Virtual Reality Football Videogame
A Social Experience

Bruno Rodrigues
bruno.rodrigues@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

May 2019

Abstract

Technology is in constant evolution, and the gaming industry is not an exception. The latest developments have been in the Virtual Reality (VR) field, with powerful hardware being developed and release by some of the world's biggest companies, but big gaming studios have not been following the same path. VR games have the potential to take the gaming level one step forward by putting the player in the game's virtual world and creating endless possibilities to be explored by developers' creativity. However, in VR users can feel isolated from the people and the world around them, so shared VR experiences play an important role in the success of VR.

In order to better understand the inherent challenges of the creation of VR content, we propose the development of a VR game. This game should be played by two people, stressing the need of exploring VR as a way of interaction and communication between people, instead of delivering isolated experiences. It consists in a football game with the players facing each other. It is not supposed to be a game on which players run through the pitch; instead they are in their respective goal, kicking the ball in order to score on the opposite goal, and trying to defend the other player's kicks. By making this game a shared experience, we intend to create and strengthen ties between people through a method so propitious to this, which is simply having fun with another person.
Keywords: Virtual Reality; Head-Mounted Display; Football; Social Game

1. Introduction

Today's VR is still taking its first steps. After an unsuccessful "VR fever" in the 90's the technology is now reviving under new studies, developments and hardware capabilities, and the first generation of commercial Head-Mounted Displays (HMD) is already in the market. Since 2012, with the Kickstarter campaign for the Oculus Rift, VR has started a renaissance from its failure in the 90's. Different companies started developing their products, and most arrived to the market by 2016.

Opinions still diverge, but with the continuous investment in the technology by some of the main companies in the area, we believe this time VR is here to stay. Nevertheless, only hardware isn't enough, and so it is fundamental that new innovative games and experiences continue to be developed.

By nature VR is a more personal experience. When people put the headset on they get separated from the ones around. In general humans have the need to socialize, and that doesn't change by strapping on a VR headset [17]. We all enjoy sharing the things we enjoy with the people we like. That is one of the strengths of social media, how easily we can share things with others. Individual games have the potential of creating "more real" experiences, because it is only the player and the virtual world, with no outer interference. But from being real they can start being lonely. That is why VR social experiences can play an important role in the success of VR.

VR social experiences, either in a multiplayer game with strangers, or in a social chat with a group of friends in a virtual environment, have the ability of transporting the real world connections to the virtual world, simulating not only our presence, but other's as well.

2. Background

2.1. What is VR

The term VR, as we use it today, consists of replacing our reality, our real world, with a new simulated environment, that from our perspective appears as real.

We are aware of the world around us through our senses: sight, taste, smell, hearing and touch. This means that our experience of reality is a combination of the information given by our senses,

and how our brain processes that information. That said, what if we present our senses with made-up information? The way we see our world would also change as a response to it [24]. This is what VR is all about, tricking the senses with "false" information in order to make us perceive *another reality*.

In a more technical term, VR is a 3D computer generated environment presented to the user in a way that makes it feel and appear like a real environment, with which users can interact.

To interact with a virtual world, the user can wear special goggles, called HMDs. These goggles have either one or two High Definition (HD) screens through which the user sees the Virtual World. To date, the first generation of HMDs are already in the market, available from different brands, specifications and purposes.

2.2. Exploring VR

Virtual Reality comprises a collection of technologies: 3D displays, motion tracking hardware, input devices, software frameworks, and development tools. [13, p. 7].

VR has two main components: the stereoscopic displays, or HMDs, and the motion tracking hardware. These are the components that make it possible for us to actually see the simulated environment and explore it.

From a functional point of view, we can also divide VR into two categories: detection and visualisation. Detection is the input of the system, in which the tracking system is included. Visualisation is the output, and correspond to what the user sees through the HMDs display.

2.2.1 Detection

In the real world, the things we see, hear and touch are a direct mapping from the world itself. When talking about VR this is not true anymore, as we don't see what is in front of us. However, the simulated environment has to be aware of changes in the real world, at least the user's head, because a head movement requires a change in the virtual world. If other real world objects are being mapped to that virtual world they have to be tracked too, otherwise the experience will not correspond to what is expected to happen. This is one of the key factors to cause the user a sense of total immersion.

When designing a virtual experience, the way the world is graphically presented to the user and the graphics quality are very important. But if we really want the user to be fully immersed, to be *present* [4], it is crucial to turn real world actions and movements into virtual ones. If the user looks left, the "virtual user" has to look left and the image has to reflect that. When this take some time to happen, or does not happen at all, it causes confusion and may lead to motion sickness.

The system detects real world changes through its sensors. All HMDs have incorporated sensors with the purpose of tracking head movements. The tracking technology used is different from headset to headset, but they all enable the user to look around in an immersive way. While some require laser sensors and cameras, others use incorporated gyroscopes and accelerometers to recognise movements within a 3D space.

In addition to head tracking technology, other equipment can be used to provide more in-depth experiences. The Kinect [7], introduced by Microsoft for the Xbox 360 game console, is a widely used technology due to its low price and high quality performance. The Kinect v2 (released for Xbox One) provides an HD color camera, a depth sensor, infrared emitters and an array of microphones [8]. Being built for gaming, these sensors work together with the objective of tracking full body motion. Given Kinect's great results, it is widely used for research and academic purposes. If integrated with an HMD, it would make it possible to track the entire player and not only head movements.

2.2.2 Visualisation

Regarding HMDs we can divide them into three types: tethered, smartphone and standalone. The first, like HTC Vive, has its own display and works connected to a computer through a cable. The smartphone type, like Samsung VR, uses smartphones as the HMD display. Finally, the standalone type, like Oculus Go works by itself, not needing a computer nor a smartphone.

All have pros and cons. The tethered ones have the advantage of having greater computing power and much better life-like graphics. However, as they have to be wired to a computer, portability or walking through the room is not an option. The smartphone type is the exact opposite, while being wireless and portable, the rendering capability cannot be compared to a computer, and graphics are more rudimentary. In between them are the standalone, which offers a VR experience more powerful than smartphone VR but less powerful than tethered VR [18].

Another difference between these types is the technology they have, use or rely on. While the first type described uses cameras and sensors to track movements, and it is up to the computer to process that information, the second one works based on the smartphone sensors like gyroscope and accelerometer. The third type has all the processors and sensors incorporated like a smartphone, but can dedicate all the processing power to its end.

Another advantage of standalone headsets is the cost, it doesn't need an expensive computer nor smartphone.

2.3. Current VR Problems

2.3.1 Industry

For the last 7 years some of the biggest companies in the world have been investing millions of dollars in order to create the best HMDs we can imagine. VR still has some problems, but the idea that stands now is that maybe things are not going as fast as initially thought, but they will get there [14].

On a keynote at VRLA in 2017 John Riccitiello, CEO of Unity, presented his view on the challenges VR would need to solve:

Price: the VR experience is expensive. This is probably the number one thing preventing it to be more consensual by now. Both Oculus Rift and HTC Vive cost over $400. Besides that, a VR-ready computer is also needed, and that can easily cost more than twice the headset value. PlayStation VR comes a little cheaper, but it also needs both the headset and the PS4 itself, making a total around $600. Even the apparently cheaper solutions like Gear VR require Samsung flagship devices.

Usability: tethered headsets are not very user friendly. There are USB cables, external cameras mount and calibration. A consumer product should be much simpler and portable.

The proof that these were valid concerns is the release of the first standalone headsets, at the end of that year. Standalone headsets clearly point at giving consumers a VR-ready experience out of the box. No cables and cameras are needed, it is easily packed, and because it is standalone equipment, the price is also lower.

2.3.2 Content Quality

Another often mentioned problem is the lack of good content, which is ultimately connected to the low number of users.

VR games have characteristics that differentiate them from the normal games. With all the greatness of VR other problems different from the usual ones also appear. The most serious problem all VR developers have to deal with and take special care when developing applications is motion sickness, also referred to as simulation sickness or VR sickness. Motion sickness "is the condition which appears due to a mismatch between the information that the person receives through his vestibular system (inner ear and brain) and the visual data" [2]. When this happens, the brain gets confused by the different signals that are being received and thinks it is because it is being poisoned. The defence mechanisms involve nausea and vomit.

Another causing problem of motion sickness is latency. In this context, latency is the delay between the user doing some movement and that movement being seen in game. Because of that, the refresh rate should always be above 60Hz to keep the game moving smoothly and prevent stuttering.

Many experiments have been done in order to understand how this effect can be mitigated or even completely removed, and some techniques were discovered. The first two techniques we'll address are almost rules. When dealing with VR it is crucial that the camera doesn't move unless the user has that intention. This means that camera animations are "forbidden". Also, when movement does happen, it is very important not to have head wobbling (those little up and down movements typical of first person games), for the dizziness and nausea it provokes [6].

In games that the player has to walk across the virtual world, the best way to do it so that VR sickness doesn't happen is not to walk at all. Instead, teleportation was found to be the best way to deal with the need of moving the player. Because the movement is instantaneous the brain doesn't get confused. The downside is that it momentarily brake the immersion. A good alternative is to make the player walk, but without acceleration. Moving at constant speed is said to really help reduce motion sickness. Another trick to add when walking is to reduce the Field of View (FoV) of the camera, because fewer objects are seen in movement by peripheral vision.

Still, sometimes there is the need to move the camera yourself and break the first "rule", like scene transitions for getting into a vehicle, falling on/getting up off the ground. Those animations involve too many head movements, more than enough to make the user dizzy. And in fact, the really important thing is the state, either inside the car or outside the car, all the in between is not needed. In those situations a common approach is to create fade through black or blink transitions. Tom Forsyth, former Software Architect at Oculus, who was involved in the creation of the SDK, perfectly described some of the referred techniques at the first edition of Oculus Connect conference, in 2014 [3].

The setup of the environment also plays a key role in avoiding motion sickness. Repetitive visual patterns, like stairs and checkered floors, should be eliminated or at least smoothed or blurred. The shaders and materials used in the scene should

also be appropriate to the VR environment, taking into account the different rendering capacities of a smartphone compared to a computer. Low frame rates also induce sickness, besides breaking the sense of immersion.

Apart from motion sickness, caution also has to be taken with the users' fatigue. By their very nature VR games are much more exhaustive than normal video games. Because the screen is very near to the eyes and the headset weights on the head, playing periods should be shorter, with more breaks in between, when compared to normal games. Using very bright colors makes the eyes get tired more quickly [2]. Some consequences of getting tired in VR and long playing periods include eye soreness, trouble focusing, dizziness, disorientation and loss of spatial awareness [5].

2.3.3 Mentality

The downside of Social VR is that some people do not know how to behave. These days people are hidden behind the anonymity of the Internet; their bad actions are able to pass undetected. On social networks we see people being rude to other people all the time, simply because they can.

Unfortunately, these kind of behaviours also migrated to Social VR. In a survey of 609 people, the results of which are available online, people have reported they avoid Social VR because of the fear of harassment [23].

Given the importance of Social VR, harassment and fear of harassment are big obstacles to the growth of VR as a whole.

For this project we thought about those VR problems and in what way we could help tackle them. We had identified those problems to be of three kinds: industry, content quality and users' mentality. It was obvious that our effort should focus on the content, as we wouldn't be able to act on the hardware and industry issues, related with pricing and mobility. We could act on the creation of mechanisms to prevent and signalise bad behaviour, but for that some psychology knowledge would also be desired.

3. Methodology

When designing the idea of what experience to create, two points were clear: it was obviously going to be something in VR, and it would need to involve at least two people.

What we came up with was the idea to create a football VR game to be played between two people, one against the other. Players are placed in a virtual football pitch, one on each goal, and have the objective of any football game, scoring more goals than the opponent. For that, players use their feet to kick the ball and try to score, and any part of the body to try to stop the other player from scoring. Although players are allowed to move freely they can only do it inside a certain area, as they are not supposed to move all around the pitch.

Another detail is that we don't want to make it as a penalty shoot-out kind of game, where before each turn the ball is placed on the mark. Instead, the ball should move freely from one player to the other, except when it gets out of reach or a goal is scored.

The social part of the project is given by having a two-player game, with the users being in the same virtual and real space. We think that the fact that they can directly talk to each other increases not only the state of presence, but also the competitiveness among them.

We divided the game implementation in two distinct main areas that we opted to tackle separately, the VR part and the Kinect part.

3.1. Kinect

The first thing we did with Kinect was installing the Kinect for Windows SDK (note that the Kinect is a Microsoft product, as so, all the development we are going to refer next is only valid for Windows). Besides the necessary libraries it also installs a set of tools, components and samples that can be useful for developers. With Kinect Studio it is possible to turn on the Kinect and visualise all types of data from the Kinect sensor like RGB image, IR image or the skeleton mapping.

Microsoft also provides a Unity package that contains some basic scripts and examples. These examples track the user and create a skeleton representation of the person detected in front of the camera.

For our game a skeleton representation wasn't enough, as we pretended for each player to have his own 3D body. It creates a better sense of immersion if the players are able to see their own arms and legs, as well as the opposite player. To create the 3D models for the players we used an online character generator provided by Autodesk [1]. With it we generated a 3D avatar dressed as a football player, so we could start playing with skeleton animations. We also took the chance to start thinking about the visual aspect of our game scene. It was still quite early to concern about that, but as we were working a little with 3D models we opened SketchUp, which is a program for creating 3D models, and built our own simple football field, with suitable measurements and a simplistic style. Finally, we put our model inside the pitch, creating our game and test scene for the next parts.

From there we passed to the next step, which was having our own 3D avatar being controller by

Kinect, and that turned out to be our first big challenge. We started by making a better research on the references we had regarding the use of Kinect with Unity, ZigFu and Middle VR for Unity. We had the idea of ZigFu being less powerful and complete, but nevertheless it was our first choice, because it had a bigger community and also we could try to get in contact with former students who had used it before. However, what we found was unexpectedly bad. It looked outdated and the documentation regarding Unity was obsolete. To sum up, the project is dead, so we changed our attention to Middle VR for Unity. This product is developed by a company of name Middle VR with offices in France and China, that is focused on the Professional business, developing AR and VR applications for other companies' needs [11]. Also, they commercialise their software for developers to use. Although not being the main focus, their software for Unity also supports interaction with most interaction devices, including Kinect [12]. Given the few information available we were not very enthusiastic about the product, and even the free version was out of question as it would force us to have a Middle VR watermark. Prices are not publicly available and there isn't any version to download; a small form has to be filled with company name, person details and role. Nevertheless, we though that if the product was capable of providing such an easy integration as they advertise, we should give it a try. So, we filled the form, asking for prices information and a trial version, but we never received back any answer.

While waiting for an answer we started searching for alternatives to help us on our problem. We could have tried to parse the Kinect data ourselves and map it to our avatar bones, but that would have been a very low-level task that although challenging and interesting didn't quite match the learning objectives we had in mind for this project.

We managed to solve the problem with the help of an Unity plugin, Kinect v2 Examples with MS-SDK and Nuitrack SDK, that can be found on the Unity Asset Store. Although being paid, the developer of this package, Rumen Filkov (which has given us great help and support every time we needed) provides it for free for any student who wants to used it for education, lecturing or research, which was our case.

3.1.1 Kinect Plugin

This plugin contains several C# scripts, resources and over 50 demo scenes. It is very complete and has examples on how to do practically everything with the Kinect, from moving custom avatars, face tracking, clothes fitting room scenes, and so much more. For our case we were only interested on the Avatars Demos. These demo scenes show a 3D avatar that is animated by the user movements. This is exactly what we wanted, so we investigated how it was being done, using which components, so that we could replicate that setup on our avatar.

Some components are needed for it to work. One of them is the *KinectManager* which is the main component of this package. The *KinectManager* is the responsible for controlling the sensor and getting the data streams and all other components rely on the data provided by it [10]. The other needed component from this package is the *AvatarController*, that transfers the Kinect-captured user motion to the humanoid model, whose component it is added to [9]. In other words, is the component responsible for making the avatar move.

The third needed component is the *Animator* component, which is provided from Unity. It is used to assign an animation to a GameObject and typically requires a reference to an *Animator Controller* [19], which defines the animation. However, in this case, we use it without any controller because the animation is given by our movements.

After adding all of this to our scene we were able to move our arms, legs and head, and even give some little steps to both sides, back and forth.

3.2. Network

After successfully completed the avatar movement, the next challenge was to create a networked demo scene on which two computers with a Kinect each would transfer the information between them so that two avatars could be moved independently from one another.

This was without any doubt the part of the project that took us the longest. We have never had any experience on creating a networked Unity application, so we had to investigate and learn the existing methods and possibilities ahead.

3.2.1 UNet

We started by studying the UNet, which is Unity's multiplayer solution. Today UNet is already being deprecated and a new Network solution is being developed by Unity to gradually replace UNet over the next two to three years, but for now it is still one option available for use.

UNet provides two different Network APIs, one High Level API (HLAPI) and one Low Level API, or Network Transport API. Given our knowledge on the area and the needs for the project the HLAPI was more suitable, as it is not an advanced multiplayer game. With this API it is possible to [20]:

- 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 *NetworkManager* is Unity's component responsible for managing all the network aspects of the game. Its responsibilities include controlling the loading of game scenes for all players, spawning new objects, keeping the game state and performing the matchmaking. It can be extended if there is the need for custom functionality or listening to any of the provided server or client callbacks. One already existing extension is the *NetworkManagerHUD* which provides a basic UI for players to join the game. In the Unity Asset Store it is also possible to find an Unity developed package called Network Lobby. Among some prefabs, scripts and resources, it has a very interesting component that is the *LobbyManager*. It provides a more advanced and prettier HUD and some extra properties when compared with the other referred Managers. Its purpose is to show the proper interface to allow players to join a game, display a pre-game lobby where players can change their name and see the other players who have joined the same room. When the minimum number of players have joined the room, a countdown appears and then the game starts.

When configuring the game scenes, they should have only one *NetworkManager* each, that can even be a persistent game object throughout all game. That object should not have any other network component, being the most common case the creation of an empty game object just for this. On all other game objects that need to be synchronised through the network we must have a *NetworkIdentity* component. It controls a GameObject 's unique identity on the network, and uses that identity to make the networking system aware of that GameObject [22]. In this component we configure the authority of that game object, that is, who is the responsible for managing this game object, the server or the client. The possible values are Local Player, Server Only or none. Besides that, the moving objects should also have a *NetworkTransform* component, which is responsible for the synchronization of the object's movement and rotation across the network.

In network games a *Server* is an instance of the game that manages several aspects of the game, which other players connect to. Each player also runs an instance of the game, and is called a *Client*. With UNet the same thing happens, but one of the clients can also be the server at the same time; this “special” game instance has the name of *Host*. This concepts of authority and the server/client relation are all explained and well documented on Unity's Manual [21].

Regarding our game we opted to use the *LobbyManager* component. To create the game, the *LobbyManager* requires the setup of a few things. First we need to create two scenes a pass them to the *LobbyManager*. The *Offline Scene* is the same scene on which we are configuring the Manager. It will be the loaded scene when the application starts. When the two clients are connected, the game transitions to the *Online Scene*, which is our scene with the pitch. Two other important properties of the lobby that need to be set are the Lobby Player Prefab and Game Player Prefab. The first is the object that represents the player while in the lobby. The second is the “real” player object, which in our case is the 3D football player avatar. Both prefabs need to have the already mentioned *NetworkIdentity*.

Our game has four main game objects: the pitch, the ball, the Kinect Manager and the game player(s). Because this is a network game we needed to set the right network components to each of them. The first “rule” is simple, attach a *NetworkTransform* to all game objects that need to move or rotate, which in our case is the ball and the player. By having a network component they also need to have *NetworkIdentity* to really work, as well as have their authority specified. For the player it is simple, it must have a Local Player authority, as it is a client game object. As for the ball, it is not supposed to be managed by any client, but also cannot exist only on the server, so the default state should be maintained. The pitch is a static object, so it does not need a *NetworkTransform*. While it is only really needed in the game scene, we choose to create it also on the lobby scene, placing it nicely as the background of the main menu and the lobby menu. Finally, the Kinect Manager object could by analogous to the pitch, as it also does not need a *NetworkTransform* and is only needed on the game scene, however we chose to make it a little different. The Kinect Manager passes trough an initialization process that depending on the network and the machines could take time enough so that the scene transition after the countdown end was not as smooth as we pretended. To deal with that, we instantiate the Kinect Manager in the lobby scene and enable the *DontDestroyOnLoad* param-

eter, because by default Unity destroys all game objects when transitioning from one scene to another. This way the object state is kept and it does not need to be initialised again.

With this we though we had everything in place, so we launched two instances of the game, one as a host and the other as a client, joined the same room and both players appeared when the game scene was loaded. However, none of the avatars was moving. We noticed that although each player object had its *AvatarController* component, the list of registered controllers on the Kinect Manager was empty. That is because it is expecting the *AvatarController's* to be already on the scene when it is initialised. As in our case that does not happens, we needed to manually register both controllers when the players are being created.

Notice how we only call *RegistPlayer* if the condition *isLocalPlayer* is verified, and if not we disable the player camera. This is because in with UNet, there are multiple instances of the Player object, one for each client. The client needs to know which one is for "themselves" so that only that player processes input and has the camera attached. This is an attribute of every network component that is set during creation. Each client will have his own player with this parameter as true because we configured the game player prefab's authority is set to Local Player.

After fixing this, we tried again. This time we could move our player, as it was already registered, but the movements were not synchronised to the other client, that is, the player stayed still. We realised that we had made a mistake about a basic concept: that adding a *NetworkTransform* to the player was not enough, because the avatar movements are not changes in the player position, but in the bones of the skeleton. There is one network component that at first sight we though could help, the *NetworkAnimator*, which allows the synchronisation of animation states for networked objects. This is what we needed, the problem is that it does that by synchronising the state and parameters from the *AnimatorController*, which we do not use as explained before. Therefore, the only solution we were seeing was to add a *NetworkTransform* to each bone and have them being synchronised one by one.

3.2.2 KinectDataServer/KinectDataClient

We put the previous solution on standby and went searching for different alternatives. Going back to the Kinect plugin we talked about in the previous section, there was a demo scene called KinectDataServer. The purpose of this demo was to show how to run a Kinect application on a non-Windows platform, for example a smartphone HMD. The (Windows) machine on which the Kinect is connected to runs an application with the *KinectDataServer* component, and the smartphone runs another application with the *KinectDataClient*, which receives the sensor data over the network (sent from the *KinectDataServer*) instead of receiving directly from the Kinect. Contrary to our approach, these components use UNet Low Level API.

Despite our needs being different from the demo's purpose, we researched and made many experiments in order to explore the way these components interacted with one another, and we ended up building a solution architecture exploiting the many configurable parameters that they provide. The basic idea was to "simulate" synchronization by having four Kinect Data Clients, connected to two Kinect Data Server. Each Kinect Data Server was responsible for sending the sensor data to two Kinect Data Clients, one on his local player and the other on the remote player.

While we eventually managed to make this solution work, there was a very noticeable delay between the person movements and the player movements, even on the local player, because it is receiving the data over the network. As we saw previously this is one of the main causes of VR sickness, and so we continued looking for alternatives.

3.2.3 Photon Unity Networking (PUN)

The next thing we tried was PUN. Photon is the number one network engine, and being Unity one of the top game engines, Photon have created PUN which is a Unity package for the creation of multiplayer games. Like the UNet, it takes care of all the game low level tasks, like handling connections, matchmaking and so on. It has some functional differences from UNet, for example while on UNet the game runs in Unity instances (the server or host), PUN uses dedicated servers for it, but overall features are relatively equivalent.

As PUN is used by a lot of people, and we didn't know what we could do with it, we decided to study it a little to see if it has anything related with Avatars that could be helpful. However, we only got to discover something similar to the *NetworkAnimator* from Unity, requiring an *AnimatorController* with the animations configuration.

3.2.4 Creepy Tracker

The *Creepy Tracker* is an open-source toolkit to ease prototyping with multiple commodity depth cameras by automatically selecting the best sensor to follow each person [15]. The ability to work with multiple Kinect sensors caught our attention,

because our project also needs two Kinects, although not in the same way, as in our case each Kinect is only tracking one person and their data does not need to be merged. Also, it is able to transfer Kinect data trough the network, used on the telepresence portal example.

Another interesting example scenario mentioned the use of Kinects for VR applications, as a way to create a large gaming area, similar to HTC Vive Lighthouse tracking system. Because of out interest on these scenarios we went to talk with Mauricio, one of the developers of the toolkit. We explained what our project was about, what we needed to do with the Kinect and questioned if the *Creepy Tracker* could be helpful for our project. After some analysis, we concluded that while it could indeed be used in some ways, it wouldn't be the most adequate project for it to be used in, as the biggest strength of the toolkit is its ability to handle the data from multiple Kinects, which is something we do not need to do.

At the end, Mauricio suggested that the approach of assigning a *NetworkTransform* to all the bones should be simple and performant, and probably would solve the problem.

3.2.5 Network Skeleton Component

As we were about to return to the multiple *NetworkTransform* approach, while doing some research about it we found a thread in a Unity Forum with the title "Synchronizing an entire skeleton" [16]. There, a member of Unity was sharing a component that "synchronizes the bones of an entire skeleton. This is intended to be used by dynamic animation systems driven by user input, such as Kinect systems doing motion capture for shared VR/AR environments". This component, that was never released with any version of Unity, answered exactly to our problem, also advertising to be much more efficient that using multiple bone *NetworkTransform*. Overall, following the usage instructions was easy, but we still had to pay attention to one point. For skeletons with too many bones, the amount of data could be larger than the supported limit of about 1440 bytes, which was our case. Our solution was to set the *SyncLevel* parameter, which tells how deep in the hierarchy should the component synchronise the bones' transforms. The deeper bones of our skeleton are the fingers of the hands, which we do not care about in this game. As so, by setting the level to sync only until the hand bone we were able to cut in the amount of data to be sent.

As a result, we were able to have each user moving its own avatar, with less latency than we had with the Kinect components.

After so many tentatives with some different technologies, it was a fulfillment to have finally achieve that goal. We were close to the solution right from the start, but it was also great that we had to explore and learn such different strategies to solve the problem.

3.3. Adding VR

Having the synchronization of skeletons complete we had two options, we could either start working on the game physics and mechanics, like being able to kick the ball, or we could start thinking about integrating VR in the project. Although strictly speaking those options were not connected, it proved to be simpler to add the VR rendering first because it was being difficult to have a notion of distance to the ball without the VR headset.

Enabling VR in Unity is a trivial task, as it is just a matter of ticking a checkbox on the player settings. But after enabling it we need to configure the game camera's properties, including its position, and by then we started having some issues to deal with. One of the problems we faced was that by placing the camera inside the player game object head we could see some interior meshes that we did not want, such as part of eyes and teeth. The other problem remits to the headset session space, and where does it consider its original position to be. We will deepen them and the found solutions.

3.3.1 Inside Head Rendering

Even before we started enabling VR on the project, we had already placed the camera inside the player's head to get the feeling of seeing through the player's eyes. However, we were not expecting to see anything besides the pitch, as by default Unity has the backface culling enabled, so it was a surprise to see the teeth and some other parts of the player. We realised this was not a matter of culling the back of some meshes, nor meshes having its normal vector pointing inwards; these meshes we were seeing were really inner parts of the head mesh.

Even though we knew this problem since the beginning of the development, we only started working on it by the time we enabled VR, because with normal camera rendering we easily moved the camera somewhere else, but know that was not possible as we need the camera to be the player's eyes.

We had some possible solutions in mind on how to counter the issue, starting by adjusting the camera's near plane. All the objects being rendered are inside the view frustum. The near plane cuts out everything that is too near the camera, while the far plane cuts everything that is too far away. The size of these planes is given by their distance

to the camera, and the camera's FoV. On our case, as we needed to exclude those meshes inside the head we needed to increase the value for the near plane.

Another alternative was to adjust the camera position, by placing it more close to the undesired meshes, so that they would get cut by the near plane. Despite both options seemed to have worked with a normal camera, with the VR headset on, the head movements and rotations showed that depending on those movements, the tracked head position by the HMD and by the Kinect can differ in a way that some artefacts would still appear. Also, because we were making changes on the near plane, it would also cut out parts we wanted to show up, like shoulders or even the hands, if placed too near the camera.

At the same time we were trying the previous solutions, it seemed quite obvious that the ideal scenario would be for those meshes not to exist at all. We imported the 3D model in 3ds Max and started deleting the interior vertices. As we are not experts on modeling in general, and 3ds Max in particular, many times we ended up deleting more than we wanted, creating "holes" on the model head and having to go back with it. We noticed that some specific elements of the model did not belong to the main body mesh, and that gave us the idea of separating the head from the rest of the body into two different meshes. So, we selected all the head vertices (much more easier than trying to delete inner vertices) and than detached from the rest of the model. When importing this modified model in Unity, the model parts now showed the head apart from the rest.

Unity allows placing the scene objects on different layers, as well as choosing which layers each game camera should render. We followed this approach, creating a new layer for those objects to be ignored, like the eyes and head. Next, we removed that layer from the camera's rendered layers. This way, no matter the different movements the player makes, the head part will never be visible.

Notice that we can only do this for the local player, otherwise we would not be able to see the other player's head.

3.3.2 Tracking Origin

Our first experience with VR in this project was with an Oculus Rift DK2. Using it was very simple, as we just positioned a game camera in the player's head and it run without problems. This was in the very beginning, even before we had the network part working. In the latest stages of the project we began developing with an HTC Vive, and we noticed things were a bit different with this one. When strting the game, the camera would be positioned way above the player. We found out this behaviour exist because these two devices use different tracking origins. There are two types of tracking origin:

- **Device Tracking Origin:** at some point during the device initialization or start of the session, its real world position is used to mark the origin of the session space. This is what happens with the Oculus Rift; the initial position of the game camera (player's head) will match the physical position of the headset. To give an example, if the headset is on the user's head when the game starts, it will appear Ok in game. On the other hand, if during that initialization the headset is placed on top of a table, that will be its origin. By the time the user straps it on the head the game camera position will rise the same distance between the table and the head.

- **Floor Tracking Origin:** the origin of the session space is the floor, and the game camera maps the real height of the device. This is used by the OpenVR SDK, which is used for using Unity with the Vive.

When a game camera is being used with VR it is not possible to change its position, because it is being computed all the time by the tracking system. If we need to manipulate its position or rotation, the way to go is to add it as a child of another game object, and manipulate this one instead. This is the correct way to handle the use of devices with different tracking origins.

On our game we have a 3D model of the player, and two things need to happen at the same time: the player's feet must be touching the ground, and the game camera must be at the eyes height. Because users have different heights, but are using the same game player, which height is fixed and does not match everyone's, only one of these restrictions was being respected if using Floor Tracking Origin. For example, if the user is taller then the player object, either we get the feet on the ground and the camera floats above the player, or stick the player's head to the camera and the feet do not touch the floor. To accomplish both the both at the same time we needed to add a parent node to the camera, which we called *FloorOffset*. As the game player height does not change, we can attribute it to a constant and then, in runtime, calculate the difference between that height and the real user's height. That difference is then applied to the offset node.

3.4. Gameplay

Having the network synchronisation of the skeletons and the integration with VR that completed it was time to start implementing the game itself, with the logic and physics.

We started by creating two different physics materials for two important pieces of the game, the ball and the ground. In the physics material we configure the bounciness and friction of all objects using that material.

So that game objects can interact with each other using the physics engine, they need to have both a *Rigidbody* component attached to them and at least one collider. For the ball the best choice was obviously a sphere collider, while for the ground a box collider was adequate. Regarding the player body we created many capsule colliders, with different sizes, that would "envolve" all the player's members. It is possible to ask Unity to compute a mesh collider adjusted to the game object's mesh, but it is much less performant. Besides, most times there is not even a noticeable difference in behaviour between an approximation like this one and a mesh collider.

When two colliders enter the same space they trigger a collision and the physics engine runs the calculations regarding the direction of the objects colliding, the mass of the rigidbody components, and other factors. Colliders also have a property to set them as being only triggers. which is used to know when two game objects intersect each other, but without creating a collision.

For the kick on the ball we attached a script to it that upon collision checks its own position and the colliding foot position in order to know the resulting direction vector. Then it applies a force of type impulse to the ball, depending on Force constant we have defined, the collision impulse and the calculated direction.

One example of use of the trigger colliders is to detect when the ball enters the goal, or eventually is able to escape the pitch. On those cases, when that event is triggered we know that we need to reposition the ball in its original position.

The other feature we planned to implement was to keep the ball always moving, so that it did not need to be re-positioned after every single kick. From our perspective, it would not only be more challenging but also provide a better immersion for the players, because in the real world objects to do teleport from one point to another, so we wanted to minimize the occurrences of this situations as possible as we could. We have given it some time to think about ways to achieve that, but we were not able to implement them yet. Nevertheless, it would involve creating a sort of attraction force in the goal direction, so that the ball does not stop far away from the players. That could be complimented with some invisible walls on all four sides of the pitch, that would rebound the ball in the goals direction.

4. Results & discussion

As the purpose of this project was entertainment and creating social interaction (instead of educational for example), the main goal we proposed to achieve was to provide users an interesting, fun and interactive experience. It was also important to test if usual problems associated with VR, like motion sickness, were avoided, otherwise users would not enjoy the time spent playing the game.

From our own experiments during the many stages of development, at first we were a little afraid of moving. Opposed to most VR games, here we really see and control our legs and arms, creating a weird but good sense of being in another body. Regarding that, one thing we noticed was how different people perceived the distance to the floor. Because the no one was the same height but the avatar's height is fixed, some got the impression of being too near the floor. But the feeling of being in a football pitch is great, despite it not having the same level of realism it could have if developed by good games studio.

Unfortunately we never got to make real tests with users, just some colleagues that helped us with during the development process by moving at the Kinect or using the headset for us, so that we could fix and tweak settings and values on Unity.

5. Conclusions

In spite of the lack of investment from big game studios in the development of good games, every day new VR content gets its way to the market. In the beginning most VR experiences were isolated, but people have started realising how fun and better VR be when shared.

Based on that, we decided to create a VR game with a social component. The result was a game on which each player is placed on one goal, and they can kick the ball from one goal to the other, trying to score and not concede goals. Having another person there playing with us really makes a difference, because even though we only see their virtual representation, we can hear and we react to the same events.

Although the gameplay is not quite finished with some parts needing to be perfected, we think that the game already provides a good amount of fun for the people trying it. We are able to kick the ball in the direction we pretend, and see the other person moving, and it makes us feel that we are indeed in the company of somebody.

References

[1] Autodesk® Character Generator.

https://charactergenerator.autodesk.com/.

[2] Combating VR Sickness: Debunking Myths and Learning What Really Works. https://vr.arvilab.com/blog/combating-vr-sickness-debunking-myths-and-learning-what-really-works.

[3] Connect: Developing VR Experiences with the Oculus Rift. https://www.youtube.com/watch?v=addUnJpjjv4.

[4] B. A. Davis, K. Bryla, and P. A. Benton. *Oculus Rift in Action*. Manning Publications Co., 2015.

[5] Here's what happens to your body when you've been in virtual reality for too long. https://www.businessinsider.com/virtual-reality-vr-side-effects-2018-3.

[6] How a developer can reduce motion sickness in VR games. https://skarredghost.com/2016/11/28/how-a-developer-can-reduce-motion-sickness/.

[7] Kinect. http://www.xbox.com/en-US/xbox-one/accessories/kinect-for-xbox-one.

[8] Kinect for Windows v2: Sensor and Data Sources Overview. https://www.youtube.com/watch?v=qdchwqjddlw.

[9] Manual - Avatar Controller. https://ratemt.com/k2docs/AvatarController.html.

[10] Manual - Kinect Manager. https://ratemt.com/k2docs/KinectManager.html.

[11] Middle VR. http://www.middlevr.com/home.

[12] Middle VR for Unity. http://www.middlevr.com/middlevr-for-unity.

[13] T. Parisi. *Learning Virtual Reality, Developing Immersive Experiences and Applications for Desktop, Web and Mobile*. O'Reilly Media, Inc., 2015.

[14] Skarredghost. https://skarredghost.com/2017/07/13/virtual-reality-isnt-fad-will-succeed-right-time/.

[15] M. Sousa, D. Mendes, R. K. D. Anjos, D. Medeiros, A. Ferreira, A. Raposo, J. a. M. Pereira, and J. Jorge. http://doi.acm.org/10.1145/3132272.3134113. ISS '17, pages 191–200, New York, NY, USA, 2017. ACM.

[16] Synchronizing an entire skeleton. https://forum.unity.com/threads/synchronizing-an-entire-skeleton.355636/.

[17] The future of VR is social: A conversation at Engadget Experience. https://www.engadget.com/2017/11/02/vr-social-medium-engadget-experience/.

[18] Types of VR Headsets. https://www.aniwaa.com/guide/vr-ar/types-of-vr-headsets/.

[19] Unity Manual - Animator Component. https://docs.unity3d.com/Manual/class-Animator.html.

[20] Unity Manual - Multiplayer Overview. https://docs.unity3d.com/Manual/UNetOverview.html.

[21] Unity Manual - Networking HLAPI System Concepts. https://docs.unity3d.com/Manual/UNetConcepts.html.

[22] Unity Manual - Networking Identity. https://docs.unity3d.com/Manual/class-NetworkIdentity.html.

[23] Virtual Harassment: The Social Experience of 600+ Regular Virtual Reality (VR) Users. https://virtualrealitypop.com/virtual-harassment-the-social-experience-of-600-regular-virtual-reality-vr-users-23b1b4ef884e.

[24] Virtual Reality Society. http://www.vrs.org.uk/virtual-reality/what-is-virtual-reality.html.