COMP390

2021/22

An Online Multiplayer Game
Marble Battle Royal

Student Name:	Lizhenghe Chen
Student ID:	201521681
Supervisor Name:	Coope Sebastian

DEPARTMENT OF COMPUTER SCIENCE

University of Liverpool
Liverpool L69 3BX

Acknowledgements

I have experienced and learned a lot during the project's development. I realised that game development is a challenging project, but the process is exciting and rewarding.

I want to thank my parents for giving me sufficient financial and psychological support.

Secondly, I would like to thank Professor Sebastian for giving me this valuable opportunity and for his patience in advising and support. I enjoyed and was happy to have a systematic opportunity to learn about the project development of the game.

In addition, I would like to thank my friends and seniors for their advice and testing feedback: Mr Zhang, Mr Gao, Mr Cui, Mr Yang, Miss Kate and Dr Sebastian.

COMP390

2021/22

An Online Multiplayer Game

Marbles Battle Royal

DEPARTMENT OF
COMPUTER SCIENCE

University of Liverpool
Liverpool L69 3BX

Abstract

Multiplayer games have always been the most popular ways of entertainment, and it allows players to collaborate, compete, make decisions and think while relaxing. Multiplayer games also make remote social interaction more attractive and valuable. This project aims to develop a Wide Area Network 3D Online multiplay Desktop Game: Marbles Battle Royal, using Unity Game Engine and *Photon Network* Engine. The project is based on a low poly style with a single big map for players to have fun with their friends from different locations.

The project will aim to develop a worldwide online game that allows players to control balls to survive and have fun. This dissertation will describe how to implement Each ball roll, jump, sprint, and break and how these let players avoid damage or hits and push other players to die. The game rules let the last survivor till the end be the winner, like the fashion Battle Royal games. It also describes many game designs like Players should choose balls with different attributes, values and outlooks(skins). Moreover, it will also describe how to Implement other valuable tools and mechanisms to make the game more fun and competitive and introduce the solution for making Terrain with different friction coefficients and vegetation for a better game and visual experience. It will also give a brief overview of the background music, sound effects, user interface, and settings.

The project will also explore and implement several technical aspects. Network communication, network coordination and game logic usually encounter technical problems that need to be solved. The game's physics, interface, video quality and performance optimisation are also crucial in the project exploration and implementation.

Table of Contents

ACKNOWLEDGEMENTS **3**

ABSTRACT **6**

LIST OF FIGURES **9**

APPENDICES **10**

1 INTRODUCTION **11**

1.1 GAME DEMAND PROBLEM 11
1.2 AIMS AND OBJECTIVES 12
1.3 PROJECT RELEVANCE 12
1.4 CHALLENGES 13

2 BACKGROUND **14**

2.1 UNITY GAME ENGINE 14
2.2 *PHOTON NETWORK* ENGINE 16
2.3 TECHNICAL TOOLS 17
2.4 ECONOMIC & LEGAL FEASIBILITY 17

3 DESIGN & REALIZATION **18**

3.1 BASIC PHYSICAL & CONTROL DEVELOPMENT 19
3.1.1 Camera with *Cinemachine* 19
3.1.2 Ball Movement Controller 21
3.1.3 Ball Jump Controller 22
3.1.4 World Environment Building 26
3.1.5 Menu 29
3.1.6 Home Page Menu's Network 32
3.2 CHARACTER 38
3.3 GAME RULES & GAME BALANCE 39
3.3.1 Constraint condition 39
3.3.2 Damage System 39
3.3.3 Competition Rules 42
3.3.4 Recovery Area & Damage area 42
3.4 IN-GAME NETWORK 44
3.4.1 Game Managers 44
3.4.2 Avoid Confusion 45
3.4.3 Synchronising Physics 46
3.4.4 Online Structure 48
3.4.5 Synchronising Player's Data 49
3.5 GAME HUD SYSTEM 50
3.5.1 Player Status Interface 50
3.5.2 Billboard 50
3.5.3 Ranking board 51
3.5.4 Game Events Board 52
3.6 TRAINING GROUND TUTORIAL 53
3.7 NPC DESIGN 54
3.8 OTHER GAME TOOLS 55

4 GAME OPTIMISATION....56

4.1 CODE OPTIMISATION 56
4.2 GRAPHICS OPTIMISATION 58
Texture resolutions....58
Player quality settings....58
Culling....58
LOD Group....59
Vegetation and Particle density....59
Final optimisation results....59

5 TESTING & EVALUATION....60

Single Engine Test....61
Multitask Single Device Test....61
Multi Devices Users' Test....61

6 CONCLUSION....63

7 BCS CRITERIA & SELF-REFLECTION....64

REFERENCES....66

APPENDICES....68

List of Figures

FIGURE 1-1 18 MONTHS AFTER COVID-19, PLAYERS INCREASING RATE STILL MORE THAN 50% [2] 11
FIGURE 2-1UNITY DEFAULT SCRIPT 15
FIGURE 2-2 SKETCH MAP ABOUT P2P AND CLIENT-SERVER [9] 16
FIGURE 2-3 DEVICES LIST 17
FIGURE 3-1 SCRUM FRAMEWORK DESIGN 18
FIGURE 3-2 FREELOOK FOR HUMAN BODY (LEFT) & FOR BALL(RIGHT) 20
FIGURE 3-3 VARIOUS OPTIONS TO ADJUST THE CINIMACHINE CAMERA TO FIT THE CHARACTER 20
FIGURE 3-4 TO LET THE CAMERA INTENTIONAL AVOID/ACROSS SOME SPECIFIC OBJECTS 21
FIGURE 3-5 KEY MOVEMENT COMPONENTS OF THE MOVEMENTCONTROLLER.CS 22
FIGURE 3-6 GROUNDED METHODS 23
FIGURE 3-7 COLLISION METHOD 24
FIGURE 3-8 THE JUMP METHOD'S FINAL EFFECT 24
FIGURE 3-9 THE COMBINATIONS OF RAYCAST AND COLLISION WITH COOLING TIME 25
FIGURE 3-10 MULTIPLE TERRAIN BLOCKS. YELLOW: SAND; BLUE:ICE; GREEN: GRASS 27
FIGURE 3-11: FRICTION COEFFICIENT μ. F IS THE FORCE THAT OVERCOMES THE FRICTION PARALLEL TO THE CONTACT SURFACE, AND N IS THE PRESSURE PERPENDICULAR TO THE CONTACT SURFACE.... 27
FIGURE 3-12 UNITY PHYSIC MATERIAL SETTING PANEL 28
FIGURE 3-13 GAME ENVIRONMENT PHYSIC MATERIAL TABLE; ACCORDING TO THE LAWS OF PHYSICS, THE DYNAMIC FRICTION FORCE IS ALWAYS LESS THAN THE STATIC FRICTION FORCE 28
FIGURE 3-14 LEFT: IN GAME UI BUTTONS OBJECTS. RIGHT: BUTTON PREFAB IN UNITY ASSETS PANEL, EACH OBJECT CAN HAVE DIFFERENT TEXT AND CLICK FUNCTION(OVERWRITE), BUT WILL HAVE SAME SOUND EFFECT AND STYLE(PREFAB CONTROL) 29
FIGURE 3-15 UNITY ANIMATION SYSTEM PANEL 30
FIGURE 3-16 FINAL GAME START-UP MENU 31
FIGURE 3-17 THE PUN WIZARD PANEL IN UNITY 32
FIGURE 3-18 THREE CORE SCRIPTS ATTACHED TO THE MENU CANVAS 33
FIGURE 3-19 PARTIAL START CODE IN NETWORKMANAGER.CS 34
FIGURE 3-20 UNITY INPUT SYSTEM 35
FIGURE 3-21 PART OF CODES LOGIC IN MENUMANAGER.CS 35
FIGURE 3-22 HOME PAGE MENU'S USE CASE DIAGRAM 37
FIGURE 3-23 4 KINDS OF BALLS AND THEIR COMPONENTS 38
FIGURE 3-24 KINETIC ENERGY, M IS THE OBJECT'S MASS, V IS THE CURRENT SPEED 39
FIGURE 3-25 PART OF DAMAGE SYSTEM LOGIC IN COLLISIONDETECT.CS 40
FIGURE 3-26 CODE SEGMENT IN EACH COLLISION DAMAGE OBJECT'S SCRIPT 41
FIGURE 3-27 HEALING AREA'S MAIN LOGIC IN COLLISIONDETECT.CS 42
FIGURE 3-28 DAMAGING AREA'S DAMAGE MECHANISM IN PLAYERMANAGER.CS 43
FIGURE 3-29 DAMAGE ZONE SHOWCASE 43
FIGURE 3-30 SEQUENCE DIAGRAM SHOWS THE MANAGER SCRIPTS' RELATIONSHIP 44
FIGURE 3-31 THE CORE LOGIC OF CONFUSION AVOIDANCE, SHOULD ATTACH IT TO ALL SCRIPTS THAT ARE REPEATED BUT SEPARATE FOR INDIVIDUAL PLAYERS 45
FIGURE 3-32 ATTACH PHOTON SYNC VIEWS TO ANY OBJECTS THAT NEED TO SYNC PHYSIC DATA 46
FIGURE 3-33 IN-GAME LOCAL PLAYER USECASE 48
FIGURE 3-34 CORE USAGE OF PUN RPC 49
FIGURE 3-35 A VISUAL DIAGRAM SHOWING HOW THE GAME SYNCHRONIZES THE PLAYER'S PERSONAL INFORMATION (HEALTH, DEATHS, KILLS), PHYSICAL INFORMATION (POSITION, SPEED), AND REAL-TIME GAME EVENTS 49
FIGURE 3-36 PLAYER HUD SHOWCASE 50
FIGURE 3-37 GAME BILLBOARD SHOWCASE 50

Figure 3-38 Score Board with its class diagram, showcase and sort logic 51
Figure 3-39 Some showcase of Game Events Board 52
Figure 3-40 Control Tips(left corner) and Tutorial(top) 53
Figure 3-41robot's jump coroutine showcase 54
Figure 3-42 robot's find closet target algorithm 55
Figure 4-1 Only objects that are in the line of sight of the player's camera will be rendered 58
Figure 4-2 At LOD 0, closer to the camera, it shows a mesh with many small triangles. At LOD 1, far from the Camera shows the mesh with far fewer triangles [5] 59
Figure 4-3 In part of the performance showcase, The frame rate will be reduced in scenes with dense vegetation 59
Figure 5-1 Players' feedback and responses 62

Appendices

Appendix 1 Script Lifecycle Flowchart. The following diagram summarises the ordering and repetition of event functions during a script's lifetime. 68
Appendix 2 All Class' names with their responsibilities, All in Assets\Scripts folder 69
Appendix **3** A complete list of the game's features 71

1 Introduction

1.1 Game Demand Problem

COVID-19 (started in March 2020) had a notable impact on the way people play video games. More and more people are willing to spend more time on video games with each other due to the pandemic effects [1]. Over half of Americans have played multiplayer games in recent years, and players spending on multiplayer games has increased steadily since at least 2020 [1] [2].

The Battle Royale game category became most known in 2017 with the PUBG Battle Royale Games show up. This multiplayer online video game category merges a survival sport's hunt, explorations, survival, and research components. The player or team needs to survive until the end gameplay is known as Battle Royale Game.

The majority of players have been in pursuit since this kind of game appeared. The COVID-19 pandemic positively impacted the global battle royale game market. Due to the extensive lockdowns, online gamers have witnessed a drastic surge over 2020. The Global Battle Royale Game market increased significantly from 2017 to 2020, and projections are made that the market will rise in the next four years (2021~2025) [3].

Figure 1-1 18 months after COVID-19, players increasing rate still more than 50% [2]

It is evident that the demand for video games, especially multiplayer games, has grown dramatically.

1.2 Aims and Objectives

The project aims to use Unity and third-party Network to implement an online multiplayer battle royale type game in the worldwide network and should be at least able to run on Windows platforms. The project should allow players from different areas to join the game through a central server (basically in continental terms, such as EU, NA, Asia). The game should also have a reasonable user menu interface, including the player's basic information and game mode selection. The player should be able to create or find rooms through the menu.

It should accommodate several or even dozens of players when playing the game. Players should enjoy a comfortable picture, and comfortable hand feel to control the ball when playing. The ball should move physically, such as torque and force. The balls can attack other players, such as collision damage or collision out of the map.

A damage zone will force players to concentrate in one place, just like the PUBG, increasing gameplay and speeding up game progress. Players will be eliminated when they run out of health or fall off the map. While playing, players can collect and use various tools to assist them, thus increasing the gameplay and challenge. The game map should have varied Terrain and environments, such as hills, potholes, jungles, and different surfaces, with different friction coefficients and elasticity. A game ends if and only if the last player survives.

The game should have good graphics, user interface and sound, and provide the player with more information while playing, such as collision, scroll, jump, kill, death and rank.

1.3 Project Relevance

This project aims to develop a low-threshold, real-time multiplayer online game that caters to the needs and convenience of most players. The game should be a low-space (less than 500MB) multiplayer game with low specs (office laptops with low CPU and GPU performance) and beautiful graphics.

The game should have a Small File Size, Fast Start-Up, Low Entry Barrier, Good Gameplay, Video Quality, Sound effects, User Interface, Have Basic Single and Multiplayer Modes (With Lots of space To Expand); it can transplant to mobile devices in the future.

Players are increasingly eager to play more games, as mentioned in 1.1. Therefore, the game should be able to meet the needs of most players to a certain extent, so the implementation of the project is valuable

1.4 Challenges

Game development is a very systematic and challenging area that needs the support of a full range of knowledge and a variety of tools.

The biggest challenge in this project is to design and implement a proper network logic. It includes syncing player health values, health bar, location, player information and master client logic. The game engine will instantiate all players with their components, and reasonable distinctions and arrangements for them are challenging. However, each client only needs to control their character, and if not managed, the game will be a mess.

Secondly is physics development, it includes a camera and balls. Physical performance includes how the camera follows the player, how balls jump and roll, and how to manage collision in online mode.

The following need to pay attention to is the communication between different scripts. Each object may have its script, and various variable passes and method calls may occur between scripts. Also, the logic between objects, scripts, and multiple scripts will be more complex and abstract in online mode. It is very tedious to sort out and manage them.

The final issue is the game performance optimisation. It includes graphic optimisation for GPU: texture resolution, shadow, global illumination, camera visible distance, LOD groups, culling and other operations. Also, the code optimisation for CPU: reduces loops, unnecessary code execution, and changes to more efficient method logic or APIs.

2 Background

The way to develop online games and single-player games are similar; they both need to use various software tools to assist in the development. The most critical part is the game engine. The earliest game development usually realised the graphics and physical logic directly through the code, which resulted in the game's lack of real and pictorial sense. More importantly, this made the code and hardware highly specific to single-game development and could not effectively reuse the game resources and code.

2.1 Unity Game Engine

Game Engines are extensible software that can be used as the foundations for many different games without significant changes. A game engine usually consists of a graphics renderer or render Engine, physics engine, sound Engine, collision detection, scripting and localisation support [4]. Nowadays, there are many successful Game Engines, like Epic Games' Unreal Engine 4, Valve's Source engine, Crytek's CRYENGINE® 3 and Electronic Arts DICE's Frostbite™ engine. The Unity game engine—has become fully featured reusable software development kits that can be licensed and used to build almost any game imaginable [4].

Unity is suitable for cross-platform game development and runtime engine supporting various platforms. The games created by Unity can easily fit on different platforms (like Play Stations, Xbox series, Android, Apple iOS, Windows Linux and Mac OS. Unity also provides powerful tools and an integrated editor environment that lets developers preview, analyse, optimise, and give developers a comprehensive assets management pipeline and resources [5].

Unity supports scripting in javascript, C# or Boo, ideally for object-oriented projects. In addition, it also has a powerful animation system supporting animation retargeting (the ability to play an animation authored for one character on a different character) and support for networked multiplayer games [4].

By default, Unity C# scrips derive from a class named **`MonoBehaviour`**. The **`MonoBehaviour`** is a built-in class that allows the script to work with Unity. There are two default functions inside the Class called **`Update`** and **`Start`**. **`Update`** function updates and goes through the function every frame update of the game object. On the contrary, the **`Start`** function only gets called at the game's start (or when activated). Any code that needs to get called continuously, such as the movement script, should be inside the **`Update`** function [5].

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerScripts : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
    }
    // Update is called once per frame
    void Update()
    {
    }
}
```

Figure 2-1Unity Default script

`Monobehaviour` also includes **`Awake()`**, **`OnEnable()`**, **`FixedUpdate()`** and many other functions that have their own execution order in Unity Lifecycle. It is imperative to arrange them and put proper codes inside these functions [6], the exact order will be shown in Appendix 1.

Java and C# are object-oriented languages heavily influenced by C++; their syntax and features have many similarities. Both languages depend on third party frameworks and reusable components [7].they are reasonable solutions for game development.

Unity provided students with free professional game engine software and resources, so Unity is an ideal tool for this project.

2.2 *Photon Network* Engine

Systems located across LAN or the Internet could run games with each other in peer-to-peer(P2P) client-server(C-S) models [8] [9], but it is also possible to combine two architectures as a hybrid [10].

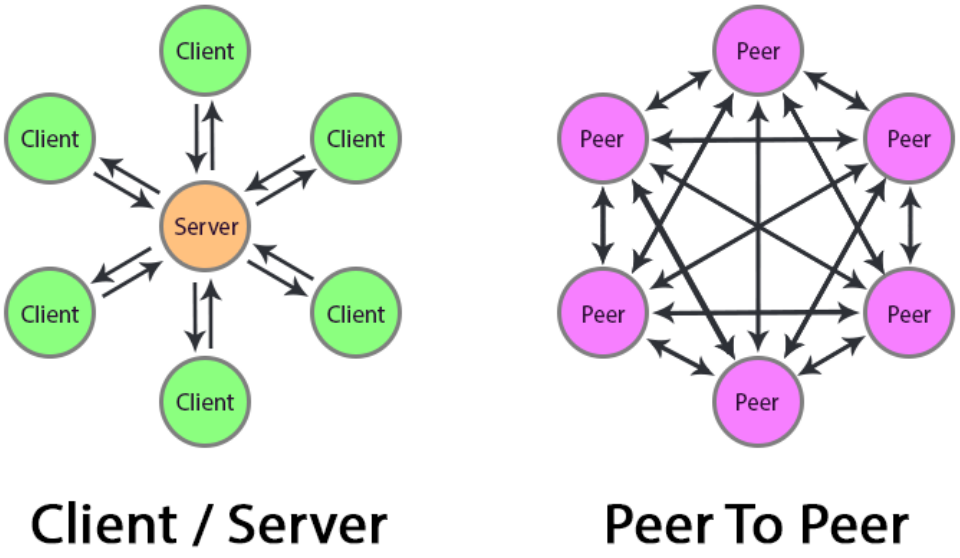


Figure 2-2 Sketch map about P2P and Client-Server [9]

P2P does not need a central server, it is suitable for data distribution (user-created content is dynamically synced), and it is more stable and private (like digital currency) [11]. However, P2P is more complex to implement than the C-S; P2P also hard to prevent game cheating. Game latency is usually much higher; one client connection issue can easily influence the whole server. In general, the P2P technical requirements are higher. C-S is easier to implement. Its advantages include high performance, low latency, easy anti-cheat, and anti-nuisance. However, it needs to implement and maintain a server. However, all players will be influenced when the server has a problem [11]. In this project, C-S will be used as the network communication framework.

With the development of the Internet and the decrease in internet costs, online multiplayer has become very popular. In this Unity project, *Photon Network* PUN SDK will be used to handle the online multiplayer module. It could let developers pay more attention to game development than to building servers. PUN is based on a client to server architecture third party server, tight to Unity integration that can quickly develop and launch multiplayer games worldwide. It supports several useful features that help developers create a complete and stable multiplayer game environment [12].

2.3 Technical Tools

Below software and technologies were used in making Marbles Battle Royale

Development Device: Laptop, CPU i7-9750H, GPU GTX 1660Ti, RAM 16GB

Test Devices (All are laptops):

CPU	GPU	RAM	OS
Intel i7-10750H	RTX 2070	16GB	Windows11
Intel i7-10750H	RTX 2060	16GB	Windows11
Intel i7-9750H	GTX 1660Ti	16GB	Windows11
AMD R7-4700H	GTX 1060Ti	16GB	Windows11
Intel i7-8550U	MX150	8GB	Windows10

Figure 2-3 Devices List

Game Engine: Unity 2021. 2.. F Professional

Network Engine: Unity *Photon Network* 2

Code Editor: Visual Studio Code 1.66.0

Programming Language: C#

3D builder: Blender 3.0

Image and UI: Adobe Photoshop 2022

Partial Game Assets: Unity Asset storedeve

2.4 Economic & Legal Feasibility

A high-performance personal Laptop/Computer will be an ideal development device, and this is the only economic cost during the developing process. The rest of the technical software all has free student support. Almost all game assets are from Unity Asset Store's free assets with standard EULA agreement. The rest of the material was created personal using Blender3D and Photoshop. Except a few codes are from official open-source documents. The rest code and the game design are entirely original.

3 Design & Realization

To enhance the development efficiency and avoid unnecessary waste (like a large number of architecture changes or found fundamental irreversible errors), a proper development process is essential. This project tends to use a scrum framework [13]. Each stage only has a primary or even ambiguous target to achieve. The urgent affair is to achieve each core function at each stage. Players will be able to play each semi-finished Iterative game, give feedback (weekly sprint), and modify, fix, and perfect each core function until moving to the final stage. After all basic and core requirements are accomplished, publish the final version and then updates, fixes and iterations based on user feedback. This dissertation will merge Design and Realisation and follow Figure 3-1, describing each part of the design and implementation process.

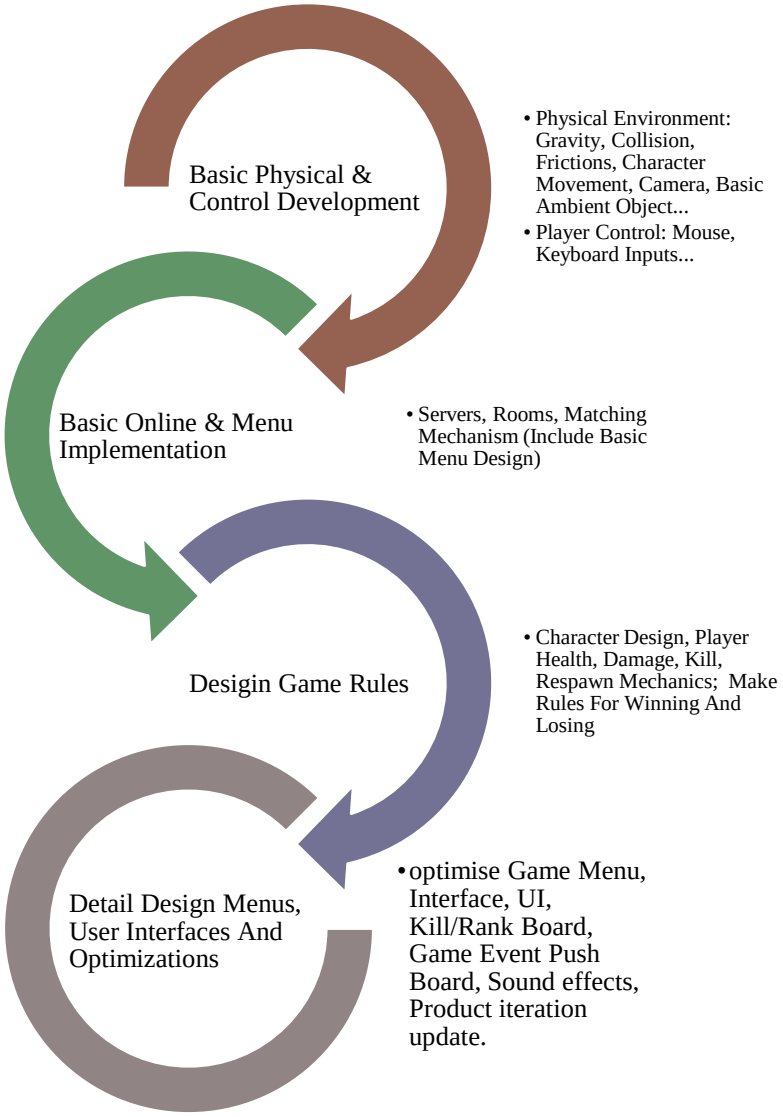


Figure 3-1 scrum framework design

3.1 Basic Physical & Control Development

Physics is one of the core parts of this project, and it is directly linked to gameplay and game manoeuvrability. Since it is at the beginning, no more elements should be added or designed. The main goal is to implement the ball controller and the camera behaviour. Create, test, and adjust the ground, terrains, and vegetation, making sure the gravity, fiction, and other physical effects are well and playable.

3.1.1 Camera with *Cinemachine*

Camera control is an essential component of player experience. The camera viewpoint defines the amount of information shown to the players. It also directly impacts the perceived challenge and aspects of the experience, such as player frustration [14].

The camera should follow the ball's rolling and jump in this game, but not that easy. The ball also needs to roll according to the camera's direction, which the player controls. So finally need a camera that can follow the ball, controlled by the player to look around, avoid the camera crossing the objects and have a comfortable movement effect. These goals are not easy to accomplish through personal coding since they involve many physical and mathematical calculations. The customised code will limit the movement of the camera and even the ball, which is not conducive to later adjustment and innovation. Therefore, after trying to write some codes to control the camera's motion and the sphere and finding it unsatisfactory, a more efficient solution: *Cinemachine*.

Cinemachine is a suite of modules for operating the Unity camera. *Cinemachine* solves the complex mathematics and logic of tracking targets and gives excellent solutions for many other physical camera demands like composing a switching. It could significantly reduce the number of time-consuming manual manipulations and script revisions during development. It works in real-time across all genres, including FPS, third person, 2D and other game modes [15].

To set up the *Cinemachine*, use Unity's package manager and install the *Cinemachine* package. Add the needed camera type and bind the character or object want to track to the component. *Cinemachine* has four primary purposes and behaviours: composer, transposer, free-look and 2d-specific features. Freelook will be the camera movement logic in this game development; it provides a third-person camera experience. This *Cinemachine* Virtual Camera provided three-camera orbits(rigs) around its subject: Top, Middle, and Bottom [15], as shown in Figure 3-2, the camera will move (controlled by the user or scripts) around within these orbits; these orbits will always follow the players' character, with various adjustable and expandable motion modes and parameters.

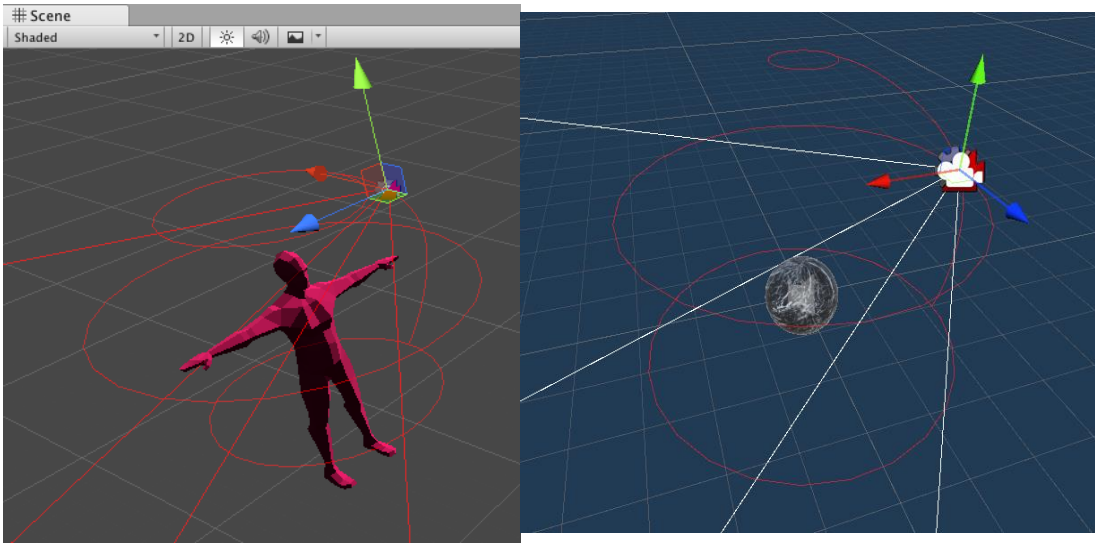
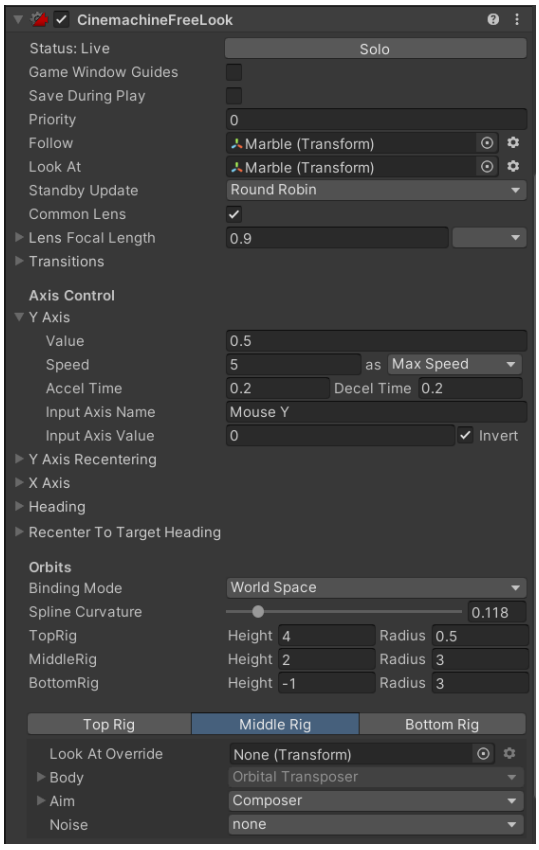


Figure 3-2 freelook for Human Body (Left) & for Ball(Right)



Make sure the *Cinemachine* camera follows and looks at the ball by assigning the ball's transform to the options panel (Figure 3-3). The *Cinemachine* will overwrite the Unity built-in camera settings to set all camera settings in this panel. **`X Axis`** and **`Y Axis`** movement can be adjusted; **`Accel Time`** will let the camera move with some buffer, making the video more smoothly. More options in the **`Orbits`** setting can adjust the size and position of each Rig based on the size and motion of the ball. **`Spline Curvature`** could change how smoothly the camera travels between each Rig. Each Rig and its movement can be set differently, such as switching the maximum viewing Angle and camera rotation range in each Rig's **`Body`** and **`Aim`** options (Figure 3-3).

Figure 3-3 Various options to adjust the Cinimachine camera to fit the character

To let the camera avoid obstacles and go through specific objects, add a *Cinemachine Collider* to the camera (Figure 3-4) and change the target objects' tag or layers to let the collider recognise them and act collider [15].

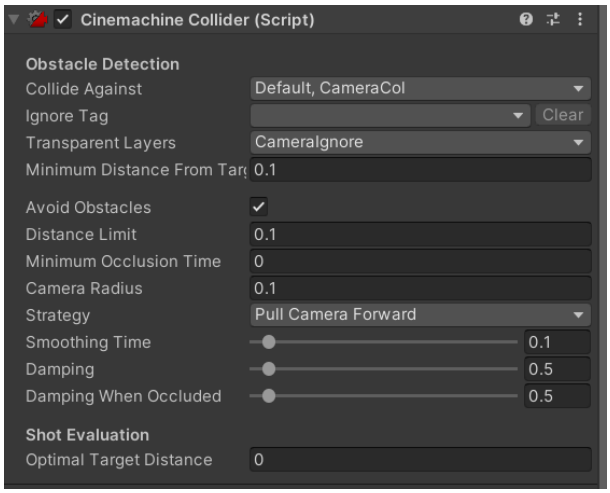


Figure 3-4 To let the camera intentional avoid/across some specific objects

3.1.2 Ball Movement Controller

A **Movementcontroller** script will be attached to the ball objects. The core job in this script is to read the player's keyboard and mouse input to control the basic ball rolling appearance. The main point of this part is to introduce how to make the ball roll with physical characteristics.

The general solution to make a ball move is to use Unity's `Rigidbody. Addforce (…)` API to make the ball's centre of mass pushed by a direct force. This method cannot make the ball roll spontaneously and make the whole movement look like an external force pushing the sphere without any natural or physical effect. The better solution is to let the sphere have the torque to roll by itself, driven by the contact surface friction between the ball and the touch surface. This way can make the ball roll like a car tyre in real life that will have its torsion and grip, which let the ball move more real and silky. It can also have the nature of the skid and elegant could improve the game experience to some extent. Unity provided an API that can change objects' torque called `Rigidbody.AddTorque(Vector3, ForceMode ). Vector3 torque` is the force direction; here will be set to the camera's direction so the ball's rotation will follow the camera changes. Changing the ball's maximum angular velocity (`float MaxangularVelocity`) can make the ball switch between two max speed and torque forces when the player inputs speed-up commands.

The control codes should be inside the `FixedUpdate` rather than the *Update* function. All physics calculations and updates occur immediately after `FixedUpdate`; `FixedUpdate` is called on a reliable timer, independent of the frame rate [5].

```
public class MovementController : MonoBehaviour
{
   // The code shown here is partially and does not represent the final runnable code
   private const float torque_multiplier = 3;// speed up multiplier
   private void Start()
         {       rb = GetComponent<Rigidbody>();//get ball's rigibody compoment
                 speedUp_torque = initial_torque * torque_timer;// precalculated player speed-up speed
                 Camera = transform.Find("ThirdPersonCamera/MainCamera");// get player's camera
        }
   private void FixedUpdate()
        {
         horizontalInput = Input.GetAxis("Horizontal"); //keyboard A & D
         verticalInput = Input.GetAxis("Vertical"); //keyboard W & S
         rb.maxAngularVelocity = (Input.GetKey(KeyCode.LeftShift) ? speedUp_torque : initial_torque);
         rb.AddTorque(Camera.transform.right * rb.maxAngularVelocity * verticalInput);
         rb.AddTorque(-Camera.transform.forward * rb.maxAngularVelocity * horizontalInput);
        }
```

Figure 3-5 key movement components of the MovementController.cs

3.1.3 Ball Jump Controller

A **JumpController** script will also be attached to the ball objects. One of the core jobs in this script is to read the player's keyboard input to control the basic ball Jumping appearance. The main point of this part is to discuss the appropriate jumping method for the ball.

First, change the gravity in Unity's global physics settings from 9.8m /s^2 to an immense value such as 15m /s^2; this will allow the ball to fall faster and avoid the feeling of weightlessness and slowness and with no extra physical calculations to simulate the jumps and falls of the spheres. Then similar way as 3.1.2, the script will continuously monitor the player's space keyboard input once the user presses the space bar, **`Rigidbody.AddForce(Vector3.up * jumpforce)`**; this API will give the sphere a global vertical upward (Vector3.up) force to simulate the jump appearance.

3.1.3.1 When to Jump?

Generally, only a grounded ball can jump; otherwise, the player can keep jumping until outside the map. This will involve determining when the sphere is on the ground. The mainstream solution has two, one is to use the collision detection, and the other is to use the ray cast. In order to make the principle more intuitive, a simple schematic Figure 3-6 will assist the department's introduction.

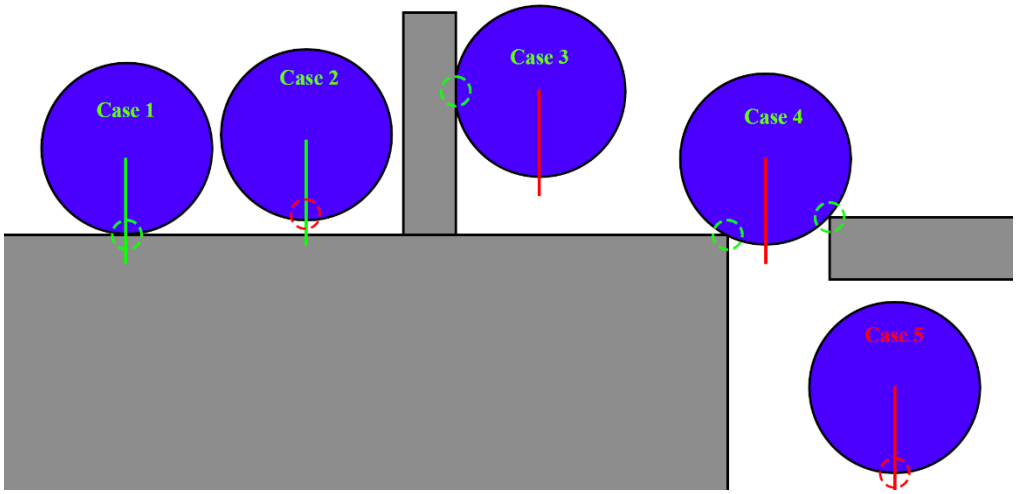


Figure 3-6 grounded methods

In the collision method, two Unity collision APIs will be used to detect the collision: `Collider.OnCollisionEnter(Collision)` and `Collider.OnCollisionExit(Collision)`; the principle is to set a Boolean variable `onCollisionGrounded`. When the ball touches the object, set this variable as true; when the ball leaves that object, set it to false, as shown in Figure 3-7.

This method is shown as dotted circles in Figure 3-6. The green circle represents true, which means landing, and the red represents false, which means not landing. This method can effectively play a role in absolutely flat ground. However, the physics engine will determine that grounded is false when the surface is not flat or when the ball is slightly out of the ground with even a minimal distance; this leads to the players running on the flat ground cannot jump in time as expected, as shown in Case 2 in Figure 3-6.

In the actual user experience feedback, when the player has multiple collision points in complex Terrain, there is a chance that this method will fail. Changing **`OnCollisionEnter`** to **`OnCollisionStay`** could avoid this situation but still cannot avoid the issue above.

```
void OnCollisionEnter(Collision col)
    {
        OnCollisionGrounded = true;
    }
    void OnCollisionExit(Collision col)
    {
        OnCollisionGrounded = false;
    }
```

Figure 3-7 collision method

Better results will be achieved with the ray cast method using Unity API **`Physics.Raycast`** sends a distant ray straight down from the geometric centre of gravity of the player's sphere. If the ray hits an object within that distance, it will return true; otherwise, it returns false, as shown by the vertical line in Figure 3-6. The benefit of this is that the ray can exceed the ball's radius (**`jumpThreshold`**) so that the ray still can hit the ground even when the ball is above the ground a little. However, there is another problem with this method. When the player wants to jump from the object's edge, although the sphere touches the ground, the ray is emitted from the sphere's centre, so it cannot be vertically projected onto the ground, as shown in Figure 3-6 Case4. As a result, players cannot jump at the edge of the Terrain or when players are above the gaps.

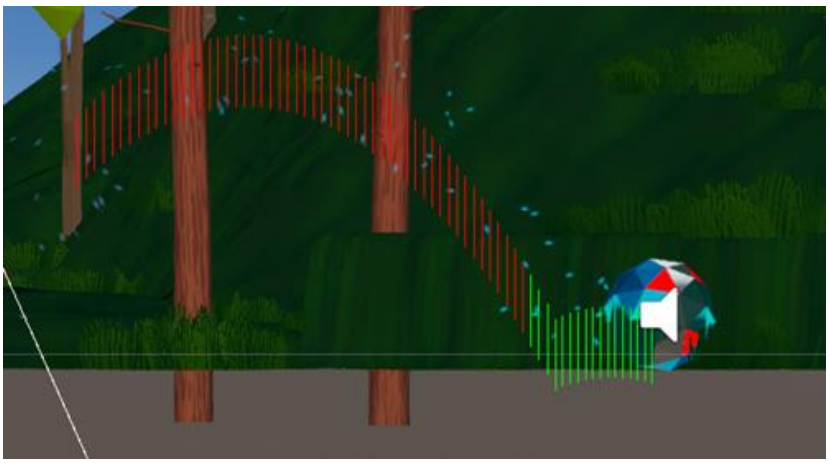


Figure 3-8 The jump method's final effect

Therefore, choosing to judge by combining the above two schemes can shove the problem. If the ray cast returns true, an object must be under the sphere. Collision detection is used when radiography returns false, which makes up for what ray cast misses. If either method returns true, the ball must touch the ground, and if both methods return false, the ball must not jump, Figure 3-9.

However, there are still problems with this method. When the player encounters vertical Terrain and trees, collision detection returns true. The player can jump continuously to jump off walls or climb trees, as shown in Case3. The solution is to increase the cooling time of the player's jumps. It lets the player wait a certain amount of time after each jump before jumping again; the actual effect is shown in Figure 3-8.

```
public bool onTheGround()//a combination of touch and raycast
    {
        bool isRayGrounded = Physics.Raycast(transform.position, Vector3.down,
jumpThreshold);
        if (isRayGrounded ||( !isRayGrounded && OnCollisionGrounded))
        {
            combinedGrounded = true;
            //Debug.Log("i'm grounded");
        }
        else
        {
            combinedGrounded = false;
            //Debug.Log("not grounded");
        }
        return combinedGrounded;
    }
    void JumpMethod()//put this method in FixedUpdate to accumulate time
    {
        if (JumpTime >= JumpcoolingTime)
        {
            JumpCommand();//allow the user to jump
            return;
        }
        JumpTime += Time.deltaTime;
    }
```

Figure 3-9 the combinations of raycast and collision with cooling time

3.1.4 World Environment Building

This step will only systematically confirm the physical environment of the game and develop and test the map at a superficial level. After the network and game rules are finalised and stabilised, the Terrain's vegetation and objects on the map will be systematically designed.

Unity provides various tools that let developers create environmental features such as landforms and vegetation on Unity's built-in Terrain features [5]. Unity terrain editor consists of tools that let developers create and modify landscapes in the Unity Editor. Developers can use the paintbrush tool to draw heights, hills, and potholes on the surface and "plant" various vegetation such as trees and flowers. It also allows for painting the surface with various materials to present the various styles of colours. The map editor also provides several options to adjust the map's parameters, which will be discussed in section 0. This section describes how to make maps with multiple coefficients of Friction and elasticity.

Generally, there are two ways. The first one is to abandon the map editor to set up the ground with the entity object. Such benefits can be for more diverse Terrain. Each object map of each object can be given a *Physical Material* (including the Friction and elastic). However, it cannot be a fast batch of topographic map editors for drawing and change. Vegetation cannot be added or deleted in a large area. It is not convenient for later modification and performance optimisation.

Using the map editor directly is more efficient, which is the solution for this project. However, because every Terrain in Unity is fixed and rectangular, *Physical Material*s can only be attached to a single object. If only one Terrain is used to draw the Terrain, the richness and beauty will be far less than the first method. The solution is to "compress" the size of each Terrain to a smaller square under the original resolution and use multiple Terrain blocks to form a game map. Each Terrain can be individually edited and endowed with its physical properties. At the same time, some particular Terrain objects can be manually added under Terrain as in the first scheme, and then Terrain is rich and complete.

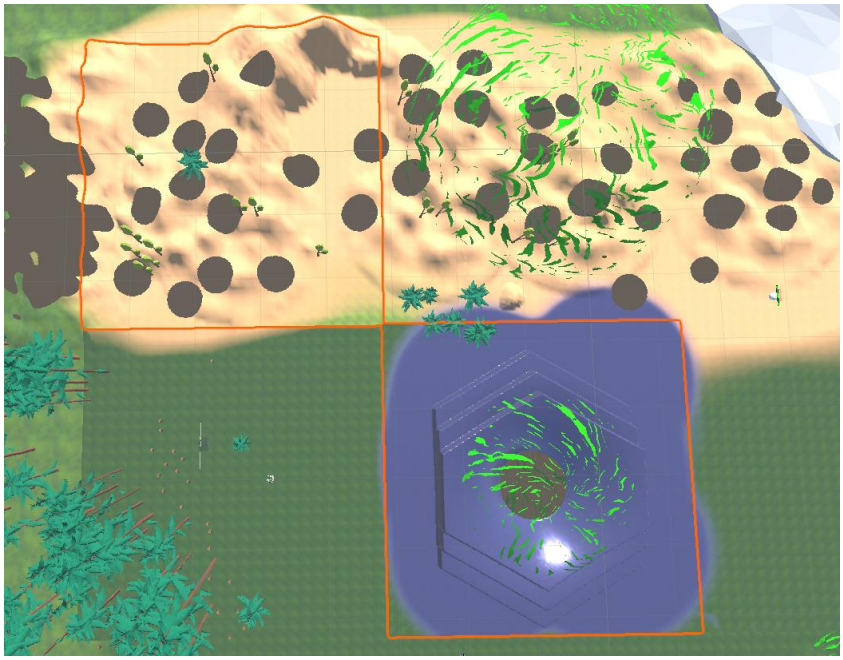


Figure 3-10 multiple Terrain blocks. Yellow: Sand; Blue:Ice; Green: Grass

Unity provides Physic Material that gives each object its physical surface. The surface properties include Friction and Bounciness, as defined by Unity's Physic Material [5]. Friction comes in two forms, dynamic and static. Static Friction (Value: 0~1) is used when the object is lying still. It will prevent the object from starting to move. If a large enough force is applied to the object, it will start moving. *Dynamic Friction* (Value: 0~1) will come into play [5], *Dynamic Friction* will now attempt to slow down the object while in contact with another, like when the ball is slipping. Both frictional coefficients μ should be less than one due to the friction formulation:

$$\mu = \frac{F}{N}$$

Figure 3-11: Friction Coefficient μ. F is the force that overcomes the friction parallel to the contact surface, and N is the pressure perpendicular to the contact surface.

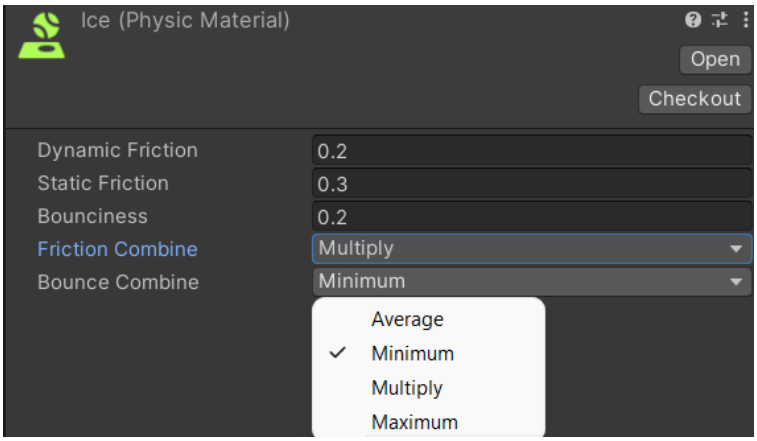


Figure 3-12 Unity Physic Material setting panel

According to the chosen model, the same Bounciness and friction effect are applied when two bodies are in contact. There is a case when the two colliders in contact have different combined modes. In this particular case, the highest priority function is used. The priority order is as follows: Average < Minimum < Multiply < Maximum [5]. For example, if one material has an Average set, but the other has a Maximum, the combined function is Maximum since it has higher priority.

The surface type	**Static friction**	***Dynamic Friction***	**Bounciness**
Sand	1	0.8	0.2
Ice	0.3	0.2	0.2
Grass	0.7	0.5	0.2
Ball	1	1	1

Figure 3-13 Game Environment Physic Material Table; According to the laws of physics, the dynamic friction force is always less than the static friction force

Balls' friction and Bounciness will be set to 1 to simulate the utterly elastic collision and for the best ball collision.

3.1.5 Menu

The ease of use of a game's controls and the interface is closely related to fun ratings for that game [16]. The player's initial experience in the game is critical to their continuation to play. Usability tests can ensure that new users (and more advanced users) can quickly and effectively start a game from the game's menu system [17].

To design a beautiful user interface, text and buttons and their style will be essential. All in the game text will use the *TextMeshPro* package since it includes more content and solutions, like font styles [5]. Buttons design is indispensable to using *Unity Prefab System*. Unity's Prefab system allows developers to create, configure, and store a game object complete with its components, property values, and child game objects as a reusable Asset [5].

A game UI usually have a uniform style to make the interface more elegant. Hence, uniform buttons and text style are essential. The prefab system can solve this issue that the developer only needs to modify the original template. Like the template, all game objects will be modified together following the template. Prefab objects can also be overwritten if the developer needs some Prefab instances to differ from others [5].

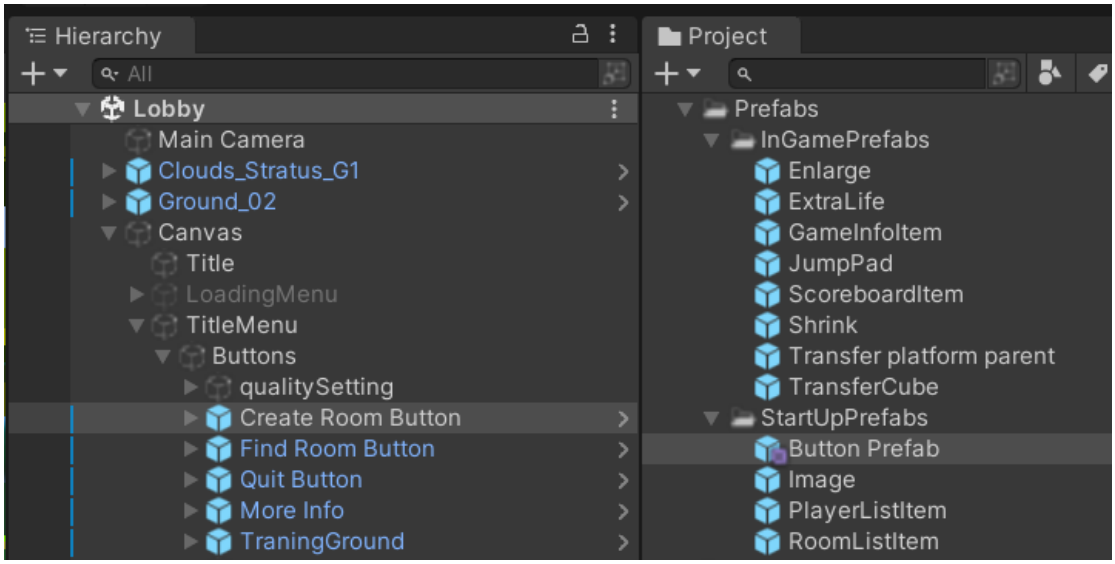


Figure 3-14 Left: in game UI buttons Objects. Right: button prefab in Unity assets panel, each object can have different text and click function(overwrite), but will have same sound effect and style(prefab control)

After the game start and ends with splash scenes, the camera will begin having a cinematic dark to bright and near to far animation, giving players a feeling of soft transition. Then the home page menu will show up ultimately. In this game, almost all lens switching will have a smooth transition from dark to light, making the frames less abrupt and smoother during switching. These cinematic lenses were animated by the *Unity Animation System* and saved as a Prefab that can be reused for multiple cameras and objects.

The game menu background will cover the game's primary content as much as possible, such as damage and recovery areas, vegetation and Terrain, as shown in Figure 3-16. The game's title will gradually be presented at the top to the player. Several balls will roll over the hill, which will partly reflect the theme and character of the game.

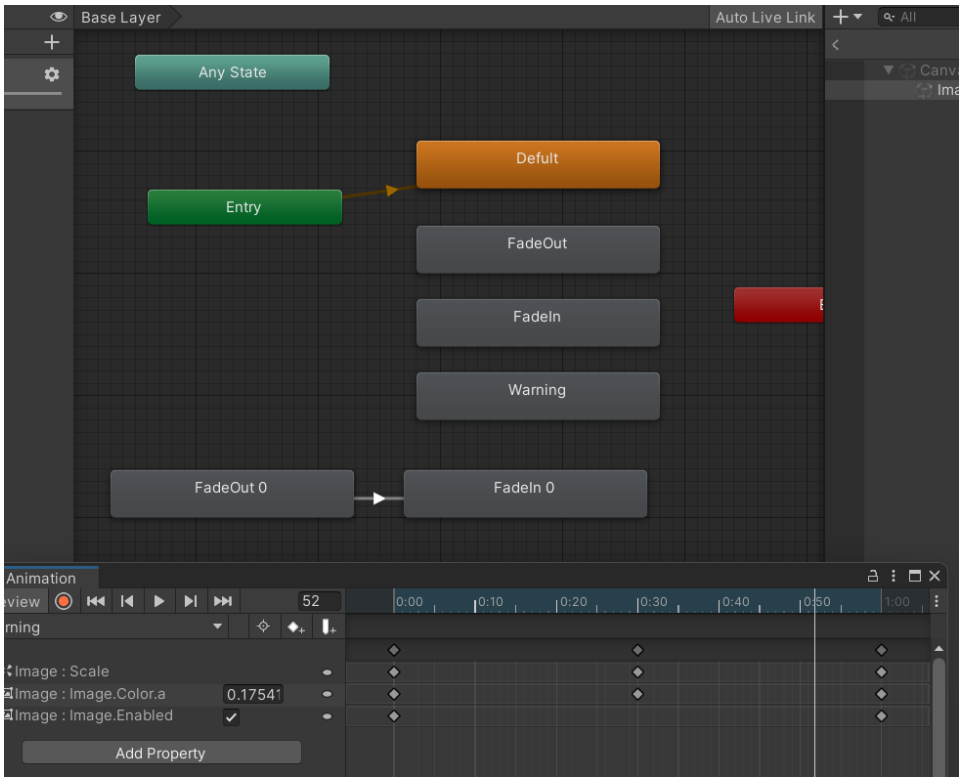


Figure 3-15 Unity Animation System Panel

The most frequently clicked buttons will be placed in the centre of the screen, including finding and creating rooms, selecting characters and changing names. The text at the screen's bottom will tell the player what they should do or what has not been done yet. For example, suppose the player does not select a character or enters the wrong name format. In that case, the text will flash and turn red to prompt the player to change the name or select the character. Other menu buttons will not be used until the player completes the correct selections.

After the player selects a ball, the character's name will be displayed at the bottom of the screen. The character's unique attributes will be displayed on the right. The development of characters' attributes will be shown at 3.2.

In addition, there will use some text areas in various parts of the menu to give players hints. There will be a game setting panel on the left corner, including video quality settings, background music settings and training ground options. Some text background is a grey rock rather than a pure colour UI background, making the UI more natural.

Although the final menu is rich in content and functions, it may be too messy for some players. Some users said that the menu's text might be too much. It may be because of the confusion of the red area pattern in the background, so this object in the menu is to be considered removed.

Figure 3-16 Final Game Start-Up Menu

3.1.6 Home Page Menu's Network

After the game's physical aspect is almost resigned, the development processes will move to the network deployment. This step is synchronised with menu development. It involves creating and finding rooms and connecting to the server when the game starts. Set up *Photon Network* will need two steps [6]:

1. Get Photon Unity Networking (PUN) APP ID:

It is straightforward to get a free App ID from Photon Network's official website. After registering an account, Photon provides a free version of PUN products with limited players but full functions. This App Id will be used to set up the unity project.

2. Set up the PUN network in Unity:

Install the PUN package in a Unity project. The PUN 2 version is in Unity's official Assets Store; fill the App ID into the wizard panel. The network set-up is complete and then can start developing the network logic.

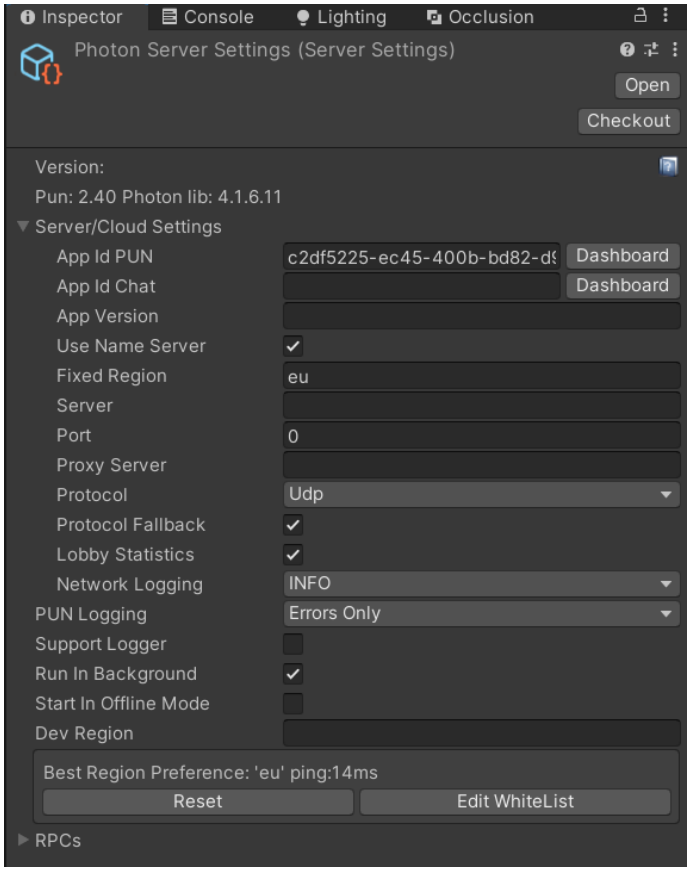


Figure 3-17 the PUN wizard panel in Unity

A **NetworkManager** script will work in the home page scene. This script will handle all network management and buttons' functions for the home page and combine them with the **MenuManager** script. Besides, the **RoomManager** script will control the player's game scene switching (Figure 3-18).

Each object that needs a PUN network function for the Photon Network must include the Photon network's **PhotonView** script. A **PhotonView** identifies an object across the network (**viewID**) and configures how the controlling client updates remote instances [6].

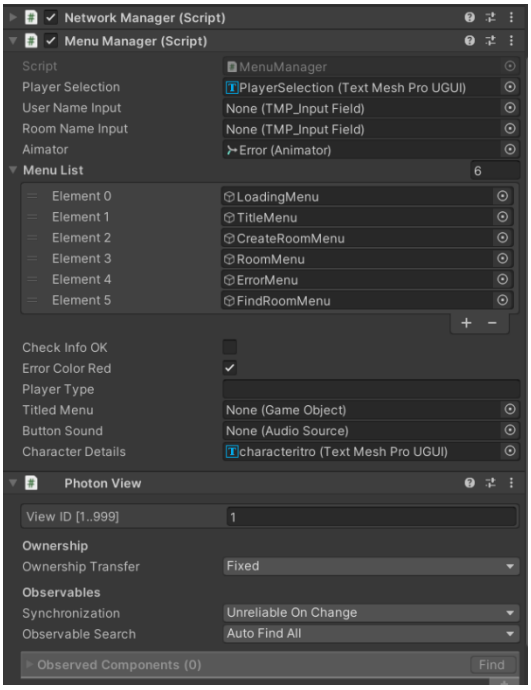


Figure 3-18 three core scripts attached to the Menu Canvas

All menu pages will be game object type and attached to MenuManager.cs. A `gamepbject[] menuList` will store the menu page list and prepare it later.

`PhotonNetwork.ConnectUsingSettings()` at the start will connect to the Photon server as configured in the wizard panel in Figure 3-17.

Once the game is connected to the server, **`OnConnectedToMaster()`** will be called: Figure 3-19, which means the client is connected to the Master Server and ready for matchmaking and other tasks; overwrite this function so it can do something once the game is connected to the server, here will set to open the Title menu (homepage), also give the player a random Nickname, then bring the player to the lobby(**`PhotonNetwork.joinLobby()`**) here I set all players to force connecting to the European server, PUN can also automatically choose the best server for a player, but then players from the different region cannot play together.

After joining the lobby, the code will wait for the player to choose their character and game mode.

```
void Start()
    {
        SetMenu();
        PhotonNetwork.ConnectUsingSettings();//connect to the PhotonNetwork as in wizard
settings
    }
    private void SetMenu()// Instantiate menu
    {
        MenuManager = this.GetComponent<MenuManager>();
        roomManager = GameObject.Find("RoomManager").GetComponent<RoomManager>();
        /* Instantiate menus according to MenuManager.menuList */
        MenuManager.OpenMenu(LoadingMenu);//give a loading menu before title menu
    }
   public override void OnConnectedToMaster()
    {
        /* Open Title Menu, set player's name and join to lobby */
    }
```

Figure 3-19 partial start code in NetworkManager.cs

To open a menu, iterate the **`menuList`**, only active the target menu, and set the rest as inactive. This can avoid open menu mistakes and is easy for code testing and menu management: Figure 3-21.

To avoid any user's input error, handling the invalid input for players' names and player types is necessary. Unity provided some helpful input limitations, as shown below (Figure 3-20); this could save a lot of code work. The only thing is to handle null input, and **`string.isNullorempty (Check variable)`** can check whether the input string is null or not.

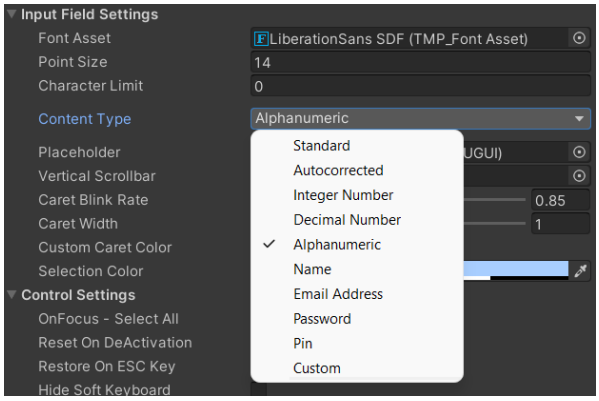


Figure 3-20 Unity Input system

```
public void OpenMenu(GameObject menuName)
  {
      CheckInfo();//check if the info (Nickname, character selection) are correct
      if (checkInfoNotOK)
      {return; // if it is incorrect, player cannot open other menu}
    // refresh the whole menu list, close all the menus except the one wants to open:
      foreach (GameObject menu in menuList)
      {
          if (menu == menuName) { menu.SetActive(true); }
          else menu.SetActive(false);
      }
  }
public void CheckInfo()
  {
     /*check weather player's name and character chosen are empty, the input type
     can be set in unity editor to avoid special characters, numbers or Uppercase …
     if false:
     set error text and do something; else: set player's properties and do next*/
  }
```

Figure 3-21 part of codes logic in MenuManager.cs

3.1.6.1 Ball Selection

Put the `Raycast` function into the user's mouse cursor; when the player clicks on the 3D ball objects in the Title menu, the ray will hit the ball and return the ball's name. The name will be saved as player selection sting variable `playerType` and wait to be used to instantiate the ball later.

3.1.6.2 Find Room

When the player enters the Find Room Menu, all available rooms will be listed in the layout. Each room will have a button object and a `RoomListItem` Class to save the room's name and other details. This Class will also contain a function that, once the player clicks the button, will call the `JoinRoom()` function from the **NetworkManager.cs** and `PhotonNetwork.JoinRoom(roomInfo.name)` will bring the player to the target room. A Room menu will show all players' names in the current room, then wait for the master client to start the game. All public room information will be saved as a dictionary: `Dictionary<string, roominfo> cachedRoomList`: <Room Name, Room Details >.

3.1.6.3 Create Room

After the user clicks the "Create Room" button on the home page (Title Menu), the player will be asked to set a room name and enter the Room menu waiting for other players to join. The room menu is the same as the Find Room one.
The player who creates the room will be the master client; only the master client can start the game. If the master leaves the room, the master privilege will automatically be switched to another player in the same room. The left player can join back to this room, but once the game start, nobody can create, join or find this room again until the current room's game ends.

3.1.6.4 Training Ground

Same as Create Room, the difference is that all players will be able to create or join this room. All players have the right to start the game. There will always be a single room called Training Ground to let players join. More information about the Training Ground will be able in 3.5.

A simple Use Case Diagram (Figure 3-22) will show the comprehensive user actions in the Homepage Menu. Set **NetworkManager.cs**, **RoomManager**.cs and **PlayerManager**.cs as Actors since they will manage and gather the game information. Each player will have these processes locally; only a few data (Nickname, Room information) will be transferred to Photon Server. The **PlayerManager** is generated and run by **RoomManager** only after the player clicks to start the game. Three managers (

Appendix 2)will be locally owned by each user and will upload the user's networking activities to the server.

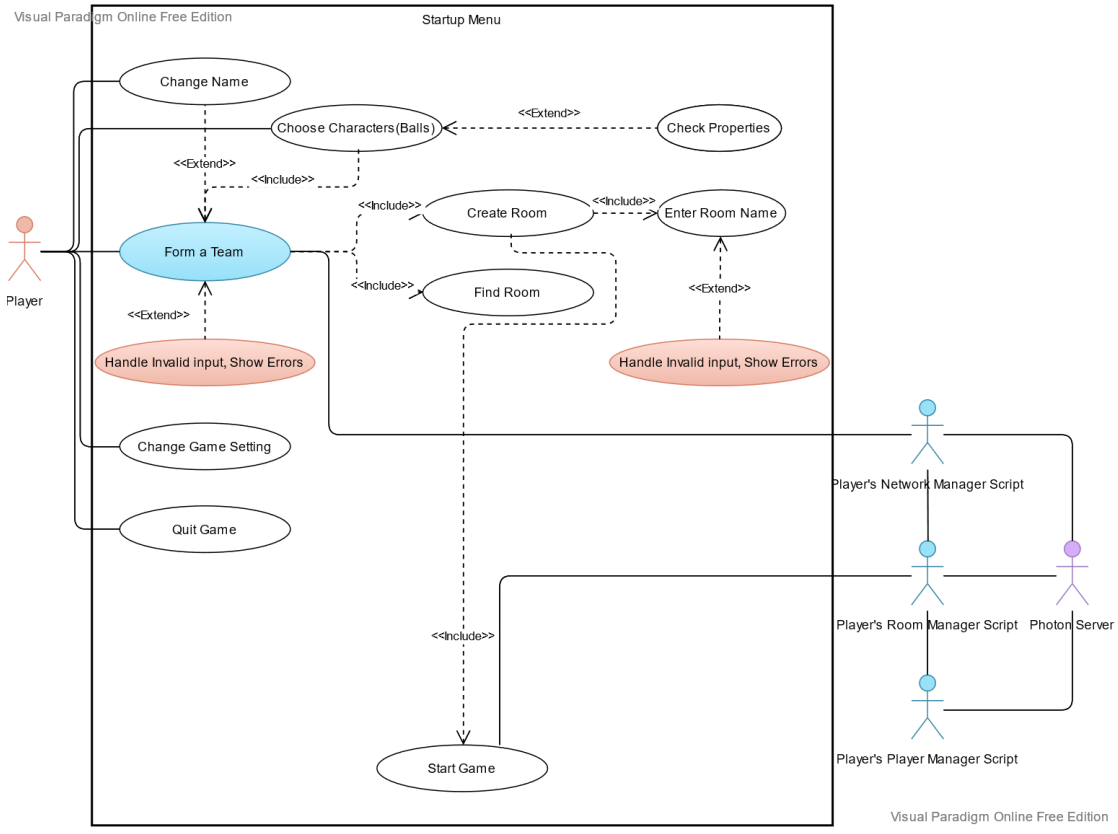


Figure 3-22 Home Page menu's Use Case Diagram

3.2 Character

The game was designed with various character types, which could make the game more fun and more fruitful. The game currently has four balls for the player to choose from while browsing the main menu. The character type will be saved in the `playerType` variable and instantiated by the **PlayerManager**.cs once the game starts. Each ball has its extraordinary appearance and particle system. At the same time, each sphere is a Unity Prefab containing all of the player's functions. These prefabs contain different spheres and appearances. It also contains the menu in the game, the game HUD, sound effects and its controller, camera, bulletin board, manipulation, and other attributes. Prefab leads to all spheres being customised for their appearance, physics and even sound and interface. A high degree of customisation allows each sphere character to have distinctive characteristics. Most importantly, each sphere's movement speed, jumping ability, mass, defence, and damage can be defined separately, and these sphere attributes can be easily read by the start menu code and displayed in the lower right corner of the home page, as shown in Figure 3-16

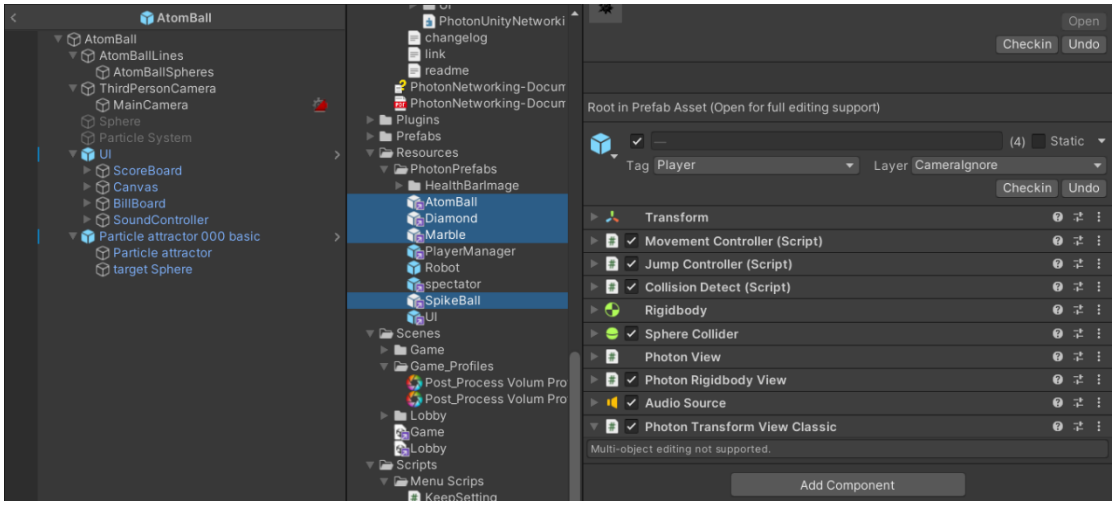


Figure 3-23 4 kinds of balls and their components

3.3 Game Rules & Game Balance

3.3.1 Constraint condition

First of all, there should be a limit on the number of players in each game. Due to the free Photon service usage and limitation, there should be a maximum of ten players in each game, and the players' names should not be duplicated.

When a player successfully joins a room, use `PhotonNetwork.PlayerList` can get an Array that includes all Players' information and use `players. Length > Limitation` to judge if the current room has reached the maximum limit after the current player joined; Use iterator to compare each player's name with the current player. They judge if more than one player has the same name as the current one. Suppose the current player does not satisfy one of these two conditions. In that case, the current player will be kicked out of the room immediately and push the error message to the error menu. All these judgements are included in **NetworkManager.cs**. In addition, the player must select a ball and a non-empty nickname before the player the online mode.

3.3.2 Damage System

Once the game starts, players will be randomly generated in the starting area of the map. Each player will be given 100 health units. When players collide, the faster one will damage the other one. The damage detection and calculation are in **CollisionDetect**.cs. The final damage value will transfer to the player's **PlayerManager**.cs and be sent to the game server. The amount of damage will be determined by the relative speed of the two spheres and the attacker's mass, which can be understood as Kinetic Energy (Figure 3-24). At the same time, *v* is the current ball's velocity, and *m* is the ball's mass.

$$E_k = \frac{1}{2}mv^2$$

Figure 3-24 Kinetic Energy, m is the object's mass, v is the current speed

Moreover, since the ball's physical attributes are in 3.1.4, the collision is elastic, so the collision of the sphere follows the law of conservation of kinetic energy and the law of conservation of momentum. Fortunately, the Unity engine has taken these into its physics engine. The collision performance between the spheres is already reasonable. In order to reduce the calculation burden and enormous damage of the square calculation, the game will use the momentum formula $p=mv$ to represent the damage. The velocity here will use the **`collision. RelativeVelocity`** to the unity engine when the collision happens; this variable is the relative linear velocity of the two colliding objects [5].

The final collision damage value will be calculated as follows:

```
void OnCollisionEnter(Collision collision)
  {
      if (!photonView.IsMine) // this will be introduce in 3.4
      {
          return;
      }
      if (collision with a player) {
           //true means player get make damage from another player:
           If (Player_Velocity < other_Player_Velocity) {
           // p = mv:
           hitForce = relativeVelocity * currentPlayer's mass
                 If (current Player's health- hitForce <=0) {
                 Processed Player Death operation in playerManager.cs.
                       }
                 else {
                 set player's new health in playerManager.cs.
                       }
           }
       }
  }
```

Figure 3-25 part of damage system logic in CollisionDetect.cs

When acquiring the initial speed of the Collision objects, it is worth mentioning if they directly use **`OnCollisionEnter (Collision Collision)`** when the collision happens. The accessing speed is the speed **after** the collision. A typical phenomenon is that if the speed of the colliding object is 0 before the collision, the speed obtained after the collision is not 0, instead is the speed of the next Engine's frame after the collision. It will have a severe impact on collision judgment! Unity does not specify this issue in the documentation. However, it probably has something to do with Unity's cycle order Appendix 1. The physical collision (**`Internal physics update`**) appears to have occurred before the **`OnCollisionEnter`** interface, causing the object velocity obtained through the **`OnCollisionEnter`** to be later than the actual moment before the collision.

After much trial and error, a solution to append a piece of code to every object that might need to calculate the collision damage. This code would have a public variable **`Player_Velocity`** to record the object's real-time speed. Since **`Update`** is later than **`OnCollisionEnter ()`**, Place the code in the **`Update`** loop to ensure that the data is not flushed until after the object is hit, keeping the speed before the collision. It is also possible to put it in a **`FixedUpdate`** loop, where the **`FixedUpdate`** happens the earliest, so the initial velocity can be prepared before the collision. Either way can get the velocity of the object at the moment before the collision:

```
void FixedUpdate()//or Update()
    {
        Player_Velocity = rb.velocity;
//since the collision will get the late velocity that give
wrong damage, so we need to get the velocity in fixed
update (earlier)
    }
```

Figure 3-26 code segment in each collision damage object's script

3.3.3 Competition Rules

Each player will have five chances to respawn in a game; if players fall out of the map or run out of health, they will lose one chance. Once a player loses all five chances, they will be eliminated and no longer available to control the ball. The ball will be deleted, and the player will turn into Spectator mode. In Spectator mode, players can shuttle around the game map by flying a drone to watch the game, enjoying the scenery, adjusting camera parameters, and checking the ranking and game events. The entire game will end when only the last player is left. The winner will be announced, and then all players will automatically return to the main menu. Players will be able to view real-time events and rankings throughout the whole game process and can quit at any time

3.3.4 Recovery Area & Damage area

The game has green sphere recovery Areas and a red Damage wall. Players in spherical areas can gradually restore their health value until fully charged. The recovery Area generally moves or appears in the danger zone (has holes or NPC enemies). When players want to recover health, it can force them to more challenging areas.

The recovery area works by invoking Unity API **`OnTriggerEnter`** and **`OnTriggerExit`**. The healing area is a 3D sphere object. Set the collider as a trigger so the sphere will not produce collision but only detect objects that enter the sphere. The player's ball entering the sphere means healing; exit means stop healing.

```
    void OnTriggerEnter(Collider other)
    {
        if (other.name == "HealthArea"){inHealthArea =
true; }
    }

    void OnTriggerExit(Collider other)
    {
        if (other.name == "HealthArea") { inHealthArea =
false; }
    }
```

Figure 3-27 healing area's main logic in CollisionDetect.cs

The damage zone is a non-collision vertical plane. If the player's position is behind the damaging wall's location (X-coordinate), the player will suffer continuous damage. The wall will gradually move towards the end of the map, and the amount of damage will increase according to the distance between the wall and the end of the map. In that case, players must move towards the smaller areas, just like in the Battle Royale games.

```
 void PoisoningEffect()
    {
        damageareaPosition = damagearea.transform.position;
        if (player.transform.position.x < damageareaPosition.x)
        {
      //do damage action according to the distance between the end of
the map
        }
    }
```

Figure 3-28 Damaging area's damage mechanism in PlayerManager.cs

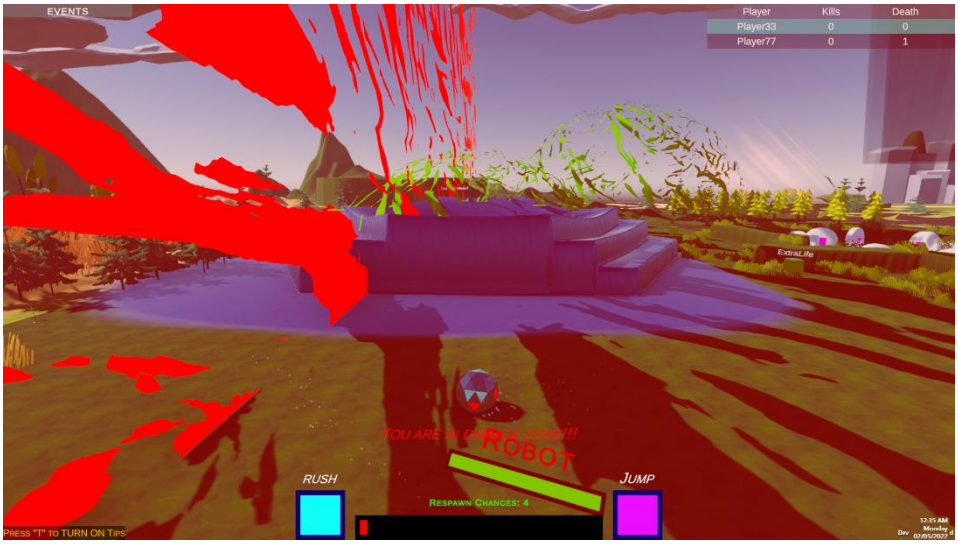


Figure 3-29 Damage zone showcase

3.4 In-Game Network

Network logic is one of the most changing and confusing parts of this project. Since all players' prefabs will be instantiated in each user's local computer in the Unity engine, so it is necessary to recognise what each player should control and what online messages they should get and send.

3.4.1 Game Managers

In order to let players be born on the same map, it will need to create networked Game Objects in Photon, using `PhotonNetwork.Instantiate("prefab name", Vector3 Position, Rotation, Custom data)` to birth the player's Managers and game objects; the `prefab name` will be each player's selected ball's name.
Each Manager is a script and will have their own duty. A simple sequence diagram Figure 3-30 shows all managers' relationships; More details are in Appendix 2.

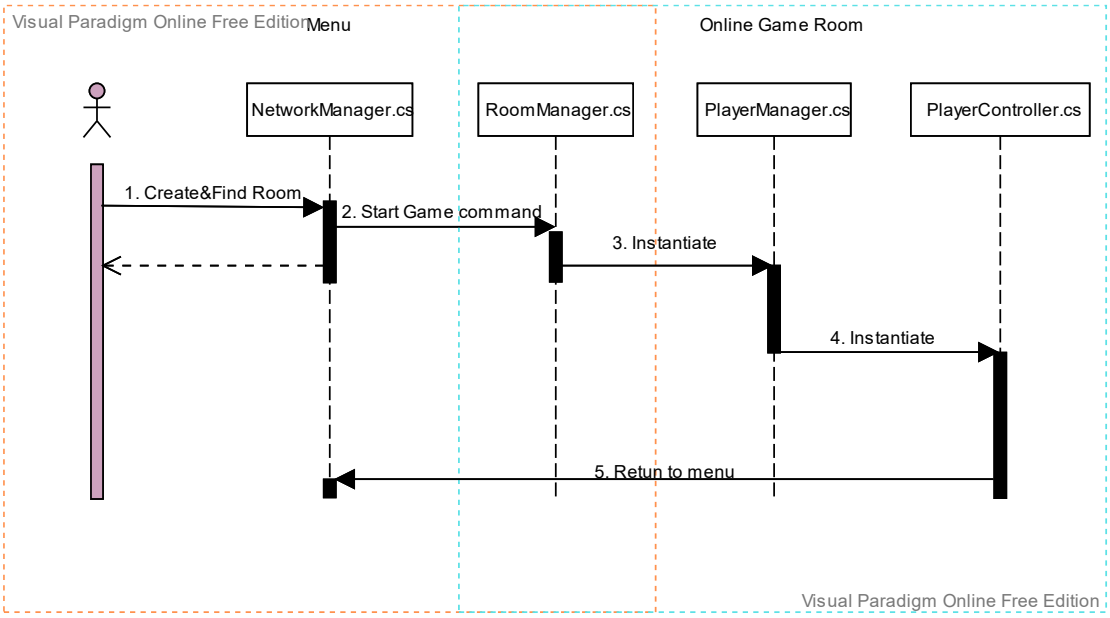


Figure 3-30 Sequence Diagram shows the Manager scripts' relationship

RoomManager will bring the player to the battle zone; when a player is in a game and playing, the **PlayerManager** script will be in an empty game object. It will instantiate the player's ball and Spectator. Each player will have their own **PlayerManager** on their local computer. If there are five players in a game, each computer will have all 5 **PlayerManagers**. **Playermanager** will handle almost all correspondent players' data, such as health, chances, kills and death counts. It also contains the game rules algorithm; all player's data networks receive and send will also be managed in **PlayerManager**. All movement and its sound effects will be handled in the player's controller's script, and it will have two scripts called **MovementController**.cs and **JumpController**.cs and sound effects (like footsteps) will be in **JumpController** because it handles the grounded algorithm, more details are in Appendix 1

3.4.2 Avoid Confusion

Since all player's object components will be instantiated in each player's local computer, so each player's computer will contain all play's cameras, controllers, managers (**PlayerManager**) and UI. Suppose the game does not differentiate these components. In that case, each player will display and run all player components locally —Which will cause disastrous results. It will need **PhotonView** components in PUN to distinguish player objects, as shown before in 3.1.6.

The PUN API `photonView.IsMine`will is used to recognise the local player's component. This API will return `True` if the **PhotonView** is "mine" and can be controlled by this client.

```
    private PhotonView playerPhotonView;

void Awake()
    {
    playerPhotonView = GetComponent<PhotonView>();
    }
void Start()
    {
    //if this photon View is not belonged to the local player:
   if (!playerPhotonView.IsMine) //or this.PhotonView.IsMine
        {
/*Destroy this player's camera, UI; disable all scripts that
should not occur on the device*/
         }
    }
void Update()
    {
//if not local player, then it should not do the below operations
and return:
   if (!playerPhotonView.IsMine){ return;}
    /*loop operations, like movement controller*/
    }
```

Figure 3-31 The core logic of confusion avoidance, should attach it to all scripts that are repeated but separate for individual players

3.4.3 Synchronising Physics

Synchronising each player's position, velocity, rotation and scale, and other physics data is essential for collision and for players to get other players' latest positions. Photon Network provides these solutions in their PUN product [6].

PUN provides several transport protocols, such as UDP and TCP. The project uses the UDP protocol because of its low latency, multi-user transmission, and lower overhead. To synchronise the physical information of an object for all players, it will need to attach the corresponding Photon component and adjust its settings.

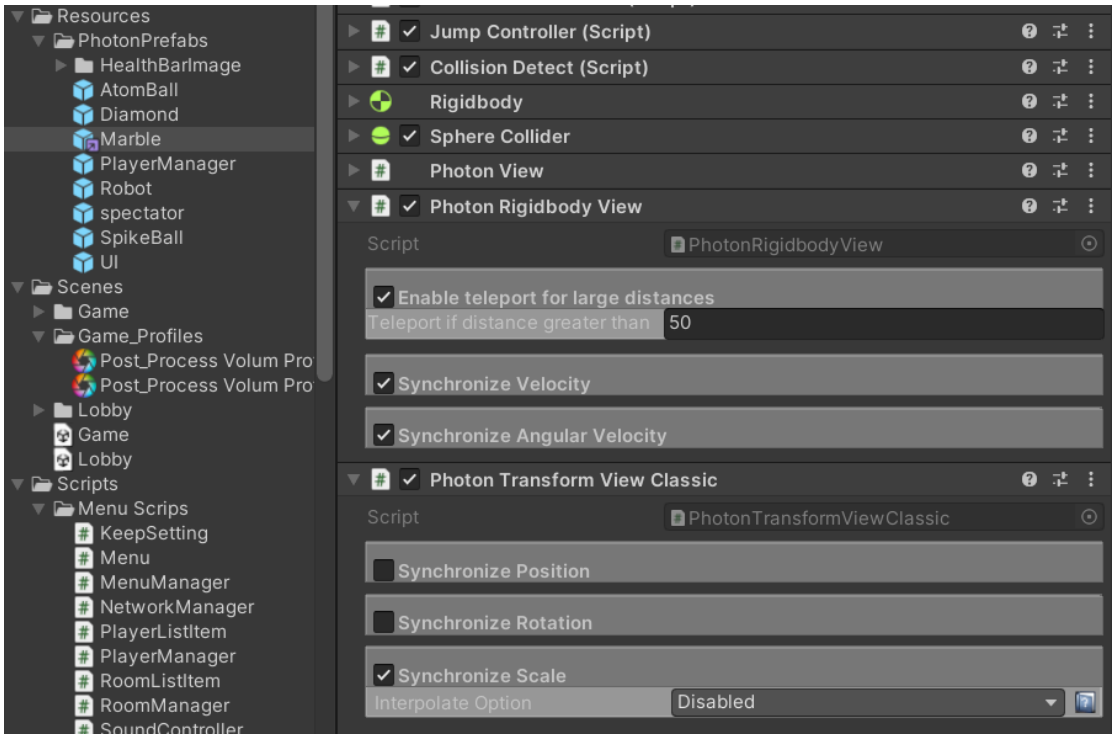


Figure 3-32 attach Photon Sync Views to any objects that need to sync physic data

Photon Rigidbody View: This group will automatically synchronise the rigid body position, velocity and angular velocity of the corresponding object, but not the same rotation and orientation of the object, which means that other than the position of the object, the physics information will be different for each player, but it is sufficient for sphere movement.

Photon Transform View: This component will synchronise the game objects' position, rotation, and scaling. It also provides many different options to make the objects move more smoothly. This component is only suitable for character movement and first-person shootings but not suitable for physical collisions. Collisions will not happen since it does not sync rigid bodies. This project will not use it to synchronise the position of objects but only to synchronise the scaling of objects.

Due to the uncertainty of the Unity physics engine, Photon Rigidbody View cannot ensure that every player gets precisely the same position [6], which leads to a certain chance that collision will not happen correctly, but this situation hardly occurs in actual tests. Although it is possible to use only Transform View and design the own algorithm to simulate the collision, the workload is enormous and challenging to consider in all cases. Development has also tried to enable and debug both **Views** to get more accurate synchronisation and collision during the development process. However, the effect is not as good as using Photon Rigidbody View alone. However, the result is not as good as using Photon Rigidbody View alone. It also increases the burden on the server, so finally, only Photon **Rigidbody View** is used to synchronise all object positions and velocities, and it works well.

3.4.4 Online Structure

In general, the structure of online games is complex and abstract. All of the player's components and objects will be instantiated on each player's computer. Unless controlled explicitly by the network, they will run locally on each player's computer. This means that some components for non-native players must be differentiated and disabled/removed; otherwise, chaos will result. Local players can only control their sphere and view their UI (such as event board, camera, scoreboard, HUD). These components of other players will be disabled and deleted locally. Players' physical information will be uploaded to the server via Photon Rigibody and Transform View and synchronised to all players. The game event board, scoreboard, and player information will be synchronised through the local player's PlayerManager and computed and displayed locally. **Error! Reference source not found.** shows the in-game local player's use case diagram. Combined with Figure 3-35, they form this online game's overall logical structure.

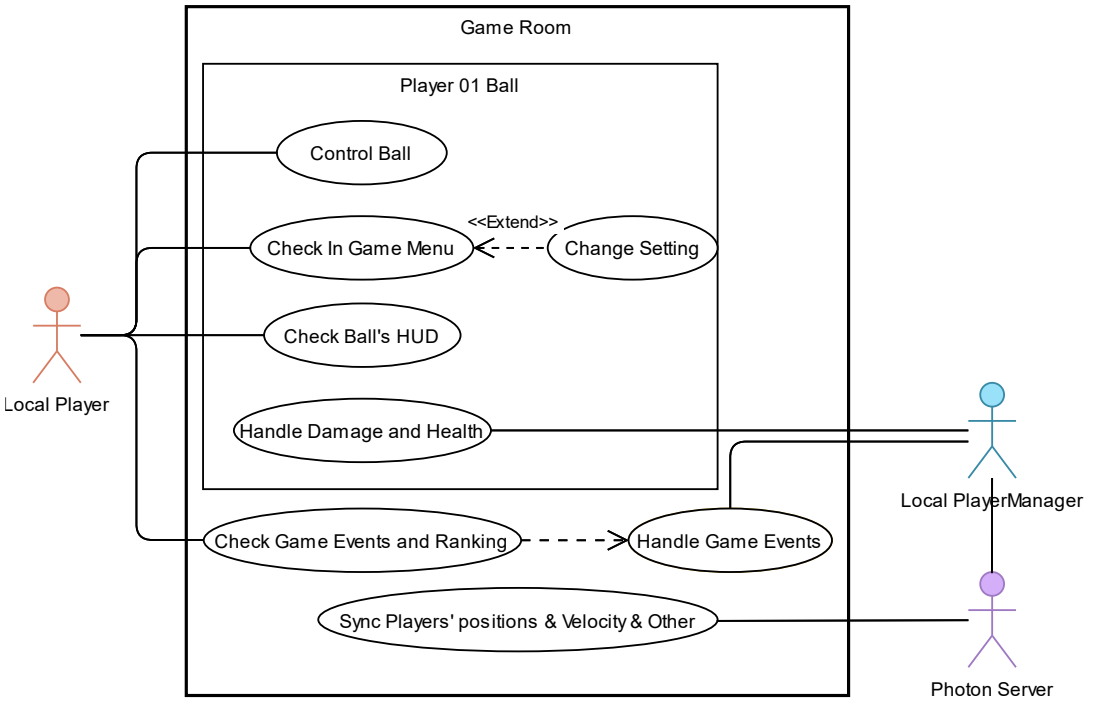


Figure 3-33 In-Game local player useCase

3.4.5 Synchronising Player's Data

Besides synchronising the physical information of the object, the player should also have real-time access to all the player's personal information, such as the player's Nickname, health, kills, deaths, and all the player deaths and kills other messages. It will use the **"Remote Procedure Calls" (RPCs)** provided by PUN to accomplish the above information [6].

PRC means the same method calls on remote clients in the same room. For example, if a player wants to sync its health data to all clients, the scripts need to do the following: When syncing `currentHealth` data sync, the scrips will need to call an RPC function and let this function execute on all clients (`RPCTarget.All`). A visual diagram Figure 3-35 shows more specific details.

```
 private void Damage (float damageValue)
    {
/*Damage action, like currentHealth -= damageValue*/
this.photonView.RPC("SendHealthData",RpcTarget.All,currentHe
alth);
    }
  [PunRPC] //sync _xxvalue to all
    void SendHealthData(float _currentHealth)
```

Figure 3-34 core usage of PUN RPC

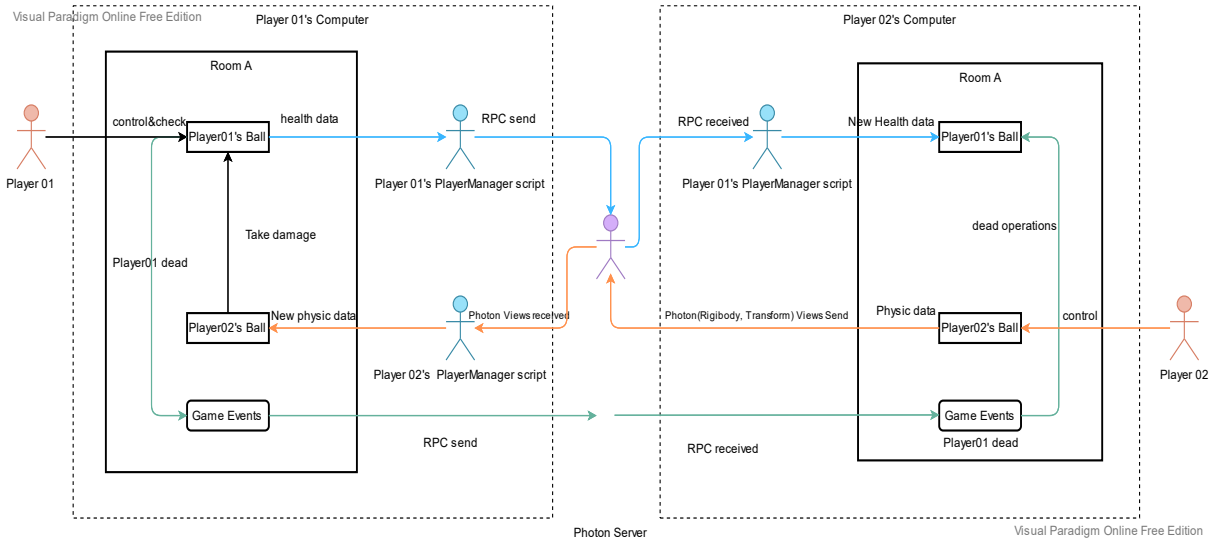


Figure 3-35 A visual diagram showing how the game synchronizes the player's personal information (health, deaths, kills), physical information (position, speed), and real-time game events

3.5 Game HUD System

The game referenced some of the designs of the heads-up display that games often use, such as player Billboard, rankings and game event broadcasts, also the player skills and health bars display.

3.5.1 Player Status Interface

The player's Status interface provides information about the player while playing, including the player's blood level, respawn count, jump and sprint skill cooling times, and interface effects when recovering and wounded. The cooldown will be displayed as the degree of colour rotation filling in the square. The interface will turn red when the player is injured, indicating that the player is taking damage and healing.

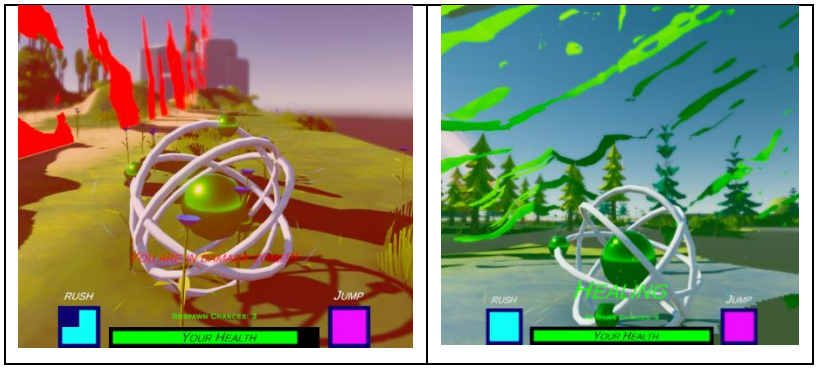


Figure 3-36 player HUD showcase

3.5.2 Billboard

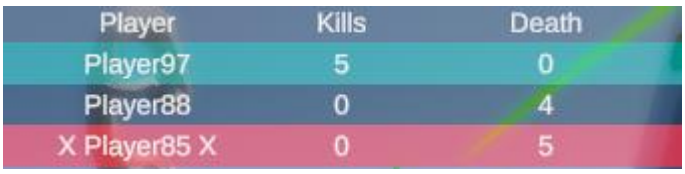
Billboard is a UI interface hovering over the heads of all players, and this component will display the name and blood level of the corresponding player. The blood level is represented by a rectangular bar filled with different colours. For example, the player's blood level is green if it is greater than 70%, yellow if it is 40% to 69%, and red if it is less than 40%, which can visually reflect the player's current status. Billboard can penetrate objects, i.e., not blocked by other objects, so that players can have faster and more direct access to the location and status of other players.

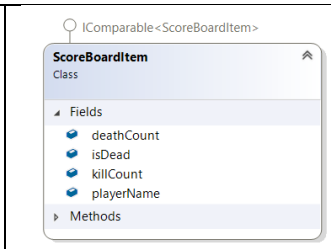
Figure 3-37 game billboard showcase

3.5.3 Ranking board

Each player will be able to view the ranking board within the game, which will display and sort the number of kills and deaths of the players currently present in real-time. The scoreboard will be calculated and displayed separately on each player's local computer. The ranking board uses Unity's UI component layout group, making it very simple to align and layout multiple UI components (such as rectangles and text) in the order in which the rectangles are initialised. Each rectangle will contain a player's name, number of kills, number of deaths and elimination status and will encapsulate these data in a class: `Class ScoreBoardItem` so that when all the player information is obtained through each player's **PlayerManager**, all the information can be put into the Class, and all the players' classes can be stored in a `List <ScoreBoardItem>`, and then sort them directly based on the number of kills and deaths by implementing `IComparable<T>`, fist compares with kill count, if kill count is identical, then compare with the death count. Also, the display of local players(blue) and eliminated players(red) will be differentiated, making it easier for players to get intuitive information.

Figure 3-38 Score Board with its class diagram, showcase and sort logic






```
Public int CompareTo(ScoreBoardItem other)
  {
     If (other.killcount > this.killcount)
      {
          Return 1;
      }

      else if (other.killcount == this.killcount)
      {
          If (other.deathcount > this.deathcount)
            { return -1; } else return 1;

      }
      else return -1;
  }
```

3.5.4 Game Events Board

Events of the game will be broadcasted in real-time via the top left corner of the game interface. The broadcasted information includes but is not limited to player join and leave, drop, kill and elimination information. All the broadcast information will be generated and passed to **GameInfoManager**.cs, then broadcasted to all players' event boards via PUN's RPC. More relationships and processes can be found in Figure 3-35.

Same as the scoreboard, each message in the event board is an encapsulated class **GameInfoManager** that contains the event message string and a delete time float. The deleted time is intended to be automatically deleted after a while has been broadcast. It has a shuffling effect and makes room for new messages. The delete countdown can be done using Unity's `Invoke (string methodName, float time)` method.

Figure 3-39 Some showcase of Game Events Board

3.6 Training Ground Tutorial

When the player selects the tutorial on the start screen, there will be a guidance interface in the upper part of the screen after entering the training ground. Players can adjust the progress through Q and E. Some parts of the guidance will require the player to perform some operations before entering the next step. The tutorial will introduce the player to all the game rules and mechanisms so that the player is familiar with them and faster to start the whole game.

Besides, there will be a control tip on the left corner during all types of game modes, and the player could press T to switch the tip's display; this could remind the player if they forgot how to control the ball

Figure 3-40 Control Tips(left corner) and Tutorial(top)

3.7 NPC Design

The game will have several non-player character (NPC) robot spheres to make the game more challenging. Many of the robot's components overlap with the player's movement, jumping, and billboard. The difference is that the movement and jumping of the robot are triggered at a random time and stop randomly so that the robot's movement is more regular and realistic. The random waiting time will be implemented using Unity **`Coroutine`** and **`Random`** functions.

```
  void Start()
     {
       StartCoroutine(WaitJump()); //only do on master client
      }
IEnumerator WaitJump()
    {
        while (true)
        {
            if (hasTarget)
            {
                rb.AddForce(Vector3.up * jumpforce);
          JumpCoolingTime = UnityEngine.Random.Range(MinTime, MaxTime);
            }
             //wait every random second then jump
            yield return new WaitForSeconds(JumpCoolingTime);
        }
    }
```

Figure 3-41robot's jump coroutine showcase

These robots will automatically look for the nearest player on the map as the attack target, Figure 3-42. If the player enters its attack range, the robot will start moving and attacking the player. All robots' behaviour, such as movement, damage calculation and events broadcast, will only run on the *master client*, so it will need to handle the situation if the master client quits the game. Since the master client will automatically switch to another player, it will need to overwrite the PUN **`OnMasterClientSwitched()`** and let the robot run on the new master client. Furthermore, the damaged wall's movement is similar to this method.

```
public GameObject FindClosestEnemy()
 {//only do on master client
     GameObject[] players = GameObject.FindGameObjectsWithTag("Player");
     GameObject closestTarget = null;
     var distance = attackRange;
     Vector3 position = transform.position;
     foreach (GameObject player in players)
     {
    // calculate distance between one player and robot:
    float pointsDistance = Vector3.Distance(player.transform.position, position);
          if (pointsDistance < distance)
          {
              closestTarget = player;
              distance = pointsDistance;
          }
     }
     Array.Clear(players, 0, players.Length); //clear the array for next search
     return closestTarget; //return the closet player
 }
```

Figure 3-42 robot's find closet target algorithm

3.8 Other Game Tools

There are many other interactive props in the game:

The jump pad that can bounce the objects;
Moving platforms that can carry the player to other sides;
Traps that can fool the player;
Teleportation cubes that can transfer objects to another side immediately;
Recovery cubes that can give the player extra respawn chance;
Scaling cubes that can make players shrink or enlarge.

All the box functions are integrated into the **ToolBox** script and have different roles depending on the box's name. More details can be found in Appendix 2 and Appendix 3

4 Game Optimisation

Game optimisation is also an essential aspect of game development that needs to be considered. The optimisation determines the player's experience and the threshold of the game [18]. A failed optimisation will lead to lower performance devices not running the game. Higher performance devices may encounter lag and frame drops or even cannot run the game smoothly.

In Unity, most game optimisation can start with code optimisation and graphics optimisation**.** Exploring gameplay performance usually requires the **Unity Profiler**. The Profiler gathers and displays data on the application's performance in the CPU, memory, renderer, and audio [5]. This tool will support this project's most of the optimisation processes. Instead of explaining what the Profiler does, here only focuses on what is optimised.

4.1 Code optimisation

The Start function is suitable for initialising any code before the gameplay.

Unity `GetComponent` will take up slightly more system performance because it usually has to iterate to find all the components on an object. The derived classes of Unity's Object tend to take up even more performance, such as `GameObject.Find()`, and `GameObject.findGameObjectswithtag()` will both iterate to find all the objects in the current scene, which will cost much overhead when there are many objects in the scene [19]. If these methods are then placed in a loop, such as `Update(){}`, then the game will iterate through once for each frame, which will significantly impact performance.

The most straightforward solution would be to set object search in `Awake(){}` and `Start(){}` when the game scene initialises. Since these methods will only be run once the code file is activated, if the performance-consuming code is used outside of these start functions, try to call it when the game needs it.

When looking for a Game Object, `Transform.Find*` would be better than `GameObject.Find*`, because this method will only traverse through the object's children, narrowing the traversal.

This project requires many objects and variables, so almost all variables are placed at the scripts' beginning; `transform.Find` is used as much as possible; set many costs codes to execute as few frequently as possible.

Three of the most apparent optimisations were:

- **Synchronising player status information**

In the early development, the `RPC` function sent all information about the player (such as blood, kills, deaths, elimination status) was placed directly in the `Update` loop, and the algorithm for determining the champion (traversing the game to find objects marked as "Player") was also placed in the `Update`,

They were removed from the loop because it is not always necessary to synchronise and judge the above information:
Only when the player was injured or recovered will the player's blood be synchronised;
Only when the player dies or is killed will the other information be synchronised and judged champion once.

- **The scoreboard algorithm.**

A similar principle applies to the scoreboard. Since the scoreboard needs to find all the **PlayerManager** in the game and get each player's information separately and involves sorting and list adding and clearing, this takes up the highest resources. The Unity profiler shows that if the scoreboard is constantly refreshed in the `Update` loop, it will take about 5% of the CPU resources. Finally, it was changed to update the scoreboard only when a player dies or is killed.

Furthermore, use Unity's `Coroutine` method to let Unity run a code snippet every few seconds, such as the robot to find the enemy algorithm can find the nearest player every few seconds in Figure 3-41, which can also reduce part of the burden

In addition, reducing the use of new can reduce the generation of garbage and new classes to a certain extent, such as:

```
Transform.position += new Vector3(10, 0, 5); change to
Vector3 _position = … ;
Transform.position += _position;
```

Above are very obvious CPU optimisations (about 10%). However, at the same time, it is not easy to distinguish when to refresh once. Sometimes, the network and latency will cause no update information, but fortunately, all problems are overcome

4.2 Graphics optimisation

Image optimisation is also a very critical part of reducing GPU stress. Since the project uses high-quality images and post-processing to improve the image quality, it will require slightly higher graphics performance.

Texture resolutions

The first thing to reduce is the texture's resolution. The game is mainly Poly painting style, so it does not need high texture resolution. In the beginning, the texture resolutions are 2048 or even 4096, directly downgraded to 512 or lower; this will significantly reduce the load on the GPU (8%).

Player quality settings

It also adds options for players to adjust the overall image quality according to their devices: very-low, low, medium, high, very high, and ultra, so that players can choose the image quality that best suits their device's performance. These image quality adjustments include several aspects such as the view distance, vertical sync, anti-aliasing, ray rendering, LOD, shadows, and image resolution, and these usually most influent the game performance.

Culling

Occlusion culling is a process which prevents Unity from performing rendering calculations for Game Objects that are entirely hidden from view (occluded) by other Game Objects [5]. Cameras perform culling operations in every frame that examines the Renderers in the scene and exclude (cull) those that do not need to be drawn.

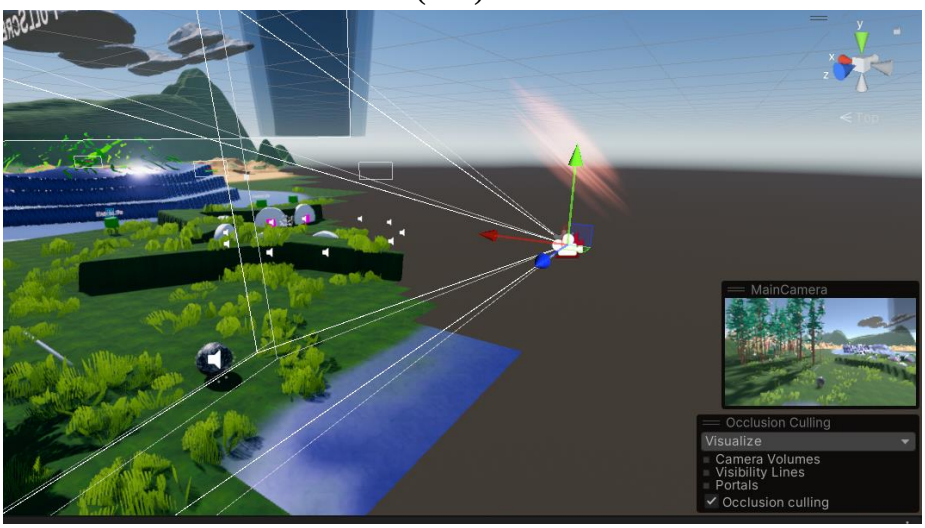


Figure 4-1 Only objects that are in the line of sight of the player's camera will be rendered

LOD Group

This game uses LOD for almost all vegetation, Level of detail (LOD) is a technique that reduces the number of GPU operations that Unity requires to render distant meshes. When a Game Object in the scene is far away from the camera, the player will see minor detail than the Game Object when it is close to the camera [5]. The player will not focus on distant objects, so there is no need to load more details to distant objects. Here is an example:

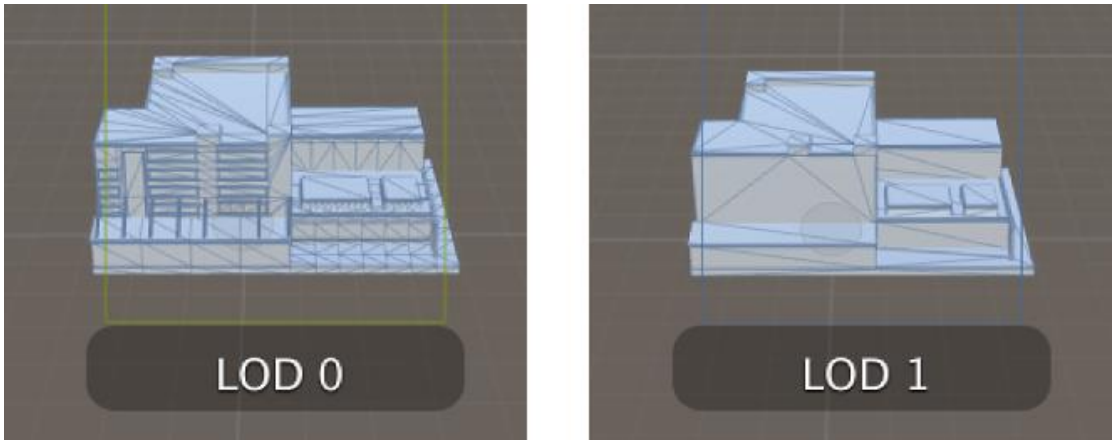


Figure 4-2 At LOD 0, closer to the camera, it shows a mesh with many small triangles. At LOD 1, far from the Camera shows the mesh with far fewer triangles [5]

Vegetation and Particle density

Suppose the vegetation and particle effects in the game are too dense and numerous. In that case, the computer will spend more resources to calculate and render them. So, this game will try to reduce vegetation density to a reasonable level and make more adjustments through Terrain's settings panel and will not use particle effects except for players and robots.

Final optimisation results

In the test devices list Figure 2-3, all devices can run this game smoothly with beautiful game video quality, and some test data are shown below:

Device Configuration	Resolution	Video Quality	Average Frame Rate	GPU Usage	CPU Usage	RAM Usage
i7-9750H, GTX 1660Ti, 16GB	2K(2560x1440)	Ultra	60	45%	15%	(750MB)6%
i7-8550U, MX150, 8GB	1080P(1080x1920)	Medium	60	80%	23%	(750MB)10%

Figure 4-3 In part of the performance showcase, The frame rate will be reduced in scenes with dense vegetation

5 Testing & Evaluation

The content of the test evaluation includes the game quality, smoothness, performance occupancy, operation feel, physics simulation, online performance and gameplay and other aspects. How each criterion should be assessed will not be presented here, as the criteria are too subjective and diverse. However, the following criteria are objective and necessary to determine the success of a game project:

- The networking function must be perfect and stable, and it must be possible to have more than one player playing together in a wide area network
- The game menu interface should be easy to use and simple to understand without bugs.
- The player should be able to control the game character smoothly
- The game should be almost free of crashes and flashbacks
- All game rules and mechanics in the game must work adequately and reasonably.
- The performance requirements of the game must be adapted to most windows computers with discrete graphics cards. They must run the entire game as smoothly as possible.
- The game should have an excellent graphic and art design.

Currently, the above criteria are agreed to have been met based on personal and other testers' feedback.

Single Engine Test

The central part of the game development is tested and evaluated in the local Unity Engine Editor, a very efficient and vital part of the game development. The developer can view the output of errors and debug information in time through the console in the editor. It is easy to make changes immediately, and the running effect is the same as the final product.

Multitask Single Device Test

The session will simultaneously run multiple game processes (including Unity Engine Editor) on the same device. These processes through the Internet for multiplayer games rather than a local connection, so it can visually feedback the multiplayer online error and bug, but also can be a more intuitive display of the actual situation of the game performance and its performance, while most of the errors can be instantly output to the unity console, to facilitate debugging and change

Multi Devices Users' Test

Both the above tests and evaluations were run locally by the developer, so they do not fully represent the quality of the game. Others will be invited to test and evaluate this section. In this step, the game will be packaged into the "Development Build" package, which can output and send testing information. Distribute these test packs to players, let them play together, then collect test information and feedback, update a new version and distribute and test them again for a more efficient product iteration. The following Figure 5-1 shows the feedback from the session players at different times, from the very beginning to the present, with the Responses results.

Feedback	Responses
Players sometimes cannot jump again after hitting a tree	Resolved More details in 3.1.3.1
Players can go through the ground mould	Resolved This is because of the mistake of rigid body setting,
The Terrain has cracks	Resolved This is caused by the difference in height between different blocks after chunking the Terrain in 3.1.4, which I was ignored when the map was drawn.
Player camera control has sensitivity some slow	Resolved *Cininemachine* sets this; readjust it
The menu screen sometimes does not show the created rooms	Resolved This was caused by a faulty algorithm that needed to be refreshed every time there was a network update, and I ended up using the official solution Matchmaking Guide | Photon Engine
The range of movement of the camera up and down is too small	Resolved *Cininemachine* sets this; just readjust it
Game event sometimes shows the same message repeatedly	Resolved This is caused by incorrect code logic and not using `photonView.IsMine`correctly causing the **PlayerManager** to run on one side for each player
Lack of sound effects	Resolved Later, the game added perfect sound effects, such as collisions, jumps and footsteps changing according to speed.
Too many words on the menu screen	Resolved More details in 3.1.5
When back to the menu player need to renter name and reselect the ball	Resolved This is because Unity will rerun the script every time the scene is switched, which will be refreshed. Use `DontDestroyOnload(object)` to stop this mechanism and put the information that needs to be kept in it; the related script is **KeepSetting**.cs
Play competitive mode in the United States using the European server will crash the game	Not Resolved The player is playing Training Ground mode without a problem. The program does not show any error, so the problem has not been solved

Figure 5-1 Players' feedback and responses

6 Conclusion

This project is dedicated to developing a multiplayer online battle royale game with a ball theme using Unity Engine and Photon Network Engine. The game is planned to enable multiplayer online functionality over a vast area network. Players can control various balls to fight against other players within the game and keep moving with the ever-shrinking safety zone. Players can hit each other to cause damage or knock enemies off the map until the last player survives and wins. At the same time, it includes various environments, props, and enemies to make the game more exciting and challenging.

There are many features implemented step by step in this project, and not all of them are elaborated in this dissertation, so all of them will be listed to show the game implemented features in Appendix **3** and all their Class's main tasks in Appendix 2.

Overall, the project has primarily achieved the blueprint expected in the original plan. Although there are no player shields and other skills envisioned early in the detailed proposal because new game mechanics replaced them. The rest of the plans have been successfully implemented. Many additional features have been implemented during the project's development, such as NPC, springboards, the healing area, and functional cubes; these could make the project very extensible. The project, with good internal reusability and the ability to iterate on many new features and modes, also has a specific game experience.

7 BCS Criteria & Self-Reflection

- **An ability to apply practical and analytical skills gained during the degree programme.**

I have learned COMP222 for Game Design and COMP 282 for C#, and the early period of learning java object-oriented programming; these have played an essential role in developing this project. COMP201 Software Engineering helped me become familiar with project development and have opportunities to learn UML diagrams and Multiple project development frameworks; the COMP208 group software project also and can help me manage my project; I also participated in the University's summer project and learned some UDP and remote-control knowledge with Unity project simulation.

- **Innovation and/or creativity.**

Some artificial intelligence characters might help my game become innovative; also, the fun will be the priority because it is a game. This project has several interesting interactive props and a dynamic and prosperous game map, with the combination of sphere and low poly style as the theme of the battle royale game seems to be very novel and creative.

- **Synthesis of information, ideas and practices to provide a quality solution together with an evaluation of that solution.**

I always prefer to practice turning my idea come true. My project is managed by using many adequately helpful information and knowledge. This project fully utilised my programming, project management, software development skills and almost comprehensive knowledge. Agile development played an important role in guiding me to implement and improve the game efficiently and steadily step by step.

- **That your project meets a real need in a wider context.**

My project could satisfy this requirement; the online game is still in rapid growth and high demand, and Battle Royale games are still in great heat. It could bring many people to enjoy it, let them relax and have fun, and even enjoy the game with their friends. Nevertheless, the reusable also works; it will not be just a marble game. With the object-oriented development of many games, its online functions and other components could reuse to develop many other types of games.

- **An ability to self-manage a significant piece of work.**

I have shown my best attitude and efforts to this project; I arranged the work with reasonable and attainable pieces to guarantee my project processions' good quality and efficiency. The entire project is backed up by multiple network drives in real-time. The version is controlled via GitHub; it allows players to view the latest developments and changes, allowing rapid iteration of the game development. In addition, the project used my website to show the game and provide download links for players [20].

- **Critical self-evaluation of the process.**

I always keep evaluating my project by myself regularly to make sure most of the processes are in the best situation. The whole development process of the game was challenging. I encountered unprecedented difficulties, but I persevered and conquered them all. After the actual combat of this project, I realised how limited my ability was. Game development and the full use of game engines require a qualified developer to have almost all knowledge, including physics, mathematics, programming, network communication, design, photography, modelling, media art, optimisation, and project management.

Due to my lack of knowledge and project experience in this development process, I spent much unnecessary time fixing and improving a bug or function. However, these difficulties only need a few simple steps for a professional person to solve. At the same time, I found my programming skills still need to be improved, the reusability, logicality and simplicity of the code are still far from those professional developers, and some of the code may be too redundant. I also found that there are still many Unity functions that I have not learned. These features could make the game much more effective and efficient and speed up the development process, so I look forward to continuing to learn in the future.

Overall, I am still pleased because this project meets and exceeds my expectations. It also strengthened my programming and project management skills, improved various aspects of my ability, it gave me the fun of game development.

References

[1] E. Irpan, A. Gohil and N. TenBoer, “2021 Gaming Report,” 2021. [Online]. Available: https://create.unity.com/2021-game-report. [Accessed 4 2022].

[2] Unity, “Unity Gaming Report 2022,” 2022. [Online]. Available: https://create.unity.com/gaming-report-2022. [Accessed 4 2022].

[3] “Global Battle Royale Game Market: Size & Forecast with Impact Analysis of COVID-19 (2021-2025),” researchandmarkets, 11 2021. [Online]. Available: https://www.researchandmarkets.com/reports/5481229/global-battle-royale-game-market-size-and. [Accessed 4 2022].

[4] J. Gregory, Game Engine Architecture, 3 ed., New York: A K Peters/CRC Press, 2018.

[5] “Unity Documentation,” Unity Technologies, 16 4 2022. [Online]. Available: https://docs.unity3d.com/Manual/index.html. [Accessed 3 2022].

[6] “Photon Unity Networking 2,” Photon Engine, 2022. [Online]. Available: https://doc.photonengine.com/en-us/pun/v2/getting-started/pun-intro. [Accessed 4 2022].

[7] “Comparison of C Sharp and Java - Wikipedia,” wikipedia, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Comparison_of_C_Sharp_and_Java. [Accessed 4 2022].

[8] E. Cronin , F. Filstrup and A. Filstrup , “A Distributed Multiplayer Game Server System,” 4 5 2001. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.23.2459. [Accessed 4 2022].

[9] A. MARK , “BEGINNERS GUIDE TO NETWORKING,” Game Maker, 10 10 2019. [Online]. Available: https://gamemaker.io/en/blog/beginners-guide-to-networking. [Accessed 4 2022].

[10] I. Barri, C. Roig and F. Giné, “Distributing game instances in a hybrid client-server/P2P system to support MMORPG playability,” *Multimedia Tools and Applications,* vol. 75, pp. 2005-2029, 2016.

[11] “Limitations of p2p multiplayer games vs client-server,” Stack Exchange, 6 2015. [Online]. Available: https://gamedev.stackexchange.com/questions/67738/limitations-of-p2p-multiplayer-games-vs-client-server. [Accessed 4 2022].

[12] P. Network, “PUN,” Photon Network, [Online]. Available: https://www.photonengine.com/en-US/PUN. [Accessed 4 2022].

[13] A. Quality, “The Scrum Papers: Nut, Bolts, and Origins of an Agile Framework,” Academia.edu, 2011. [Online]. Available: https://www.academia.edu/43872765/The_Scrum_Papers_Nut_Bolts_and_Origins_of_an_Agile_Framework?bulkDownload=thisPaper-topRelated-

sameAuthor-citingThis-citedByThis-secondOrderCitations&from=cover_page. [Accessed 4 2022].

[14] G. N. Yannakakis, H. P. Martínez and A. Jhala, “Towards affective camera control in games,” *User Modeling and User-Adapted Interaction,* vol. 20, no. 4, pp. 313-340, 2010.

[15] “About Cinemachine | Cinemachine | 2.8.4,” Unity Technologies, 20 12 2021. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.cinemachine@2.8/manual/index.html. [Accessed 4 2022].

[16] R. J. Pagulayan, K. Keeker, D. Wixon, R. L. Romero, T. Fuller and D. Gunn, “User-centered Design in Games,” *Human Factors and Ergonomics,* pp. 795-822, 2012.

[17] P. D. John, S. Keith and R. Pagulayan, “A survey method for assessing perceptions of a game: The consumer playtest in game design,” *Game Studies,* vol. 5, no. 1, 2005.

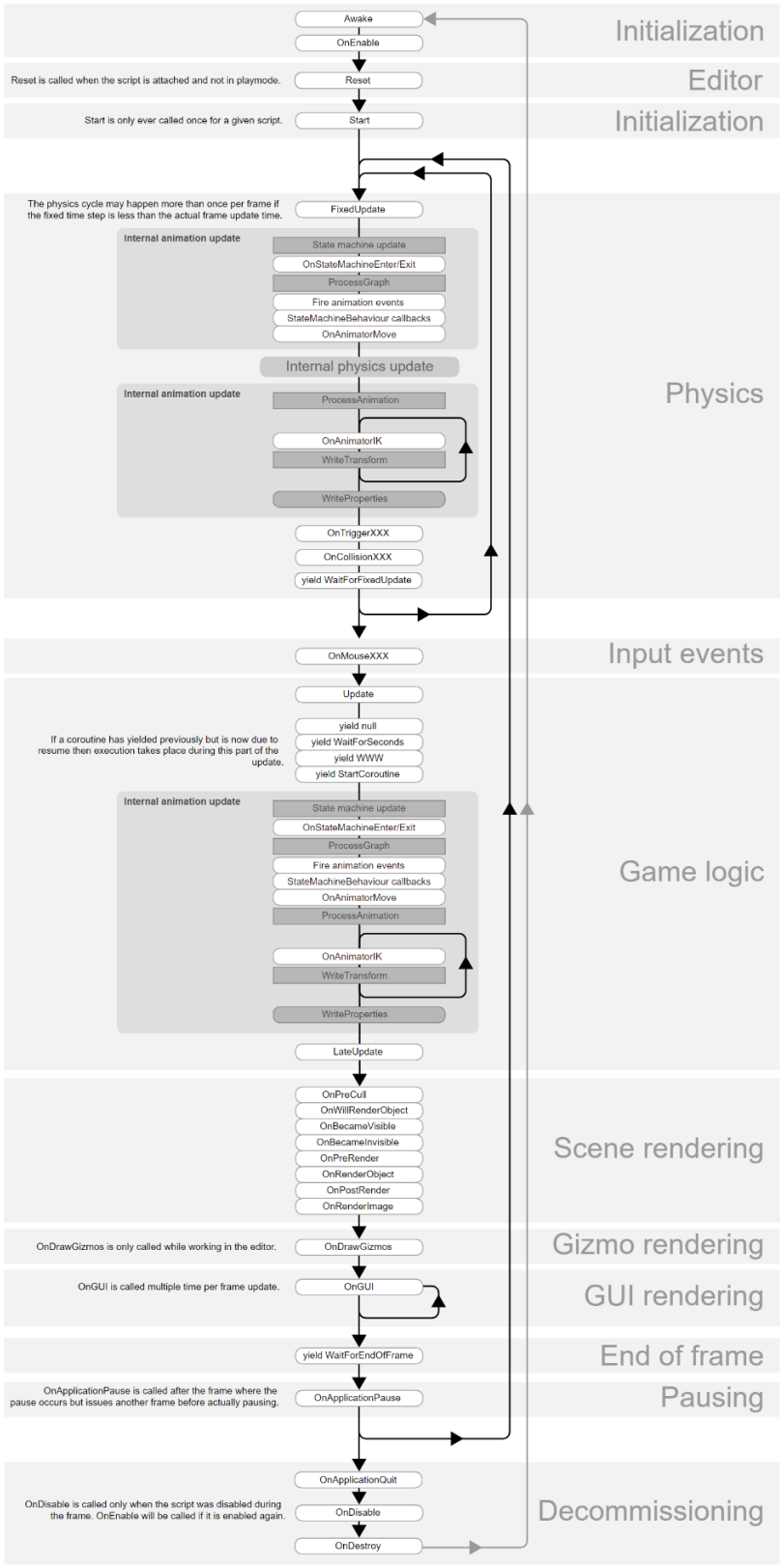
[18] T. Peter, “What 'optimization' really means in games,” 22 9 2016. [Online]. Available: https://www.pcgamer.com/uk/what-optimization-really-means-in-games/. [Accessed 4 2022].

[19] H. Mohamed, “Tips for Code Optimization in Unity,” Medium, 11 8 2021. [Online]. Available: https://medium.com/nerd-for-tech/tips-for-code-optimization-in-unity-947b1fd9b6a9. [Accessed 4 2022].

[20] L. Chen, “Personal Unity Online Game Development,” DoggyChen, 2022. [Online]. Available: https://doggychen.com/personal-unity-online-game-development/. [Accessed 4 2022].

Appendices

Appendix 1 Script Lifecycle Flowchart. The following diagram summarises the ordering and repetition of event functions during a script's lifetime.



Appendix 2 All Class' names with their responsibilities, All in Assets\Scripts folder

Class name	**Main task**
Animateloading.cs	Control all animation playback
Buttonsoundctrl.cs	Included in the button's prefab to control the button's sound
Collisiondetect.cs	Detects and calculates the player's collision damage and transfers it to Player Manager
Jumpcontroller.cs	Control the player's jump as well as the player's sound effects, which can be customised with various attributes
Jumppad.cs	Control of the operation of the springboard is included in the springboard's fit, which allows you to customise the strength
Leftandright.cs	Control the movement of the moving blocks in the game, customisable
Menu Scrips/keepsetting.cs	Save in-game settings and information about the player's name, character role, etc.
Menu Scrips/menumanager.cs	Control game menu switching, character selection and input detection
Menu Scrips/NetworkManager.cs	This script is responsible for all menu interface networking operations and menu logic
Menu Scrips/playerlistitem.cs	This script is responsible for displaying information about each player as they enter the room. It encapsulates the player's name and other network information
Menu Scrips/**PlayerManager**.cs	This script is the core script when the game is initialised by the room manager and manages all the information and logic of the player; it saves and processes all kinds of information of the player, such as the amount of snow, death, and resurrection times, etc. It also controls the resurrection and death of the player and undertakes the task of sending and receiving player data, and contains the logic of the game rules
Menu Scrips/RoomListItem.cs	Similar to Playerlistitem, except that this one show room information
Menu Scrips/RoomManager.cs	The script will always be present in all game scenes and will not be deleted. It will control the player's scene switching
Menu Scrips/soundcontroller.cs	This script controls the setting of all game background music
Movementcontroller.cs	This script controls all the player's scrolling operations and receives scrolling commands
Rotateandshake.cs	This script controls all the player's scrolling operations and receives scrolling commands

Class name	Main task
Spectatormovement.cs	In observer mode, the script is responsible for all of the player's movement controls and menu operations
Toolbox.cs	This script includes all in-game functions of the cube props, such as teleport cube, restore cube, and zoom cube
Touchup.cs	This script controls how all in-game traps work and is only activated when the player touches them
Trainninggroundscrips/lookatcamera.cs	This script allows all in-game prop labels and fonts always to face the player
Trainninggroundscrips/robotbillbord.cs	This script controls all the billboard functions of the in-game robot
Trainninggroundscrips/robotcontroller.cs	This script controls all movement operations of the robot
Trainninggroundscrips/tranninggroundobjects.cs	This script ensures that objects can only appear in the training ground
UI Scrips/Billboard.cs	This script controls the player's billboard and the way all in-game billboards move
UI Scrips/GameInfoItem.cs	Similar to Playerlistitem, except that this one encapsulates the information about game events and automatically sets the destruction time
UI Scrips/GameInfoManager.cs	Control all game events in the game and instantiate them as GameInfoItems and display them to the player
UI Scrips/gamesetting.cs	Responsible for all screen settings
UI Scrips/guidancetext.cs	This script controls all of the player's prompts in the tutorial
UI Scrips/ingameuimanager.cs	This script controls the logic of all in-game UI and the logic of the player heads-up display
UI Scrips/scoreboard.cs	This script encapsulates all the functions and information of the scoreboard
UI Scrips/scoreboardmanager.cs	This script controls the refreshing and sorting of the scoreboard
Wallfoward.cs	This script controls how the damage zone moves
Cameradist.cs	This script controls the distance and focus of the camera while the player is manipulating the sphere
Getspeed.cs	The script is used to assist players in having near-realistic interaction when standing on the teleportation platform

Game menu.

- Players can launch and enter the game quickly
- The game has a dark to light transition animation from near to far
- The start menu will show the main features and style of the game in the background
- The game's start menu is clear and concise and includes the main functions of finding a room, creating a room, training mode, changing the name, selecting a player, and exiting the game
- All menu buttons in the game will have a pressing effect and press sound
- Players can view and change their names through the menu
- Players can select four different game characters and view their details on the menu screen. Each type of sphere has its appearance, quality and speed. The different speed and quality of the game will be directly linked to the damage
- The game has a beautiful and quiet background music
- The player can adjust the soundtrack on/off and volume level, as well as the picture quality on the left side of the menu
- The game's start menu automatically detects whether the player has chosen a character, entered a legal nickname and prompted the player to meet these requirements; failure to do so will prevent the player from playing the game
- The game's menu screen will have various prompts to tell the player what to do or remind the player what this feature is
- The player's name, chosen character, and settings will be saved until the game is restarted
- Players can quickly join and create rooms
- Duplicate player names in the same room or more than the rated number of players will be placed in a room, and the player will be prompted with an error
- All network errors in the game will force the player to go back to the start menu with all errors and will automatically try to reconnect to the server

In-game.

- The player can control the sphere smoothly using the keyboard and mouse. The sphere is rolled by physical torque so that it will be a very comfortable and smooth experience
- Players can control the ball to roll, accelerate, sprint, jump, and brake
- The player can adjust the camera movement and distance very comfortably and avoid the camera through the mould
- The game map has a variety of objects for players to interact. For example, the teleportation platform can transport the player to other places, and the player can jump on the platform and show inertia;
  There will be a springboard in the game. When the player steps on the springboard are automatically bounced into the air;
  There are traps close to the ground colour in the game. The player may be pushed into the air after stepping on it or maybe plunged into the ground;
  There are currently three kinds of special squares in the game. There are three special squares in the game. Green squares represent players disappearing after touching and can increase one chance to respawn;
  White squares represent players who may be zoomed in or out after touching;
  Transparent squares represent players who will be transported to a specific place after touching;
  There is a giant funnel in the centre of the map for players to have fun.
- There are many kinds of vegetation and Terrain in the game, and different ground materials represent different Friction and elasticity
- There are several green spherical recovery areas in the game map, which will appear in complex terrain areas.
- The game has specific sound effects for jumping, rolling, and sprinting, and other players in the field can hear these sound messages in time to assist their judgment
- There will be a red wall of damage at the beginning of the map that keeps moving forward in competitive mode. The wall itself does not collide with the body, but if the player is behind the wall, he will suffer continuous damage. The amount of damage will increase depending on the distance of the wall from the map's midpoint
- Players can collide to damage other players or push players off the map. Each player has five chances to respawn and will change to spectator mode when the player has used up all five chances. In this mode, the player can fly across the map and enjoy the scenery while watching the battle
- Players can check their skill cooldown time and current blood level in time. At the same time, the whole screen colour will change when recovering and suffering damage. Players can also check their current name and blood level hovering above other players in time. Objects will not obscure these billboards

- Players can view the events happening in the game in real-time in the top left corner of the screen. All in-game players join, quit, die, kill will be broadcast to all players on the event board; players can view the names of the players present, the number of kills, and the number of deaths in real-time in the top right corner of the screen, and sort according to this information
- Players can exit the game at any time during the game, and all in-game information will be followed and refreshed
- Players can respawn in the training ground mode, and there is a particular tutorial to guide players and introduce all the mechanics and rules of the game
- In all game modes, players can view all controls at any time by using the tips in the bottom left corner

Game graphics

- Players can choose from 5 different levels of graphics settings
- The game has gorgeous and high-quality low poly graphics
- The player can adjust the camera parameters to achieve a cinematic image effect, such as focal length and aperture.
- The game is optimised for both code and graphics, so it should run smoothly on a good number of devices