School of Computer Science

Game Exhibition Entry

Andrius Sirvys
016319273

April 13th, 2020

A Final Year Project submitted in partial fulfilment
of the requirements for the degree of
BSc (Computer Science and Information Technology)

Contents

List of Figures

1 Introduction

The goal of this project was to develop a game that would enter the Games Fleadh competition, which takes place in LIT Thurles on Wednesday the 4th of March 2020. This project was to be completed either as a team-entry or as a solo-entry. My game was a solo-entry.

1.1 Why I chose the project

Throughout my studies at NUI Galway, I had completed several modules which involved creating games. However, the games always had to be limited in scope and appropriate to the module. By choosing this project, I can create a bigger game with more features. Throughout my life, games have been a big part of it, and now I had an excellent opportunity to create something truly my own. Entering into the Games Fleadh competition was another big appeal of this project. It would offer a chance to showcase what I have made to others as well as meet other students who are passionate about games sharing ideas and maybe someday, who knows, creating a game together.

2 Research

The research to create my game consisted of looking at existing games and what elements were interesting—then seeing how could I combine these elements to make something new, unique, and fun. I also knew that my game had to be online and multiplayer, as games networking was something I always wanted to explore more.

In my preliminary report, I discussed how I wanted to make a cooperative multiplayer game, in which two players have to work together to overcome puzzles and progress through a narrative-driven game. The experience I wanted to create was something similar to games such as Portal 2(1) (multiplayer mode) or A Way Out(2). However, as I further developed this idea, I found that it didn't align with the primary goal of this game which was to be an online multiplayer game. The concept was narrative-driven, and it was for two players only. The game I needed to make had to have multiple people playing online together. The game wouldn't be exciting to play by yourself or with one other person. It would bring groups of people together. What I had decided to create was an online multiplayer first-person shooter.

It would be unique because I would combine the portals element from Portal 2(1) with a first-person shooter. The biggest challenge came with how would I achieve my primary concepts, playing online, and implementing multiplayer. Researching multiplayer in Unity, I found that it had a feature called UNet(3), but it was deprecated and no longer used. As I looked for multiplayer solutions in Unity, I came across a package called Photon(4), it an externally developed and maintained package that helps integrate multiplayer in Unity. After reading the documentation and understanding some of the concepts behind Photon, I decided to use it in my project.

The last research I needed to do was regarding portals. I understood the concept of portals in the real world, but what I needed to understand is how they work inside a game. To understand how portals work inside a game, I found a video by DigiDigger(4) explaining the mathematical and logical concepts on how to create them. With most of the background research that was necessary to start the game completed, it was possible to begin development and work towards achieving the goals.

3 Implementation

3.1 Design Ideas

I chose a multiplayer first-person shooter because it is such an iconic genre. Most people, at one point or another, have either played or experienced this type of game. It has existed and evolved over the years, continuously refined. At a glance, first-person shooters appear simple, but there are countless strategies, possibilities, and play-styles that it accommodates. By creating solid, core gameplay elements, you allow the multiplayer to shine by enabling the player. Of course, it needs some sort of twist. It is adding a new feature to the genre, such as portals, alike to the game Portal 2. Portals would add a whole new level of strategy, allowing the player to place, and go through portals. Using the portals for defence and offence. Combined with being in a maze-like environment, where the arena in which the players play consists of narrow corridors with twists and turns, making the portals become a crucial element of gameplay—another level of strategy.

3.2 Objectives

In hopes of achieving my idea, there were a few key-objectives I had to complete. Additionally, there were a few secondary objectives that would enhance the overall experience and gameplay. The fundamental and primary objectives of my game were as follows:

(1) Play online multiplayer seamlessly. Create lobbies where people can join, as well as discover other players' lobbies.

(2) Be able to view every single player score within a game.

(3) Shooting has to work great as well as feel great. Animations and sound effects.

(4) Players need to be able to see other players, as well as control only their character rather than anyone else.

(5) Be able to place and use portals how expected. Teleport players as well as look like it's a portal to another place in the game.

After these fundamentals, these were some of the secondary objectives which I thought would enhance the game:

- Each player has their own colours. To know which portals belong to which player.
- Being able to shoot bullets through your own portals, as well as throw grenades through portals.
- Multiple different playable characters.
- Able to see your character model by looking through the portal.

3.3 Structure

Here I want to discuss the logical structure of the game before taking a more in-depth look into each element. Since the game is online and multiplayer, I had to be thinking about the chain of command/control in every line of code that I wrote. I used the Photon 3D Networking Framework(5) to help me achieve this.

When executing player movement, primary camera control, shooting, or interacting with the UI, all of these elements had to be controlled only by the respective local player. However, certain features and actions had to be synced across the network to all players so that multiplayer is possible. For example, if I place a portal in the game world, all of the players should be able to see it and interact with it. But they shouldn't be able to spawn or de-spawn it, or even shoot through it.

Now let's look at how all of the elements interact. Starting with scripts that appear in the scene, you can see the relationships indicated in Figure 3.1.

FXManager.cs is used to sync all of the different visual effects across all players. Every single client runs this script locally as it synchronises the different visual and particle effects between players.

UIManager.cs is responsible for the UI elements in each player's game, indicating their health, ammo, and more. Since it is specific and controlled by only the local player, we don't need it attached to every person in the game.

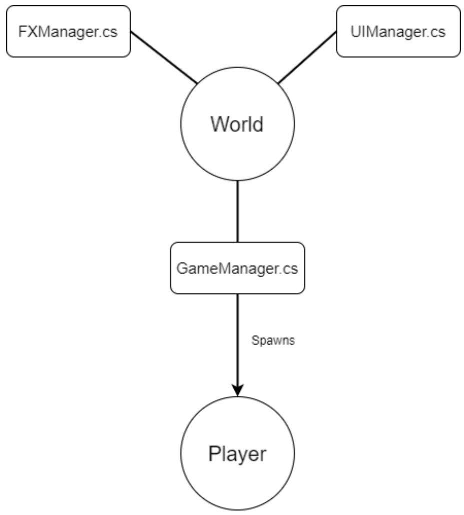


Figure 3.1: Relationship graph, how do these scripts interact.

GameManager.cs is responsible for running and executing the game for the local player. When a local player joins the game, it spawns their character as well as all the other player characters who are in the game.

Every playable character has many different scripts attached to it. The functionality each script is determined by whether the character belongs to the local player or someone else over the network. Looking at the relationship graph referred in Figure 3.2.

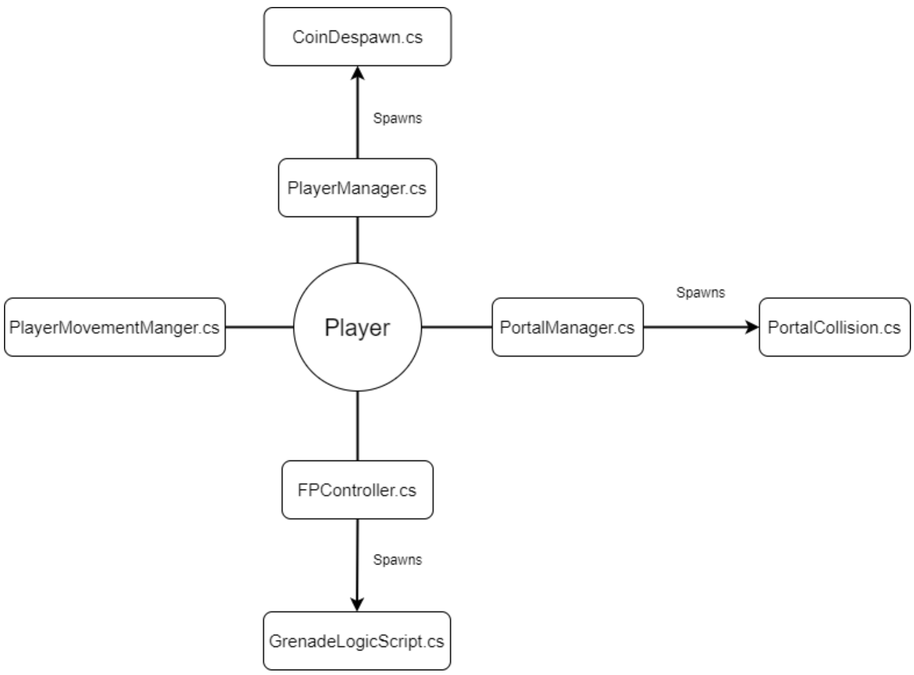


Figure 3.2: Relationship graph, how do these scripts interact.

PlayerMovementManager.cs controls the movements of the local player as well as the sound effects associated with movement. It attaches the main camera to the local player's character while also handling coin pickup and score-points addition.

PlayerManager.cs instantiates the characters spawned by GameManager. It activates the chosen character model by each player as well as configuring first-person models for the local player. It also handles health, death, coin instantiation, and respawning.

FPController.cs gets only executed on the character controlled by the local-player. It controls all of the first-person interactions and animations the local player triggers.

PortalManager.cs controls the portals made by each player. Since every single character has this script attached, it means that it controls two portals per character.

PortalCollision.cs gets spawned when the PortalManager.cs instantiates a portal over the network. It reports collisions to the parent PortalManager.cs script to handle teleportation and other functions.

4 Development

In this chapter, I am going through and explaining how I coded and achieved the different elements within my game. For creating the game, I used the game engine called Unity. Some of the reasons I chose Unity were, that it is relatively easy to get started with, especially as someone who has done a lot of Java as C# is quite similar. In addition, Unity has many built-in systems and functions, such as a rendering pipeline and a physics system.

Since my game is multiplayer, every section covers how the system works for the local player as well as how it interacts and functions over the network. There are a few features and improvements I would like to implement in the future, as well as restructuring some of the code and order of execution as I have a lot more experience now creating a multiplayer game. These features and improvements get discussed in a later chapter, but right now, let's delve into the details.

4.1 Photon

As mentioned before, the networking element of my game uses the add-on package for Unity called Photon 3D Networking Framework(5). Through integrating Photon into my game, I get access to many incredibly powerful features. The features I mostly used from Photon were connecting to the network, lobby creation, player-matching, auto-masterclient switching, instantiating objects over the network, remote client calls, networked custom player properties, custom property synchronisation, transform synchronisation and animation synchronisation. Throughout the report, every Photon feature used gets discussed more-in-depth.

4.2 User Interface

The user interface aims to be simple and straight to the point. I wanted the players to be able to get into a game as quickly as possible.

The player's name gets automatically populated with a random number. However, the player can change the populated name with a name of their choice, as other players see this name in the lobby and during the game. Once completed, they press on "Connect", which connects them to the Photon network. The text found in TextInputField becomes the "NickName" for this player.

The players now see the main menu, which has a few different options.

When the player selects to "Create Room", they are taken to a new menu where they can specify the name of their room. When they press the "Create Room" button, a room using that name gets made. However, if the player left the text field empty or a room under that name already exists, we assign a random name called "Room (random number)".

If the player selects "Join Random Room", they are immediately connected to an existing available room in the lobby. If there are no currently available rooms in the lobby, a room gets created, and the player gets placed in that room.

4.2.1 Show Rooms

The player can browse all the current open rooms by selecting "Show Rooms". There, a scroll-able list view gets shown listing the current open rooms, as well as their current capacity and how many players maximum can play. At the moment, each game can accommodate up to 8 players.

Using Photon, every time there is a new room either added or removed from the lobby, it calls "OnRoomListUpdate" to which it passes all of the current rooms in the lobby. Overriding the Photon method, I insert another method called "UpdateCachedRoomList". Here, for every single room that we receive from Photon, we check if it's currently unavailable/closed and if it's cached. If so, we remove it from our cache of currently available rooms. However, if the room is available and isn't currently cached it's added to the cachedRoomsList.

After running "UpdateCachedRoomList", "UpdateRoomListView" is called. The job of this function is to update the UI to show the available rooms for the player.

```
1 reference
private void UpdateRoomListView()
{
    foreach (RoomInfo info in cachedRoomList.Values)
    {
        GameObject entry = Instantiate(RoomListEntryPrefab);
        entry.transform.SetParent(RoomListContent.transform);
        entry.transform.localScale = Vector3.one;
        entry.GetComponent<RoomListEntry>().Initialize(info.Name, (byte)info.PlayerCount, info.MaxPlayers);

        roomListEntries.Add(info.Name, entry);
    }
}
```

The rooms that were all cached get looped through, and a "RoomListEntryPrefab" becomes instantiated. This prefab contains what you see in the picture above, the layout as well as the buttons and information. The information populates by calling the "Initialise" function within the RoomListEntry script attached to the prefab. Using a prefab for every entry makes it so that the UI can display many rooms. Once the list is too long to fit within a users screen, a scrollbar appears.

4.2.2 Inside Room Panel

When inside a room a similar technique used in Show Rooms is deployed, a prefab called "PlayerListEntry" becomes instantiated. These prefabs are managed and checked by utilising Photon functions which activate when players enter and leave rooms. To provide some customisation to each player the "PlayerListEntry" keeps track of each player and their choices. Looking below, options to select a character and press-ready are only available to the entry that belongs to the local user. Checking if the local player's actor number matches the ownerid set by the "PlayerListEntry".

Utilising an inbuilt Photon feature, I can track what number is the current player in the room. Anytime I create/destroy a PlayerListEntry prefab, I add/subtract to "OnPlayerNumberingChanged".

```
0 references
public void OnEnable()
{
    PlayerNumbering.OnPlayerNumberingChanged += OnPlayerNumberingChanged;
}


public void OnDisable()
{
    PlayerNumbering.OnPlayerNumberingChanged -= OnPlayerNumberingChanged;
}
```

Then depending on their player number, the colour next to their username is changed accordingly.

```
2 references
private void OnPlayerNumberingChanged()
{
    foreach (Player p in PhotonNetwork.PlayerList)
    {
        if (p.ActorNumber == ownerId)
        {
            PlayerColorImage.color = ClipperGate.GetColor(p.GetPlayerNumber());
        }
    }
}
```

As mentioned in section 3.2, one of my secondary objectives was to allow players to choose from multiple different characters. As the game begins, the player moves from the lobby to the arena, and their selected custom properties must be saved somewhere. In addition, to be able to show their chosen character correctly to every other player connected to the game, it means that the properties must be uploaded and accessible to every player in that game.

Photon once again provides functionality to achieve this. Photon allows setting custom player properties, which is a hashtable which gets uploaded and synced to every player connected in that game. When the player interacts with the dropdown menu seen earlier, the function "SelectedCharacter" executes.

```
1 reference
public void SelectedCharacter(Dropdown choice)
{
    Hashtable props = new Hashtable() { { ClipperGate.CHOSEN_CHARACTER, ClipperGate.GetCharacter(choice.value) } };
    PhotonNetwork.LocalPlayer.SetCustomProperties(props);

    print(ClipperGate.GetCharacter(choice.value));

}
```

Creating a hashtable entry, to which the local players custom properties are updated. Once a player selects "READY?" a green tick-mark appears next to their entry, this is also shown to every other player in the room by utilising the custom properties hashtable to sync selections. When all of the players have indicated that they are ready, the start game button appears to the masterclient/owner of the room and the game can be started.

4.3 Static Game Values

Looking at some of the code provided, you can see that the script ClipperGate.cs getting called at various instances. This script helps with Photon custom player properties as well as storing some static values that get accessed throughout the game. It contains string representations for the different keys used in player properties hashtables.

```
27 references
public class ClipperGate
{
    public const float PLAYER_RESPAWN_TIME = 10.0f;
    public const float PLAYER_MAX_HEALTH = 100f;
    public const float PLAYER_START_SCORE = 0.0f;

    public const string PLAYER_HEALTH = "PlayerHealth";
    public const string PLAYER_READY = "IsPlayerReady";
    public const string CHOSEN_CHARACTER = "SelectedCharacter";
    public const string PLAYER_SCORE = "PlayerScore";
```

ClipperGate.cs is also where the different player colours are stored and accessed. Reason being that the colours used to represent the player in the lobby need to be the same colours used for their portals. Lastly, ClipperGate.cs stores the available character models that get used in the game.

```
3 references
public static string GetCharacter(int charChoice)
{
    switch (charChoice)
    {
        case 0: return "SM_Chr_Boss_Female_01";
        case 1: return "SM_Chr_Business_Female_04";
        case 2: return "SM_Chr_Developer_Female_02";
        case 3: return "SM_Chr_Business_Male_02";
        case 4: return "SM_Chr_Developer_Male_01";
        case 5: return "SM_Chr_Security_Male_01";
    }

    return "SM_Chr_Boss_Female_01";
}
```

4.4 Player Instantiation

When pressing the "Start Game" button, the arena scene is loaded using Photon. With Photon, when the masterclient switches scene, all of the other players are synced and switch the scene as well. The GameManager.cs instantiates the player prefab using Photon over the network, appearing on every single player's screen and assigning ownership. The client that instantiates the object over the network is the owner of the said object in the network. This method gets re-used when respawning the player.

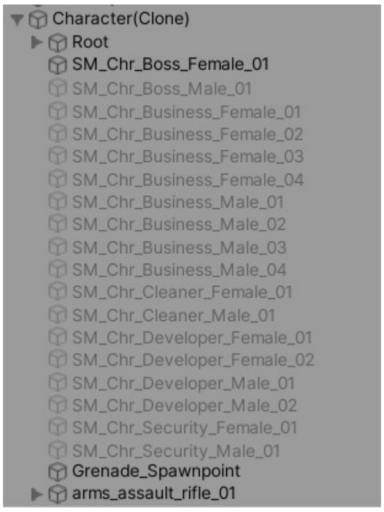
```
2 references
void SpawnPlayer()
{
    if (PlayerManager.localPlayerInstance == null)
    {
        int index = Random.Range(0, spawnpoints.Length);
        PhotonNetwork.Instantiate(this.playerFab.name, spawnpoints[index], Quaternion.identity, 0);
    }
}
```

Once the player prefab has loaded, we activate the PlayerManager.cs finishes the initialisation. In the lobby, the players were able to select a character model from 6 different options. Now we need to load their correct choice onto the character model, so it shows correctly within the game.

```
0 references
void Start()
{
    foreach (Player p in PhotonNetwork.PlayerList)
    {
        if (p.ActorNumber == photonView.OwnerActorNr)
        {
            string characterSelection = (string)p.CustomProperties[ClipperGate.CHOSEN_CHARACTER];
            health = ClipperGate.PLAYER_MAX_HEALTH;
            ConfigureCharacter(characterSelection);
        }
    }
}
```

The list of players is retrieved from Photon and looped through. If the instantiated character object belongs to that particular player, then their character model choice is retrieved from the Photon custom properties. Passing this string value to ConfigureCharacter.

The way the character model is structured is that there are 18 different available models attached to it as well as the first-person view.



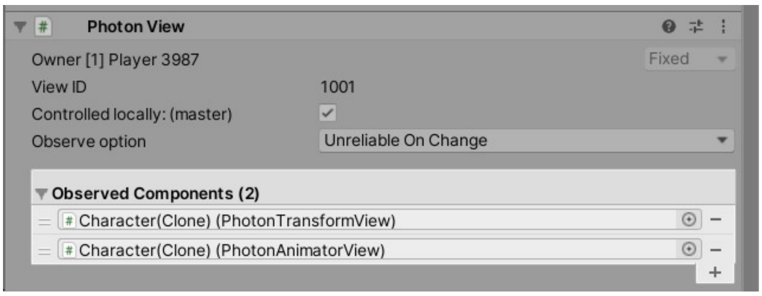
By default, every single mesh render of these models is turned on. The ConfigureCharacter goes through every renderer and checks if it is tagged as "PlayableCharacter" and if the name matches the character selected. If not, the renderer gets turned off.

```
void ConfigureCharacter(string characterSelected)
{
    var characters = this.gameObject.GetComponentsInChildren<Renderer>();

    foreach(Renderer c in characters)
    {
        if (c.CompareTag("PlayableCharacter") && c.name != characterSelected)
        {

            c.gameObject.SetActive(false);

        }
        else if (c.name == characterSelected && photonView.IsMine)
        {

            c.gameObject.layer = 8;

        }
```

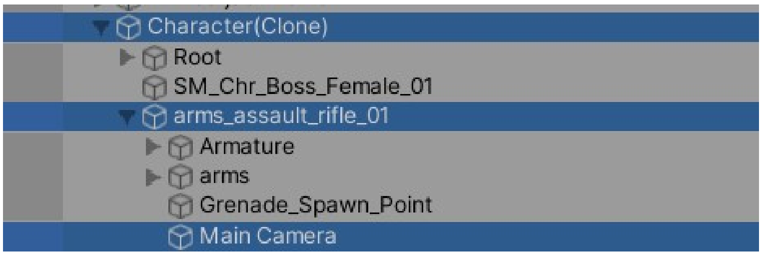
There's an additional step where if a match gets found and it belongs to the local player, the renderer gets moved to layer 8, which is "ThirdPersonView". Lastly, if the character doesn't belong to the local player, "arms_assault_rifle01" is found within the object and the mesh renderer gets turned off. The reason for doing this is that the local player needs to be in a first-person view, as they're playing and controlling the character. However, every single other player needs to be represented by the character model they've chosen. The main camera, which represents the view of the player, is set not to render anything in the layer "ThirdPersonView". Meaning the local player only sees the first person model while everyone else appears in the models they chose in the lobby.

4.5 Movement and Camera

Most of the code related to movement and camera gets only executed on the character belonging to the local player. If the code were to run on every single character, it would cause chaos as players could control each-others characters. I don't need to worry about syncing the player's position and animations over the network, as using Photon Transform and Animation View it handles it for me.



The first thing is to check using Photon if this character belongs to the local-player, if it doesn't the code execution stops. Once the local-player gets established, the main camera of the scene is childed to the first-person view model.



Movement in the world is controlled by either the arrow-keys or "WASD". The character moves relative to the location it's looking to. The view is controlled by the mouse which, in turn, turns the character side to side—allowing the camera and first-person-model to turn along with it. The reason why the camera is childed to the first-person-model is that the camera also looks up and down. When looking up and down, only the first-person model is rotated rather than the whole character. If the entire character were rotated, unexpected behaviour would occur, floating and twisting mid-air.

```
move = (transform.right * x + transform.forward * z) * speed;
move.y = vSpeed;
characterController.Move(move * Time.deltaTime);

float mouseX = Input.GetAxis("Mouse X") * mouseSensitivity * Time.deltaTime;
float mouseY = Input.GetAxis("Mouse Y") * mouseSensitivity * Time.deltaTime;

xRotation -= mouseY;
yRotation += mouseX;

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


fpvModel.transform.localRotation = Quaternion.Euler(xRotation, 0f, 0f);
this.gameObject.transform.Rotate(Vector3.up * mouseX);
```

The character is controlled via a CharacterController. The CharacterController component in Unity is controlled by the player and is not affected by the physics engine. To move the CharacterController, a method called Move gets called. The method Move, moves the CharacterController by the given Vector3, in what direction by how many units. This method is called every Update, which then means the player's movement speed is tied to their framerate. To prevent this, I utilise Time.deltatime, it provides how much time has passed since the previous frame was rendered. Multiplying the Move function by Time.deltatime makes the players framerate independent from the movement speed of the character.

4.6 First Person Mode

The first person mode is required to build and create a first-person game. Having already covered how the first person model gets set up and activated in section 4.4 and how the movement and camera are handled for the characters and first-person view in section 4.5 I will be discussing how the FPController handles the shooting, grenade throwing and portal-placing.

The FPController also controls the animations, sound and fire-rate however I won't be going into the details of these.

Looking into grenade throwing, whenever a user presses "G", we start our chain to throw a grenade. When a user throws a grenade, this grenade must spawn and appear in every single player's game after which it needs to self-destruct. Instead of instantiating the grenade over Photon, I utilise the feature called "Photon Remote Procedure Calls" (PunRPC). PunRPC makes it easy to execute actions for every single player connected to the game. PunRPC only activates the method on the same object that called it from another client or person.

```
    GetComponent<PhotonView>().RPC("ThrowGrenade", RpcTarget.All);

}
```

```
[PunRPC]
0 references
private void ThrowGrenade()
{
    StartCoroutine(GrenadeDelay());
}
```

Since Photon always synchronises the characters positions, it's possible just to execute the code that throws the grenade and let the local client code handle the rest. Once the grenade explodes, it sends out a spherical RayCast to detect which objects it hit, if a player gets hit then using Photon RPC a function called "TakeDamage" is executed over the network. The appropriate player receives damage.

Whenever the player shoots a bullet, a ray is cast from the centre of the screen. The ray is cast directly from the centre if the player is aiming down the sights of the gun, if they're not aiming down, then a small random spread factor gets applied.

```
Ray ray = Camera.main.ViewportPointToRay(precision);
RaycastHit hit;


if(Physics.Raycast(ray, out hit, 1000, layerMask))
```

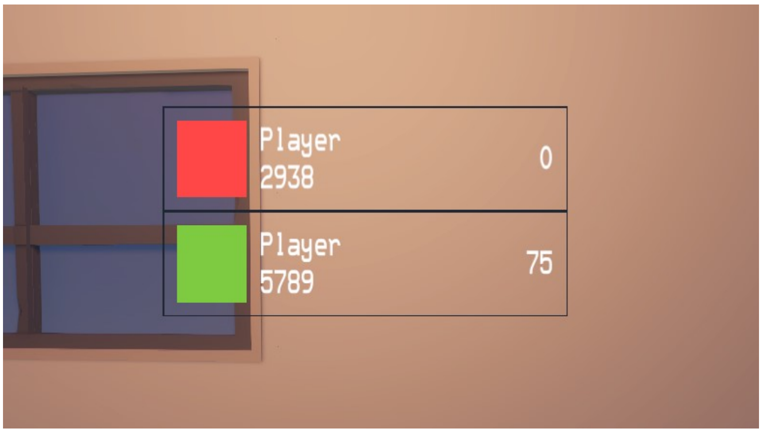
The returned RayCastHit determines the next steps. Comparing tags, it checks if it hit another player. If so, the same method is deployed as when getting hit by a grenade, "TakeDamage" gets executed by Photon RPC on the target. If a portal is hit and we are the owner of that portal, the code to teleport our bullet through the portal is called and executed by the PortalManager.cs, refer to section 4.8.7.

Every single bullet that gets shot, whether it hits anything or not creates a particle effect that makes it look like it was an actual bullet, as well as leaving a mark on the object hit. These effects are all activated through Photon RPC so that each player can see them; I further explain these effects in section 4.9.

Placing portals also follows a similar approach. Whenever the user presses a button to place a portal, we send a RayCast to see where they want to put the portal. Portals can only be placed on objects tagged "PortalPlace". Reason being, to prevent placing portals on furniture and such other odd items which would result in unforeseen circumstances. If it is possible to place a portal, then a Photon RPC call is made to the PortalManager.cs to spawn a portal at this location. Every single person in the game should be able to see other players portals, that's why Photon RPC gets used. Once the portal is in the world, the PortalManager.cs handles the rest of the portal logic.

4.7 Scores

Having established all the gameplay elements, now the players need a way to see who's winning and who's losing. When inside a game, each player can access the scoretable by pressing TAB on the keyboard. The scoretable shows each player's colour, name and current score.



The entries in the table are created and populated using the same principles discussed in section 4.2.2. Every player's score is retrieved from the Photon customer property hashtable and displayed on the scoretable. The scoretables is updated when the player presses TAB to view it, removing unnecessary hashtable lookups.

The PlayerManager.cs tracks the player's health.

```
//only execute the inputs if it's the local player
if (photonView.IsMine)
{
    if (health <= 0f)
    {
        float score = (float)PhotonNetwork.LocalPlayer.CustomProperties[ClipperGate.PLAYER_SCORE];
        if (score >= 25)
        {
            Hashtable newScore = new Hashtable { { ClipperGate.PLAYER_SCORE, (score - 25f) } };
            PhotonNetwork.LocalPlayer.SetCustomProperties(newScore);
        }
        PlayerManager.localPlayerInstance = null;
        PhotonNetwork.Instantiate(coinsPrefab.name, this.gameObject.transform.position, Quaternion.identity);
        GameManager.instance.Respawn(photonView);
    }
}
```

When their health reaches 0, the death sequence is activated. If the player currently has 25 or more points, then 25 points are removed from their score, and the custom property is updated. The localPlayerInstance gets set to null as the character representing the player in the world is destroyed. Over the Photon Network, coins are instantiated and spawned at the location of their death. Finally, the respawn sequence is started by the GameManager.cs. It destroys the character and starts a countdown determined by the "PLAYER_RESPAWN_TIME" set in ClipperGate.cs, after which the player gets spawned again, and the initialisation sequence commences.

The coins are laying on the ground available for any character currently playing to pick them up. Collisions by the character are handled inside PlayerMovementManager.cs. Once the script checks that the collision happened with coins, a Photon RPC is called on the coins to "Destruct".

```
else if (hit.gameObject.tag == "Coins")
{
    hit.transform.GetComponent<PhotonView>().RPC("Destruct", RpcTarget.All);

    float score = (float)PhotonNetwork.LocalPlayer.CustomProperties[ClipperGate.PLAYER_SCORE];
    Hashtable newScore = new Hashtable { { ClipperGate.PLAYER_SCORE, (score + 25f) } };
    PhotonNetwork.LocalPlayer.SetCustomProperties(newScore);
}
```

The reason why the player cannot destroy the coins themselves is that a different player over the PhotonNetwork instantiated these coins. As mentioned previously, the client which instantiates an object over the PhotonNetwork has ownership of it. It would be impossible for this player, who didn't create the coins to destroy it. The "Destruct" function gets executed on every client connected to this game, ensuring that the client who did spawn the coins destroys them as well. The challenge with this approach is that the process is not instantaneous. Since there is quite a bit of back and forward the coins aren't destroyed immediately, and the player who steps on it gets too many points as the collisions keep triggering. I have tried disabling the box collider on the coins once they have encountered a collision. However, it doesn't seem to fix the issue. Will need to research more solutions.

4.8 Portals

One of the biggest challenges I faced was to implement portals. The implementation took a lot of tinkering and testing to get it working as intended.

In most games, and mine included, portals are just a well-disguised illusion. It is not possible to create a true portal inside a game unless it utilises non-euclidean spaces, which most game-engines do not support. You would have to create a custom engine for non-euclidean spaces to exist.

So the question is, what is a portal inside a game. It's usually a flat plane/object that projects the view of a camera attached to the opposite portal, refer to figure 4.1. Using a RenderTexture in Unity to achieve this, the RenderTexture gets attached to the portal plane.

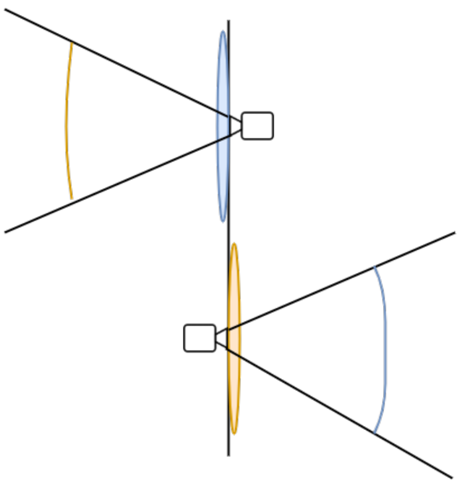
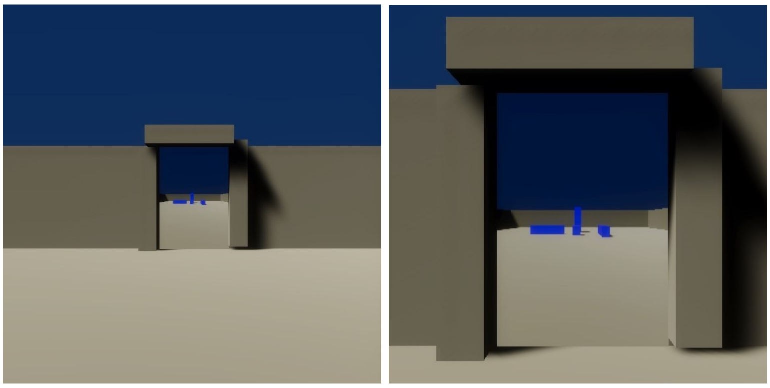


Figure 4.1: Portal and Camera Relation.

However, just this isn't enough. The portal appears to be only an image, utterly static and flat. As you can see below, no matter how far away or to the side the player is from the portal, the image on the portal stays the same—ruining the effect.



4.8.1 Perspective

In achieving a realistic effect, it's needed to know the position and rotation of the player relative to the portal they're looking through. As the player gets closer to the portal, the objects within the portal should appear bigger and closer. Then as the player rotates, they should see the projection also changing to see a different part of the object through the portal. This perspective effect is all done through manipulating the camera attached to the opposite portal.

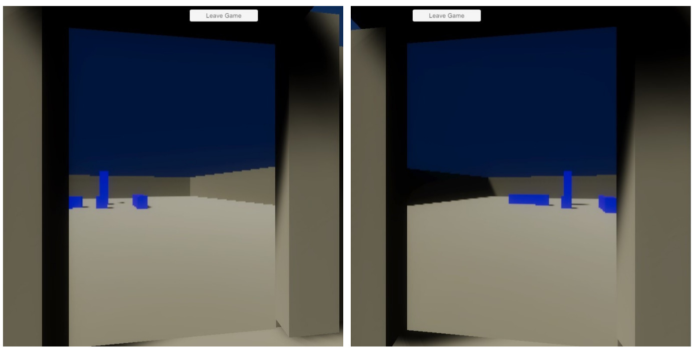
4.8.2 Rotation

To rotate the attached camera of the opposite portal, what I need to do is calculate the player's relative rotation to the portal they are viewing. Then I set this relative rotation to be the rotation of the camera attached to the opposite portal. The reason why it is relative is that the portals don't always face the same direction, their facing direction depends on the surface they are placed. If I simply took my current rotation and made the portal camera mimic me, I would end up with the effect breaking the second it's not a direction I pre-programmed.

To calculate this relative rotation, I take the right vector of the portal the player is looking through, and the forward vector of the main player calculating the angle difference. This difference then tells me the offset by which I must rotate the camera on the Y-axis. Then I need to do a similar thing to offset the portal camera when the player is looking up and down. This time I use the up vector of the portal and the forward vector of the player to calculate the angle.

```
float horizontalDiff = Vector3.Angle(mainPortal.right, cameraTransform.forward);
float verticalDiff = Vector3.Angle(cameraTransform.forward, mainPortal.up);


otherCam.transform.localEulerAngles = new Vector3(verticalDiff - 90, horizontalDiff - 90, 0);
```

I apply this calculated relative rotation to the camera of the opposite portal. The rotation gets applied to the local rotation rather than the world rotation. Each camera is parented to their respective portal, allowing the effect to work no matter the direction.



4.8.3 Position

To achieve a depth effect, as well as side views, the position of the opposite portal camera has to be manipulated. As mentioned earlier, at the moment, the projection looks like it has no depth. It's because the camera attached to the opposite portal always sits directly on that portal, its view never changes as shown in Figure 4.2.

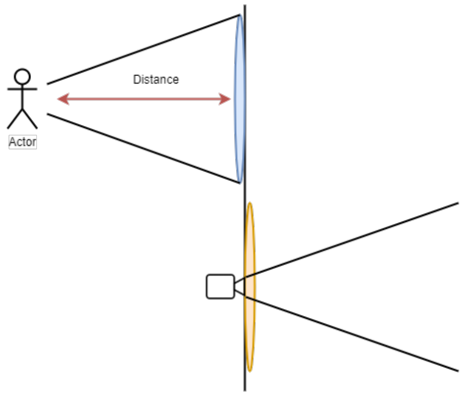


Figure 4.2: Camera sits on the portal, no matter the distance.

Now I calculate the distance of the player, to the portal they're looking through. Then I offset the opposite camera by this distance as shown in Figure 4.3. It's finally starting to give a depth effect to the portal.

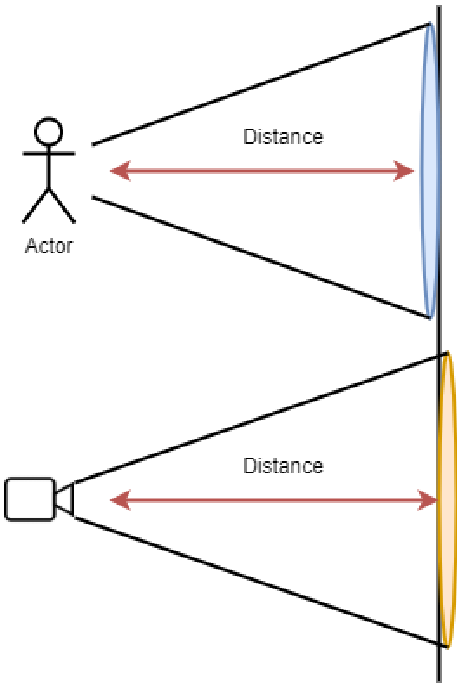


Figure 4.3: The player's distance offsets the camera.

Calculating the straight line distance between the player and the portal won't tell us vertical and horizontal distance, which is needed to be able to move the opposite camera. These distances are calculated relatively rather than on world co-ordinates as the portals are placed in different locations and rotations. To do this calculation, I employ the Law of Sines.

$$\frac{\sin A}{a} = \frac{\sin B}{b} = \frac{\sin C}{c} \tag{1}$$

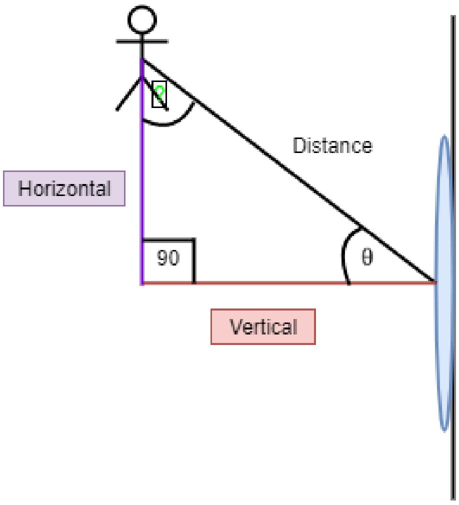
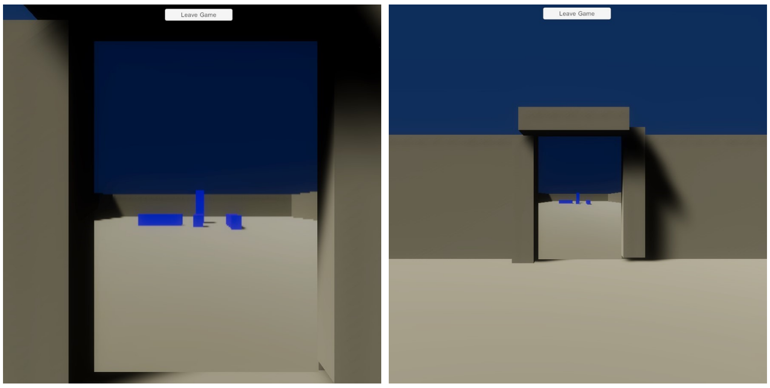


Figure 4.4: Top down view, Law of Sines.

Using the straight line distance between the player and the portal, as well as the forward vector of the portal, I can find θ. Since I know there is always a right angle in my calculation no matter the position; I can find the last unknown angle.

```
Vector3 distance = cameraTransform.position - mainPortal.position;

var heading = cameraTransform.position - mainPortal.position;
heading.y = 0;
var distance2 = heading.magnitude;
var direction = heading / distance2;

float angleY = Vector3.SignedAngle(mainPortal.forward.normalized, direction, mainPortal.up.normalized);

float angleUnknown = 180 - 90 - angleY;

float horiDistance = (distance2 / Mathf.Sin(90 * Mathf.Deg2Rad)) * Mathf.Sin(angleY * Mathf.Deg2Rad);
float vertiDistance = (distance2 / Mathf.Sin(90 * Mathf.Deg2Rad)) * Mathf.Sin(angleUnknown * Mathf.Deg2Rad);



otherCam.transform.localPosition = new Vector3(horiDistance * -1, distance.y, (vertiDistance) * -1);
```

Having all the required information, I use Law of Sines to get the Horizontal and Vertical distance of the player from the portal. Finally, I transform the opposite portal camera by these distances. The transformations are in the local world space as the camera has to move relative to its portal. The distances are also inversed because the portal camera has to be behind the portal rather than in front. As the player moves back and forward, side to side, or jumps, there is a feeling of some depth.



4.8.4 Viewport

Very close to achieving the desired effect, however one major challenge remains. The cameras have a viewport that is much larger than the size of the portals. Right now it's displaying everything they see onto our portal, which ruins the depth effect and makes the portals feel not right. The portal plane has to display only part of the camera's viewport, the part that matches its size. Referring to Figure 4.5, it shows the cameras full viewport, with the white overlay representing the size of the portal and what should be seen instead.

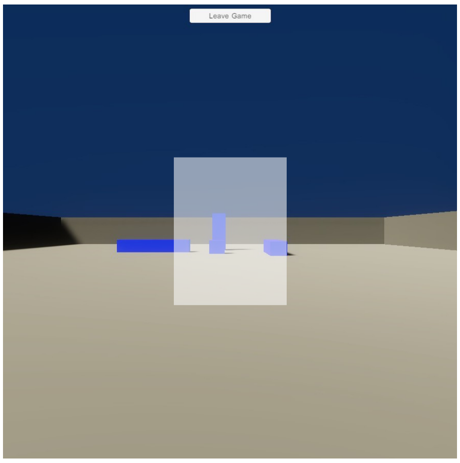
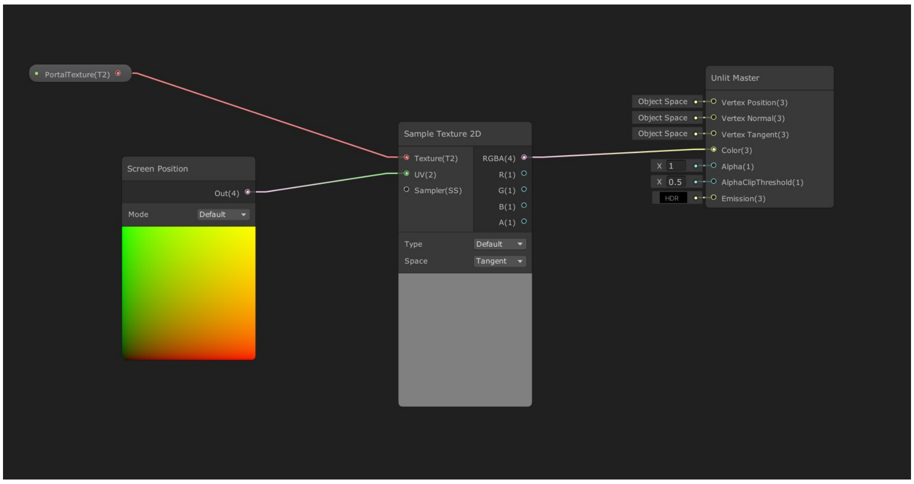
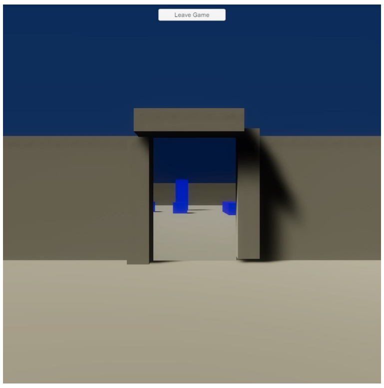
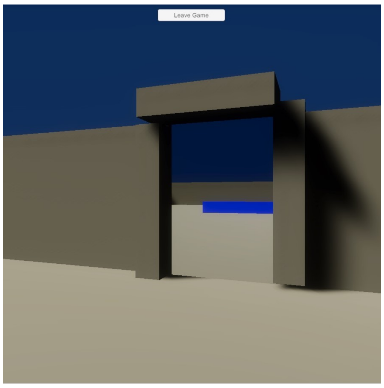


Figure 4.5: Full camera viewport, overlaid with the desired viewport

t this point, I utilised a Unity feature called Shader Graph. Shader Graph allows someone with no shader coding knowledge to create a custom shader they can utilise. This custom shader is attached to the portal plane, and the RenderTexture is given as an input. The shader uses the position of the portal in screenspace to remap the rendering UVs onto the RenderTexture.







Now the portal shows the correct viewport on its mesh, which is relative to the size of the portal.

4.8.5 Clipping Plane

There is one obstacle remaining. When a portal gets placed, since the camera is behind the portal, the surface that the portal sits on blocks the view of the camera. What needs to happen is the camera needs to adjust its near clip plane to the plane of the portal, not rendering anything between it and the portal. However, the camera moves side to side, so this clipping plane isn't always perpendicular to the forward vector of the camera. A technique called Oblique Projection is utilised. Using the example and explanation from Tom Hulton-Harrop(6), I was able to implement a clipping plane. The portal effect is fully realised.

4.8.6 Teleporting Players

The portals are made from two meshes; there's a mesh around the border of the portal and another mesh on the inside of the portal. For a player to get teleported, they have to collide with the inner mesh. If there were no outside border to the portal, the player would be able to touch any part of it and instantly be teleported. The teleportation would be jarring and ruin the effect as they get placed in front of the portal. What the outside collider allows is to funnel the player to the centre of the portal, where after teleportation they're in the correct position, and the effect looks seamless.

The player gets teleported, it is executed during a FixedUpdate because of the CharacterController component. If teleporting the player was executed inside Update, the player would simply not teleport, or they would fall through the floor because of collider conflict. Unity executes physics updates only during a FixedUpdate, which isn't run every frame like Update. The player is placed slightly in front of the portal, as not to touch it and activate teleportation again, facing the same relative direction as they entered.

4.8.7 Shooting and Grenades

Shooting through a portal has few calculations as well. When talking about shooting, what I mean is casting a ray, as discussed earlier in how shooting is handled. Since the portal can be placed on any surface, the angle and position at which the ray hit the portal has to be calculated in the portal's local space. Doing it in local space makes it possible to cast a new ray at the location of the other portal with the correct position and direction. If these calculations weren't relative, then the new rays would not behave as expected.

In getting the relative direction that the ray hit the portal, the direction is converted into local co-ordinates of the hit portal.

```
Vector3 localDirection = tempPortalHit.transform.InverseTransformDirection(directionForward);
localDirection.z *= -1;
localDirection.x *= -1;
Vector3 transformedDirection = tempPortalOpposite.transform.TransformDirection(localDirection);
```

The z-direction is also inversed. The inversion is because the ray travels backwards in the portal's local space. However, when the ray exits the opposite portal, it needs to be travelling forward. A similar thing happens to the x-direction; it gets inverted as otherwise the ray won't be flipped. Once the calculations are complete, the new direction is found by converting the local co-ordinates back to world co-ordinates relative to the opposite portal. The local co-ordinates work because both portals are the same, they're the same prefab so all the calculations can be applied to either.

Now that the direction is found, the position at which a ray is created on the opposite portal needs to be known as well. The same process repeats as with calculating the relative direction.

```
Vector3 distance = cameraTransform.position - mainPortal.position;

var heading = cameraTransform.position - mainPortal.position;
heading.y = 0;
var distance2 = heading.magnitude;
var direction = heading / distance2;

float angleY = Vector3.SignedAngle(mainPortal.forward.normalized, direction, mainPortal.up.normalized);

float angleUnknown = 180 - 90 - angleY;

float horiDistance = (distance2 / Mathf.Sin(90 * Mathf.Deg2Rad)) * Mathf.Sin(angleY * Mathf.Deg2Rad);
float vertiDistance = (distance2 / Mathf.Sin(90 * Mathf.Deg2Rad)) * Mathf.Sin(angleUnknown * Mathf.Deg2Rad);



otherCam.transform.localPosition = new Vector3(horiDistance * -1, distance.y, (vertiDistance) * -1);
```

This time only the x-axis gets inversed, this is because in a portal the x-axis is flipped for when you exit. The slight addition to z is to prevent the new ray hitting the portal itself, getting placed slightly more forward.

Grenade teleportation follows similar logic, except that teleportation is only executed within FixedUpdate to prevent odd physics behaviour.

4.8.8 Performance

Cameras and RenderTextures take up many resources. The more portals on the scene, the higher the performance hit. With with just a few working portals, it can make it impossible to play on lower-end machines, even though there is nothing visually intensive. To improve performance, I had to implement some conditions. The most apparent step was to stop the portal effect working when the player isn't looking at it.

```
1 reference
bool VisibleFromCamera(Renderer renderer, Camera camera)
{
    Plane[] frustumPlanes = GeometryUtility.CalculateFrustumPlanes(camera);
    return GeometryUtility.TestPlanesAABB(frustumPlanes, renderer.bounds);
}
```

Looking at an extension script, created by Michael Garforth (7) from Unity, I was able to implement a way to detect if a particular renderer was visible from a camera. When I call to update the portal effects, what I first do is check if the player can see that portal.

```
if (!VisibleFromCamera(mainPortal.GetComponent<MeshRenderer>(), cameraMain))
{
    if (otherCam.targetTexture != null)
    {
        otherCam.targetTexture.Release();
        otherCam.enabled = false;
    }
    return;
}
if(otherCam.enabled == false)
{
    otherCam.enabled = true;
    CreateRenderTexture(otherCamTex, otherCam, mainPortal, border);
}
```

If the portal isn't visible, then the RenderTexture is removed from the camera's output and is released. The camera is also disabled. The second the player's camera sees the portal again, if the camera was disabled, it gets re-enabled, and a new RenderTexture is assigned.

These few steps helped improve the performance of the game significantly. A two-fold improvement in performance on lower-end machines. However, there is more room for improvement.

4.9 Effects

FXManager.cs handles the different particle effects within the game. Since the shots fired by players need to be visible to all of the connected players, the effects are all activated via Photon RPCs. Let's walk through the effect of shooting another player.

```
[PunRPC]
0 references
private void ShotPlayer(Vector3 origin, Vector3 hitPoint)
{
    var ParticleEffect = Instantiate(Prefabs.playerImpact, hitPoint, Prefabs.playerImpact.transform.rotation);
    ParticleEffect.transform.LookAt(origin);

    Vector3 dir = (hitPoint - origin).normalized;
    var BulletEffect = Instantiate(Prefabs.bulletPrefab, origin, Quaternion.identity);
    BulletEffect.GetComponent<Rigidbody>().velocity = dir * 200;


    var emitParams = new ParticleSystem.EmitParams();
    emitParams.position = origin;
}
```

When the Photon RPC is executed, we get two Vector3 points. Where the shot came from and where it hit. The origin is always the tip of the gun rather than the centre of the screen. The impact effect is shown by instantiating it at the hitpoint and turning it to face the origin point. For the bullet effect, the direction is calculated by subtracting the origin from the hitpoint. The bullet effect prefab is instantiated at the origin, and the velocity is increased in the direction of the hitpoint. Making it look like an actual bullet got shot.

There are a few variations depending on which objects got hit, but all of them follow a very similar structure.

5 Testing

Gameplay testing is a core principle in creating any video game. It gets used to test if a game concept is fun, get feedback from users and discover bugs. By getting people to play the game, they encounter new situations and offer a different perspective which can help in creating the game.

Testing was a core part of ClipperGate. Since this game is an online multiplayer experience, there are a lot of behaviours and bugs I wouldn't notice by play-testing the game myself. Once there was a solid foundation to ClipperGate, I enlisted my friends and family to join and play. I recorded their feedback as well as the unexpected behaviour that appeared during gameplay.

Here are some of the bugs that got discovered during gameplay testing:

1. Players could remove other peoples portals from their local session.
2. When teleporting through portals, other players to the local-client appeared as sliding across the arena.
3. Picking-up dropped coins would award more than 25 points. Sometimes 100 or more.
4. Performance impact of portals on low-end devices, especially many portals.
5. The arena was missing some walls/floors and falling off the map.

Most of the bugs found during gameplay testing, I would not have been able to discover myself, as they relate to networking. Removing other's portals was fixed by checking ownership of the portal before removal. A player receiving more points than expected occurred because destroying and object over the network isn't instantaneous. This bug was fixed by disabling the attached collider once a collision registered. Some performance improvements were added in as mentioned in the portals section. I edited the level to include some walls/floors.

Some bugs and issues I wasn't able to fix within my given time. Doing more testing and fixing bugs is something I would like to come back to.

6 Conclusion

6.1 Lessons Learned

Having worked with Unity before I had some knowledge of the fundamental principles and functions. This game was much broader in scope than anything I had created in Unity previously. To help me track my progress, as well as create backups, I enabled Unity Collab for the project. With Unity Collab, it was possible to utilise version control. Every time I upload my changes to Unity Collab, I include a commit message which describes that has changed. Later if I need to revert to an earlier version because something went catastrophic, there are multiple restore points available in Unity Collab. Version control was a lifeline when I was tinkering with Lighting, and it went completely wrong.

Having good foundations for the game is essential. To try and pre-plan the structure and logic of the game as much as possible. Once there are many features, it becomes increasingly difficult to change the core structure. One of the best strategies is to try and split everything into individual components that can be pieced together rather than having something more monolithic. It gives the freedom to adjust and restructure your game.

6.2 Objectives Achieved

Looking at the objectives set out in section 3.2, these are the objectives achieved.

(1) Play online multiplayer seamlessly.

(2) Be able to view every single player score within a game.

(3) Shooting has to work great as well as feel great. Animations and sound effects. (Half-achieved)

(4) Players need to be able to see other players, as well as control only their character rather than anyone else.

(5) Be able to place and use portals how expected. Teleport players as well as look like it's a portal to another place in the game. (Half-achieved)

The secondary objectives achieved were:

- Each player has their own colours.
- Being able to shoot bullets through your own portals, as well as throw grenades through portals.
- Multiple different playable characters.
- Able to see your character model by looking through the portal.

All of the essential objectives which make up the core of the game got completed. Given the time constraints, I am quite happy with what I achieved. Some of the objectives I didn't implement to my satisfaction; nevertheless, they can always be improved. The secondary objectives also made it into the game; these were smaller features but improved the experience of the overall game.

6.3 Future Work

I would not consider this game finished yet. There is definitely a lot of room for improvement and with more work and polish could become a really fun game. Some of the specific features that still need tweaking and fixing would be the portals. At the moment they can only be placed on vertical-flat-surfaces. This limits their usability and overall fun. Another issue is that the player isn't correctly rotated after teleportation. After teleportation the player is facing forwards from the direction of the portal. If the player enters the portal backwards, when they teleport they should be facing the opposite portal directly, however they get rotated to face forwards and it ruins the portal effect.

When teleporting through portals there is still more work to be done to make it seamless. At the moment the player appears on one side, and the re-appears on the other. For a complete portal effect, the player should be able to stand in the middle of the portals. To do this would need to thinker more with colliders as well as spawning objects to mimick appearance.

Since now I have quite a bit more experience with Photon there are some elements of my game that I would like to change networking wise to be more optimal and function better. One of the bigger issues I face using Photon is that I rely on Photon Transform View, which handles syncing the different player positions across the network and game. The reason being is when a player, who is controlled by someone over the network is teleported by the portals, they just slide from the one portal to the other. To the local player it looks like they teleported but to everyone else they slid across the map. From what I understand this occurs because Photon Transform View interpolates the position of the characters between the position points it receives. Making them appear to slide across the map. I hope that I could create my own solution to sync the player positions to avoid this interpolation when using portals.

Would be great to improve the level design as well as the animations and sounds. Due to the limited amount of time I had, none of the assets used in the game were created by me except for the Game Logo. To have assets made specifically for this game and have them all fit together thematically. To have more time to explore level design and learn about key principles.

6.4 Final Thoughts

1. Creating a game is very difficult. It needs lots of time and patience. The amount of elements that go into it require so many different skill-sets and people coming together. In this project I focused mainly on features and programming, but as I was creating it bit by bit I wanted it to become bigger and better. Having my own custom characters for the game, creating unique animations to match the feel, making objects and structures to create and awesome game environment, music and sounds to create an atmosphere. It really makes me want to be part of a team and create something, as it would be even more ambitious than what I could achieve myself.

2. Creating a game is fun, it's hard work but it's fun. It takes time to perfect a certain element to be exactly the way you desire, but when it works you feel great. My favourite part of this experience was when I got my friends to play my game, and also in the game competition in LIT Thurles when people would come up and try it out. Was it perfect? No, but everyone really enjoyed it and could see the potential in it. It really made it feel that it was worth making.

3. An online game has to be made from the ground up. It was a lot of trying and testing to understand how the different networking components intertwine. Who gets control and who doesn't, how to best sync everyone up so it's an equal experience. The more features and functions I added to the game, the easier it became

to understand and think from a multiplayer perspective. There are definitely still some features of Photon I haven't mastered or fully understood but I feel a lot more confident with it.

7 Assets

These are the graphical/art/sound/animation assets that were used within my game:

- David Stenfors. Low Poly FPS Pack - Free (Sample). `https://assetstore.unity.com/packages/3d/props/weapons/low-poly-fps-pack-free-sample-144839`.
- Synty Studios. POLYGON - Office Pack. `https://assetstore.unity.com/packages/3d/props/interior/polygon-office-pack-159492`.
- dustBUST. Polygon Power. `https://www.dafont.com/polygon-power.font`.
- Dan Mecklenburg Jr. DEC Terminal Modern. `https://www.dafont.com/dec-terminal-modern.font`.
- Unity. Making Portals with Shader Graph in Unity! (Tutorial). `https://youtu.be/TkzASwVgnj8`.
- Jean Moreno (JMO). War FX. `https://assetstore.unity.com/packages/vfx/particles/war-fx-5669`.
- Innovana Games. Hand Painted Seamless Wood Texture. `https://assetstore.unity.com/packages/2d/textures-materials/wood/hand-painted-seamless-wood-texture-vol-6-162145`.
- Devtoid. Gold Coins. `https://assetstore.unity.com/packages/3d/props/gold-coins-1810`

Bibliography

[1] Valve. Portal 2, April 2011.

[2] Hazelight Studios. A way out, March 2018.

[3] Unity. Unet. `https://docs.unity3d.com/Manual/UNet.html`.

[4] DigiDigger. How were the portals in portal created? | bitwise, 2017. `https://www.youtube.com/watch?v=_SmPR5mvH7w`.

[5] Photon. Photon. Computer Software. `https://www.photonengine.com/pun`.

[6] Tom Hulton-Harrop. unity-portal-rendering, 2015. `http://tomhulton.blogspot.com/2015/08/portal-rendering-with-offscreen-render.html`.

[7] Michael Garforth. Isvisiblefrom, 2011. `http://wiki.unity3d.com/index.php?title=IsVisibleFrom`.