Multi-Player Virtual Reality Game based on Hand Tracking Interaction

Nikolaos Ioannou

Diploma Thesis
School of Electrical and Computer Engineering

Supervisor: Aikaterini Mania
Technical University of Crete
October 2017

Abstract

This thesis implements a multiplayer virtual reality gaming experience via the SteamVR platform utilizing both the Oculus and the Vive virtual reality systems along with their respective controllers. SteamVR allows us to target both VR systems through a single SDK called OpenVR and develop actions for the controllers using VRTK. The players, who can be from anywhere in the world, share a common networked virtual environment designed and developed in Unity3D where their actions and events are synchronized through the Photon Unity Networking solution. Photon connects the players through a low latency dedicated server utilizing a client-server model and the UDP communication protocol. Rendering optimization techniques such as Occlusion Culling and Baked GI were applied in the scene to reduce the GPU load on lower spec computers.

The project features a simple pre-game tutorial for the users to familiarise themselves with the controller setup of their virtual reality system and when ready, transfers them to the networked environment. In this environment, they can walk, swim, grab things and throw them, shoot arrows from a bow with vibrance haptic feedback and more. They are asked to cooperate and solve a series of puzzles that are presented to them in a linear manner. Great care was taken to create a realistic and enjoyable experience by constant modification based on user feedback. The results were very satisfactory, with a very small percentage of the users feeling nausea or other uncomfortable symptoms and the majority of them felt amazed with the immersiveness of the experience.

Acknowledgements

It truly has been a journey, working and studying my way up to this point. All of this would not have been possible without the constant support of my family, friends and colleagues.

First of all, I would like to thank my supervisor, Associate Professor Aikaterini Mania, for introducing me to the amazing world of 3D graphics and giving me the opportunity to write a thesis on such a subject. Her support and guidance have been crucial in its completion.

I would also like to thank the rest of the thesis committee, Associate Professor Michail Lagoudakis and Assistant Professor Vasileios Samoladas for their time in reading and reviewing this thesis.

A big thanks goes to my fellow lab members and students for their suggestions and feedback on improving this project.

Last but not least, I would like to thank my family who patiently supported me all these years.

Table of Contents

1 Introduction

1.1 Concept

Virtual reality is a relatively new, fast-paced technology with applications ranging from simple indie games to top-tier military and business training programs. With the recent advancements in hardware, VR has entered the consumer market in affordable prices and it is expected to reach almost every household in the near future [1]. With this expansion new needs are going to arise, such as connecting all these different virtual reality systems through fun and interactive multiplayer applications for the users. Major social media companies are already developing social VR experiences [2] and the gaming community is not very far behind. Companies like Steam have developed SDKs for communication with various VR systems including Oculus and Vive along with APIs to easily allow programmers to develop applications and games. All that combined with a networking solution such as Photon allows the development of multiplayer environments and applications where users can join and interact regardless of their virtual reality system brand or location.

Oculus Rift and HTC Vive are the two virtual reality systems currently leading the market for desktop VR and they both come with a set of handheld controllers. Both systems behave very similar in manners of tracking and interaction which makes it easier to target them both with a single executable of a game. The Head-Mounted Display(HMD) is used for head tracking, their controllers for hand tracking and the tracking sensors in conjunction with the HMD and the controllers provide positional tracking [3]. Oculus uses the Constellation [6] tracking system while Vive uses Lighthouse [7].

The goal of this project was to create a multiplayer gaming experience in order to connect two users and their virtual reality systems, from anywhere they may be located, in a single VR environment. Photon Unity Networking was used to implement the network capabilities required to connect the users. This environment where the users are placed had to be as realistic as possible given our hardware capabilities and interactive to a certain degree. The users had to be able to interact both with their environment and each other and the ultimate challenge was for them to cooperate and solve certain tasks. Several techniques were used to optimize the game for mid-spec hardware. All of the above will, of course, be explained in greater detail in the following chapters.

1.2 Thesis Outline

This thesis is divided into six main chapters based on their content. The first two chapters include general background information and knowledge in order to better help you understand the subject of this thesis. Detailed development of the project follows along with how the different parts of the game were put together and the final chapter explains how we evaluated the system and the application based on users' feedback. Our conclusions and suggested future work complete this chapter. The structure is explained below in greater detail.

2 - Background: We begin with an introduction to virtual reality and a presentation of the head-mounted displays currently leading the market - Oculus, Vive and PSVR - discussing which ones we are using in our project and why. The technologies behind them like positional, head and hand tracking and how they are implemented in our application is explained. An insight of 3D Modeling and 3D Rendering along with several optimization techniques such as Frustum/Occlusion Culling is presented and then a brief description of popular game engines and why Unity3D is our choice for this project follows. We take a closer look to the network parameters that are going to play a role in our implementation such as latency and bandwidth. We discuss the selection between TCP and UDP for our communication protocol, the various game models for player to player connection and we and finally we present the two networking solutions for the Unity engine, explaining why Photon is the better choice. To conclude, we describe popular genres of multiplayer games.

3 - Utilized Software: In this chapter we take a closer look to all the elements that our application consists of. Main components of Unity, our selected game engine, are thoroughly presented and analysed and then we move on to the tools that help us develop the virtual reality aspect of the game. OpenVR and SteamVR for targeting Oculus and Vive with a single executable and VRTK because it provides us with an abstract interface for developing actions for both systems' controllers. Last but not least, a more detailed view of Photon Unity Networking solution with some of its main features such as Instantiation and Synchronization is presented along with some other general tools that aided in the designing process.

4 - User View: A detailed description on the creation of all the different aspects of the environment is given. From the tutorial scene to the last level, we explain how the terrains, the different models and the hands were created and the interactions with the virtual world. A description of the two systems' controllers - Oculus, Vive - is given and the hands and their features in particular are analyzed.

5 - Implementation: This is the most important chapter of the thesis as it presents in detail all the aspects of the development process of our game. At the beginning, we explain how all the interactions in our game work between the players and the environment along with what steps were required in order to match certain actions to different controller setups. A closer look is taken into the integration of the different GameObjects and their purpose. The whole networking process, from integrating Photon into our game and selecting the correct server setup to implementing features such as object instantiation and game state synchronization is explained step by step with script snapshots of certain actions like real-time updating of game state variables. Problems faced while implementing multiplayer are also described here. Moreover, an explanation on the integration of the virtual reality tools takes place along with the audio implementation and the techniques used for optimizing the game.

6 - Evaluation, Conclusions and Future Work: We present our system evaluation questionnaire which features three distinct sections, General Information, Immersion and Simulator Sickness. The first part was to see if the user had any experience and to what degree with our technologies, the second involved the game and the user's perception on it and the final section was about the user's feelings and well-being after the experiment. The results from our testers are then presented and discussed. Conclusions follow as well as hints about potential future work that could expand the subject of this thesis.

2 Background

2.1 Virtual Reality

The definition of virtual reality comes, naturally, from the definitions for both ‘virtual’ and ‘reality’. The definition of ‘virtual’ is near and reality is what we experience as human beings. So the term ‘virtual reality’ basically means ‘near-reality’. This could, of course, mean anything but it usually refers to a specific type of reality emulation.

We know the world through our senses and perception systems. In school we all learned that we have five senses: taste, touch, smell, sight and hearing. These are however only our most obvious sense organs. The truth is that humans have many more senses than this, such as a sense of balance for example. These other sensory inputs, plus some special processing of sensory information by our brains ensures that we have a rich flow of information from the environment to our minds.

Everything that we know about our reality comes by way of our senses. In other words, our entire experience of reality is simply a combination of sensory information and our brains sense-making mechanisms for that information. It stands to reason then, that if you can present your senses with made-up information, your perception of reality would also change in response to it. You would be presented with a version of reality that isn’t really there, but from your perspective it would be perceived as real. Something we would refer to as a virtual reality. So, in summary, virtual reality entails presenting our senses with a computer generated virtual environment that we can explore in some fashion.

Figure 2.1: Virtual Reality [16]

Answering “what is virtual reality” in technical terms is straightforward. **Virtual reality is the term used to describe a three-dimensional, computer generated environment which can be explored and interacted with by a person** [4]. That person becomes part of this virtual world or is immersed within this environment and whilst there, is able to manipulate objects or perform a series of actions.

This is more difficult than it sounds, since our senses and brains are evolved to provide us with a finely synchronised and mediated experience. If anything is even a little off we can usually tell. This is where you will hear terms such as immersiveness and realism enter the conversation. These issues that divide convincing or enjoyable virtual reality experiences from jarring or unpleasant ones are partly technical and partly conceptual. Virtual reality technology needs to take our physiology into account. For example, the human visual field does not look like a video frame. We have (more or less) 180 degrees of vision and although you are not always consciously aware of your peripheral vision, if it were gone you’d notice. Similarly when what your eyes and the vestibular system in your ears tell you are in conflict it can cause motion sickness. Which is what happens to some people on boats or when they read while in a car. If an implementation of virtual reality manages to get the combination of hardware, software and sensory synchronicity just right it achieves something known as a sense of presence. Where the person really feels like they are present in that environment.

2.2 Head-Mounted Displays (HMDs)

The phrase “Virtual Reality” probably conjures up a very specific image in your mind: those weird yet awesome helmets. They are called head-mounted displays, or HMDs, and they are probably the most instantly recognizable objects associated with virtual reality. As such, they are also referred to sometimes as “Virtual Reality headsets”, or “VR glasses”. HMDs attach straight to your head and present visuals directly to your eyes, and perhaps most excitingly, to your peripheral vision as well. If a device conforms to these criteria you may consider it an HMD in the broadest sense.

HMDs are not only used in virtual reality gaming, they’ve also been utilized in military, medical and engineering contexts to name a few. Moreover, these devices can be used to create something called augmented reality, which overlays digital information through an HMD filter onto the real world.

These devices have an incredible range of use. In order to allow for the best possible experience with an HMD, a number of technologies need to be incorporated. Let’s elaborate on a few.

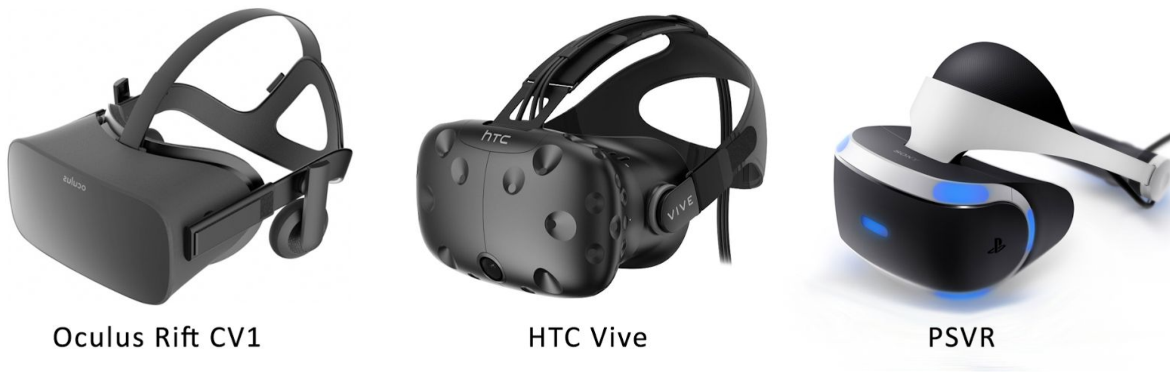


Figure 2.2: Current-gen HMDs

In this thesis, we developed our application based on the specifications of the Oculus Rift CV1 and the HTC Vive. Both systems run on around the same hardware specs on the PC, while the PSVR requires a Playstation to run which is why it was not even considered for implementation.

2.2.1 Positional Tracking

Positional tracking is a technology that allows a device to estimate its position relative to the environment around it. It uses a combination of hardware and software to achieve the detection of its absolute position. Positional tracking is an essential technology for virtual reality (VR), making it possible to track movement with six degrees of freedom (6DOF).

It has to be noted that head tracking is not the same as positional tracking. While head tracking only registers the rotation of the head (Rotational tracking), with movements such as pitch, yaw, and roll, positional tracking registers the exact position of the headset in space, recognizing forward/backward, up/down and left/right movement.

Positional tracking VR technology brings various benefits to the VR experience. It can change the viewpoint of the user to reflect different actions like jumping, ducking, or leaning forward; allow for an exact representation of the user's hands and other objects in the virtual environment; increase the connection between the physical and virtual world by, for example, using hand position to move virtual objects by touch and detect gestures by analyzing position over time.

It is also known that positional tracking improves the 3D perception of the virtual environment because of parallax (the way objects closer to the eyes move faster than objects farther away). Parallax helps inform the brain about the perception of distance along with stereoscopy. Also, the 6DOF tracking helps reduce drastically motion sickness during the VR experience that is caused due the disconnect between the inputs of what is being seen with the eyes and what is being felt by the ear vestibular system.

There are different methods of positional tracking. Choosing which one to apply is dependent on various factors such as the tracking accuracy and the refresh rate required, the tracking area, if the tracking is indoor or outdoor, cost, power consumption, computational power available, whether the tracked object is rigid or flexible, and whether the objects are well known of can change.

Positional tracking VR technology is a necessity for VR to work properly since an accurate representation of objects like the head or the hands in the virtual world contribute towards achieving immersion and a greater sense of presence.

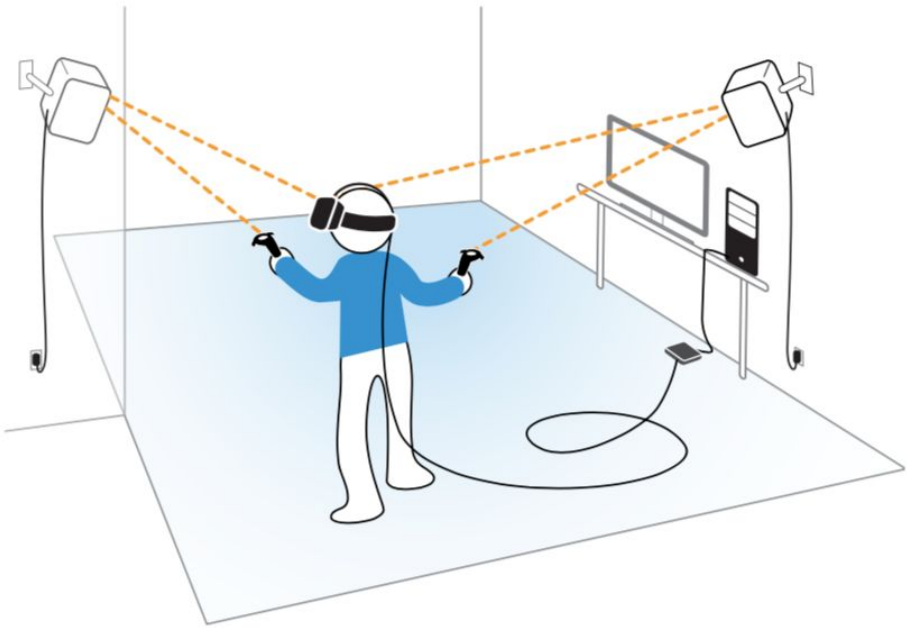


Figure 2.3: Positional Tracking

Our game makes use of the positional tracking in setting the transform of the user in the virtual environment through his real-world movements.

2.2.2 Head Tracking

To acknowledge your head's position and transform that into other data, your HMD needs some accurate head tracking technology. Thanks to advances in smartphone technology, we can now put a multi-axis accelerometer on a chip, allowing infrared tracking cameras to accurately watch markers on the HMD and relay positional data to the computer. Mobile HMDs cannot make use of external camera tracking because they are not for use in a fixed location – but some new technologies, such as the Microsoft Hololens, and Google Project Tango, can use multiple sensors in addition to accelerometers for positional calculation.

It is wise to note that some HMDs, especially those that use your smartphone, can only track the direction in which you are looking. Dedicated HMDs with the astonishing capability to track another axis do exist, and they even allow the user to lean in to look more closely at their virtual surroundings. This is a vital extra if you want the full immersion experience.

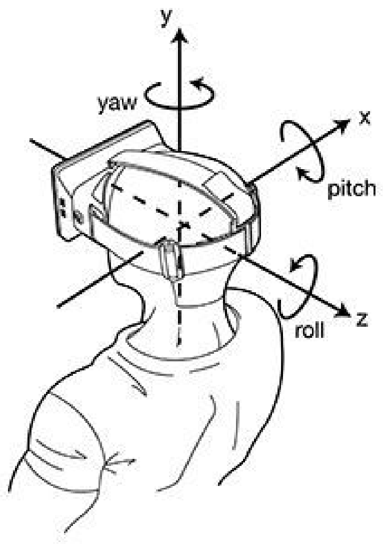


Figure 2.4: Head Tracking

Head tracking was also implemented in the application the same way positional tracking was. The movement and rotation of the player's head is used to move and rotate his in-game virtual head.

2.2.3 Hand Tracking

Much like head and positional tracking, hand-tracking describes the process of constantly capturing a user's hands position and movements in real 3D space. It is very obvious that being able to interact with a virtual environment by simply using your hands (as in real life), will bring the realism and interactivity of virtual reality to a whole new level. It usually works by allowing the user to see a 3D computer generated replica of his hands in the virtual world, according to their current position and rotation in real 3D space. For example, a user could point his (real) finger to touch a virtual button, thrust his hand forward to punch a virtual enemy or make a gesture to trigger an event.

There are two main techniques that implement hand-tracking:

The first technique, requires a user to constantly hold special controllers with his hands, which act as positional tracking devices for each individual hand. A similar process to positional tracking is followed, measurements are taken between the controllers and base stations in a room. Then, the position of each hand can be calculated in relativity to the head and room and be projected in the virtual world.

The second technique involves a design similar to eye-tracking which includes image capturing and image processing. One or more camera sensors located on the HMD point towards the user's hands, capturing their movement in several frames per second. Then advanced mathematical and image processing algorithms analyze each image/frame, producing a 3D representation of how the device sees the hands.

This thesis makes use of the first technique by using the controllers of each HMD.

2.2.4 Specifications

There are various quality factors that distinct HMDs from one another. Every current or older generation HMD has its advantages and disadvantages, depending on the application it is intended for. The most common HMD specifications are the following.

Display Technology

Unlike the AMOLED displays of devices like smartphones, which you view from a distance, a VR headset's screen is magnified and only inches from your face, so you can easily make out the subpixels. This is where Sony's PSVR has an advantage. Its 1080p screen's RGB stripe matrix is superior to the Samsung PenTile matrix of the other HMD displays. In a square of four pixels there are more subpixels and also, importantly, an even number of each color.

On a PenTile matrix screen, there are fewer subpixels and most of them are green. That means the perceived resolution (especially when looking at it closely) of the PSVR screen is close to that of the Vive and Rift. When looking at a PenTile display closely the display looks grainier, the screen-door effect (SDE, the grid of the gap between pixels) is more pronounced, and gradients look worse. Samsung uses a PenTile matrix in its AMOLED panels because it costs less to achieve the desired resolution, and the panels have a longer life. The HTC Vive and Oculus Rift use the Diamond PenTile subpixel matrix found in

Samsung's new AMOLED screens, with less perceived SDE because of the higher pixel density [5].

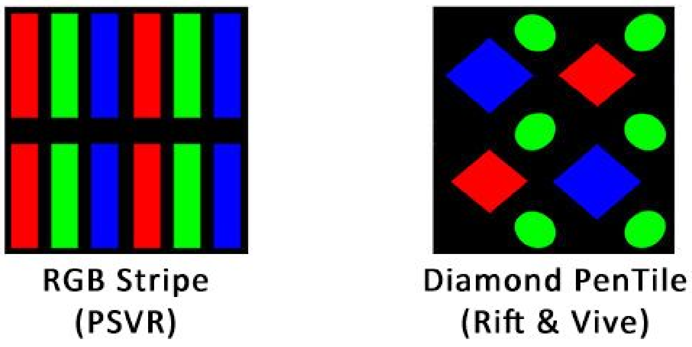


Figure 2.5: Display Technologies used in various HMDs

Refresh rate

The refresh rate refers to how quickly a display can change its content over a particular length of time. Typically, modern LCD computer monitors can do this 60 times per second, or at 60Hz. This also corresponds to a maximum *frame rate* of 60 frames per second – or 60fps – where one frame equals a complete image on your screen.

As a comparison: cinematic films generally run at a uniform rate of 24fps. 60fps is gaining traction, especially in 'action' footage where motion blur is less than ideal. To put this all simply; the more frames used in a given second, the smoother and crisper motion appears. Since virtual reality is meant to impose a certain sense of immersion and realism, crisp 'lifelike' motion and a lack of motion blur is crucial to the overall experience. It seems so far that 60fps is the minimum needed to achieve this, but HMDs as high as 120Hz are already in existence. Both Oculus Rift and HTC Vive currently run at 90 Hz refresh rate.

Optics

In order to create that immersive feeling of inhabiting a virtual world, you have to stretch that flat image to fill your visual field entirely. This is commonly referred to as *optics*. Experimentation by the University of South California indicated that a HMD would achieve the visuals needed for a convincing virtual reality if it had a field of view (FOV) of between 90 and 100 degrees. This effect is achieved by using the lenses in a HMD to take a flat image and change it into something that fills every nook and cranny of our eye line. The quality of the lenses used in your headset is vitally important; an HMD that uses cheap lenses may have a poor picture quality, low clarity, and some unwanted distortion. HTC Vive uses Fresnel lenses and the Oculus Rift has hybrid Fresnel lenses to keep the lenses thin and bend the light in a way that helps us to see clearly.

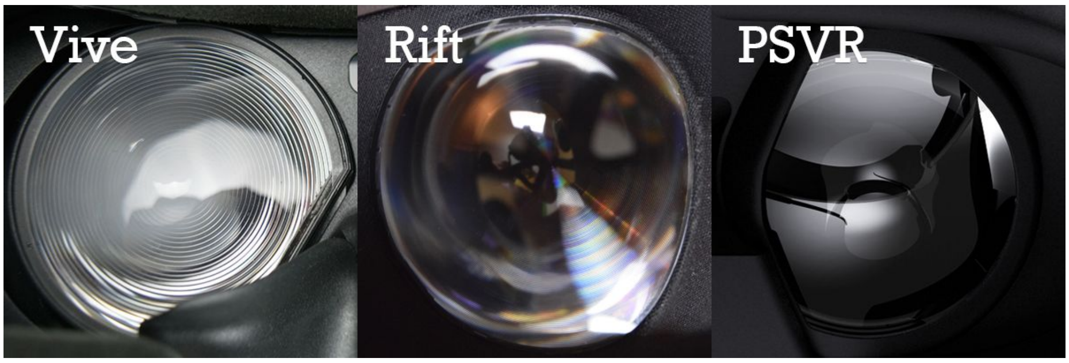


Figure 2.6: Lenses in various HMDs

Other quality factors

There are of course several other factors that contribute to the quality of the HMD and to the user's experience. Some of them include adjustable IPD(InterPupilary Distance), latency, weight and more.

2.3 3D Modeling

3D modeling is the use of software to create a virtual three-dimensional model of some physical object. It is used in many different industries, including virtual reality, video games, 3D printing, marketing, TV and motion pictures, scientific and medical imaging and computer-aided design and manufacturing CAD/CAM.

3D modeling software generates a model through a variety of tools and approaches including:

- simple polygons.
- 3D primitives - simple polygon-based shapes, such as pyramids, cubes, spheres, cylinders and cones.
- spline curves.
- NURBS (non-uniform rational b-spline) - smooth shapes defined by bezel curves, which are relatively computationally complex.

2D geometric polygon shapes are used extensively in motion picture effects and 3D video game art. Creating approximations of shapes made with polygons is much more efficient in raster graphics, which are required for real time 3D gaming.

In art for video games and motion picture effects, a model might start as a rough-out using polygon primitives or NURBS, or as a design made by following contours on multiple 2D isometric views. If the model is to be animated, careful consideration of the arrangement of continuous edge loops must be maintained in the model's polygons around areas of deformation such as joints. Once a model is adequately built, an artist might arrange the coordinates of the model to match its 2D textures in a process called UV mapping, a process that is kind of like trying to design and tailor with a computer mouse. Areas that require more detail are given more space in the UV map. This can be done either using a repeating texture such as a checker board as a placeholder or by using an existing texture.

Generally the next step might be to texture the model, which is to apply either hand-painted or photograph-based 2D images, usually TGA (Targa bitmap), to the model that will define:

- Its color with a color map.
- Its reflectivity and reflected color with a specular map.
- Its surface texture, defined through light-play with a bump or normal map, or a deformation map for actual added geometric detail.

Animated models require an extra step of rigging, which is like giving them a virtual skeleton with bones and joints along with the controllers to manage it. The way the texture of these joints influences the surface texture under deformation must be defined in skinning, where one paints the weight of joint influence on the textures directly on the model's polygons; a polygon painted more heavily is more heavily influenced by a selected joint’s movement. The model is then ready for the animator.

More computational and expensive methods of making models such as NURBS may be used, along with complex shaders that interact with particle-based light, in rendered graphics when real time is not a necessity.

3D Modeling was used quite heavily in this game, either by already made assets from the Unity Asset Store or by models customly created in 3ds Max.

2.4 3D Rendering

3D rendering is the 3D computer graphics process of automatically converting 3D wire frame models into 2D images with 3D photorealistic effects or non-photorealistic rendering on a computer.

Rendering is the final process of creating the actual 2D image or animation from the prepared scene. This can be compared to taking a photo or filming the scene after the setup is finished in real life. Several different, and often specialized, rendering methods have been developed. These range from the distinctly non-realistic wireframe rendering through polygon-based rendering, to more advanced techniques such as: scanline rendering, ray tracing, or radiosity. Rendering may take from fractions of a second to days for a single image/frame. In general, different methods are better suited for either photorealistic rendering, or real-time rendering.

Rendering for interactive media, such as games and simulations, is calculated and displayed in real time, at rates of approximately 20 to 120 frames per second. In real-time rendering, the goal is to show as much information as possible as the eye can process in a fraction of a second (a.k.a. in one frame. In the case of 30 frame-per-second animation a frame encompasses one 30th of a second). The primary goal is to achieve an as high as possible degree of photorealism at an acceptable minimum rendering speed (usually 24 frames per second, as that is the minimum the human eye needs to see to successfully create the illusion of movement). In fact, exploitations can be applied in the way the eye 'perceives' the world, and as a result the final image presented is not necessarily that of the real-world, but one close enough for the human eye to tolerate. Rendering software may simulate such visual effects as lens flares, depth of field or motion blur. These are attempts to simulate visual phenomena resulting from the optical characteristics of cameras and of the human eye. These effects can lend an element of realism to a scene, even if the effect is merely a simulated artifact of a camera. This is the basic method employed in games and interactive worlds. The rapid increase in computer processing power has allowed a progressively higher degree of realism even for real-time rendering, including techniques such as HDR rendering. Real-time rendering is often polygonal and aided by the computer's GPU.

2.4.1 Global Illumination (GI)

Global Illumination (GI) is a system that models how light is bounced off of surfaces onto other surfaces (indirect light) rather than being limited to just the light that hits a surface directly from a light source (direct light). Modelling indirect lighting allows for effects that make the virtual world seem more realistic and connected, since objects affect each other’s appearance. One classic example is “color bleeding” where, for example, sunlight hitting a red sofa will cause red light to be bounced onto the wall behind it. Another is when sunlight hits the floor at the opening of a cave and bounces around inside so the inner parts of the cave are illuminated too.

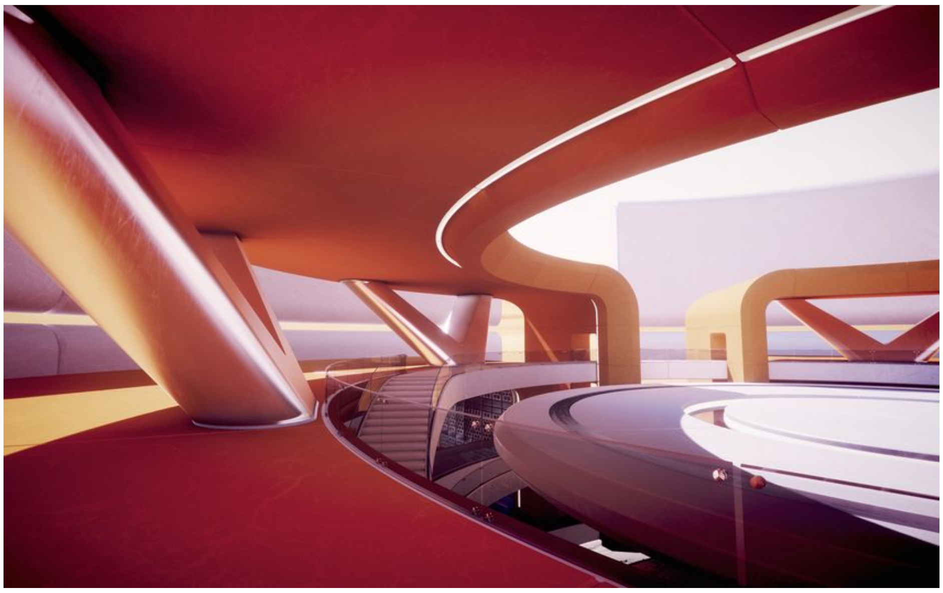


Figure 2.7: GI in a rendered scene, note the subtle effect of indirect lighting

Baked GI and Precomputed Realtime GI

Traditionally, video games and other realtime graphics applications have been limited to direct lighting, while the calculations required for indirect lighting were too slow so they could only be used in non-realtime situations such as CG animated films. A way for games to work around this limitation is to calculate indirect light only for objects and surfaces that are known ahead of time to not move around (that are static). That way the slow computation can be done ahead of time, but since the objects don't move, the indirect light that is pre-calculated this way will still be correct at runtime. Many game engines support this technique, called Baked GI (also known as Baked Lightmaps), which is named after "the bake" - the process in which the indirect light is precalculated and stored (baked). In addition to indirect light, Baked GI also takes advantage of the greater computation time available to generate more realistic soft shadows from area lights and indirect light than what can normally be achieved with realtime techniques.

Additionally, a new technique exists called Precomputed Realtime GI [8]. It still requires a precomputation phase similar to the bake mentioned above, and it is still limited to static objects. However it doesn't just precompute how light bounces in the scene at the time it is built, but rather it precomputes all possible light bounces and encodes this information for use at runtime. So essentially for all static objects it answers the question "if any light hits this surface, where does it bounce to?" This information about which paths light can propagate by for later use is then saved. The final lighting is done at runtime by feeding the actual lights present into these previously computed light propagation paths.

This means that the number and type of lights, their position, direction and other properties can all be changed and the indirect lighting will update accordingly. Similarly it's also possible to change material properties of objects, such as their color, how much light they absorb or how much light they emit themselves.

While Precomputed Realtime GI also results in soft shadows, they will typically have to be more coarse-grained than what can be achieved with Baked GI unless the scene is very small. Also note that while Precomputed Realtime GI does the final lighting at runtime, it does so iteratively over several frames, so if a big a change is done in the lighting, it will take more frames for it to fully take effect. And while this is fast enough for realtime applications, if the target platform has very constrained resources it may be better to to use Baked GI for better runtime performance.

In our game, Baked Illumination was used for a performance boost since we do not need to calculate the light for every static triangle and Realtime Illumination was used for moving objects.

2.4.2 Hidden Surface Determination

Hidden surface determination is a process by which surfaces which should not be visible to the user (for example, because they lie behind opaque objects such as walls) are prevented from being rendered. Despite advances in hardware capability there is still a need for advanced rendering algorithms. The responsibility of a rendering engine is to allow for large world spaces and as the world's size approaches infinity the engine should not slow down but remain at constant speed. Optimising this process relies on being able to ensure the deployment of as few resources as possible towards the rendering of surfaces that will not end up being rendered to the user.

There are many techniques for hidden surface determination. They are fundamentally an exercise in sorting, and usually vary in the order in which the sort is performed and how the problem is subdivided. Sorting large quantities of graphics primitives is usually done by divide and conquer [9].

Frustum Culling

Frustum Culling is a rendering technique where the CPU skips telling the GPU to draw anything that lies outside of the "view cone" of the camera. (Because the display is a rectangle, that view cone is a four-sided pyramid; a pyramid with its top cut off is called a "frustum" in geometry, and for various arcane technical reasons the view cone used in 3D graphics does have its point cut off, hence the name. And we are skipping drawing, i.e. discarding from consideration, i.e. "culling".) This may also save the CPU some work (or cost it some!), but let's stick to the GPU.

Because the only things that are skipped are things outside the view cone, they were never visible anyway. If we asked the GPU to draw them, it would do some (different, more expensive) work determining they weren't visible, so skipping them in the CPU doesn't change the final image at all. Because no pixels would have been affected, and because of the way the system works, no pixel shader work would ever have been done for those objects at all, only vertex shader. So, frustum culling saves vertex shader work but not pixel shader work (because no pixel shader work would have been done).

To a very rough approximation, an outdoors scene with a 90-degree view cone sees 1/4 of the full 360 degrees, so if we compare what would happen if we draw the full scene to drawing only within the view cone, we skip approximately 3/4 of the vertex shading work.

GPUs use the same hardware to compute vertex shaders as to compute pixel shaders, so, for a fixed frame rate, the time saved on vertex shading can now be spent on more pixel shading, either by increasing the complexity of the pixel shading (e.g. image quality), or by increasing the number or geometric complexity of objects in the scene (spending that savings on both vertex and pixel shading).

Occlusion Culling

Occlusion Culling is a feature that disables rendering of objects when they are not currently seen by the camera because they are obscured (occluded) by other objects. This does not happen automatically in 3D computer graphics since most of the time objects farthest away from the camera are drawn first and closer objects are drawn over the top of them (this is called “overdraw”). Occlusion Culling is different from Frustum Culling. Frustum Culling only disables the renderers for objects that are outside the camera’s viewing area but does not disable anything hidden from view by overdraw. Note that when you use Occlusion Culling you will still benefit from Frustum Culling.

The occlusion culling process will go through the scene using a virtual camera to build a hierarchy of potentially visible sets of objects. This data is used at runtime by each camera to identify what is visible and what is not. Equipped with this information, the game engine will ensure only visible objects get sent to be rendered. This reduces the number of draw calls and increases the performance of the game.

The data for occlusion culling is composed of cells. Each cell is a subdivision of the entire bounding volume of the scene. More specifically the cells form a binary tree. Occlusion Culling uses two trees, one for View Cells (Static Objects) and the other for Target Cells (Moving Objects). View Cells map to a list of indices that define the visible static objects which gives more accurate culling results for static objects.

It is important to keep this in mind when creating your objects because you need a good balance between the size of your objects and the size of the cells. Ideally, you shouldn't have cells that are too small in comparison with your objects but equally you shouldn't have objects that cover many cells. You can sometimes improve the culling by breaking large objects into smaller pieces. However, you can still merge small objects together to reduce draw calls and, as long as they all belong to the same cell, occlusion culling will not be affected.

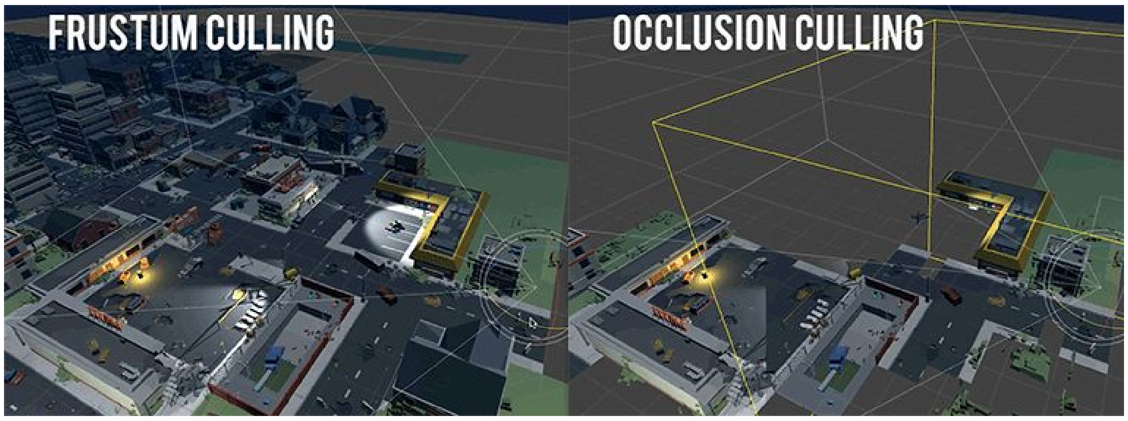


Figure 2.8: Differences in Frustum and Occlusion Culling [17]

Unity by default applies Frustum Culling in its scenes. However, with some custom settings in our game we enabled Occlusion Culling because it gives an enormous performance boost. As it is easily extracted from the figure above, rendering takes way less time to complete for a single frame due to Occlusion Culling.

2.4.3 Level of Detail (LOD)

When a GameObject in the scene is a long way from the camera, the amount of detail that can be seen on it is greatly reduced. However, the same number of triangles will be used to render the object, even though the detail will not be noticed. An optimisation technique called Level Of Detail (LOD) rendering allows you to reduce the number of triangles rendered for an object as its distance from camera increases. As long as your objects aren't all close to the camera at the same time, LOD will reduce the load on the hardware and improve rendering performance.

The images below show how the LOD level used to render an object changes with its distance from camera. The first shows LOD level 0 (the most detailed). Note the large number of small triangles in the mesh. The second shows a lower level being used when the object is farther away. Note that the mesh has been reduced in detail (smaller number of larger triangles).

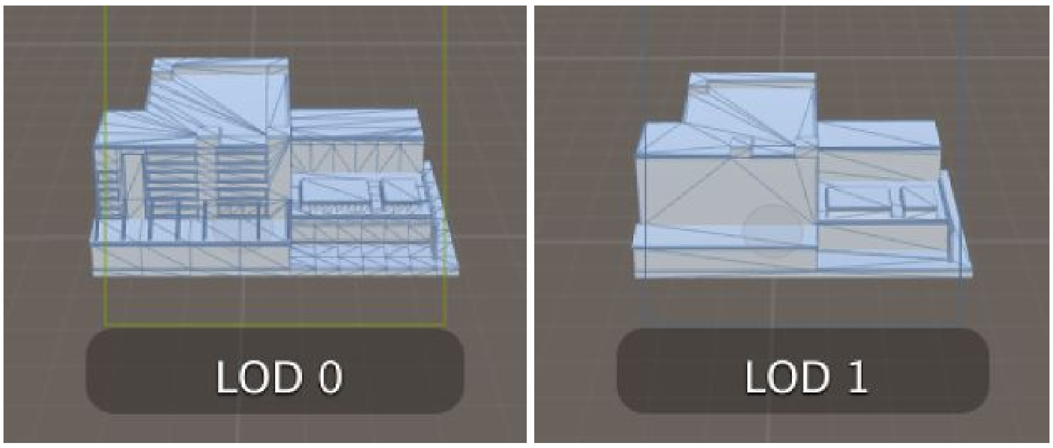


Figure 2.9: Differences in LOD models

Specific models like the rocks in our game had different LOD levels, so Unity loads a lighter model when we are far from them and cannot see the details.

2.4.4 Frame rate

Frame rate (also known as frame frequency) is the frequency (rate) at which an imaging device produces unique consecutive images called frames. The term applies equally well to computer graphics, video cameras, film cameras, and motion capture systems. Frame rate is most often expressed in frames per second (FPS), and is also expressed in progressive scan monitors as hertz (Hz).

The human eye and its brain interface, the human visual system, can process 10 to 12 separate images per second, perceiving them individually [10]. The visual cortex holds onto one image for about one-fifteenth of a second, so if another image is received during that period an illusion of continuity is created, allowing a sequence of still images to give the impression of motion.

In video, film, computer graphics and even more in Virtual Reality, frame rate output is critical, as it decides whether consecutive frames will be perceived by the brain as separate images or not, thus giving the desired illusion of motion.

Frame rate plays a crucial role in the immersiveness of our gaming experience, so great care was taken in optimization ensure as higher frame rate as possible.

2.5 Game Engines

A game engine is a software framework designed for the creation and development of video games. Developers use them to create games for consoles, mobile devices and personal computers. The core functionality typically provided by a game engine includes a rendering engine ("renderer") for 2D or 3D graphics, a physics engine or collision detection (and collision response), sound, scripting, animation, artificial intelligence, networking, streaming, memory management, threading, localization support, scene graph, and may include video support for cinematics. The process of game development is often economized, in large part, by reusing/adapting the same game engine to create different games, or to make it easier to port games to multiple platforms.

Modern game engines are some of the most complex applications written, often featuring dozens of finely tuned systems interacting to ensure a precisely controlled user experience. The continued evolution of game engines has created a strong separation between rendering, scripting, artwork, and level design. It is now common, for example, for a typical game development team to have several times as many artists as actual programmers.

Several game engines exist in the market, some are free some are paid. We are going to take a brief look in 3 of the most popular free game engines as of this moment and explain which one is more suitable for this project.

2.5.1 Unity3D

Unity is a cross-platform game engine developed by Unity Technologies, which is primarily used to develop video games and simulations for computers, consoles and mobile devices. First announced only for OS X, at Apple's Worldwide Developers Conference in 2005, it has since been extended to target 27 platforms.

Unity is marketed to be an all purpose engine, and as a result supports both 2D and 3D graphics, drag and drop functionality and scripting through its 3 custom languages(C#, JavaScript, Boo). The engine targets the following APIs: Direct3D and Vulkan on Windows and Xbox 360; OpenGL on Mac, Linux, and Windows; OpenGL ES on Android and iOS; and proprietary APIs on video game consoles. Within 2D games, Unity allows importation of sprites and an advanced 2D world renderer. For 3D games, Unity allows specification of texture compression and resolution settings for each platform that the game engine supports, and provides support for bump mapping, reflection mapping, parallax mapping, screen space ambient occlusion (SSAO), dynamic shadows using shadow maps, render-to-texture and full-screen post-processing effects. Unity also offers services to developers, these are: Unity Ads, Unity Analytics, Unity Certification, Unity Cloud Build, Unity Everyplay, Unity IAP, Unity Multiplayer, Unity Performance Reporting and Unity Collaborate which is currently still in beta.

2.5.2 Unreal Engine

The Unreal Engine is a game engine developed by Epic Games, first showcased in the 1998 first-person shooter game Unreal. Although primarily developed for first-person shooters, it has been successfully used in a variety of other genres, including stealth, MMORPGs, and other RPGs. With its code written in C++, the Unreal Engine features a high degree of portability and is a tool used by many game developers today.

The current release is Unreal Engine 4, designed for Microsoft's DirectX 11 and 12 (for Microsoft Windows, Xbox One, Windows RT); GNM (for PlayStation 4); OpenGL (for macOS, Linux, iOS, Android, Ouya and Windows XP[l]); Vulkan (for Android); Metal (for iOS[l]); and JavaScript/WebGL (for HTML5 web browsers).

2.5.3 CryEngine

CryEngine is a game engine developed by game developer Crytek , which has been used in all of their titles. It is known for its ability to produce stunning, eye-catching graphics and visuals, featuring advanced shader and lighting systems. Because of this, CryEngine clearly targets only powerful PCs and high-end consoles. It comes with VR support and a large amount of advanced visual features, tools, audio/physics systems and character and animation systems. Its main programming language is C++.

2.5.4 Selecting the Engine

Unreal Engine and Unity3D are currently ahead of the competition as the two most popular game engines available to the public. This is due to the fact that they both succeed in providing high-end graphics, a large variety of usable tools, great support for platforms and devices, without compromising usability and efficiency. It is important to note that these 2 engines offer a large community support, which is also something that has to be considered when choosing an engine. CryEngine is also a great and powerful engine with remarkable capabilities, however its complicated structure and smaller community excluded it from our consideration.

In conclusion, taking into account the advantages and disadvantages of each engine, Unity proved to be the ideal choice for this project, mainly due to its efficiency, large community support and ease of use.

2.6 Multiplayer and Networking

2.6.1 Network Parameters

There are three main network parameters that affect a user’s perception of a multiplayer online game: latency, bandwidth and packet loss. Additionally, what kind of network you choose to set up, peer-to-peer or client-server for example, has an impact on the user’s gaming experience (see section 2.6.3).

Latency

Latency is the time it takes for the terminal to send information to a server or to receive the reply. The round trip latency from the terminal to the server and server to terminal, is usually called the round trip time (RTT). This round trip latency varies greatly from network to network, wired networks to wireless mobile networks, and even as a function of congestion on a given network. Of course, it chiefly depends on where the server is situated with respect to the user’s terminal. If you use the server in the same local network, you will of course have lower latency than if the server is on the other side of the world. This is why you usually play online games with others from the same continent.

Of course latency not only depends on the client/server geographic positions. Latency is also caused by the following other three factors:

- Inadequate network performance: Packets either gets dropped along the way, as the network can not handle the offered load of packets, or the packets due to congestion takes a longer path from source to destination.
- Inadequate server processing power: If the terminal wants information from the server that is not cached, or that the server needs to use a lot of processing power to handle the request, there will be processing delay. This affects many multiplayer games during the busiest hours when the server has to deal many different client requests.
- Inadequate terminal processing power: A slow terminal may have difficulties processing all the received data and thereby create a delay. This can sometimes easily be solved by shutting down other programs running on the terminal or by upgrading to a more powerful terminal or if it is possible upgrading the hardware of the terminal.

Due to different loads on the network, it gets congested at different times during the day. Therefore you can not expect that the round trip time is time independent. This time dependent latency is something a game developer should be aware of, because gamers will always play, regardless of the time, but more will be playing at the peak playing times. An assumption that peak playing times are in the evening and during weekends can easily be tested, by pinging a gaming server during different hours.

Bandwidth

Generally when we speak of bandwidth we talk about the useful frequency band which a channel can transmit. A high bandwidth leads to a high throughput. Usually the throughput is measured bytes (or bits) per second. However, we often talk about the bandwidth of a particular channel in terms of the maximum potential throughput. In multiplayer gaming this is translated in how much information a client can send or receive at a given moment, along with everything else happening in the network. For this reason we want to keep that to a minimum by optimizing our code to send and as a result receive from the server as little information as possible.

Packet loss

If packets never arrive at their destination they are considered lost. This loss of packets can be caused by many factors, including network saturation, degradation of the signal through the network medium, faulty network hardware, or faulty packets being rejected by nodes. Packet loss can significantly degrade the quality of a gaming experience [11][12]. Packet drops due to network saturation can be caused by different factors, such as a bad route or lack of sufficient throughput on a bottleneck link or router.

Jitter

Jitter, the delay variance, is almost as important for the gaming quality as latency. It can be caused by routers queuing packets, due to congestion or prioritizing traffic. At high loads packets may take an alternative route. This causes packets to arrive at the destination with differing delays. The source can't avoid jitter - it is going to happen or not, but the source can enable the receiver to deal with it by timestamping the packets it sends, thus allowing the receiver to deal with them in a suitable fashion. Then when they arrive to the end node they can be stored in a “de-jitter” buffer until they are delivered in the right sequence and with appropriate interarrival spacing based on the timestamps. This will increase the end-to-end delay for some packets, and must be seen as a trade-off between delay and increased packet loss. The later packet loss occurs in the receiving terminal, when the packet arrives too late to be useful – it will be discarded. This is the worst form of packet loss, since it has actually been delivered and used resources all along the path.

All these network parameters were considered in the development of the game. However, since our game only featured two players and a limited amount of networked resources at any given time, it hardly put any load on the network. Furthermore, the evaluation was performed inside the Technical University of Crete's network where the internet speed and bandwidth are very high.

2.6.2 Protocols

The internet is mainly based on two protocols TCP and UDP. These two protocols have both advantages and disadvantages for multiplayer game development purposes.

TCP

TCP is a reliable byte stream protocol. When a packet is lost, it will be retransmitted. To achieve this every packet has to be acknowledged, and this of course consumes bandwidth. If there is traffic flowing in the other direction, then this acknowledgement can be piggybacked on the outgoing traffic so the overhead of acknowledging a sequential stream of bytes is only 4 bytes. Due to the fact that TCP is connection orientated it has more overhead. The header size of TCP is at least 40 bytes while the header size of UDP 20 bytes. However, header compression is generally very effective and can reduce this down to one or two bytes in most cases.

The game is dependent on the performance of higher-layer transport protocols such as TCP, which affects the perception of the technology by the end user. This is due to TCP's limitations in a wireless network, where the protocol influences the traffic strongly due to its slow-start and congestion-avoidance mechanism. TCP assumes that a packet loss occurs due to congestion, which usually is the cause in a wired network; where congestion is avoided by lowering the sending rate. In a wireless environment, however, non-congestion packet loss occurs at a much greater rate than in a wired network. Therefore TCP unnecessary decreases its throughput when implemented in a mobile network if packet losses occur. The slow-start mechanism of TCP influences the end user experience of web browsing to a much greater extent than the radio link. This also affects gaming which produces similar traffic as web browsing.

UDP

UDP is a datagram protocol, and if packets are lost they are simply lost. This, however, consumes less bandwidth for the same amount of data compared with TCP. Of course this also reduces latency due to the fact that there is no waiting for retransmission. Due to the fact that the game state traffic is more time sensitive, but relative tolerant of losses (in this it is similar to voice over IP traffic), UDP is preferred by game developers. For example, if one update is missed due to packet loss there will be a new update slightly later that the terminal will be able to use. However, TCP can be used in parallel with UDP. For example, in a game application where packet delivery is important such as player-to-player chat; hence TCP may be used for the chat, while the game state is transmitted with UDP.

The selected networking solution we will examine in chapter 2.7.5 uses UDP protocol in its communication since transmission reliability is not that essential and packets with new information will always be sent with the next update.

2.6.3 Game Models

Peer-to-peer

A peer-to-peer connection is only based on clients, who all are connected to each other. The game status is updated individually at every client. This solution is perfect for the game provider because there is no need to invest in a server. Another advantage is that the network is very stable. If one client goes down, the remaining client still make up a network and continue to send information to each other.

The drawbacks are plenty. All the clients have all the game information, so it will be much easier for one gamer to cheat by hacking the game information in his terminal. Also the implementation is rather complicated in comparison to the other models. To avoid divergence in game state due to delays and other factors, synchronization has to occur between clients to avoid divergence. Another disadvantage is that the network traffic generated increases exponentially and a player can easily run out of bandwidth. To understand why the network traffic rises exponentially the reader should think of the numbers of open connection. With three clients, each connected to the other two and sharing their game states, then the network needs 3*2=6 connections. If we now instead have four clients, they all each connect with three others giving us 4*3=12 connections. With five clients 5*4=20 connections are needed, thus we have an exponential increase. Note however, the numbers of connections to and from each client only increase linearly.

Other disadvantages include the difficulty of keeping all players synchronized (minute differences between peers can escalate over time to game-breaking paradoxes) and the difficulty of supporting new peers joining part-way through a game. Moreover, each peer must wait for every other peer's message before simulating the next "network frame", resulting in all players experiencing the same latency as the player with the worst connection.

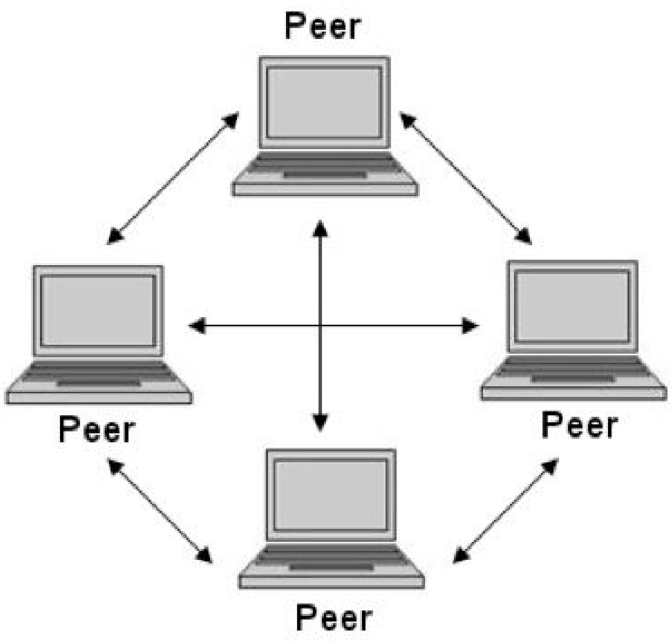


Figure 2.10: Peer-to-peer architecture

Client-server

A client-server connection is based on a server. This server stores and processes all the game data it receives from all the connected clients. Then it only updates those clients with the data they need, thus every client receives a unique update. The limited information that the server sends to its clients is good from a traffic point of view and leads to lower latency. Also from a coding point of view this model is preferable, because little code needs to be added to support this sharing of state information, and it can easily be separated from the game code. These are the reasons why most modern multiplayer game implement this solution.

One drawback however, is that bottlenecks at the server still can occur, due to high load from many clients (who are sharing this machine because of the allocations of resources to each of the instances of the game). This architecture has another big disadvantage; someone has to pay for the server and this cost can be substantial if the game is to support many clients and to maintain the latency below some threshold.

The client server model is preferable from a cheating point of view. Every client has a scope in the game world, stored in the server. Only the server knows the complete game state and the different clients' scopes. The server individually updates each client only within their scope. Therefore the data received by each client contains only information about his or her scope, not the other clients, until the scopes from two different clients interfere with each other in the server's game world. If you want to cheat, you usually want to know information about your opponent before your opponent can learn about your state; however, as this information lies in other client's scope – until you reach the interaction distance you can not know the state of the opponent. So, because you do not receive that information, you would have to hack the game's server to get this information and this is likely to be detected. If you can't directly eavesdrop another client's traffic, this makes it difficult to have global knowledge of the game state. Therefore a server-client model is better, if the developer wishes to avoid players cheating in the game.

This is also the solution we are using for the development of the game in this thesis and the one the selected networking solution offers.

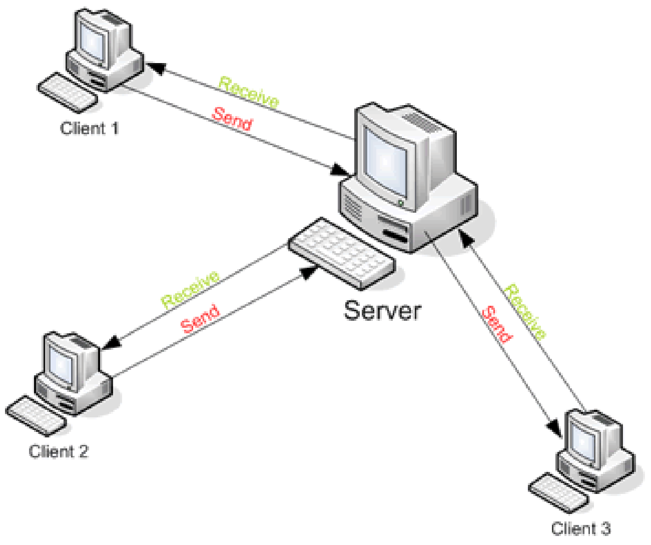


Figure 2.11: Client-server architecture

Network server

To avoid bottlenecks a network server based architecture is ideal for Massively multiplayer online role-playing game (MMORPG) games which is a game genre with many participants connecting to the same server. The clients connect to one or more servers, which are in turn interconnected with each other through a local network. The local network enables the servers to exchange a huge amount of data very quickly. This model allows many clients to be connected to a server without causing saturation of a single server.

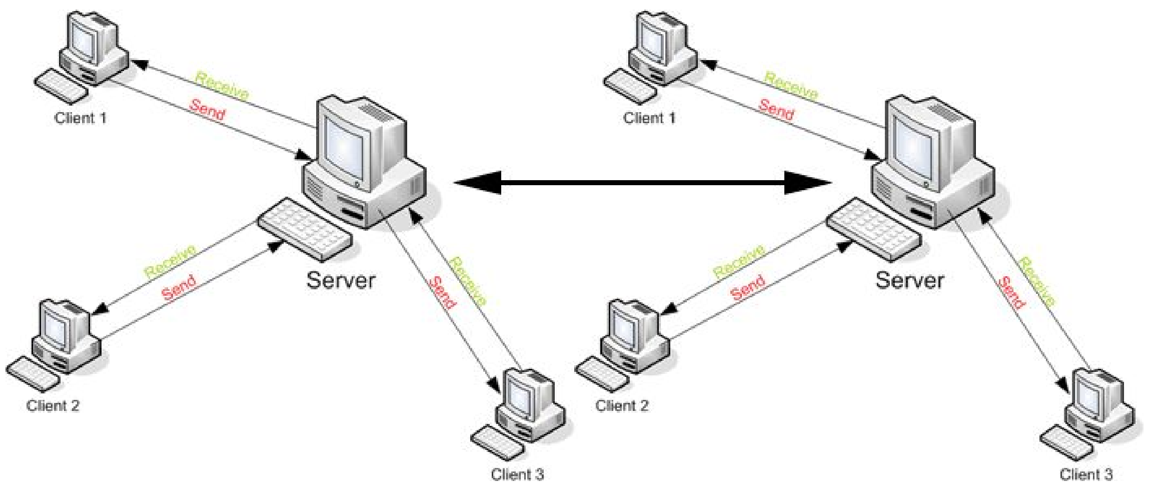


Figure 2.12: Network server architecture

2.6.4 Unity3D Networking Solutions

Since we chose Unity3D as our game engine, we are going to explore this engine's options for integrating networking in the game. There are two popular choices, UNET(Unity's built-in networking) and (PUN) Photon Unity Networking plugin. We will examine them further below.

UNET

UNET is the built-in solution for networking in the Unity game engine. It provides a different API for two kinds of users:

- Users making a Multiplayer game with Unity. Users in this category should start with the High Level API (HLAPI).
- Users building network infrastructure or advanced multiplayer games. Users in this category should start with the NetworkTransport API.

The High Level API (HLAPI) is a system for building multiplayer capabilities for Unity games. It is built on top of the lower level transport real-time communication layer, and handles many of the common tasks that are required for multiplayer games. While the transport layer supports any kind of network topology, the HLAPI is a server authoritative system; although it allows one of the participants to be a client and the server at the same time, so no dedicated server process is required. Working in conjunction with the internet services, this allows multiplayer games to be played over the internet with little work from developers. Using the HLAPI means you get access to commands which cover most of the common requirements for multi-user games without needing to worry about the "lower level" implementation details. The HLAPI allows you to:

- Control the networked state of the game using a “Network Manager”.
- Operate “client hosted” games, where the host is also a player client.
- Serialize data using a general-purpose serializer.
- Send and receive network messages.
- Send networked commands from clients to servers.
- Make remote procedure calls (RPCs) from servers to clients.
- Send networked events from servers to clients.

The HLAPI is built from a series of layers that add functionality as shown below:

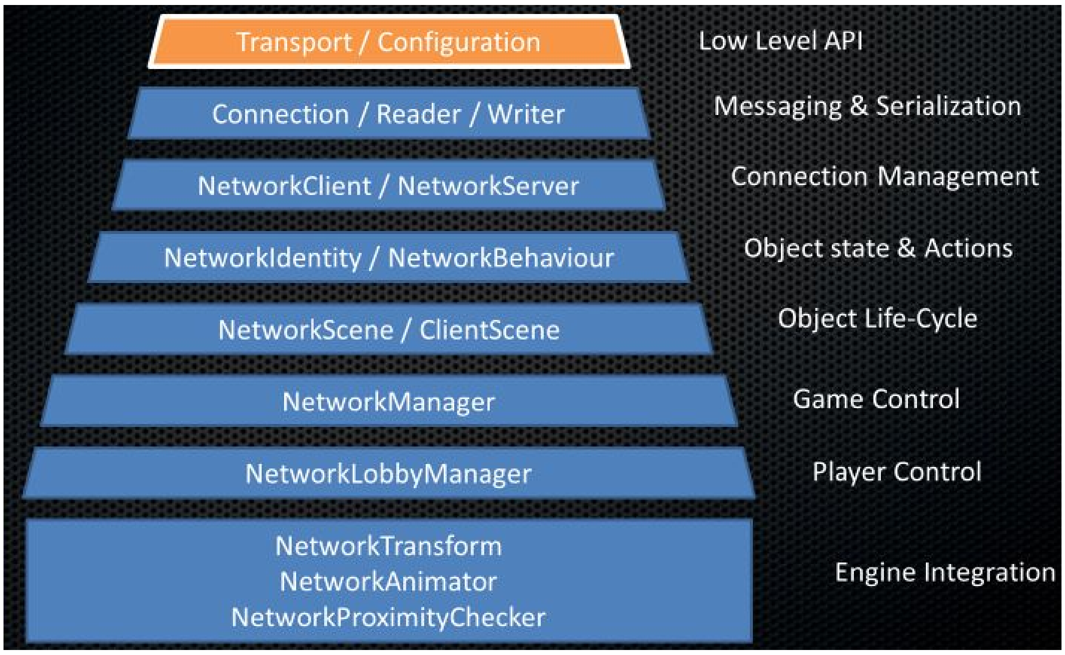


Figure 2.13: UNET layers

Unity’s networking is integrated into the engine and the editor, allowing you to work with components and visual aids to build your multiplayer game. It provides:

- A NetworkIdentity component for networked objects.
- A NetworkBehaviour for networked scripts.
- Configurable automatic synchronization of object transforms.
- Automatic synchronization of script variables.
- Support for placing networked objects in Unity scenes.
- Network components

Unity offers Internet Services to support your game throughout production and release, which includes:

- Matchmaking service
- Create matches and advertise matches.
- List available matches and join matches.

- Relay server
- Game-play over internet with no dedicated server.
- Routing of messages for participants of matches.

PUN

Photon Unity Networking (PUN) is a Unity package for multiplayer games. It provides authentication options, matchmaking and fast, reliable in-game communication through its Photon backend. It provides basically everything UNET offers in its HLAPI such as synchronization, RPC calls and transfer of messages.

PUN offers:

- Realtime Cloud. PUN games are hosted in the globally distributed Photon Cloud to guarantee low latency and shortest round-trip times for the players worldwide.
- Cross-platform. It does not matter if the game is developed for mobile, desktop or any other platform.
- Highest scalability. You can seamlessly and automatically scale the number of users.
- Matchmaking API. Players can be matched randomly, by parameterized searches or they can pick a room if you create a list for them to choose from.
- Customization. You have access to the source code so, any change you want to make you can easily do it.
- Client-server architecture. No punch-through issues whether with Reliable UDP, TCP, HTTP or Websockets.
- Flexibility.
- Free(for 20 CCU - concurrent users).

Also, one of its key features is the different variety of services you can easily implement along with it, such as Photon Voice for in-game networking communication and chat.

2.6.5 Selecting the Networking solution

Both solutions seem quite good at first glance. But let's have a look at the main differences between them:

Host Model

Although both solutions are server-client based, UNET server runs via a Unity client(that is one of the players), while PUN has a dedicated server. What this means is that, with UNET if the client-server's connection drops then the whole system goes down. Something that does not happen with PUN since all connections go through the server.

Connectivity

Unity networking works with NAT punch-through to try to improve connectivity: since players host the network servers, the connection often fails due to firewalls/routers etc. Connectivity can never be guaranteed, there is a low success rate. Photon has a dedicated server, there is no need for NAT punch-through or other concepts. Connectivity is a guaranteed 100%. If, in the rare case, a connection fails it must be due to a very strict client-side network (a business VPN for example).

Performance

Photon beats Unity networking performance wise. Since the Unity servers are player hosted latency/ping is usually worse; you rely on the connection of the player acting as server. These connections are never any better than the connection of the dedicated Photon server.

Features & Maintenance

Unity does not seem to give much priority to their Networking solution. There are rarely feature improvements and bugfixes are as seldom. The Photon solution is actively maintained and parts of it are available with source code. Furthermore, Photon already offers more features than Unity, such as the built-in load balancing and offline mode.

Master Server

The Master Server for Photon is a bit different from the Master Server for plain Unity Networking: In the second case, it's a Photon Server that posts room-names of currently played games in so called "lobbies". Like Unity's Master, it will forward clients to the Game Server(s), where the actual gameplay is done.

Community Support

Due to the reasons above and more, the majority of the developing community chooses PUN over the UNET for their networking thus offering more support for newcomers in the forums.

For all the reasons listed and after trying out UNET ourselves before making the final choice, Photon Unity Networking seemed like the better solution for our project. It is more robust and complete with tutorials and detailed documentation to help each step of the way.

2.7 Real-time Multiplayer Games' Genres

A multiplayer video game is a video game in which more than one person can play in the same game environment at the same time. Video games are often single-player activities, putting the player against preprogrammed challenges or AI-controlled opponents (which lack the flexibility of human thought). Multiplayer games allow players interaction with other individuals in partnership, competition or rivalry, providing them with social communication absent from single-player games. In multiplayer games, players may compete against two (or more) human contestants, work cooperatively with a human partner to achieve a common goal, supervise other players' activity, co-op, and objective-based modes assaulting (or defending) a control point. Multiplayer games typically require players to share the resources of a single game system or use networking technology to play together over a greater distance.

Ken Wasserman and Tim Stryker in 1980 identified three factors which make networked computer games appealing [13]:

1. multiple humans competing with each other instead of a computer
2. incomplete information resulting in suspense and risk-taking
3. real-time play requiring quick reaction

Real time multiplayer games can be divided into many genres. Below we will see the most popular ones and the basic ideas behind them.

2.7.1 TBS

Turned Based Strategy (TBS) games are best compared to regular board games, such as chess or monopoly. Each player takes his turn acts based upon this user's strategy, for example raise an army or explore a map, then the next player takes their turn. Usually the timeline of the game advances at each turn. While the player waits for his turn, he or she can't interact with the game, although this player may be computing their response to each of the other players' possible actions. Such a game requires only a modest bounded latency of the network. Some popular games in TBS genre have been Civilization, different versions of Worms and more.

2.7.2 RTS

In Real Time Strategy (RTS) games the player has to gather resources, building a base, raising an army, upgrading weapons, and take control of specific units, then act. RTS games don't involve turns like the more classic TBS games. Each player has control of his army in real time, thus he or she can at anytime interact with the game, rather than needing to wait for his or her turn. This results in more action, but each player still has to plan their strategy and tactics (far) in advance. Early in the game the player has to make a decision: what buildings he is going to construct and what kind of army he wants to raise. He also has to consider if he is going to rush the enemy, or wait until he has a really big army before attacking. As a result, the outcome of such games is more dependent on slow execution of strategies and tactics, rather than the player's reaction time hence latency isn't that crucial in such real time strategy based multiplayer session. Additionally the outcome of the game is not influenced significant by the latency experienced by the players. Popular games in RTS have been Age of Empires Starcraft, Warcraft 3 and more.

2.7.3 FPS

In First Person Shooter (FPS) games the gamer sees his character from a first person point of view. He is the character and sees the game world through this character's eyes. Usually the player has to move about in a virtual world, finding different weapons, which he then uses for killing alien, monsters, or in the case of a multiplayer game other players. In such games the gamer would like to have as close real time experience as possible. This is because the gamer's reactions are fundamental to their outcome of the game. If he experiences greater latency he may, in the worst case, already be dead before he even notices the shot. This obviously requires a very low latency. Popular games in this genre have been Counter Strike, Call of Duty, Overwatch and more.

2.7.4 MMORPG

Massively multiplayer online role-playing games (MMORPG) have become very popular over the years. The idea is that a large number of players interact with each other in a virtual world. The gamer creates a character and enters the virtual world with him or her and take control of his/her abilities. Usually the virtual world is a fantasy world where magic and strange creatures exist. The goal of the game is for the gamer to develop his character by gaining experience which gives the character stronger abilities and hence can fight harder monsters and accomplish more complex quests. The interaction among the players is usually by completing quests together or by changing, buying or selling items among each other or fighting. An interesting phenomenon is that these games have created a virtual economy. Money gained by killing monsters, can for example be used for buying items to your character. But this virtual economy has also impacted the real economy by gamers selling virtual money for real money. Also different items in the games can be bought for real money on auction sites such as Ebay. Popular games include World of Warcraft, RuneScape, Guild Wars 2 and more.

3 Utilized Software

3.1 Unity3D Game Engine

Unity 3D is a powerful cross-platform 3D game engine with a user friendly development environment. Unity 3D helps developers create games and applications for mobile, desktop, the web, and consoles. Its 3D environment is suitable for laying out levels, creating menus, doing animation, writing scripts, and organizing projects. Unitys primary goal may be the development of 3D video games, however, it is also suitable to create other kinds of interactive content, such as animations, simulations or 3D visualizations.

Unity is a fully integrated development engine that provides functionality to create interactive 3D content. With Unity the developer can assemble assets into scenes and environments, add lighting, audio, special effects, physics and animation, simultaneously play, test and edit the application, and when ready, publish to chosen platforms, such as Mac, PC and Linux desktop computers, Windows, the Web, iOS, Android, Windows Phone 8, Blackberry 10, Wii U, PS3 and Xbox 360. Unitys complete toolset, intuitive workspace and rapid, productive workflows help users to drastically reduce the time, effort and cost of making interactive content.

Below we are going to see some of Unity's key components in more detail.

3.1.1 Hierarchy

The Hierarchy window contains a list of every GameObject in the current Scene. Some of these are direct instances of Asset files (like 3D models), and others are instances of Prefabs, which are custom objects that make up most of the game. As objects are added and removed in the Scene, they will appear and disappear from the Hierarchy as well. By default, objects are listed in the Hierarchy window in the order they are made. You can reorder the objects by dragging them up or down, or by making them "child" or "parent" objects.

Unity uses a concept called Parenting. When you create a group of objects, the topmost object or Scene is called the "parent object", and all objects grouped underneath it are called "child objects" or "children". You can also created nested parent-child objects (called "descendants" of the top-level parent object).

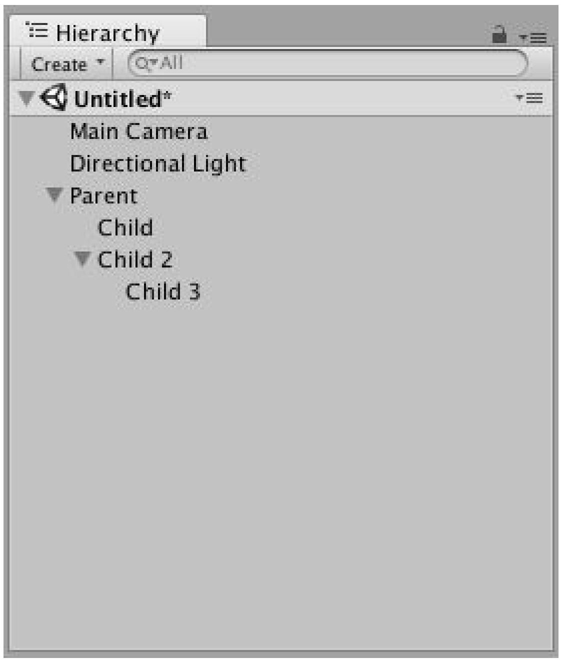


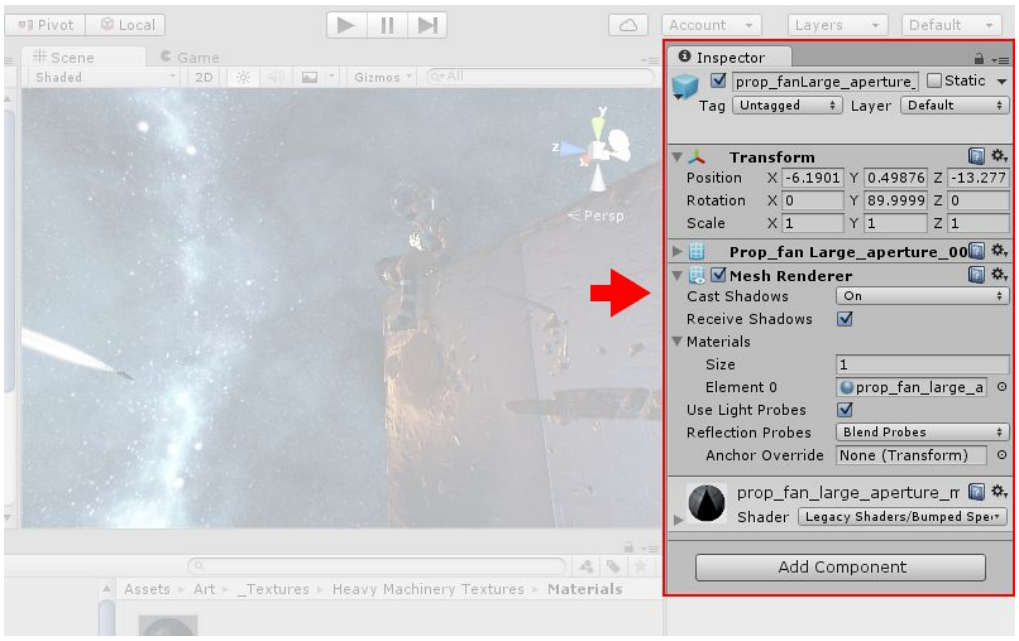
Figure 3.1: In this image, Child and Child 2 are the child objects of Parent. Child 3 is a child object of Child 2, and a descendant object of Parent.

3.1.2 Inspector

Projects in the Unity Editor are made up of multiple GameObjects that contain scripts, sounds, Meshes, and other graphical elements such as Lights. The Inspector window (sometimes referred to as “the Inspector”) displays detailed information about the currently selected GameObject, including all attached components and their properties, and allows you to modify the functionality of GameObjects in your Scene.

When you select a GameObject in either the Hierarchy or Scene view, the Inspector shows the properties of all components and Materials of that GameObject. You can then examine and edit those properties to your liking.

When GameObjects have custom script components attached, the Inspector displays the public variables of that script. You can edit these variables as settings in the same way you can edit the settings of the Editor’s built-in components. This means that you can set parameters and default values in your scripts easily without modifying the code.



```
prop_fanLarge_aperture_   Static
Tag Untagged   Layer Default

Transform
Position   X -6.1901   Y 0.49876   Z -13.277
Rotation   X 0         Y 89.9999   Z 0
Scale      X 1         Y 1         Z 1

Prop_fan Large_aperture_00

Mesh Renderer
Cast Shadows        On
Receive Shadows     ✓
Materials
  Size              1
  Element 0         prop_fan_large_a
Use Light Probes    ✓
Reflection Probes   Blend Probes
  Anchor Override   None (Transform)

prop_fan_large_aperture_m
Shader   Legacy Shaders/Bumped Spec
```

Figure 3.2: The Inspector window

3.1.3 GameObjects & Components

GameObjects are the fundamental objects in Unity that represent characters, props and scenery. Every object in the game is a GameObject. This means that everything you can think of to be in your game has to be a GameObject. However, a GameObject can't do anything on its own; you need to give it properties before it can become a character, an environment, or a special effect. A GameObject is a container; you add pieces to the GameObject container to make it into a character, a light, a tree, a sound, or whatever else you would like it to be. Each piece you add is called a component.

In the image below, we see a solid cube object has a Transform component to dictate its position and existence in the virtual space, a Mesh Filter and a Mesh Renderer component to draw the surface of the cube, and a Box Collider component to represent the object's solid volume in terms of physics.

Some of the most used Components in Unity are:

Transform Component

Every GameObject contains a Transform component. When creating a GameObject a Transform component is added automatically. It is impossible to create a GameObject without one or remove it. The Transform component is one of the most important and most frequently accessed components. It defines a GameObject's position, rotation, and scale in the game world based on the x,y,z coordinate system. These parameters are initialized by hand and/or can modified in runtime by script to make objects move, rotate and more. It is important to note that when scripting functionality such as movement, Unity considers the Z axis as forward/backwards , Y axis as up/down and X axis as left/right.

Mesh Components

3D meshes are the main graphic object primitive of Unity. Various components exist in Unity to render meshes. The most commonly used components are the Mesh Renderer and the Mesh Filter. They are used in collaboration in order to display an object. The Mesh Filter takes a mesh from your assets and passes it to the Mesh Renderer for rendering on the screen. The Mesh Renderer takes the geometry from the Mesh Filter and renders it at the position defined by the object's Transform component. When importing mesh assets, Unity automatically creates a Mesh Filter along with a Mesh Renderer. Another component is the Text Mesh. It generates 3D geometry that displays text strings.

Physics Components

Physics components allow the user to give objects realistic motion and reaction to collisions by simulating physics laws. Unity has NVIDIA PhysX physics engine built-in. A physics engine is a computer software that provides an approximate simulation of physical systems. This allows for unique realistic behavior and has many useful features. A Rigidbody component makes the object that is attached to be affected by gravity or linear and angular forces and collide with other objects. There is also a variety of Collider components (Mesh, Box, Sphere, Wheel collider) which surround the shape of an object for the purposes of detecting physical collisions.

Rendering Components

These are the components that have to do with rendering in-game and user interface elements, as well as lighting and special effects. The Camera component is essential as it is used to capture and display the world to the player. It can be customized and manipulated to fulfill the requirements of the user's application. The GUI Texture and GUI Text components are made especially for user interface elements, buttons, or decorations as well as displaying text on screen. Another important rendering component is the light component. It brings a sense of realism. Lights can be used to illuminate the scenes and objects, to simulate the sun, flashlights, or explosions just to name a few.

Audio Components

These components are used to implement sound. The most important component here, is the Audio Source component, which as the name suggests, plays a sound file at the location of the game object it is attached to. The developer can set parameters such as sound volume, pitch and change the sound file to be played at any time. These parameters can also be changed by script during runtime.

Script Component

The Script component is used to attach a script onto a game object. Scripts are often attached to objects, to define their behavior and trigger effects upon specified conditions. More about scripts in the following section.

3.1.4 Materials, Shaders & Textures

Rendering in Unity is done with Materials, Shaders and Textures. There is a close relationship between these three components in Unity.

Materials are definitions of how a surface should be rendered, including references to textures used, tiling information, colour tints and more. The available options for a material depend on which shader the material is using.

Shaders are small scripts that contain the mathematical calculations and algorithms for calculating the colour of each pixel rendered, based on the lighting input and the Material configuration.

Textures are bitmap images. A Material may contain references to textures, so that the Material's shader can use the textures while calculating the surface colour of an object. In addition to basic colour (albedo) of an object's surface, textures can represent many other aspects of a material's surface such as its reflectivity or roughness.

A material specifies one specific shader to use, and the shader used determines which options are available in the material. A shader specifies one or more textures variables that it expects to use, and the Material Inspector in Unity allows you to assign your own texture assets to these these texture variables. For most normal rendering - by which we mean characters, scenery, environments, solid and transparent objects, hard and soft surfaces etc., the Standard Shader is usually the best choice. This is a highly customisable shader which is capable of rendering many types of surface in a highly realistic way. There are other situations where a different built-in shader, or even a custom written shader might be appropriate - such as liquids, foliage, refractive glass, particle effects, cartoony, illustrative or other artistic effects, or other special effects like night vision, heat vision or x-ray vision, etc.

3.1.5 Scripting

Scripting is an essential part of Unity as it defines the entire behavior of the game or application. Even the simplest game needs a script to respond to input. Scripts can be used to create graphical effects, control physical behavior of objects or characters, trigger effects upon specified conditions and generally bring a game to life. Unity supports three programming languages: C# which is similar to C++/Java, Javascript and Boo. The scripts can be written and edited in MonoDevelop, which is an integrated development environment (IDE) within Unity. An IDE combines a text editor with additional features for debugging, auto-complete and other project management tasks.

Scripting is linked with the component based architecture Unity uses. As it was mentioned above, the behavior of GameObjects is controlled by the components that are attached to them. Thus, components can be accessed or modified by script at any time to achieve desired behavior and functionality. Each script makes its connection with the internal workings of Unity by implementing a built-in class called MonoBehaviour. This class refers to the component that can be enabled or disabled. Javascript automatically derives from the class without the need to be declared, whereas the other two languages have to explicitly declare the class.

When a script is created, there are two functions automatically declared in it, Start() and Update().

- The Start function is the place where initialization occurs. It is called when the GameObject is enabled and is used to initialize an object's position, state and properties or load other scripts and GameObjects for later use.
- Update is the function that implements game behavior. It is called in every frame and is crucial for checking and modifying the state of various parts of the application. From a programming standpoint each game or application runs in loop, which allows it to run smoothly regardless of a user's input or lack thereof. For a change to occur an event must be activated. The Update function checks every frame for such events, which can be changes to position, state and behavior and determines the outcome. The Start function is called by Unity before the Update function is called for the first time.

There are other kinds of event functions that can trigger a change. For instance, there are the input event functions, that track the mouse movement and input. These functions allow a script to react to user actions with the mouse. Some examples of those functions are OnMouseDown, which is called when the user has pressed the mouse button, OnMouseUp which is called when the user has released the mouse button, OnMouseEnter which is called when the mouse enters a GUI element, etc. Another important function is the OnGUI function. Unity has a system for rendering Graphical User Interface (GUI) controls over the main action in the scene and responding to clicks on these controls. The code handling those events is treated somewhat differently from the normal frame update and is placed in the OnGUI function, which is called multiple times per frame update.

Apart from the functions provided by Unity, the developer can create his/her own functions in order to control or determine the behavior of a GameObject, change the properties of a component or altering the overall state of the application. In order for these custom functions to be executed, they have to be called inside a Unity event function, like Update.

The most commonly used functions were presented briefly above, as well as the concept of how they are used. The basic notion of the Unity scripting is that the scripts are components that can control the GameObject. Each component property corresponds to a script variable and the scripts can access not only the components of the GameObjects they are attached to, but also other GameObjects.

3.1.6 Scene

The Scene window is the interactive view into the world that the user is creating. Scene View can be used to select and position scenery, characters, cameras, lights, and all other types of Game Objects. Being able to Select, manipulate and modify objects in the Scene View are some of the first skills somebody will need to begin his first steps in Unity.

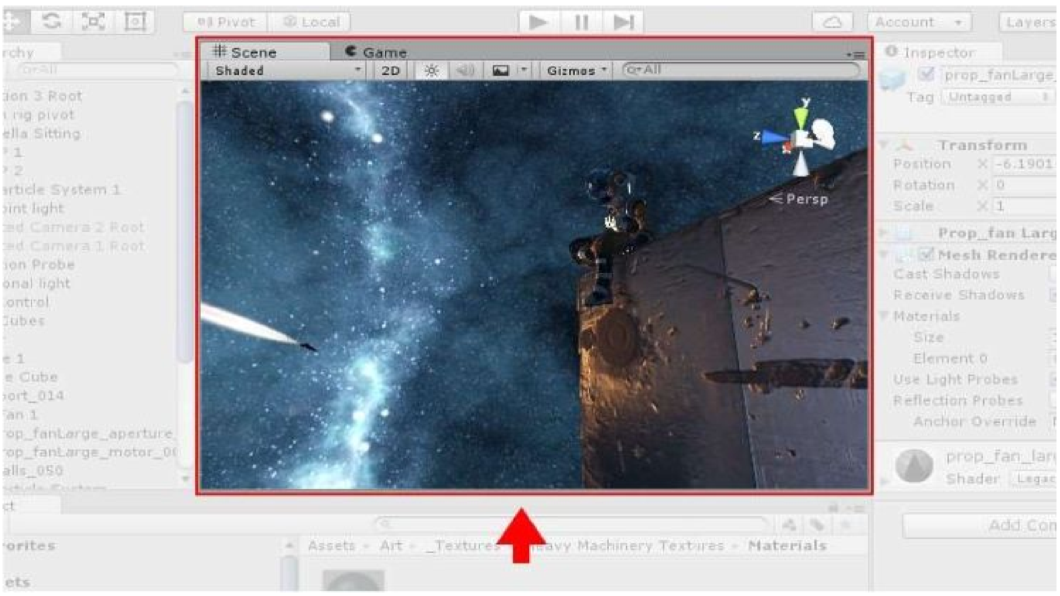


Figure 3.3: The Scene window

The Scene Gizmo is in the upper-right corner of the Scene view. This displays the Scene view Camera's current orientation, and allows the user to quickly modify the viewing angle and projection mode. In order to Move, Rotate, Scale, or Transform individual GameObjects, the user can use the four Transform tools in the toolbar in the upper-left corner outside of the Scene view. Each has a corresponding Gizmo that appears around the selected GameObject in the Scene view. To alter the Transform component of the GameObject, the user can use the mouse to manipulate any Gizmo axis, or type values directly into the number fields of the Transform component in the Inspector.

The Scene view control bar provides the user with the opportunity to select between various options for viewing the Scene and also control whether lighting and audio are enabled. These controls only affect the Scene view during development and have no effect on the built game.

3.2 Virtual Reality Tools

3.2.1 OpenVR SDK

The OpenVR API provides a game with a way to interact with Virtual Reality displays without relying on a specific hardware vendor's SDK. It can be updated independently of the game to add support for new hardware or software updates.

The API is implemented as a set of C++ interface classes full of pure virtual functions. When an application initializes the system it will return the interface that matches the header in the SDK used by that application. Once a version of an interface is published, it will be supported in all future versions, so the application will not need to update to a new SDK to move forward to new hardware and other features.

The API is broken down into six primary interfaces in the VR namespace:

- IVRSystem - Main interface for display, distortion, tracking, controller, and event access.
- IVRChaperone - Provides access to chaperone soft and hard bounds.
- IVRCompositor - Allows an application to render 3D content through the VR compositor.
- IVROverlay - Allows an application to render 2D content through the VR compositor.
- IVRRenderModels - Allows an application access to render models.
- IVRScreenshots - Allows an application to request and submit screenshots.

3.2.2 SteamVR Plugin

The SteamVR Plugin allows developers to target a single interface that will work with all major virtual reality headsets from seated to room scale experiences. Additionally, it provides access to tracked controllers, chaperoning, render models for tracked devices. SteamVR's compositor allows you to preview your content in VR using Unity's play mode, while leaving the normal game window to act as your companion screen on the main monitor. It is mainly used to implement Steam support in the game and enable easy access to the OpenVR SDK.

Some of the key features SteamVR Plugin includes are already made prefabs and scripts to help you transform the game into a complete VR experience. Some of these prefabs and features are listed below:

[CameraRig] - This is the camera setup used in most VR scenes. It is simply a default camera with the SteamVR_Camera component added to it and some more complementary components. It also includes a full set of Tracked Devices which will display and follow any connected tracked devices (e.g. controllers, base stations and cameras).

[SteamVR] - This object controls some global settings for SteamVR, most notably Tracking Space. This object is created automatically on startup if not added and defaults to Standing Tracking Space. It also provides the ability to set special masks for rendering each eye (in case you want to do something differently per-eye) and some simple help text that demonstrates rendering only to the companion window (which can be cleared or customized here).

Camera layering: One powerful feature of Unity is its ability to layer cameras to render scenes (e.g. drawing a skybox scene with one camera, the rest of the environment with a second, and maybe a third for a 3D hud). This is performed by setting the latter cameras to only clear the depth buffer, and leveraging the cameras' cullingMask to control which items get rendered per-camera, and depth to control order.

Camera scale: Setting SteamVR_Camera's GameObject scale will result in the world appearing (inversely) larger or smaller. This can be used to powerful effect, and is useful for allowing you to build skybox geometry at a sane scale while still making it feel far away. Similarly, it allows you to build geometry at scales the physics engine and Nav Mesh generation prefers, while visually feeling much smaller or larger. Of course, if you are building geometry to real-world scale you should leave this at its default of 1,1,1. Once a SteamVR_Camera has been expanded, its "origin" Transform should be scaled instead.

Camera masking: By manually adding a GameObject with the SteamVR_Render component on it to your scene, you can specify a left and right culling mask to use to control rendering per eye if necessary.

Events: SteamVR fires off several events. These can be handled by registering for them through SteamVR_Events.<EventType>.Listen. Be sure to remove your handler when no longer needed. The best pattern is to Listen and Remove in OnEnable and OnDisable respectively.

3.2.3 Virtual Reality Toolkit (VRTK)

VRTK is a collection of useful scripts and concepts to aid building VR solutions rapidly and easily in Unity3D.

It covers a number of common solutions such as:

- Support for SteamVR and Oculus SDK
- VR Simulator allowing building for VR without the need of VR hardware
- Laser Pointers on controllers and headset
- Curved Pointers on controllers and headset
- Play area cursors
- Pointer interactions
- Snap drop zones for objects
- Body Physics that supports leaning over objects
- Teleporting
- Dash Movement
- Touchpad Movement
- Move in place/Run in place Movement
- Climbing
- Object interactions: touching, grabbing using objects
- Two hand manipulation of objects: hold guns with two hands, resize objects, etc.
- Highlighting objects
- Controller haptic feedback
- Controller effects: highlighting, opacity, visibility
- 3D controls such as buttons, levers, doors, drawers, sliders, knobs. etc
- Panel menus
- Radial menus
- Interacting with Unity3D UI elements with pointers or real world collisions
- Drag and drop Unity3D UI elements around canvases and into special drop zones

3.3 Photon Unity Networking

As mentioned in section 2.6.5, the networking solution chosen is PUN (Photon Unity Networking). Below we are going to look into some of its core elements.

3.3.1 Matchmaking

Getting into a room to play with (or against!) someone is very easy with Photon. There are basically three approaches: Either tell the server to find a matching room, follow a friend into their room, or get a list of rooms to let the user pick one. For modern games it's best to use quick and painless server-side matchmaking, so using room listings is discouraged.

By design, Photon Cloud is region locked. That means all clients that are connected must be in the same region, regardless of device or platform in order for the matchmaking to take place. Several parameters can be applied for the rooms, such as max number of players, waiting period until a certain amount of players is reached, etc. Below we will briefly mention some of Photon's most popular matchmaking options.

Matchmaking Checklist

This matchmaking option allows the players to choose which room to join.

Random Matchmaking

In this matchmaking solution, as the name implies, the players are joined into rooms randomly(without asking them as in the previous solution). If a room exists they will join, otherwise one is automatically created.

Not So Random Matchmaking

Totally random matchmaking is not always something players will enjoy. Sometimes you just want to play a certain map or mode (two versus two, etc.). In Photon Cloud you can set arbitrary "Custom Room Properties" and use them as filter in `JoinRandom`.

Matchmaking Slot Reservation

Sometimes, a player joins a room, knowing that a friend should join as well. With Slot Reservation, Photon can block a slot for specific users and take that into account for matchmaking. To reserve slots there is an `expectedUsers` parameter in the methods that get you in a room (`JoinRoom`, `JoinOrCreateRoom`, `JoinRandomRoom` and `CreateRoom`).

When you know someone should join, you pass an array of UserIDs. For `JoinRandomRoom`, the server will attempt to find a room with enough slots for you and your expected players (plus all active and expected players already in the room). The server will update clients in a room with the current `expectedUsers`, should they change. To support Slot Reservation, you need to enable publishing UserIDs inside rooms.

Lobbies

Photon is organizing the rooms in so called "lobbies". There is a default lobby but the clients can create new ones on the fly. Lobbies begin to exist when you specify a lobby in `CreateRoom`. Like rooms, lobbies can be joined. In a lobby, the clients only get the room list of that lobby. Nothing else. There is no way to communicate with others in a lobby.

Thus, it is better practice to not join lobbies: Often clients just get a long list of room names and players pick one randomly to finally start playing. There is not a lot information in a long list of room names, which have all the same ping. To give your players more control over the matchmaking, usage of filters for random matchmaking is advised. Multiple lobbies can still be useful, as they are also used in (server-side) random matchmaking.

Skill-based Matchmaking

You can use lobbies of the SQL-type to implement your own skill-based matchmaking. First of all, each room gets a fixed skill that players should have to join it. This value should not change, or else it will basically invalidate any matching the players in it did before. As usual, players should try to get into a room by `JoinRandomRoom`. The filter should be based on the user's skill. The client can easily filter for rooms of "skill +/- X". `JoinRandomRoom` will get a response immediately as usual but if it didn't find a match right away, the client should wait a few seconds and then try again. You can do as many or few requests as you like. Best of all: The client can begin to relax the filter rule over time.

It's important to relax the filters after a moment. It should be considered that a room might be joined by a player with not-so-well-fitting skill but obviously no other room was a better fit and it's better to play with someone. You can define a max deviation and a timeout. If no room was found, this client has to open a new room with the skill this user has. Then it has to wait for others doing the same. Obviously, this workflow might take some time when few rooms are available. You can rescue your players by checking the "application stats" which tell you how many rooms are available. You can adjust the filters and the timing for "less than 100 rooms" and use different settings for "100 to 1000 rooms" and again for "even more".

3.3.2 Instantiation

In about every game you need to instantiate one or more objects per player. For objects that should synchronize in a networked game, you need to use a special workflow.

PUN can automatically take care of spawning a networked object by passing a starting position, rotation and a prefab name to the `PhotonNetwork.Instantiate` method. Requirement: The prefab should be available directly under "Resources" folder (to load it at runtime) and it must have a `PhotonView` component in order for it to be identifiable in the network.

GameObjects created with `PhotonNetwork.Instantiate` will usually exist as long as you are in the same room. When you swap rooms, objects don't carry over, just like when you switch a scene in Unity. When a client leaves a room, all others destroy the GameObjects owned/created by that player. If this doesn't fit your game logic, you can skip this step by setting `PhotonNetwork.autoCleanUpPlayerObjects` to false for your game.

Alternatively, the Master Client can create GameObjects that have the lifetime of the room by using `PhotonNetwork.InstantiateSceneObject().` The object is not associated with the Master Client but the room. By default, the Master Client controls these objects but you can pass on control with `photonView.TransferOwnership().` For ownership transfer see sections 3.3.4 and 5.2. It is perfectly fine to place PhotonViews on objects in a scene. They will be controlled by the Master Client by default and can be useful to have a "neutral" object to send room-related RPCs.

When you load a scene, Unity usually destroys all GameObjects currently in the hierarchy. This includes networked objects, which can be confusing at times. Example: In a menu scene, you join a room and load another. You might actually arrive in the room a bit too early and get the initial messages of the room. PUN begins to instantiate networked objects but your logic loads another scene and they are gone. To avoid issues with loading scenes, you can set `PhotonNetwork.automaticallySyncScene` to true and use `PhotonNetwork.LoadLevel()` to switch scenes.

There is also the option for Manual Instantiation, that means not relying on the Resources folder. The main reason for wanting to instantiate manually is gaining control over what is downloaded for streaming web players. You can send RPCs to instantiate objects. Of course you need some way to tell the remote clients which object to instantiate. You can't just send a reference to a GameObject, so you need to come up with a name or something for it. As important as the type of object, is a network id for it. The `PhotonView.viewID` is the key to routing network messages to the correct GameObject/Scripts. If you spawn manually, you have to allocate a new `viewID` using `PhotonNetwork.AllocateViewID()` and send it along. Everyone in the room has to set the same ID on the new object. Keep in mind that an RPC for instantiation needs to be buffered: Clients that connect later have to receive the spawn instructions as well.

3.3.3 Synchronization and State

Games are all about updating the other players and keeping the same state. You want to know who the other players are, what they do, where they are and how their game world looks. PUN offers several tools for updates and keeping a state.

Object Synchronization

With PUN, you can easily make certain game objects "network aware". Assign a PhotonView component and an object can sync the position, rotation and other values with its remote duplicates. A PhotonView must be setup to "observe" a component like a Transform or (more commonly) one of its scripts. Some script implements `OnPhotonSerializeView()` and becomes the observed component of a PhotonView. In `OnPhotonSerializeView()`, the position and other values are written to a stream and read from it.

Remote Procedure Call (RPC)

You can mark your methods to be callable by any client in a room. If you implement '`ChangeColorToRed()`' with the attribute [PunRPC], remote players can: change the GameObject's color to red by calling: `photonView.RPC("ChangeColorToRed", PhotonTargets.All);`.

A call always targets a specific PhotonView on a GameObject. So, when '`ChangeColorToRed()`' gets called, it only executes on the GameObject with that PhotonView. This is useful when you want to affect specific objects. Of course, an empty GameObject can be put into a scene as "dummy" for methods that don't have a target really. For example, you could implement a chat with RPCs but that is not related to a specific GameObject.

RPCs can be "buffered". The server will remember the call and send it to anyone who's joining after the RPC got called. This enables you to store some actions and to implement an alternative to `PhotonNetwork.Instantiate()`. A drawback is that the buffer will grow and grow, if you are not careful.

To decide which synchronization method is best for a value, it's usually a good idea to check how often it needs an update and if it needs a "history" of values or not.

For frequent updates, use Object Synchronization. In doubt, your own script can skip updates by not writing anything into the stream for any number of updates. Positions for characters change frequently. Each update is useful but is likely to be replaced by a newer one quickly. A PhotonView can be setup to send "Unreliable" or "Unreliable On Change". The first will send updates in a fixed frequency - even if the character did not move. The latter will stop sending updates when the GameObject (character, unit) rests.

Changing equipment on a character, using a tool or ending a turn of a game are all infrequent actions. They are based on user input and probably best sent as RPC. The line to using Object Synchronization is not a very clear one. If you do Object Synchronization anyways, it can make a lot of sense to "inline" some actions with the more frequent updates. As example: If you send a character's position anyways, you can easily add a value to send a "Jumping" state along. This does not have to be a separate RPC then.

RPCs are not sent in the moment when you call `photonView.RPC("rpcName", ...)`. Instead, they are buffered until the Object Synchronization frequency sends an update anyways. This aggregates RPCs into less packages (avoiding overhead) but introduces some variable lag. To avoid this local lag, you could finish an update loop with RPCs by calling `PhotonNetwork.SendOutgoingCommands()`. This makes sense when your game relies a lot on RPCs to send turns, etc.

Unlike Object Synchronization, RPCs might be buffered. Any buffered RPC will be sent to players who join later, which can be useful if actions have to be replayed one after another. For example a joining client can replay how someone placed a tool in the scene and how someone else upgraded it. The latter depends on the first action. Sending buffered RPCs to new players takes some bandwidth and it means your clients have to play back and apply each action before they get into the "live" gameplay. This can be annoying and excess buffering might break weak clients, so use buffering with care.

3.3.4 Ownership Transfer

Ownership Transfer allows you to pass control of any networked object. In PUN, every object can only be controlled by one client. When a client instantiates something, it will be the owner of that new object, `PhotonView.isMine` is true on that client only and if you use `OnPhotonSerializeView`, only this client will write to the PhotonStream. The others just receive and update accordingly. If you want to pass control of a GameObject to another client, you will first have to configure the PhotonView. The object that can change ownership must be set as “Takeover” or “Request”. “Takeover” means that any client can take the GameObject from the current owner. “Request” asks the owner to pass ownership over and can be rejected.

Ownership Transfer by itself should be relatively straightforward. Control of GameObjects can be requested and transferred and if the current owner is gone, the Master Client takes over. The GameObject's lifetime is not affected when control changes. By default, all GameObjects created by one player will be destroyed when he/she leaves.

RPCs are not bound to the lifetime of the GameObject. If anyone uses RPCs on a GameObject he/she did not instantiate, (buffered) RPCs might still be sent to joining players. In some cases, this can be ignored but you should be aware of this. Also, you can clean up those RPCs corresponding to a PhotonView or the player who left.

3.4 Other tools

Autodesk 3ds Max (Student Version)

Autodesk 3ds Max, formerly 3D Studio, then 3D Studio Max is a professional 3D computer graphics program for making 3D animations, models, games and images. It is developed and produced by Autodesk Media and Entertainment. It has modeling capabilities and a flexible plugin architecture and can be used on the Microsoft Windows platform. It is frequently used by video game developers, many TV commercial studios and architectural visualization studios. It is also used for movie effects and movie pre-visualization. For its modeling and animation tools, the latest version of 3ds Max also features shaders (such as ambient occlusion and subsurface scattering), dynamic simulation, particle systems, radiosity, normal map creation and rendering, global illumination, a customizable user interface, new icons, and its own scripting language.

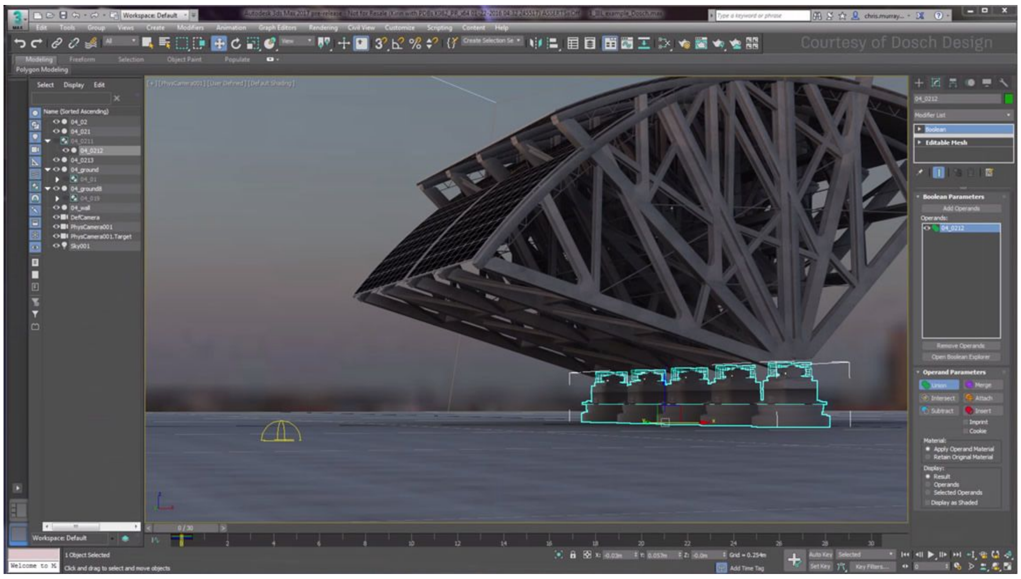


Figure 3.4: 3ds Max window

In this project, 3ds Max was used to model some basic models such as the bottom of the lake along with the pipe system, and the top of the cave. Since Unity’s terrain creation system does not allow for inverted terrains or holes through them, it was a necessary step to take.

Audacity

Audacity is a free open source digital audio editor and recording computer software application, available for Windows, OS X, Linux and other operating systems. Audacity was started in the fall of 1999 by Dominic Mazzoni and Roger Dannenberg at Carnegie Mellon University and was released on May 28, 2000 as version 0.8.

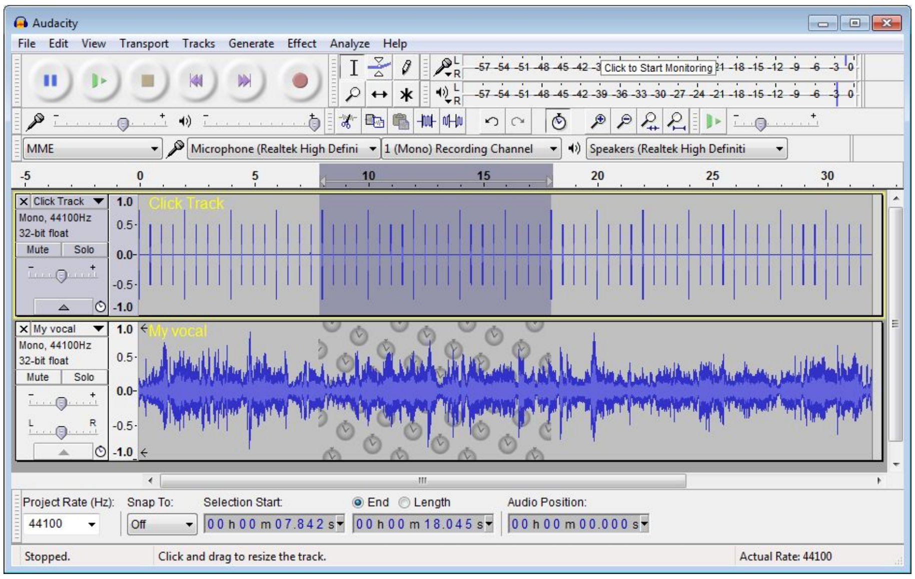

Audacity
File Edit View Transport Tracks Generate Effect Analyze Help
Click to Start Monitoring
MME
Microphone (Realtek High Defini
1 (Mono) Recording Channel
Speakers (Realtek High Definiti
Click Track
My vocal
Mute Solo
Project Rate (Hz): 44100
Snap To: Off
Selection Start: 00 h 00 m 07.842 s
End Length
00 h 00 m 18.045 s
Audio Position: 00 h 00 m 00.000 s
Stopped.
Click and drag to resize the track.
Actual Rate: 44100

Figure 3.5: The Audacity window

Audacity was used for cropping, resizing and generally manipulating free sounds in order to use them effectively in the game.

4 User View

The players are initially inserted into a pre-game non-networked tutorial level where they are presented with a basic view of their in-game interaction mechanics. They have a clear view of the actual controllers they handle in the real world and in addition, when they look at them, help text appears on the buttons of the controllers to give them a better understanding of what each button does. When they feel ready to move on, they step into a blue orb and are transferred to the networked game.

They start in a forest which gives them a first look of the other player, their controllers which have now transformed into hands to make it all more realistic and the basics of the networked environment, how interactions like picking up a rock and throwing it are synchronised, etc. They can roam around there for a while but eventually they have to move to the bridge to advance in the game. The game through networked variables understands when both of them are in the bridge and drops them in the lake below. While in there they can swim around and are required to pull two levers to move a big rock that covers the entrance to the next level. Swimming through the now open entrance, they reach a climbable wall that leads to the ruins. In this final level they have to solve three tasks which award one orb each. Each task features an element of the nature, so one task is a puzzle involving fire, one air and the other earth. Collecting all 3(plus one additional orb from an explanatory already solved task), so 4 in summary, they complete the game.

4.1 Tutorial

In every game where complex interactions are involved, it is advised to make the user go through some kind of tutorial. This is to make sure he understands the different mechanics he is going to face, how to interact with the game in general and better enjoy the experience. Based on this concept, a tutorial on how to use his VR system's controllers was introduced. All of the actions he would go through playing the game were explained in a simple and interactive way.

A pre-game scene was loaded with the user in the center and the interactable objects around him. These interactable objects were constructed from Unity primitives to keep the scene clean and simple for the purposes of the tutorial. To better help him understand how the buttons in his controllers work, a special feature was implemented. When he turned his head and looked at a controller, explanatory text connected to each button would appear, as shown below. How this is achieved is explained in chapter 5.1.2.

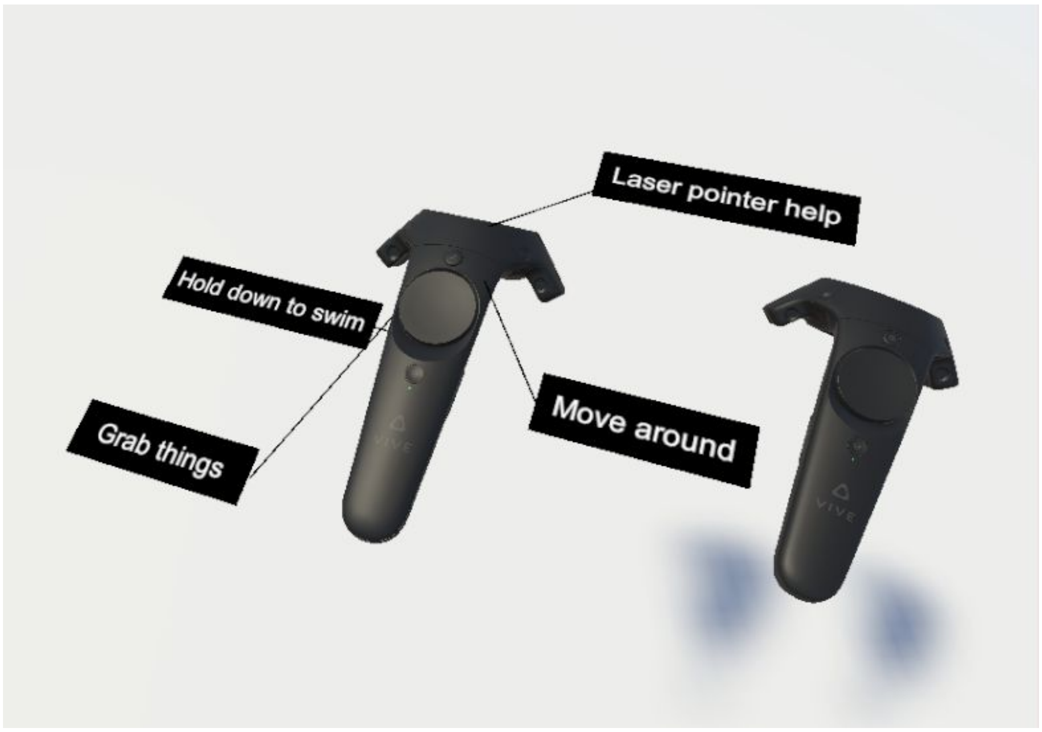


Figure 4.1: Explanatory text for controllers' buttons. Notice how our focus is on the left controller so the help text is displayed only on that controller.

This pre-game scene also included grabbable/throwable objects, locomotion and swimming. All of which were there to familiarise the user with the in-game mechanics which will be analysed in greater detail further below.

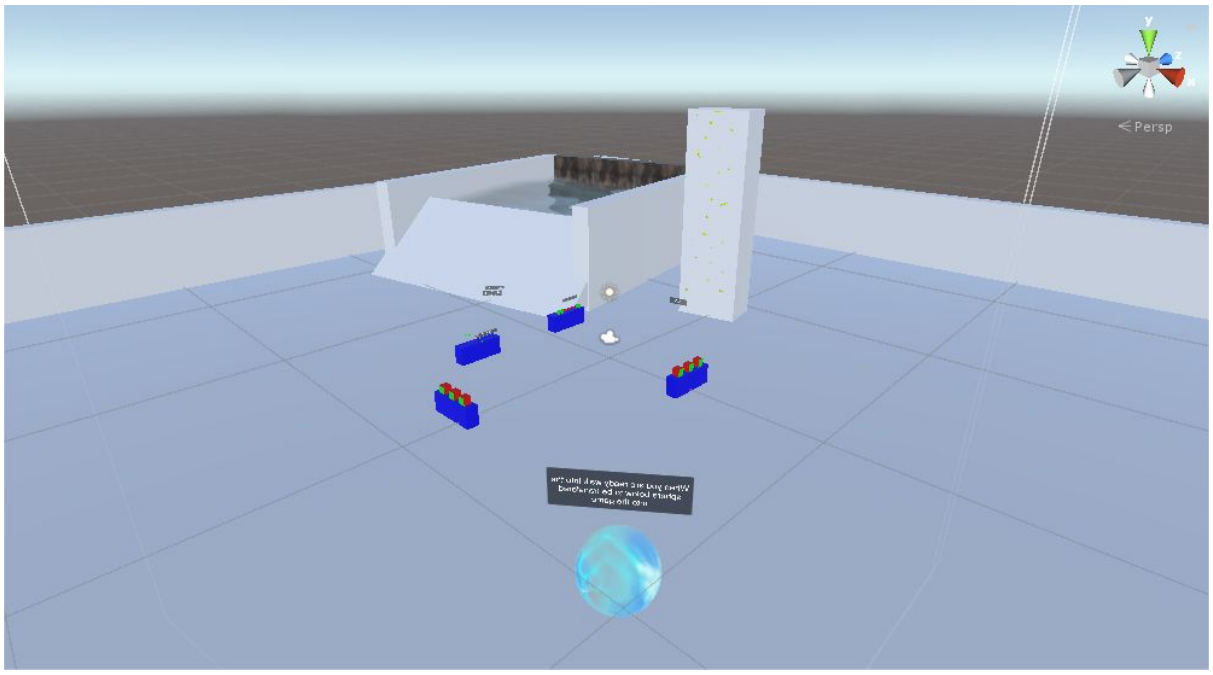


Figure 4.2: The tutorial scene

4.2 Game Environment

The game consists of quite a big and complex environment. If we were to divide it into sections, it would be composed of four main parts. For the sake of reference, we will call them “The Forest”, “The Dive”, “The Climb” and “The Ruins”. Note that this chapter explains how the scene was created from assets and models along with a brief description on the players’ walkthrough. The more technical aspects of the game like how the players are allowed certain actions in one place and others elsewhere are explained in chapter 5.

All models and terrains used in the creation of the environment have a Material attached. Materials, as was already explained, use different shaders to render different results. The most common material uses Unity’s Standard Shader which applies a color on an object or some kind of texture we import through unity’s assets. Then we can customize it to occlude certain areas of the object’s color/texture, give it some illusion with normal maps or just make it specular, and more.

4.2.1 The Forest

The Forest is composed of three Unity Terrains, each one with its own height map. Height maps, as the name suggests, are used to give height to the terrains. They are basically grayscale images with values ranging from black(#000000) to white(#ffffff). Black means the height at the certain coordinate of the terrain is 0 and white means the height is the maximum we have set in the terrain’s parameters. Differences between black and white are used to produce mountains and hills, or depending on how the game is set, crevices and ravines.

After the heightmap is applied, the terrain will most likely appear to have a jagged texture. This is due to the Unity Engine trying to map the pixels of the heightmap image onto the terrain’s coordinate system. For this reason, a smoothing filter was applied to reduce the tris count of the terrain and make it more optimised for the GPU to process. The smoothing filter for the terrain was downloaded from the Unity Asset store as a free package.

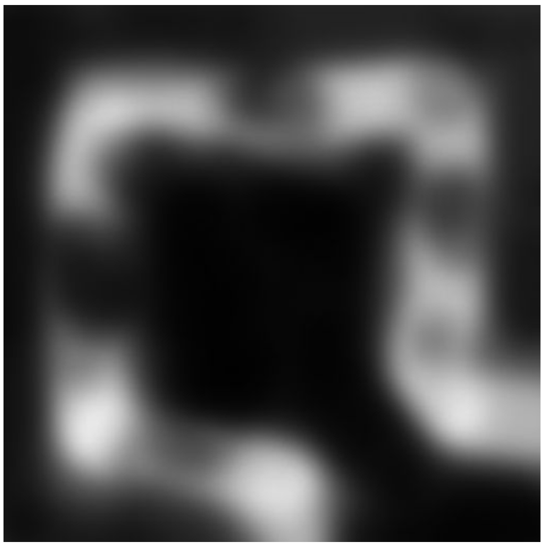


Figure 4.3: Height map of the main terrain for the Forest

The rest of the forest is composed of a number of assets combined together to produce a nice optical result. Some were downloaded for free from the Asset Store, others are provided in the Unity Standard Assets or are simple Unity Primitives(Cubes, Cylinders, etc.) and some were made using the Autodesk 3ds Max Student Edition. These assets include trees, bushes, rocks, bow, arrow, particle effects and more. Some assets include LOD models, as a method to improve runtime performance which was explained in chapter 2.4.3.

To further aid immersion, a Windzone was implemented to make the trees and bushes move as if they were being hit by wind. This is already provided by Unity by going to Create -> 3D Object -> Windzone. However, for the trees or any other kind of foliage to respond to the Windzone effect they must be implemented as part of the terrain with a Bending factor set in its parameters. On top of all of that, ambient forest sound effects were added.

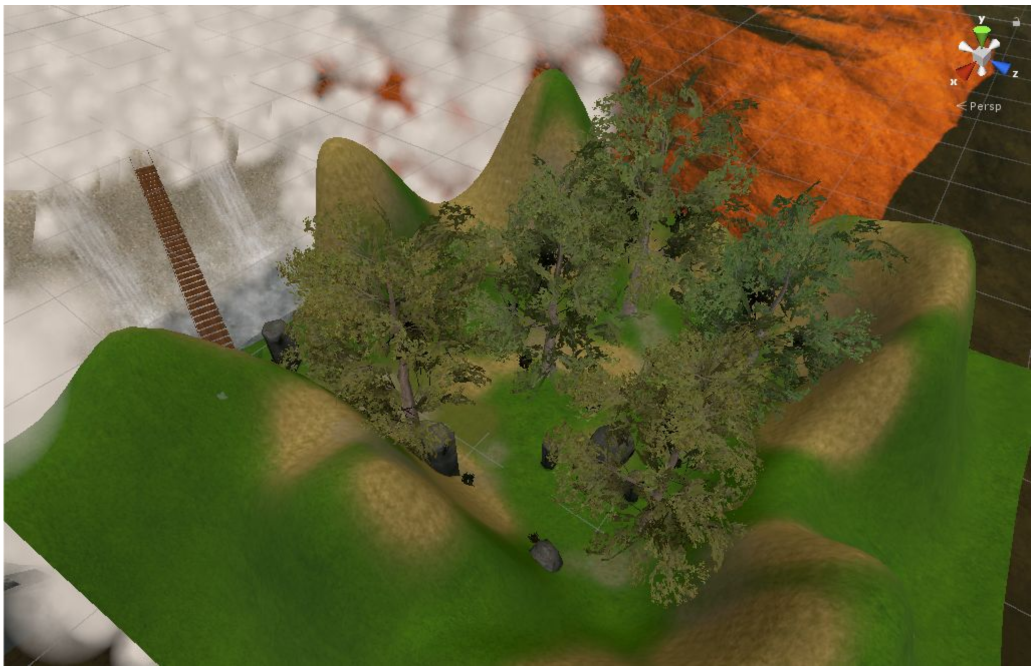


Figure 4.4: The Forest

Just outside the forest, there is a bridge where the players are requested to cross in order to advance further into the game. The bridge is consisted of Unity Primitives with textures and a small animation to make it look like as if pushed around by the wind - since this does not respond to the Windzone. The other side of the bridge is hidden under heavy clouds and two waterfalls. Both these items are particle effects. Particles are small, simple images or meshes that are displayed and moved in great numbers by a particle system. Each particle represents a small portion of a fluid or amorphous entity and the effect of all the particles together creates the impression of the complete entity. Using a smoke cloud as an example, each particle would have a small smoke texture resembling a tiny cloud in its own right. When many of these mini-clouds are arranged together in an area of the scene, the overall effect is of a larger, volume-filling cloud.

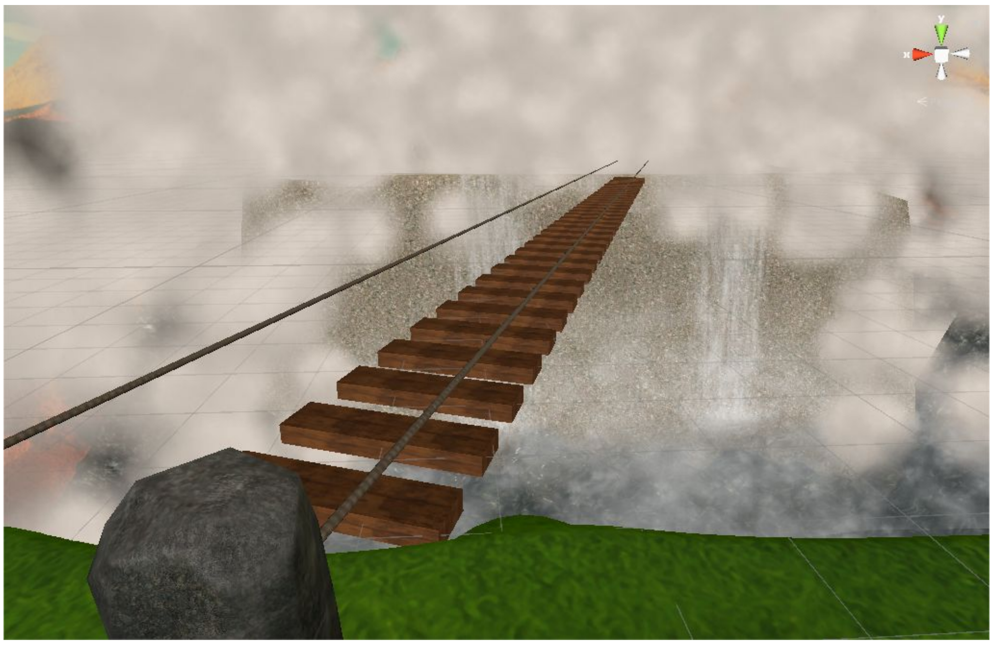


Figure 4.5: The bridge

4.2.2 The Dive

As the players try to cross the bridge, they freeze and the bridge starts to move violently while at the same time losing steps, resulting in the players falling to the water below.

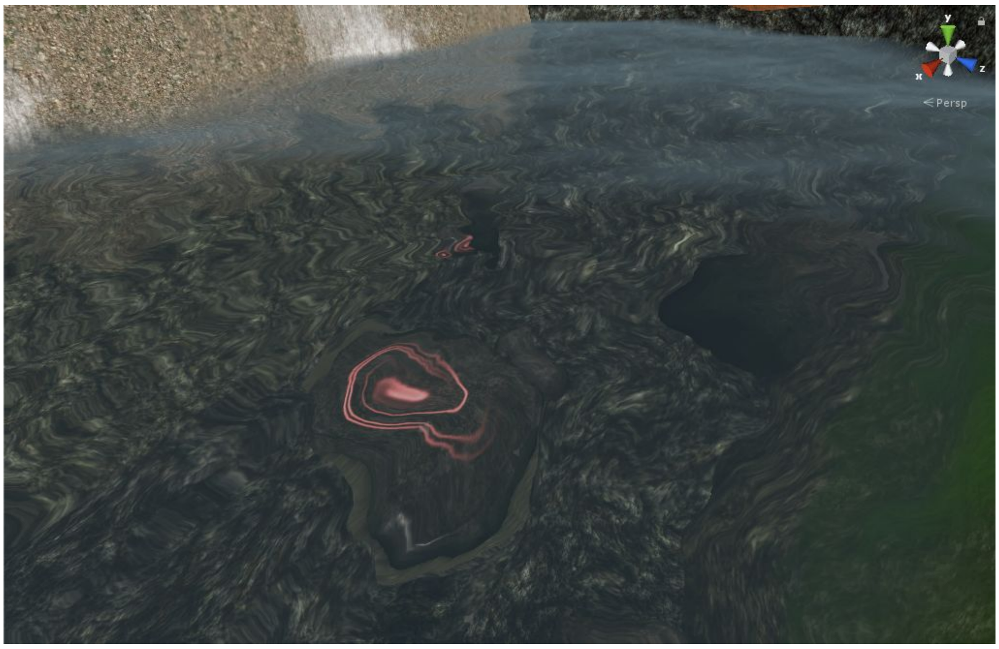


Figure 4.6: The lake

The surface of the water is a complex shader provided by Unity's Standard Assets. The in-water effect is achieved with a camera trick. An Image Effect from the Unity Standard

Assets named ColorCorrectionCurves is used to apply a blue-ish effect in the camera's image. A combination with other effects such as BlurOptimised was considered but resulted in too much eye fatigue and was eventually removed.

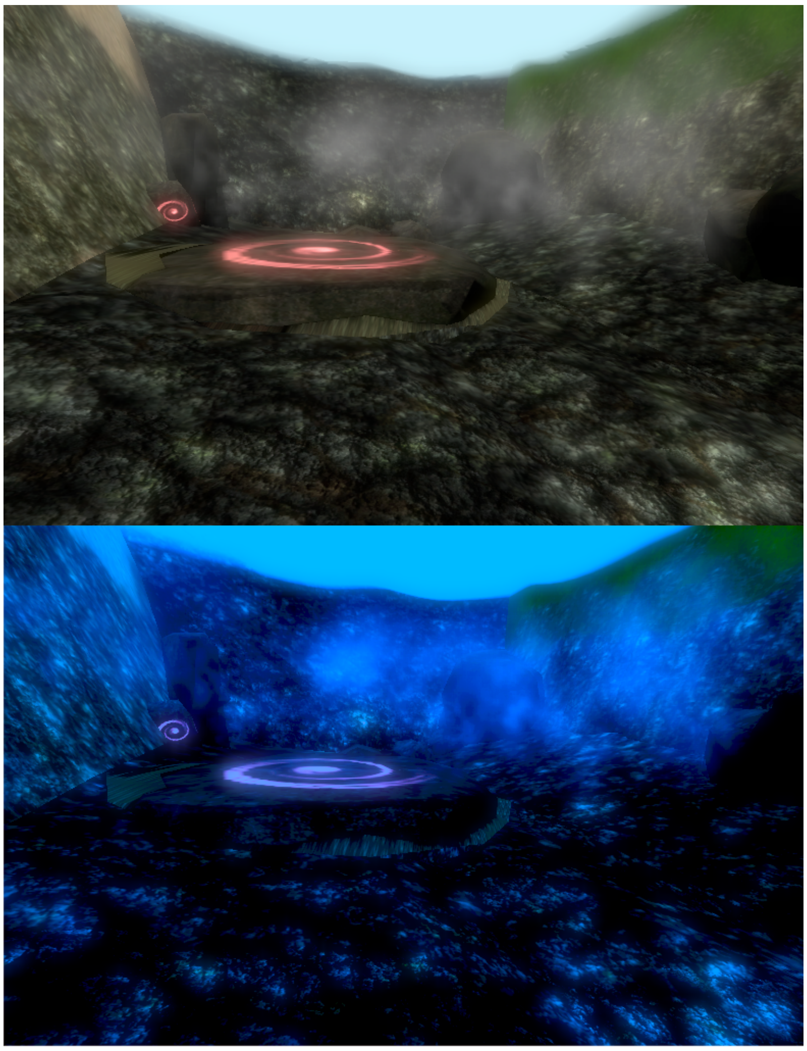


Figure 4.7: Top: View without color correction for underwater
Bottom: View with color correction

After falling into the water, the players find themselves into a small lake. The lake and the pipe, which will be explained below, are one model and were designed in Autodesk 3ds Max, using simple cylinder primitives and imported to Unity afterwards. The result can be seen in the figure below. The model is clearly not ideal due to our limited knowledge of Autodesk 3ds Max but it serves its purpose quite well.

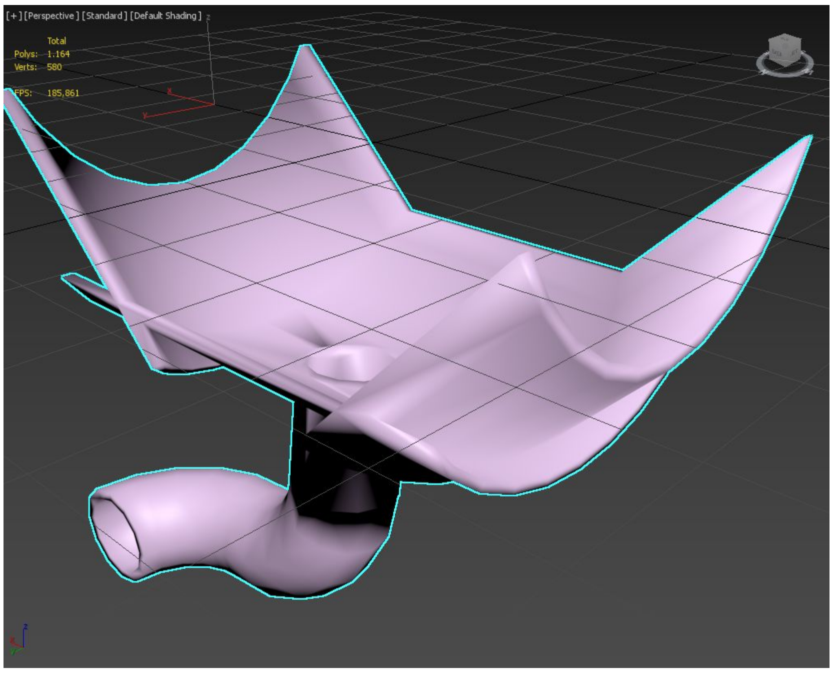


Figure 4.8: Model of the lake and the pipe in Autodesk 3ds Max

Since Unity does not handle colliders for imported complex models very well, we covered this whole model and all other complex custom ones with primitive colliders for the physics events. The lake has several rocks in its bottom and one big rock with a glowing red symbol, which is the entrance to the pipe. Two smaller rocks with the same red symbol are in each side of the lake to indicate the position of two levers that need pulling to activate the big rock. The players have to swim, each to one of the levers and pull them down. When both are pulled, the big rock begins to move and the entrance to the pipe is exposed.

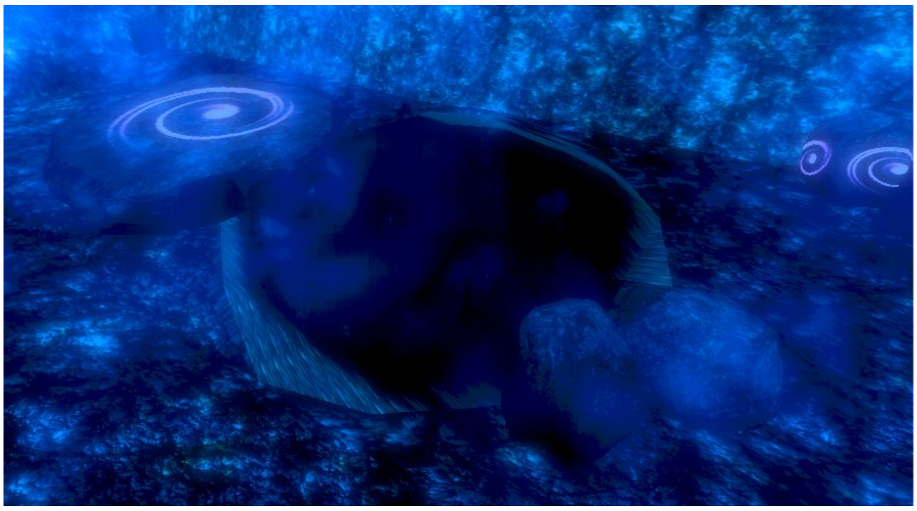


Figure 4.9: Pipe entrance exposed

The pipe is a dark series of rocks. Many of them have a glowing blue symbol to show the path the players must take next. Swimming through the pipe they reach the next section of the game. The Climb.

4.2.3 The Climb

The Climb is two Unity terrains facing each other and a custom model we made for the ceiling from Cylinder primitives in 3ds Max, along with a number of rocks used as grabbing points for the players to grab and climb. They have a glowing green symbol to make them easy to distinguish in this dark humid area. The tricky part, however, is that not all of the rocks are safe to grab. Some detach when you try to grab them and result in the player falling back into the water and starting over.

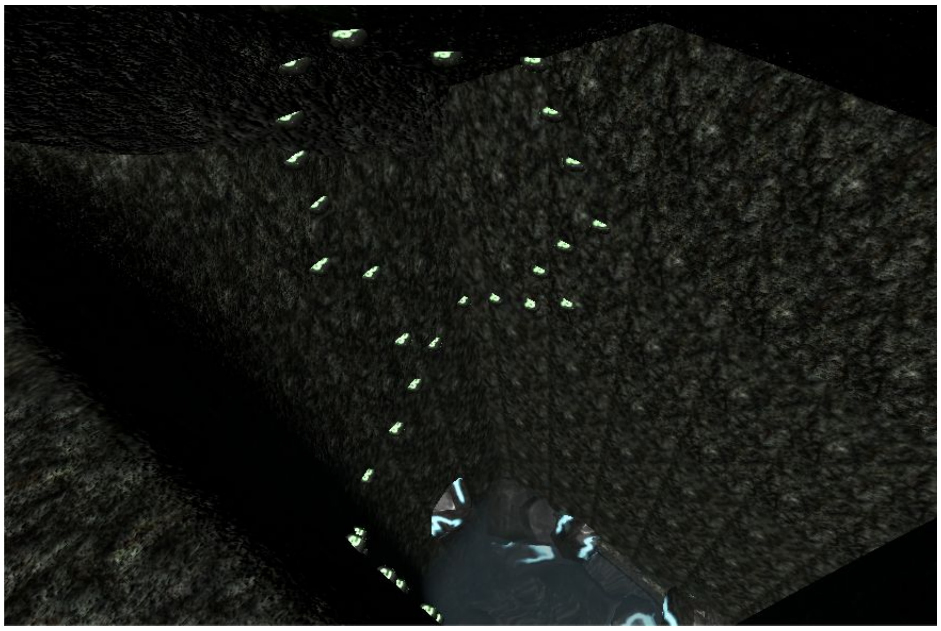


Figure 4.10: The Climb

4.2.4 The Ruins

This is the final act of the game and the most complex in the making. It is a cave with three cooperative tasks for the players to solve. In the ground it has a simple Unity terrain with no heightmap, just something for the players to walk on. Since Unity doesn't support terrain flipping, an external solution had to be imported for the top of the cave like we did for the ceiling of the Climb. A plane was constructed in 3ds Max and a height map was applied on it. Then it was exported, imported to Unity and flipped. The rest of the cave scene was composed of several free ruins assets that were downloaded from the Asset Store and placed in the scene, along with a few torches, some lit and some unlit, which the players could light if they wanted to.

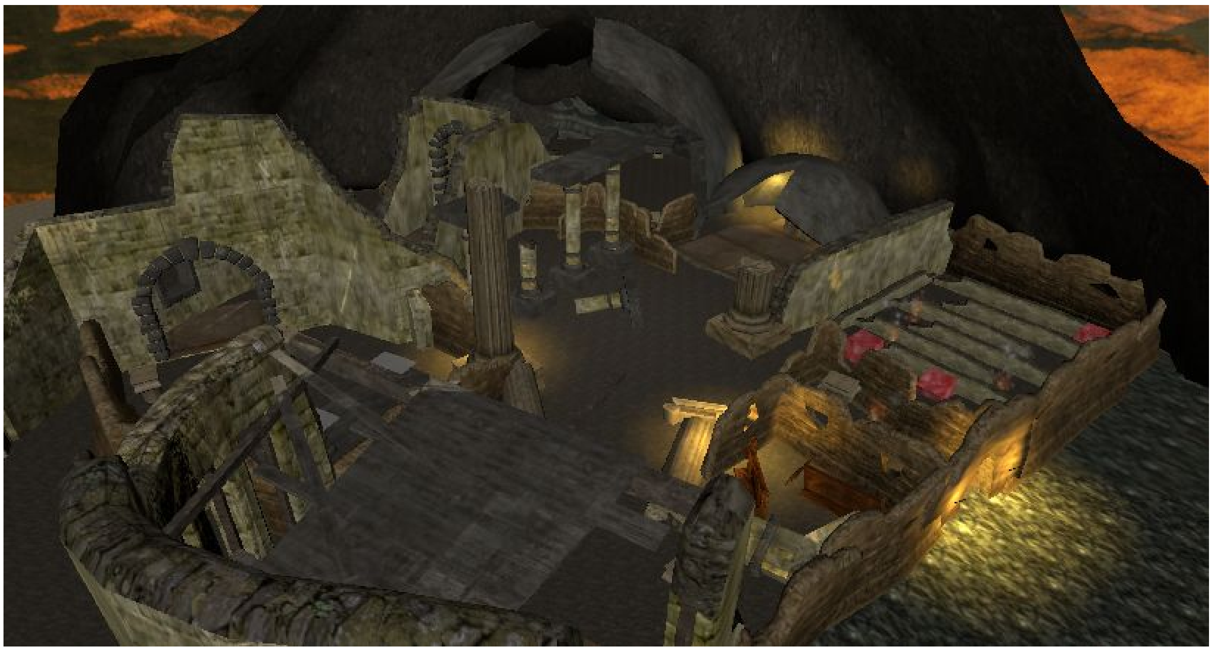


Figure 4.11: The Ruins

The three tasks that the players had to complete were inspired by the four elements, but one element(Water) was shown completed, so that the players can easily understand what is asked of them. Upon completion of each task, a small elemental ball is presented, to place it in the main altar. After placing all three balls in the altar, the game ends. But let's see the three tasks in more detail below.

Fire

In this task, the players have to guide a ball from a starting point to the end without getting the ball hit by the flames. One player carries the ball and the other signals him which flames are going to emerge. From certain points of the table flames emerge and if the ball is in the area and gets hit, they have to start over. When the ball reaches the end point safely, the fire elemental ball is awarded.

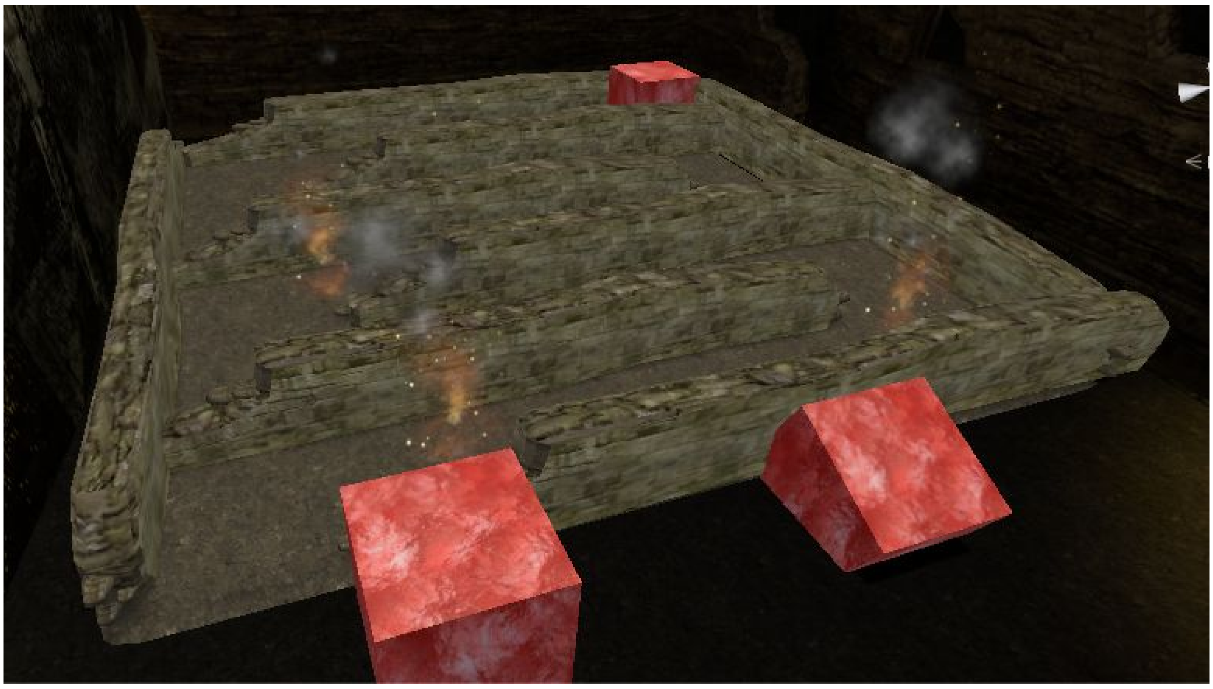


Figure 4.12: Fire task

Air

In the Air task, the players have to shoot randomly generated flying pumpkins with their bow and arrows. If they manage to hit a certain number of pumpkins before a timer runs out, the air elemental ball is awarded. They need to cooperate though, since the number of pumpkins they have to hit cannot be reached by one person in the given timeframe.

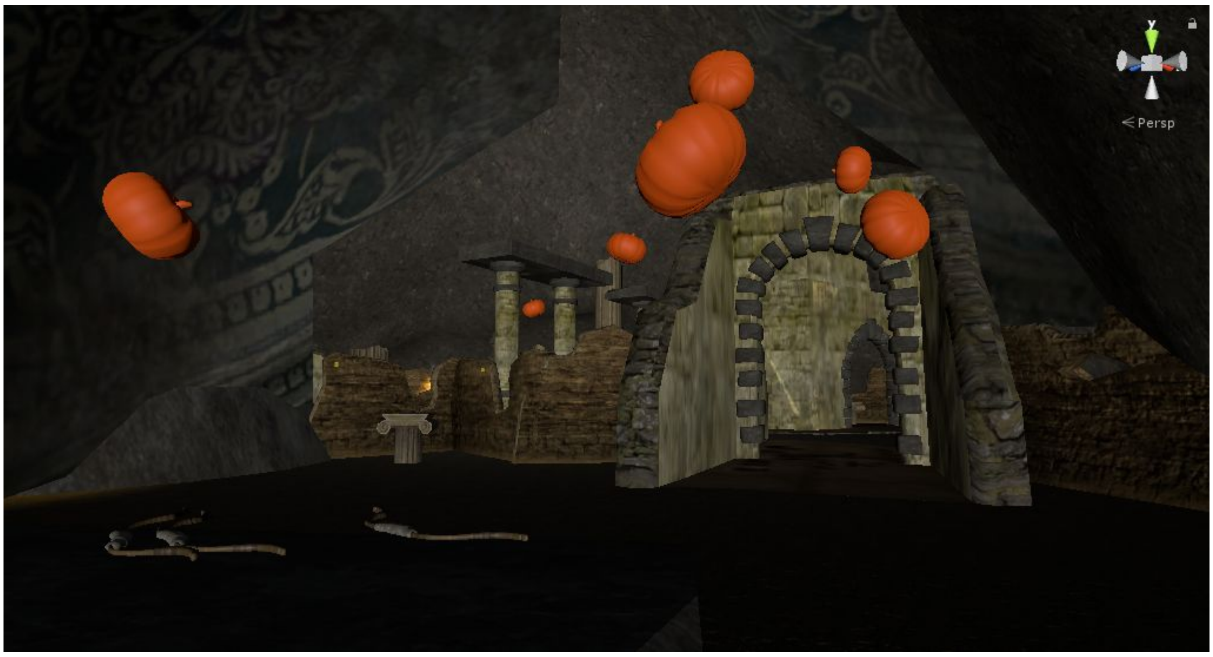


Figure 4.13: Air task

Earth

In this task, one player has to stack two columns of rocks on a table while the other defends them from angry little bats flying around to take them down. When both stacks reach a certain height, the task is complete and the earth elemental ball is awarded.

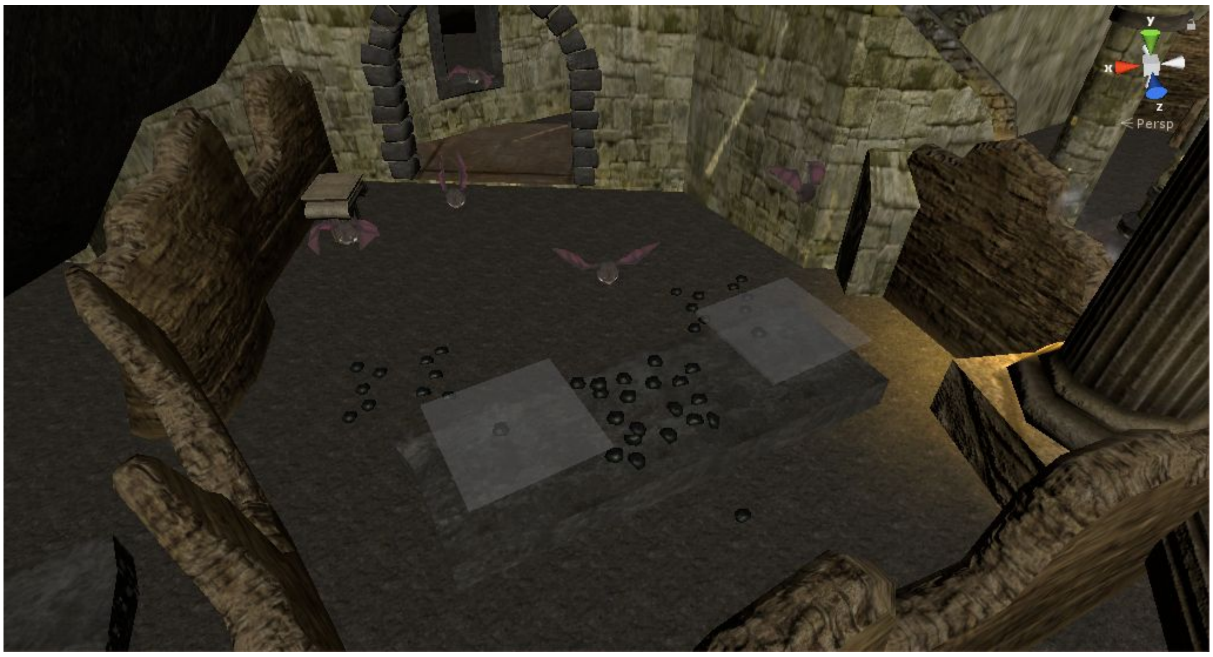


Figure 4.14: Earth task

4.3 Handheld Controllers

In order for virtual reality to match a certain standard in immersiveness, it must have some kind of interaction mechanism with the virtual space. There are some methods for complete hand tracking capabilities such as Leap Motion, but in general, virtual reality games and applications do not require that much tracking detail. Soon after the launch of the HMDs, the companies that developed them released controllers to be used in our virtual worlds. They offer lightweight, low latency and great tracking solutions for our interactions. There are, of course, many types of different controllers in the market but in this thesis we make use of two of them and those are the ones we are going to examine in more detail. The Oculus Touch and the HTC Vive controllers.

4.3.1 Oculus Touch Controllers

A motion controller system known as Oculus Touch is available for Oculus Rift. It consists of a pair of handheld units for the left and right hand, each containing a joystick, 3 buttons, and two triggers - one for grabbing and one for shooting or firing. The controllers are fully tracked in 3D space by the Constellation system [6], so they may be represented in the virtual environment. Oculus Touch also features a system for detecting finger gestures made when holding the controllers.

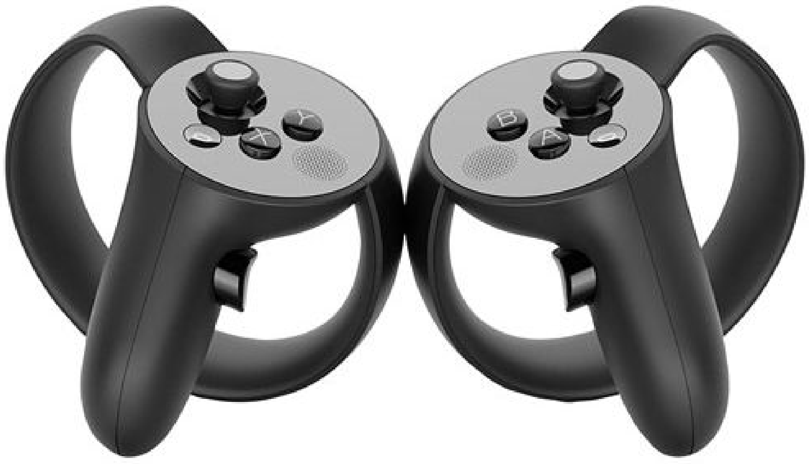


Figure 4.15: Oculus Touch controllers

Figure 4.16: Usage of Oculus Touch controllers

4.3.2 HTC Vive Controllers

HTC's solution offers a pair of controllers that are not matched specifically to one hand. Each controller can easily be used either by the left or the right hand and features two buttons, a touch-joystick, two grip buttons on the sides and one trigger. Much like the Oculus Touch, they are fully tracked in 3D space but by the Lighthouse system [7].

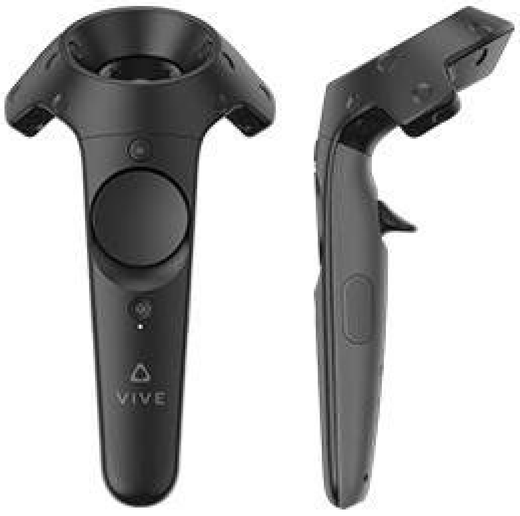


Figure 4.17: HTC Vive controllers

Figure 4.18: Usage of HTC Vive controllers

4.4 In-game Hand Models

Every VR game that uses hand controllers for input, has either the controllers' models shown to the player or some other kind of model according to the game's genre. For example, a FPS shooting game would most likely have some kind of gun instead of the controller model. In this game however, it was most appropriate to use simple hand models, since most actions don't require special equipment. The same model for left and right hand was used, but flipped. Both hands have two states. The default relaxed state and the grabbing one. The transition between the default and the grabbing state is activated by pressing the grip button in the controller.

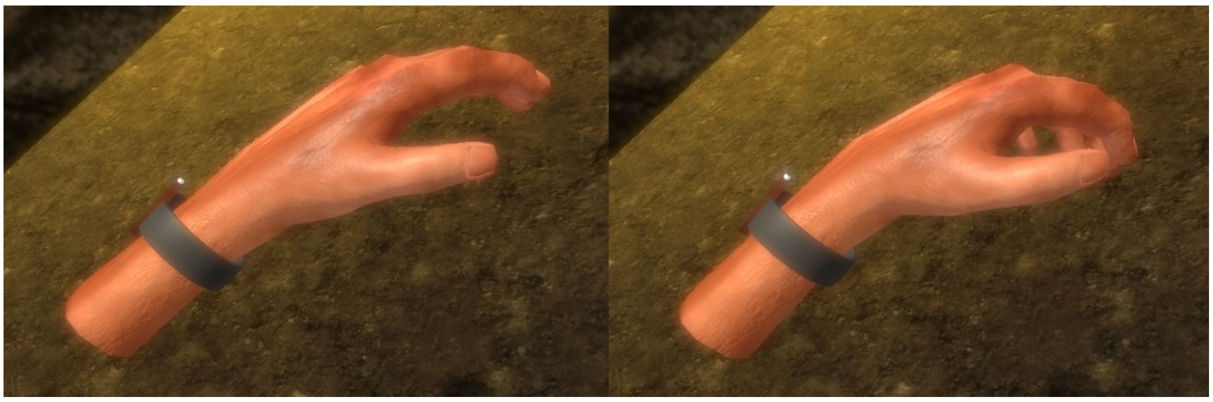


Figure 4.19: Left: Relaxed state - Right: Grabbing state

In the left hand, a wrist watch was added to show the current real-world time. If a player pressed a button in his controller(for Oculus users the X button - for Vive users there is only one button), then from a certain point in the watch a laser pointer was emitted. This laser pointer was used to display help text, in a screen located in the right hand model. In the right hand, there was the screen that was mentioned above and by pressing a button(for Oculus users the B button - for Vive users there is only one button here as well), a flashlight was activated. This is extremely useful in dark areas such as the pipe in the lake that was mentioned in the previous chapter.

The laser pointer was not networked, since help text is only displayed on the person requesting it, but the flashlight was networked. More details about this will be given in chapter 5.2.

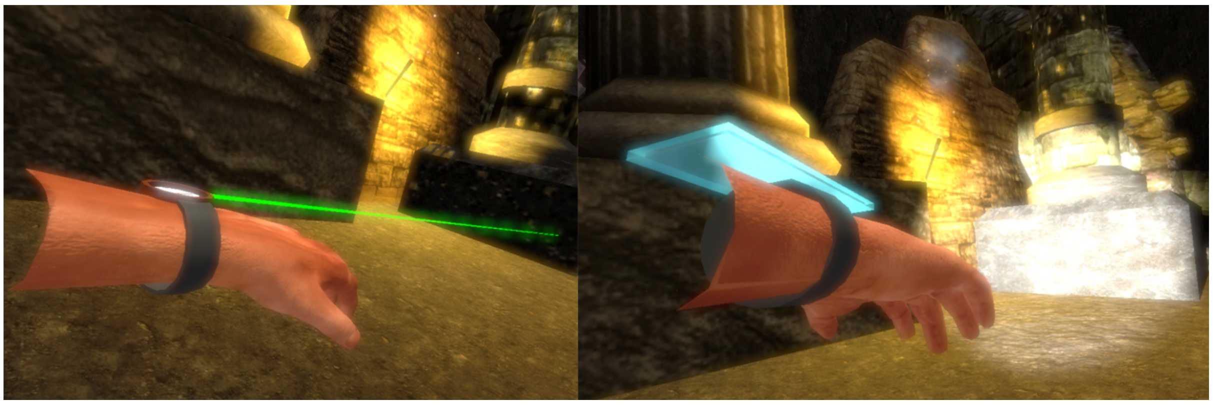


Figure 4.20: Left: The laser pointer - Right: The flashlight

Generally, the hands are visible all the time throughout the game. They deactivate only when the player grabs a bow or an arrow, since that is a complex action for the hand and makes it easier to have them disappear. They reappear the moment the player detaches the bow and/or arrow.

5 Implementation

5.1 Interactivity

5.1.1 Mapping to Different Controllers

In the development of this game, both Oculus and HTC Vive were used. As already demonstrated, each comes with a set of very different controllers yet quite similar. Some actions can be mapped effectively to one another, but some require a more delicate touch.

One of the main differences came with locomotion and it was the touchpad/analog stick that was used for this action. Vive uses the touchpad and Oculus uses the analog stick. Both controllers have two states in their systems, simple touch-and-move and press-and-move. If we used the same exact script for both inputs then it was uncomfortable in one of the two controllers. If touch-and-move was used, then in the Vive the player could move unintentionally simply because he had his finger on the touchpad, while if press-and-move was used, in the Oculus each time someone wanted to move he had to press the analog stick inside and move it. That was bypassed by identifying the HMD and changing the input parameter accordingly, as we will see further below.

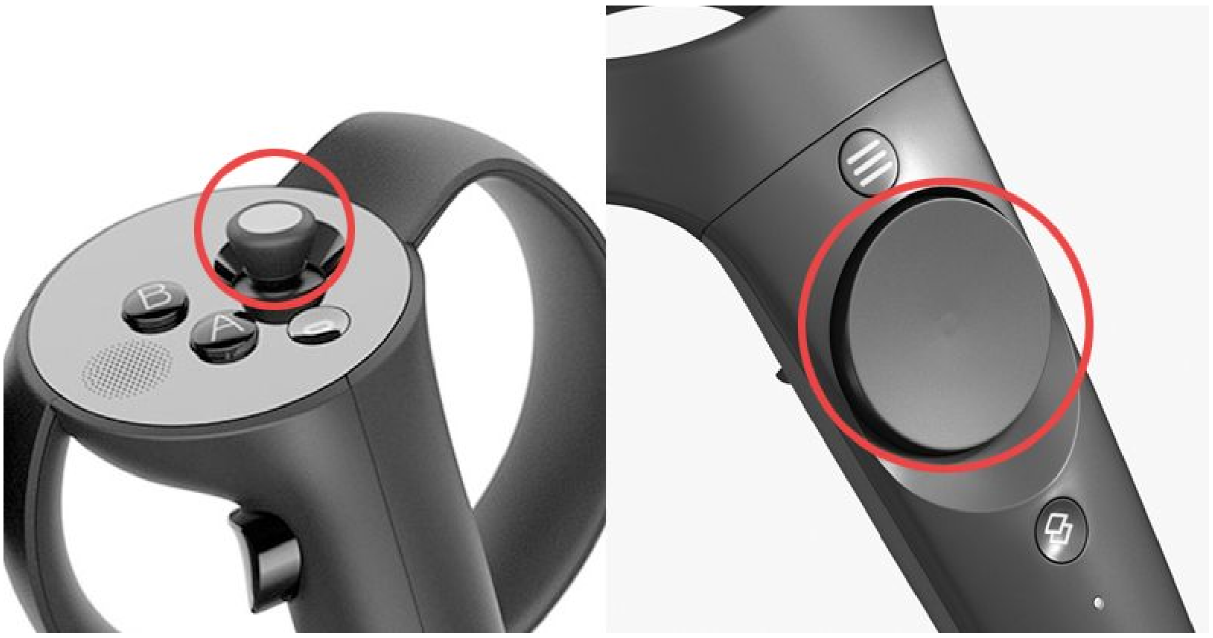
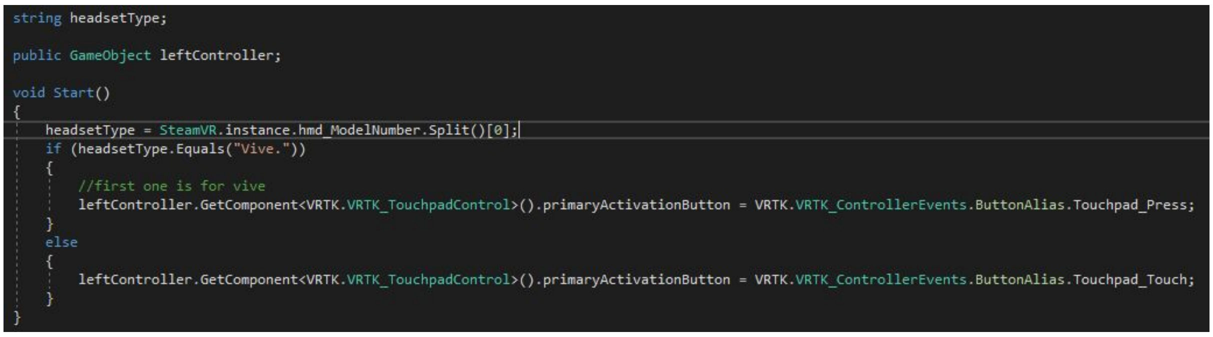


Figure 5.1: Oculus analog stick (left) and Vive touchpad (right)

One other main difference, is that Oculus has two buttons in each controller while Vive only has one, as can be seen in figure 5.1 as well - notice that the buttons A and B are taken into consideration for Oculus and the button with the three lines for Vive. The other buttons you see are system buttons, used, for example, for exiting applications or returning to their menus. That problem could not be bypassed through scripting since it is a hardware limit, so one button action was used for each controller.

The main problem though was how could we know what kind of HMD the user is using, so that we can make the mapping of the controllers. Things were made quite easy here by using SteamVR Plugin, OpenVR and the Virtual Reality Toolkit(VRTK). Implementation of these will be shown in more detail in chapter 5.3 but let us explain briefly what was used for this part.

SteamVR Plugin allowed us to run the game through Steam which gave us access to Steam's own SDK for VR development called OpenVR. Unlike Oculus SDK which only supports the Oculus, OpenVR supports both Oculus and Vive, which is exactly what we need to develop the game for both platforms. VRTK then uses this SDK as a base to implement a set of actions for the controllers of both systems using a single API. That means OpenVR recognizes what HMD the player is using and acts accordingly. This is the point we exploited to extract the information. After studying the code of SteamVR Plugin we found the variable that kept the `modelNumber` of the HMD and of course its type. This was then used to change the parameter needed for mapping the locomotion action, as shown below.



```
string headsetType;

public GameObject leftController;

void Start()
{
    headsetType = SteamVR.instance.hmd_ModelNumber.Split()[0];
    if (headsetType.Equals("Vive."))
    {
        //first one is for vive
        leftController.GetComponent<VRTK.VRTK_TouchpadControl>().primaryActivationButton = VRTK.VRTK_ControllerEvents.ButtonAlias.Touchpad_Press;
    }
    else
    {
        leftController.GetComponent<VRTK.VRTK_TouchpadControl>().primaryActivationButton = VRTK.VRTK_ControllerEvents.ButtonAlias.Touchpad_Touch;
    }
}
```

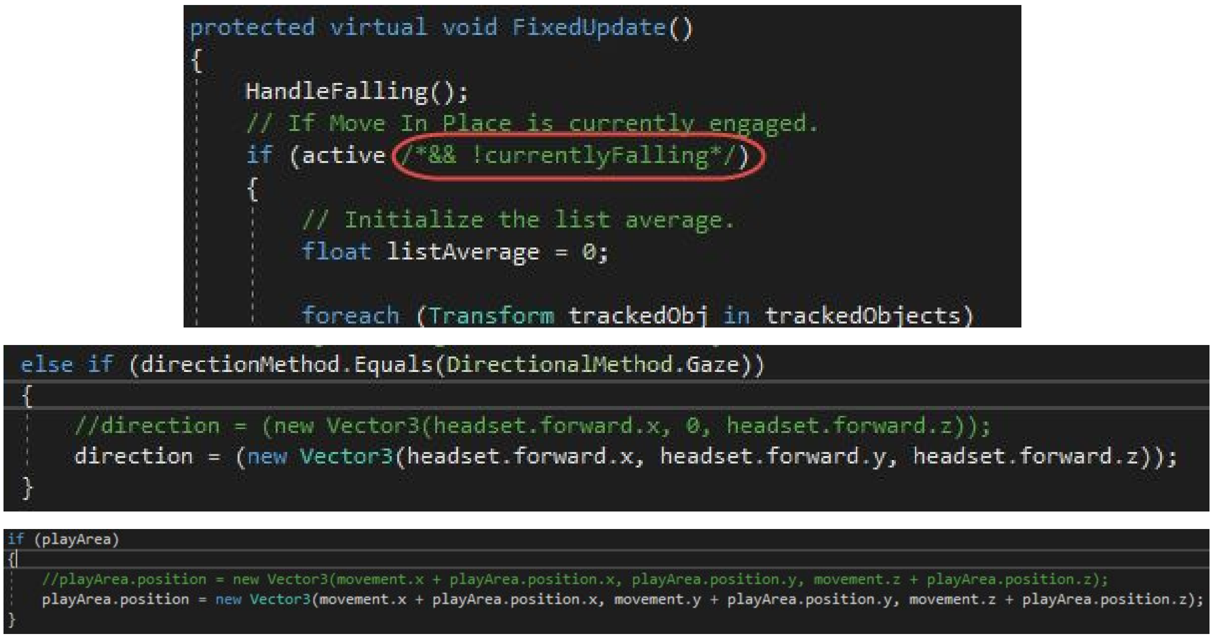
Figure 5.2: Mapping locomotion to different controllers

5.1.2 Players' Interactions

With the use of VRTK, performing interactions with the controllers is very easy. Actions like moving around in the 3D space, grabbing objects, producing laser pointers and more, are only some scripts away. However, custom actions like swimming or displaying help texts on pointer required a more in-depth look and this is where our scripting takes over.

By VRTK standards, all its interaction related scripts are added under a GameObject with the VRTK SDK Manager attached. This lets us choose our SDK, which in our project is OpenVR(also referred to as SteamVR) and link the Camera GameObject along with the controllers' GameObjects. Below that, the Aliases are used to temporarily load our scripts on an alias GameObject and VRTK assigns them in runtime to the actual controllers respectively.

A PlayArea GameObject is used to load general player related scripts such as body physics, teleportation or headset collision events, should we need them in our project. This GameObject was used to attach the swimming and the climbing scripts, as these actions are not controller dependent but general ones. To achieve the swimming effect, a couple of things were required to happen at the same time. First of all, the gravity in our body physics had to be disabled, otherwise we were going to be stuck at ground level. Secondly, the base of the logic was a script that enabled locomotion in the ground kinda like `TouchPadControl` but without the need of pressing any buttons but by moving the controllers called `MoveInPlace`. The faster you were moving them the faster you were travelling, up to a certain custom limit. This script was studied thoroughly and then edited in three very particular parts. The first part was removing a boolean variable `currentlyFalling` from the main if clause. Since we have gravity disabled and want to simulate swimming we would not be at ground level and by game logic we would constantly be in continuous fall. This would cause our script to skip the main if clause and as such `currentlyFalling` is removed from the equation. The second and third part are very similar and are about displacement of the player and his play area in the 3D space and not just in x(right-left) and z(forward-backward) - y being up and down. Displacement happens by moving our hands/controllers and going towards the point our head/headset points to.



```
protected virtual void FixedUpdate()
{
    HandleFalling();
    // If Move In Place is currently engaged.
    if (active /*&& !currentlyFalling*/)
    {
        // Initialize the list average.
        float listAverage = 0;

        foreach (Transform trackedObj in trackedObjects)
```

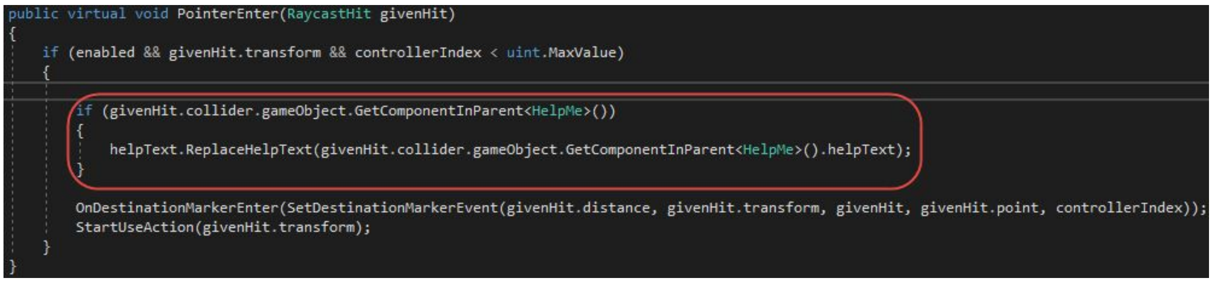
```
else if (directionMethod.Equals(DirectionalMethod.Gaze))
{
    //direction = (new Vector3(headset.forward.x, 0, headset.forward.z));
    direction = (new Vector3(headset.forward.x, headset.forward.y, headset.forward.z));
}
```

```
if (playArea)
{
    //playArea.position = new Vector3(movement.x + playArea.position.x, playArea.position.y, movement.z + playArea.position.z);
    playArea.position = new Vector3(movement.x + playArea.position.x, movement.y + playArea.position.y, movement.z + playArea.position.z);
}
```

Figure 5.3: Changes in the Move In Place script for swimming

There are some scripts that VRTK requires in order for the controllers to start working and become interactable. Those are `ControllerEvents` and `ControllerActions` for general controller initialization and then InteractTouch and InteractGrab to make the controllers respond to touching and grabbing. After that, each controller was set up with its own kind of actions, specifically with what we wanted them to do.

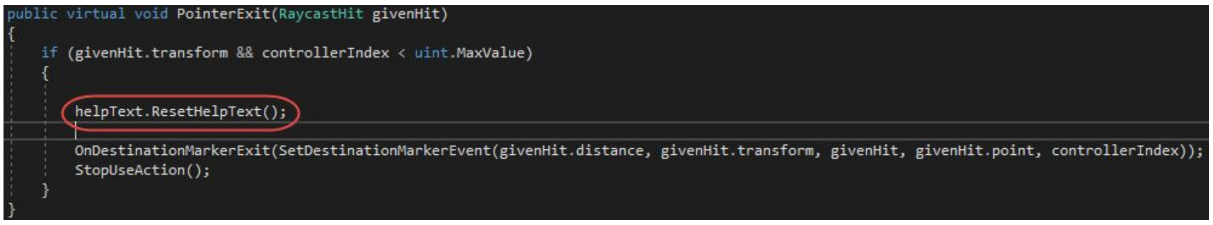
For the left controller, we added a TouchPadControl, a Pointer logic along with its line renderer and the HelpText script which held the logic for displaying and replacing the help text in the right hand. For the TouchPadControl further scripting components were required. We had the choice to make sliding movements or teleportation We chose the former as it is more intuitive and realistic than teleporting from point to point. So a SlideObjectControlAction was added for sliding back and forth along the Y axis and a RotateObjectControlAction for rotating around ourselves. The values used for speed, acceleration, etc. were based on experiments conducted while wearing the headset. We had to modify the pointer logic so that it displayed help messages, based on the object it hits, given that the hit object had a message to be displayed, as shown below.



```
public virtual void PointerEnter(RaycastHit givenHit)
{
    if (enabled && givenHit.transform && controllerIndex < uint.MaxValue)
    {
        if (givenHit.collider.gameObject.GetComponentInParent<HelpMe>())
        {
            helpText.ReplaceHelpText(givenHit.collider.gameObject.GetComponentInParent<HelpMe>().helpText);
        }

        OnDestinationMarkerEnter(SetDestinationMarkerEvent(givenHit.distance, givenHit.transform, givenHit, givenHit.point, controllerIndex));
        StartUseAction(givenHit.transform);
    }
}
```

Figure 5.4: Replacing Help Text

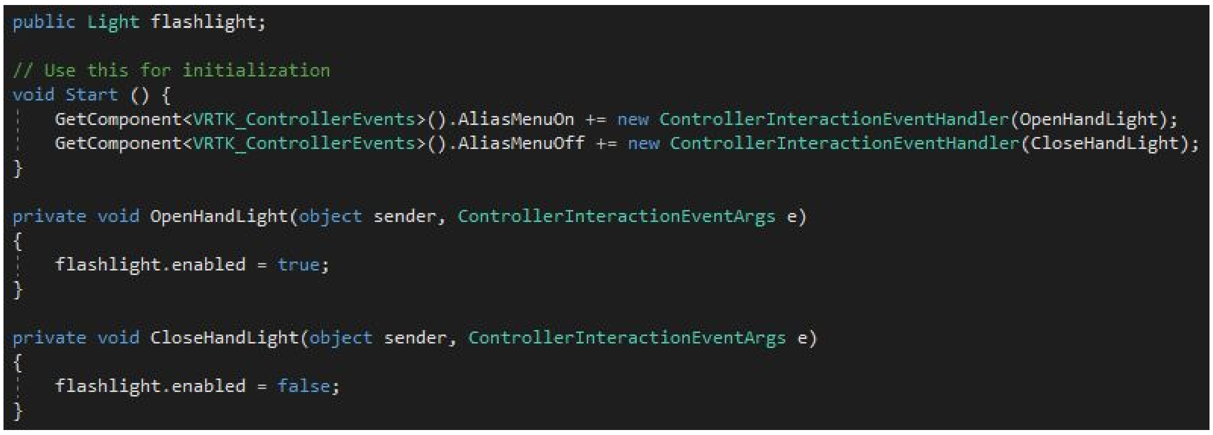
If the pointer left the object in question, we had to reset the message so that it continues to display whatever it was displaying before.



```
public virtual void PointerExit(RaycastHit givenHit)
{
    if (givenHit.transform && controllerIndex < uint.MaxValue)
    {
        helpText.ResetHelpText();

        OnDestinationMarkerExit(SetDestinationMarkerEvent(givenHit.distance, givenHit.transform, givenHit, givenHit.point, controllerIndex));
        StopUseAction();
    }
}
```

Figure 5.5: Resetting Help Text

The right controller was used to open the flashlight, so a Spot Light component was added to it and a OpenFlashlight script to control that light. We took advantage of the emitted ControllerEvents for pressing a certain button and registered two new events to happen when it was pressed or released. The logic behind it is shown below.



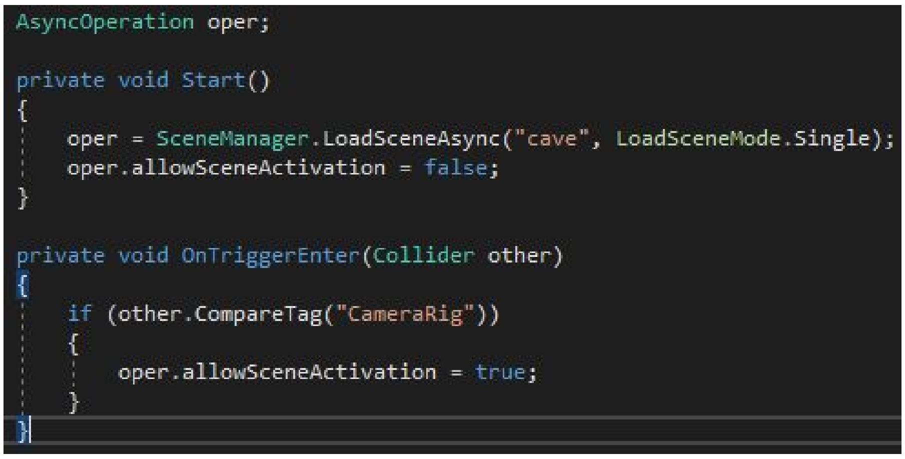
```
public Light flashlight;

// Use this for initialization
void Start () {
    GetComponent<VRTK_ControllerEvents>().AliasMenuOn += new ControllerInteractionEventHandler(OpenHandLight);
    GetComponent<VRTK_ControllerEvents>().AliasMenuOff += new ControllerInteractionEventHandler(CloseHandLight);
}

private void OpenHandLight(object sender, ControllerInteractionEventArgs e)
{
    flashlight.enabled = true;
}

private void CloseHandLight(object sender, ControllerInteractionEventArgs e)
{
    flashlight.enabled = false;
}
```

Figure 5.6: Opening/closing flashlight

Tutorial Scene

Specifically for the tutorial scene in the PlayArea gameObject we added a VRTK script for tracking the position of the controllers according to our headset's gaze. And in both controllers we added an additional child GameObject with a script called `ControllerTooltips`. This script manages the behavior of the tooltips, their position, what text is gonna be displayed, etc. The general idea of how this works is that our headset casts a raycast and when it hits a controller or both then their tooltips are being activated, otherwise they remain inactive.

The next unique thing in the tutorial scene is our transition from this scene to our actual game. This transition is activated when the player walks into the blue orb that is visible in Figure 4.2. But, since our game is a complex scene that would take several seconds to load, we did not want to have the player wait with a loading screen in their view. What we do, is load 90% of the game scene asynchronously at the time the tutorial scene is loaded. And then, when the player decides he wants to step into the game, he walks into the blue orb, the rest 10% is loaded and the scenes change with minimum loading time.



```
AsyncOperation oper;

private void Start()
{
    oper = SceneManager.LoadSceneAsync("cave", LoadSceneMode.Single);
    oper.allowSceneActivation = false;
}

private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("CameraRig"))
    {
        oper.allowSceneActivation = true;
    }
}
```

Figure 5.7: Loading of the game scene

The second parameter you see in the `LoadSceneAsync` method means that the scenes will replace one another erasing the current one from the memory. The other option we have is `LoadSceneMode.Additive` which keeps the current scene open and loads the new one on top of it. Since we do not have need of the tutorial scene once the actual game starts, we choose to delete it from the memory.

5.1.3 Interactable GameObjects

To make a GameObject interactable, two things are required. Logic in the GameObject doing the interaction such as a controller and logic in the GameObject being interacted with. The former was explained above, so let us examine the logic in the latter. There are several interactable GameObjects in this game, from rocks to bows and arrows to torches and more.

Essential scripts for interactions

In order to enable grabbing of a certain GameObject, it must have an `InteractableObject` script with `IsGrabbable` set to true. This script has many options for customization, like allowing us to set if we want to grab the object as long as we keep the grab button pressed or use it by pressing the trigger button. For example, say we grab a gun, then we press the trigger button to shoot bullets. It also allows us to set a `GrabAttachMechanic` which is how and from where this object will be grabbed and a `SecondaryGrabAction`. The most common secondary action is allowing us to swap grabbing hand. All of the interactable objects in the game have at least the `InteractableObject` script attached.

Lever

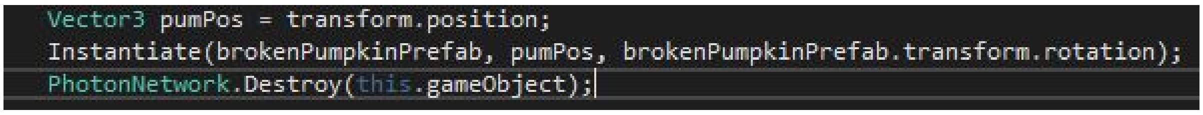
The pulling down mechanism for the levers in the lake is provided by an inherited class of the `VRTKLever` which itself is an inherited class of the base `VRTKControl`. It has parameters for angle, friction, spring strength and more. The way it works is it creates a hinge joint in the lever and when grabbed it allows you to move this joint around the z-axis(or any other axis you set).

Bow and arrow

Perhaps the most interactive object in the game is the bow and arrow. Its main logic is taken from a VRTK example but it was heavily reworked to make it network aware. In the example, when it was grabbed, the main GameObject was destroyed and a clone GameObject was becoming a child of the controller that was doing the grabbing. As we will see in chapter 5.2 this was a problem due to not keeping a consistency in the game and not being able to network its transform. So a new script was written and attached to the bow to expand its interactable logic and also make it interact with the arrows that we want to nock and shoot. The arrow logic also had to be reworked to interact with the new bow logic. Arrows are created by moving a controller behind our head and grabbing as if we are grabbing an arrow from a real quiver. This invisible quiver object follows the transform of the headset which by definition is the in-game camera. The script used in the bow and arrow to deactivate the hands' models when grabbing them is called `InteractControllerAppearance`. Apart from the grab option it also provides options for disappearance when touching and/or using objects but they were not useful in this case and were ignored.

Pumpkin

An other interactable object is the pumpkin. When it is hit with an arrow, its current position and rotation are stored and then the model is replaced with an already made broken one at runtime.



```
Vector3 pumPos = transform.position;
Instantiate(brokenPumpkinPrefab, pumPos, brokenPumpkinPrefab.transform.rotation);
PhotonNetwork.Destroy(this.gameObject);
```

Figure 5.8: Replacing the pumpkin

This broken model was generated in 3ds Max with a Fracture Voronoi script. This script breaks any mesh to as many parts as we want while preserving its volume.

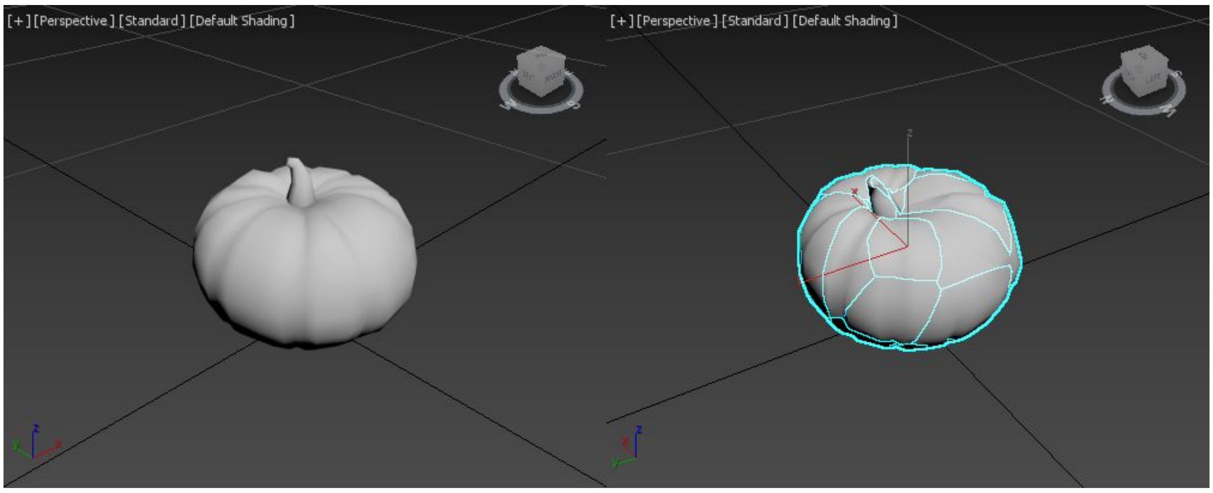
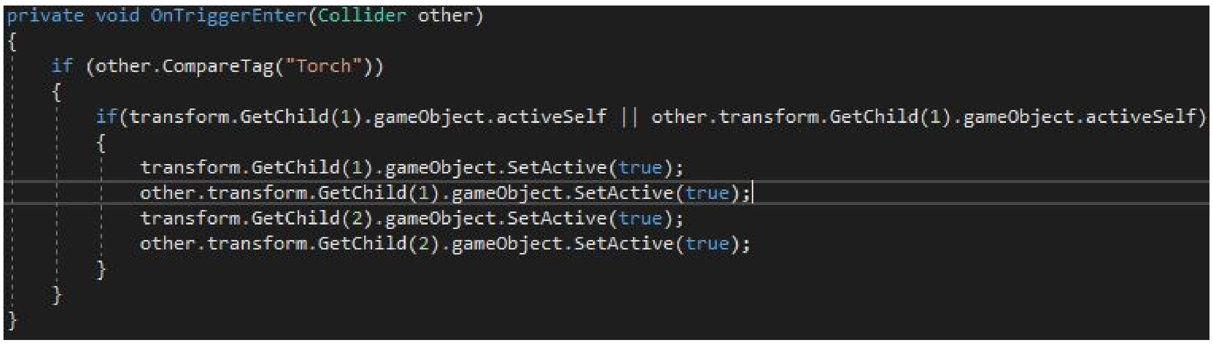

[+] [Perspective] [Standard] [Default Shading]
[+] [Perspective] [Standard] [Default Shading]

Figure 5.9: Left: Whole pumpkin model - Right: Broken pumpkin model

Some torches in the last section can be grabbed and used to light other torches. This is achieved with a collision check between the top of the torches and by enabling the unlit torch's fire particle effect. Below is a demonstration of how this is done in code.

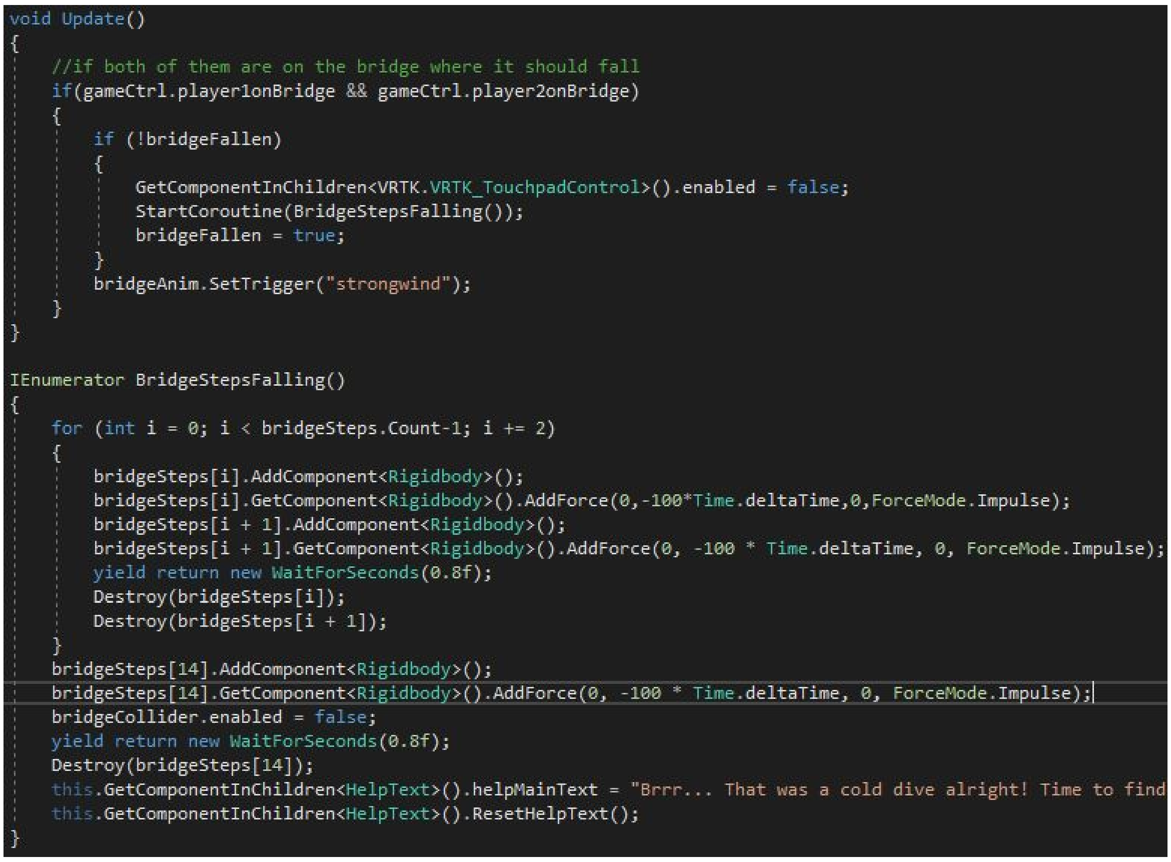


```
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Torch"))
    {
        if(transform.GetChild(1).gameObject.activeSelf || other.transform.GetChild(1).gameObject.activeSelf)
        {
            transform.GetChild(1).gameObject.SetActive(true);
            other.transform.GetChild(1).gameObject.SetActive(true);
            transform.GetChild(2).gameObject.SetActive(true);
            other.transform.GetChild(2).gameObject.SetActive(true);
        }
    }
}
```

Figure 5.10: Lighting the torch method

Bridge

The bridge is not an interactable object in the same sense as a lever or a pumpkin, rather an object that is triggered when the two players are far enough inside to trigger a collider. When the game logic identifies both players in the bridge's trigger collider it disables the players' movement mechanism and the violent movement animation on the bridge starts playing. At the same time, specific steps in the bridge have gravity enabled one after another, so they start falling and are destroyed soon after to free some memory. The collider that kept the players on the bridge without them falling is disabled and they fall in the water below.



```
void Update()
{
    //if both of them are on the bridge where it should fall
    if(gameCtrl.player1onBridge && gameCtrl.player2onBridge)
    {
        if (!bridgeFallen)
        {
            GetComponentInChildren<VRTK.VRTK_TouchpadControl>().enabled = false;
            StartCoroutine(BridgeStepsFalling());
            bridgeFallen = true;
        }
        bridgeAnim.SetTrigger("strongwind");
    }
}

IEnumerator BridgeStepsFalling()
{
    for (int i = 0; i < bridgeSteps.Count-1; i += 2)
    {
        bridgeSteps[i].AddComponent<Rigidbody>();
        bridgeSteps[i].GetComponent<Rigidbody>().AddForce(0,-100*Time.deltaTime,0,ForceMode.Impulse);
        bridgeSteps[i + 1].AddComponent<Rigidbody>();
        bridgeSteps[i + 1].GetComponent<Rigidbody>().AddForce(0, -100 * Time.deltaTime, 0, ForceMode.Impulse);
        yield return new WaitForSeconds(0.8f);
        Destroy(bridgeSteps[i]);
        Destroy(bridgeSteps[i + 1]);
    }
    bridgeSteps[14].AddComponent<Rigidbody>();
    bridgeSteps[14].GetComponent<Rigidbody>().AddForce(0, -100 * Time.deltaTime, 0, ForceMode.Impulse);
    bridgeCollider.enabled = false;
    yield return new WaitForSeconds(0.8f);
    Destroy(bridgeSteps[14]);
    this.GetComponentInChildren<HelpText>().helpMainText = "Brrr... That was a cold dive alright! Time to find
    this.GetComponentInChildren<HelpText>().ResetHelpText();
}
```

Figure 5.11: Triggering the bridge events

Lake Puzzle

For the lake puzzle logic, we make use of the two levers and the big rock that covers the pipe entrance. When both levers lock in their down state - by having the players move them and transmit their transform information via the network - we activate the lerping of the big rock and replace the right hand's text screen with the new help message.

GameController

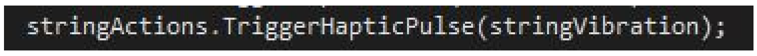
This is a GameObject with scripts which holds all the game information regarding the puzzles and tasks. Though, logic for individual puzzles is stored in separate scripts that are attached and connected to the `GameController`. This GameObject is spawned through the network and is not created at the load of the scene like other GameObjects because it also holds variables that need to be transmitted to the other player, like their position on the bridge and variables for the tasks.

Elemental orbs

The 4 orbs that the players receive from completing the 3 final tasks(Water is already completed), go to their places in the altar. They are made of Unity Primitive Spheres with a simple material applied. In the altar there are 4 places for the orbs. Each orb can be placed in either of the places, and when all 4 are placed, the game ends.

5.1.4 Haptic Feedback

To increase the immersiveness in certain parts of the game, a haptic feedback on the controllers was implemented. For example, when pulling the string of the bow or lighting a torch. Since the controllers only have vibration as feedback, that is what we are using. This is done easily through scripting just by calling a simple method called `TriggerHapticPulse` on a Controller Action as shown below. The parameter passed in the method is the intensity of the haptic pulse.



```
stringActions.TriggerHapticPulse(stringVibration);
```

Figure 5.12: Method call for haptic feedback

5.2 Networking

Implementing networking in a game can be quite tricky without the proper tools. Several problems must be overcome such as handling the network flow, the game states and more. This is where PUN steps in. It provides us with the high level API we need to implement these features effortlessly without the need of studying up on complex mathematics on telecommunications. It also handles the servers and all the background network parameters.

The next step is figuring out what we really need to network in our game. For this, we have to take into consideration the network parameters that were mentioned in 2.6.1 and make network aware only the objects that really are crucial to the multiplayer part. This will keep the network flow to a minimum and will not consume a lot of bandwidth. Since we are developing a small game, this was not really something to trouble us but the networked objects were kept to a minimum nonetheless. For example, if a tree is there just for decoration, it does not really need to be network aware but a rock that can be thrown around has to be.

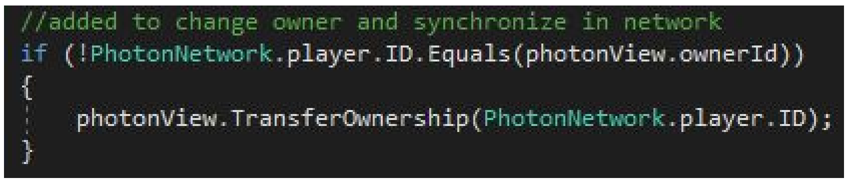
5.2.1 PUN Integration

Like most Unity custom packages, PUN was downloaded from the Asset Store for free. Photon offers a free service with the limit of 20 concurrent users. All you have to do is register in their website and create an application ID. This application ID is then inserted in the AppId field of the settings. After setting the server parameters for our game PUN was ready to be integrated. The Parameters include the Hosting model which is Photon Cloud, the Region which is Eu(Europe) since we want to be matched to a server located as near to our location as possible and the Protocol we want to use in our data transmission which is UDP.

A new GameObject was created named `NetManager` that was going to handle the establishment of the connection. The sample code for that is provided by PUN in a file called `ConnectAndJoinRandom`. Several changes were made to accommodate the initialization of certain scripts and variables for our game. In `OnJoinedRoom` method we added instantiators for our player prefabs and any other prefab that had to be instantiated in the network from the start, such as the elements in the Forest level and all the networked game logic. A different script was created named `SpawnThings` that handled all the instantiations for this part. All future network instantiations for prefabs were handled in the methods that requested them.

The `NetManager` script is what connects our Scene to the PUN's dedicated server for our game. The first player to connect is the Master and all others after him are the Clients. However, this does not mean that if the Master disconnects then the Clients disconnect along with him. If this happens, then the Master attribute simply passes on the next Client in the list and the game continues. This was one of the main features that distinguished PUN over Unity's own Networking solution.

To understand how multiplayer gaming works, first we have to understand some basic rules revolving around networking. Every GameObject spawned in the network must have a network ID. This ID is unique and allows the server to store information about every particular object and transmit them to other players. This ID is applied to a GameObject by using a PUN script called `PhotonView`. This script also allows us to set if the owner of the GameObject is Fixed, Takeover or Request. Fixed means that the owner cannot change and only the player that spawns this object has control over it in the network. Takeover means that the owner can change in an instant by a simple action such as grabbing and Request means that the player that wants ownership of the object has to ask the current owner's permission. All of these actions are, of course, implemented through scripting. For example, ownership takeover happens when the function `TransferOwnership` is called on the photonView of the object and passing as a parameter the player's network ID as demonstrated below.



```
//added to change owner and synchronize in network
if (!PhotonNetwork.player.ID.Equals(photonView.ownerId))
{
    photonView.TransferOwnership(PhotonNetwork.player.ID);
}
```

Figure 5.13: Changing ownership of an object

It is essential to emphasize that for any network action a player takes, he must have permission to do so by the network. That means that if he wants to move, for example, a simple rock from place A to place B, he must first take ownership of the rock, as shown above, and then perform the action. Should he not take ownership first, he will be faced with a strange phenomenon. In his state of the game, the rock will move as long as he is performing the action but since he is not the owner of it, the network does not recognize the displacement and slowly moves the rock back to its original place. And the only one who will see this happen is himself. All other clients along with the owner of the rock will see nothing, simply because the network has not synchronized anything.

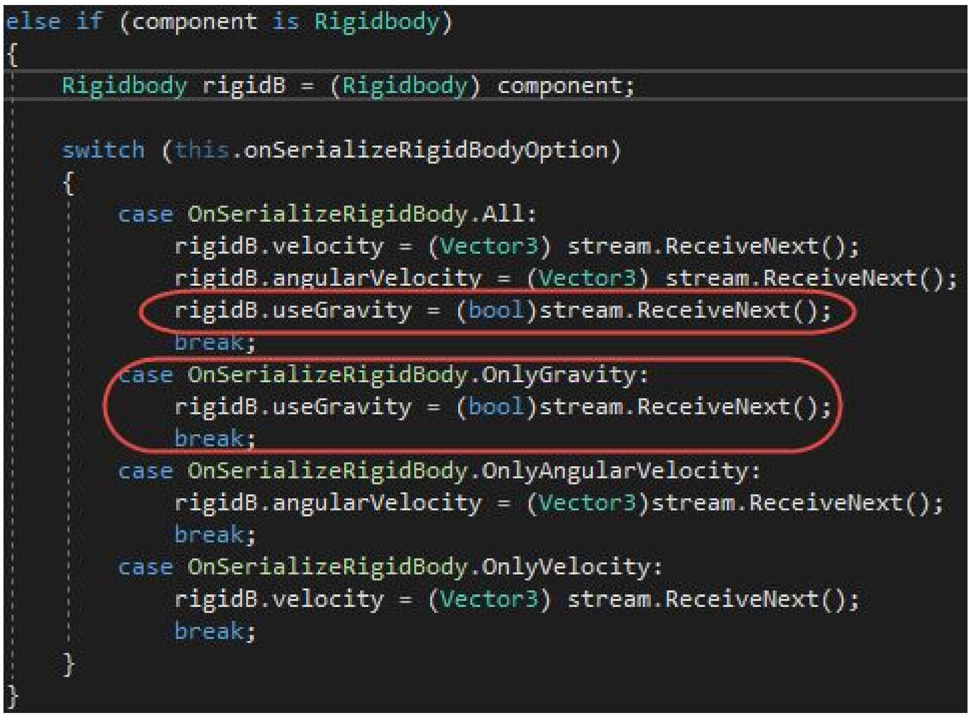
For usual matters, like the synchronization of the Transform component, PUN provides us with already made solutions. The `PhotonTransformView` is attached to a GameObject with a Photon View already attached. If there is not one, a Photon View is added automatically. This happens because, as we explained above, the network must have a unique reference(ID) for each networked object. This Transform View allows the synchronization of position, rotation and scale of the GameObject but we do not necessarily have to synchronize all three of them. If the object's scale stays consistent throughout the game, then we do not need to send information for this part.

A similar script exists for synchronization of the Rigidbody component. However, for some reason PUN developers only chose to implement synchronization of the object's

`Velocity` and `AngularVelocity`. Although these were useful, `Gravity` also had to be synchronized in certain GameObjects in our game that are grabbable and use physics like rocks. Since we do not network the interactions themselves but rather the results of the interactions, the rocks' gravity had to be disabled in the client's clones while the interaction in the master was taking place. Otherwise, all the client sees is a rock constantly falling because of its own gravity even if the position is networked as well. In the master, there is no such problem since his interaction script takes control of the rock physics through script logic.

To implement this feature, the hundreds of lines in the PUN code had to be studied and understood, at least at the level that the implementation could happen. Changes were made to two main scripts and a third small script that was used for displaying the additional information at the Inspector window. These two main scripts are called `PhotonView` and `PhotonRigidbodyView`. The former is the general handler for all synchronization requests emitted from TransformView and RigidbodyView, as well as any custom requests emitted from a MonoBehaviour component that we will see further below.

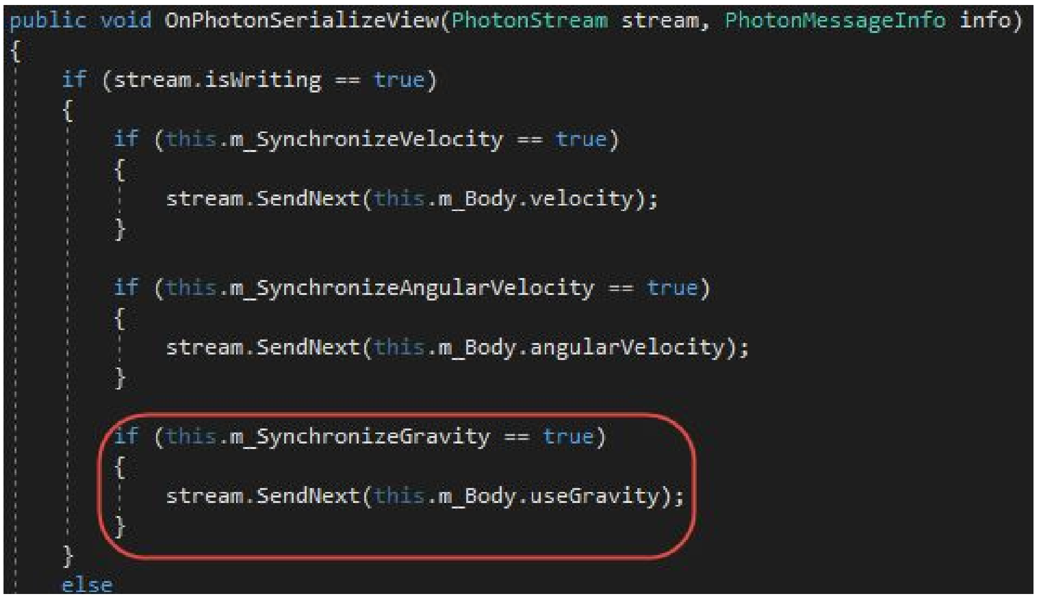
The changes in the Photon View script were a declaration of the use of gravity parameter along with velocity and angular velocity, and then serialization and deserialization of it in the already declared PUN methods.



```
else if (component is Rigidbody)
{
    Rigidbody rigidB = (Rigidbody) component;

    switch (this.onSerializeRigidBodyOption)
    {
        case OnSerializeRigidBody.All:
            rigidB.velocity = (Vector3) stream.ReceiveNext();
            rigidB.angularVelocity = (Vector3) stream.ReceiveNext();
            rigidB.useGravity = (bool)stream.ReceiveNext();
            break;
        case OnSerializeRigidBody.OnlyGravity:
            rigidB.useGravity = (bool)stream.ReceiveNext();
            break;
        case OnSerializeRigidBody.OnlyAngularVelocity:
            rigidB.angularVelocity = (Vector3)stream.ReceiveNext();
            break;
        case OnSerializeRigidBody.OnlyVelocity:
            rigidB.velocity = (Vector3) stream.ReceiveNext();
            break;
    }
}
```

Figure 5.14: Implementation of gravity in the DeserializeComponent method

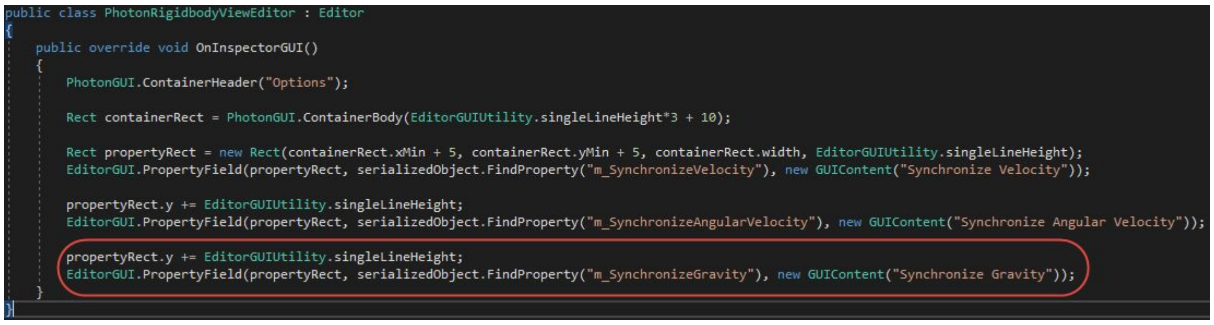
The changes in the Rigidbody View revolved around the passing of the gravity parameter in the object's own gravity. The `OnPhotonSerializeView` method checks whether the network data stream is writing and so, it reads the gravity value to write it to the stream or the other way around.



```
public void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info)
{
    if (stream.isWriting == true)
    {
        if (this.m_SynchronizeVelocity == true)
        {
            stream.SendNext(this.m_Body.velocity);
        }

        if (this.m_SynchronizeAngularVelocity == true)
        {
            stream.SendNext(this.m_Body.angularVelocity);
        }

        if (this.m_SynchronizeGravity == true)
        {
            stream.SendNext(this.m_Body.useGravity);
        }
    }
    else
```

Figure 5.15: Writing the gravity value to the network data stream

As for the small change for the Inspector's window this was done on the `PhotonRigidbodyViewEditor` script which handles the appearance of the script in the Inspector panel. All we did was allocate some space for the extra line.



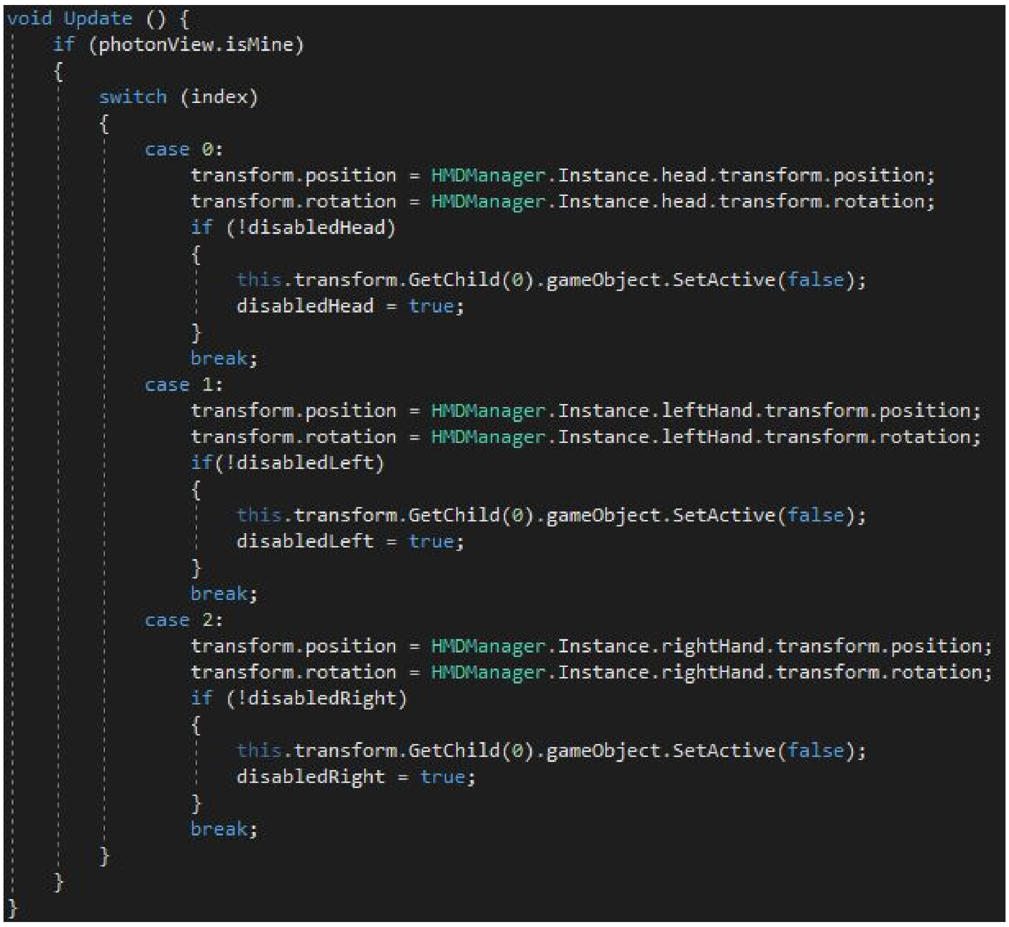
```
public class PhotonRigidbodyViewEditor : Editor
{
    public override void OnInspectorGUI()
    {
        PhotonGUI.ContainerHeader("Options");

        Rect containerRect = PhotonGUI.ContainerBody(EditorGUIUtility.singleLineHeight*3 + 10);

        Rect propertyRect = new Rect(containerRect.xMin + 5, containerRect.yMin + 5, containerRect.width, EditorGUIUtility.singleLineHeight);
        EditorGUI.PropertyField(propertyRect, serializedObject.FindProperty("m_SynchronizeVelocity"), new GUIContent("Synchronize Velocity"));

        propertyRect.y += EditorGUIUtility.singleLineHeight;
        EditorGUI.PropertyField(propertyRect, serializedObject.FindProperty("m_SynchronizeAngularVelocity"), new GUIContent("Synchronize Angular Velocity"));

        propertyRect.y += EditorGUIUtility.singleLineHeight;
        EditorGUI.PropertyField(propertyRect, serializedObject.FindProperty("m_SynchronizeGravity"), new GUIContent("Synchronize Gravity"));
    }
}
```

Figure 5.16: Changing the appearance of the Rigidbody View in the Inspector

5.2.2 Connecting the Players

For the game to be a real multiplayer, the players had to be able to see each other in some form or way. With the current HMD technology we do not really have a way to track specific areas of the body apart from the head and hands, so these three areas were the ones given in-game models to represent the actual user. Additional models for the feet and the body in general were taken into consideration but without actual tracking information they exhibited very unrealistic behaviour and were removed. As was mentioned in chapter 4.3, we have constructed a set of hands for in-game use. Those hands were also made network aware. And a generic head model was used to represent the player's head in-game.

Both of these objects are handled in the same script since they both concern the player. Two different prefabs were made for the hands, where different means that they did not have any interaction logic attached to them because as mentioned earlier, we do not network interaction logic, only the aftermath of the interactions on the objects. They had a `PhotonView` attached and a `PhotonTransformView` to network their position and rotation. Then we attached a custom script to them to copy the actual controllers' and head's transform onto these network aware prefabs, so that the other clients can see what we are doing. We also created some logic to disable them in our local machine since we do not need to see them.



```
void Update () {
    if (photonView.isMine)
    {
        switch (index)
        {
            case 0:
                transform.position = HMDManager.Instance.head.transform.position;
                transform.rotation = HMDManager.Instance.head.transform.rotation;
                if (!disabledHead)
                {
                    this.transform.GetChild(0).gameObject.SetActive(false);
                    disabledHead = true;
                }
                break;
            case 1:
                transform.position = HMDManager.Instance.leftHand.transform.position;
                transform.rotation = HMDManager.Instance.leftHand.transform.rotation;
                if(!disabledLeft)
                {
                    this.transform.GetChild(0).gameObject.SetActive(false);
                    disabledLeft = true;
                }
                break;
            case 2:
                transform.position = HMDManager.Instance.rightHand.transform.position;
                transform.rotation = HMDManager.Instance.rightHand.transform.rotation;
                if (!disabledRight)
                {
                    this.transform.GetChild(0).gameObject.SetActive(false);
                    disabledRight = true;
                }
                break;
        }
    }
}
```

Figure 5.17: Networking the players' prefabs

5.2.3 Synchronization of GameObjects

Synchronization of the GameObjects is a major part of our game. Every client needs to see whether someone grabs and moves a rock or fires an arrow from a bow. This is requires a couple of things to work together. First of all, all objects that are going to be synchronized need to be spawned either by a client directly with `PhotonNetwork.Instantiate` which will give ownership of them to the client that spawned them or from the scene with `PhotonNetwork.InstantiateSceneObject` which will give ownership of them to the scene and by extension to the Master client. For this to work, the prefab must have a `PhotonView` attached to gain a `networkID`.

If we want to network its transform then a `PhotonTransformView` is required and if we want to network the physics applied to it then a `PhotonRigidbodyView` is required. Both of these scripts send their messages across the network with the `OnPhotonSerializeView` method. This method is what we also used to send custom variables regarding game states as we will see in the next chapter. This method acquires the data stream that Photon opens and uses to stream its information through the internet and checks whether it is time to write on it or read from it. If it is time to read from it, then all the information from the streamed variables are applied onto the object's variables and overwrites them. On the other hand, if it is time to write to it, all the information of the object's variables are written on the stream and sent across the network to its clone objects on the clients.

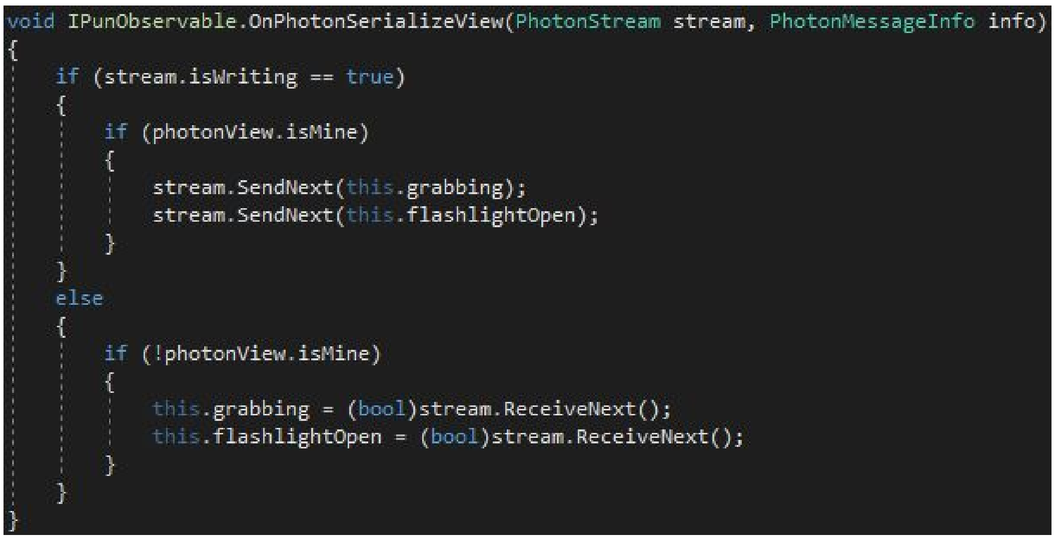
One may wonder how Photon knows which object from all the clones is the one that has the correct information to be streamed. This is where PhotonView's networkID comes in play. It checks which client has the ownership of the object in question and streams that client's information on the object. Every other client is then synchronized based on that information. For another client to take ownership of the GameObject, he has to either just take it or request it from the owner, depending on the object's ownership settings we talked about in the previous chapters.

In this game all GameObjects that could change owners were set at Takeover, so that every GameObject could change owner directly when requested. This was done in a custom script that registered events for grabbing and ungrabbing the object and listened to them throughout the game. So when a grab event on the object was fired, a method we named `Toss_InteractableObjectGrabbed` was called and checked if the player was already owner of the object. If he was not, then the ownership was transferred to him. Several modifications of this script were made and applied to different objects depending on what we wanted to happen upon grabbing them. For example, bows and arrows had a modification that also disabled gravity on grab, specific grabbable rocks on the Climb had a modification that added a Rigidbody component when grabbed to make them fall, etc. These grabbable rocks had also a script attached that disabled them from being grabbed after they were grabbed for once. This was achieved by listening to their grab event and setting a boolean variable to true. This variable was then sent over the network to all the clones of this rock to disable their Grab mechanism as well.

5.2.4 Synchronization of Game States

Apart from synchronizing GameObjects states, Game state at certain points had to be synchronized as well. For example, one player must know when the other player was on the bridge, opening his flashlight or doing some grabbing. This was achieved by serializing custom variables across the network. Declaration and use of these variables is done the same way as any other variable. But since the script containing these variables had to have a specific method for serializing them over the network it had to be an inherited class of `Photon.Monobehaviour` and `IPunObservable` and also declared in the object's `PhotonView`. The first inheritance gives us access to the object's PhotonView without having to get a reference of it every time we want to call something from it. The second inheritance has the required information for the `OnPhotonSerializeView` method to work.

The logic for networking the game states is pretty much the same in all the scripts. We declare some variables and their usage on the script and then declare the `OnPhotonSerializeView` as well, with all the variables we want to stream in it.



```
void IPunObservable.OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info)
{
    if (stream.isWriting == true)
    {
        if (photonView.isMine)
        {
            stream.SendNext(this.grabbing);
            stream.SendNext(this.flashlightOpen);
        }
    }
    else
    {
        if (!photonView.isMine)
        {
            this.grabbing = (bool)stream.ReceiveNext();
            this.flashlightOpen = (bool)stream.ReceiveNext();
        }
    }
}
```

Figure 5.18: Synchronizing grabbing and flashlight states of the right hand

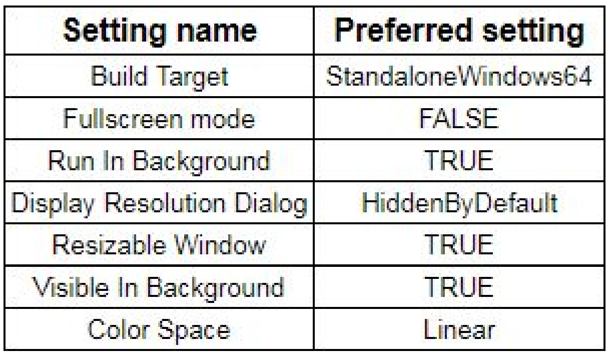
5.3 Virtual Reality

Two free Unity packages were used for full integration of VR with support for both Oculus and Vive simultaneously. The first and main package is SteamVR Plugin which also provides the support for the OpenVR SDK and the second package that requires SteamVR Plugin in order to work, the Virtual Reality Toolkit, also referred to as VRTK. It is worth noting that SteamVR provides some scripting solutions for interacting with the HMDs' controllers but VRTK was found to be an even more complete and better documented solution, with demo scenes to further understand how to integrate it.

5.3.1 SteamVR Plugin Integration

SteamVR Plugin allows the game to be executed through the well-known platform Steam. Obviously, each computer that runs the game has to have Steam installed as well as SteamVR. The OpenVR SDK does not have to be downloaded by itself since it comes already as a package of the SteamVR installation and is implemented in Unity through the plugin.

The reason this was used was the need for the game to run both in Oculus and in Vive with the same executable. If we used the Oculus SDK we would not have support for HTC Vive and its controllers. And since Vive, by default, runs through SteamVR with the OpenVR SDK, this was the only solution. Upon importing, it runs through our Project Settings and provides us with its suggestions for a better VR experience. We can choose to use all, some or simply ignore all of them.



Setting name	Preferred setting
Build Target	StandaloneWindows64
Fullscreen mode	FALSE
Run In Background	TRUE
Display Resolution Dialog	HiddenByDefault
Resizable Window	TRUE
Visible In Background	TRUE
Color Space	Linear

Figure 5.19: SteamVR Plugin’s settings suggestions

The Build Target states the target OS we are going to build our game for when it is ready. In this case the preferred setting is Windows 64bit. We do not want the game to run in Fullscreen mode in the host PC since that would consume additional resources that better be available for the VR system.The Run In Background and Visible In Background settings enable the game to operate even if it is not in the desktop user’s focus all the time. The Display Resolution Dialog is a window that is by default enabled in every game built with Unity. It allows the user to set different resolutions before starting his game in order for him to be able to set what is best for his system’s hardware. However, in VR if the game does not run in Full HD settings it is not going to appear correct, so that dialog window is disabled.

Perhaps the most interesting setting here is the last one, the Color Space. Unity by default has Gamma Color Space for its scenes. However, for realistic lighting, Linear Color Space is preferred. A significant advantage of using Linear space is that the colors supplied to shaders within your scene will brighten linearly as light intensities increase. With the alternative, Gamma Color Space, brightness will quickly begin to turn to white as values go up, which is detrimental to image quality. Another main benefit of Linear is that shaders can also sample textures without Gamma (midtone) compensation. This helps to ensure that color values remain consistent throughout their journey through the render pipeline. The result is increased accuracy in color calculations with improved overall realism in the eventual screen output.

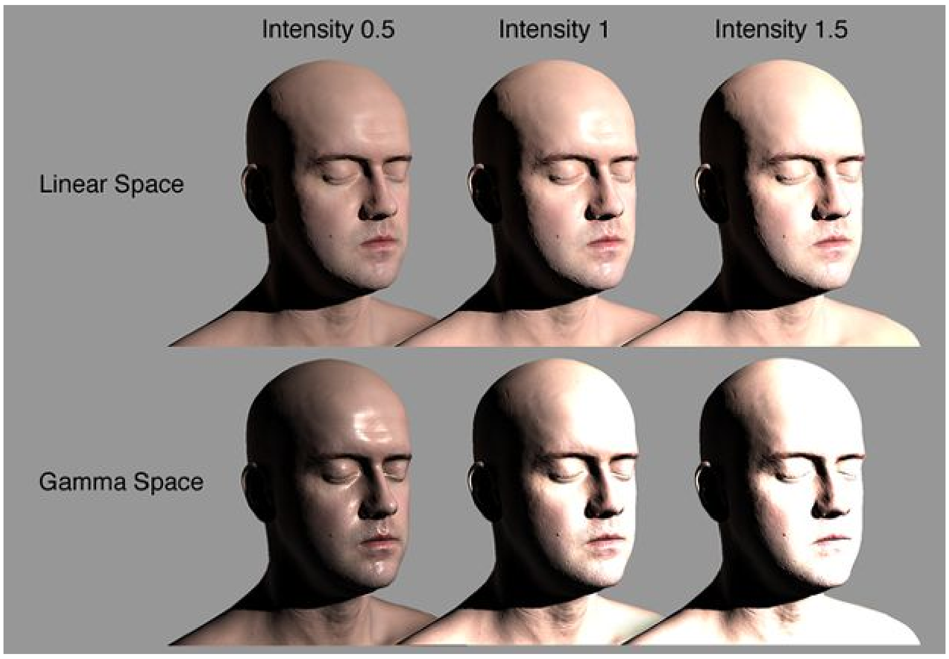
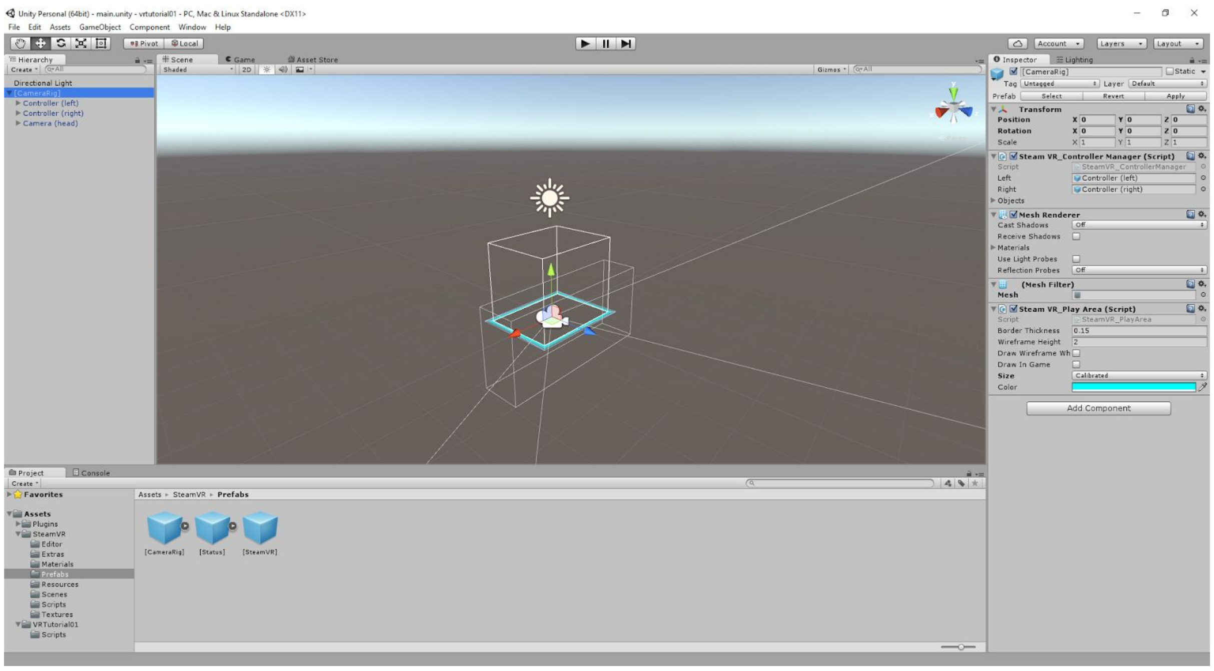


Figure 5.20: Comparison between Linear and Gamma Color Space in different intensities

SteamVR comes with a prefab CameraRig to use for VR. It includes support for the controllers and is what was used as our main camera in the game. This prefab also had the option to draw the play area around us, so that we know our boundaries. However, it was disabled because it interfered with the immersiveness. The boundaries though still appear if we move to close to the edges of our play area.



[CameraRig] Static
Tag Untagged Layer Default
Prefab Select Revert Apply

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Steam VR_Controller Manager (Script)
Script SteamVR_ControllerManager
Left Controller (left)
Right Controller (right)
Objects

Mesh Renderer
Cast Shadows Off
Receive Shadows ☐
Materials
Use Light Probes ☐
Reflection Probes Off

(Mesh Filter)
Mesh

Steam VR_Play Area (Script)
Script SteamVR_PlayArea
Border Thickness 0.15
Wireframe Height 2
Draw Wireframe Wh ☐
Draw In Game ☐
Size Calibrated
Color

Figure 5.21: The drawn Play Area in the CameraRig

5.3.2 VRTK Integration

VRTK is basically a collection of scripts for VR use. It requires SteamVR in order to function and comes with a variety of actions and methods. Its scripts are used like any other script, by dragging them to our GameObject's component list. Based on what we want to do, we have a variety of actions, ranging from controller to headset actions or a combination of both such as pressing a button to teleport where our headset points.

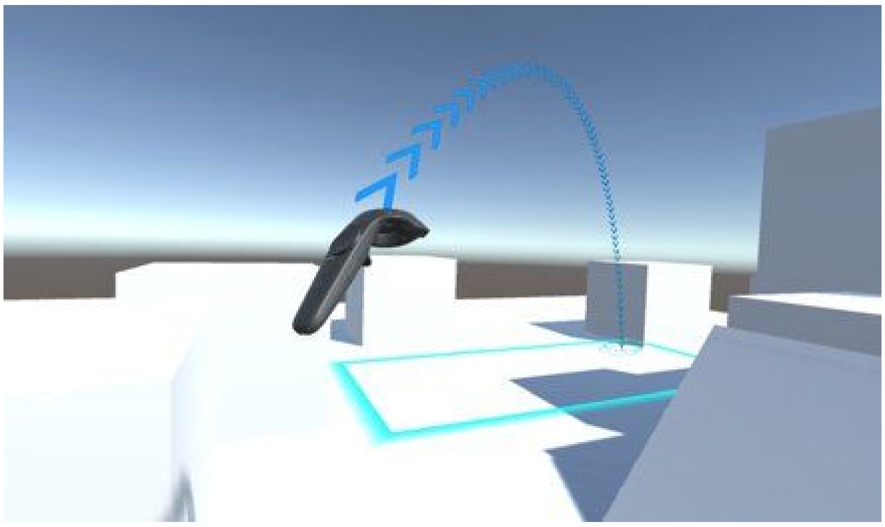


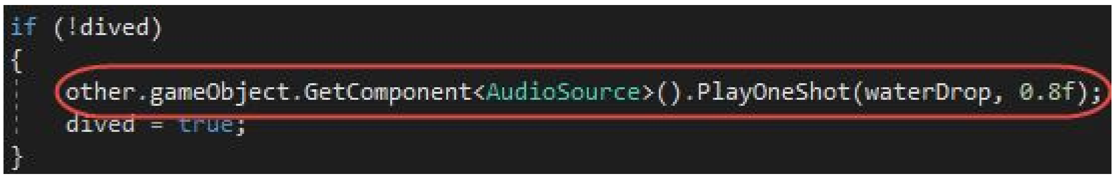
Figure 5.22: Teleporting action with VRTK

5.4 Audio

Audio plays an important role when trying to simulate any environment. Virtual reality experiences are no different. It is not the same being inside a forest without hearing the wind blowing, birds singing, etc. unless you are trying to achieve an eerie atmosphere. In this game a variety of sounds were used. Ambient noises and sound effects were primarily found in sites with free sounds and/or youtube and then edited in Audacity to reduce the file size and crop them to our needs.

Audio Source components were added to the GameObjects that the sound is supposed to emanate from and the sound clip was loaded. If we wanted the sound to play from the start the "Play on Awake" option is checked and if we want the sound to loop endlessly the "Loop" option is checked as well. Best volume for each sound was decided based on testing. Last but not least, we had the option to make the sound "appear" as 3D. What this means is that if we have audio emanating from a GameObject directly in front of us and turn our head 90º to the left we will only hear the audio from the right headset speaker. This proves very useful in situations where we have to identify the exact position of an enemy. However, in this game a semi-approach solution was decided as best, with the Spatial Blend set to 0.5, 0 being completely 2D and 1 being completely 3D.

Some sounds were required to play after a certain event took place. This can only be achieved through scripting. We pass a reference of the GameObject's Audio Source component to the script and we call the `PlayOneShot` method. This method takes two arguments, the first is the audio clip to be played and the second is the volume at which it will be played. An example of the use of this method is shown below.



```
if (!dived)
{
    other.gameObject.GetComponent<AudioSource>().PlayOneShot(waterDrop, 0.8f);
    dived = true;
}
```

Figure 5.23: Playing a sound through a script

5.5 Optimization

Optimization plays an important role in video games, especially when it comes to Virtual Reality which puts a big strain on the GPU compared to simple games that are played on a monitor. Developers have found several tricks to reduce the load on the GPU and this is basically done by rendering as little as possible. This is one of the main reasons low poly games have become so widespread in the VR gaming communities.

5.5.1 Terrain Smoothing

One of the first techniques to use to reduce tris count, was terrain smoothing. This basically interpolates between points in the mesh of the terrain and extracts a median. This automatically reduces the faces of the terrain that the GPU has to render and improves the frame rate. A free C# script was used for this, to smooth the terrains that our height maps generated. It is quite common for a terrain to have a jagged look after a height map is applied to it, since the engine tries to map the pixels of the height map onto the coordinate system of the terrain.

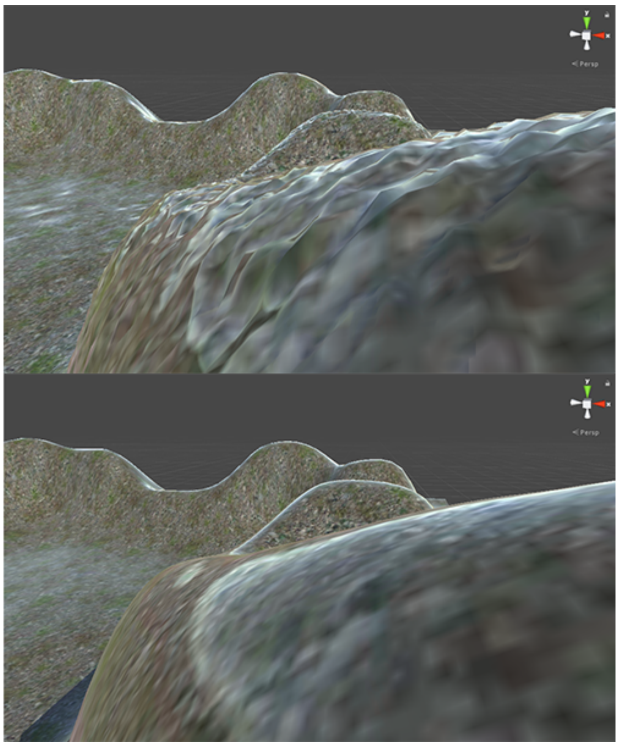


Figure 5.24: Top: Jagged terrain after the height map
Bottom: Smoothed terrain

5.5.2 Lighting

Baking of the lighting, along with the Occlusion Culling we are going to see next are two of the best techniques for optimization. The calculations for light bounces and all the lighting elements are done beforehand and are stored in images called Lightmaps. So, at runtime, Unity already knows how to draw a certain pixel that is always going to be there at only has to handle light calculations for moving and spawnable objects.

First thing we have to do to apply this technique, is set any objects like the terrains, the trees, the rocks, etc that are static objects as Lightmap Static. This “tells” Unity that this object will not change its transform and can perform the light calculations for it based on that. Apart from setting the object as static, we also have to set its material's Global Illumination as Baked.

Third parameter to be set is the Light itself. In our scene we have a Directional Light source as our primary light, which represents the sun. Light components in Unity have 3 options for Baking, "Realtime", "Baked" and "Mixed". Realtime means that no light emission from this object will be baked onto a lightmap and will always be calculated at runtime. Baked means that all light emission from this Light will be calculated for Lightmap static objects and will ignore all non-Lightmap static. And finally, Mixed means that light emission will be calculated for the Lightmap static objects and in addition, at runtime, this light will also contribute to the moving objects' rendering. Since we have both static and moving objects in our game, the Mixed option was chosen.

To actually Bake the lighting after performing all the steps mentioned above, one final step is required. We have to open the Window -> Lighting panel and choose our general Lighting settings. In this window we can choose our Skybox and whether our Ambient light is multicolor or one color or even reflected from our Skybox, and then choose whether to bake that as well or have it as Realtime. Since it would not really make that much difference in our eyes and it would save as quite a lot of fps, we chose to bake it. The rest of the settings for the Precomputed Realtime GI and the Baked GI were left at default. After choosing the settings and pressing "Build", the light calculations begin. Depending on the hardware in our disposal and the settings we choose, this can take from a few minutes to a few hours. Obviously, higher resolution settings equal to more processing time.

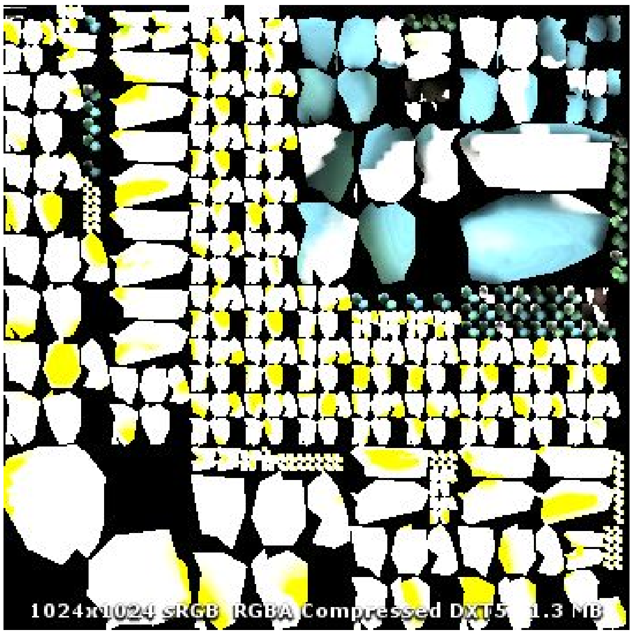


Figure 5.25: A lightmap image

5.5.3 Occlusion Culling

Occlusion culling is another neat trick for optimizing a game's framerate. All major game titles and even indie ones use occlusion culling, since it is a fast, easy and cheap way to save rendering time. Unity3D has it already built-in, but it is not enabled by default when you create a project.

In order to use Occlusion Culling, there is some manual setup involved. First, your level geometry must be broken into sensibly sized pieces. It is also helpful to lay out your levels into small, well defined areas that are occluded from each other by large objects such as walls, buildings, etc. The idea here is that each individual mesh will be turned on or off based on the occlusion data. So if you have one object that contains all the furniture in your room then either all or none of the entire set of furniture will be culled. This doesn't make nearly as much sense as making each piece of furniture its own mesh, so each can individually be culled based on the camera's viewpoint.

You need to tag all scene objects that you want to be part of the occlusion to Occluder Static in the Inspector. The fastest way to do this is to multi-select the objects you want to be included in occlusion calculations, and mark them as Occluder Static and Occludee Static. Completely transparent or translucent objects that do not occlude, as well as small objects that are unlikely to occlude other things, should be marked as Occludees, but not Occluders. This means they will be considered in occlusion by other objects, but will not be considered as occluders themselves, which will help reduce computation. When using LOD groups, only the base level object (LOD0) may be used as an Occluder.

Upon setting all the objects you want as Occluders and/or Occludees, you go to Window -> Occlusion Culling to open the Occlusion window. In this panel, you can work with occluder meshes, and Occlusion Areas. We have not used Occlusion Areas in our game since we wanted the whole scene to be included in the calculations, so this part will not be covered. In the Object tab of the Occlusion Window and have a Mesh Renderer selected in the scene, you can modify the relevant Static flags.

The Bake window has a "Set Default Parameters" button, which allows you to reset the bake values to Unity's default values. These are good for many typical scenes, however you will often be able to get better results by adjusting the values to suit the particular contents of your scene. The three values available are explained below.

- **Smallest Occluder** - The size of the smallest object that will be used to hide other objects when doing occlusion culling. Any objects smaller than this size will never cause objects occluded by them to be culled. For example, with a value of 5, all objects that are higher or wider than 5 meters will cause hidden objects behind them to be culled (not rendered, saving render time). Picking a good value for this property is a balance between occlusion accuracy and storage size for the occlusion data.

- **Smallest Hole** - This value represents the smallest gap between geometry through which the camera is supposed to see. The value represents the diameter of an object that could fit through the hole. If your scene has very small cracks through which the camera should be able to see, the Smallest Hole value must be smaller than the narrowest dimension of the gap.

- **Backface Threshold** - Unity's occlusion uses a data size optimization which reduces unnecessary details by testing backfaces. The default value of 100 is robust and never removes backfaces from the dataset. A value of 5 would aggressively reduce the data based on locations with visible backfaces. The idea is that typically, valid camera positions would not normally see many backfaces - for example, the view of the underside of a terrain, or the view from within a solid object that you should not be able to reach. With a threshold lower than 100, Unity will remove these areas from the dataset entirely, thereby reducing the data size for the occlusion.

We found that the default settings did a pretty good job, so we did not change them. By clicking Clear, all the calculated data are removed and by clicking Bake they are calculated again. Once the data is generated, you can use the Visualization tab to preview and test the occlusion culling. This takes place in the Scene View.

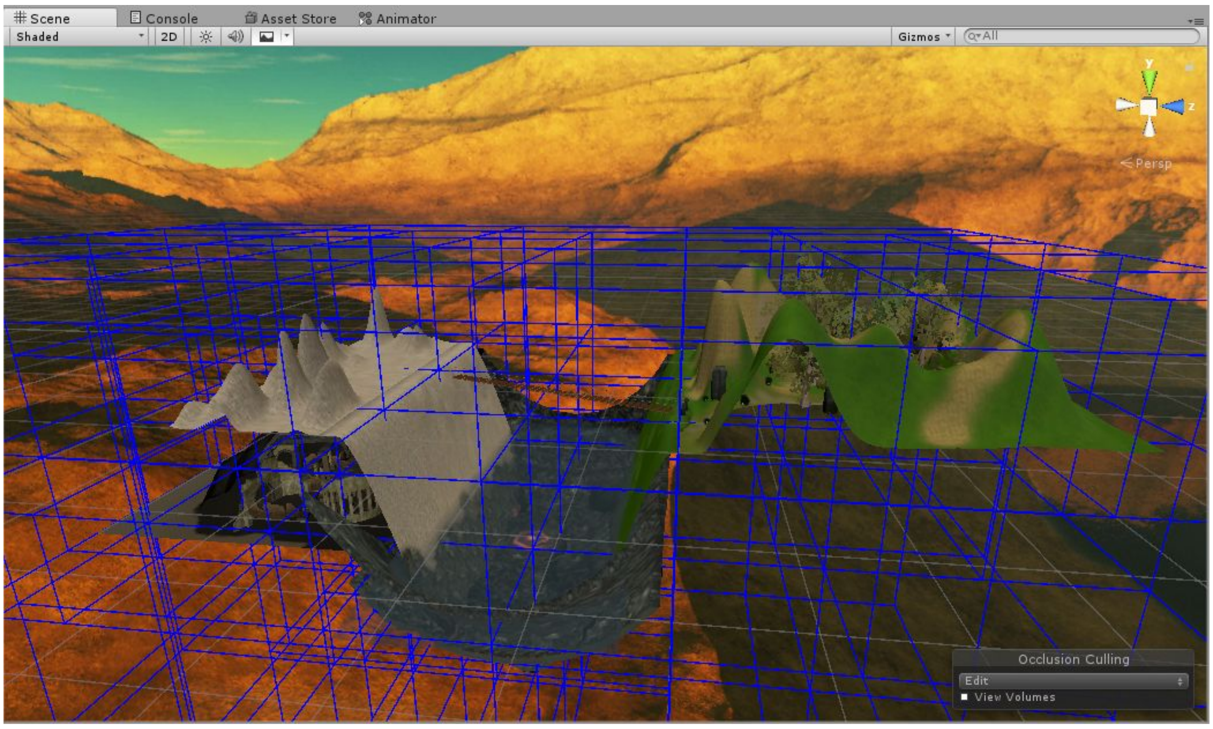
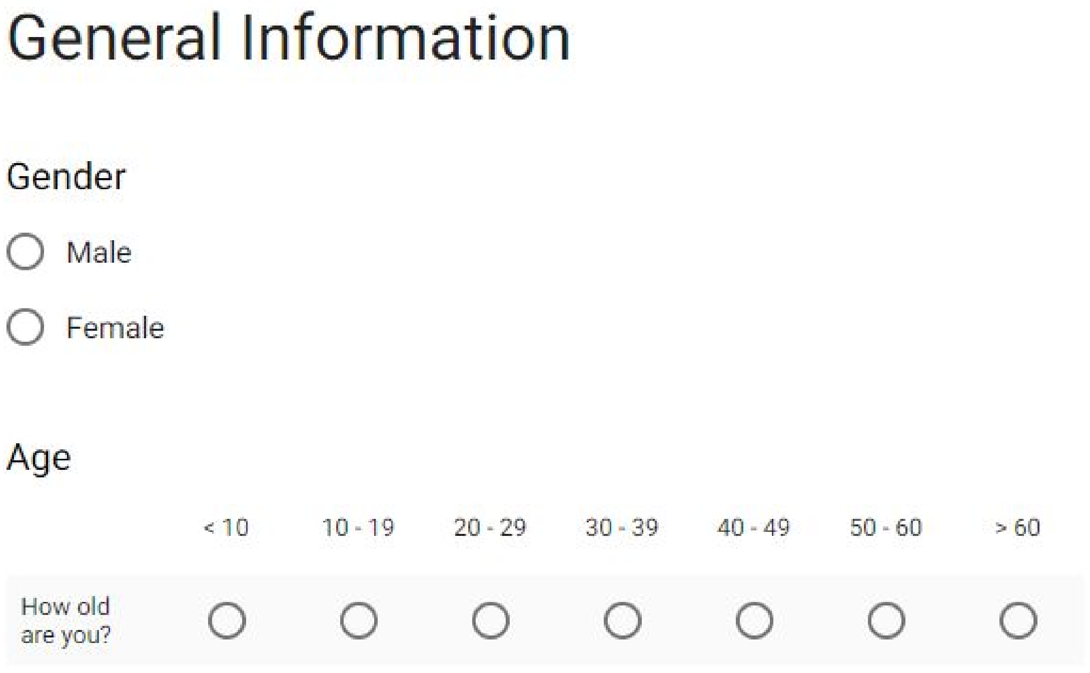


Figure 5.26: Breaking the scene as part of the Occlusion calculations

6 Evaluation, Conclusions and Future Work

6.1 System Evaluation and User Feedback

Even from the first stage of the application, user feedback has been a crucial part of the process. Constant changes were being applied to the game based on the testers' feedback. A questionnaire for the virtual experience [14][15] was constructed and further developed to include the network aspect of the application. It consists of 3 main parts, General Information, Immersion and Simulator Sickness. The aim was to acquire some general information on the testers and then see how well they do in the game. Immersion and sickness were taken into account and the results were considered in the improvement of the game.



General Information

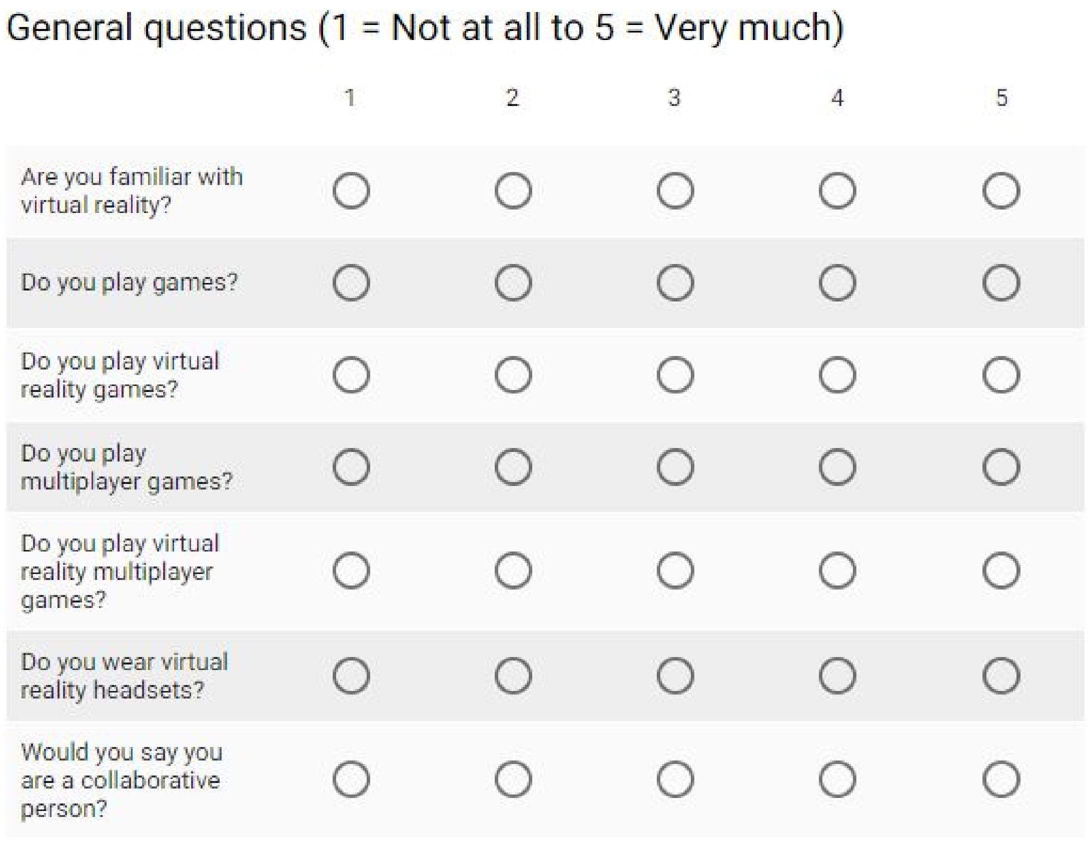
Gender

○ Male

○ Female

Age

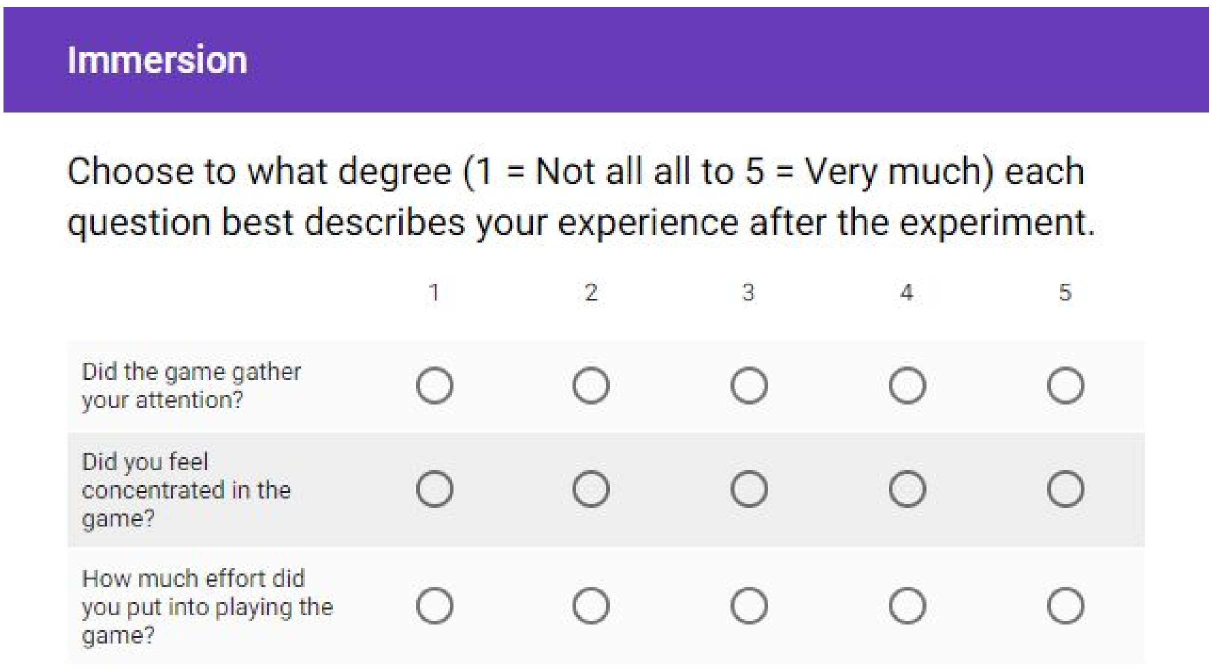
	< 10	10 - 19	20 - 29	30 - 39	40 - 49	50 - 60	> 60
How old are you?	○	○	○	○	○	○	○



General questions (1 = Not at all to 5 = Very much)

	1	2	3	4	5
Are you familiar with virtual reality?	○	○	○	○	○
Do you play games?	○	○	○	○	○
Do you play virtual reality games?	○	○	○	○	○
Do you play multiplayer games?	○	○	○	○	○
Do you play virtual reality multiplayer games?	○	○	○	○	○
Do you wear virtual reality headsets?	○	○	○	○	○
Would you say you are a collaborative person?	○	○	○	○	○

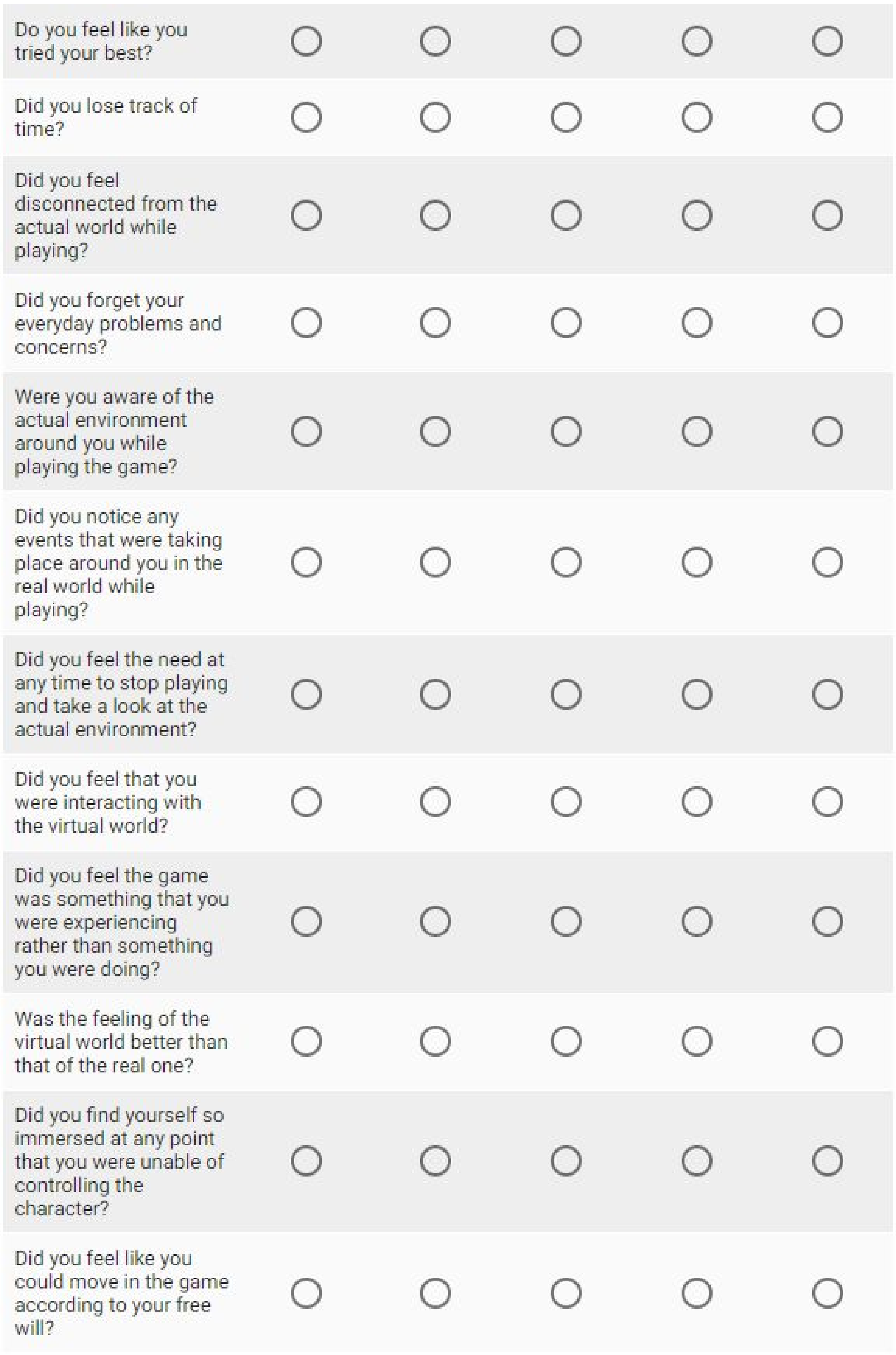
Figure 6.1: Questionnaire's General Information section



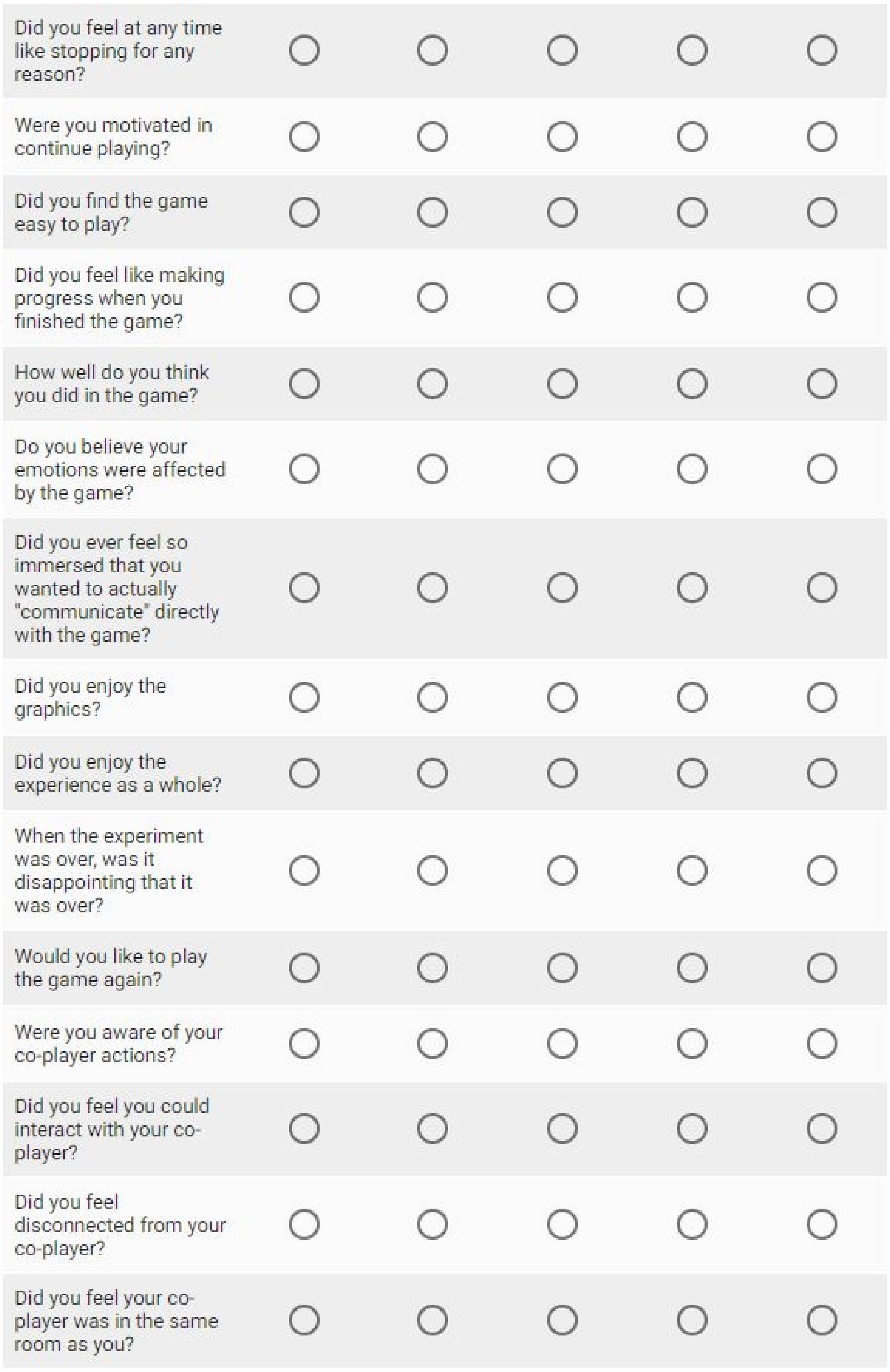
Immersion

Choose to what degree (1 = Not all all to 5 = Very much) each question best describes your experience after the experiment.

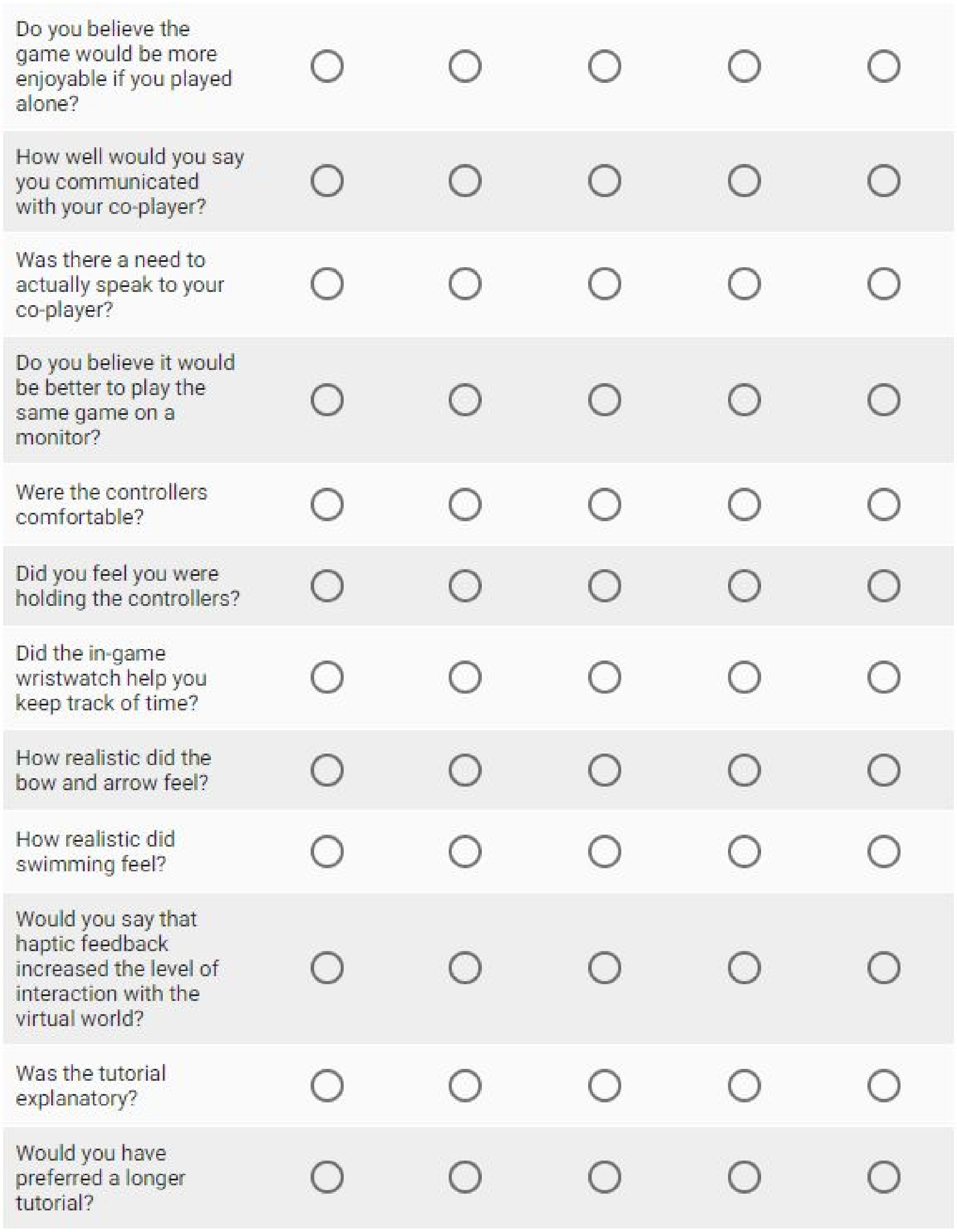
	1	2	3	4	5
Did the game gather your attention?	○	○	○	○	○
Did you feel concentrated in the game?	○	○	○	○	○
How much effort did you put into playing the game?	○	○	○	○	○



Do you feel like you tried your best?	○	○	○	○	○
Did you lose track of time?	○	○	○	○	○
Did you feel disconnected from the actual world while playing?	○	○	○	○	○
Did you forget your everyday problems and concerns?	○	○	○	○	○
Were you aware of the actual environment around you while playing the game?	○	○	○	○	○
Did you notice any events that were taking place around you in the real world while playing?	○	○	○	○	○
Did you feel the need at any time to stop playing and take a look at the actual environment?	○	○	○	○	○
Did you feel that you were interacting with the virtual world?	○	○	○	○	○
Did you feel the game was something that you were experiencing rather than something you were doing?	○	○	○	○	○
Was the feeling of the virtual world better than that of the real one?	○	○	○	○	○
Did you find yourself so immersed at any point that you were unable of controlling the character?	○	○	○	○	○
Did you feel like you could move in the game according to your free will?	○	○	○	○	○

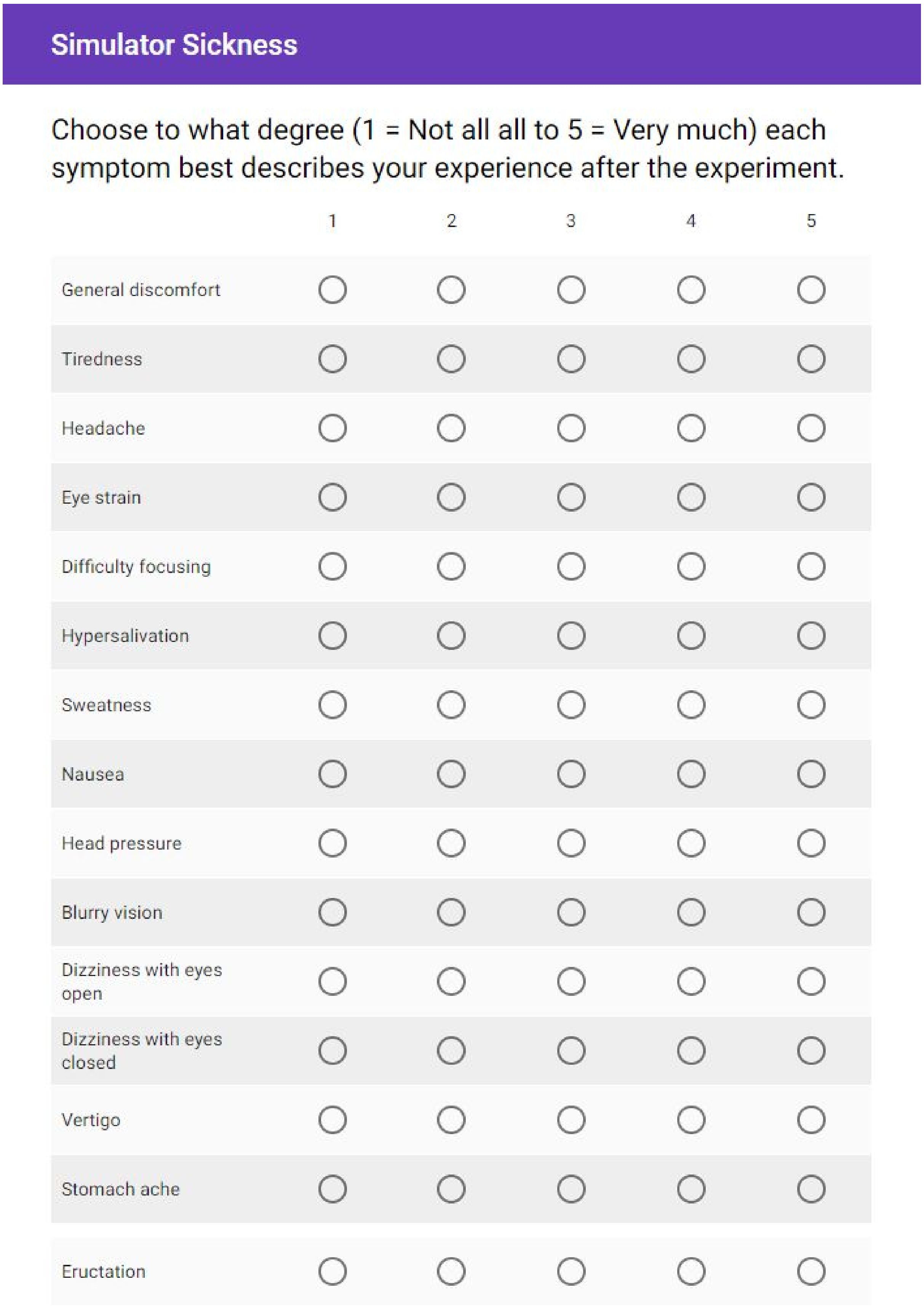


Did you feel at any time like stopping for any reason?	○	○	○	○	○
Were you motivated in continue playing?	○	○	○	○	○
Did you find the game easy to play?	○	○	○	○	○
Did you feel like making progress when you finished the game?	○	○	○	○	○
How well do you think you did in the game?	○	○	○	○	○
Do you believe your emotions were affected by the game?	○	○	○	○	○
Did you ever feel so immersed that you wanted to actually "communicate" directly with the game?	○	○	○	○	○
Did you enjoy the graphics?	○	○	○	○	○
Did you enjoy the experience as a whole?	○	○	○	○	○
When the experiment was over, was it disappointing that it was over?	○	○	○	○	○
Would you like to play the game again?	○	○	○	○	○
Were you aware of your co-player actions?	○	○	○	○	○
Did you feel you could interact with your co-player?	○	○	○	○	○
Did you feel disconnected from your co-player?	○	○	○	○	○
Did you feel your co-player was in the same room as you?	○	○	○	○	○



Do you believe the game would be more enjoyable if you played alone?	○	○	○	○	○
How well would you say you communicated with your co-player?	○	○	○	○	○
Was there a need to actually speak to your co-player?	○	○	○	○	○
Do you believe it would be better to play the same game on a monitor?	○	○	○	○	○
Were the controllers comfortable?	○	○	○	○	○
Did you feel you were holding the controllers?	○	○	○	○	○
Did the in-game wristwatch help you keep track of time?	○	○	○	○	○
How realistic did the bow and arrow feel?	○	○	○	○	○
How realistic did swimming feel?	○	○	○	○	○
Would you say that haptic feedback increased the level of interaction with the virtual world?	○	○	○	○	○
Was the tutorial explanatory?	○	○	○	○	○
Would you have preferred a longer tutorial?	○	○	○	○	○

Figure 6.2: Questionnaire’s Immersion section



Simulator Sickness

Choose to what degree (1 = Not all all to 5 = Very much) each symptom best describes your experience after the experiment.

	1	2	3	4	5
General discomfort	○	○	○	○	○
Tiredness	○	○	○	○	○
Headache	○	○	○	○	○
Eye strain	○	○	○	○	○
Difficulty focusing	○	○	○	○	○
Hypersalivation	○	○	○	○	○
Sweatness	○	○	○	○	○
Nausea	○	○	○	○	○
Head pressure	○	○	○	○	○
Blurry vision	○	○	○	○	○
Dizziness with eyes open	○	○	○	○	○
Dizziness with eyes closed	○	○	○	○	○
Vertigo	○	○	○	○	○
Stomach ache	○	○	○	○	○
Eructation	○	○	○	○	○

Figure 6.3: Questionnaire’s Simulator Sickness section

Results

The testing and the completion of the questionnaires was done by 12 pairs(24 people), 17 male and 7 female. Security measures were taken into account, to prevent the testers from hurting themselves or others during the process. There was no discrimination between those with experience and those without, meaning the pairs were assigned randomly. Results varied slightly between those who had experience with virtual reality headsets and had played games and those with no experience at all. Specifically, 15 people had experience with virtual reality, 20 were gamers and 12 had gaming experience in virtual reality. Unfortunately, the age group was not spread across the spectrum since the testers were mostly friends and colleagues at the age of 20-39. In the figure below we can see a summary of the answers given in the General Questions.

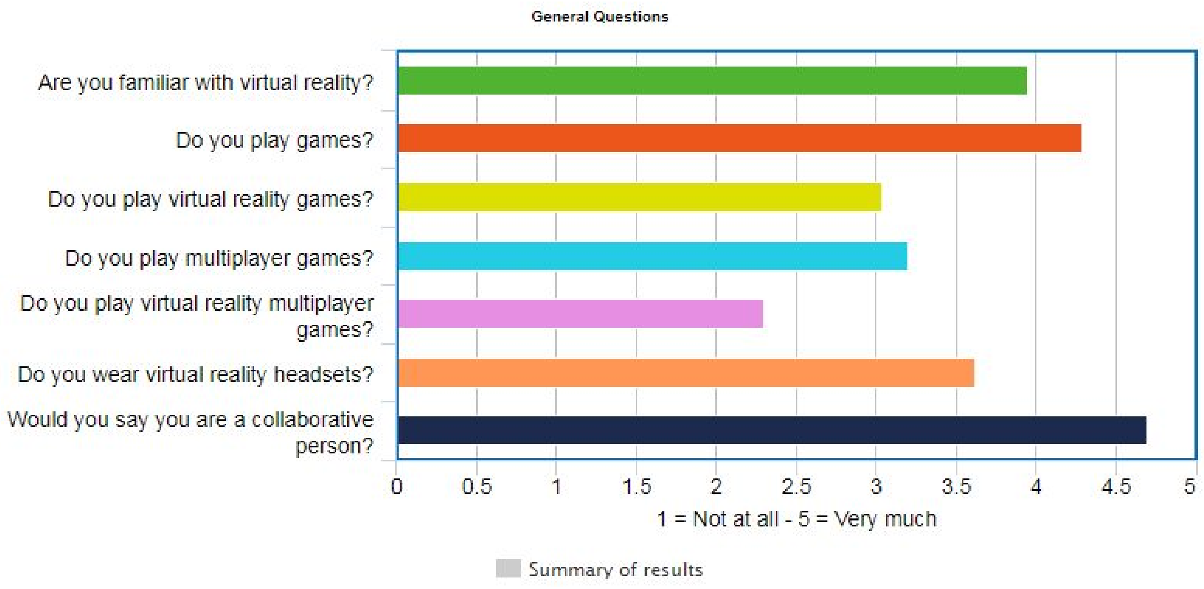


Figure 6.4: Summary of General Questions results

Common results were the liking of the tutorial system and its ease to understand. 15 people said they felt disconnected from the real world while playing and all enjoyed playing with a partner and stated it would not be the same experience without it. 2 people abandoned the experiment because they felt frightened of moving forward - both gave up when they reached the bridge in the Forest - and 20 stated they would like to play again. The partners of those who abandoned the game were matched together to complete the experiment. Interactivity in the game was highly praised and they all stated the immersion was way higher than what was anticipated. More detailed results are shared in the following figure.

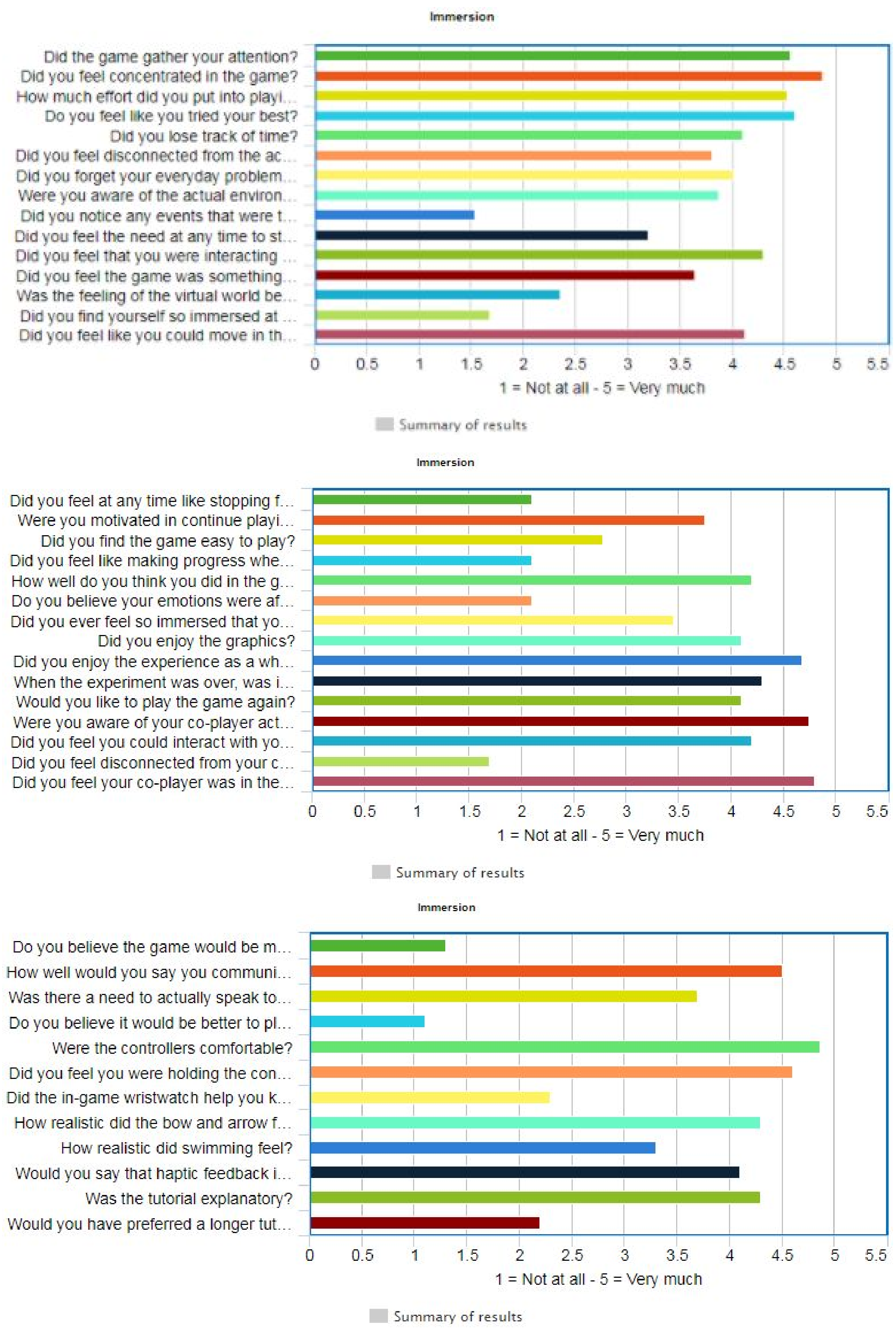


Figure 6.5: Summary of Immersion results

Those with virtual reality experience vastly enjoyed the game and mentioned little to none simulation sickness. Nausea, eye strain and sweatness were the only symptoms that gathered points in the range of 2 and 3, and even that were only 3 people. Such symptoms were expected to a degree since the lenses in virtual reality headsets are still far from perfect and researches are still ongoing on the subject. More detailed results are demonstrated below.

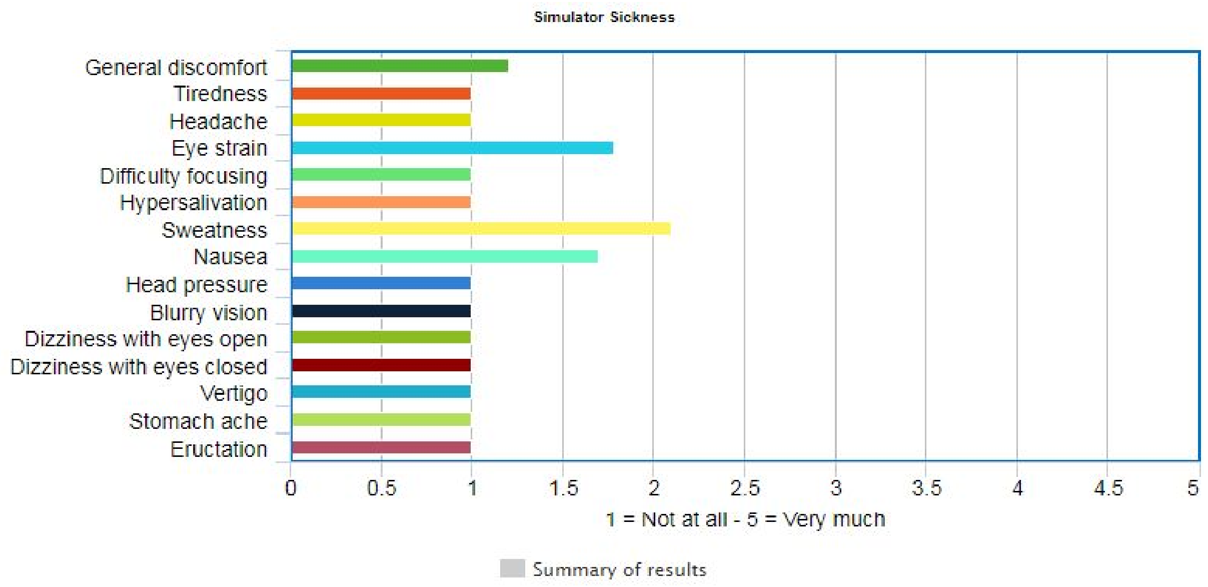


Figure 6.6: Summary of Simulator Sickness results

The testers with no previous virtual reality experience were more prone to simulator sickness at the beginning and more specifically, motion sickness and nausea but became used to it after a few minutes and did not express a desire to abandon the experiment. They were also more afraid to perform certain actions due to them not having any VR experience but we reassured them it was totally safe and continued the experiment successfully.

As already mentioned, constant feedback was crucial in the development process of the game. No questionnaire was constructed for this part and users were left to express themselves freely when they felt the need to. Examples of implementations or changes made due to user feedback are the in-hand help text screen to guide them through the game, the vibration haptic feedback for a more realistic experience and the addition of the flashlight in the right controller to light up the dark areas of the game and aid navigation.

6.2 Conclusions and Future Work

With all the recent advancements in technology and hardware, it is pretty clear that virtual reality is here to stay. From simple games to military applications, the possibilities are endless. Everyday, more and more people gain access to a virtual reality system and the majority of them want to connect with their families and peers in this new virtual world. We created a game where two people from anywhere in the world can put on their HMDs, grab their controllers and jump in a fun and immersive cooperative environment. In order to create the best experience possible, this environment was constantly modified in the development process based on the useful feedback of our testers.

It was a challenge to develop a single application targeting multiple virtual reality systems and their controllers. And, of course, connecting those VR systems in a single multiplayer environment. Many aspects had to be taken into account. Creating a realistic 3D world with high interactivity was step one. Performance was next, since it plays a major role on how the user perceives this environment - very low frame rate results in nausea and motion sickness. Implementing the multiplayer features by keeping the network load to a minimum. And, last but not least, constant testing from our users to eliminate all the negative aspects of our game.

There are, however, several aspects of the game that could be improved and expanded. The in-game version of the players could be made more realistic to include full body models or even 3D scanned and implemented as this would definitely increase the immersion. Considering the game was developed for a two-player mode, several design changes could be made to include tasks for even more cooperative players and thus even more fun. It would also be interesting to see what other interaction mechanisms someone could implement, such as maybe flying(which can be done with some changes in the swimming logic that was implemented) or even driving. Performance wise, some models could definitely be lighter in poly count or have more LODs included.

All in all, as amazing as virtual reality may be, it still remains in its “teens” and several problems exist and are waiting to be solved. Motion sickness and nausea, for example, plague a percentage of the population regardless of the application. With all the research being conducted on virtual reality, the future looks very promising.

References

[1] VR will be in "almost every household", June 2014
http://www.gamesindustry.biz/articles/2014-06-20-vr-will-be-in-almost-every-household-oculus

[2] Facebook Spaces finally delivers on social VR, April 2017
https://www.engadget.com/2017/04/18/facebook-spaces-hands-on

[3] Inside-out v Outside-in: How VR tracking works, and how it's going to change, May 2017
https://www.wareable.com/vr/inside-out-vs-outside-in-vr-tracking-343

[4] What is Virtual Reality?
https://www.vrs.org.uk/virtual-reality/what-is-virtual-reality.html

[5] HMD Specifications, Displays, Lenses And FoV, May 2016
http://www.tomshardware.co.uk/vive-rift-playstation-vr-comparison,review-33556-3.html

[6] Constellation
https://xinreality.com/wiki/Constellation

[7] Lighthouse
https://xinreality.com/wiki/Lighthouse

[8] Unity Tutorials [Graphics] - Introduction to Precomputed Realtime GI
https://unity3d.com/learn/tutorials/topics/graphics/introduction-precomputed-realtime-gi

[9] Hidden surface determination
https://en.wikipedia.org/wiki/Hidden_surface_determination

[10] Paul Read, Mar-Paul Meyer, Restoration of Motion Picture Film
https://books.google.gr/books?id=jzbUUL0xJAEC&pg=PA24&redir_esc=y#v=onepage&q&f=false

[11] Kuan-Ta Chen, Polly Huang, and Chin-Laung Lei. Effect of Network Quality on Player Departure Behavior in Online Games, IEEE Transactions on Parallel and Distributed Systems, Volume 20, Issue 5, May 2009, pp. 593-606

[12] T. Yasui, Y. Ishibashi, and T. Ikedo. Influences of network latency and packet loss on consistency in networked racing games, Proceedings of NetGames '05. ACM Press, 2005, pp. 1-8

[13] Ken Wasserman, Tim Stryker. Multimachine Games, BYTE Magazine, December 1980, Volume 5, Number 12 - Adventure, pp. 24

[14] C Jennett, AL Cox, P Cairns, S Dhoparee, A Epps, T Tijs, A Walton, Quantifying the experience of immersion in games, International journal of human-computer studies 66 (9), 641-661, 2008

[15] Kennedy, Lane, Berbaum, Lilienthal, Simulator Sickness Questionnaire, Graphics Interface, 1993

[16] http://tangledtech.com/wp-content/uploads/2013/08/Atlas-Real-Life-Holodeck-Technology-400x250.png

[17] https://cgcookie.com/articles/maximizing-your-unity-games-performance