# Report for Defence 1
# Aperture


SCOTT TALLEC"α"

ABHISHEK BOSE"β"

RAJAT JOHN"Ω"

SOFIANE BEKHAT "γ"

# Contents

# 1 Introduction

Since the creation of our group at the start of the year, we have been impatient to work on our project. So we decided to tackle it the most as soon as possible, to have a certain advance on our forecasts, on the one hand because we are very motivated, on the other hand because by getting ahead, we will be easier to add more and more content to our game.

Our game therefore has the functional basics and is now playable. Of course, this is still a draft and is far from finished. This is why we are not going to stop there and want to propose a game as complete as possible.

Finally, our group is very close-knit. We get along very well, we often group together to work, or discuss (very often of the project, by the way!). We have worked tirelessly and believe we are ready for the defense.

Here are the details of the work accomplished so far, with the progress done, and the work to be done until the next defense.

# 2 Advancement in the project

## 2.1 Level Design and Graphics

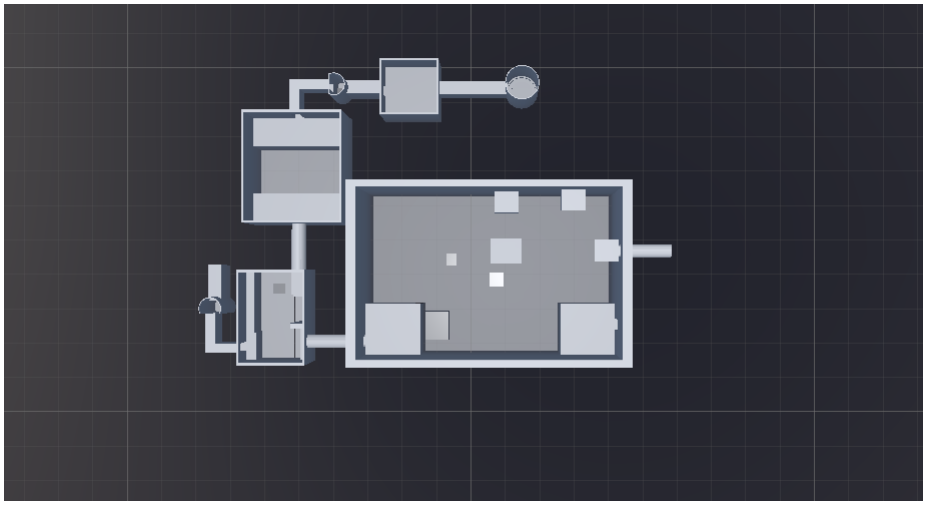
In this part, we will illustrate the main idea of the level design of the game and how it is structured. We will talk about the graphics of the game and explain the code behind the user interface.

Main concept

The level design and the graphics of a game is very important, In fact it is the first thing that one sees in a game and especially in our case , a portal type game where the design of a level is a crucial element for the game in order to make solvable puzzles.

The main idea is to have different levels with a lot of variety and different level of difficulty, that will increase throughout the game.Therefore, the first level is a tutorial level, where the player can learn the controls of the game and also its mechanics. The next levels will be designed according to the puzzles that will be implemented for the game.

The structure of the tutorial level



This level is short.In fact, as it is said before, it is only a tutorial level that will help the player to get familiar with the game commands. It starts from a tall cylindrical building (on the top right of the map) , where the player is spawned. Then the player will wonder through some corridors and different rooms (specifically four rooms). The rooms are different , each of them is

designed for a purpose. The rooms are made by using a powerful modelling software, Blender. Furthermore, there are two room (bottom two) that have movable platforms, and these platform are made by using an animation that makes the platform moving from one side to another and also a script that allows the player to attach on the platform while the platform is moving. Here is the script :

```
public class Player_attachement : MonoBehaviour
{
    public GameObject Player;

    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject == Player)
        {
            Player.transform.parent = transform;
        }
    }
    private void OnTriggerExit(Collider other)
    {
        if (other.gameObject == Player)
        {
            Player.transform.parent = null;
        }
    }

}
```

Graphics

For the graphics, we chose to build the game in a HDR environment, given by unity, that allows to render all the assets and textures in HD, and it also comes with some high quality lightning effects and also with ambient occolusion and volumetric fog, which all will be used later on the game.

## 2.2 User Interface

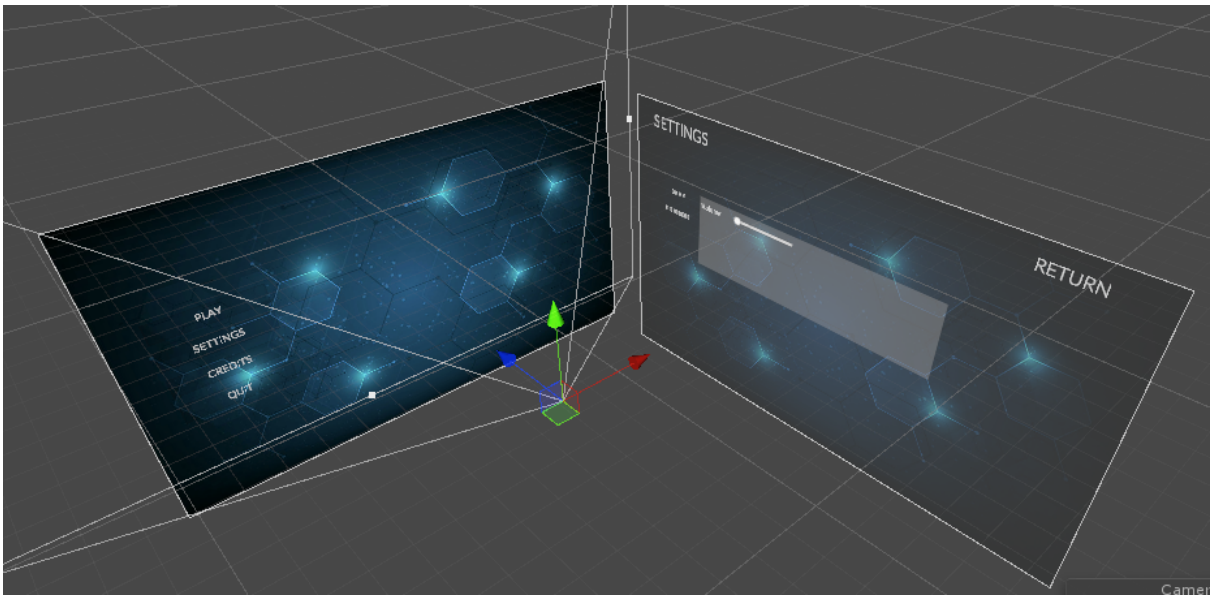
Designing a good user interface is not that easy as we presumed. In fact , what we wanted is a responsive and a modern looking User Interface, and with this menu it is, or at least, it will be. When the single player button is clicked , the first level of the game loads.

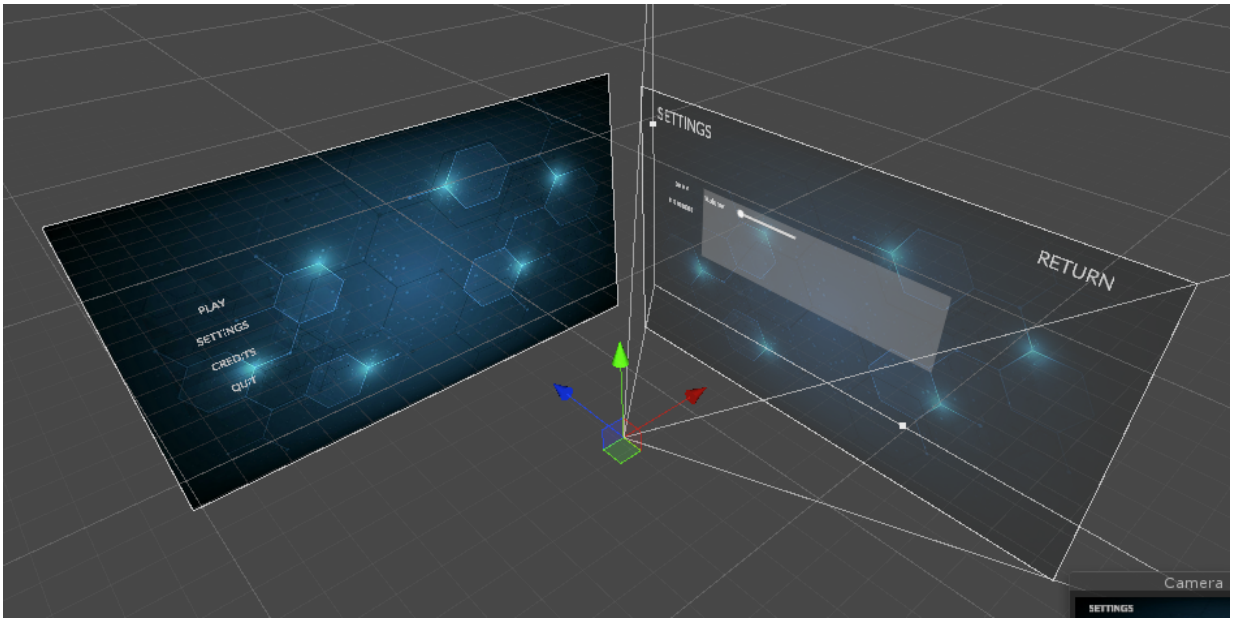

the Main menu is is composed of four buttons :
Play button :

Once one clicks on the button , a sub menu pups up. With the sub menu, the player has a choice to play on single player mode or multiplayer mode.

Settings Button :

This button allow to move from the main menu to the the setting menu.
This button , perhaps, is the most difficult button That we had to implement so far, because it involves the so called camera transition. Camera transition is an animation that moves the camera from one point to another.In this case, the points are the main menu and the setting menu.




First of all , we had to animate the camera , so we can have it moving from one menu to the other.Then Implement a code that allows me to call the animation. This is the following code :

```
public void camera_position_2()
{
    not_play();
    Camera.SetFloat("Animate", 1);
}
```

Animate is the name of the parameter and 1 is the value that triggers the animation. In fact , this animation is triggered when the parameter value is above 0,5.In this case, the value is 1, so the animation will be executed and we will have a transition between the two menus.

Credits Button :

This button allow to jump from the main menu to the credits menu, in which we show all the information about the game and the developers who made the game.

Quit Button :

This button allows you to close the game. Once this button is clicked, a small window pups up. When the window pups up, you have the choice to quit or not the game (by pressing No , the game will not be closed, by pressing yes, the game will close).

There is one thing to point out , which is the fact that when the quit button is pressed , the rest of the buttons will be disabled.

In the following page, you will find the code of the main menu :

```
    public class main_m : MonoBehaviour
{
    public Animator Camera;

    //Panels

    public GameObject main_panel;
    public GameObject sub_panel_play;
    public GameObject sub_panel_settings;
    public GameObject single_mode_button;
    public GameObject multi_mode_button;
    public GameObject sub_panel_quit;
    public GameObject yes_button;
    public GameObject no_button;
    public GameObject game_panel;
    public GameObject control_panel;

    //Buttons

    public Button play_button;
    public Button setting_button;
    public Button credit_button;
    public Button quit_button;

    //Functions

    public void play()
    {
        if (sub_panel_play.activeSelf && single_mode_button.activeSelf && multi_mode_button.act
        {
            sub_panel_play.gameObject.SetActive(false);
            single_mode_button.gameObject.SetActive(false);
            multi_mode_button.gameObject.SetActive(false);
            return;
        }

        sub_panel_quit.gameObject.SetActive(false);
        sub_panel_play.gameObject.SetActive(true);
        single_mode_button.gameObject.SetActive(true);
        multi_mode_button.gameObject.SetActive(true);
    }

    public void not_play()
    {
        sub_panel_play.gameObject.SetActive(false);
        single_mode_button.gameObject.SetActive(false);
        multi_mode_button.gameObject.SetActive(false);
    }

    public void camera_position_2()
    {
        not_play();
        Camera.SetFloat("Animate", 1);
    }

    public void camera_position_1()
    {
        Camera.SetFloat("Animate", 0);
    }
```

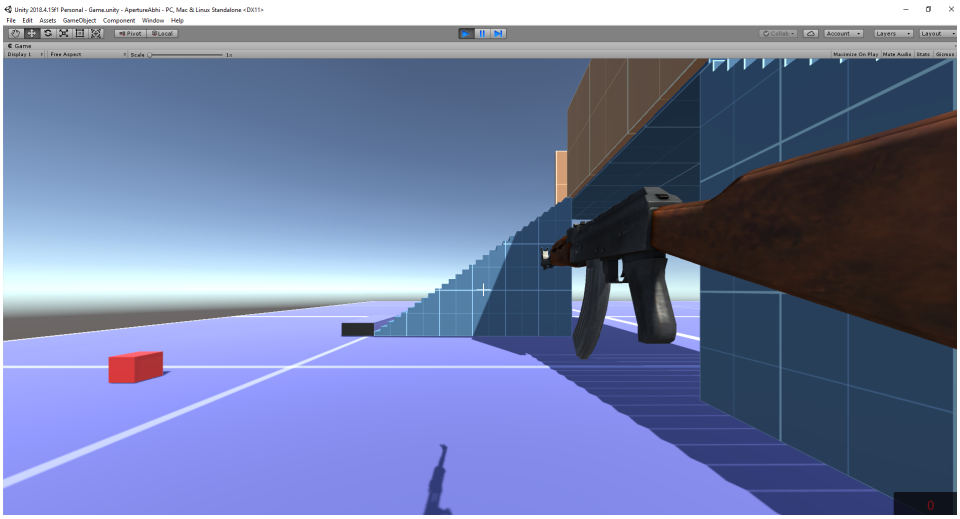
```
public void Quit()
{
    not_play();
    sub_panel_quit.gameObject.SetActive(true);
    yes_button.gameObject.SetActive(true);
    no_button.gameObject.SetActive(true);
    play_button.interactable = false;
    quit_button.interactable = false;
    setting_button.interactable = false;
    credit_button.interactable = false;
}

public void answer_yes()
{
    Application.Quit();
}

public void answer_no()
{
    sub_panel_quit.gameObject.SetActive(false);
    yes_button.gameObject.SetActive(false);
    no_button.gameObject.SetActive(false);
    play_button.interactable = true;
    quit_button.interactable = true;
    setting_button.interactable = true;
    credit_button.interactable = true;
}

public void LoadScene()
{
    Application.LoadLevel(1);
}
```

Finally, we look into the HUD. The HUD is used to display useful information to the player. For now, we will display the cross-hairs to aim the gun and an ammo counter.



## 2.3 Gameplay

### 2.3.1 Movement

The important aspect of a game, is how the camera rotates, specially in an FPS game. I did not find it hard to implement that apart from understanding Quaternion angles, and how it works.

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

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

    transform.localRotation = Quaternion.Euler(xRotation, 0f, 0f);
    playerbody.Rotate(Vector3.up * mouseX );

}
```

As we all know without the movements, a game is as good as dead fish. Therefore, we implemented the basic controls of the player like jump, moving in all in directions. The controls are QWERTY based. In our code, We are trying to get the horizontal axis input and vertical axis input. Along, with that, we are checking, if the user has pressed the "LeftShift". If yes, the player will move around with a greater speed.

```
if((Input.GetButton("Vertical")||Input.GetButton("Horizontal"))
   &&Input.GetKey(KeyCode.LeftShift))
{
    float x = Input.GetAxisRaw("Horizontal");
    float z = Input.GetAxisRaw("Vertical");
    Vector3 move = transform.right * x + transform.forward * z;
     Vector3 _velocity = move.normalized * sprint;
    _rb.MovePosition(_rb.position + _velocity * Time.fixedDeltaTime);
}
if((Input.GetButton("Vertical")||Input.GetButton("Horizontal"))
   &&!Input.GetKey(KeyCode.LeftShift))
{
    float x = Input.GetAxisRaw("Horizontal");
    float z = Input.GetAxisRaw("Vertical");
    Vector3 move = transform.right * x + transform.forward * z;
    Vector3 _velocity = move.normalized * speed;
    _rb.MovePosition(_rb.position + _velocity * Time.fixedDeltaTime);
}
```

As one can see in the code, after getting the inputs from the user, I am using the the method MovePosition to move around. It's a method from RigidBody class

For these basic controls, I could have used CharacterController class as well, but there is a method called AddForce in RigidBody. I wasn't able to find an equivalent method to AddForce. I use AddForce for jumping purpose.

### 2.3.2 Jumping

Tutorial level:

For jumping in the tutorial level, we are using the AddForce method from RigidBody class. We are checking if the user has pressed the button "Space", if yes, the player jumps. Now you might be wondering, just by that, the user can jump multiple times. But I have variable which is of bool type. So, whenever it's colliding with a collider which are tagged by "Ground", my variable gets converted to true which means, the player is actually on the ground. So, the player can jump again.

```
_rb.AddForce(Vector3.down -
            downwardForce * Time.fixedDeltaTime, ForceMode.Impulse);
if (Input.GetButton("Jump") && isGRounded)
{
    _rb.AddForce(Vector3.up +
                thrusterForce * Time.fixedDeltaTime, ForceMode.Impulse);
    isGRounded = false;
}
private void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.CompareTag("Ground"))
    {
        isGRounded = true;
    }
}
```

For coming back, apparently the gravity wasn't enough to bring the player back with optimum speed. So I had one line of code to deal with that.

### 2.3.3 Game mechanics

The mechanics of the game is something that we want perfected as a puzzle-platformer must include some aspects of an FPS game and also manage to include the mechanics of power-ups to finish the various levels.

While doing our research on the implementaion of shooting, we discovered that there are two types of shooting, namely, prefab and raycast. prefab, although easier to implement, did not have the versitality and speed required to keep up with our vision for the game. Hence, we settled on raycast.

We learnt that in the raycast method, the gun does not actually "shoot" anything. Instead it draws a ray between the game object and the object it is facing and uses the information provided by the ray to make changes. For example, the length of the ray can tell the gun how far away the target is and then, depending on the script, tells the gun what to do next.

Code

```
public class gundamage : MonoBehaviour
{
    public int dmg = 20;
    public float targetdis;
    public float range = 15.0f;
    void Update()
    {
        if (Input.GetMouseButtonDown(0))
        {
            if (Ammo.currentammo > 0)
            {
                RaycastHit shot;
                if (Physics.Raycast(transform.position,
                 transform.TransformDirection(Vector3.forward), out shot))
                {
                    targetdis = shot.distance;
                }

                if (targetdis < range)
                {
                    shot.transform.SendMessage("dmginflict", dmg);
                }
                Ammo.currentammo -= 1;
            }
        }
    }
}
```

Here, we can see that as soon as the ray hits a target, it will run the method "dmginflict" with "dmg" as the parameter present in the script of the object that the ray hits. Change in the ammo can be observed which will be explained later on in this section.

Moving on to the AI and the bot, we need something simple for the first version. It is basically a moving target for the player to shoot.

Code

```
public class AI : MonoBehaviour
{
    public float movespeed = 3f;
    public float rotspeed = 100f;
    public bool iswandering = false;
    private bool rotL = false;
    private bool rotR = false;
    private bool iswalk = false;
    void Update()
    {
        if (!iswandering)
        {
            StartCoroutine(Wander());
        }

        if (rotR)
        {
            transform.Rotate(transform.up * Time.deltaTime * rotspeed);
        }
        if (rotL)
        {
            transform.Rotate(transform.up * Time.deltaTime * -rotspeed);
        }

        if (iswalk)
        {
            transform.position += transform.forward * movespeed * Time.deltaTime;
        }
    }

    IEnumerator Wander()
    {
        int rottime = Random.Range(1, 3);
        int rotwait = Random.Range(1, 4);
        int rotLorR = Random.Range(1, 2);
        int walkwait = Random.Range(1, 4);
        int walktime = Random.Range(1, 5);
        iswandering = true;
        yield return new WaitForSeconds(walkwait);
        iswalk = true;
        yield return new WaitForSeconds(walktime);
        iswalk = false;
        yield return new WaitForSeconds(rotwait);
        if (rotLorR == 1)
        {
            rotR = true;
            yield return new WaitForSeconds(rottime);
            rotR = false;
        }
        if (rotLorR == 2)
        {
            rotL = true;
            yield return new WaitForSeconds(rottime);
            rotL = false;
        }
        iswandering = false;
    }
}
```

Here we can see that the bot randomizes the direction and distance of its movement. This way, we now have a simple working bot.

Now, we have to implement its health and a method to remove its health when it gets shot.

Code

```
public class bothealth : MonoBehaviour
{
    public int health = 20;
    void dmginflict(int dmg)
    {
        health -= dmg;
    }
    void Update()
    {
        if(health <= 0)
            Destroy(gameObject);
    }
}
```

Here we can see that the game object is destroyed when the health reaches zero.

Every gun has its limits. In our case, it is a generic one, the ammo. We implemented a universal script in the scene to keep track of the player's global ammo. And in the "gundamage" script we can see that the ammo variable is being decreased with each shot and the player cannot shoot without ammo.

Code

```
public class Ammo : MonoBehaviour
{
    public static int currentammo;

    public int internalammo;

    public GameObject ammodisp;
    void Update()
    {
        internalammo = currentammo;
        ammodisp.GetComponent<Text>().text = "" + internalammo;
    }
}
```

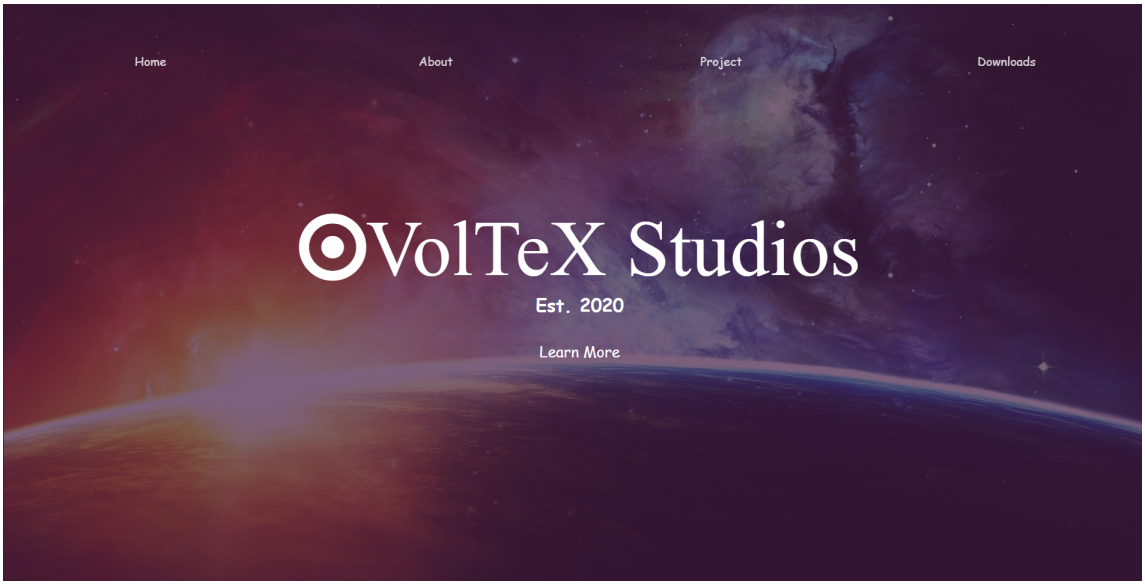
Now that we have our ammo, we need a way increase it once we run out. This is where the pickups come into the game.

Code

```
public class ammodrop : MonoBehaviour
{
    private void OnTriggerEnter(Collider collider)
    {
        Ammo.currentammo += 3;
        this.gameObject.SetActive(false);
    }
}
```

This script can be added a simple cube and whenever the player triggers it, the cube disappears and the global ammo variable is increased by 3.

## 2.4 Website




For the website, we used CSS and HTML. The given graphic is the home page. It has a navigation menu bar. If someone points at it, it will display the options.

```
<html>
<head><meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="stylesheet" href="styles.css" >
<link rel="stylesheet" href= "https://stackpath.bootstrapcdn.com/font-awesome/4.7.0/css/font-awesome.min.

</head>
<body>
<br>
<br>
<ul margin= "35px">
<li><a href="file:///C:/Users/Bose/OneDrive/Desktop/game/index.htm">Home</a></li>
<li><a>About</a>
<ul>
<li><a>Our Team</a></li>
<li><a>News</a></li>
<li><a>Resources</a></li>
</ul>
</li>
<li><a>Project</a>
<ul>
<li><a>Aperture</a></li>
</ul>
</li>
<li><a>Downloads</a>
<ul>
<li><a>Book of Specifications</a></li>
<li><a>1st Report</a></li>
<li><a>2nd Report</a></li>
<li><a>Final Report</a></li>
</ul>
</li>
</ul>
<br>
```

```
<br>
<div class="name">
<i class="fa fa-dot-circle-o" aria-hidden="true">VolTeX Studios</i>
</div>
<h1>Est. 2020</h1>
<div class="Learn">
<a class="btn">Learn More</a>
</div>
</body>
</html>
```

## 2.5 Network

### 2.5.1 Research

We started on this endeavour from scratch, neither knowledge, nor code : nothing. Nevertheless, this was an integral and enormous part to realise. At the point where we started, most of the game had already been well on its way. Re-implementing an the Network after the task proved to be a much greater task than we expected. The engine used an abundance of Callbacks, something we were not familiar with so using the system proved harder than expected.

However, we still tried our best to get the network aspect of the game off the ground. After hearing many suggestions from our peers, we decided to use the Photon Engine PUN.

### 2.5.2 Setting Up The Network

On top minimising the time it takes to access data, treat it and therefore have a high frame rate, now the network adds a new difficulty: we must transmit information in a timely manner and to do this we need to use as less memory as possible. Coming from the fact that this was meant to be a Co-Op multiplayer, this won't way as much on the game as if we were dealing with let's say 10 players.

There are multiple ways to inform a client of another client's movement: 1. Sending directly which keys have been pressed (and therefore simulating the it's movement) 2. Communicating the position of the player constantly 3. Send the Direction and speed of the player.

## 2.6 The Steps of Multiplayer

The connection is almost invisible to the user. First once the instance of the game has connected with the photon servers the Multiplayer button will appear otherwise it shows up as Offline. Then if we click on Multiplayer

a search will ensue as to whether there exists an open room or not for the game. If this is the case then the instance will join the room, otherwise it will create a new room. There is also a waiting button which allows us to abort the search of a room. This may seem obsolete at this point in time but once a delayed search has been implemented it will make sense. For now the user will almost instantly join or create a game.

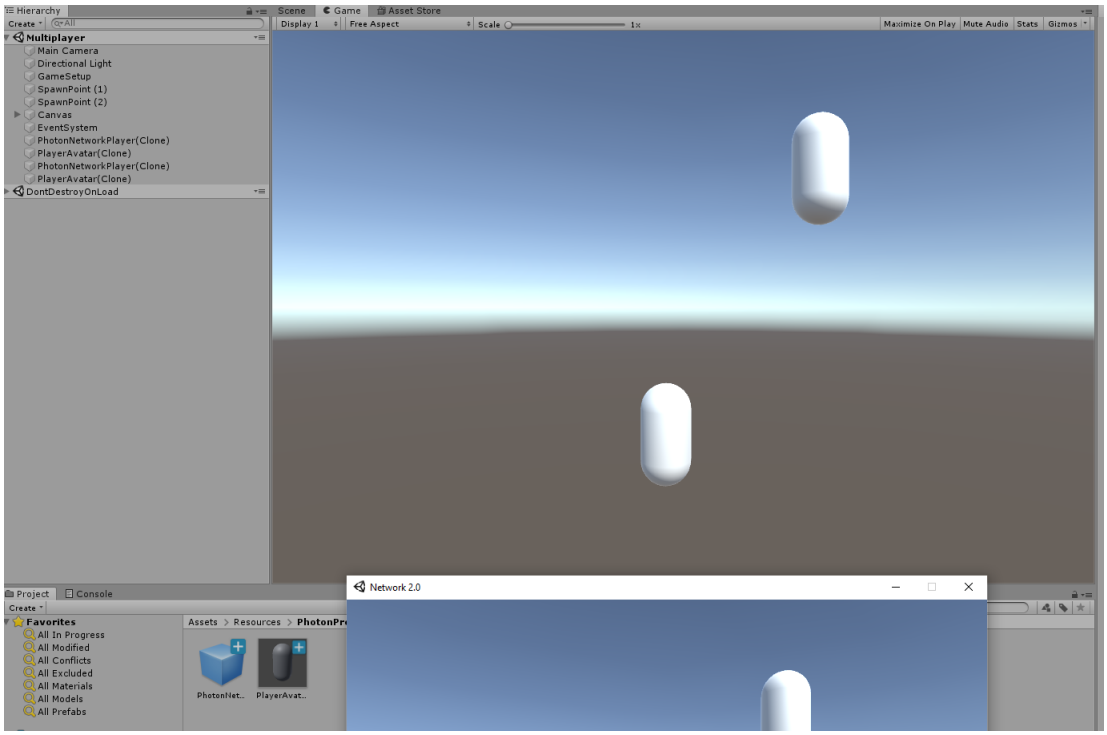
```
void CreateRoom()
{
    int randomRoomName = Random.Range(0 , 10000);
    RoomOptions roomOps = new RoomOptions() {IsVisible = true, IsOpen = true, MaxPlayers = 2};
    PhotonNetwork.CreateRoom("Room" + randomRoomName, roomOps);
    Debug.Log("Created Room");
}

public override void OnJoinRandomFailed(short returnCode, string message)
{
    Debug.Log("Tried to join a random game but failed. There must be no open games available");
    CreateRoom();
}


public override void OnCreateRoomFailed(short returnCode, string message)
{
    Debug.Log("Tried to create a new room but failed, there must already be a room with the same name
    CreateRoom();
}
```

## 2.7 PlayerMovement

In order to make sure that the PlayerMovement is tracked, aka the Transforms we used Photon's Photon View which transmits all the information to the server. In order to extract all the information about movement we used PhotonTransformView which tracks both Position and Rotation (even scale but we won't use that).




Here the two instances of the game, one in the editor the other through a build, if I move one it moves on the other instance which you can see the Hierarchy. Here we added Spawn Points to be able to load the player in a specific position on the scene. The distinction we made between player 1 and player 2 is one is the host while the other is not as shown below.

```
private PhotonView PV;
public GameObject myAvatar;


void Start()
{
    PV = GetComponent<PhotonView>();
    if ((PV.IsMine) && PhotonNetwork.IsMasterClient)
    {
        myAvatar = PhotonNetwork.Instantiate(Path.Combine("PhotonPrefabs", "PlayerAvatar"),
            GameSetup.GS.spawnPoints[0].position, GameSetup.GS.spawnPoints[0].rotation, 0);
    }
    else if (PV.IsMine)
    {
        myAvatar = PhotonNetwork.Instantiate(Path.Combine("PhotonPrefabs", "PlayerAvatar"),
            GameSetup.GS.spawnPoints[1].position, GameSetup.GS.spawnPoints[1].rotation, 0);
    }

}
```

All the Scenes are synchronized using photon so we load a scene that will also match on both screens.

# 3 And After?

We have therefore made good progress with this defense, but we do not stop here! Even if what remains to be done here is attributed to some people in the group, we will develop them, for the most part, together, in order to allow each of us to be able to code different things. Here are the forecasts for the 2nd defense.

## 3.1 Level Design

For the level Design, new levels will be added, with high quality textures, different size and different puzzles to solve.

### 3.1.1 Gameplay

New power ups will be implemented and the AI will be implemented, to make the game more challenging.

### 3.1.2 User Interface

Regarding the main menu, new buttons and animations will be implemented, to make the main menu look cooler , and modern. And for the HUD , we will implement the health bar and a pause menu.

## 3.2 Network

Next step is syncing the network to the gameplay, that means having multiple players each with their camera, also syncing the players health and damage server wide on top every interactable object. Eventually, adding custom matchmaking and a waiting room.