation is similar to Puzzle Level 1's. Please refer to section 5.4.2.

For pc players with the controller, they will need to hold the left and right trigger buttons (see buttons 8 and 9 of Figure 38). The implementation is similar to Puzzle Level 1's. Please refer to section 5.4.2.

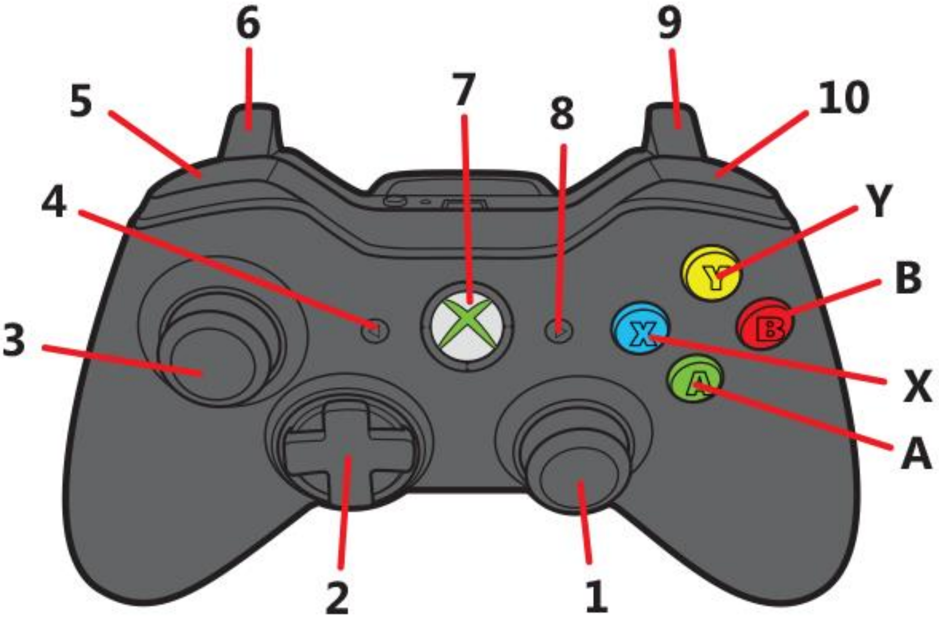


Figure 38: Xbox controller with label

For mobile player control, unlike pc player, considering that mobile player may need more convenient control. They don't need two buttons, instead, they just need to hold a virtual interact button (see Figure 39) like the mobile player of puzzle level 1. The implementation is similar, please refer to section 5.4.2.

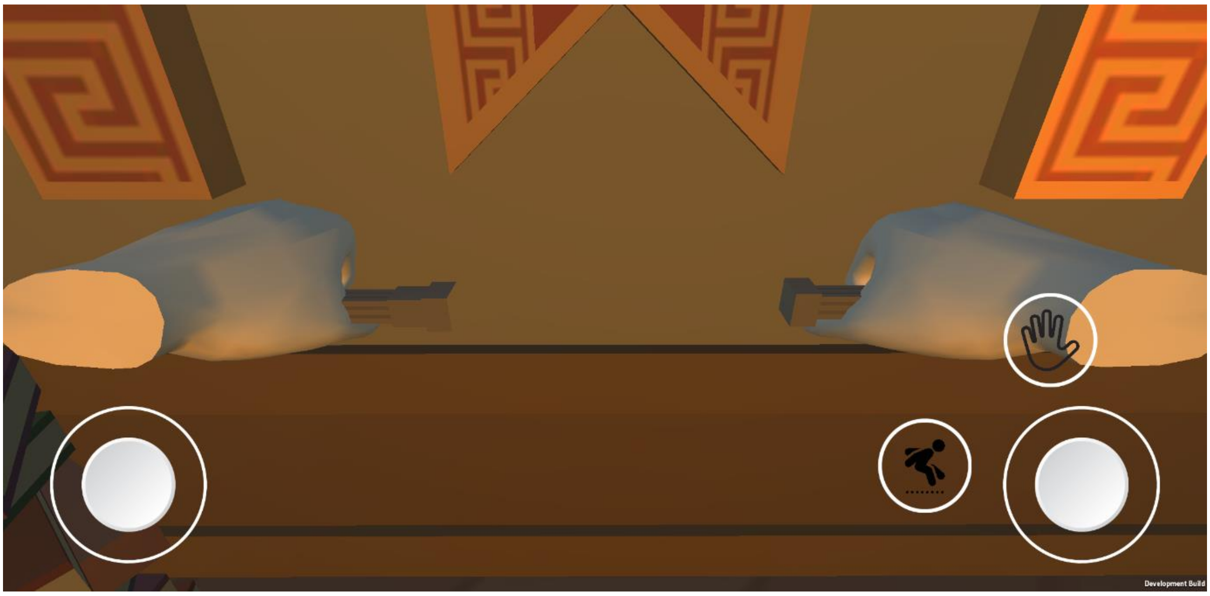


Figure 39: screenshot for mobile player in puzzle level 2

For VR players, the control is quite different, it is more intuitive (see Figure 40). First, they have to move their hands (controllers) to the ends of the conductors. Then, they press the trigger and grasp button (see Figure 41) at the same time to grip the conductor. Conventionally, pressing trigger and grasp button at the same time means grab action.

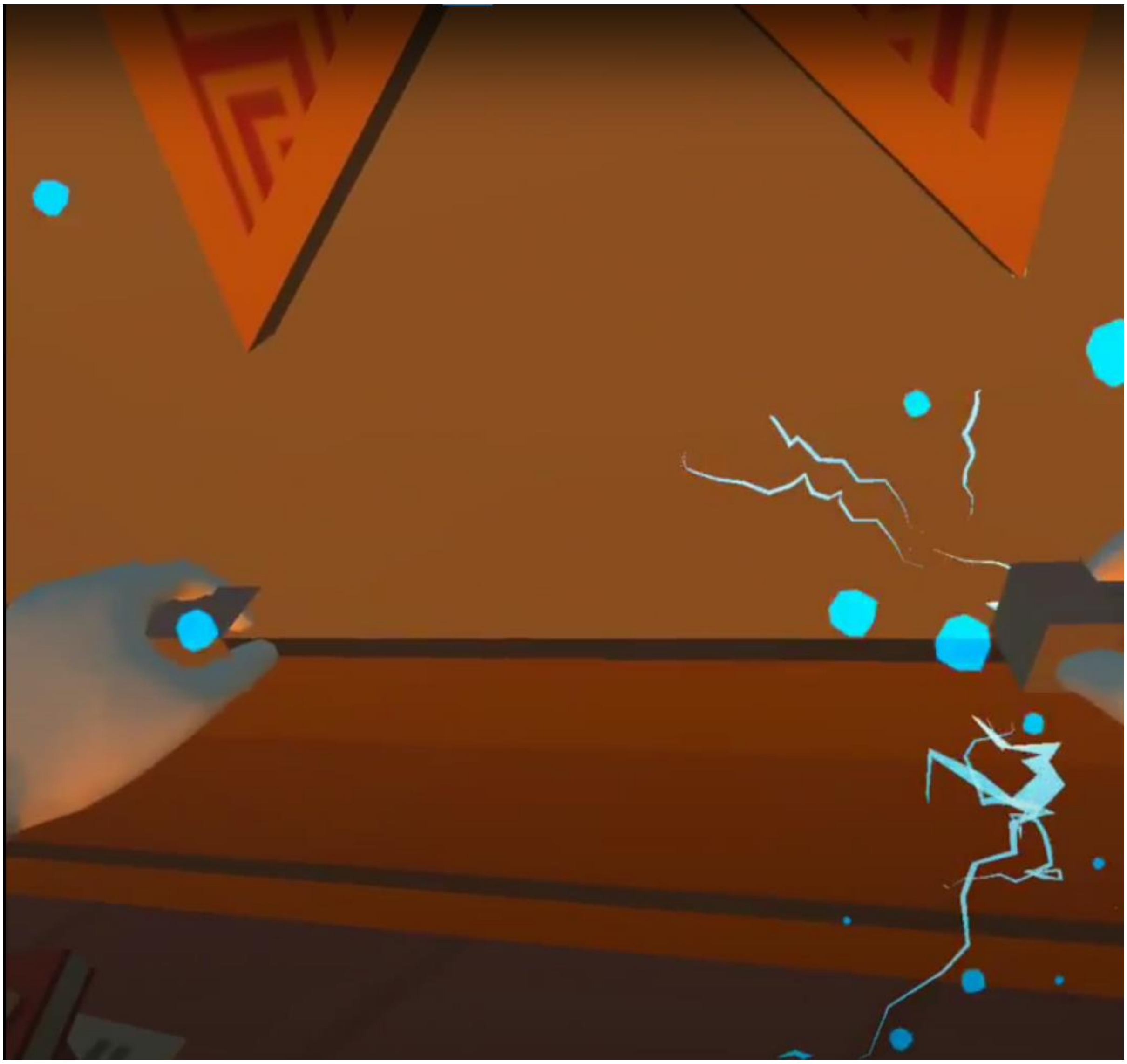


Figure 40: Screenshot of puzzle level 2 VR player, showing how they grip both end of conductor

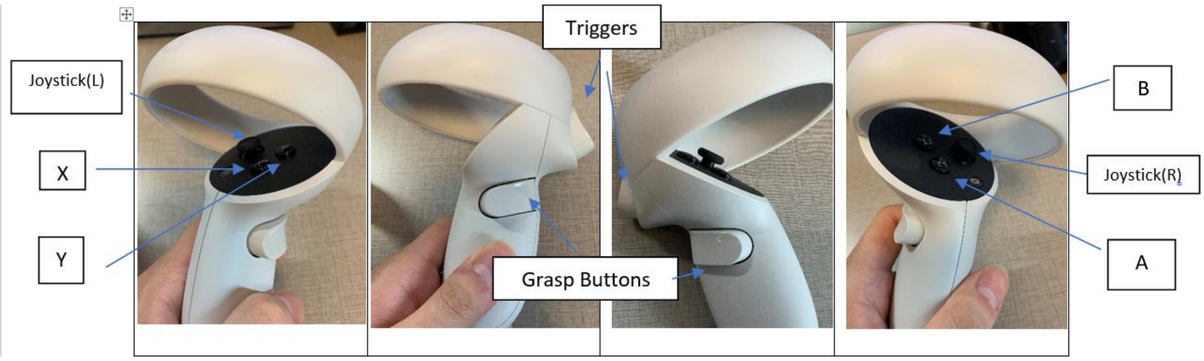


Figure 41: Oculus Quest 2 controller with label

To implement this, first, we need to detect whether the player's hand is in the proper place. As a result, we need to use a collider to detect whether the hand is in contact with the green boxes (see Figure 42). If the player is in the proper place and pressed the grasp and trigger button for both ends, then the gap is counted as closed.

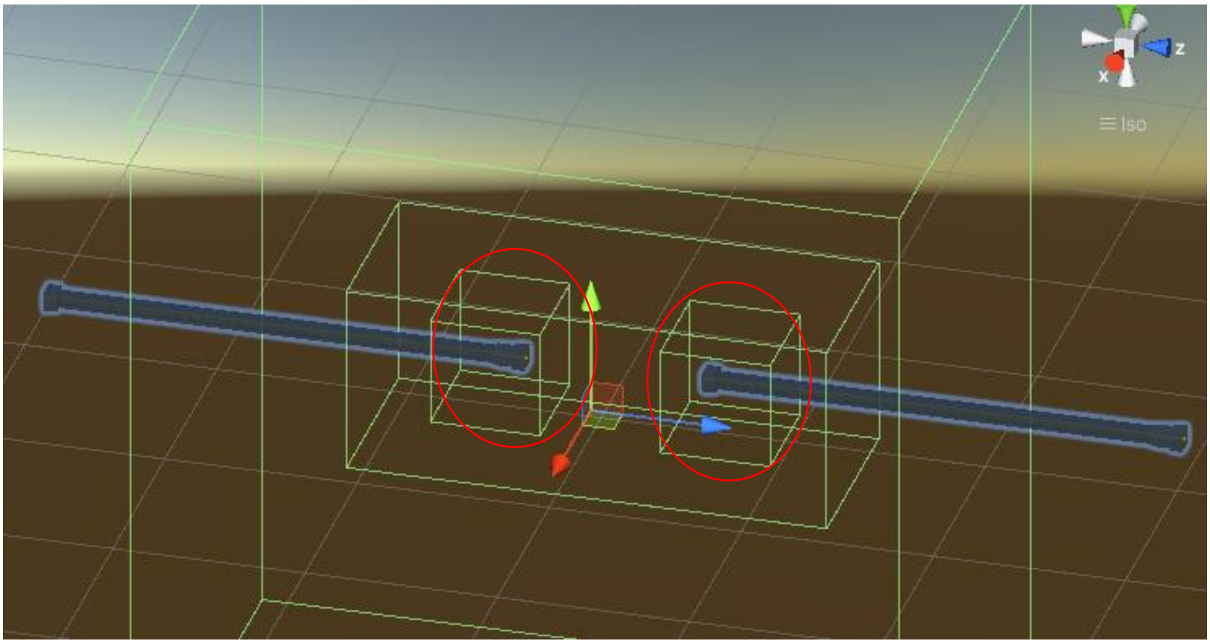


Figure 42: showing collider of conductor gap

In order to give a more realistic experience, there is one more thing needed to be addressed, which is the hand grab animation. To make a smooth hand grab animation, we need to make use of the blend tree. Blend tree can blend animation smoothly by incorporating parts of them to a different degree [13]. For normal animation, what we can choose is, whether we play the animation or not. There is a good reason that we need a blend tree for hand animation but for other ordinary animations we do not need to do so. Imagine the player pressing the grasp and trigger button and releasing immediately. What they will see is the hand, go all the way from the complete release state (see Figure 43) to the fist state (see Figure 44), and then move all the way from fist state back to the complete release state. Or the once released, the hand suddenly flashes to the fist state and starts playing the release animation (depends on what is the exit time setting in the animator). Neither of these is responsive nor realistic. Ordinary animation cannot handle it because it fixes the starting point of animation.

With the help of the blend tree, we can further fine-tune the “degree of animation”. For example, we can have a 50% release and 50% fist state. With the help of blend tree, if the player release before the hand state transition is completed (for example when the animation is 60% fist 40% release state), the animation will go back from 60% fist to 59% immediately, instead of going from 100% fist state.

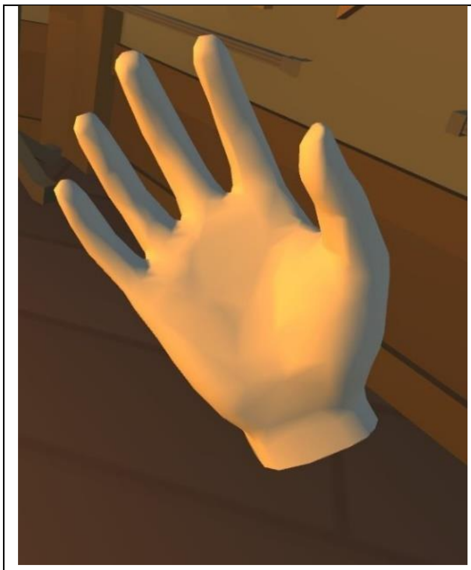


Figure 43: hand complete release state

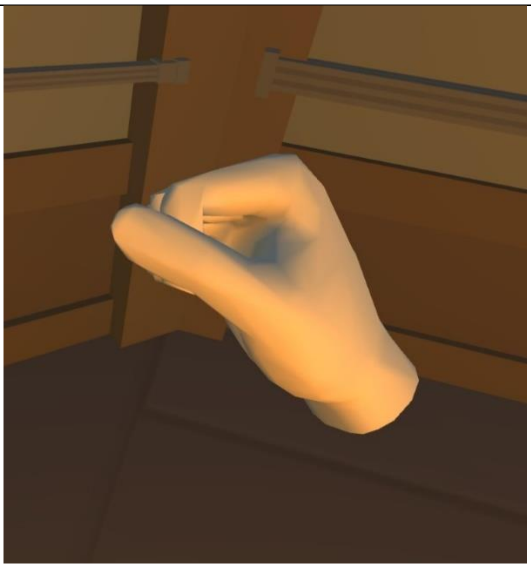
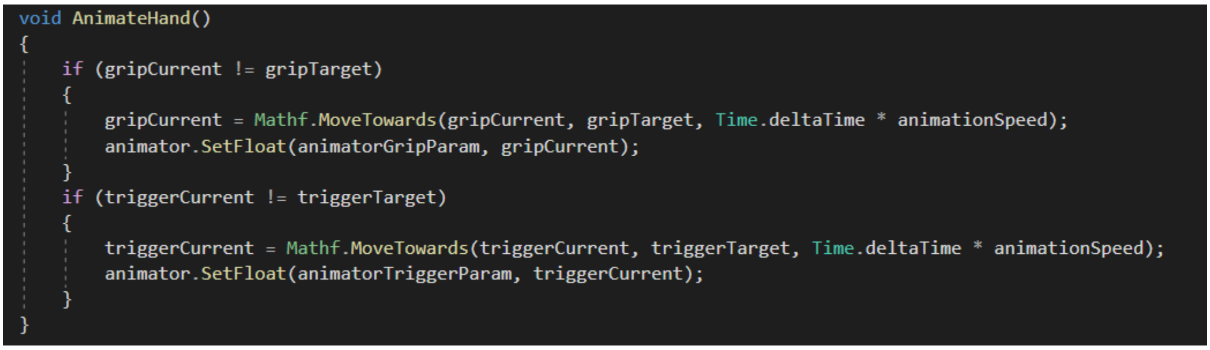


Figure 44: hand fist state

So, we smoothen the transition by the following lines of code (see Figure 45). If the current grip value (gripCurrent) is not equal to the target grip value (gripTarget), we will update the grip current value by Mathf.MoveTowards() function. This function calculates the interpolated value from two values. The first parameter is the start value; the second parameter is the end value; the third parameter is the interpolation value between the start value and end value. And we use animator.SetFloat() to pass the value to the blend tree to let it play the animation with the “correct degree”. For example, if the interpolation value is 75%, it will play 75% fist state mixed with 25% release state. This function is called in the Update() function (a function called every frame), which means the value is updated frequently to ensure the hand is responsive enough.



```
void AnimateHand()
{
    if (gripCurrent != gripTarget)
    {
        gripCurrent = Mathf.MoveTowards(gripCurrent, gripTarget, Time.deltaTime * animationSpeed);
        animator.SetFloat(animatorGripParam, gripCurrent);
    }
    if (triggerCurrent != triggerTarget)
    {
        triggerCurrent = Mathf.MoveTowards(triggerCurrent, triggerTarget, Time.deltaTime * animationSpeed);
        animator.SetFloat(animatorTriggerParam, triggerCurrent);
    }
}
```

Figure 45: Animate Hand function

5.6. Boss Fight

5.6.1. Game Design

The goal of the boss fight level is to beat a dragon by destroying all the weak spots of the dragon. However, the weak spots are hidden. Players will be given a different weapon. Please go to: http://i7.cs.hku.hk/~tlchan/FYP21057/Source/index.html for video demo.

One player will get a wand, which deals no damage, however, it can reveal the weak spots (see Figure 46).

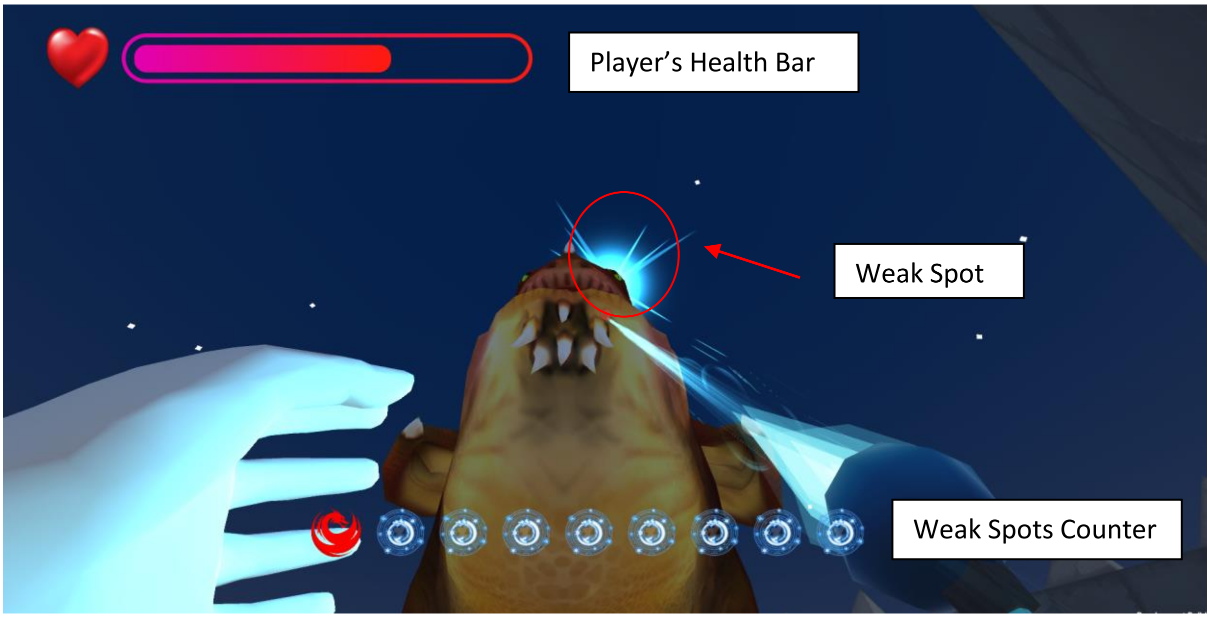


Figure 46: Screenshot of player using wand.

The other player will be given a magical gun, which deals damage to the dragon if and only if it hits the spots of the dragon (see Figure 47). And the player can only reveal a new weak spot only if there are no outstanding weak spots. That means he cannot create a new weak spot if the previous weak spot is not hit by the other player.

In terms of game design, both players rely on each other in order to beat the dragon. When all 8 weak spots are destroyed, the players win. Cooperation is the only way to defeat the dragon.

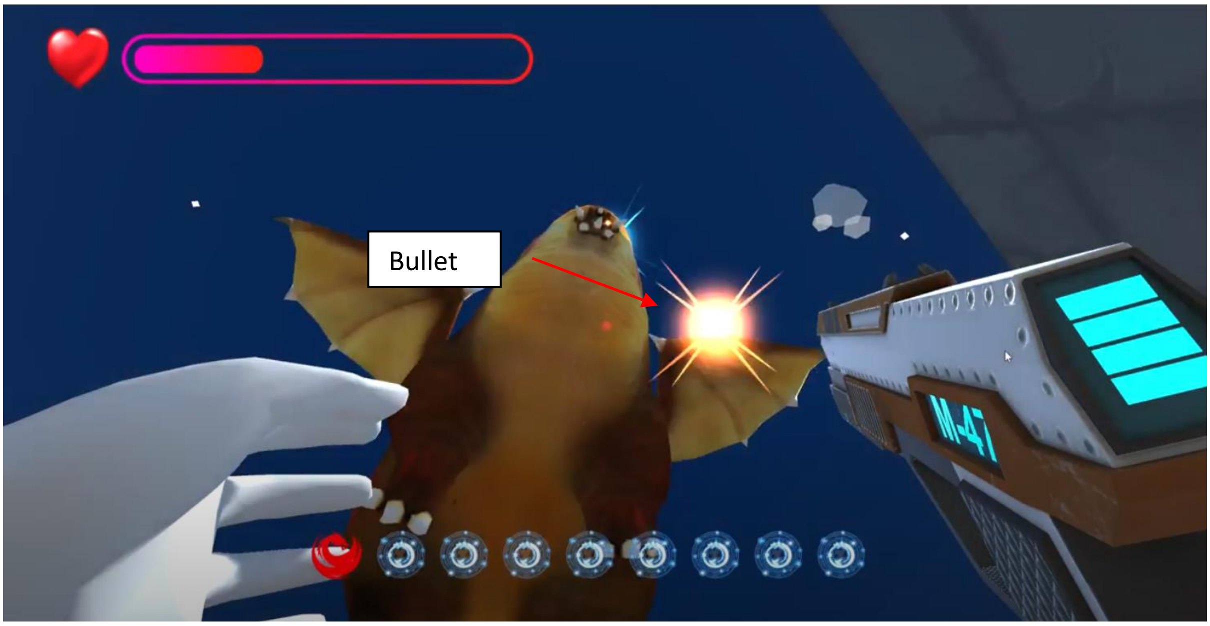


Figure 47: Screenshot of player using a magical gun

When one player died, he will go into the dead state (see Figure 48). The other player can go into the respawn area (see Figure 49) to respawn the other player. If unfortunately, both players die, the game is over.

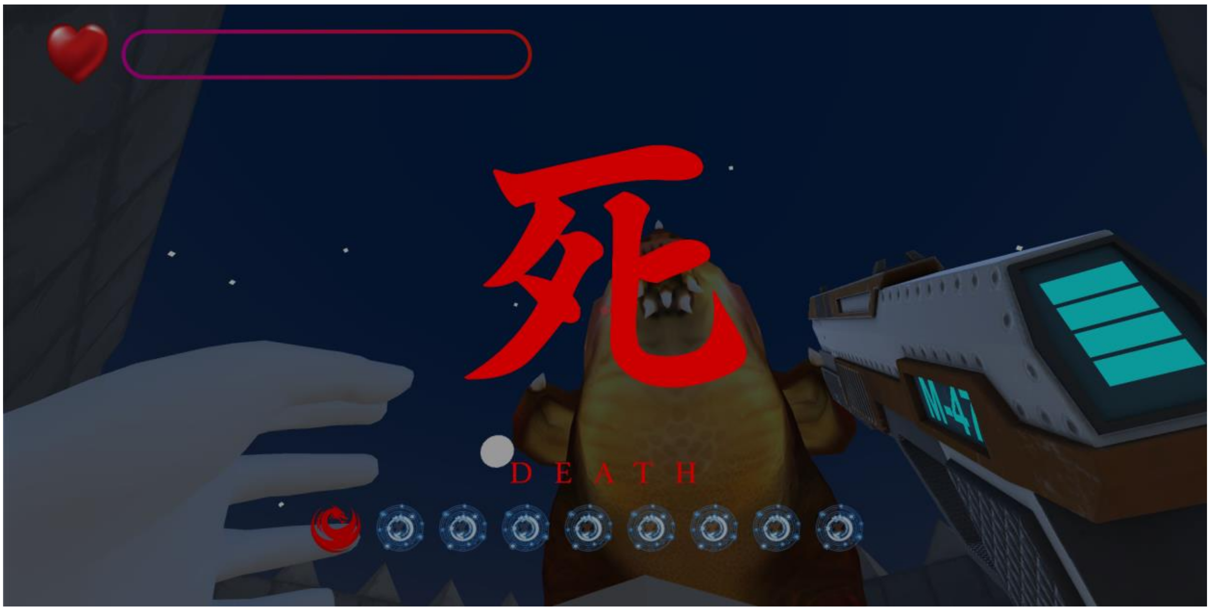


Figure 48: Screenshot showing the dead player.

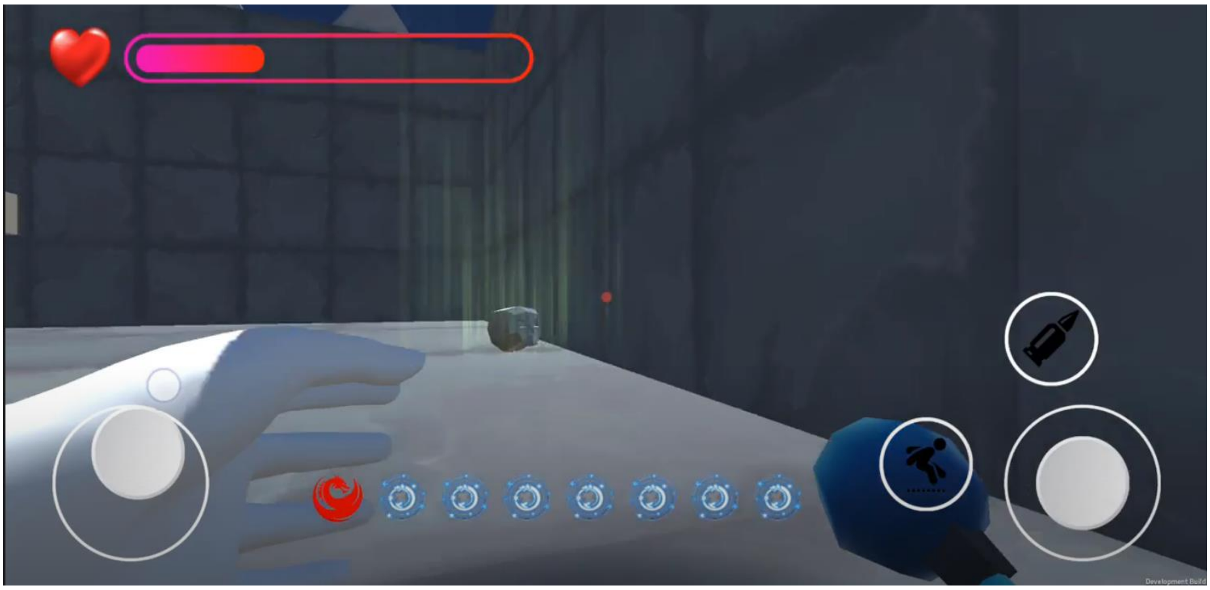


Figure 49: showing respawn area of player.

5.6.2. Control

For pc players with a mouse and keyboard, they will use the sign in the UI to aim and left click of the mouse to fire like most the first-person shooter.

For pc players with a controller, they will use the right trigger (see Figure 50 button 9) to fire.

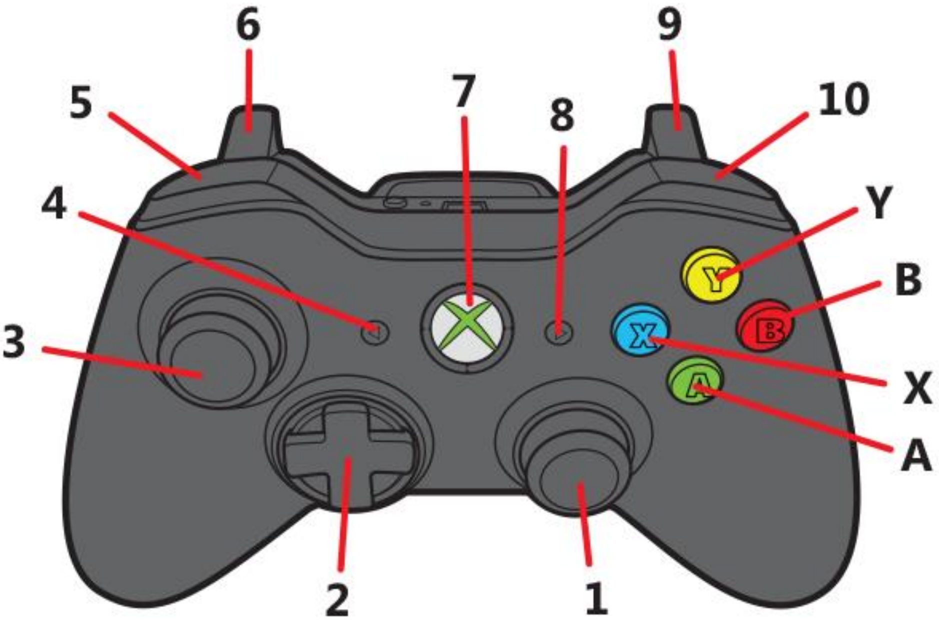


Figure 50: Xbox controller with label

For mobile players, they will be using the virtual fire button (see Figure 51).

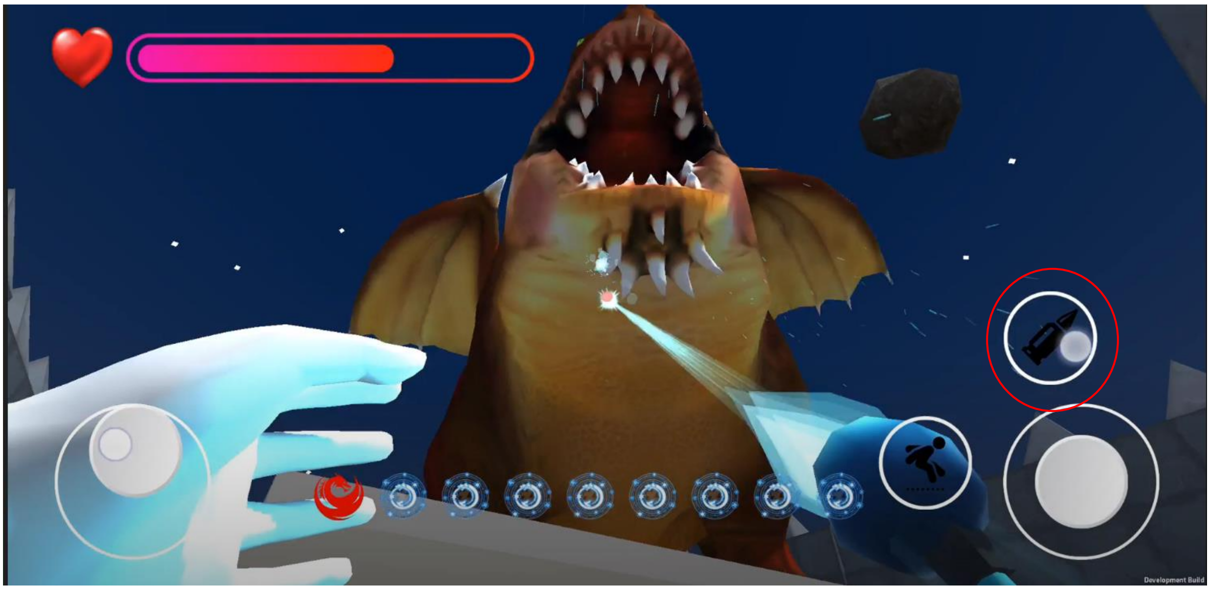


Figure 51:Screenshot of mobile player in boss fight level, showing the fire button (in red circle)

For VR players, they will use the right trigger (see Figure 52).

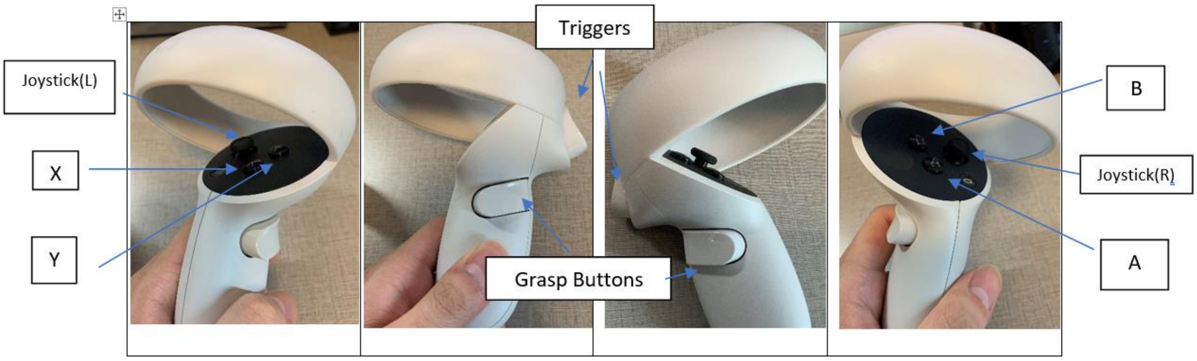
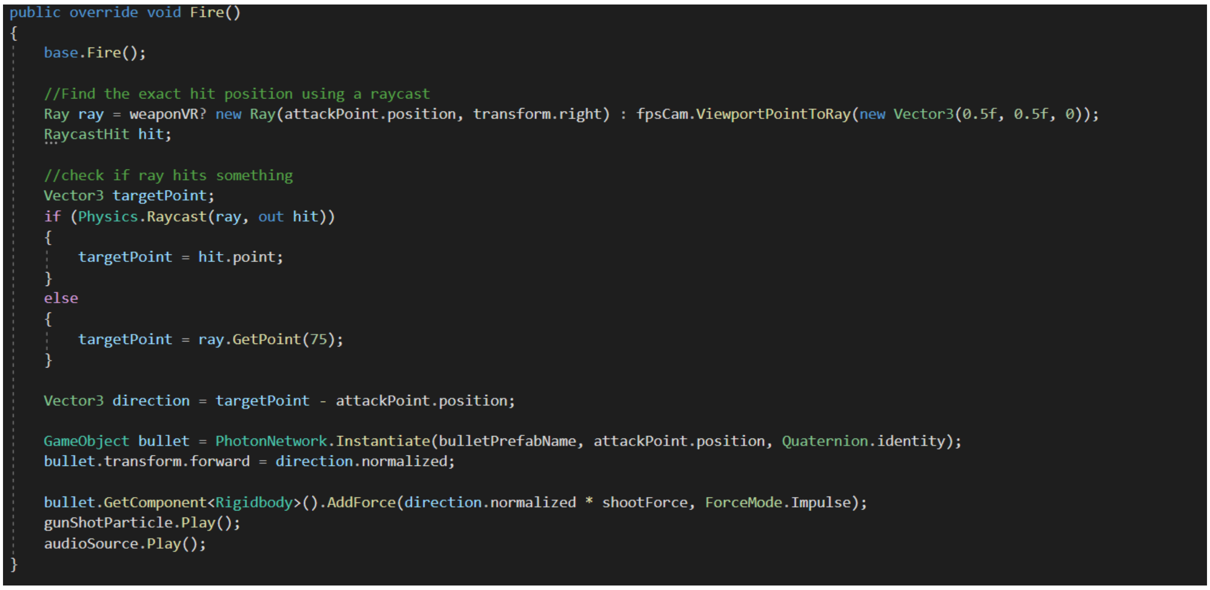


Figure 52: Oculus Quest 2 controller with label

5.6.3. Implementation – Shooting

To implement a gun that can fire bullets, we need to use raycast (see Figure 53). We have to handle two cases. The first case is when the player using non-VR devices, so the bullet must be shot in forwards (relative to the camera). In this case, similar to how we raycast the interactable object for keyboard and mouse users (see section 5.4.2), we use ViewportPointToRay() function to generate a ray forward from the camera point. The second case is the VR case, considering the player can shoot in any direction, we have to shoot the ray from the attack point of the gun (see Figure 54). Secondly, we have to check if the ray hit anything, if it hit anything, that is our aiming point. If it is not hitting anything (maybe the player is aiming at the sky or somewhere empty). In this case, we have to create a target point for the gun as an aiming point, ray.GetPoint(75), means to get the point of 75 unit array from the ray starting point. After that, we calculate the direction vector as the facing direction for the bullet. Then, we instantiate the bullet and set its orientation properly. We use PhotonNetwork.Instantiate() to ensure the bullet also spawned in the other player's computer. Finally, we add impulsive force to the rigidbody component of the bullet to create the shooting effect.



```
public override void Fire()
{
    base.Fire();

    //Find the exact hit position using a raycast
    Ray ray = weaponVR? new Ray(attackPoint.position, transform.right) : fpsCam.ViewportPointToRay(new Vector3(0.5f, 0.5f, 0));
    RaycastHit hit;

    //check if ray hits something
    Vector3 targetPoint;
    if (Physics.Raycast(ray, out hit))
    {
        targetPoint = hit.point;
    }
    else
    {
        targetPoint = ray.GetPoint(75);
    }

    Vector3 direction = targetPoint - attackPoint.position;

    GameObject bullet = PhotonNetwork.Instantiate(bulletPrefabName, attackPoint.position, Quaternion.identity);
    bullet.transform.forward = direction.normalized;

    bullet.GetComponent<Rigidbody>().AddForce(direction.normalized * shootForce, ForceMode.Impulse);
    gunShotParticle.Play();
    audioSource.Play();
}
```

Figure 53: Fire function of weak spot gun

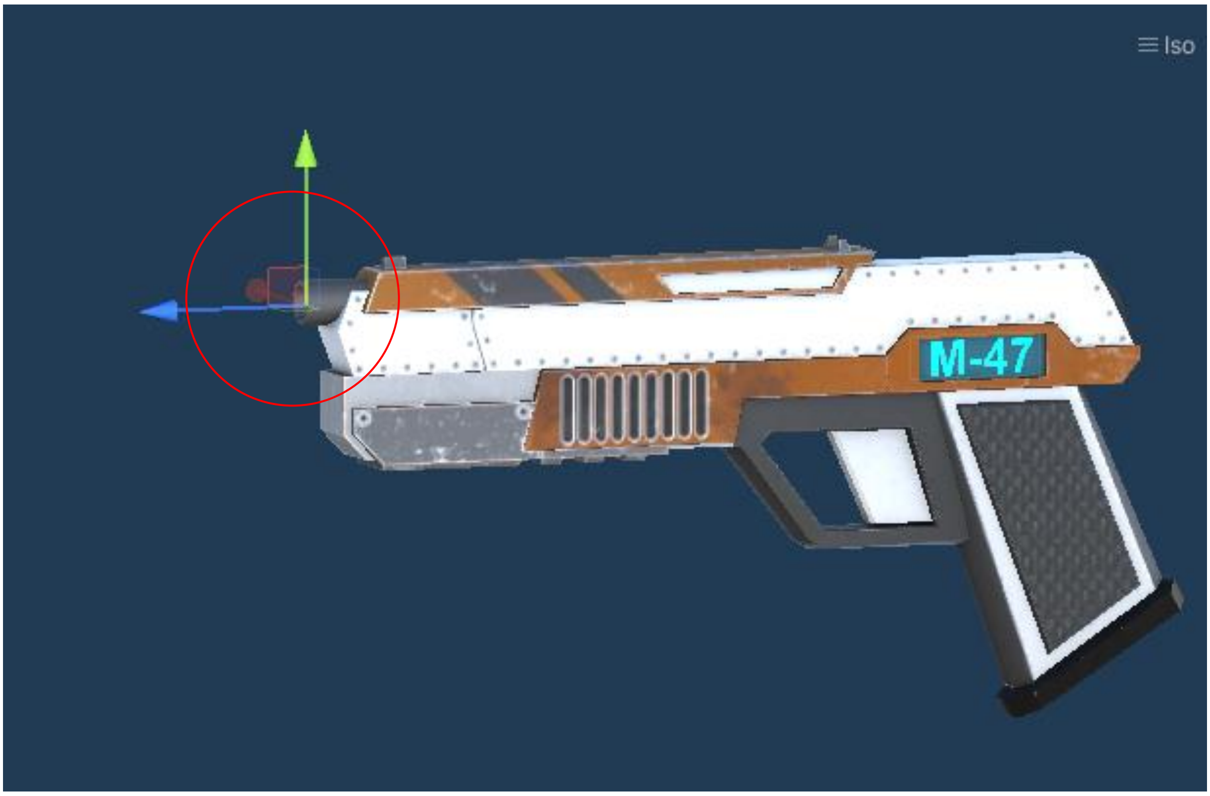
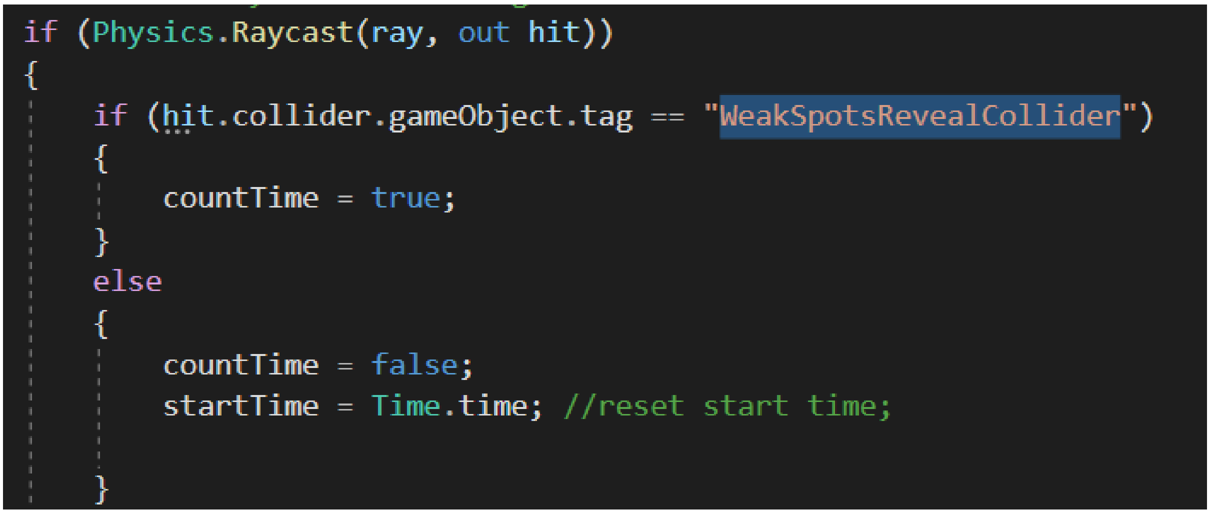


Figure 54: showing attack point of weak spot gun (circled)

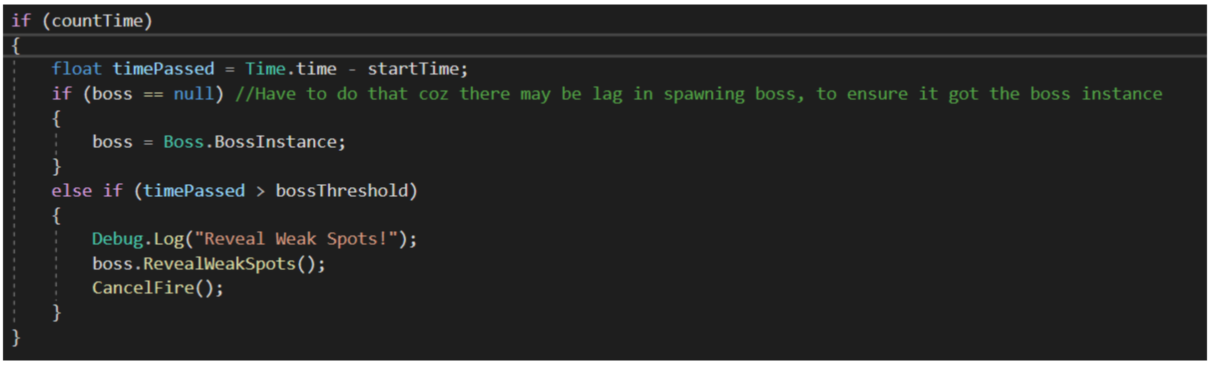
I faced one problem when I am using the weak spot gun. That is when I shoot a lot of bullets, the game start becomes lagging. After investigation, I found out that, when the bullet hit the boss, the bullet game object will be destroyed as I programmed. However, I miss a case where the player missed their shot. In that case, the bullet will fly infinitely towards empty space. This will cause game lagging because it wastes a lot of computing power to track the state of the missed bullet that is far far away. The solution is to add a timer to the bullet if a certain amount of time is passed. Destroy the bullet to enhance game efficiency.

The raycast implementation of the wand is similar (see Figure 55). The difference is this time detect whether the ray has collided with something with the tag "WeakSpotsRevealCollider", which is a collider of the dragon. And then I used the update function to act as a timer (see Figure 56). Time.time is the current game time, startTime is a time that we keep resetting until the ray hit the weak spots reveal colliders. So, the difference (time passed) between them is the time passed since the ray is hitting the boss. When this value passes a certain threshold, it will inform the boss object to reveal its weak spots.



```
if (Physics.Raycast(ray, out hit))
{
    if (hit.collider.gameObject.tag == "WeakSpotsRevealCollider")
    {
        countTime = true;
    }
    else
    {
        countTime = false;
        startTime = Time.time; //reset start time;

    }
```

Figure 55: code segment showing the raycast logic of wand



```
if (countTime)
{
    float timePassed = Time.time - startTime;
    if (boss == null) //Have to do that coz there may be lag in spawning boss, to ensure it got the boss instance
    {
        boss = Boss.BossInstance;
    }
    else if (timePassed > bossThreshold)
    {
        Debug.Log("Reveal Weak Spots!");
        boss.RevealWeakSpots();
        CancelFire();
    }
}
```

Figure 56: code segment showing timer logic in the Update() [a function that is called every frame].

5.6.4. Boss - Attack

The boss contains three types of attack: fireball attack, rock attack, laser attack. For the fireball attack (see Figure 57), the dragon will directly shoot a fireball from its mouth toward one of the players. For the rock attack (see Figure 58), the dragon will scream and summons a number of rocks that fall from the sky to hit players. For laser attack (see Figure 59), the dragon will summons 4 lasers, each moving in a certain direction (see Figure 60).

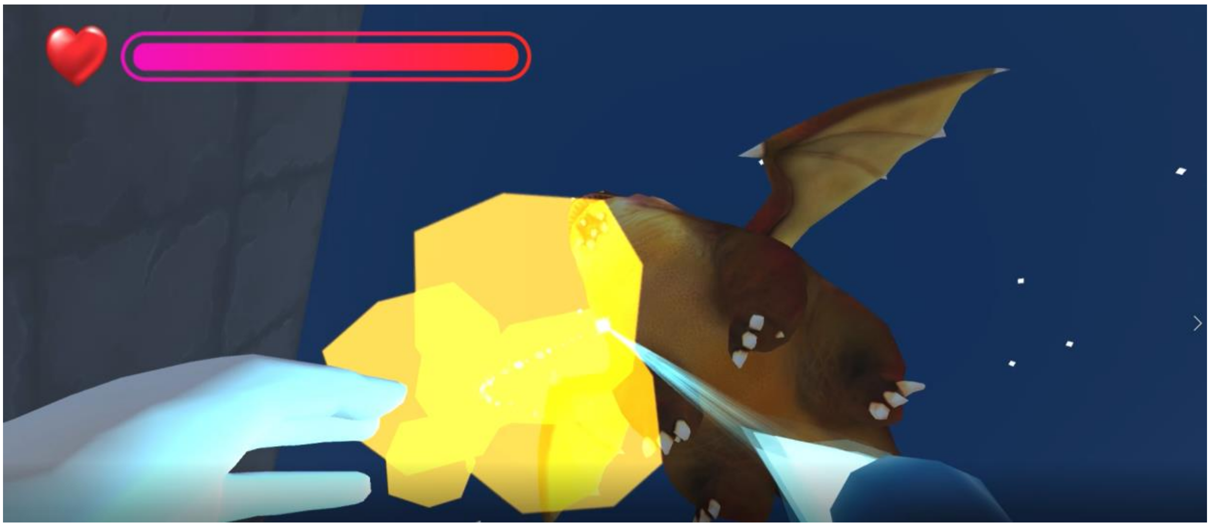


Figure 57: screenshot showing fireball attack

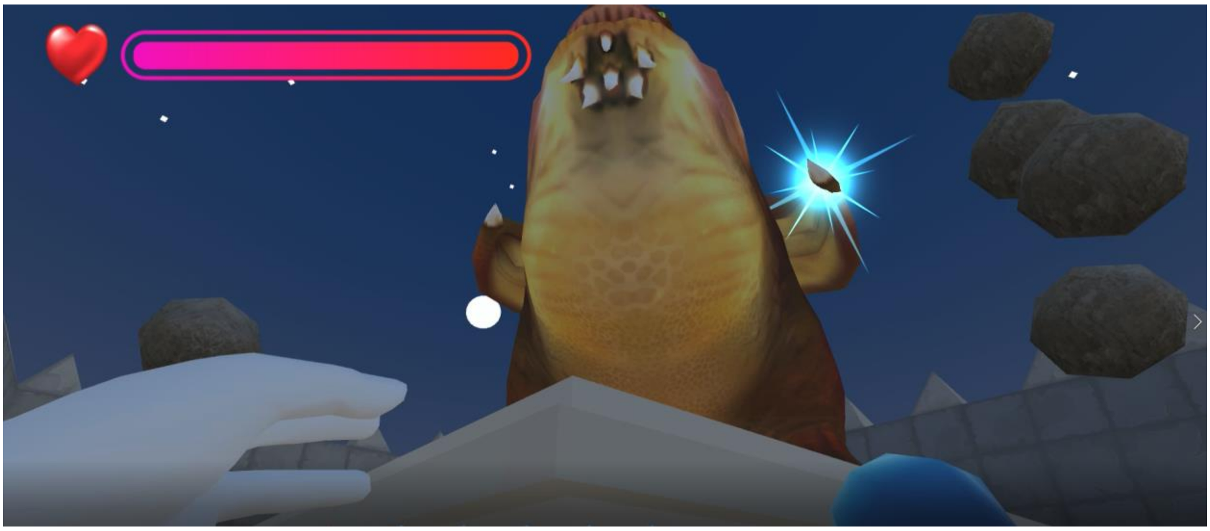


Figure 58: screenshot showing rock attack

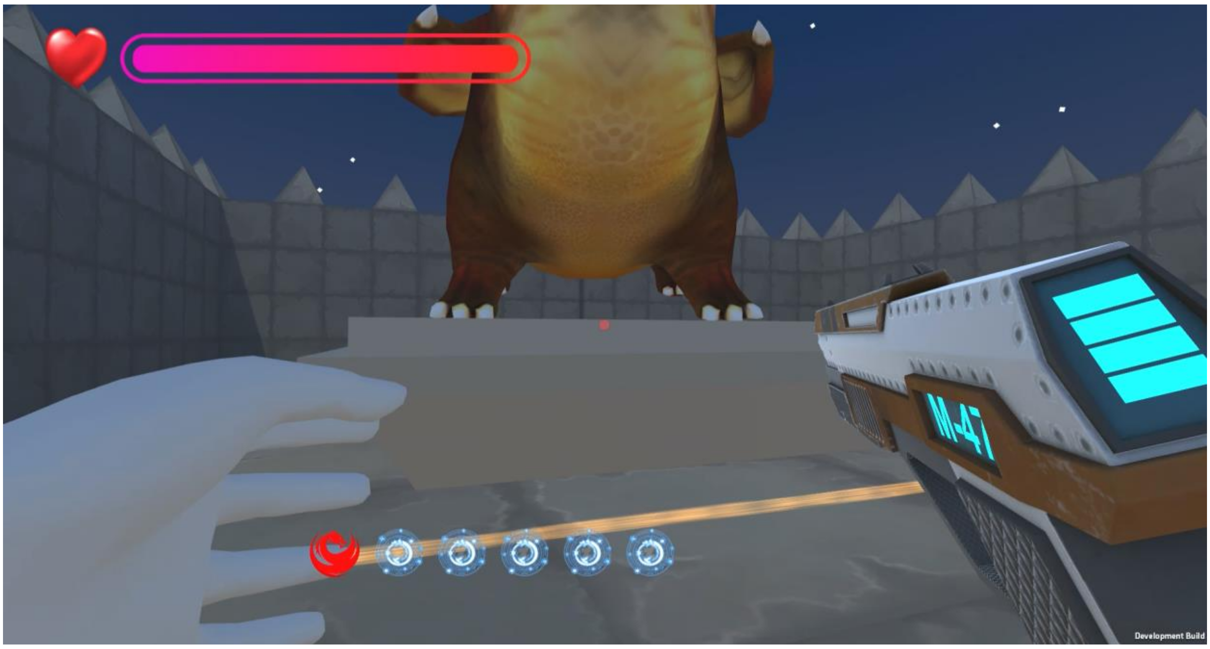


Figure 59: screenshot showing lazer attack

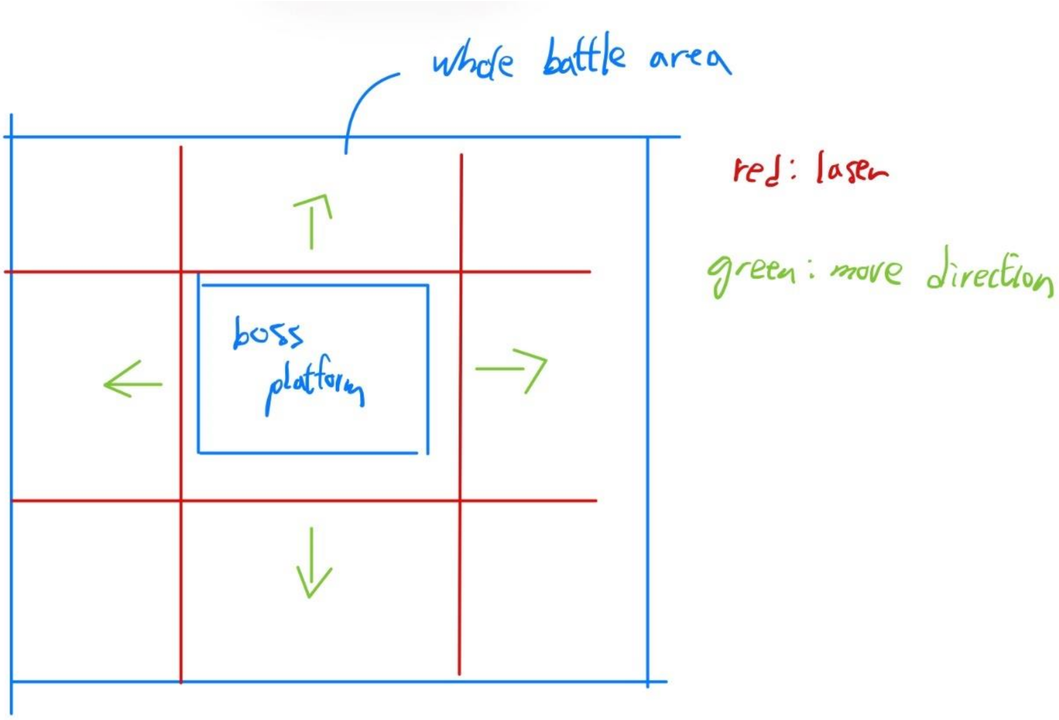


Figure 60: graphical illustration of lazer attack (bird view)

5.6.5. Cooperation Element

There are in total 8 weak spots on the dragon (see Figure 61). They are distributed in left/right eye, jaw, left/right wing, left/right back leg, and tail.

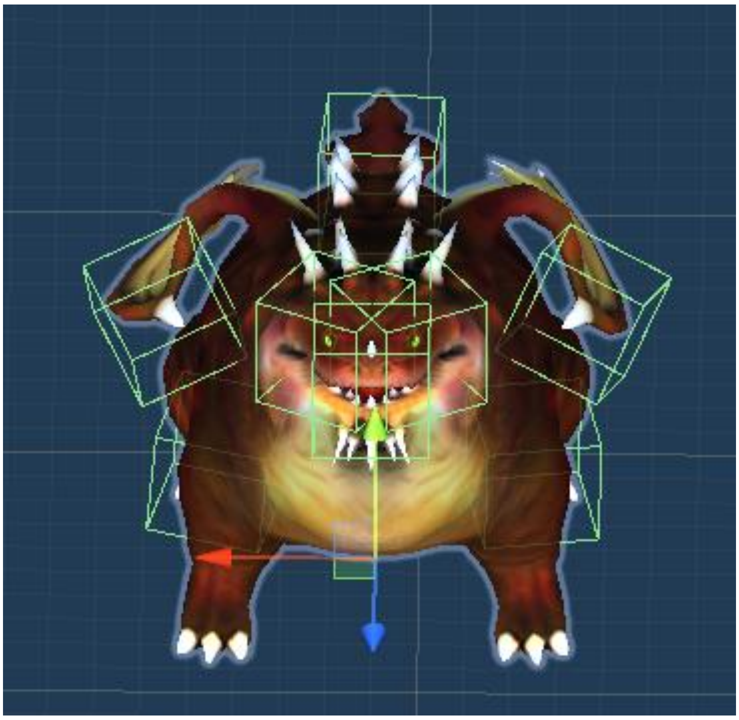


Figure 61: showing weak spot colliders (green boxes) of the dragon

Note that the dragon is not always facing a particular player. It will randomly choose a player as a target after a certain interval. The dragon will always face the targeted player. If it uses the fireball attack, it will always aim at the targeted player.

That is why the relative position of players is important. Take the below graph (see Figure 62) as an example, player A is targeted, player B is untargeted, and the weak spot is in the dragon's back leg. Case 1, if player B is the weak spot gun user, player A has to move towards his left to draw the dragon's attention while player B needs to move towards his right to aim at the weak spot properly. Case 2, if player A is the weak spot gun user, he may need to wait until the dragon flies and launches the fireball attack such that player A can aim at its leg (see Figure 57). Or player A can also wait until the boss switches its attention to player B. This example illustrates the importance of cooperation.

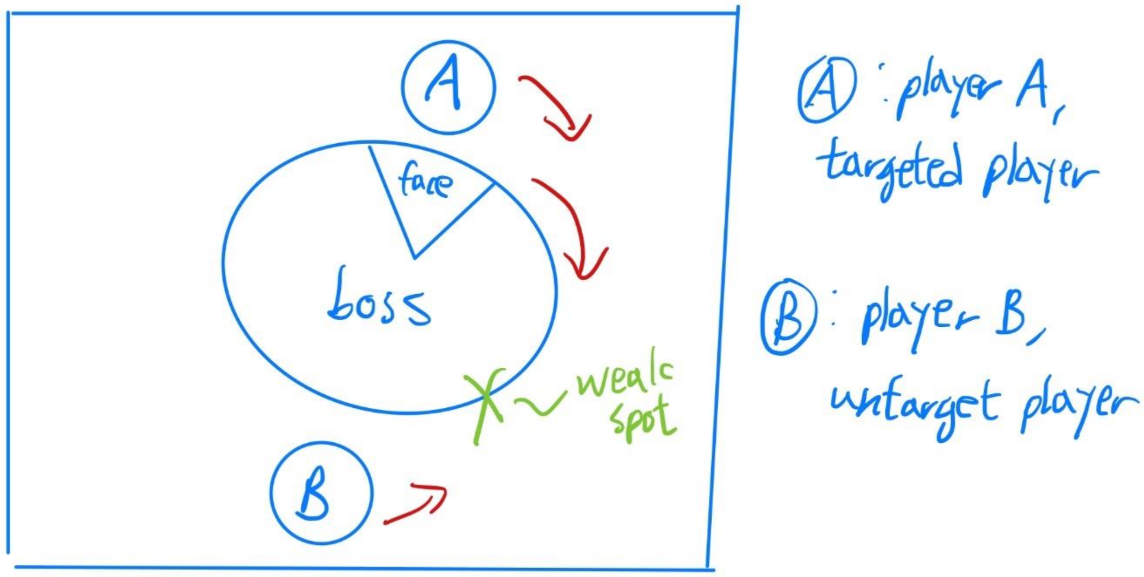


Figure 62: graphical illustration of the importance of relative position of player A, B and the boss (bird view)

5.7. Mobile Adaptation

The main challenge of supporting mobile platforms is not supporting the control. The main challenge comes from how to fill the gap between its computing power and laptops' computing power.

5.7.1. Asset Choice

I started considering the mobile adaptation issue at the very beginning of the project. In terms of asset choice, for most particle and 3D models, I choose the low poly art style. Low poly means a low number of polygons. The number of polygons is deeply affecting how intense the graphic computation needed to be. An example is given in Figure 63. You can see the image on the right is more realistic and smoother, which means a higher number of polygons. And on the right, you can see the model is more angled, which is low poly.

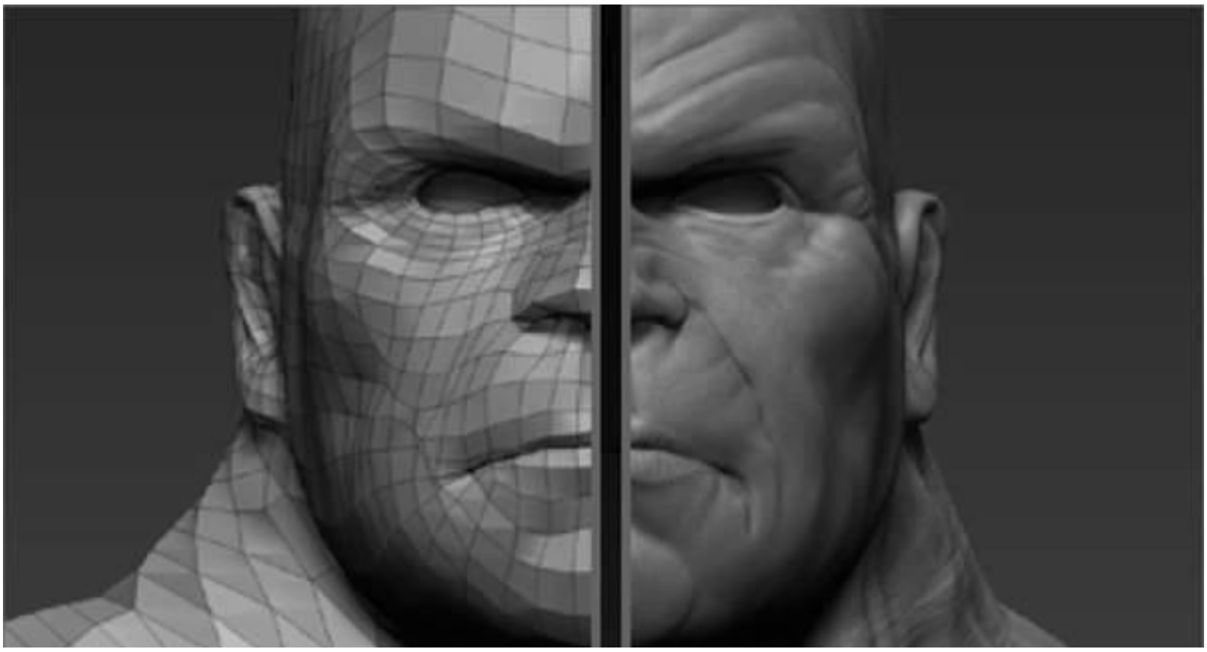


Figure 63: example demonstrating the difference between low poly and high poly 3D model

5.7.2. Game Design

Apart from the number of polygons, the number of objects and rendering distance also contribute a lot to the burden on the GPU of mobile phones. To address this through game design, I designed to have both puzzle levels 1 and 2 indoor. Although the boss fight is outdoor, the battlefield is not big and is confined by the wall, so the phone does not need to render something that is very far away. Therefore, the number of objects can be confined effectively.

Shadow is also another performance issue to be concerned about. For stationary objects, for example, windows, books, their shadow is fixed, we can simply prebake the light and shadow, it is not too costly. However, for moving objects, the lighting and shadow have to be recalculated in real-time. As mentioned, puzzle levels 1 and 2 are indoor scene, so lighting and shadow is not a big deal. However, for the boss fight level, it is an outdoor scene. The way I try to deal with the shadow issue is to set the time at night, so I do not have to take care of the shadow.

6. Difficulty and Limitation

During the development, I faced some unexpected issues. This section is to share what I learn from them and how to solve it.

6.1. Lighting Issue

There is a concept of scenes in Unity. For example, this game consists of four scenes, one menu scene and each level correspond to one scene. The lighting looks normal (see Figure 65) for the entry scene in my game. However, if I am trying to transit from one scene to another scene (i.e., the second scene onwards), the lighting will become much darker (see Figure 64). It is not an issue for that particular scene because no matter which scene I take as the entry scene, the lighting is fine.

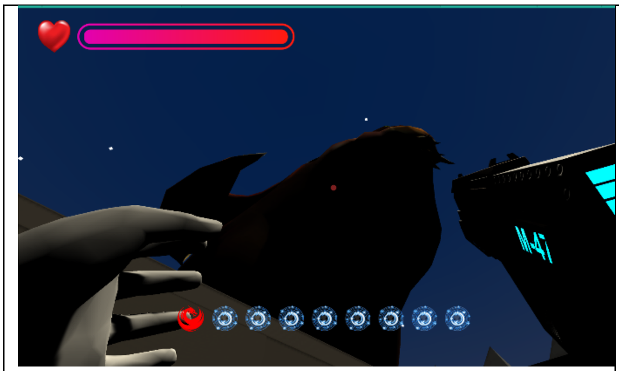


Figure 64: showing darken lighting

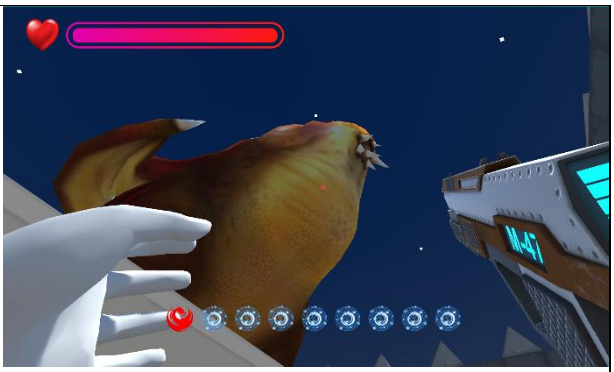
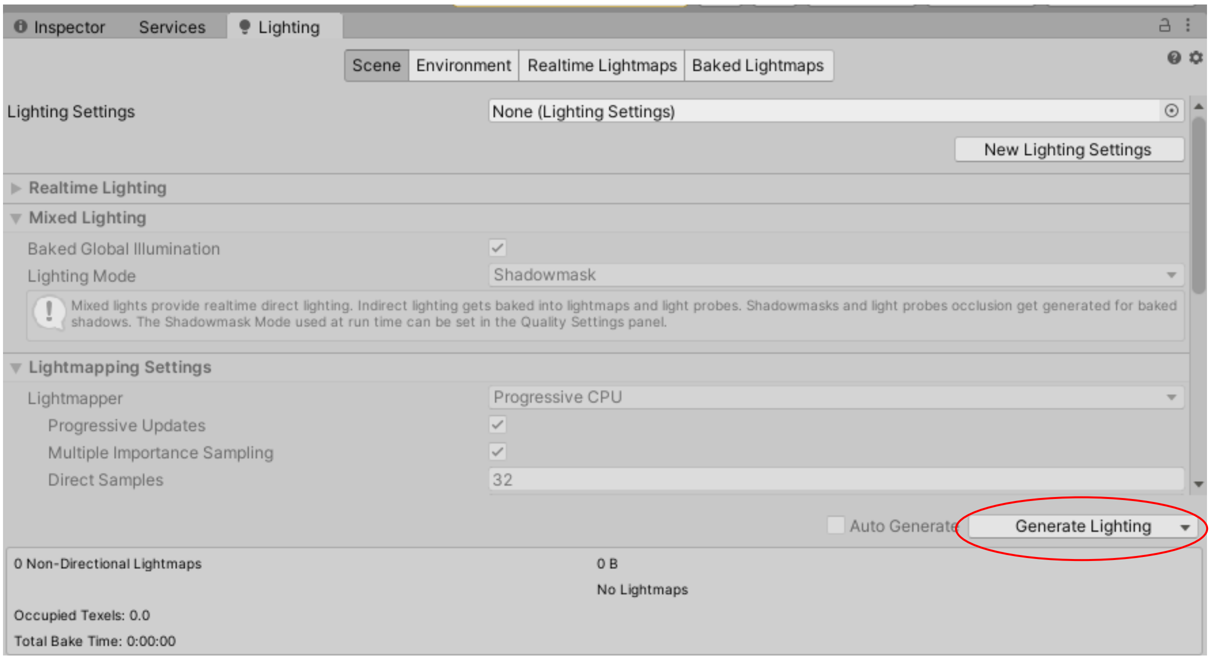


Figure 65:showing normal lighting

After rounds of investigation and debugging, I realise that Unity only automatically bakes all the lights for the first scene. As a result, if I transit to the other scene, which is not pre-baked, the lighting will not be functioning normally.

The solution is to manually prebake the lighting of all the scenes before I actually run the game so that lighting data of non-entry scenes is calculated before building your game to devices. Window → Rendering → Lighting to bring up the lighting tab (see Figure 66). Then click generate lighting, Unity will start baking the light for the scene.



Lighting Settings: None (Lighting Settings)

▶ Realtime Lighting

▼ Mixed Lighting

Baked Global Illumination ☑

Lighting Mode: Shadowmask

Mixed lights provide realtime direct lighting. Indirect lighting gets baked into lightmaps and light probes. Shadowmasks and light probes occlusion get generated for baked shadows. The Shadowmask Mode used at run time can be set in the Quality Settings panel.

▼ Lightmapping Settings

Lightmapper: Progressive CPU

Progressive Updates ☑

Multiple Importance Sampling ☑

Direct Samples: 32

□ Auto Generate

0 Non-Directional Lightmaps 0 B

No Lightmaps

Occupied Texels: 0.0

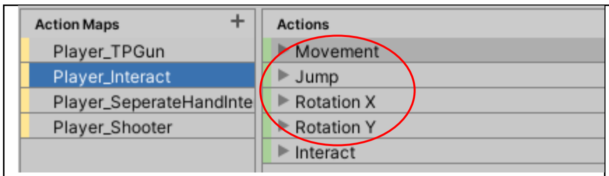
Total Bake Time: 0:00:00

Figure 66: showing the lighting tab in Unity

6.2. Inheritance of action map

To support multiplatform and levels, the key to having a maintainable programming structure is inheritance. As a result, I want a feature to support action map inheritance. For a game containing different levels, in each level players may have different sets of actions, but maybe some actions are commonly used and shared in many action maps. For example in this project, in both puzzle levels and boss fight level, players need to move around, jump, and look around (see Figure 67 and Figure 68Figure 68). If I can create a new action map that inherits a “base action map” containing these actions, I can simply add new specific action for that level (for example shoot action in the boss level is not needed for puzzle level).

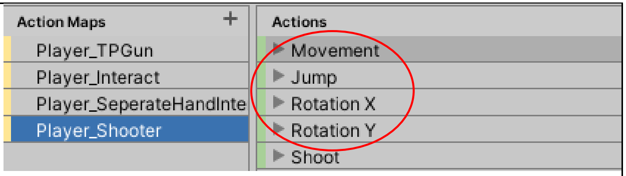
Without inheritance, maintainability is reduced. For example, if I want to change the button map to the “Jump” action, I have to go through all action maps and change each of them. However, after the investigation in the documentation, action map inheritance is not supported by Unity now. As a result, I have no choice but to copy all basic actions to all the action maps.



Action Maps: Player_TPGun, Player_Interact, Player_SeperateHandInte, Player_Shooter

Actions: Movement, Jump, Rotation X, Rotation Y, Interact

Figure 67: action map for puzzle level, the duplicated part in red circle



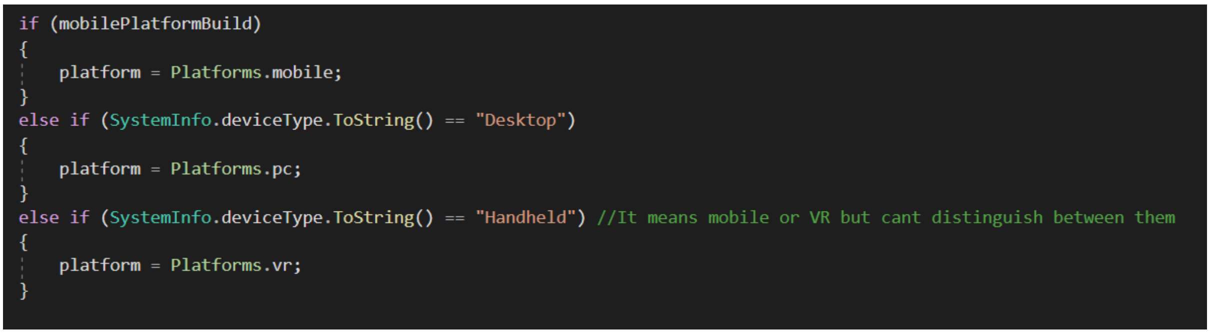
Action Maps: Player_TPGun, Player_Interact, Player_SeperateHandInte, Player_Shooter

Actions: Movement, Jump, Rotation X, Rotation Y, Shoot

Figure 68: action map for boss fight level, the duplicated part in red circle

6.3. Distinguish between VR and mobile platform

This product target three platforms: mobile, PC, and VR. I have to act differently for each of these platforms. For example, I have to spawn the mobile UI control canvas for the mobile platform, I have to spawn the VR player object instead of the ordinary player object for the VR platform, etc. What I want to achieve is that I can build a game that can distinguish the platform and act accordingly. Figure 69 shows the code for me to distinguish the platforms. SystemInfo.deviceType.ToString() will return the device type. However, after reading the documentation, there are only three defined device types, they are console (like PS5, Xbox), Desktop, and handheld. Both mobile phones and Oculus headsets are categorized as handheld devices. The function Unity created cannot further distinguish what kind of handheld machine is it. Indeed, there are also other alternatives like SystemInfo.deviceModel, it returns the model of the device like, “iPhone 11”, “Google Pixel 3a” etc. It is true that I can simply use this to distinguish whether it is an Oculus device or not. However, this is not a good practice either. I will need to manually update the device list if a new device is used. The workaround I am currently using is simply to use a bool created by me, mobilePlatformBuild to act as the flag to distinguish whether it is the mobile platform or VR platform.



```
if (mobilePlatformBuild)
{
    platform = Platforms.mobile;
}
else if (SystemInfo.deviceType.ToString() == "Desktop")
{
    platform = Platforms.pc;
}
else if (SystemInfo.deviceType.ToString() == "Handheld") //It means mobile or VR but cant distinguish between them
{
    platform = Platforms.vr;
}
```

Figure 69: code segment for platform differentiation

7. Future Development

In terms of technology, now oculus players have to use the VR controllers to control the game, I can support hand tracking in VR puzzle levels (in this case the player will not even need the controller) so that the player can have an even more intuitive experience. I was not doing that in this project because I have a boss fight level where the player is holding weapons. In order to unify the experience so that I can have a more structured (class inheritance) and reusable code, I use VR controllers for all levels.

In terms of game design, I can expand the game in some directions.

First, the enemy AI (the dragon) is not smart enough. Now the dragon simply decides the target player randomly. I can improve the AI such that I will target the player who shoots more in front of it so that the player can help their partner distract the dragon strategically.

Second, due to time and resources limitations. There are only three levels. Bigger and more complicated levels can be added in the future.

Third, even though the control is not that hard to explore, it will be better if the game contains a tutorial to teach the player how to control it.

In terms of user experience, now I do not implement the vibration feature. I think it will be nice to have the vibration feature, especially for the VR player.

8. Conclusion

Given the rapid expansion of the VR gaming market, it is profitable to increase the hardware coverage and player base of a game. This project will be valuable to the gaming industry as a demonstration. Therefore, the goal of the project is to develop a game that supports cross-platform multiplayer. To achieve this without increasing too much developmental cost, this project supports multiple platforms with one generic code base that can handle different kinds of inputs from the touchable screen, keyboard and mouse, gamepad, and VR controller. The final deliverable is a 2-player multiplayer cooperative game that contains three levels, two puzzle levels, and one boss level, supporting window PC (keyboard and mouse, and controller), Android phone, and VR devices (oculus quest 2).

However, some limitations prevent the project from achieving the goal perfectly. In the market, there are too many gaming platforms. It is impossible to support all these platforms for two reasons. Firstly, to test my game on all platforms, all kinds of hardware are needed, for example, Nintendo Switch, PlayStation Machine, Xbox Machine, Mac Machine, etc. It is impossible to purchase these gaming consoles and devices with $1000 (FYP budget). Secondly, the time is limited, different hardware has its interface (analogy: like different adaptors, use different ports), it is infeasible to support all of them within seven months. Although this project cannot support all gaming platforms, this project does support the gamepad which is commonly used for different gaming platforms, making the project sufficient to prove the feasibility to support all platforms if time is allowed.

In addition, a polished game should not be expected. In normal game development, there will be at least two roles – game programmer and artist. Without proper art support and sufficient time, a demo-sized unpolished game will be developed as a result.

Reference

[1] Britannica,“Virtual reality,” *Encyclopædia Britannica*. [Online]. Available: https://www.britannica.com/technology/virtual-reality. [Accessed: 26-Oct-2021].

[2] J. Clement, “Video game, gaming industry revenue,” *Statista*, 01-Jun-2021. [Online]. Available: https://www.statista.com/statistics/278181/global-gaming-market-revenue-device/. [Accessed: 24-Oct-2021].

[3] J. Clement, “Global VR Gaming Market Size 2024,” *Statista*, 29-Jan-2021. [Online]. Available: https://www.statista.com/statistics/499714/global-virtual-reality-gaming-sales-revenue/. [Accessed: 24-Oct-2021].

[4] S. Petite and J. Lennox, “All cross-platform games (PS5, Xbox Series X, PS4, Xbox One, switch, PC),” *Digital Trends*, 01-Apr-2022. [Online]. Available: https://www.digitaltrends.com/gaming/all-cross-platform-games/. [Accessed: 04-Apr-2022].

[5] N. Gilbert, “74 virtual reality statistics you must know in 2021/2022: Adoption, usage & market share,” *Financesonline.com*, 06-Apr-2021. [Online]. Available: https://financesonline.com/virtual-reality-statistics/. [Accessed: 24-Oct-2021].

[6] GlobalStats , “Mobile Operating System Market Share Worldwide,” *StatCounter Global Stats*. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide. [Accessed: 24-Oct-2021].

[7] Unity Technologies, “Multiplatform,” *Unity*. [Online]. Available: https://unity.com/features/multiplatform. [Accessed: 24-Oct-2021].

[8] Unity Technologies, “Multiplayer and networking,” *Unity*. [Online]. Available: https://docs.unity3d.com/Manual/UNet.html. [Accessed: 10-Apr-2022].

[9] Photon, “Pun,” *Photon Unity 3D Networking Framework SDKs and Game Backend | Photon Engine*. [Online]. Available: https://www.photonengine.com/pun. [Accessed: 24-Oct-2021].

[10] samyam. "Why You Should Use The New Input System In Unity + Overview," *YouTube*, Feb 28, 2021. [Video file]. Available: https://www.youtube.com/watch?v=GyKBoDF_Oxo. [Accessed: Dec 15, 2021].

[11] Photon, “Synchronization and State,” *Synchronization and State | Photon Engine*. [Online]. Available: https://doc.photonengine.com/en-us/pun/current/gameplay/synchronization-and-state#object_synchronization. [Accessed: 21-Jan-2022].

[12] Photon, “RPCs and Raiseevent,” *RPCs and RaiseEvent | Photon Engine*. [Online]. Available: https://doc.photonengine.com/en-us/pun/current/gameplay/rpcsandraiseevent. [Accessed: 21-Jan-2022].

[13] Unity Technologies, “Blend trees,” *Unity*. [Online]. Available: https://docs.unity3d.com/Manual/class-BlendTree.html. [Accessed: 14-Apr-2022].

Appendix

Detail Timeline for this project

Date	Task
Summer 2021	Choose/Draft FYP Topic
15 Sept 2021	Technical Investigation • VR Development • Cross-platform Multiplayer • Built-in Voice Chat
3 October 2021	Deliverables of Phase 1 (Inception) • Detailed project plan • Project web page
15 October 2021	Finalize game design
31 December 2021	Puzzle Level 1 and 2 Implementations
10-14 January 2022	First Presentation
23 January 2022	Deliverables of Phase 2 (Elaboration) • Preliminary implementation • Detailed interim report
31 January 2022	Boss Fight Level Implementation
28 February 2022	VR Control Integration
31 March 2022	Touchable Screen (Mobile) Control Integration
15 April 2022	Final Test and Adjustment
18 April 2022	Deliverables of Phase 3 (Construction) • Finalized tested implementation • Final report
19-22 Aril 2022	Final presentation
4 May 2022	Project exhibition
31 May 2022	Project competition (for selected projects only)

Table 1: Timeline for this FYP