Department of Informatics

Technical University of Munich

Master's Thesis in Informatics: Games Engineering

Collaborative VR: An Investigation of Replication and User Representation in Distributed Systems

Matthew Fairchild

Department of Informatics

Technical University of Munich

Master's Thesis in Informatics: Games Engineering

Kollaboratives VR: Untersuchung zu Replikation und Nutzer Repräsentation in verteilten Systemen

Collaborative VR: An Investigation of Replication and User Representation in Distributed Systems

Author:	Matthew Fairchild
Supervisor:	Prof. Dr. Gudrun Johanna Klinker
Advisors:	Dr. Ralf Rabätje
	Sandro Weber
Submission deadline:	February 15, 2017

I confirm this master's thesis is my own work and that I have documented all sources and materials used.

_____

Matthew Fairchild

Table of Contents

1 Prelude

1.1 Introduction

> *"Virtual reality was once the dream of science fiction. But the internet was also once a dream, and so were computers and smartphones. The future is coming […]"*

This is a quote from Mark Zuckerberg, CEO of Facebook, after announcing the company's acquisition of Oculus VR on March 25, 2014 [1]. And looking at the growing popularity of virtual reality (VR) nowadays, he seems to have been correct.
While VR has been around for decades, it has only been in the last few years that the technology has become affordable and powerful enough for VR to become a consumer product.
Sites like Statista [2] and Venturebeat [3] state that the market for virtual reality, despite still being a niche product at the moment, will continue to grow for the foreseeable future.

Due to decreasing cost of VR hardware, virtual reality is starting to be adopted by more and more end-users. And while a growing number of companies as well as individuals create content for VR, best practices – not only for technical implementation but also the design of VR applications – are not yet completely defined.
In addition, looking at industrial applications, more and more companies are starting to adopt readily available game engines to develop their applications, rather than using specialized engines like DELTAGEN. This means that they then face the same issues as individuals and independent developers when developing VR content. As such, it is of interest for all parties to investigate a number of important issues.
And while some aspects are well researched and recommended by VR makers themselves [4], other aspects are still to be defined by the developer community.
One aspect is dealing with multiple users. Collaboration is important for many types of applications, ranging from entertainment to industrial. As such, investigating the capabilities of the underlying game engines, how they scale in networked environments and which factors will influence the networked performance are of particular interest.
Another aspect is user representation. User representation is important for communication and collaboration as well as factors such as safety, when, for example, multiple users share the same physical space when immersed in a VR session. There are a number of different ways users can be represented in a virtual reality application, and all of them are used in some applications.

This is the background behind this thesis, as the next chapter will explain in more detail.

1.2 Task Description

This chapter will describe the tasks and purpose of the following work, which can be split into roughly two parts.

The first part is to investigate the replication of virtual objects over a network and the impact on the performance of important operations.
The second, more important part is to implement an avatar model using Unreal Engine 4, which can be set up and personalized by the end-user in a quick and easy manner.

The intended environment in which it will be used is within VR applications that allow widely-distributed users to join, communicate and interact with each other in a shared virtual environment.

The aim of the thesis is manifold:

1. It intends to develop knowledge of the ramifications of replication and consistency in this specific environment/engine.
2. It will collect information on the most important aspects of avatars based on scientific research in relevant areas, and derive a customizable avatar model based on that information.
3. It will provide an application that demonstrates such an avatar-creation process, accompanied by a guide for first use of that application and a detailed description of the implementation.

During the entire process, decisions made will be justified by either research and/or relevancy to practical use cases of the resulting application. Since this thesis is a collaboration of the TUM with an external company, all the above points will be addressed with both the desire for scientific soundness as well as practicality and feasibility. Striking a balance between those goals will thus be crucial.

1.3 Overview

As this paper is done in collaboration with the company vr-on GmbH, it will continue in the next chapter by giving a brief description of the company and the relevancy this thesis has for their work.
Thereafter, we will define and clarify a few terms that will be frequently used throughout this thesis. This will ease later understanding by establishing a common basis regarding the definition of important vocabulary
We will then concern ourselves with the hardware and software that is relevant for VR and the creation of this thesis. The reader will be given some background information regarding VR and what needs to be considered in the creation of VR applications, the hardware with which to experience VR as well as the different software solutions for VR creation and their benefits.
We will then proceed to the chapter regarding distributed applications in Unreal Engine. The measurements taken for the thesis will be presented as well as the conclusions drawn from it.
The chapter following that will address all topics related to the avatar. Previous work in the field will be described as well as this work's avatar model.
The paper will end with an outlook of possible future work/improvements to this work.

Furthermore, two appendices are included. One contains a more detailed description of the technical realization of the application, and the other provides the guide for first-time use of the accompanying application.

2 vr-on GmbH[1]

vr-on GmbH, a company based in Herrsching, was founded in 2016. They focus on developing software using game engines. Currently their development exclusively uses the Unreal Engine. Their products stage and Showroom focus on industrial users who wish to view early design data and make decisions based upon it. The focus is to enable users to collaborate using today's consumer hardware like the Oculus Rift or HTC Vive. In these collaborations, users can join shared virtual environments. They can view data and mark important parts of it, for example to indicate sections that need to be reworked in the design process. In order to support the communication process each user must be represented by a virtual model (avatar). In the current version of the software these are male and female upper bodies without the possibility to move extremities.

Figure 1: vr-on GmbH image for their "stage" software, ©vr-on GmbH

For the practical use of the collaboration it is crucial to know the limits and effects of an increasing number of users in a shared session, and the impact of different network speeds on the communication process.

[1] Information regarding vr-on has been gathered through their website, www.vr-on.com, and through consulting founding members of the company in person where additional information was of interest.

3 Definitions

This chapter defines important concepts and terms that are relevant for this work. Understanding this terminology will greatly ease reading in later chapters.

3.1 Terminology

There are certain terms that are used quite frequently in the context of virtual reality. As such, it is important to understand what these terms mean, or at least how they are defined within this paper. This section will list the most important vocabulary that is important for subsequent discussions of VR and the implications of external factors on the user.

3.1.1 Immersion

Immersion describes the ability of a virtual world to convince a user of the realness of the illusion it is creating. If the user believes all objects in the world including himself are present in the virtual space, then the virtual environment is immersive. [5]

Furthermore, immersion leads to another important feeing in VR, called *sense of presence*.

3.1.2 Sense of Presence

Having a sense of presence is another very important aspect of virtual reality. It is described as having a "sense of being there". [6] Slater argues that while immersion is a measurable factor, having a sense of presence is highly subjective. [7]
As such, there are several different definitions of "sense of presence" and a number of varying models have been defined which try to categorize the different aspects that lead to creating a sense of presence. [8] [5]
Overall, the most important properties can be summarized in one sentence:

To create a sense of presence, we must create an interactive virtual world that behaves as close as possible to our expectations of the physical world.

3.1.3 Awareness

A large part of human interaction is done non-verbally. Awareness is an important social queue which signals (the degree of) readiness of a potential interaction partner.
How awareness is conveyed can hereby vary. It can be posture, looking direction or hand signals, to name just a few. [9]

3.1.4 Uncanny Valley

Uncanny valley describes a state of human look-alikes (e.g. robots), in which it has achieved a close, but not perfect replication of the human aesthetic. Since human beings are naturally very versed in recognizing human features, especially facial features, being in a state of near, yet imperfect recreation of these human/facial features can create a feeling of revulsion. [10]

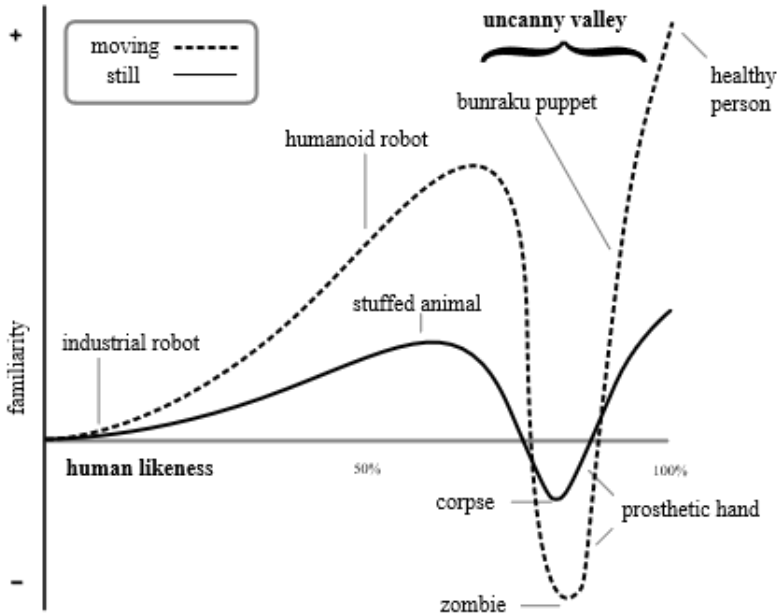


Figure 2: Expected human behavior towards objects with growing lifelikeness, ©CC BY-SA 3.0

3.2 Virtual Reality

Defining virtual reality is quite difficult, since over the years more than 60 definitions have been proposed in literature that all have their own merit.

Due to the importance of the terminology, I will nonetheless give a brief definition of the term “virtual reality” to give the interested reader a decisive description of the core element of this thesis. This definition is based upon numerous definitions provided by experts, taking elements that are common in many of them. [10] [11] [12]

Virtual Reality is a computer-generated representation of the physical world. Using special hardware, users can take part in and interact with the digital world. The user’s expectations of objects' behavior are carried over into this virtual environment, making the experience fully immersive and allowing the participants to accept the virtual world as a reality in which they are actually present.

3.3 Engine(s)

An engine can be defined slightly differently in different contexts.
In a broad sense, an engine is the backbone of an application. It is the foundation on which developers build their applications.
The primary task of an engine in general is to serve as an interface between the application and the underlying application programming interfaces (APIs) and hardware. As such, an engine is a combination of a number of specialized "sub-engines" that each bridge the gap between application and hardware in a certain submodule of the system [13].
Examples of common types of "sub-engines" are:

- Graphics
- Physics
- Sound/Audio
- Input
- Networking
- Scripting

Typically, engines also provide the developer with a user interface (UI) that eases the use of the features it provides.

Depending on which engine is used, it may implement only a subset of the just-mentioned "sub-engines". This will depend mainly on the intended target audience of the engine itself. Companies will often build their own engine for internal use only in order to ease and speed up later content creation. These engines will focus on very specific use cases and depend on the software the company creates. Other engines are made mainly for commercial use by third parties, and thus will integrate as much functionality as possible to target a broad spectrum of potential customers' use cases. Prime examples for such commercial engines that deal well with a very large spectrum of possible use cases are game engines.

4 Hardware & Software

4.1 VR

This chapter will give the reader a short introduction into the hardware and software that enable "virtual reality".
It begins by explaining some important historical factors that have made VR what it is today, before going into the most important requirements for VR applications.

4.1.1 Historical Derivation

While virtual reality as a technology has been around for many years, only in recent years has it become more and more commercially available.
By looking at the three most important aspects of VR applications, according to what

most researchers agree upon, we can see that the remarkably fast growth in computing power over the years has helped tremendously in this development.
Those three aspects are: [14]

The application must run in real-time

"Real-time" provides the reader with a lot of room for interpretation. This property is measured by the amount of frames the application can render per second, or fps for short. It is universally agreed that the more frames which can be rendered per second, the better the results for VR applications, since the application will look a lot smoother for the user. [15]
Both currently-popular PC VR headsets have a refresh rate of 90Hz [16]. And according to Michael Abrash, who worked on creating one of these headsets, matching the refresh rate with 90 fps is advisable for a good VR experience [17].

The application needs to be interactive

When the user is immersed in a virtual world, his expectations are shaped by the real world. Thus, the environment is expected to react in some way to the input of the player. While all behaviors need not be identical to the real world, interactivity is an important factor to give the user a sense of presence in the world. [18]

The application must make use of stereoscopic graphics

Since there are many different definitions of virtual reality, there is a wide gap in hardware that could be considered VR. What most researchers agree upon is that a VR application should be stereoscopic. This means rendering an individual image for each eye. The two images should be rendered from slightly offset positions, thus imitating the effect of human eyes. The brain will then have a better feeling of depth in the scene.

In the past, these factors used to be a difficult barrier to overcome in developing VR applications due to the sheer amount of computing power needed to efficiently draw a virtual world in a stereoscopic manner. In an official blog post by Oculus, one of the best-known VR headset creators, it says just the rendering requirements alone are 3 times that of normal 1080p rendering. [19]

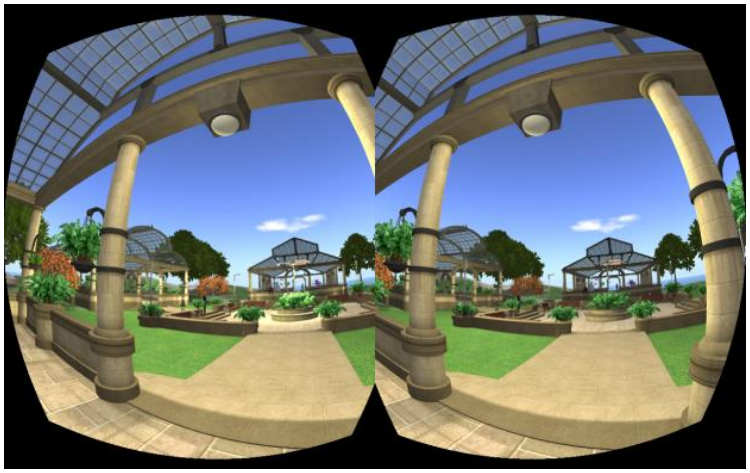


Figure 3: Stereoscopic vision of a virtual environment.

With the steady increase in computing power this is not the primary issue anymore when creating VR applications. The release of consumer-grade virtual reality headsets has transformed VR into a technology that is enjoying growing interest among companies and consumers alike.

With "real-time" being one of the key factors for VR, another area has become increasingly popular and common in VR development – the availability of commercial game engines that assist in the development of 2D as well as 3D real time applications. For a more detailed description of these engines, see chapter 4.3.

4.1.2 Application Requirements

As with any software application, there are certain requirements placed on VR applications.

First, it is important to understand that virtual reality applications bring with them a set of unique properties. These properties include a high degree of immersion and the user feeling a sense of presence in the virtual world (for a definition of these terms see chapter 3.1).
Both presence and immersion are integral and desired components of VR, and can be increased or decreased via several software engineering and hardware specific implementations. Thus, all virtual reality applications share a few requirements that assist in achieving the best possible immersive experience for the user:

4.1.2.1 Accuracy

All input and output devices need to be tracked and represented in the virtual world in order to achieve the best possible experience. The tracking technology is key here (see section 4.2.1).
The more accurate a device is tracked, the better. Losing tracking or having inaccurate tracking in VR is detrimental not only to the sense of presence of the user. It can also cause motion sickness for some people. [20]

4.1.2.2 Plausibility

When putting a person into a virtual world using virtual reality, the world does not need to be realistic for the player to be immersed. Instead, the factor that matters for the user's experience is how plausible the world appears to the user.
This plausibility may manifest itself in many different ways, but in general can be described as realistic behavior of other objects and actors in the virtual environment. [21]

4.1.2.3 Representation

To reach a perfect sense of immersion and presence, it would be desirable to place the player into an avatar that is visually a perfect copy of the player, and which mimics the movements of the player exactly. On the other end of the spectrum is no representation

at all, with the player not seeing where his hands are in the virtual world and having no virtual body at all. [21]

4.2 Devices

When it comes to virtual reality, the devices being used are integral to the experience the user will have in the virtual environment. There is a wide range of possible input/output (I/O) devices for VR depending on various factors such as hardware setup, compatibility, space and cost.

The devices used can have a direct impact on the immersion of the user, and thus on the sense of presence felt. This, in turn, means that availability and usage of certain preferred input and output devices has a direct impact on the quality of any virtual reality application.

This section will describe some of the devices used for VR in general, before going into the target devices relevant for the project associated with this work.

4.2.1 Tracking Technology

Before going into the different types of I/O devices, it is crucial to know that although the type of device is important, it alone does not suffice to judge the quality of the user experience. An important factor for the quality of presence in VR is the accuracy with which player movement, no matter how subtle, can be mimicked in the virtual environment. This is determined by the tracking technology used. The tracking system is used to keep track of position and orientation of the hardware (i.e. controllers).

In the following I will describe some common methods of tracking.

4.2.1.1 Mechanical Tracking

Mechanical tracking is a system in which the target, i.e. the device being tracked, is physically linked to a known reference point through any number of joints. The angles between any two connected joints are known, thus allowing the system to track the target very accurately. [22]

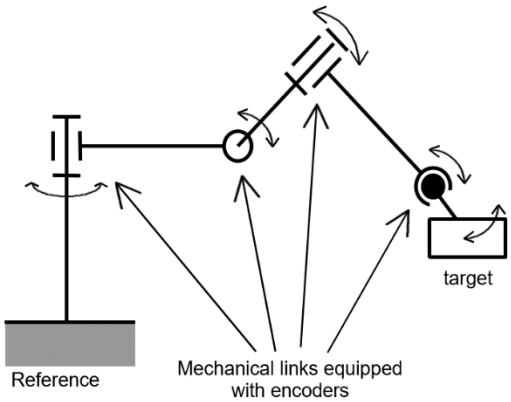


Figure 4: Mechanical tracking setup, ©J.P.Rolland et.Al.

4.2.1.2 Magnetic Tracking

A magnetic tracking system consists of a transmitter and a receiver. The transmitter is attached to the object that needs to be tracked, and consists of 3 coiled wires which create a low-frequency magnetic field. This magnetic field is detected by the receiver, which can then derive the transmitter's position and orientation relative to the receiver. [23]

4.2.1.3 Acoustic Tracking

The transmitter used in acoustic tracking has three speakers on it, which emit ultrasonic sound. The receiver, on the other hand, has three microphones, with which it can detect the ultrasonic sound emitted by the transmitter. Since the speed of sound is constant (at constant temperatures) one can calculate the time of flight between a speaker and the 3 microphones. Triangulation makes it possible to deduce the position of the transmitter using this technique. To detect orientation, additional hardware, such as gyroscopic compasses, are often used in practice. [24]

4.2.1.4 Optical Tracking

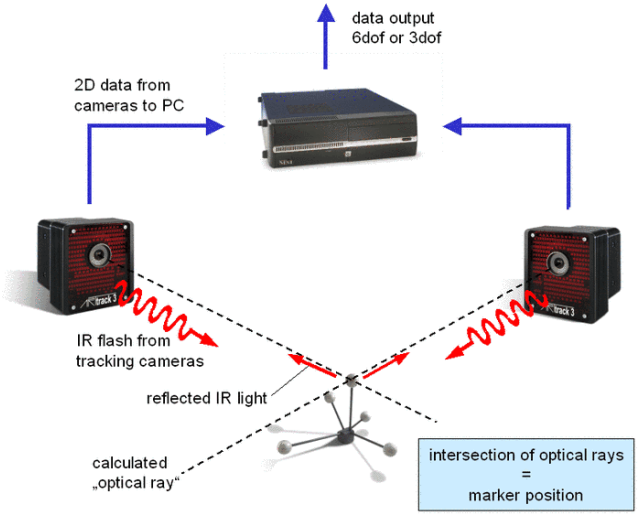


Figure 5: ART's optical tracking setup, ©Advanced Realtime Tracking GmbH'

Optical tracking is the most versatile tracking system. There are several different approaches for achieving optical tracking. Their common denominator is the fact that they use cameras to track a target by using image processing algorithms on the resulting frames. Using a known reference point the target's position and location can then be derived from the resulting calculations.

An important differentiation to make is whether the device uses inside-out or outside-in tracking. [25]

Outside-in tracking: Here, the target is equipped with markers that allow it to be tracked by an external receiver which functions as the reference point in the system. For example, the ART tracking setup uses outside-in principles. The object being tracked is prepared with trackable features, and the surrounding cameras can then recognize these markers and derive the targets position relative to the camera (see figure 5).
Inside-out tracking: Inside-out tracking has the mobile target itself equipped with the sensors needed to derive position and orientation. Thus, the target itself becomes the reference point in the system.

4.2.2 Output Devices

The degree to which the user feels a sense of presence is in part determined by the output device.
While it is not always possible to freely choose the output device used due to various possible constraints such as cost or required compatible input devices, it is important to know which hardware is available. Only then can a developer make an informed decision in the planning stage of a VR application.
The most common output devices for virtual reality applications are as follows:

4.2.2.1 Workbench

Just barely related to virtual environments, as some researchers say, is the so-called workbench. Created in the 90s by engineers from the U.S. Navy, it is essentially a large display monitor that can be viewed vertically or horizontally. Users wear special glasses to create a three-dimensional effect, making it seem as though objects are placed on an actual table. [26] For better understanding, refer to figure 6.

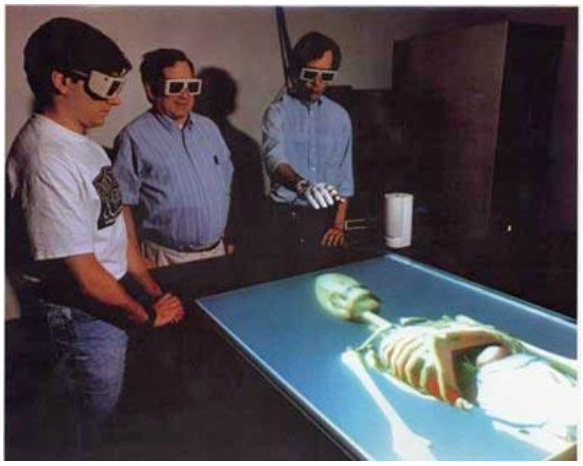


Figure 6: 3 users on VR workbench, ©US Naval Research Laboratory

4.2.2.2 CAVE

The acronym CAVE stands for cave automated virtual environment. A CAVE is essentially a small room in which a number of surrounding walls (including the ceiling and the floor), have projectors behind them displaying a synchronized environment onto the user's surroundings. The number of walls (including the ceiling and the floor) can vary from one CAVE installation to another, whereby the more walls are back-projected, the better. Each wall will have projectors displaying the image onto it. It can be a single projector per screen, or multiple projectors which project onto the wall with a slight offset, thus allowing the user to see fully three-dimensional projections inside the CAVE area via the use of special goggles. [27]

Figure 7: Image taken of the DLR CAVE at the GSOC in Oberpfaffenhofen

Depending on the available resources, CAVE installations can offer a large number of additional cues to increase immersion. Positional audio and head-tracking via previously mentioned ART tracking of the user's goggles are among the most common additions.

4.2.2.3 Head Mounted Displays

Head mounted displays (HMDs) have become the most well-known technology for displaying virtual reality. Consumer-grade hardware such as the Oculus Rift, HTC Vive, GearVR and more recently Google's Daydream technology all use these HMDs.
HMDs have displays positioned in front of 2 lenses, one for each eye. The devices obstruct the users view of the real world, thus engrossing him entirely in the virtual environment. When doing this there are a number of properties that determine the quality of HMDs and thus directly impact the immersion of the user. [28] These factors are typically:

- Weight distribution
- Field of view
- Resolution
- Update frequency

Figure 8: Left to right: PSVR, Oculus Rift, HTC Vive

4.2.2.4 Miscellaneous Output Devices

While the aforementioned output devices are arguably the most common for most scenarios, there are many more devices that can be used in specific scenarios, some of which target senses other than visual.
For example, there are haptic feedback devices which give the user a sense of touch, or olfactory displays which emit a scent. [29]

However, I will not go into more detail regarding these types of output devices due to the arguably scarce usage of these specialized and more exotic output devices and because they are not within the scope of this thesis.

4.2.3 Input Devices

While output devices are of course necessary to give the user feedback about the state of the virtual world, the input devices used are equally important. They play an important role for the immersion and sense of presence an application can achieve.

A general rule for input devices in VR is:
The closer an action performed in VR is to its real life equivalent, the more immersive the experience will be.

As is the case with output devices, there is a vast number of different input devices for numerous different scenarios, hardware setups and use cases. As such, not all possible input devices will be mentioned, but rather just a few notable examples and particularly relevant ones.

4.2.3.1 Data glove

Figure 9: VMG Lite Data Glove, ©Virtual Realities LLC

Data gloves come in different variations. They can either use optical properties to measure the curvature of each finger, or measure strain on the fingers to recognize finger positons.
They are intuitive to use due to their close resemblance to real world interactions via gestures. Through tracking of each finger, navigating the virtual hand and 3D manipulation in the virtual environment become natural.

4.2.3.2 Wands

Simply put, a wand is a controller that is tracked in 3D space. Similar to conventional game controllers it will have buttons and/or other inputs on it to allow the user to perform certain actions. This is necessary because while it is tracked, the player needs to hold the wand in his hand, making interaction unnatural and thus requiring some form of metaphor to perform common actions such as picking up or manipulating objects in the virtual environment.
An example of such wands would be the controllers of the HTC Vive (see figure 10).

4.2.4 VR Hardware Used

After getting to know the theory behind a number of tracking technologies, this chapter will briefly describe the VR hardware that will be used in this thesis.
Knowing the hardware that will be used is important for development, since not all the current VR hardware is shipped with the same amount of input devices, or give the user the same-sized play area.

This thesis has been developed with an HTC Vive.

Figure 10: HTC Vive hardware

The Vive uses an optical tracking technology they call "lighthouse". [30] This technology allows for stationary as well as room-scale VR, meaning you can either remain seated at your desk or move throughout a pre-calibrated space while enjoying VR experiences. [31] Its standard configuration sold to consumers comes with 3 devices, an HMD and two motion controllers. All 3 devices have their position and orientation tracked as long as the user is inside the pre-calibrated play area.
The motion controllers are equipped with a trackpad alongside numerous buttons to provide additional input, while the HMD can provide further input through a front-facing camera and a built-in microphone. [31]
The HMD is also the output device. Its screen has a resolution of 2160x1200 at a refresh rate of 90Hz. The headset in total has a field of view of 110 degrees. [16]

Which hardware is used is currently quite important, since it defines the parameters the user can work with while developing, i.e. what types of input are available. Developing for the HTC Vive guarantees that its owner will have all the necessary input devices since all the hardware is sold together.

4.3 Engines

In this chapter we will review the different engines on the market. More precisely, we will compare VR-capable engines that over the years have become well established in industrial usage to the commercial game engines that are becoming increasingly popular for industrial applications.

While there was previously a clearer distinction between engines used for large scale industrial purposes and entertainment software such as games, modern game engines are now becoming increasingly popular in industrial application development.
With that in mind, we will look at the typical requirements of industrial vs. entertainment apps. We will hereby focus on VR applications in both fields.

4.3.1 Industrial vs Entertainment Engines

Any VR application is faced with a certain set of base requirements. See section 4.1.2. But in addition to these prerequisites, there are additional requirements placed on the

software depending on what it will be used for.
Some may be focused on component interaction, requiring a physics engine. Others might be focused on producing realistic visuals, making the render engine most important and therefore eliminating the need for a physics engine. And some might want to place the player in an interesting world with which he/she can interact, thus needing a wide spectrum of different engines to realize all the requirements.
In the realm of VR applications there is a glaring distinction to be made between industrial applications and entertainment applications. We will therefore take a look at some well-established engines that specialize in industrial-use cases and compare them to game engines commonly used for entertainment applications.

Industrial

Note that the engines listed below are not all of the available engines. These 3 examples have been chosen because they are arguably the most prominent ones for VR usage.[2]

1. DELTAGEN [32]
    a. High visual fidelity as main focus of the engine
    b. Real-time interaction mode in rasterized graphics mode
    c. CAD importer
    d. Built in ergonomics tests (via established RAMSIS model)
    e. Full global illumination (using a cluster)

    DELTAGEN is an engine by 3DExcite. Its main focus is graphical fidelity, but it needs a cluster to achieve the best possible visuals. Its strong focus on realistic visuals combined with a built-in system to check the ergonomics of models make this engine a highly-specialized tool for product development and presentations.

    While it has a powerful rendering engine, it is lacking in other areas. There can only be one single user in the virtual environment at any point in time. While it is possible to use the engine for real-time interaction, this can only be achieved when using rasterization, which will decrease image quality significantly compared to its other rendering modes. Furthermore, collision detection is a big problem, excluding many possible use cases.

2. VRED [33]
    a. Main focus on 3D visualization
    b. Built in animation editor
    c. Full GI (using cluster)
    d. Geometry editor
    e. CAD importer

    The VRED engine from Autodesk is similar to DELTAGEN in many ways. It also provides a CAD importer, full global illumination and is focused on rendering

[2] These 3 were chosen in collaboration with this work's supervisor, who has many years of experience with VR in industrial use cases. As such, his input was the deciding factor for assessing the available industrial engines, since there is no sales or scientific data to compare the success of all the available engines.

realistic images. In that sense they serve the same niche. The differences become apparent when looking at the details. VRED comes with a built-in animation editor, for example, so adding/editing animations to existing objects becomes easier. Also, Autodesk provides the user with the possibility to edit geometry in the engine, place annotations and analyze the curvatures of models. According to the VRED website, they target industrial designers and the automotive industry.

And while Autodesk has added some special functionality for that specific target audience, VRED is primarily focused on realistic visualizations. So, similar disadvantages as with DELTAGEN apply: It is a single-user offline experience with very little functionality in most other areas such as audio or physics (if at all).

3. IC.IDO [34]
    a. "Balanced solution"
    b. Real-time user interaction
    c. Out-of-core renderer

    IC.IDO from ESI Group markets itself as a "balanced solution". Unlike the two previous engines IC.IDO implements a large number of submodules to deal with a broad range of use cases. For example, it has a robust physics engine that can simulate dynamic moving components. And unlike the previous engines, it allows for real-time user interaction without sacrificing quality in other areas.
    But while the overall image quality is lower than in VRED or DELTAGEN, it does also try to differentiate itself from the others in terms of rendering by using an out-of-core renderer, which allows for seamless visualization of data that exceed primary memory in size.

    A remaining downside even with this engine, however, is that it is a single-user offline experience.

All three of these engines are well-established in industrial applications. They have a specific target audience with certain needs and requirements, and have specialized their product to cater precisely to those customers.

In contrast, we have game engines. To be able to make a distinction between industrial engines and commercial, often freely available game engines, we will now first look at the feature set of such a game engine.
Note that there are a few different game engines available, but for the sake of this comparison we will generalize and refer to only "game-engines" as a whole, since the overall relevant feature set is basically identical. The small differences between the available game engines lie in the details, and will be relevant later.

Once we have established the differences, we can then justify the choice of engine used in this thesis by looking at the requirements of the project.

Entertainment

Typical components of game engines include:

a. Renderer
b. Audio engine
c. Networking
d. Input handler
e. Scripting engine
f. Animation engine
g. Artificial intelligence module (mostly using state machines and/or behavior trees)
h. Physics engine
i. User interface creation tools

As becomes apparent by the sheer number of components available in a modern game engine, it is well suited for a wide variety of possible use cases. The large variety in entertainment content produced with these game engines require them to be a very general tool for a large number of different tasks.
While the quality of modern game engines is very high, the fact that they balance their resources among all of these components means, in exchange, that they cannot reach the quality in any one specific area that an engine specialized on that area will be able to achieve.
The primary advantage of game engines is thus their capability to deal with many use cases adequately. Disadvantages of game engines arise in projects that require a very high quality or a unique feature in any one single component.

Example: A game engine will produce less realistic images than DELTAGEN, so if a project's primary need is realistic visuals, then it should use DELTAGEN. If, on the other hand, we want to create an application with networking capabilities to bring multiple users into one virtual environment, then we would need to use a game engine due to the lack of networking capabilities of the industrial engines.

To decide which type of engine to use, it is necessary to compare their features to the requirements of the project associated with this thesis.

4.3.2 Choice of Engine

This section provides a brief overview of the overall project in order to deduce the requirements it places on an engine.
By doing so, we can justify the type of engine being used for the development of the current application.

Requirements:

1. It needs to support the current commercial HMDs and their peripheral devices (i.e. motion controllers)
2. It needs to support a mechanism for dynamically editing mesh rigs
3. It needs to support multi-user interaction in LAN & WAN
4. It needs to have a system in place to create a UI for both desktop and VR usage
    5. Ideally, it should have an inverse kinematics solver implemented to allow puppeteering of meshes
    6. It needs to have an animation engine to import/create/edit animations of meshes
    7. Visual fidelity of the scene needs to be high

One requirement in particular comes to mind when choosing which engine type to use here, namely the fact that the project should be a multi-user application that will run over the Internet or a local network. As seen before, all the well-established industrial engines do not support multiple users, so the decision immediately becomes choosing one of the available commercial game engines.

The choice fell to Unreal Engine 4.

Figure 11: Unreal Engine Logo

While most game engines have a similar set of features, there are some distinct differences between commercial game engines. These differences will guide the decision on which one to use for any given project or company. While all the major game engines would meet the requirements of the project mentioned above, the main reasons why Unreal Engine 4 was chosen over other widespread engines like Unity are the following:

1. C++
   Any game engine that allows for scripting needs to choose which programming languages to use for its scripts. Unreal Engine 4 scripting is done in C++, a very common and well-established programming language that allows for very low level optimization. This optimization is critical for VR applications to run as best as possible.

2. Blueprints
   In addition to C++ scripts, UE4 has implemented a visual scripting system it calls "blueprints". These blueprints make it possible to define the behavior of objects without the need to know programming. Besides that, it enables quick and easy prototyping of new and/or experimental features faster than may be possible though writing scripts.
3. Open source
   The engine is completely open source. This is helpful in several ways. First, it allows the developer to adjust and tweak the core functionality of the engine if needed. The other, more common advantage, is the ability to use the engine source code in addition to one's own code when debugging unexpected behavior.
4. Visual fidelity
   While this point is subjective in nature and thus debatable, it has up to this point often been perceived as being by default the commercial engine with the highest visual fidelity. This is a very subjective matter, however, and over time the technologies of all engines have improved, and continue to improve, making a comparison very difficult.
   Regardless of any comparisons to other engines, though, the achievable quality of the visuals was another factor that was important in choosing Unreal Engine.

5 Distributed Application

Using Unreal Engine 4 gives the developer a lot of built-in functionality. One of these features is built-in replication of objects to ensure consistency among everyone's virtual environment when participating in a multi-user VR session.

While the main work of this project focuses on single-user functionality, the resulting avatar is meant for use in a multi-user environment. As such, this part of the thesis will concern itself with some properties and the performance of Unreal Engine 4's network capabilities.

5.1 Object of Investigation

Multi-user applications are a core capability of interactive software. Their intrinsic properties bring with them some unique challenges though. The biggest issues are dealing with unreliable networks and ensuring the consistency of the virtual world among all users.
Networking properties become even more important when these issues are combined with the problems VR applications introduce, such as hardware and performance requirements. Network issues and high performance requirements might escalate. Investigating the impact of specific operations in a networked environment under varying circumstances is thus an important undertaking.
As explained earlier (see chapter 2), vr-on GmbH creates multi-user VR applications. When it comes to the distributed nature of multi-user applications, there are some fundamental questions that are of interest to the user and the developer alike.

The different users in a shared session might be located in different buildings, cities or even countries. As such, we need to expect that the quality of the different user's connection will vary. Furthermore, the number of connected users might in itself affect the performance of every connected client. A number of questions arise from these assumptions, and this chapter will address them.

This part of the thesis relates to the performance of networking capabilities inherent to Unreal Engine 4. Through a specially customized level, the time it takes for a few distinct operations to be communicated and distributed over the network will be measured.
Analyzing the results enables us to conclude how certain operations behave in certain types of network conditions, how they scale with a growing number of users, and if we can expect a hard threshold in the number of clients possible in any given session as a result.

One important thing to keep in mind for this entire chapter is the conditions in which the measurements were taken. They were conducted on the intranet of vr-on GmbH in Herrsching. During the measuring process, other employees were also connected to the same network and using it for their work.
Though the measurements were always taken on the same machines, the computers that were connected to the game sessions varied greatly in computing power, ranging from high-end desktop PCs to lower-powered laptops.
Scientifically, these might not be ideal conditions. However, this can be considered a more realistic use case, since there are no hardware specifications (other than minimum requirements) that need to be met specifically to run the application. Thus, a wide range of hardware in shared networks is to be expected.
Furthermore, since this is not the main focus of the thesis, there is no claim that the ideal methodology or setup has been achieved. The focus of this section is mainly towards the scalability, and thus the relative changes in speed, as opposed to absolute measurements.

5.2 Methodology

In this chapter we will list and explain the concepts and methodologies of how the measurements we are interested in were obtained.

Due to the fact that all of the relevant logic regarding measuring network latency was done in C++, we will first explain the basic concept and most important properties of how UE4 allows for networking functionality via code (as opposed to blueprints).

5.2.1 UE4 RPC Functions

Two things are most prominent regarding UE4 networking. The first is the replication of objects, which can easily be turned on through a simple checkbox on any object blueprint. The second is RPC functions, which are defined in C++ code. These RPCs, or

"remote procedure calls", allow a user to execute a function call on a remote machine. [35]

When defining an RPC function, the developer can specify a number of different properties adherent to the function. We are most interested in (and will later make use of) the following properties that can be defined:

1. On which machine should the RPC function be called?
   A mandatory property to define is the machine which will execute the function. This can either be "Client", "Server" or "NetMulticast". Adding any of these keywords into the UFUNCTION() declaration of the function is what actually makes the function an RPC.
   a. *Client*: A function with this keyword will be called on the server and executed on the client.

```
UFUNCTION(Reliable, Client)
void Client_StartTDTTest();
```

Figure 12 (reliable) client RPC function

   b. *Server*: Declaring a function with this keyword means the function will be called from a client machine and will be executed on the server.

```
UFUNCTION(Reliable, Server, WithValidation)
void Server_StartTDTTest();
```

Figure 13: (reliable) server RPC function

   c. *NetMulticast*: A *NetMulticast* function will be called from the server, and executed on all connected clients as well as on the server itself.

```
UFUNCTION(NetMulticast, Unreliable)
void Multicast_SpawnParticleEffect(AActor* start, AActor* end, int newID);
```

Figure 14 multicast function that will be executed on all connected machines & server

2. Should the function call be declared "reliable"?
   Declaring that a function is reliable guarantees its execution. While a function that is not set to be reliable may not actually be executed due to networking errors such as packet loss, a reliable function will have its network packets resent if any errors occur during transmission.[3]
   By default an RPC function will be unreliable. To make an RPC reliable, the *Reliable* keyword needs to be added to the UFUNCTION() declaration.
3. Replicate Manually [36]
   UE4 will not automatically replicate everything in the world. And when it does not, the developer can use the feature called *ReplicateUsing*. This is a keyword that can be added to a variables UPROPERTY() declaration. When adding this keyword, you need to assign a valid function of the same class to it.

```
UPROPERTY(ReplicatedUsing = UpdateMaterial)
int currentMat;
```

Figure 15: Variable with a function assigned to it via the "ReplicatedUsing" property

[3] According to an Epic employee on the official Unreal Engine forum. https://answers.unrealengine.com/questions/395137/reliable-vs-unreliable-rpc-performance-and-orderin.html

What this does is call the assigned function anytime the variable has changed. That way, one can use any variable that is defined as replicated to trigger some additional functionality that needs to be executed manually on every machine.

All three of these properties will be used to implement the timestamping.

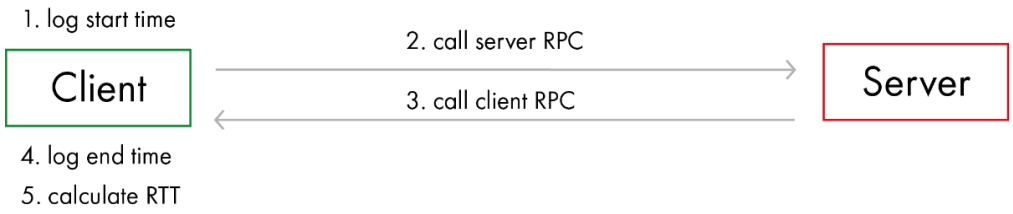
The following chapters will now explain the types of measurements that were conducted. The associated images will be based on the algorithms of the accompanying project's C++ implementation for the respective measurement.

5.2.2 Round Trip Time (RTT)

The round-trip time is a measure of how long it takes for one machine to send information over the network to another machine, and receive the acknowledgment of its arrival [37]. It is a comparatively simple measurement, and the corresponding implementation in this project is straight forward.

The process is initiated from a client, and is composed of the following steps:

1. Save the current time
2. Call a server RPC function. This function is executed on the server and is thus the first transmission that takes place over the network.
3. The server RPC just executed calls, in turn, a client RPC. Thus, it will immediately send back a message over the network to execute a function on the client.
4. This client function then saves the current time again in a second date variable.
5. Using the timestamps from the start and end of the process, we calculate the overall time that has elapsed, resulting in our RTT.



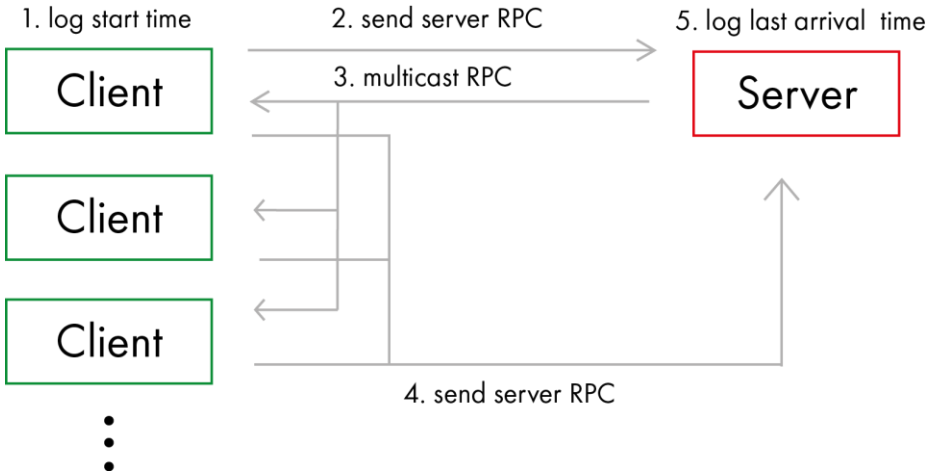
5.2.3 Total Dispatch Time (TDT)

Total dispatch time describes the time it takes for every client to answer the server after a broadcast (requesting an answer) has been sent by the server to every client in the network. [8]
To measure this, we employ an approach similar to the one used for the RTT.

Again, the client initiates the following consecutive steps:

1. Save the time at which the procedure is being initiated.
2. The client calls a server RPC function.

3. In that function, the server then executes a multicast function, which is executed on all machines that are connected to the current session.
4. On the clients, the multicast function again executes a server RPC function.
5. The server knows the number of clients connected. Therefore, the server can count the number of resulting answers it receives. As soon as the number of answers equals the number of connected clients, the server logs the time.



The server then transmits the timestamp back to the client, which can then calculate the time that elapsed from start to finish.

5.2.4 Spawn Time

Another measure we want to take is the time required to spawn an object. Spawning an object must always be done on the server. Therefore, to spawn an object from a client, we must send a request to the server, which then instantiates a new object into the virtual world. The object itself is replicated to all clients automatically through Unreal Engine's built-in networking functionality. We are interested in how long it takes from inquiring a new object to spawn until the new object is replicated to the requesting client.

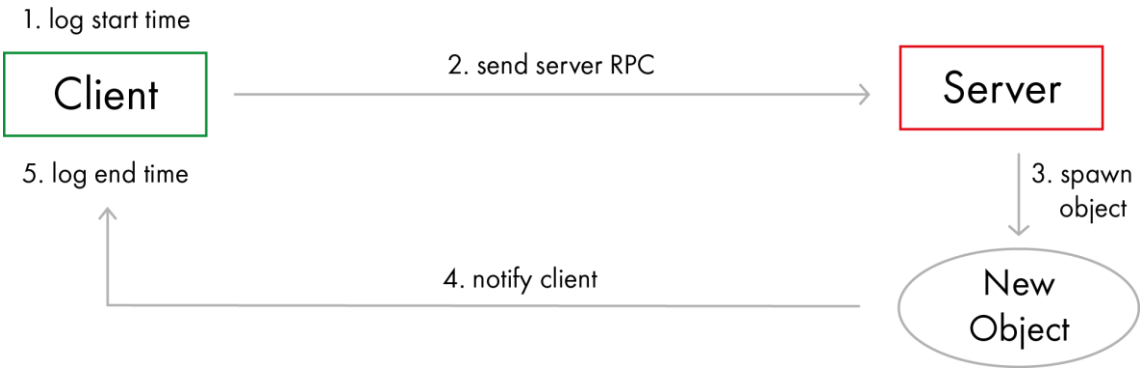
To measure the time, we need to know some essential prior setup:

Prerequisites: For measuring the time it takes to spawn an object, we must stop the measurement when the object is present on the client machine.
To do this, we use UE4's "ReplicateUsing" property. This is a property that can be assigned to a variable. So we declare a variable of the type of object we want to create, and set it to replicate.

Again, the process is initiated by a client, and the entire procedure consists of the following steps:

1. Save the starting time.
2. Execute server RPC function.
3. Server then instantiates the new object and sets the previously declared variable to the resulting reference.

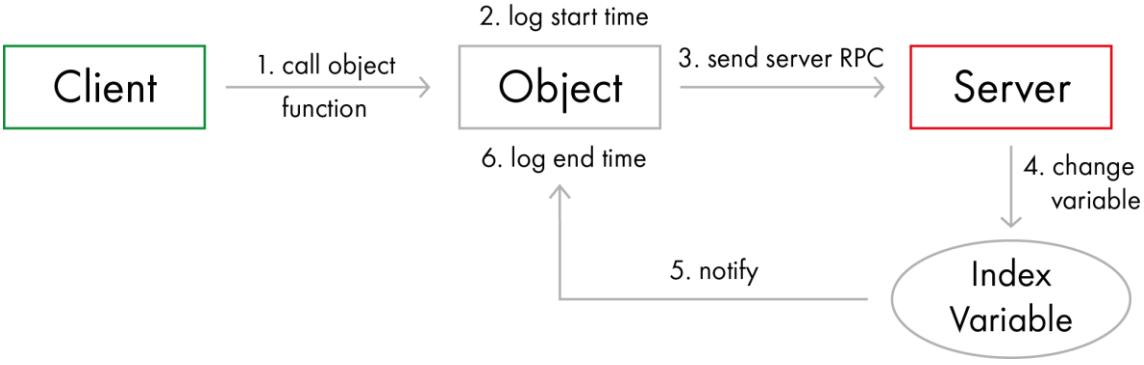
4. In turn, since the variable has changed, the "ReplicateUsing" function we have defined is executed.
5. In that function, we log the time and calculate the time elapsed from start to finish.



5.2.5 Variable Change Time

The final measure we will take is the time that elapses to change a single property of an object. In our case, we will change the material of one of the objects we have previously spawned.
To do this we must proceed similarly to the spawning procedure. The materials are stored in a vector, and the index of the currently used material is saved in a separate variable.
That index variable has a function assigned to it via the "ReplicateUsing" property. The procedure is then as follows:

1. The client obtains a pointer to one of the objects we spawned, and calls a member function of that object's class.
2. That object then saves the current time.
3. The object then initiates the actual measurement process analog to the spawning by executing a server RPC function.
4. The server function then changes the desired variable. This variable will have a function assigned to it via the "ReplicateUsing" property.
5. Upon changing the variable, this function is executed on the clients.
6. This function logs the time again and calculates how much time has elapsed during the procedure.



5.2.6 Measurements

Several measurements were taken. Every procedure was conducted multiple times, each time with a different condition inherent to the network. The different network conditions were:

- Normal network conditions
- 1 client in the network has an unusually large chance of dropping network packages
    - Measurements taken from the affected client
    - Measurements taken from a different client in the network that has no issues
- 1 client in the network lags (has an unusually large delay when receiving network packages)
    - Measurements taken from the lagging client
    - Measurements taken from a different client in the network that has no issues
- 1 client in the network has an unusually high delay and chance of dropping network packages
    - Measurements taken from that one slow client
    - Measurements taken from a different client in the network that has no issues
- All machines in the network have an unusually high chance of dropping network packages
- All machines in the network receive their network packages delayed
- All machines in the network have a delay as well as a higher chance of dropping network packages

So, 10 different sets of measurements were taken in total.

Furthermore, to compare the scalability of all the factors we are interested in, every test procedure was taken numerous times, with a continuously growing number of connected clients. Starting from 1 client up to a maximum of 5 clients in the same session, all procedures (RTT, TDT, spawning, variable change) have been tested with every network condition mentioned above.

Since all measurements were taken in the company's intranet, the conditions described above need to be created artificially.
To do this a software named "Clumsy Net Limiter" was used. [38] In contrast to many other methods of controlling network behavior, Clumsy Net Limiter is a relatively simple solution, but with fewer options than competitors. Its options suffice for our intended use because our requirements regarding modifications of network traffic were limited.

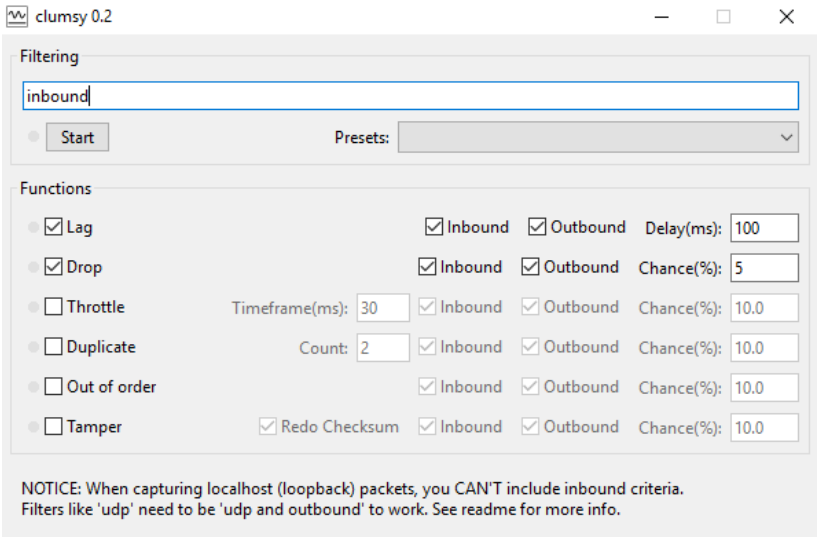


Figure 16: Screenshot of Clumsy interface

See figure 16 for an image of the interface of Clumsy, with the parameters set for our use cases. On it, we can see the following two properties we wish to toggle on/off according to which condition we wish to measure:

- Filtering defined for the inbound traffic
- Lag: Creates package delay. Here, we have set it to 100 ms
- Drop: Defines the drop chance of each packet. Set to 5%.

Now that we have outlined which measures were captured along with how it was done, we can take a look at the results.

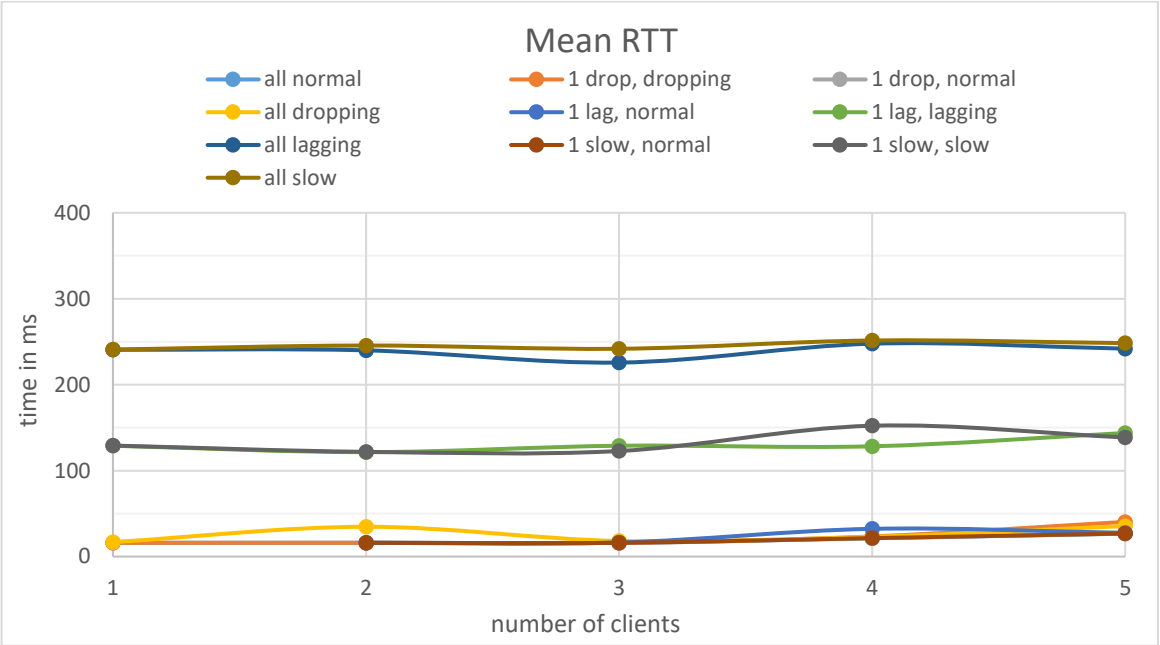
5.3 Results

In this chapter we will present the results of all the measurements taken. Here we will focus exclusively on the results and observations, leaving the inferred knowledge and conclusions for the next chapter.

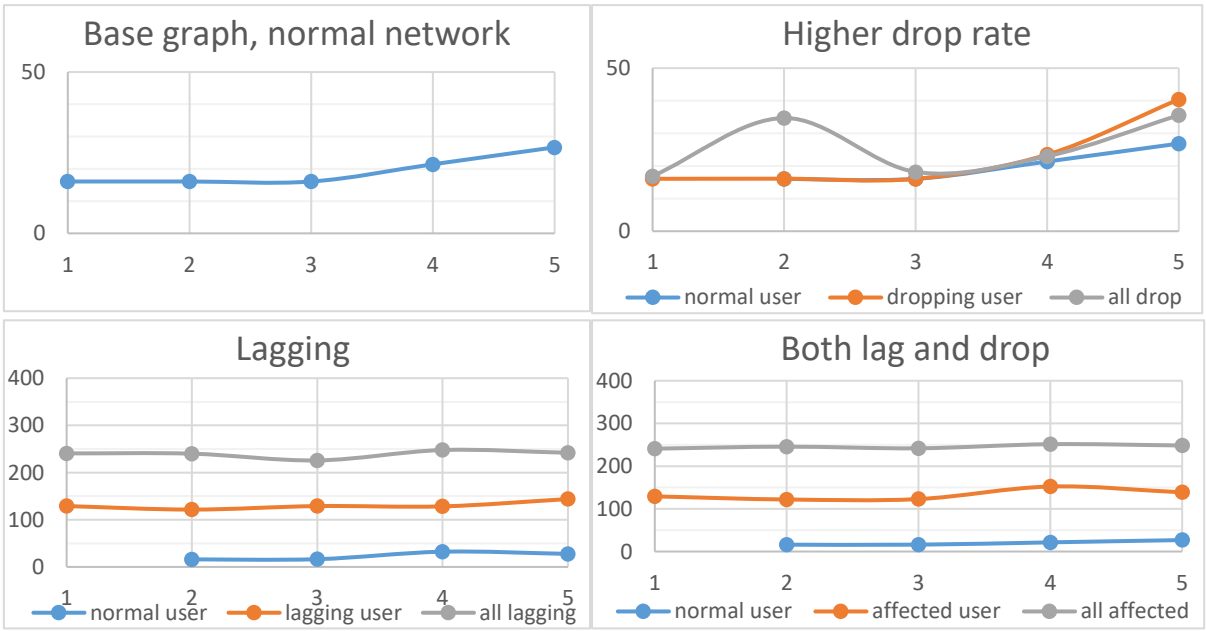
Due to the lack of space and ease of understanding, not every single measure taken will be listed. Rather, they will be grouped according to task and visualized as graphs of the mean time (in ms) and number of connected clients. Every network condition will be represented through a differently colored graph, thus enabling easy visual comparisons.

Note: Some graphs are denoted with labels such as “normal user” or “affected user”. These are measurements conducted in networks in which only a single user experiences some kind of issue. “Normal user” is the graph that shows the measurements taken from a user which has no network problems, in a session in which one user is experiencing problems. “Affected user” illustrates the measurements taken at the one user of the session that has network problems.

5.3.1 RTT Results



As can be seen, 3 tendencies seem to form. All three run roughly parallel to the x axis, which would indicate no immediate correlation with the number of clients is visible. But to better make sense of different networking conditions, we will separate the graph into 4 separate ones depicting different networking environments.

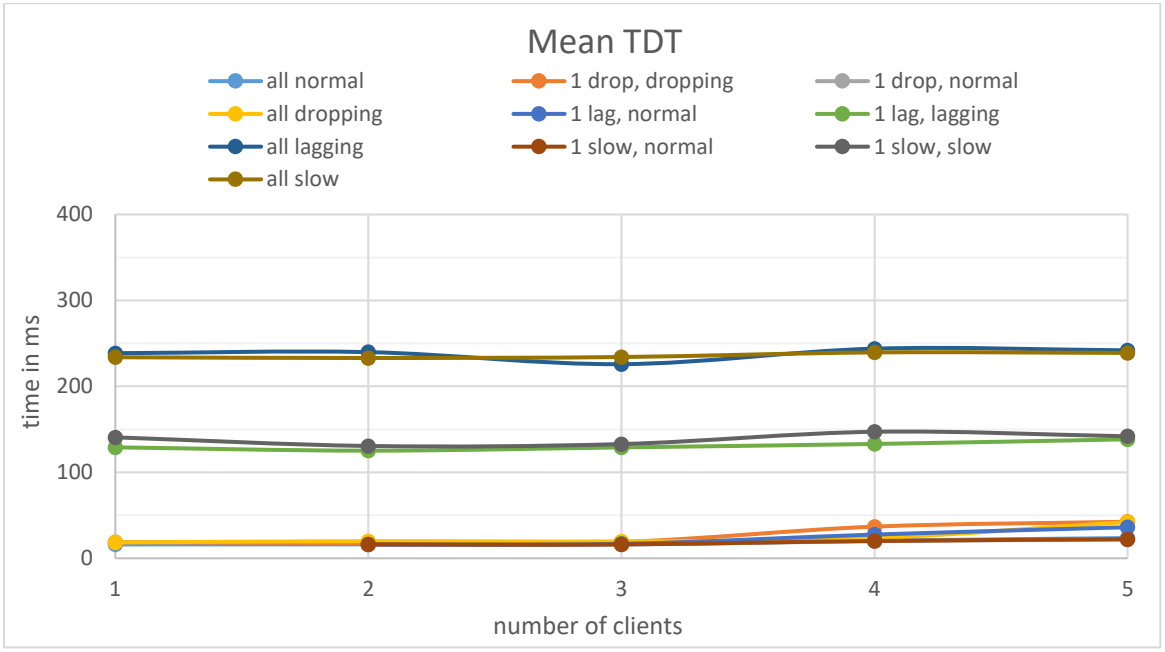


Notable:

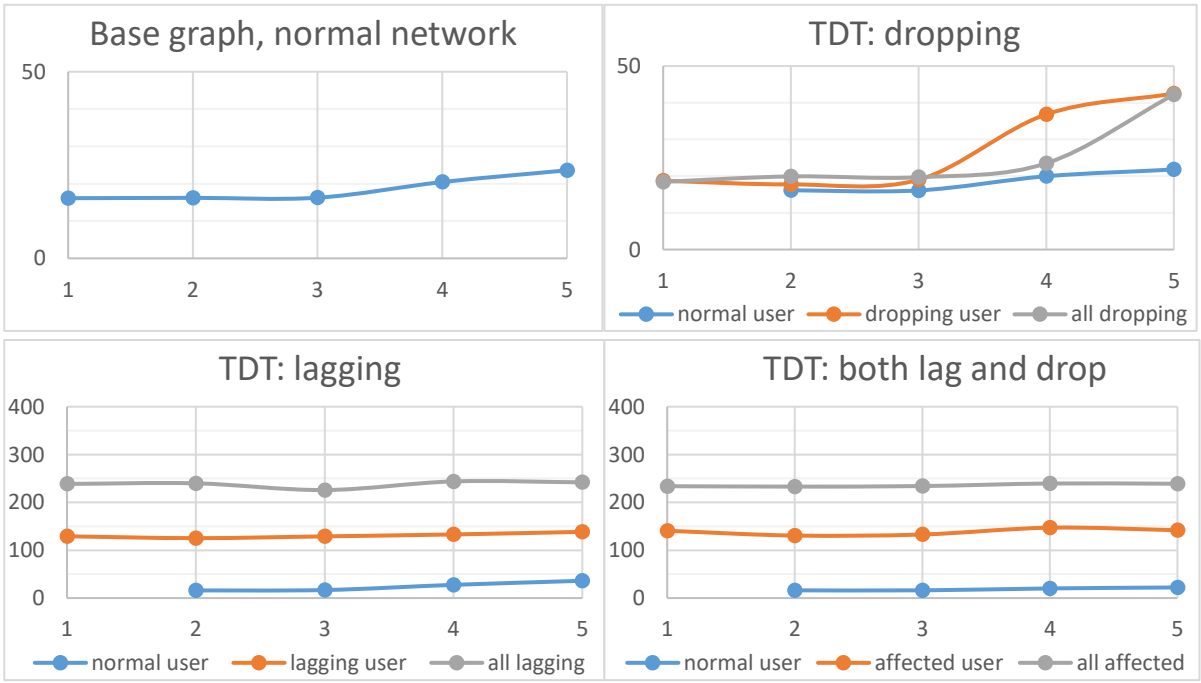
- The normal network shows an increase in measured time when more than 3 users are connected. While the absolute increase may seem small, the relative increase is notable.
- With only a higher chance to drop network packets, there is an increase at 2 clients, but otherwise all 3 measures are nearly identical.
- The graph for a lagging network and the one for lag as well as a higher packet drop chance look virtually the same, and show the following characteristics:

- The three different measures all remain a horizontal line.
- The mean times in both graphs for the same measurement (i.e. “normal user”, “affected user”, “all affected”) remain nearly at the same absolute values.
- There is a very clear order: “normal user” always has the lowest measured times, “affected user” measures are a bit higher and “all affected” results in the highest measurements.

5.3.2 TDT Results



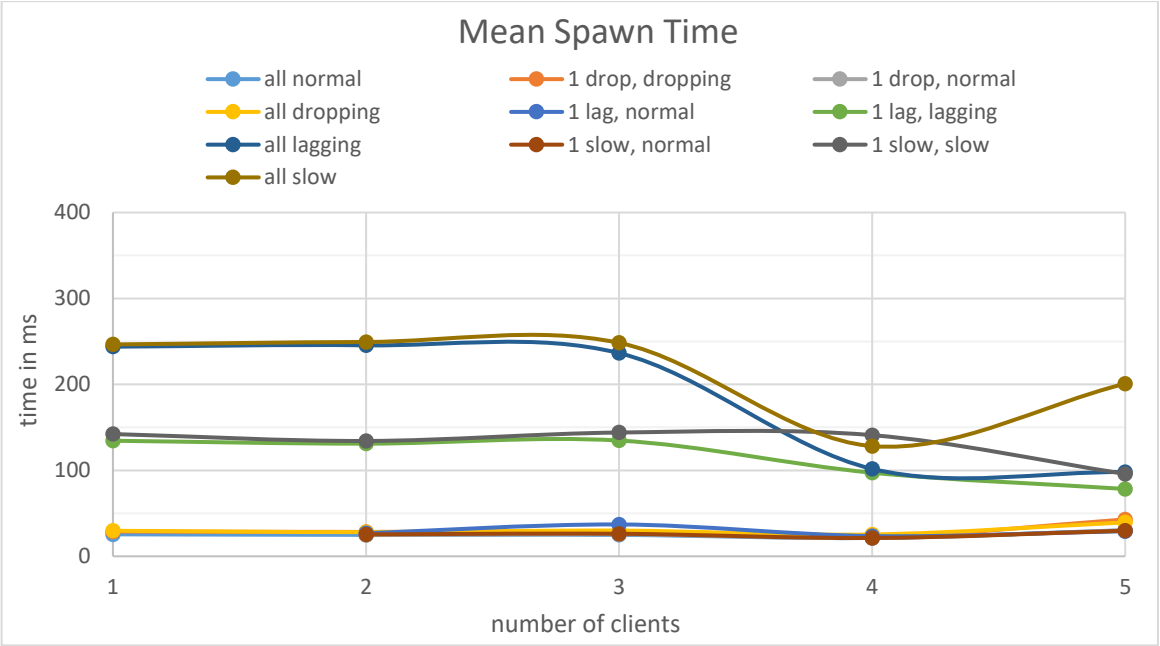
As is immediately apparent, the graph showing all measurements taken for total dispatch time looks very similar to the one visualizing the RTT values. This graph indicates that even the approximate values around which the three horizontal tendencies seem to hover are roughly the same.



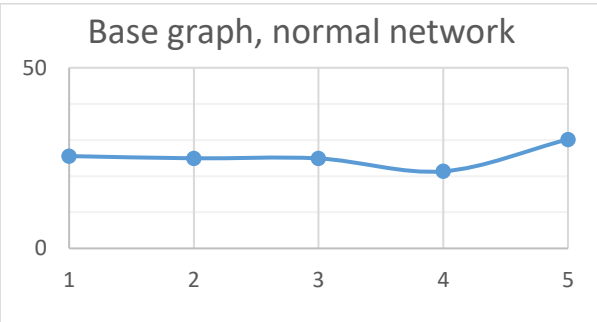
Notable:

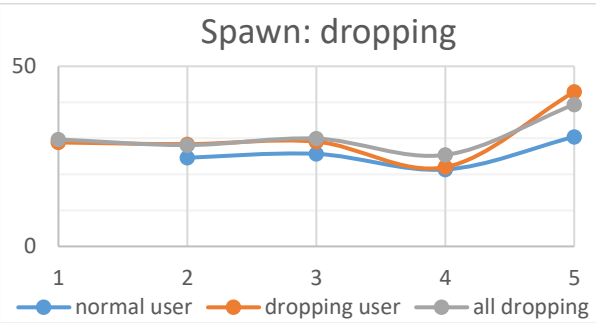
- Base graph depicting normal network and graph for higher drop chance increase slightly after 3 clients.
- The lagging network measurements and the measurements coming from the network with both lag and increased drop chance show the same characteristics as the one's for RTT:
    - Remain horizontal
    - Same measure in both graphs tend to hover around the same absolute values
    - Normal user < affected user < all affected

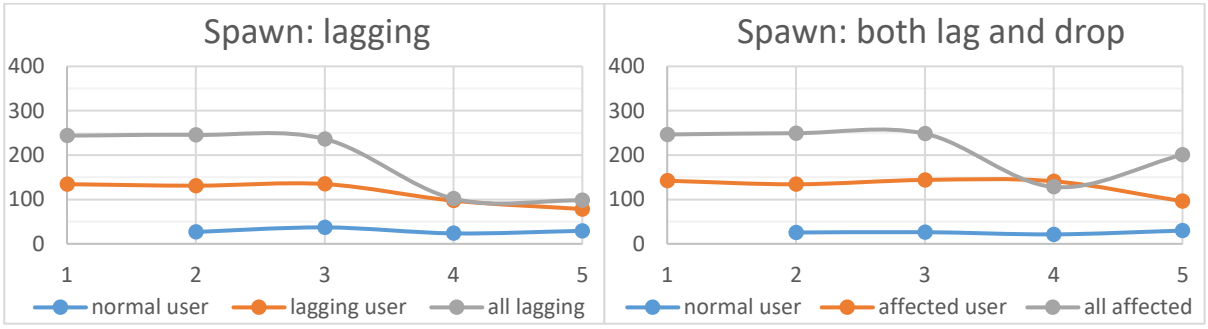
5.3.3 Spawning Results



In the graph above we see the mean results for spawning an object. While at a low number of clients the graph is similar to the ones for RTT and TDT, the later part of the graph clearly shows some interesting characteristics.



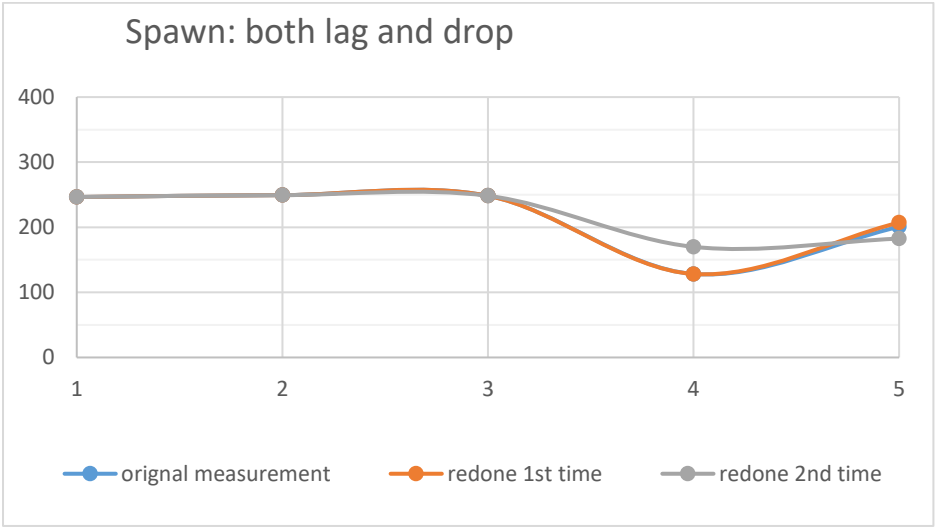




Notable:

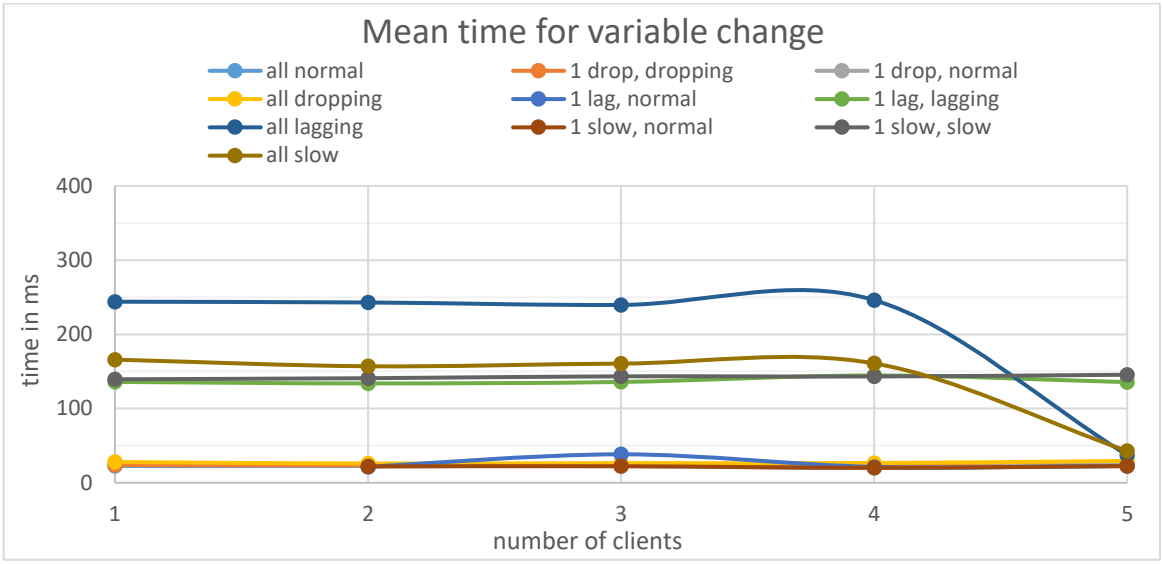
- The graph depicting a higher drop chance shows an increase at 5 clients.
- The graph depicting lag on the network shows a drop in spawning time for upward of 3 clients, when all users in the network are affected by the lag.
- Similarly, with both lag and drop chance increased, the graph shows a dip at 4 clients when all machines are affected by the issues.
- Whenever only one client in the network is lagging, then very few changes were measured when increasing the number of clients. Only when all machines were affected are there noteworthy changes.

Due to the sudden drop at 4 clients, the measurements for the network with both a higher drop chance and lag have been redone to compare results and check if the result is reproducible or an outlier.

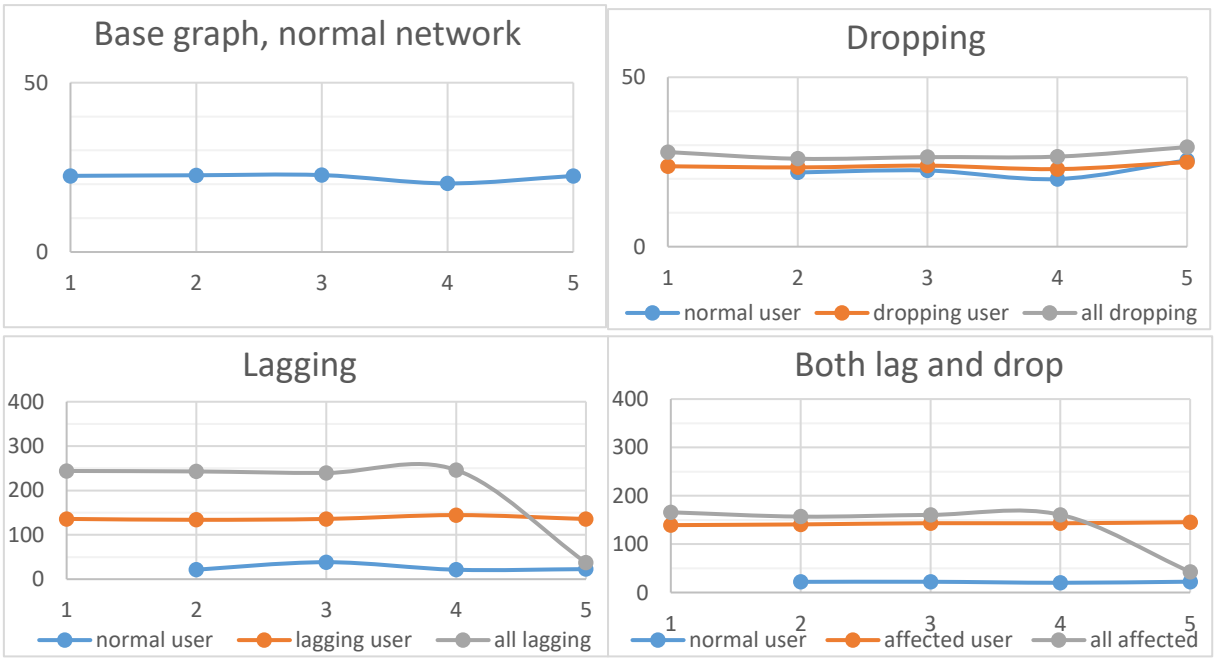


As is visible in the graph above, a second and third measurement have produced similar results.

5.3.4 Variable Change Results



On the overall graph for changing a variable we can see the same 3 tendencies as in most graphs so far. The one big change is the last measurement taken for the network in which all machines are lagging, and the one in which all are lagging and have an increased chance to drop network packets, causing a large dip at 5 clients.

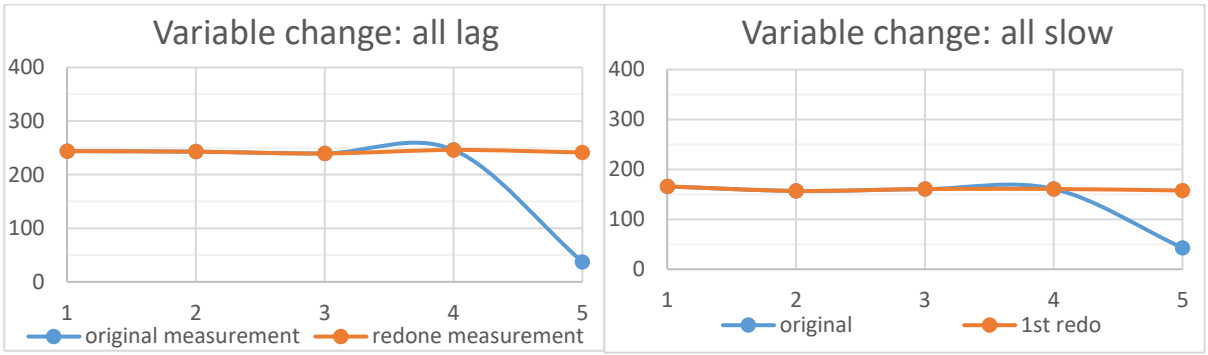


Notable:

- The base graph remains nearly horizontal
- In a network with an increased chance of dropping a packet, there is almost no change whatsoever. There is only a small overall increase in the time depending on the setup, but it barely changes relative to each other or to a changing number of clients
- In a network with lag, as well as both lag and a higher drop rate, we can see a few things:
    - Normal user < affected user < all affected.

- When only one user is affected by network issues, the graphs remain horizontal throughout.
- When all machines are affected by the issues, then there is a large drop at precisely 5 clients.

Due to the very large dip at 5 clients whenever lag is present in the network, those tests were conducted again to check if the results are reproducible or not.



As can be seen in the above graphs, a second measurement of 5 clients in a network in which all machines are lagging resulted in values that stayed nearly constant across all the number of clients measured. Redoing the measurements for a network with lag and an increased packet loss on all machines produced nearly constant values on one redo as well.

5.4 Conclusion

Given all the information gathered, the most relevant of which is presented in the graphs above, this chapter will infer the relevant information we seek based on the empiric data gathered.

In terms of the time that different operations take to be performed over the network, there is no substantial difference overall when adding clients to the session, given that no network issues are present.
While we can see a small but steady increase in some measurements when there are more than 3 clients in a session, this correlation does not seem to hold with most measurements. In general, most measurements remained nearly constant across all numbers of clients.

Inducing a higher chance to drop network packets does not seem to skew this finding in any meaningful way. This may be caused by the fact that we have declared all RPC functions to be reliable. This property causes dropped network packets to be resent when they are lost. In a network with no other issues, resending a single packet when lost will only add another RTT to the overall communication, thus not skewing the mean value significantly.
(This applies to relatively small drop chances, even when above average. If the drop chance were to be set to 80%, for example, then we expect such a large number of packets to be resent would indeed make a notable difference in measured time).

What does have a quite noticeable effect is lag. Whenever lag is introduced into the network, whether it be with or without a higher drop chance in addition, the absolute time of all operations is drastically increased.
While a network with only one user being affected by network delays shows increased measurements for only that particular user, a network in which all clients experience lag shows the increase in execution times for all users. In fact, the measured time of any operation in that case was even higher than the operations with a single faulty client showed. This could likely be explained though the issues/delay of the server cumulating with the ones of the clients when communicating with each other.

Across all measurements, there seemed to be a clear ranking of which conditions caused the most increase in execution time.

1. Increasing the chance of losing a network packet had little effect on the measured execution times, and thus it remained close to the normal network measurements.
2. Adding lag on a single client increased the measurements for that one client by a notable amount across all operations. The overall shape of the graph, however, remained roughly the same.
3. Having a faulty network with delays on all connected machines caused the highest increase in measured times. All operations were equally affected. As with one lagging user, the shape of the graphs stayed mostly the same. They only shifted upward along the time axis (y-axis).

One of our main interests was determining how network performance scales with a growing number of clients. The measurements indicate there was a very small increase in measured execution time of all operations except variable change when more than 3 clients were connected in a network with no issues. This change was present only at above three clients, but not with fewer.

Taking all measurements into account, there seems to be no correlation between the number of connected clients and UE4 network performance. The only indication of performance changes seems to be the network conditions (i.e. lag).

It is important to note, though, that some properties of the measurements suggest further investigation into the matter may be required to determine a sound result regarding this issue.
For one, the number of clients tested would need to be increased. Due to the fact that some measurements showed a very small but steady increase after three clients, it would be of interest if that trend continues at even higher numbers of clients in a session. Furthermore, the few, very sudden irregularities as seen in the spawning and variable change graphs are quite peculiar.
While further iterations of the tests showed significant variation in measured duration when it comes to changing a variable, spawning objects repeatedly displayed a slight drop at exactly 4 clients. Performing these tests in another network, perhaps with more scientific/laboratory conditions, may yield clearer results and/or produce additional data that would supplement the knowledge gained so far. This type of further insight would be helpful, if not necessary, to pinpoint the reason for this deflexion.

6 Avatar

Avatars are an important element for all virtual reality applications, and serve multiple purposes.
First and foremost, they are an important mechanism to strengthen the sense of presence and the perceived immersion of a user in the virtual environment.
The degree to which the avatar helps immerse the user depends on the properties of the avatar, which in turn may vary depending on the available hardware and platform.
While mobile VR can only track HMD rotation, other commercial headsets also track the HMDs position in space, and sometimes have motion controllers to track the position and orientation of both hands.

Another important aspect of having an avatar represent the user in VR comes into play when the application allows multiple users to share the same virtual space. In such an environment, having a virtual representation of the user is important for a large number of social cues that naturally occur when interacting with one another.
These can be unconscious actions such as portraying certain moods through gestures or body posture, or conscious actions like directing someone else's attention by pointing at an object.
Again, the properties of the avatar play an important role in how well some of these interactions may be displayed.

Thus, in the following I will first define three types of avatars, what our goal is in terms of representation in our application as well as give insight into the most important design and implementation issues of this work.

6.1 Avatar Types

As stated above, we will first define three types of avatars. The names for these avatars are not scientific or common knowledge, but rather have been given to them by the author according to their properties in order to refer back to a specific avatar configuration later in this work.

6.1.1 The "Partial Avatar":

The first type of avatar, which is arguably quite common in current applications, is one in which there are 3 distinct objects, each representing one source of input:

- 1 object representing the HMD
- 1 object representing the left motion controller
- 1 object representing the right motion controller

Figure 17:Example of partial avatar: Oculus avatar

The exact implementation may vary; while in many cases developers have a model of a hand for the left and right motion controller, it can be represented by any object, thus making this fundamentally a stylistic choice.
The notable property here is that each object is dislocated from one another, thus abstracting the human body to 3 points of information.

This is an easy and quick way to implement a user avatar. But it comes at the expense of many of the aforementioned social cues that humans use when communicating with one another. While the focus of attention can be easily seen simply by the orientation of the head, for example, it is much more difficult to infer another human's posture through only hand positions.

6.1.2 The "Complete Avatar":

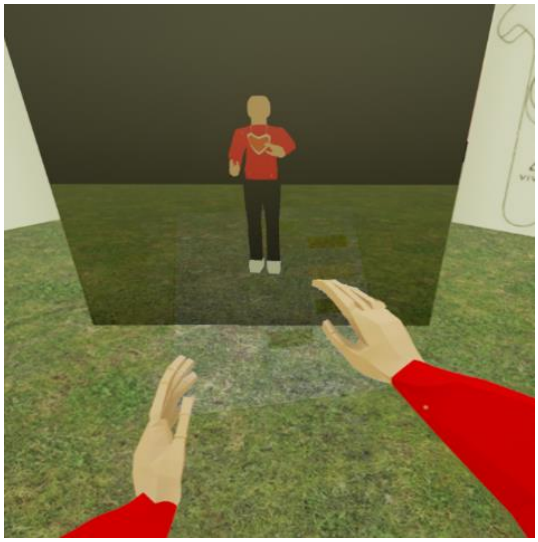


Figure 18: 1st person view of the complete avatar of this work

The second type of avatar is one which has a complete body modeled for the user.
As with the first type described the user controls 3 body parts with the available input:
The head, via the HMD, as well as both hands, using the motion controllers.
The difference lies in the fact that these 3 inputs are not disconnected in the virtual world, but control a completely modelled human avatar.
Similar to how a puppeteer controls a puppet, this type of avatar will also be controlled via a few distinct control points (the 3 just mentioned), and the rest of the body will adopt a pose that is compliant with all relevant constraints, such as joint positioning and length of body parts.

Using this type of avatar requires a lot more preparation on the content side and introduces a number of new problems on the implementation side (see chapter 6.5). The benefits from using this type of avatar, though, are ultimately a more natural and authentic way of communicating with one another in a shared virtual space.

6.1.3 The "Perfect Avatar"

The third type of avatar builds upon the idea of the complete avatar. But while the complete avatar places no requirements on the visual fidelity of the character, the perfect avatar earns its name by being an exact recreation of the user in virtual space. This would include an exact recreation of the unique features of individual users such as body proportions, facial features, etc. It would also entail giving the mesh a realistic texture that visually resembles the user.

Figure 19: Digital avatar recreations from A. Feng et.Al. [46]

No matter which type of avatar is to be used for an application, they all have certain challenges associated with them which need to be addressed. Addressing these challenges requires making choices in order to achieve a coherent, plausible and immersive experience.
So in the next chapter we will define the avatar we aim to create for this thesis, before going into the challenges posed by that choice.

6.2 Goal

In terms of the user representation, our goal is to implement a complete avatar that can be easily customized by the user. While the overall goal is to have the user be able to approximate his or her real appearance, the application developed for this thesis will not target that level of fidelity.
Instead, the focus lies on defining all parameters that need to be customizable, the technical setup required to change those parameters, and the further technical details that need to be setup to use the avatar and the application as a whole.
At the end of this work a guide is included showing how a user can create his own avatar. This will demonstrate the practical use of the resulting application.

6.3 Preliminary Work

There are several scientific publications that researched the effects of various different aspects of avatars and their influence on the user. For the next chapters, these papers will serve as a foundation to the decisions made for this thesis. As such, we will first briefly introduce the most relevant prior work in this field.

Fabri et. Al. investigated ways in which humans can communicate non-verbally inside a virtual world. Their focus was on the implementation of ways in which such non-verbal signals, mainly posture and facial expressions, can be conveyed by the avatar either autonomously or through user guidance. [39]
Another principal research was conducted by Martin Usoh and Mel Slater, who focused their work on immersion and presence. They studied which factors lead to immersion and presence, and what impact an avatar in general has on these factors. [5]
In the work of David England et. Al. they investigated temporal aspects of virtual worlds and the implications of consistency in a distributed real-time environment. While this is closely related to chapter 5, they further examine the importance and effects of avatars in such a virtual world. [9]

In a recent paper, Jean-Luc Lugrin and colleagues researched the impact of more or less realistic avatars on the user's immersion. They used a variety of complete avatars ranging from very abstract humanoid avatars (such as a robot) to realistic ones, and tested the perceived IVBO (illusion of virtual body ownership) of participants. [40]
A similar study was conducted by Gale M. Lucas et. Al, who tested the effects of having an avatar that looks like the user. They had a group play games using an avatar that looks like themselves, and a control group that plays with avatars with no resemblance to the user, and tested both the subjective and the behavioral impact. [41]
In another similar study, Mel Slater et. Al. tested the effects of embodying very specific avatars, and which changes these different embodiments resulted in. More precisely, they had participants fill out surveys before and after VR sessions in which they embodied a number of different avatars. Avatars were mostly complete and varied in numerous properties such as skin color, height, age, clothing, etc. They specifically came to conclusions regarding perceptual, attitudinal and behavioral changes by the user. [42]

Instead of researching the effects of just the pure resemblance of avatars, Donghun et. Al. focused their research on the effects of the creation process of an avatar on the user. They let one group of participants spend time building and customizing their own avatar before using it, while another group simply had their avatars assigned to them. They then looked for differences in attitude, empathy, presence and para-social interaction. [43]
More focus on a single trait of virtual avatars was put into the work of Daniel Freeman et. Al. Starting from the "established correlate of social rank and height" they explicitly explored the impact of manipulating user's height in VR. [44] Their studies involved having participants join VR sessions using a variety of avatars of different heights, and then investigating the social and behavioral impact of those varied avatars.

Armando Cruz et. Al. took a look at the relationship between presence and collaboration in a virtual environment. Based on literature review, they first defined presence and the important factors leading to it. They then did the same for successful collaboration. By comparing the results, they aimed to find out how presence and collaboration are connected, if at all, and how one can use that information to build virtual worlds. [18]

While the previous papers mostly emphasize questions about designing avatars, there are also a few papers that focus on novel technical implementations. Specifically, methods and implementations of a perfect avatar.
Dragomir Anguelov et. Al. did some work in that direction by introducing a method named SCAPE, that can create human models which include realistic muscle deformation. To do so, it requires a set of scans from a single person, from which it gathers all information needed to predict proportions and shapes in other poses. [45] They correct faulty scans, which are often noisy, and even make realistic body deformations possible with the help of an extensive database of previous human models. [45]
Using this SCAPE method, the team around Andrew Feng added additional functionality. With the scan as input, their approach produced morphable, rigged avatars. To do this, they used a database of previous human datasets and a pre-rigged avatar template. [46] The resulting avatar could then be changed in size and proportions while maintaining realism.
Another paper by the same researchers proposed a method that, in addition to rigging and weighting of the resulting mesh, can also capture the 3D scan of the user via commodity hardware.
They used a single Kinect camera, in front of which the user needs to do 4 static poses at 90 degrees to each other. Using that input, as well as RGB input from the Kinect camera, they constructed a fully rigged model of the user. [47]
Wang et. Al. proposed a similar method. They used a single Kinect camera to scan a person in a quick and easy fashion. The user had to rotate in front of the camera, and stay still for approx. 1 second 4 times while turning. The only restriction was the fact that hands must stay on the torso's plain, and legs must not be obscured by the hands. Unlike the previous paper though, Wang did not parse the image files, thus creating a mesh of the user but no accompanying texture. [48]
Similarly, Tong et. Al. also used the Kinect to create a mesh from scanning the user. In their work, they used a 3 Kinect camera setup so that each camera would scan a slightly different part of the body. Additionally, they also used the RGB camera of the Kinect to directly extract a texture for the resulting mesh. [49]

6.4 Design

This section of the thesis focuses on the problems that occurred and the questions that arose regarding the design of the avatar and the application as a whole. It also outlines how the work solved these issues.
There have been many studies over the last decades concerning the influence of certain avatars on human behavior in VR and human embodiment of more or less

anthropomorphized characters in virtual worlds. This research shall guide the design decisions made in this thesis wherever applicable.

As we have defined different avatar types earlier, note that the following extensive list of problems and decisions that need to be made are all relevant for the complete avatar, since that is the type we are interested in in the context of this thesis.

To understand the reasoning behind some of the solutions, it is important to understand some of the principles we aspire to follow. As well as basing the decisions on research wherever possible, the solutions are guided by a few simple aspirations:

- **Ease of use**
  Creating an avatar should require minimal prior knowledge of the system. An uninitiated user should be able to go through the customization process in a timely manner with minimal to no assistance.
- **Preparation time**
  The preparation prior to using the application should be minimal. The preparation time that should be minimized is specifically that of the user.
- **Immersion / Sense of presence**
  As with any VR application, having a sense of presence in the virtual environment is of great importance. Martin Usoh and Mel Slater even take it so far as to state that the measure of success of a system is the degree of presence its users experience [5]. As we will later see, this application will result not only in an avatar for VR, it will also let the user customize his avatar *in* VR.
  Thus, abiding by the properties for good VR applications, we will also aim to achieve a good sense of presence and immersive experience.

6.4.1 Degree of Realism

6.4.1.1 Problem

When deciding on how realistic an avatar needs to be, there are a few factors at play:
- Effect on presence/immersion
- Recognizability
- Required preparation
- Target audience

Creating a perfect avatar is not an impossible task, but quite cumbersome at many levels. On the other hand, highly stylized avatars are more straight forward in creation and easier for the uninitiated user to customize and setup.

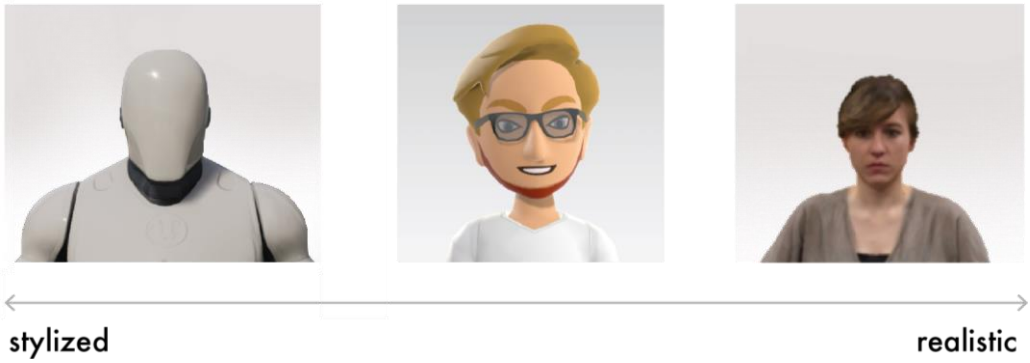


Figure 20: Examples on spectrum from stylized to realistic

The basis for striking a balance regarding this problem comes from scientific research in the area of immersion, presence and human behavior (inside virtual environments). These scientific theories will provide the necessary base on which the additional requirements can build upon.
For example: If research shows that an avatar which is slightly stylized has no detrimental effect on the user's immersion and/or presence, then the quicker and simpler creation of the stylized avatar as opposed to an inconvenient and timely creation of a realistic one is a worthwhile tradeoff for our use case.

6.4.1.2 Solution

The extent to which the avatar is a realistic visual representation of the user is important at multiple levels. It affects the preparation of the avatar mesh beforehand, the technology that can and/or must be used in combination, and the behavior of the player embodying the avatar [41].
The resulting character will be on a scale from fully stylized to photorealistic. (see figure 20)
First, the technical feasibility of creating a perfectly realistic avatar should be considered and investigated, since in terms of recognizability a perfect representation of the user would be ideal.
To create such a perfect player representation we would need to do 2 things:

1. Scan the player's body. This will create a mesh of the player with the exact physiology.
2. Take images of the player and create a texture from that which can be used on the mesh from step 1.

After completing these two steps, we would have a very good CG model of the user as a static mesh. To use it in our applications though, we would need to then rig the model so that it would move in a realistic fashion.
While it would be possible to have users create their avatar in specified institutions or perhaps departments of their company, we will favor solutions that are more flexible and cheaper to set up, since ease of use is a goal we have in mind.

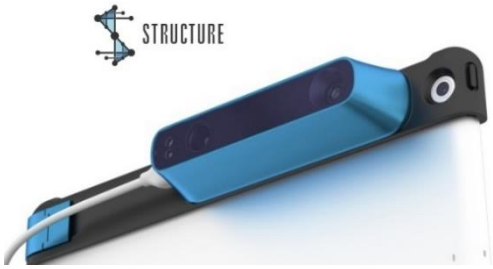


Figure 22: StructureSensor

Figure 21Microsoft Kinect

Fortunately, there is affordable consumer hardware that is capable of doing the exact two points just mentioned, namely 3D cameras such as the Kinect [50] or the StructureSensor [51], for example.

These cameras have additional depth sensors next to their normal photo cameras, so they are able to capture depth values which can be matched to the accompanied RGB values.
There are a few relatively recent papers that describe the creation of such avatars using this hardware.
The work from Dragomir Anguelov et. Al. would produce a good mesh recreation of the user. However, they do not use readily available, commercial hardware to produce the human scan in the first place. Furthermore, they produce only a mesh, but do not rig the character or add textures to it. So their method, at least without any additions, would not suffice for our needs.
The method from Tang et. Al., on the other hand, which makes use of 3 Kinect cameras, would be a more viable solution in terms of cost and space. But their resulting mesh is not rigged, so it would require further work on the character before it could be used in our applications. Furthermore, the very specific setup of the 3 Kinects makes the solution difficult to set up and immobile later on, which might pose a problem for some users.
Wang and colleagues described in their paper a method of scanning a person using a single Kinect camera, which from a cost and hardware standpoint would be a much more feasible approach. But their method does not parse image data and thus leaves the mesh untextured.
Similarly, the method proposed by Ari Shapiro et. Al. also makes use of a single Kinect camera, which makes hardware acquisition a non-issue, and they also rig and texture the resulting mesh.
While this does appear to be a more viable solution, there are two potential problems with this method. First, the setup is prone to human error and would likely need assistance on site for possible corrections when issues arise while capturing. The other problem concerns capture quality. Large or loose cloths can affect the automatic rigging, and the image quality captured is low, thus prompting Shapiro et. Al. themselves to suggest that this method be used for applications in which detail is not important and avatars are not seen close up.

So, while all the presented research has its merits, none can produce satisfactory results on their own, and thus would require further steps of processing the mesh afterwards. In some cases this would need to be done by professionals, which, in turn, would make the process cumbersome for the end-user, which is a scenario we want to avoid.

All in all, either the unreasonableness of the costs and/or necessary procedures to produce a lifelike human avatar, the lack of straight-forward methodologies or the low resulting quality of commodity hardware are a problem.

On the other hand, let us now consider using a stylized character.
Marc Fabri et. Al. found that a good avatar is not necessarily a realistic one. In fact, he even concludes in his work that too much detail on the avatar can be wasteful, since they could not find any measurable benefits in terms of immersion or presence when increasing the detail on avatars. [39]
These results are shared by Lugrin and Latoschik.
They found that the less realistic avatars were just as good, or even slightly better, for the illusion of virtual body ownership (IVBO) than the realistic avatars. [40]
They concluded that in attempting to achieve a realistic visualization of a person, even small details may destroy the illusion for the user. Furthermore, they suggest that in creating a realistic avatar, it is likely that users will experience the uncanny valley effect if not executed to perfection.
Mel slater and colleagues found that simply having a body in a virtual environment adds strongly to the user's sense of presence. In terms of the avatar, they concluded that functionality is a much more important factor for an immersive experience than appearance. [5]

Summary:
Taking all the research into account, it seems quite clear that a realistic avatar is currently not an efficient option for our use case. It would either be costly and inconvenient to create, or the quality would simply be too poor to use. And in either case we would risk the uncanny valley effect, which would likely discourage usage of the software.
On the other hand, using stylized characters has no detrimental effect on IVBO or immersion. They furthermore require less preparation by the user and can be more easily customized.

Conclusion:
Stylized avatars were chosen for this project. They currently offer the best trade-off between required preparation on the player's as well as the developer's end, and the required prior knowledge of the end-user to customize it. Additionally, they offer these benefits while being as immersive and furthering of the IVBO as realistic avatars would.

6.4.2 Mode of Customization

6.4.2.1 Problem

The question of how to customize the avatar concerns both the design and the technical implementation of the application.
The two main aspects which need to be decided upon are the following:

- Automatic vs. manual customization
- Customization in VR, desktop or both

As with the other problems so far, we can take a look at some research that deals with avatar creation for virtual reality and use it to infer an adequate model for our use case by considering further requirements placed on the application.

6.4.2.2 Solution

As mentioned in the problem statement, we can break this decision down into 2 parts, which we will answer separately.
The first thing we will decide on is if an automatic or manual customization should be implemented. To answer this question, we can use some of the information that has already been established in the previous section about the degree of realism. In that section, we have introduced research regarding creating realistic human avatars using commodity hardware. [47] [45]
Those methods aimed to create a realistic representation of the user. But we have already established our desire to go with stylized character visuals. Thus, we will disregard the realism aspect of those works and see if the remaining functionality would lend itself to creating a stylized character as well.
As a reminder, Shapiro et. Al. used a single Kinect camera which would scan the user in different poses and create a mesh out of the resulting depth data, while Anguelov et. Al. made use of expensive and bulky high-end hardware to scan the user.
Now we can already determine the better choice for our application by keeping our target audience in mind and recalling our reasoning for not going with either of these methods in the previous decision:
Using the possible setups at hand that would yield a correctly proportioned avatar mesh is both too error prone and requires too much preparation by our target audience.
Ease of use is important if we want first-time users, and quite possibly not very technically-versed users, to be able to undergo the customization process with minimal effort.
Additionally to the reasons already mentioned, research concludes that avatar choice can lead the player to identify more strongly with his avatar. [52]
Furthermore, Chung et. Al. suggested that spending time and effort in the creation of one's own avatar until satisfied with it reinforces a more positive attitude towards the avatar. [43]
Thus, disregarding automatic avatar creation and instead using a manual process in which the player sets up his character by themselves may even have a positive effect on the attitude of the player. Similar results were found by Vasalou et. Al. in their work. In 3 consecutive studies, they had participants create avatars to their liking, and had participants interact with one another with and without their avatar. They observed that users in a social application have a strong tendency to model avatars after themselves. More importantly, the customization of one's own avatar resulted in increased private self-awareness. [53] This means that the customization process itself is a beneficial part of the later VR experience.

The second aspect we must decide upon is whether to perform the customization process in desktop mode using a normal monitor, in VR using an HMD, or both.

Figure 23: Left; Example of desktop character creation in the game "Destiny"
Figure 24: Right; Example of character creation in VR from the Oculus avatar creation

There is no research that we can consult to make this determination, rather we must simply base this decision on practicality and the resulting use cases.
With that in mind, an approach in which both desktop and VR customization is possible seems to be ideal.
The application that this thesis is made in mind with will run on both conventional monitors as well as HMDs. That is, the resulting avatar will be used in both scenarios. So following that prerequisite, we cannot ensure that every user will necessarily have an HMD available, since we do not always require one. This speaks towards making creation possible on the desktop.
Additionally, from a development point of view only very few adjustments need to be made to make the VR version compatible with use in desktop mode. So practically speaking, there is little overhead to offering both versions to the end-user when beginning development with VR use in mind.

Summary:
Even without targeting photorealistic visuals, automatic mesh generation based on the player remains an expensive and/or error-prone technique that we do not want to burden the user with.
Regardless, research shows that spending the time on manually customizing one's virtual character might have positive effects on the player's attitude towards the resulting avatar.
And while desktop and VR require vastly different control schemes for interaction, the overhead of offering the user a choice between customizing in VR vs. in desktop mode is minimal.

Conclusion:
Customization will be a manual process, which the user can undergo either in VR mode or in desktop mode, depending on the available hardware. When carried out in VR, some adjustments can be made semi-automatically (by setting them automatically, but starting the process though user input).

6.4.3 DoF for Customization

6.4.3.1 Problem

When creating a system for the user in which an avatar can be customized, the designer/creator of the system must decide which properties of an avatar are customizable in the first place. This is an important decision to make, because different properties of one's avatar have different degrees of influence on the perceived immersion and the user's sense of presence in VR.
Similar to the approach towards dealing with the necessary degree of realism, scientific research has looked at different aspects of this problem.
Based upon the research done, the resulting customizable and non-customizable properties will be justified later in this work.

6.4.3.2 Solution

While there are a number research papers concerned with the influence of avatar appearance on the player's immersion and presence, I have not encountered any research that tries to identify all of the relevant, distinct properties of an avatar (in regards to the player's immersion, sense of presence or attitude towards the resulting character).
However, there are a few scientific papers that single out a specific property and investigate its influence, or that result in indicating the importance of some factors as a byproduct of what they were actually researching. We will take a look at these findings, and add some customizable features ourselves in the end where we see fit.

First, the work by Daniel Freeman and colleagues focused on height as a single factor and its impact on users.
They concluded that changing a person's height in VR indeed has an impact on the player. Not only does shortening a user make him less aggressive, but users made smaller also evaluated themselves more negatively after the play sessions. [44]
While the exact implications of height are not particularly a concern for this thesis, we can definitely derive one important piece of information from this study: height is an important factor for social interaction in VR, and thus should be represented accurately in our application.

In the early 1980s, Jaron Lanier already proposed the idea of "homuncular flexibility", which is how Lanier named his theory that people can be absolutely fine with inhabiting bodies that do not match their own, and can even be biologically strange. [54]
Another paper by Mel Slater et. Al. later investigated this claim, and how flexible people's body representation is while still maintaining immersion. Their results were similar, but far more extensive. They found that not only will people be okay with weird bodies, or ones that simply do not match their own, but that their attitude and behavior will unconsciously change depending on the body inhabited. [42]
Now, even though this information does not help us discern certain properties we will need to customize, it does provide us with a helpful piece of information:
VR avatars do not need to be a perfect representation of the user. This is a valuable

thing to keep in mind, since it means that, while we will offer customization options where possible, it is not detrimental if the options are reduced and do not allow for the perfect recreation of any human physique.
In addition to this finding, Slater specifically mentioned two properties of the inhabited avatars that had a noticeable effect on people's behavior inside the virtual world.
The first was size, which we have already identified as a necessary customization option.
The other was race. Racial biases were surveyed before and after sessions using a number of avatars with varying skin tones. While the results are not of interest to us (the interested reader should refer to the paper for further information), we can conclude from this that skin tone is another vital property of the user's avatar.
While it has behavioral effects as well, the simple property of recognizability also plays a role here, since we intend the application to be a multi-user VR environment in which quick recognition of one another is desirable. Thus, skin tone is definitely another factor to integrate into the customization options.
Quite frankly, these are all the avatar properties that are explicitly named in any research papers that could be found that are shown to have a definitive impact on the player's experience.
So from this point on, we must decide by ourselves what properties should be adjustable. At this point, we are currently changing height and skin color of the avatar. In their research, David England and Philip Gray emphasized the importance of awareness for intersocial communication in virtual worlds. [9] An important part of that awareness mechanic is signaling the focus of one's attention. This is done in large part by the direction in which a person is looking. Currently the avatar is missing a face, so adding at least limited facial features which can be changed is another sensible addition to the avatar.
From the research of Mel Slater we can also infer another factor inherent to avatars which can influence the user, and thus be beneficial to make customizable.
Through a series of tests with participants embodying a variety of human avatars he found that not only race, but also the type of body as a whole, including the style of clothing worn and the hairstyle, lead to a change in behavior, without weakening the subjective level of body ownership. [42]
So for our application, we can conclude that attire as well as hair are two additional factors that can play an important role in VR.
Though some papers suggest that exact body proportions are not needed to immerse the user in VR, this is a feature we would like to add as well. Being able to approximate one's physique slightly better is in my mind a desirable endeavor which should be an option for the user if desired.

Summary:
Limited research has been carried out to define the effect of changing single aspects of an avatar on the user.
In what research is available, skin color and height are the most influential properties of an avatar. Furthermore, adding facial features will help user communication, as well as the clothes an avatar is wearing.
Apart from that, we must add properties we consider important or relevant without scientific research backing those decisions.

Conclusion:
The following properties are shown to influence users, and should thus be customizable by the user:

- Height
- Skin tone
- Clothing (shirt, pants, shoes)
- Facial features
- Hairstyle

In addition, though no research backs up the importance of this factor, we will implement the functionality to adjust some proportions of the avatar. This is more a proof of concept and technical demonstration of how these proportion adjustments can be made than anything else. Thus, later extension in this area should be simple to develop. The proportions used as demonstration here are:

- Nose length
- Head size
- Head length
- Belly size
- Leg circumference

6.5 Implementation

In this section we will now cover the challenges and the solutions regarding the implementation of the project (in regards to the avatar).
These technical issues are of interest since they define the core behavior of the avatar, which has a strong effect on the user experience when using the application.
Note that this is not a complete list of the problems that can or will arise in the creation of all VR applications. These are merely the most pressing and interesting ones encountered during the development of this work.

6.5.1 Mesh Properties

6.5.1.1 Problem

The very first question in regards to implementing the customizable avatar is what preparation is needed. The creation of the avatar mesh needs to be done in a different software than the VR application is created in.
As such, we must define how the mesh needs to be set up beforehand for us to be able to change all desired properties at runtime using Unreal Engine 4.

6.5.1.2 Solution

All users will be presented with the exact same base avatar model at the beginning of their customization process. That baseline character mesh needs to be created in a 3D modeling software, and later imported into Unreal Engine to add the customization

functionality on top of it.
While creating the mesh, the first thing to keep in mind is what we want to be customizable later.

From a technical standpoint, most of the customizability is already created during this modelling process. So, to be able to modify all the parameters we desire later on in the application, the mesh needs to be created with certain properties in the first place. There are four things that need to be prepared in the modeling stage:

1. Model the mesh with smaller proportions than might seem "normal".
   From a technical standpoint, it is easier to stretch a mesh later in the process than to compress it beyond its original form. This is because shortening or decreasing any limbs or proportions can easily cause artefacts and unwanted effects such as clipping. This, in turn, means that creating a smaller default mesh with shorter limbs can be more flexible and easy to adjust later than a large one.
2. The mesh needs to be rigged.
   We want the player to be able to control the hands of the avatar via a pair of motion controllers, and the head via the HMD. While doing so, the mesh should always behave in a plausible manner. Plausible in this context means that the avatar should always be in a pose that is humanly possible. For example, arms should only bend at the joints (elbow, wrist, shoulder) when moving the motion controllers in space.
   For this to be possible, the mesh needs to be rigged. Rigging is the process of giving a model a skeleton. This skeleton has a hierarchical structure, at which bones connected to each other are parented to one another. The exact hierarchy is variable, and depends on the creator of the mesh, on the requirements placed on the skeleton, etc.
   After some additional fine tuning, which is not of particular interest in this thesis, this skeleton will define how and where parts of the mesh can bend and how the rest of the mesh will behave according to that change.
3. The mesh needs to be partitioned into logical groups.
   The look of the avatar is defined by the textures we put onto it. The skin color, the look of the pants, the color of the shoes, all these are simply textures we put onto the avatar.
   By default, a mesh will have only one texture, and that texture will be wrapped onto the entire avatar. We do not want to use a single texture though, since we want the user to be able to change individual aspects of the avatar without affecting the rest of the mesh's appearance. Using one texture for the entire model would mean creating a texture for every single possible combination of properties we let the user customize, which would be completely infeasible.
   Instead, we want to break the mesh into segments according to areas we want to be able to customize independently. This needs to be done after modelling the object. The modeler can then specify all vertices that should belong to a group. Then, every logical group can have its own texture assigned. That way, we can easily exchange the textures of a single group to change the look of a particular part of the mesh.

Example: The modeler will define all vertices ranging from the avatar's hip down to the avatar's ankles to be the "pants" group. Then, we can easily swap out a texture resembling blue jeans to one resembling black slacks, without affecting the rest of the character.

4. Add shape keys.
   Suppose we would like to change the length of the avatar's nose, or the circumference of the avatar's arms. To do so, we must use a certain functionality included in all common 3D modelling software. While different modeling programs might have a different name for this feature, I will refer to this feature as a "shape key", since that is the term used by the software used to create the avatar mesh for this application.
   A single shape key will define the change of a single aspect of the mesh. To create one, one must define the base state of the mesh, and the state of maximum change for that particular shape key.
   The change can then be controlled inside the modelling software and in the game engines by setting the value of a specific shape key to a value in the range [0, 1]. The mesh is then interpolated accordingly (0 being the base state of that shape key, 1 the maximum state).

   Example: When the mesh is finished, we define it as the base state of the mesh. Then, suppose we want to be able to adjust the length of the nose. We add a shape key based on the base state, and name it "GrowNose". Then, we can edit the state in the "GrowNose" shape key to reflect what it would look like at its maximum change. In this case we enlarge the nose of the character mesh. We save the shape key and export the mesh as normal. Now, through our engine we can adjust the "GrowNose" shape key value at runtime.

6.5.2 Player Size

6.5.2.1 Problem

Unlike the partial avatar, which would have the player's head (more specifically, whatever is representative of the player's head inside the VR application) float in space, the complete avatar must adjust the player mesh to the actual size of the player.
This is due to the simple fact that to create an immersive experience, we must have every user's avatar perfectly touch the ground with his feet, as if the player was standing on it.
Now, we know that the HMD will be at the correct height of the player in the real world, but we cannot guarantee the correct positioning in the virtual world at first. Through some character setup, we can get the player's view in VR to be positioned at the correct height (see appendix B for a more detailed setup), but that will still leave us with another problem, arguably more visible to the player.

The mesh we have created is a certain size. To feel like the player inhabits the avatar, his view must come from the height of the avatar's eyes. Rarely will the player's height match the avatar's height perfectly from the start. Therefore, we must somehow deal with repositioning the mesh so that it's eye height coincides with the player's HMD height.
If we simply reposition the mesh to the correct alignment, then depending on the player's real height the avatar will either be floating in the air or clip through the ground (depending on the difference between the avatar's height and real height).
But we have previously mentioned that we want the avatar's feet to always touch the ground to give a realistic representation of a person standing on the floor.
This needs to be addressed to convey a plausible virtual environment with numerous users.

6.5.2.2 Solution

As we have previously established, the size of the player is an important social cue, and one which we aim to recreate as authentically as possible.
To solve the problem of adjusting the avatar to be the correct player height, we must assume that the avatar mesh is correctly set up (as described in the previous chapter).

The HMD will naturally give us the real height of the player by default. What we need to do now is set the height of the avatar mesh to coincide with the user's real height. That is, the position of the HMD must be at the avatar's eyes.
To do that, we first get the height of the mesh, by calculating the height difference of the mesh's lowest part to its eyes. Keep in mind that the HMD needs to be at eye level, thus we calculate the height to the eyes, not the top of the scalp.
Then we compare the calculated mesh height to the actual player height. Depending on the height difference between avatar and the user, we will then need to either stretch the avatar if it is too short, or shrink the avatar if it is too tall.
We do this by changing the positions of certain bones in the avatar rig. It is important that the stretching be done using the bones. While one might think that it would be easier to scale the mesh, this would break the avatar rig, leading to random unwanted behavior upon moving the character. Instead, we can make use of the fact that the mesh will always move in accordance to its bones.
This also means that repositioning a bone further up or down will in turn also stretch or shorten the mesh at the influence area of that bone.
So to adjust the height, we first calculate the difference between current mesh and actual player. Then we reposition the height of several bones according to that difference.
In the current implementation this change happens when the user presses a button. That way, he/she can trigger the change at an adequate time (i.e. when he is standing upright).

6.5.3 Armspan

6.5.3.1 Problem

Similar to the previous issue with the player's height, arm span is also a factor which will vary greatly among users. And since the hands are 2 of the inputs controlled by the player, it is easily noticeable if the virtual arm span matches his real one.
This can be seen quite easily when spanning one's arms as far as possible and watching the virtual character's arms while doing so. The avatar will either reach its maximum span a lot earlier than the user, or it will remain with its arms at an angle while the user has stretched his arms as far as possible.

6.5.3.2 Solution

In the creation of the mesh, proportions such as shoulder width and arm length can already be accounted for. The modeler can use standard, mean body proportions to create a character that will exhibit the average body size.
This will suffice if no adjustments to the player height is done. When adjustments in height have been done, however, arm length will likely not match anymore.

When it comes to the arms, we have 2 factors that contradict one another regarding the importance of changing the length.
The first is the so called "homuncular flexibility", which states that people can inhabit strange and/or unrealistic bodies without impacting their immersion and presence. [42]
So from a single-user standpoint, adjusting the arm length is not of particular importance, since it will neither worsen nor enhance the user experience, according to current research.
When it comes to interacting with other people though, having the other person be wildly unproportioned may be strange. Thus, making arm length adjust to the player would be a worthwhile addition to the application.
When tackling the problem of stretching the mesh's arms, we have similar problems as with adjusting the height. Adjusting must happen though repositioning of the bones. So, to do that we again take the difference between the current hand position and the real hand position according to the positioning of the motion controller.
With that difference we can then infer where the bones need to be moved in order to achieve a new arm length.
At this point, due to the possible problems when shortening parts of a mesh, adjustments will happen only if the player's arms are larger than those of the mesh. If they are not, we would simply refer to the homuncular flexibility.
(But though careful avatar creation, this case could be minimized. For this work in particular, the avatar used might not exhibit the necessary properties to adjust to smaller individuals).
If the arms indeed need to be stretched, then their bones will be repositioned to stretch them accordingly.
To ensure a correct calculation of arm length, this must be done only when the arms in real life are stretched. Otherwise the length cannot be measured properly.

Because of that, this will also be triggered manually by the player, to let the player first strike an appropriate pose before manually triggering the process.

6.5.4 Body rotation

6.5.4.1 Problem

While adjusting the rotation of the player's head and hands is trivial, they alone will not suffice when moving, or even simply rotating around one's own axis in VR. Through the 3 points of information we have, we must infer further properties of the player's body. One of these is the rotation of the player's torso.

- If the rotation is not adjusted in any way, then the head and hands will rotate very unnaturally and angle themselves in relation to the player's body when turning in real life.
- If the body is rotated to always face the direction the head is pointing toward (as is often done in partial avatars), then the player will not be able to look left and/or right without inevitably adjusting the arm posture, often in an unnatural way.

Thus, inferring the body rotation in a plausible way is another non-trivial task that needs to be addressed to create an immersive experience.

6.5.4.2 Solution

Body rotation is an interesting problem for our setup. The hardware that we are using has 3 points of information:

- Head
- Left hand
- Right handFigure 10: HTC Vive hardware

Now we want to infer the rotation of the body at any given pose.
My approach in this thesis goes as follows:

1. Take the direction the HMD is looking towards and project it onto the X-Y-plane (=ground plane. See figure 25 for visualization).
2. Then, create a vector that connects the left and right hand with one another, take the vector perpendicular to that (pointing away from the player) and project that vector onto the X-Y-plane too.
3. In each frame, check if the angle between those 2 vectors is below 10 degrees. If they are, then I assume the player's body is rotated into the direction of the HMD's forward vector (projected onto the X-Y -plane again).

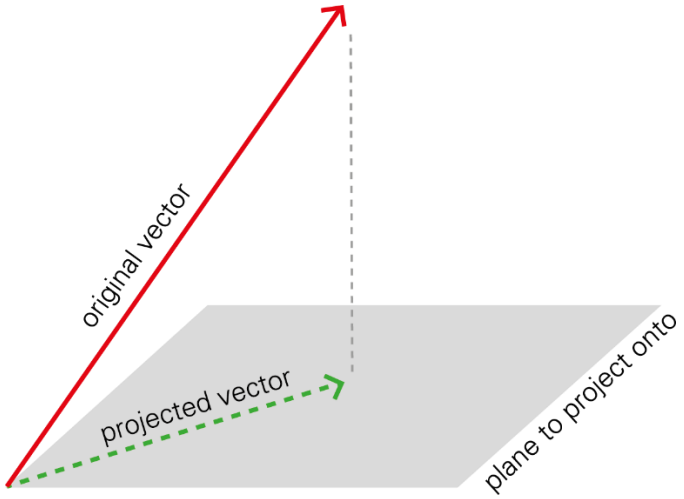


Figure 25: Visualization of vector being projected onto plane

This method is done for the following reason:
While standing, the player does not want the body to rotate with his head, since it is possible for humans to rotate their head ~90° in either direction without moving the rest of their body. And if in VR the entire body always rotates with the head at even the slightest changes, then this will cause strange behavior, since the mesh will change the arm positions to ways that are not physiologically possible.
However, if the player moves his head more than 90° in either direction, then we must assume that the player is turning his body, too. So, the body will remain at a 90° angle to the head while rotating.
The problem then becomes that when the player has stopped rotating, in real life his body will catch up with his head and align. In VR though, the body would remain at a 90° angle to the head, since no further rotation makes it turn.
At that point, the hands are used to estimate where the body's forward direction might be, by assuming that at a resting position, both hands will be approximately at either side of the user's body. If they are, then the vector perpendicular to the connection of the hands will point directly forward (from the player's point of view).
Specifically, we will continuously calculate the angle between the forward vector of the hands and the HMD's forward vector, and whenever they are within only a few degrees difference to each other we will assume the body is rotated into the same direction the head is, and set the body's rotation accordingly.
With this technique, we can now rotate the body using all the available information to the best of our abilities.

Keep in mind though that this method is a rough approximation of the body rotation, and will not suffice in many borderline cases. To accurately reflect body rotation additional information would be needed.

6.6 Conclusion

The aim behind the investigation into the user representation was to define a good customizable model with which the user can experience an immersive shared virtual session with other users, and demonstrate the implementation in an accompanying project using Unreal Engine 4.

According to a number of research papers, the best avatar to use is one that is visually stylized, as opposed to one targeting a photorealistic visual appearance. This is mostly due to the fact that with realistic avatars, immersion will break more easily when small details are not correct. Furthermore, there is little to no evidence that a photorealistic avatar has any advantages over a stylized one in terms of immersion or presence. So there is no added benefit in terms of the quality of the VR experience when using a realistic character.
Taking into account the added preparation needed, both in terms of the hardware setup and time, a simpler, stylized avatar is the ideal visual style to use for this work, and will generally be an adequate solution for most use cases.
In terms of customizability, there are a few points that were deemed important to the experience by research and were thus added to the character customization in the accompanying project.
Those factors were:

- Height
- Skin color
- Apparel

As further addition, I have chosen to also implement the following properties:

- Arm length
- Adjustable proportions:
    - Head (length and size)
    - Nose length
    - Belly size
    - Leg circumference

The latter features may improve recognizability of one another in virtual environments since they allow for a closer approximation of one's actual appearance.
And given the slight "proof of concept" nature of the accompanying work, the implementation of these features should also serve as a blueprint for future implementation and improvements on the application.

7 Outlook

This work was focused on the theory behind virtual reality avatars and their appearance. The accompanying project dealt mainly with the technical feasibility of the resulting customization options in Unreal Engine. This section will go into possible improvements and additions to the project and the theory surrounding it.

First, the design of the application can be improved, particularly the environment and the user interface which is used for character customization. The presentation of the thesis' results and the user interface for customization in a virtual environment were

Figure 26: Oculus avatar creation

Figure 27: Morph3D "Ready Room" character customization

implemented with functionality in mind. Several different approaches to character customization can be found when reviewing other VR applications. Examples include the Oculus avatars, which are customized by looking at a mirror and choosing options through buttons on the mirror's surface [55], or Morph3D's "Ready Room" character creation, in which the player customizes a miniature version of his avatar that is placed on a table in front of him [56].

Due to the novelty of the consumer VR platforms, there is no quasi standard that has established itself yet around customization. Thus, further research in this area and subsequent implementation would greatly benefit the user experience.

In terms of implementation, there are a number of possible improvements in varying areas that could be made.

Arguably the most obvious one is to add a female character and improve the overall fidelity of the mesh. Due to the proof of concept nature of this work, those features were not a high priority.
A different way in which the avatar fidelity could be improved is through the implementation of human scanning techniques like the ones mentioned above, or through similar approaches not mentioned in the context of this thesis.
As explained earlier, all the approaches in the considered research were disregarded for various reasons. However, with some improvements on the methods detailed in the papers earlier, this project could make use of the realistic recreation of the users. Furthermore, some methods were disregarded due to the argued impracticality of the setup and preparation. This is not a universal downside. Rather, some users may be willing to dedicate the space, time, and sometimes cost required for an avatar's initial setup. For those users, the option to use such a technology may be useful.
The use of a scan of the user has the potential to greatly improve the experience of the player as well as improve social and professional interactions inside the virtual environment. Therefore, keeping an eye on new research and evaluating novel approaches regarding human avatar recreation would be advisable to keep the project up to date.

But also using stylized avatars, which we have concluded will in most instances be the reasonable approach, can be improved upon.
First, the avatar moves only with the 3 inputs of the user. To add additional lifelikeness, animations could be added to the character. Examples of this include a walking

animation that plays when the avatar is moving, or an idle animation which indicates breathing through a subtle movement of the chest.
Furthermore, poses which are hard to realize accurately through tracking position and rotation alone, but beneficial to the player's expressiveness, could be added. What comes to mind is when the player folds his arms. This pose is a meaningful non-verbal signal [57], but is hard to accurately portrait with tracked hands alone. Thus, recognizing when a user is about to strike a certain pose and then switching the character to a pre-defined version of that pose may enhance non-verbal communication between users.

Appendices

Appendix A – Character Creation Walk-Through

In this appendix we will demonstrate how to generate one's avatar using the resulting project. The entire process will take only a few minutes, but the lack of a tutorial in the application itself calls for a short "how to" demonstration in this work.

There are two possible ways to customize one's avatar: via desktop or in VR.
As such, this section will describe both of those approaches.

Desktop Creation

When opening the executable file, the user will be presented with the main menu, seen in image 1.

The main menu has 4 buttons to choose from.

- The "Host" and "Search" buttons lead to the application that was used to do the measurements seen in chapter 5.
- The "Close" Button will shut down the application
- The "Avatar Creation" button will load the level in which the user can customize his avatar

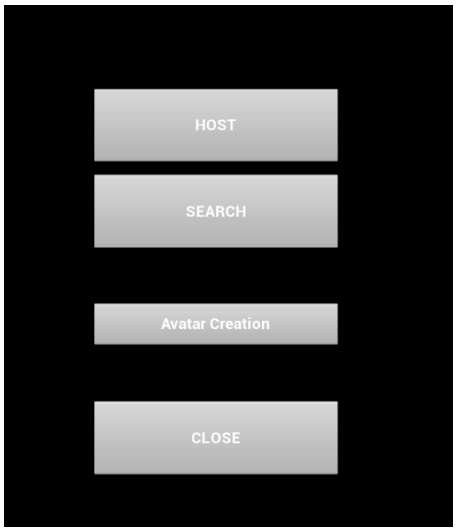


Image 1:Desktop main menu

When opening the avatar creation level, the user will transition to image 2.
The image highlights the important sections of the screen:

1. Morph targets:	Here, the morph targets can each be adjusted via a slider.
2. Textures:	Changing the appearance of certain regions of the avatar is denoted by the name of the region, and the corresponding choices next to it.
3. Save / Load	Since the avatar will not immediately be used in another scene upon creation, and since we do not want to force the user to recreate an avatar every time the application is opened, the current appearance can be saved with the corresponding button. If a saved avatar is present, then a "Load" button will also be visible, which will load the saved appearance.

The controls in this level are simple: With “A” and “D” the user can rotate the character to the left and right, respectively. That way he can inspect changes from all angles.
The entire interaction is otherwise done via the mouse. Changing textures is done by left-clicking the corresponding buttons, while the morph target sliders must be moved while the left mouse button is pressed down while on it.
Before saving, the user can enter a name via focusing on the text field by clicking on it, and then saving via the buttons next to the text field.
To exit the application, the “ESC” key needs to be pressed, which will close the entire application.

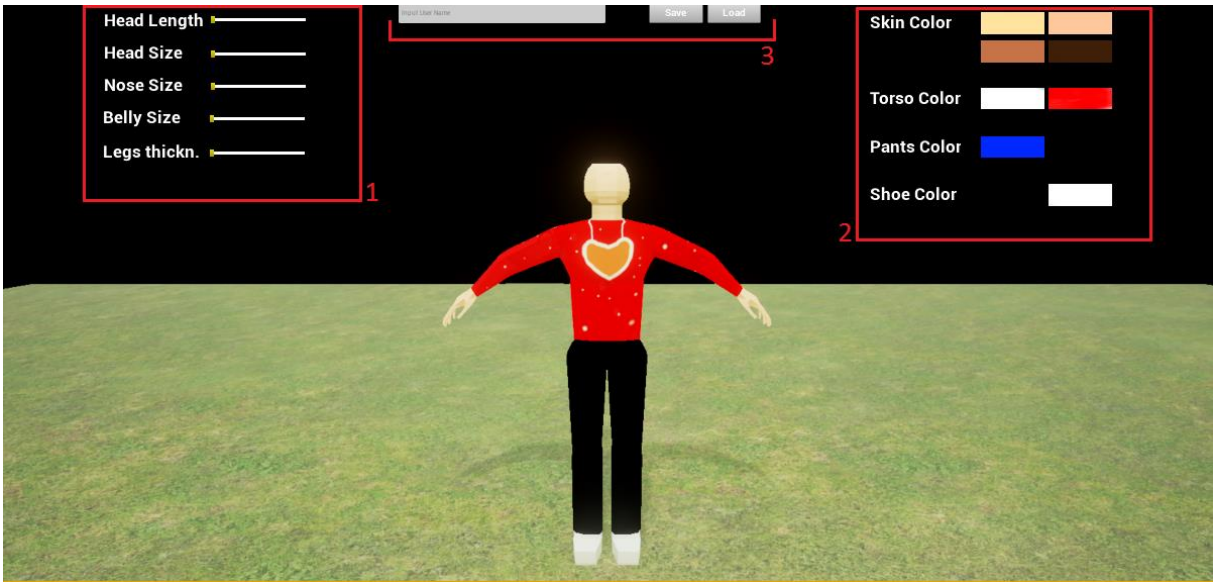


Image 2: Desktop character creation screen

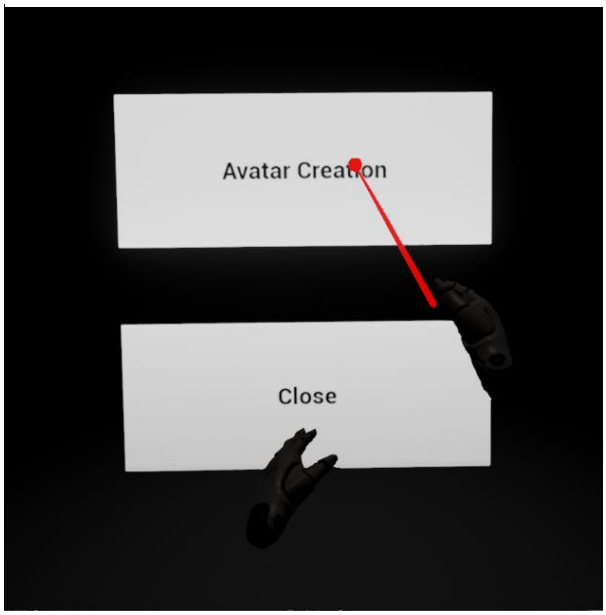


Image 3: VR main menu

VR Creation

Only though the VR character creation can the avatar be customized in total, since only in this mode can the user also adjust the height and arm span.
Naturally, the controls and the UI are quite different to the desktop version in some ways.

Upon running the application, the player will be positioned in a black space with nothing but 2 large floating buttons and his two hands.
On the right hand there is a red line that runs forward. The user can press one of the buttons by pointing at the desired button with the right hand such that the red line

collides with it, and then press the motion controller's trigger.
The two options in the VR main menu are:

- "Close": The application will shut down.
- "Avatar Creation": This will load the character customization level.

Inside the VR character customization, the player will find the same customization options as in the desktop version, alongside 2 additional features which were not present in the desktop version: adjusting arm length and avatar height.
The controls are quite different though, and thus we will go through the customization process:

When the level loads, the player finds himself standing on a plane, in front of a mirror. To the left and right of the mirror are two billboards that indicate how to access the customization functionality. The billboard to the right indicates the options on the right motion controller, the billboard on the left shows the left motion controller's options.

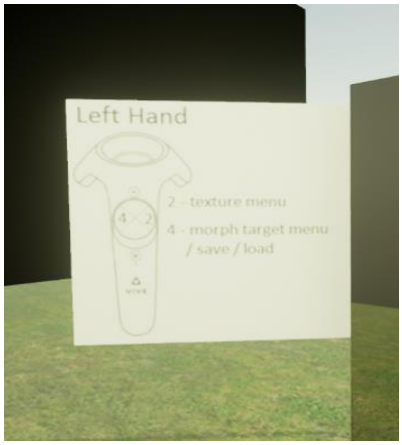


Image 4: left controller VR guide

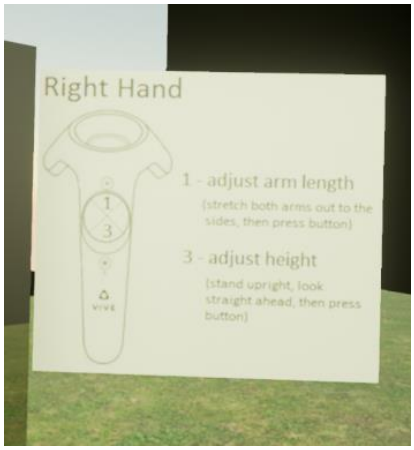


Image 5: right VR controller guide

Image 3: right VR controller guide

1. The first thing to do is adjust the character height and arms. To do this, the

Image 6: Image of a "T-pose": arms stretched to the sides

player should extend his arms sideways such that they are completely stretched out, ideally striking a so-called "T pose".
While in that pose, the right controller's "facebutton 1" needs to be pressed. The billboard indicates where exactly that button is located. This will adjust the arm length somewhat.
Then, while looking straight at the mirror, the "facebutton 3" should also be pressed. Doing this will change the height of the mesh.

2. Next, we will change the appearance of our avatar. To do so we use the instructions visible on the left billboard.
   By pressing the "facebutton 2" on the left motion controller, a small menu will appear in our character's left hand. This menu is only present as long as the button is held. Using this menu, we can change the appearance of different body areas, just like in the desktop version.
   To choose appearances, we must press the button with the right index finger. Upon touching the button the change will be made.

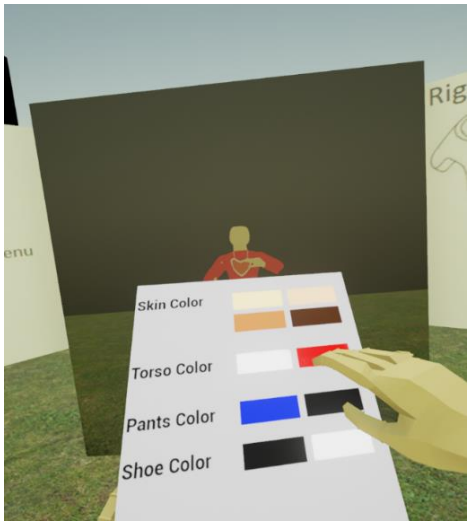


Image 7: Texture menu in VR

3. After we have chosen all of our textures, we will set some morph targets. To access the morph target menu, we will press the "facebutton 4" on the left motion controller.
   Again, a menu will appear in the left hand. To change values on the morph target slider, we need to touch the slider with the right index finger and press the trigger on the right motion controller as well. Then, we can change the slider to whatever value we want. Releasing the trigger or pulling back the right hand will both stop further adjustments of the sliders.

4. Lastly, on the morph slider menu we will also find the "Save" and "Load" button. So we will save the appearance of our avatar, before closing the application with "ESC".

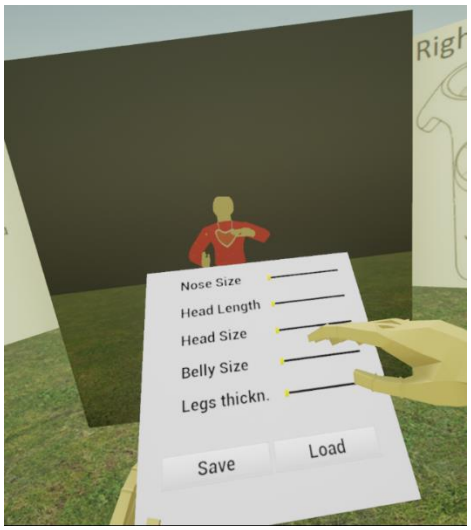


Image 8: Morph target menu in VR

Appendix B – Project Implementation Guide

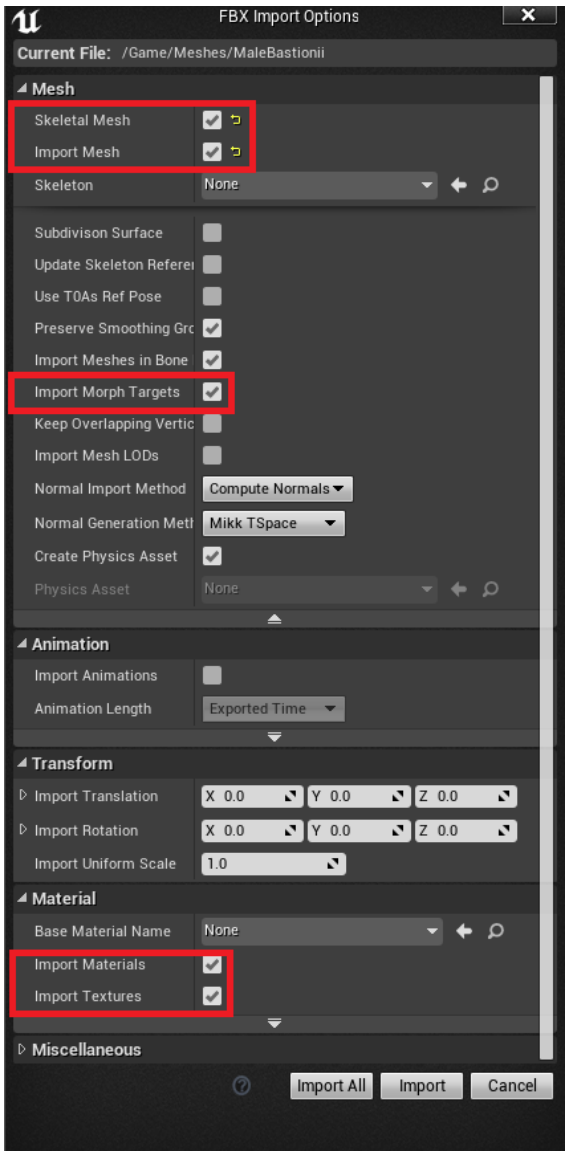


Image 9: Import screen and options

In this appendix I will give a step-by-step description of the setup creation of the avatar behavior in Unreal Engine 4. Specifically, it will detail the steps necessary to create a complete avatar that will behave in a plausible manner while being controlled via 3 inputs: 2 motion controllers and an HMD.

As mentioned earlier, the mesh must be prepared in such a way as to incorporate a variety of changes in the mesh data itself. Refer to chapter 6.5.1 for descriptions of what properties must be present in the character.
For this appendix, we will assume a correctly prepared character mesh that has been exported as a .fbx file, since the modeling process is not a core part of this thesis, and will depend heavily on the modeling software used.

The overall steps to reach our controllable avatar are as follows:

- Import the character mesh
- Create an animation blueprint that defines the behavior
- Create and set up pawn
- Combine pawn logic with the animation blueprint

We will go through each step sequentially and show the settings needed to be made in the engine.

Import the character mesh

Importing the character mesh is a quick and easy process, since UE4 provides a simple function which will automatically import any file format it supports.
It is explicitly mentioned here, though, since at the import stage the user can toggle numerous import settings, which will heavily affect the later behavior of the imported object.

Image 9 shows the import options box that is presented to the user after choosing the object to import. The highlighted options are the ones most important to us. When

importing, make sure that these options are toggled on.
While most of these options will be on by default, making sure to manually turn on the "Import Morph Targets" option is important to allow for later adjustments of specific mesh properties/proportions.

Once these options have been checked, simply clicking the "Import" button will automatically process and import all the required data, and will then place the mesh inside the projects folder structure (depending on where the user has opted to import).

Create animation blueprint

UE4 gives the user the possibility to manually define poses of a rigged mesh via so called "Animation Blueprints". These animation blueprints modify a mesh's pose by adjusting position, rotation and/or scale of single bones, thus changing the resulting pose of a character even when animations are running.

With the help of an animation blueprint, we will be able to change the mesh height as well as bind the character's hands to the position of the motion controllers.

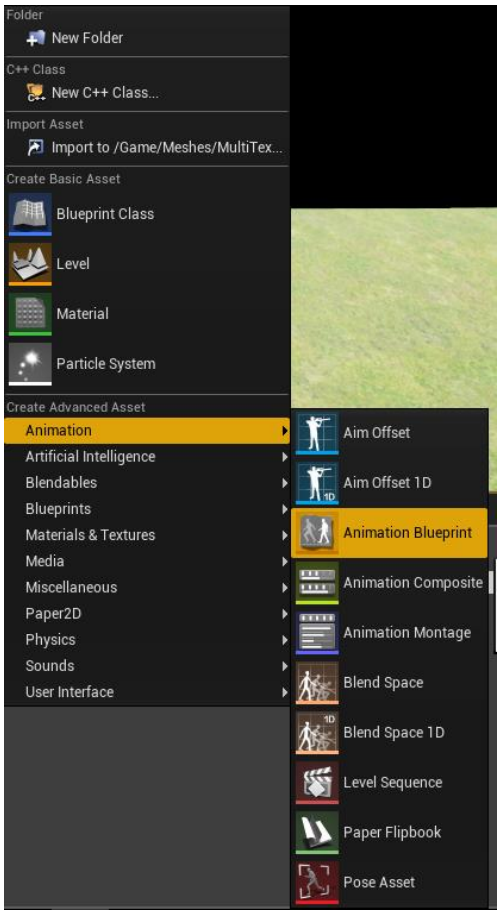


Image 11: Menu navigation to create new animation blueprint

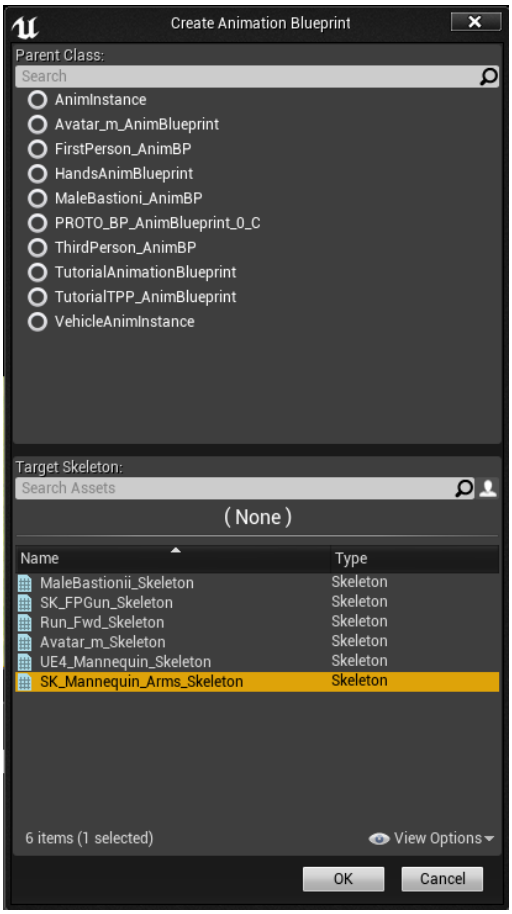


Image 10: New animation blueprint options

First off, we have to create an animation blueprint. As seen in image 11, creating an animation blueprint is found under the "Animation" option when choosing what new object to create.

Upon creation, the user is prompted to choose a skeleton for which this animation blueprint should apply (image 10). Here we choose the skeleton that has been created for our character after importing the mesh. We do not need to specify any specific parent class, since we will not need any preexisting functionality from any other blueprints.

Knowing the Skeleton

Once we have created the animation blueprint, we can start defining how to position the mesh's bones. But to do so, we must first know the bone hierarchy and the bone names of the character. We can inspect the bone hierarchy and their names by simply double clicking the skeleton that has been created during the import. The resulting window can be seen in image 12.

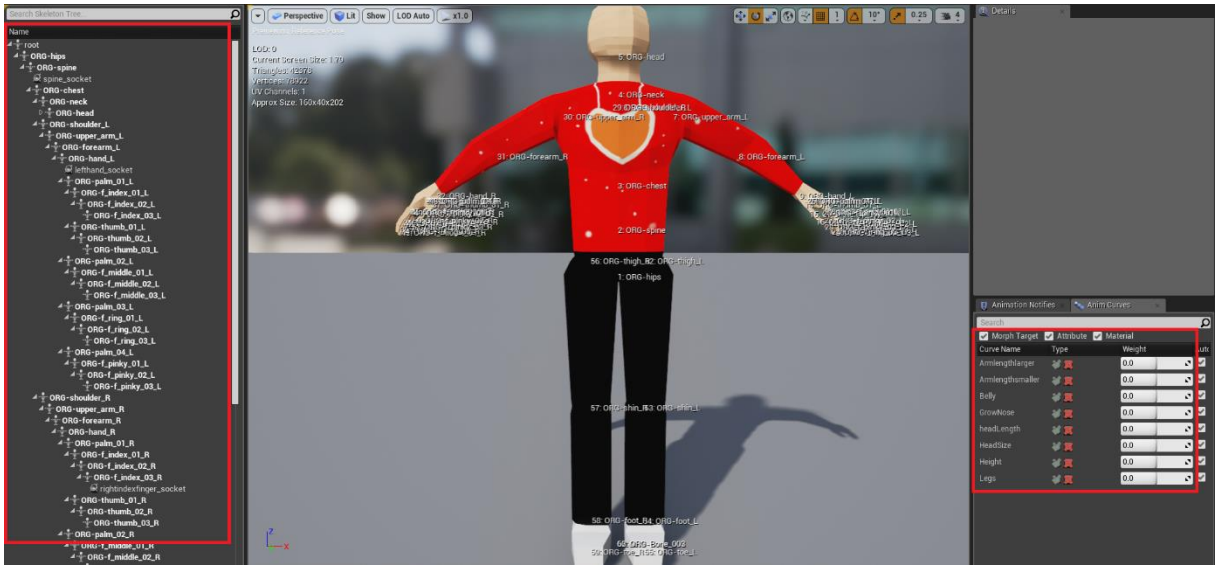


Image 12: Newly imported skeleton

In that window, we see the entire bone hierarchy on the left, the mesh in the center, as well as a panel for details and one for morph targets on the right. Note that in this window you can check if the morph targets created in the modelling software have been correctly imported into the engine.

For our purposes, we are interested in 2 specific snippets of the hierarchy. As mentioned in previous chapters, the character rig must be set up in a specific way for us to be able to adjust the avatars height and to control the arms through our motion controllers. The hierarchy must have the following properties:

- For controlling the arms we need the skeleton to include the following bone hierarchy:
    - Shoulder
        - Elbow
            - Wrist

  It is not prespecified how the hierarchy is composed beyond the wrist or before the shoulder, but the arm must resemble the real human anatomy when it comes to the number of connections between the shoulder and the wrist.

Note that the bones in the mesh may have names different than “shoulder” or “elbow”. The names are unimportant as long as the number of bones in the hierarchy is correct.
Image 14 shows the part of the bone hierarchy that matches this description in our case.
For a better understanding later on, listed here is the hierarchy of the skeleton used in this thesis and shown on the images:

- *ORG-upper_arm_L / ORG-upper_arm_R*
  - *ORG-forearm_L / ORG-forearm_R*
    - *ORG-hand_L / ORG-hand_R*

- To be able to adjust the height, the skeleton should have:
  - A bone above the hips, roughly in the stomach area.
    In this specific example: *ORG-spine*
  - The legs, ranging from the hips to the ankles, should be composed of as few bones as possible. This will simplify the process of shortening the legs when changing the mesh height.
    Again, in the current application, those bones used are:
    - *ORG-shin_L / ORG-shin_R*
      - *ORG-foot_L / ORG-foot_R*

Image 13 shows the part of the bone hierarchy that matches this description in our case.

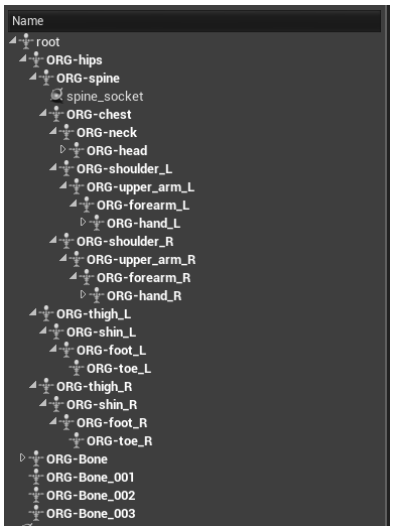


Image 15: Bone hierarchy

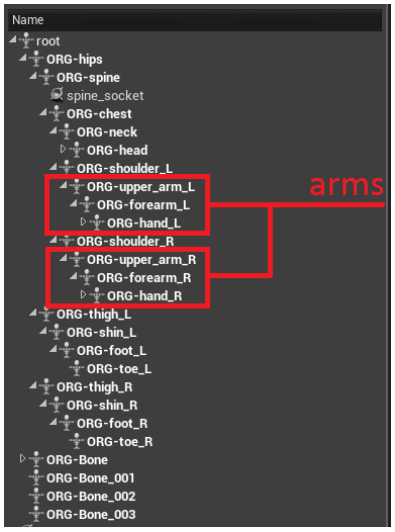


Image 14: Relevant arm bones highlighted

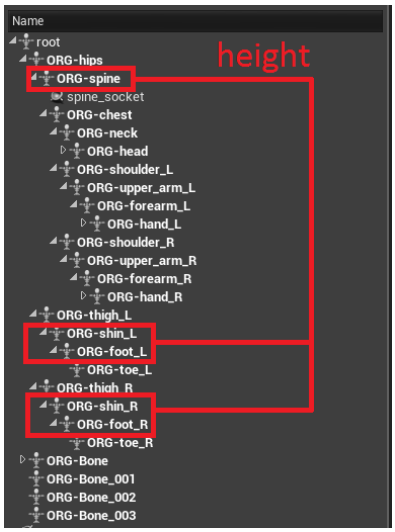


Image 13:: Relevant leg bones highlighted

Now, after inspecting the skeleton and finding the names of the bones we are going to need, we can construct the blueprint.

The Anim Graph

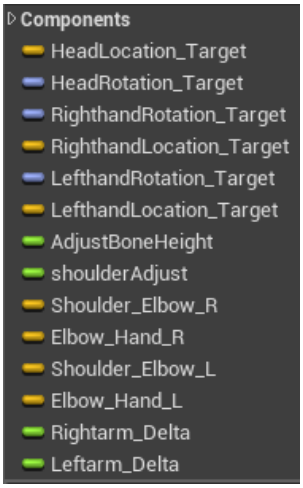


Image 16: Animation blueprint variables

First, we need a number of variables. See image 16 for the list of variables we will need in the animation blueprint. All these variables will be used to set certain parameters in the blueprint. For what we want to achieve, all the relevant work will be done in the "Anim Graph" of the animation blueprint. We can roughly partition the graph into 5 logical groups, all concerned with a different piece of the avatar. Image 17 shows the entirety of the Anin Graph, before we sequentially go through it step by step to show the setup of each group.

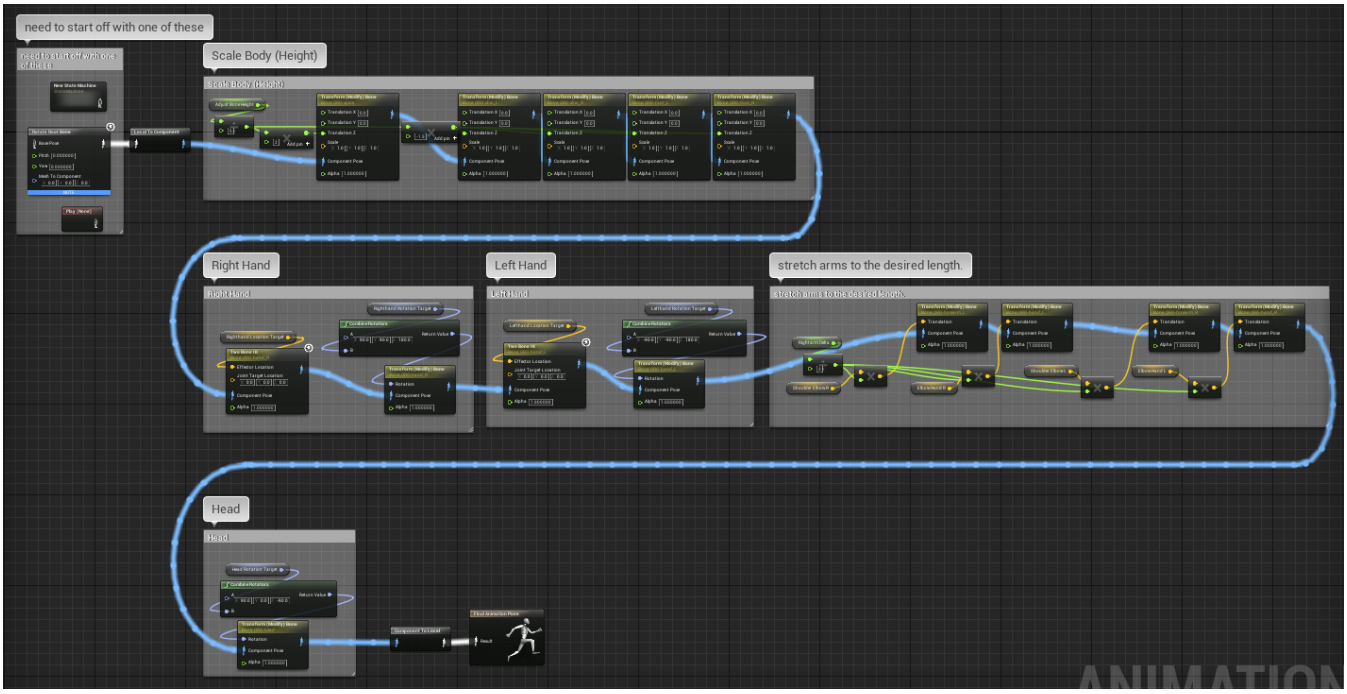


Image 17: Entire Animation blueprint AnimGraph

1. We must begin the animation graph with some state in which the mesh can reside. When dealing with objects that are continuously in some animation, one could use the animation node to retrieve the mesh state, or get the current pose from a state machine. Since we do not have any animations running, we need to get a pose we can work from without changing anything. This can be done in multiple ways. In the current implementation I have chosen to simply start off with an empty rotation of the root bone.

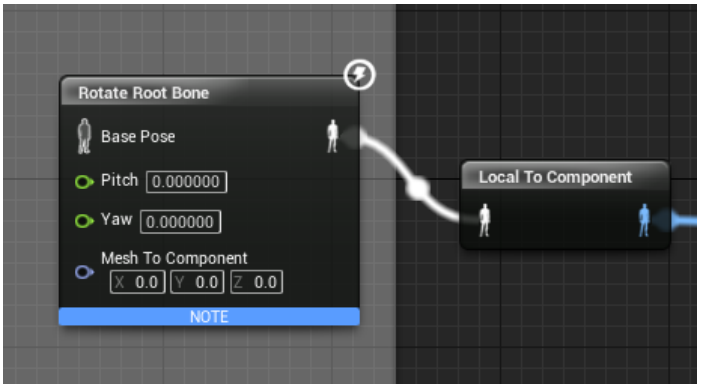


Image 18: Start node

Before working on the mesh's pose, we have to convert the pose from local to component space.

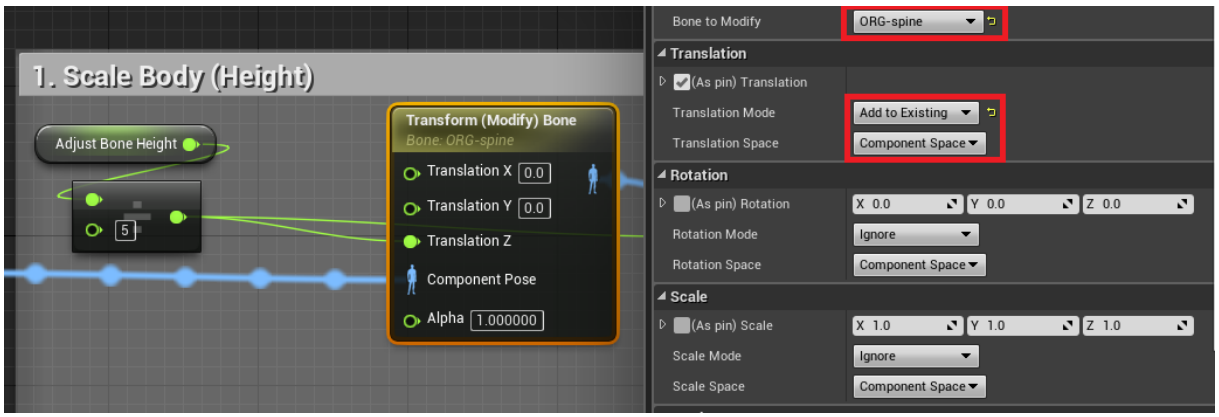


Image 19: Node settings for changing body height (excerpt)

2. The first thing we deal with is the height of the mesh. Changing the height is comprised of doing the same operation on 5 different bones. More specifically, we change the position of the following 5 bones: *ORG-spine, ORG-shin_L, ORG-shin_R, ORG-foot_L, ORG-foot_R.*
   Image 19 demonstrates the setup for one bone.

   We use the "Transform (Modify) Bone" node to manually adjust the position of a single bone.
   In the details panel of the node, we have to set some variables.
   - At the very top we have to set the bone to which this node should apply. The dropdown menu will show all bones in the skeleton defined at the creation of the animation blueprint.
   - Under the translation tab, there is "Translation Mode" and "Translation Space". The translation space we want to use is "Component Space" (this will make all translations relative to the components local space. If the local space is not known, then it can easily be inspected by looking at the skeleton object and selecting the desired bone).
     The translation mode should be set to "Add to Existing". That way we only add an offset to the bone location, rather than setting it absolute.
   - We can turn off rotation and scale by toggling the checkbox under the respective tabs. This will not change the functionality, but will make the blueprint easier to read.

   Finally, we need to give the node the offset as an input. Since we are only interested in changing the height of the mesh, i.e. the z direction, we split the input into 3 distinct input parameters and pass a fraction of our "adjust bone height" variable as a parameter. This variable stores the total height difference we want to achieve.
   We only pass a fraction of it to the node, since we want to stretch the mesh by moving 5 bones at once to achieve a certain height, hence the change is split up among all bones.
   To achieve an even stretching of the mesh, we must do one further thing: The spine and leg bones are offset into opposite directions. When stretching the mesh, the spine should be stretched upwards, while the legs stretch down, and vice

versa when shortening the mesh. Therefore, before inputting the parameter into the nodes for the legs, we must multiply the fraction by -1.

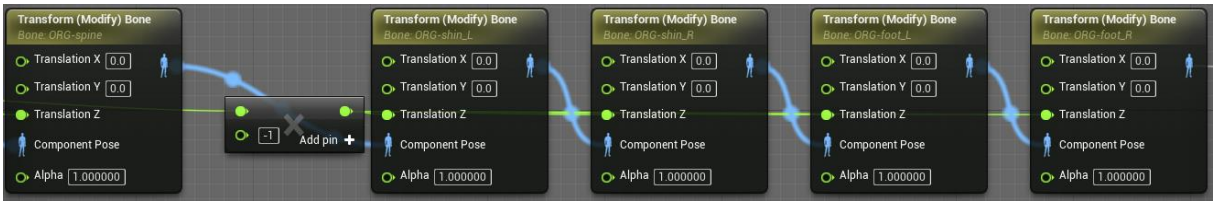


Image 20: Entire node sequence to stretch the mesh

3. Next up, we will concern ourselves with the arms. Our Right-/LeftHandRotation_Target and Right-/LeftHandLocation_Target will contain the positions and rotations of our motion controllers where we want to position the character's hands while maintaining a plausible body pose.
   We will do this by using inverse kinematics. The animation blueprint has two types of inverse kinematics solvers we can use, each of which is embodied by a single node.
   This project uses the so called "Two Bone IK" node. This node will take a bone for which the end position is predetermined and, as the name implies, solve for 2 bone positions in the hierarchy.
   This is where the importance of the hierarchy at the arms comes into play. We want to place the hand at a position in space. Furthermore, we want the entire arm up until the shoulder to adopt a realistic pose. The solver can only achieve this if we have exactly 2 bones from hand to shoulder. This is due to the fact that this specific solver will only be able to solve for 2 bones, hence the name.

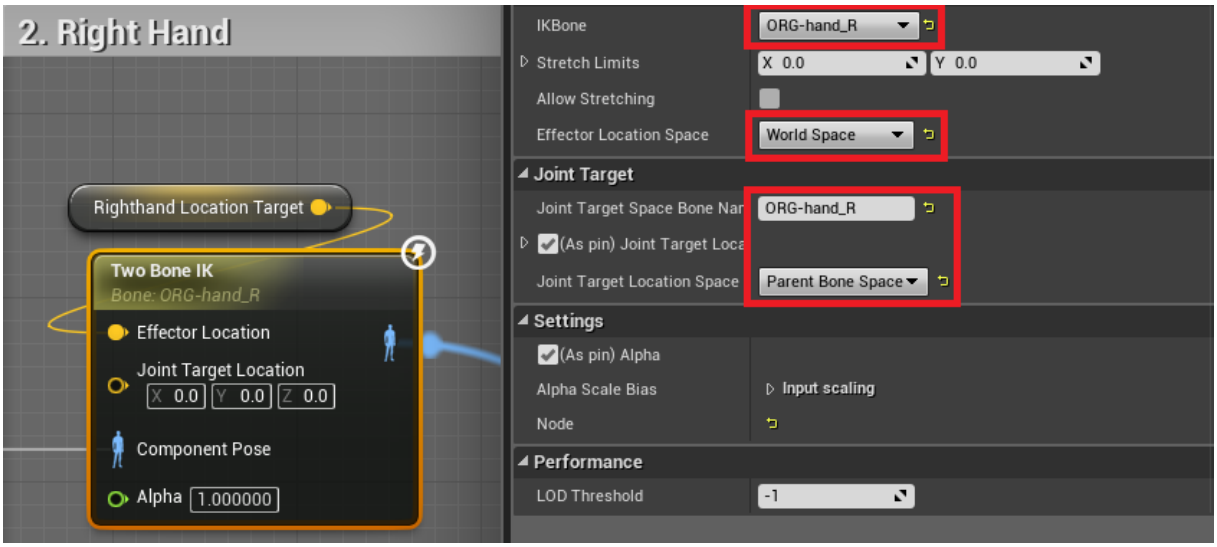


Image 21: Settings for the inverse kinematics solver node for the hands

Now that we understand why the hierarchy was so important, and what node to use for solving the problem, we shall look at the input and the settings required for the node to behave as we want it to.
As mentioned, our variables will hold the locations and rotations we want to place the hands at. Therefore, our location variable will be the input for the "Effector

Location" of the node. This denotes the position our final bone should be at.
The "Joint Location" has proven not to be hugely important in this case. This parameter would determine into which direction the joint will preferably bend. But leaving this parameter at the default setting will already produce satisfying

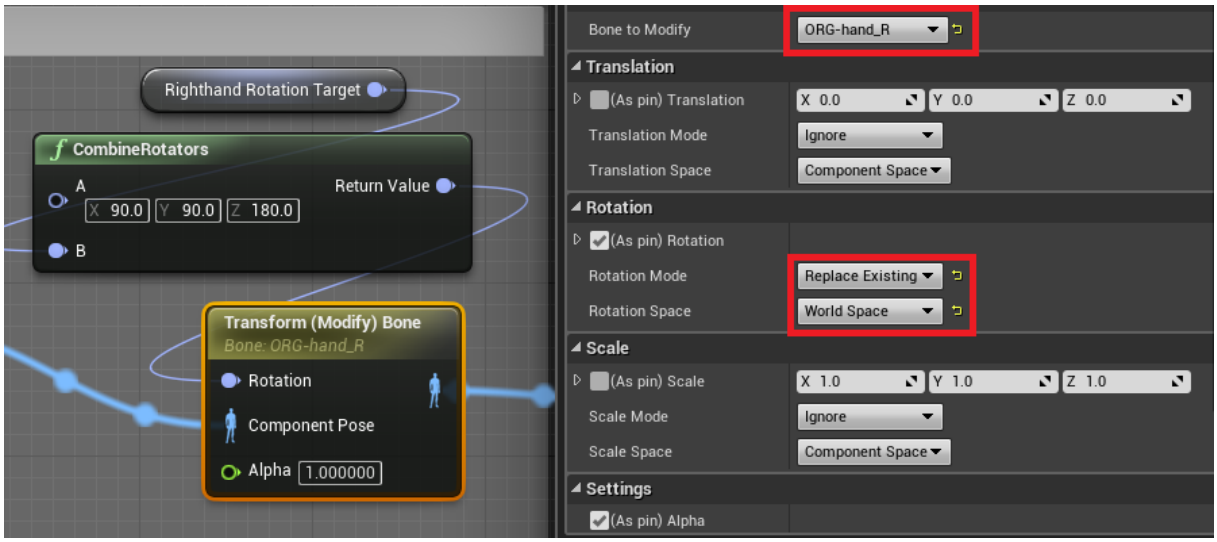


Image 22: Settings for the node that rotates the mesh hand according to motion controller

results in most cases.
Next, we must set the bone to which this node should apply. This is done via the dropdown menu next to "IKBone" in the node's details panel.
In the same tab we also need to define the location space for this change. For our purposes, we want to set this to "World Space", since that is the reference frame we obtain our motion controller location in.
In the "Joint Target" tab we then need to determine at which bone the mesh will bend. There are a few possible settings here that depend on which "Joint Target Location Space" is selected. Here, we select "Parent Bone Space" and define the same bone used for "IKBone" at the top to be our "Joint Target Space Bone Name".
This means that our arm will bend at the parent of our hand, which in our case will be the bone positioned at the elbow.

This node will set our hands position and adjust the arm as well, but we also want to rotate the arm according to the controller input. To do this we must append a "Transform (Modify) Bone" node.

This is the same node used earlier to reposition bones to adjust the character height. In this case though, we will turn off scale and rotation, and select only rotation as a property to modify (via the checkboxes in the details panel).
Again, we must specify the bone for this node at the very top.
Under the "Rotation" tab, we want the "Rotation Mode" to be set to "Replace Existing" and the "Rotation Space" to be "World Space". This is simply due to the fact that we get our controller input in world space, and want to set the absolute rotation to match our input.
The only remaining thing is to pass our rotation to the node as input. As can be seen in image 22 we rotate our hand by a fixed rotation around x, y and z

processing it. This is simply due to the fact that each bone has its own coordinate system, and for the rotations to match up, we must make sure that the coordinate system of our bone matches that of the controller. The rotations needed to match these up, if needed, will always depend heavily on the individual character.

To get our final arm position, we still need to adjust for one more aspect: arm length.

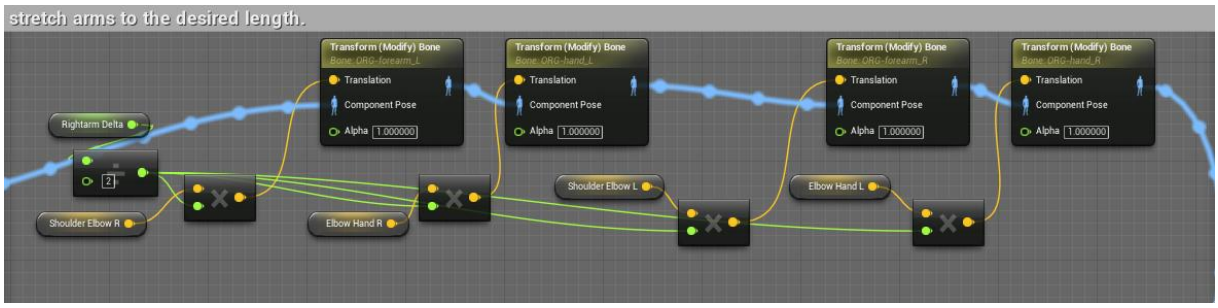


Image 23: Entire node sequence to stretch the arms

The "Left-/Rightarm_Delta" variables hold the difference (=delta) between the meshes default arm length and the players arm length, "Shoulder_Elbow_L/R" holds the normalized vector pointing from shoulder to elbow, and the "Elbow_Hand_L/R" variables save the same for elbow to hand.
To adjust the length we will use the same approach as with the mesh height. Using the "Transform (Modify) Bone" node, we will reposition the elbow and the hand. To do so we first halve the delta lengths, since we want to split the stretching among the forearm and the upper arm. Then, we multiply that value with the corresponding vector (i.e. hands will be moved along the "Elbow_Hand" vectors, and elbows along the "Shoulder_Elbow" vectors).
The resulting vector is than passed to the node, which is set to add the input to the existing position in its "Translation Mode" (as opposed to replacing it).

4. The last part of the mesh we need to adjust is the head. We want the head to rotate with the HMD to the character to create a realistic representation of the users view direction. Ultimately, the process of rotating the head correctly is equivalent to the rotation part of the hands.

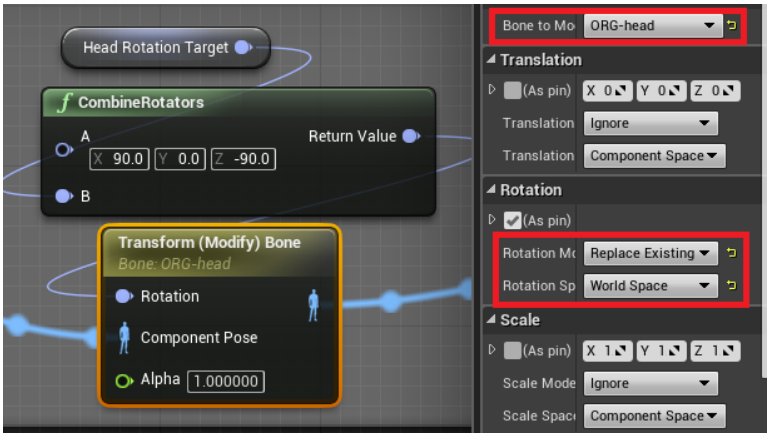


Image 24: Rotating he head according to the HMD

Using the "Transform (Modify) Bone" node we set the rotation to be done in world space, while replacing the existing rotation.

The only difference to the hands is of course the bone we set this node to apply to, as well as the input parameter. The entirety of the node settings can be seen in image 24.

Creating the Pawn

Now that we have our mesh imported and an animation blueprint that targets the corresponding skeleton, we can create our pawn.

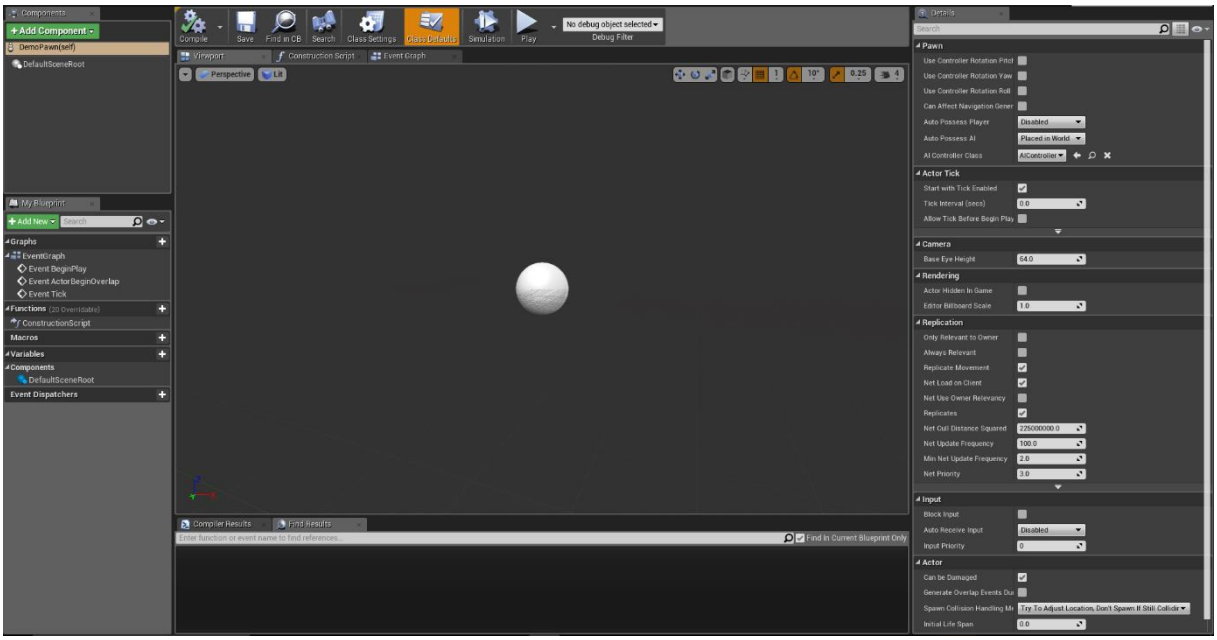


Image 25: Empty pawn blueprint screen

To do this we must first instantiate an empty blueprint. When doing so, we are presented with a choice of which type of blueprint we want. For our purposes, we will need a “Pawn”.
We are then presented with the empty default pawn blueprint, as seen in image 25.
So now we must attach all the components we need to our pawn. This is done in upper left “Components” tab, by clicking the “AddComponent” button and choosing which type of component to add.
We will need a few components. Image 25 shows the resulting hierarchy after adding all required components to the pawn. Additionally, we will need to adjust some settings in the details panel of the newly added components. The settings are as follows.

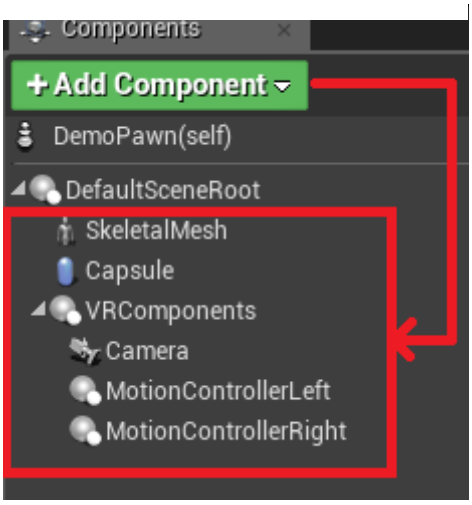


Image 26: Required components in the pawn blueprint

SkeletalMesh:
This is a “SkeletalMesh” component from UE4. This will store the mesh we imported earlier. To define the character it will use, we must set the “Mesh” variable in its details panel to the mesh we have previously created. This is done through the dropdown panel

on the right-hand side, which will list all eligible meshes in the project.
Furthermore, under the "Animation" tab, we can define the type of animation asset used in conjunction with the pawn. Of course, we want to use our previously created animation blueprint. Thus, we must first set the "Animation Mode" to "Use Animation Blueprint". After that, another dropdown menu will appear in the tab, called "Anim Class". Here we will again be presented with all possible animation blueprints in the project, and will choose the one we have previously created (note that the names in the list will always have a "_C" appended to the name given to the blueprint, as seen in image 27).

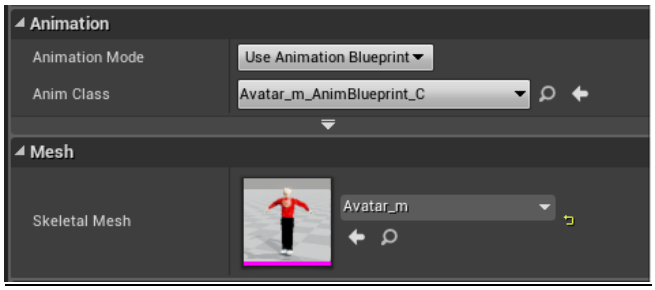


Image 27: Animation settings in the pawn blueprint to ensure the use of the previously defined animation blueprint

Capsule

The capsule component is a bounding volume. It is in theory an optional addition, but will provide some collision detection when objects in the scene collide with the avatar's body, thus it is still added.
Depending on the size of the character mesh, settings here may vary. The only 2 variables in need of adjustment, however, are the "Capsule Half Height" and "Capsule Radius" under the "Shape" tab of the capsule's details.
The Unreal Engine 4 documentation, however, provides a general suggestion to specify setting the "Capsule Half Height" to 96, and the "Capsule Radius" to 16.

VR Components

The type of this object is a "SceneComponent". Its function is mainly to group the 3 tracked objects under a common parent, giving them all the same reference frame. According to the Unreal Engine documentation, and in combination with the settings of the capsule, the only adjustment we need to make with this scene component is to set its Location to (0, 0, -110). This results in having the object at ground height, when looking at the pawn later in a scene.

Camera

The camera is created as a child object of the "VRComponents", and is of type "Camera". For the most part it is a default camera object. The only thing we need to explicitly turn on is the "Lock to HMD" variable in the camera's "Camera Settings" tab. This will make the camera automatically follow the HMD position when one is detected.

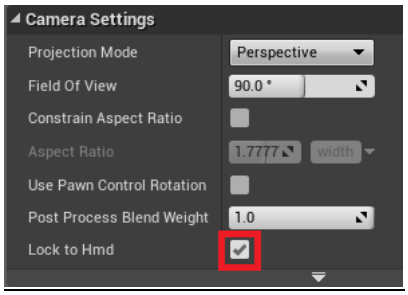


Image 28: Camera setting required to have component automatically move with HMD

MotionControllerLeft/-Right

Similar to the camera, these two components of type "Motion Controller" will automatically attach to motion controllers and move accordingly. The only thing that needs to be set up is the definition of which component shall correspond to the right hand, and which one to the left. To do so, we can simply use the "Hand" variable under the "Motion Controller" tab of the motion controllers, and make sure to set one of them to "Left" and the other to "Right".

Image 29: Motion controller object settings

Pawn Logic

The following sections will describe the setup in the pawn blueprint which, in conjunction with the animation blueprint, will achieve the overall behavior of the avatar. In the following sections, not every variable and function will be explained in detail. Rather, only the most relevant concepts necessary to understand the overall implementation will be described.

The logic we will implement will mostly be in the "Event Graph" of the avatar's (pawn) blueprint, which we have just created. In some instances, we will create a "function" for certain tasks to reduce clutter in the event graph and make it overall more readable. So, unless stated otherwise, the following blueprint events will be part of the "Event Graph".

Setting Morph Targets

Image 30: Setting a morph target in blueprints

Through the correct mesh creation and import, we now have a pawn that uses a mesh which has shape keys defined for it. Unreal Engine 4 refers to shape keys as "morph targets". Since we want to be able to change the value of these shape keys/morph targets, we need to implement functionality in the pawn that refers to its own mesh, and specifically targets the incorporated morph targets of it.
We can do this via a built-in function called "Set Morph Target" (see image 30). We define a new custom event in the event graph that takes a float value as input. Then, that event calls the function "Set Morph Target" and passes the value to the function. We must specify the object that will have the morph target, which needs to be a reference to our mesh.
To define which morph target we want to change, since we have created several different ones, we simply refer to it by name. The names of the morph targets were defined in the modeling software, and can be seen in the engine by opening the imported mesh.
We then just go through this process for every morph target our mesh has.

Changing Texture

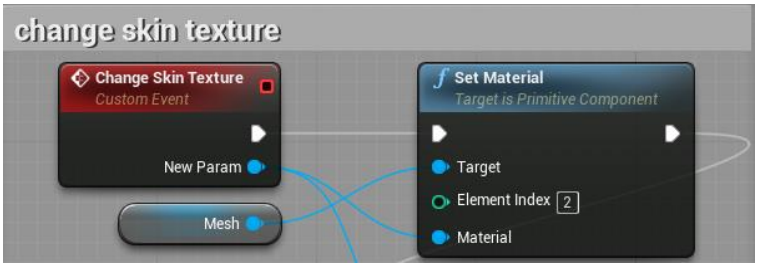


Image 31: Changing a texture in blueprints

The second core functionality regarding the customization is changing the textures of the mesh to adjust skin color and attire.
The technical implementation of this works similar to the morph target setting. We create a custom event for every separated region that has its own texture associated with it. The event takes a "Material Interface" as parameter. The event then calls the "Set Material" function on the mesh, and passes the material instance directly to the function.
The one thing left to set is the "Element Index". When a mesh has numerous textures associated with it, then they are all accessed via their specific material index. So when we set the material, we must specify the material index of the texture we want to change. By opening the skeletal mesh we have imported we can see all the different materials associated with a mesh, as well as their indices. So, depending on which texture we want to change (in image 31 we see the example for changing the skin

texture) we must adjust the “Element Index” of the function. That also means that we must create such a custom event for every texture we want to be able to change.

Setting Animation Blueprint Variables

As we have seen earlier, the animation blueprint relies on a number of variables which have previously not been set anywhere. That was because all those variables are set from outside the animation blueprint. Specifically, they are set in the event graph of the pawn blueprint.
As a reminder, the variables needed in the animation blueprint are location and rotation of both motion controllers, rotation of the HMD as well as the amount that the mesh needs to be stretched/shortened

For this purpose, a new custom event is created, called “Update AnimBP”. In this event we need to first retrieve a reference to our animation blueprint. We can do so by calling

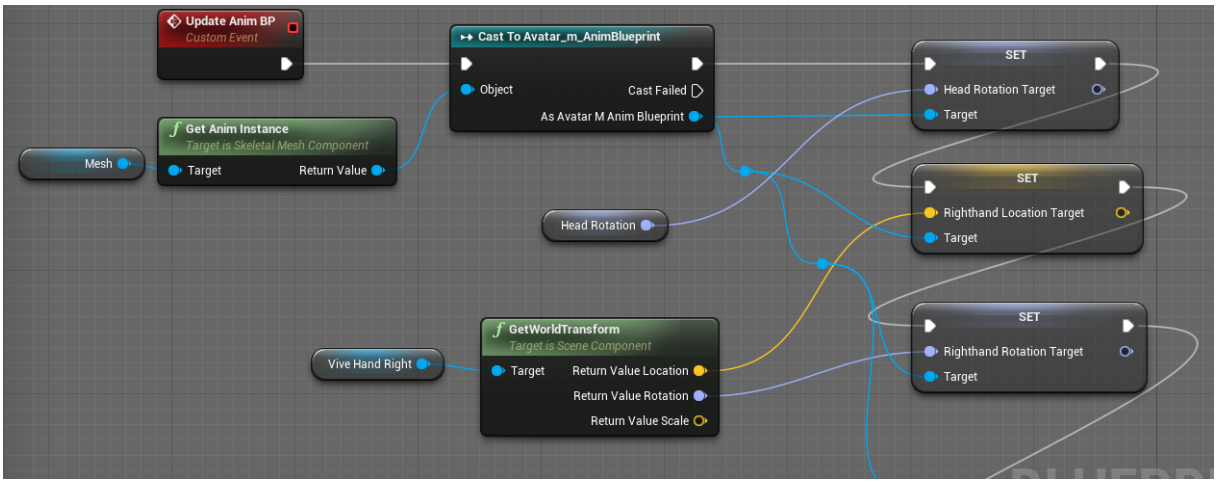


Image 32: Setting animation blueprint from pawn blueprint

the mesh’s “Get Anim Instance” function. To be able to call its functions and variables, though, we will then have to set the return value to our animation blueprint type (see image 32).

Now that we have access to our character’s animation blueprint, we can also access the variables inside of it. We make use of that fact by successively setting those variables to the appropriate values. That is, we set the “Head Rotation Target” variable of the animation blueprint to the rotation of the camera in the pawn (which corresponds to the HMD position), followed by setting “Righthand Location Target” to the location of the right-hand component and so forth.

Since these variables need to be updated continuously in the animation blueprint for the arms to be correctly tracked and displayed, this “Update Anim BP” event we have created needs to be called in the pawn’s “Event Tick” function.

The final variables needed to be set in the animation blueprint are “AdjustBoneHeight” as well as the deltas and vectors for the arm length.
For the bone height variable, setting it is done in 4 steps:

1. Get the height of the HMD

2. Get the height of the mesh
3. Calculate the difference between those two
4. Set the variable to the resulting difference

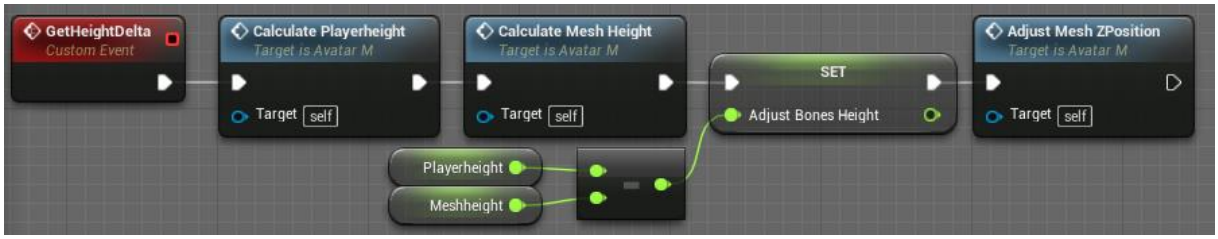


Image 33: Custom event to set the amount the avatar needs to be stretched and repositioned

As can be seen in image 33, after calculating and setting the delta height of the player and the mesh, there is another function call to an event called “Adjust Mesh Z Position”. This is another custom event we have to create, for one specific reason: After stretching or shortening the mesh, it will not stand on the ground anymore.
Thus, depending on the amount we adjust the mesh height, we must also reposition the mesh along the Z axis (= “up” axis).

To do so, we take the “Adjust Bones Height” variable, and reposition the mesh by a portion of that variable. We reposition by a portion of the variable because positioning the mesh onto the floor is only affected by the repositioning of the feet, thus we want to adjust only by that amount.

When it comes to the delta lengths of the arms, we take a similar approach:

1. Get the current position of the motion controller
2. Get the position of the mesh hand
3. Get the difference between those two

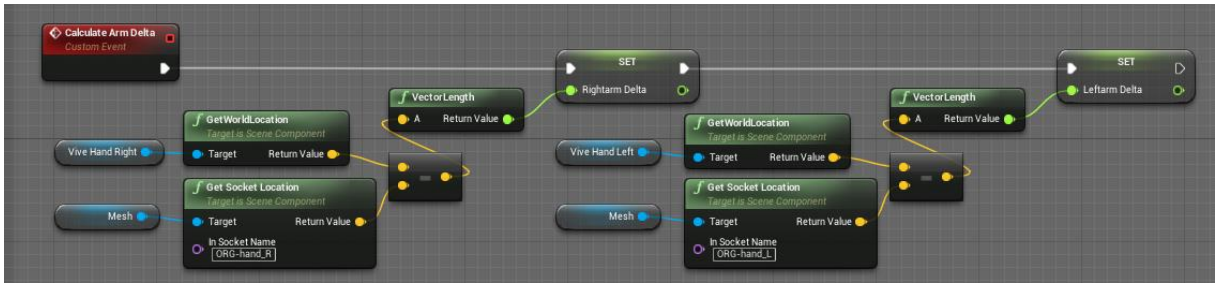


Image 34: Custom event to calulate distance the arms need to be repositioned on the mesh

This is done for both the left and the right arm. These values will then be set in the animation blueprint. What is still missing is the vectors that point from shoulder to elbow, and elbow to hand. This will be done in the “Update Anim BP” event (see image 35).
The positions of shoulder, elbow and hand are all queried, and two vectors are created from it. They are then normalized before being passed into the animation blueprint.

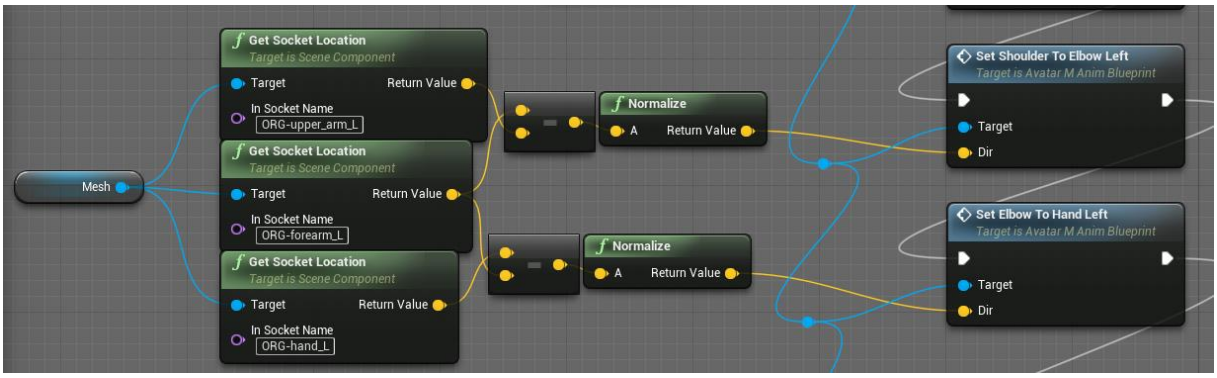


Image 35: calculating the directions into which the arms need to be stretched

Finally, we bind these functions to be called by pressing a button on the motion controllers (in this case, the "face 3" button of the right motion controller, which is the bottom button of the touch pad).

Rotate the Body

Rotating the body will require some more room in the blueprint, and for clarity sake all the logic will be put into a separate function.

In that function we will successively do a number of things to reach our end body transform.

1. Position the pawn according to the HMD position.

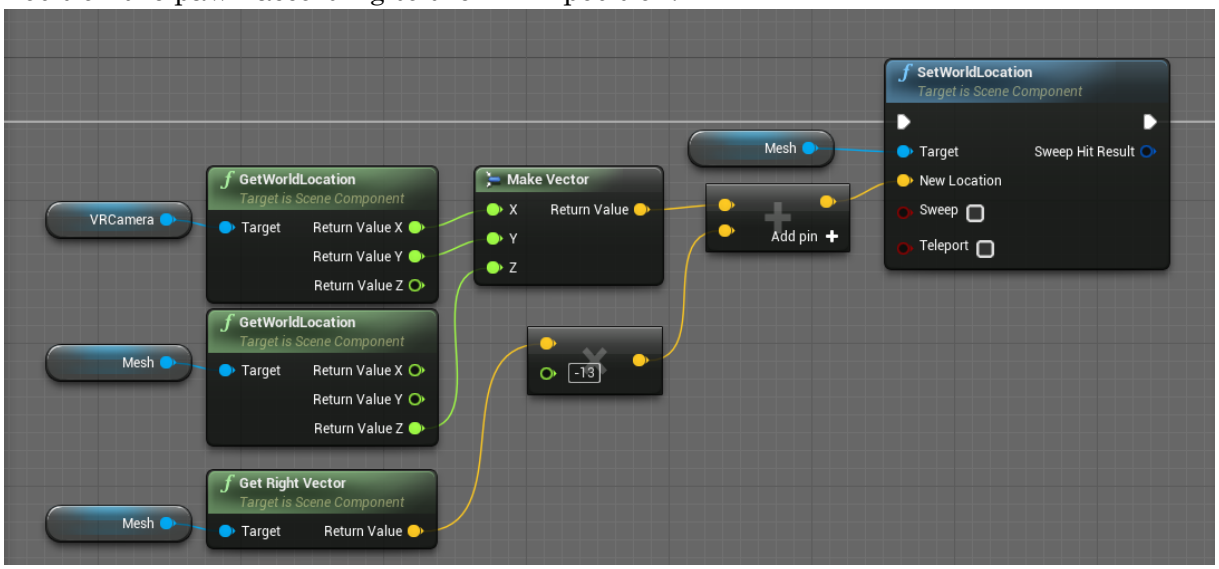


Image 36: Repositioning the mesh according to HMD

To do so we will retrieve the position of the HMD, and place the X and Y position of the mesh to match the HMD. The Z position (height) will remain the same. We will then adjust the position of the mesh to be slightly behind the HMD. In the example in image 36 it is moved back 13 units, though this will vary depending on the mesh.
We do this so that the HMD is at the very edge of the mesh, instead of center. That way, the camera will be very slightly in front of the mesh, instead of inside the mesh looking outward.

2. Rotate the body when the head rotates too far to either side.

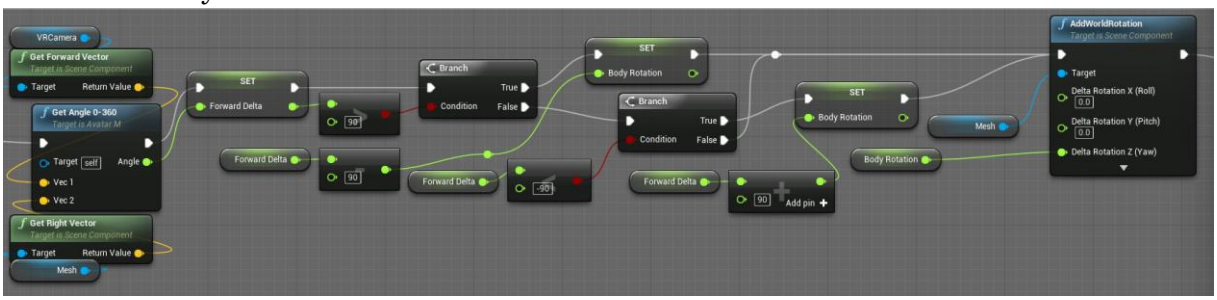


Image 37: Rotating body when head is rotated more than 90 degrees in either direction

We hereby check the angle between the camera and the mesh body on the X-Y-plane. If the angle is >90° or <-90°, then we assume the player is rotating and we rotate the body to the point where the angle is at +/- 90° exactly.

3. Rotate the body to align with the head when the HMD's forward vector and the hands forward vector (both projected onto the X-Y plane) are within a certain

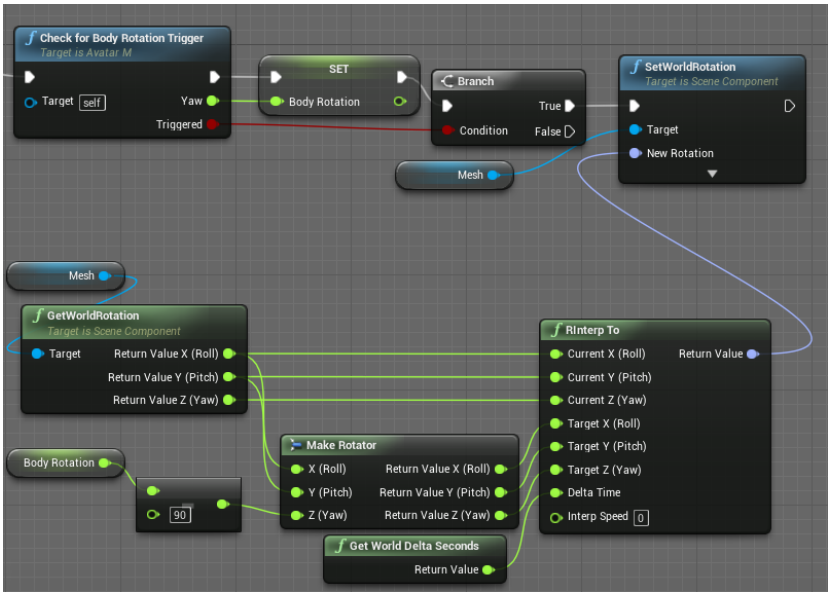


Image 38: Reorient body when arm forward vector and head forward vector nearly align

margin of each other.
This means that we will calculate the vector that connects the two hands, and create the vector perpendicular to it pointing forward.
If the angle between that vector just created and the camera (again, projected onto the X-Y plane) is below 10°, then we LERP the body's X&Y rotation to match that of the camera.

By this point, we now have a pawn that can adjust in height, moves its head and arms with the player input, rotates its body plausibly and moves in its entirety according to the HMD position.
What is still missing is giving the player access to the customization functionality that we have already set up.

Connect Logic to UI

The final thing we must do is create a UI for the player, so that he can make use of the customization options we have prepared earlier.

First, we will create the UI objects. To do so we must instantiate a widget blueprint. The widget blueprint option can be found under the "User Interface" tab when right clicking in the content browser (see image 40).

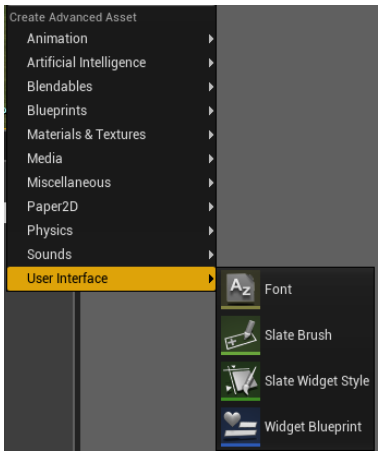


Image 40: Menu navigation to create new widget BP

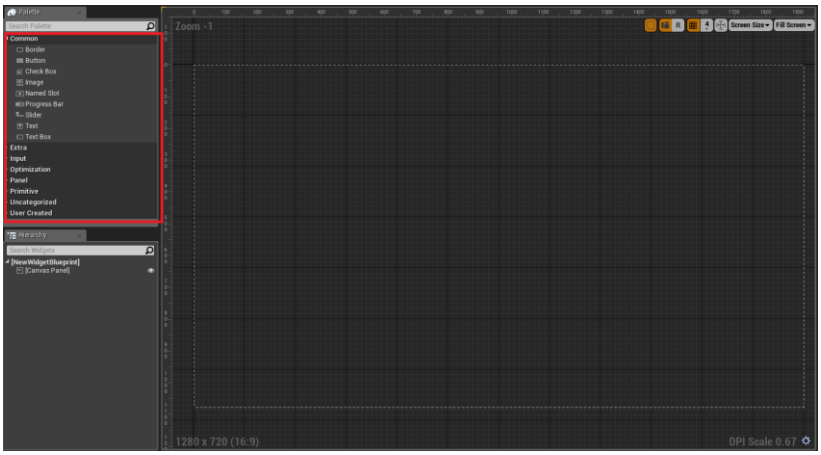


Image 39: Empty widget

We are then presented with an empty UI widget, as seen in image 39. There, I have highlighted the "Common" tab on the left. From there, we can drag and drop any UI elements we need onto the canvas and place them to our liking.

For this project we have made 2 widgets, one for the textures and one for the morph targets. We will take a look at both and how they work:

1. First, we have the texture widget. Here, we have placed text specifying the textures that can be changed, and next to it some buttons that have the colors which can be chosen. This could be any texture, for this work we have simply left it at plain-colored textures.

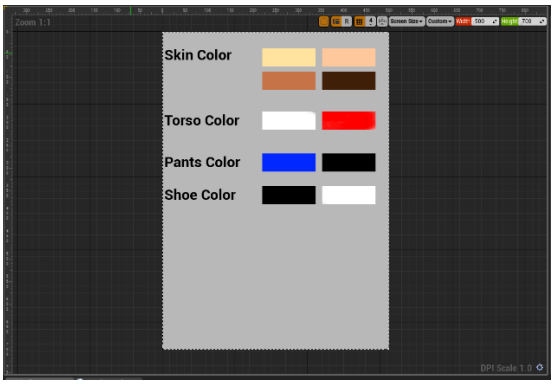


Image 41: UI created to change textures

2. For every button, one can define events that should happen on certain triggers. For example, when the button is clicked.
For each button we create, we will define a function for the "OnHovered" event. In image 42 we can see the functions for changing the skin color. When the event is triggered, we first get the player pawn and cast it to our pawn blueprint class. After that, we can access the functions we have previously defined. In this case, we call the "Change Skin Texture" of the avatar blueprint. Since we have defined a material instance to be passed to the function, when calling it we are presented with a dropdown menu that will show us all available materials we can pass.
From here it is only a matter of choosing the correct material to pass for each button.
This function is created for every button in the widget.

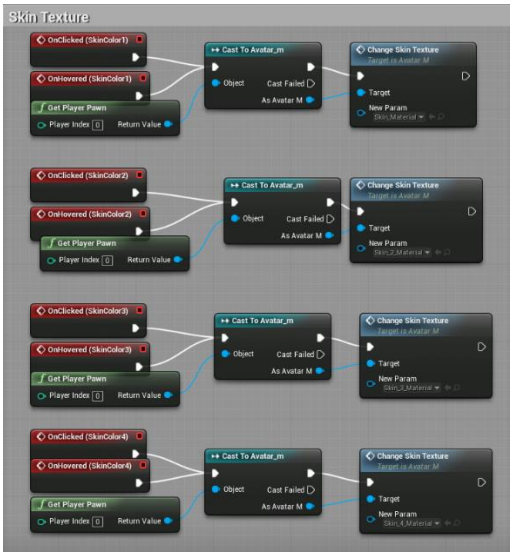


Image 42: Texture changing blueprint functions

3.
4. The second widget we need to create will adjust our morph targets.
In image 43 we can see the widget used for the morph targets.
Here, instead of buttons we control the value via a slider. The slider is set to a range from 0 to 1 via the settings in its details panel. Similar to the buttons, we can create functions that will trigger on certain events for the sliders. The event needed this time is called "OnValueChanged". For every slider we have created, we also create a function for this event.
One of the events can be seen as an example in image 44. As with the buttons earlier, we must first obtain the player pawn and cast it to our specific pawn blueprint to get access to our functions. We can then simply call the function that corresponds to the slider we have created. The "OnValueChanged" function will give us the new value of the slider as an input, and thus we can take that value and pass it though to our custom pawn function we have created earlier, which will then set the mesh's morph target to match the new value.
For every slider we have created, we need to define the corresponding

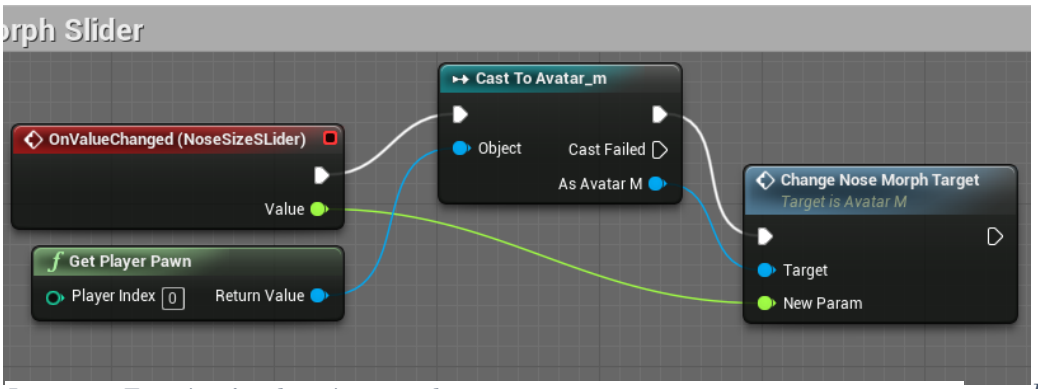


Image 44: Function for changing morph target

Image 43: UI for changing morph targets

"OnValueChanged" event.
Now that we have the widgets, we must create an actor that will hold them in order for them to be placed in a 3D scene.
So for every widget, we must instantiate a new blueprint actor, add a "Widget" component to it, and set that widget component to be the "User Widget" just created (see image 45).

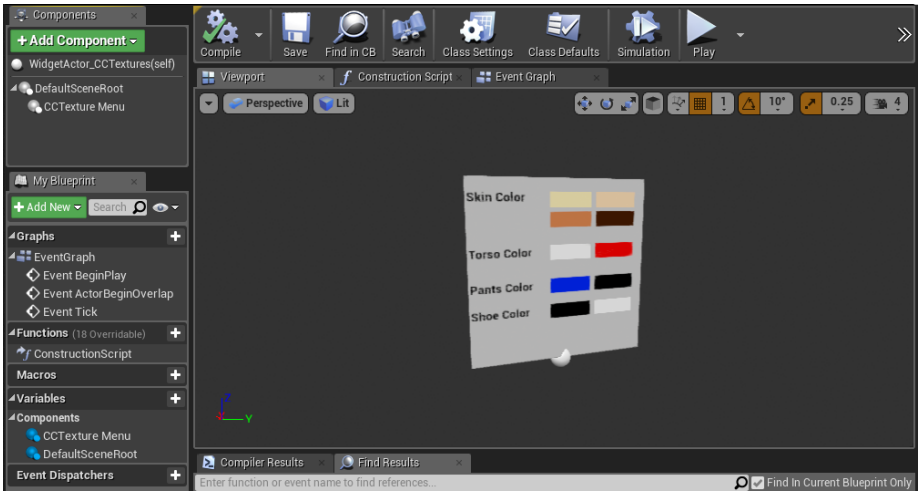


Image 45: Setting the widget in the widget actor

The last thing we need to do is give the player a way to access the 3D widget and interact with it.
For that, we need to add some more components to the pawn. We need to add a "Widget Component" and a "Widget Interaction Component". In the pawn hierarchy, we place the widget component under the left motion controller, and the widget interaction component under the right motion controller. The resulting hierarchy can be seen in image 46, with the two new components highlighted.

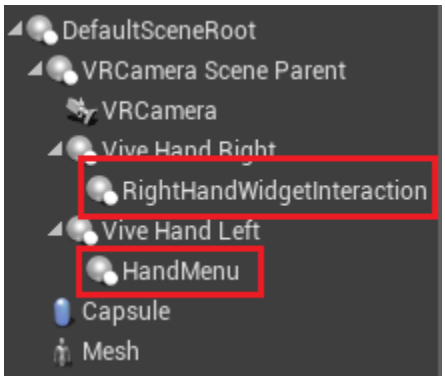


Image 46: New component hierarchy

As with the other components so far, we have to make sure the settings of the new components are correct. Luckily, there are only very few adjustments that need to be made with these two. The widget component under the left hand will need to be repositioned to a point where the menus we have created will be easily visible. This is a trial and error process that is highly subjective, and will differ from mesh to mesh.
The widget interaction component will also need to be repositioned. We want it to be placed at the tip of the right index finger.
Then, under the "Interaction" tab, we want to set the "Interaction Distance" very low. Here I have set it to 3.

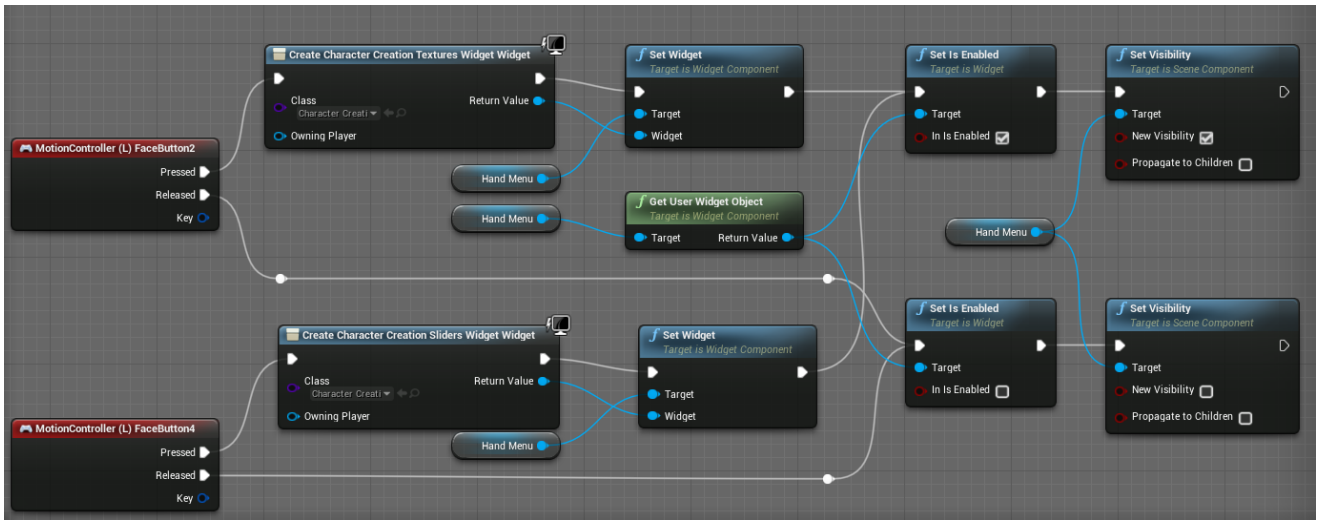


Image 47: Triggering visibility of widget actors depending on player input

Lastly, in the pawn's "Event Graph" we will have different input from the left controller toggle either of the previously created widget actors by instantiating the desired widget and setting the widget component on the left hand to this new object (see image 47).

Desktop Version

As stated earlier, we want to enable customization on a desktop as well. I mentioned that this does not need many changes when starting from a VR-oriented solution. So in this chapter I will introduce the changes that need to be made to make customization possible in desktop mode as well.

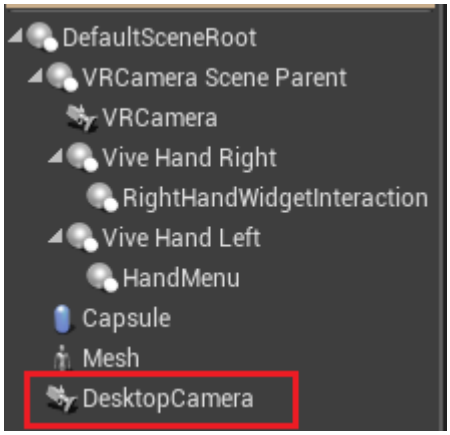


Image 48: Adding another camera to component hierarchy for desktop use

First of all, when on a desktop, we need to have a full view of the avatar, instead of placing the camera in a way to convey us inhabiting the character.
Therefore, we will add another camera to the pawn blueprint (see new hierarchy in image 48). This camera will be placed in such a way that it will have a full-frontal view of the avatar.

Once the camera is placed, we will add some logic that will determine which camera to use in any given situation. To do that we create a new custom event in the pawn's event graph, called "Activate Cameras". This function will check for an HMD. Depending on the return value, we will simply turn one of the cameras on and the other one off.

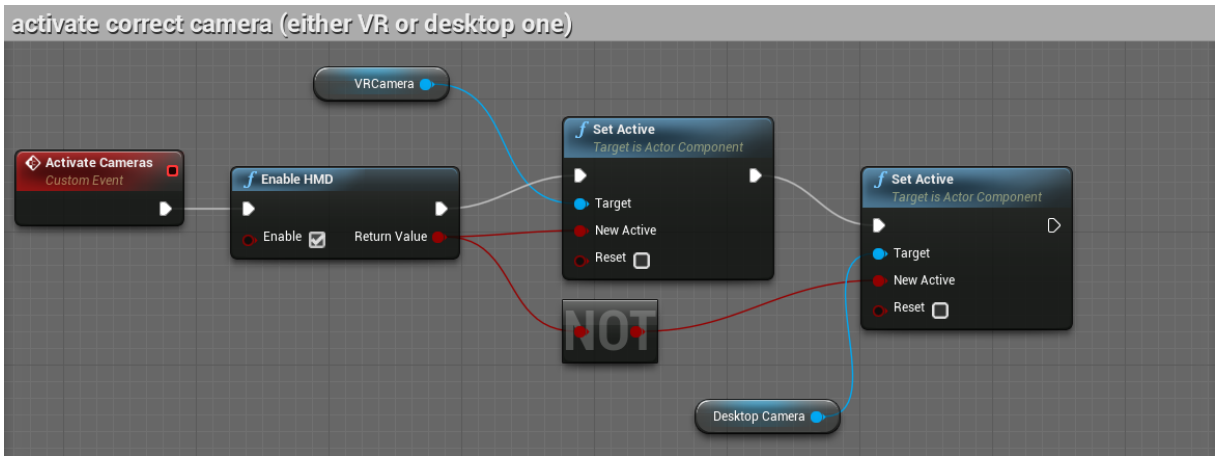


Image 49: Toggle correct components , depending on mode (VR or desktop)

Next, we must change the UI for desktop use. Luckily, we can reuse all the functionality and layout of the two widgets we have created earlier.
All we need to do is create a new widget, and copy everything from the existing two widgets over to this new one.
We can reposition the components on the canvas until we have a layout we are happy with, and save it. The resulting desktop widget of this project can be seen in image 50.

Image 50: Widget used for desktop

Now that we have created this new UI, we have to make sure to use it when, and only when, we are in desktop mode. To do so we will add some functionality to the level blueprint of the customization level.

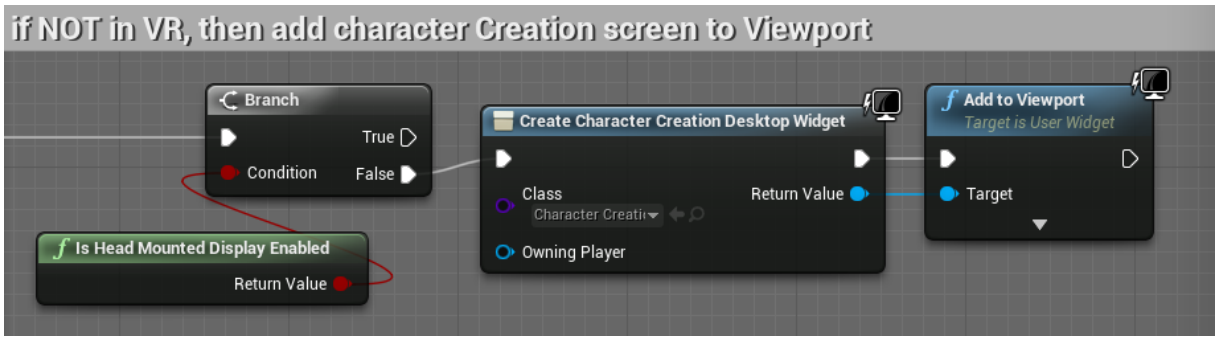


Image 51: Use new desktop widget UI if not in VR, toggle at start

We again check for an HMD, and if there is no HMD connected then we create an instance of our desktop widget and add it to the viewport.

From this point on, further additions are merely quality-of-life improvements for desktop use, such as making sure the mouse cursor is visible when in desktop mode or the possibility to rotate the mesh through keyboard input. These and any other additions are optional and thus not explicitly detailed in this work.

References

[1] M. Zuckerberg, "facebook," 25 march 2014. [Online]. Available: https://www.facebook.com/zuck/posts/10101319050523971.

[2] "statista," 2017. [Online]. Available: https://www.statista.com/statistics/426276/virtual-realiy-revenue-forecast-worldwide/.

[3] "venturebeat," 7 december 2016. [Online]. Available: http://venturebeat.com/2016/12/07/despite-slow-2016-vr-is-still-forecast-to-hit-14-billion-in-consumer-software-sales-in-2020/.

[4] Oculus VR, LLC., "Oculus," 2017. [Online]. Available: https://static.oculus.com/documentation/pdfs/intro-vr/latest/bp.pdf.

[5] M. S. M. Usoh, "an exploration of immersive virtual environments," 1995.

[6] Barfield, "Presence and performance within virtual worlds," 1995.

[7] M. Slater, "Measuring Presence: A response to the witmer and singer presence questionnarire. Presence : Teleoperators & Virtual Environments," 1999.

[8] Holobar, "a distributed virtual reality based system for neonatal decision making training," 2006.

[9] P. G. David England, "Temporal aspects of interaction in shared virtual worlds," 1998.

[10] Pollick, "InSearch of the Uncanny Valley," 2009.

[11] Steuer, "Defining Virtual Reality: Dimensions Determining Telepresence," 1992.

[12] H. J. R. M.A.Gigante, Virtual Reality Systems, 1993.

[13] Z. Peng, "Vergleich von 3D Game Engines," 2010.

[14] C. Anthes, "Virtual Reality Lecture 2016 (Introduction lecture)," Munich, 2016.

[15] Zielinski, "Exploring the Effects of Image Persistance in Low Frame Rate Virtual Environments," 2015.

[16] "digitaltrends," 16 october 2016. [Online]. Available: http://www.digitaltrends.com/virtual-reality/oculus-rift-vs-htc-vive/.

[17] M. Abrash, "http://media.steampowered.com/apps/abrashblog/Abrash%20Dev%20Days%202014.pdf," 2014.

[18] a. cruz, "can presence improve collaoration in 3D virtual worlds?," 2014.

[19] A. Binstock, "Oculus Blog," 15 may 2015. [Online]. Available: https://www3.oculus.com/en-us/blog/powering-the-rift/.

[20] E. M. Kolasinksi, "Simulator Sickness in Virtua Environments," 1995.

[21] M. Slater, "Place illusion and plausibility can lead to realistic behavior in immersive virtual environments," 2009.

[22] T. C. W. Barfield, "Fundamentals of Wearable Computers and Augmented Reality," 2000.

[23] N. Pollard, "VR lecture, Carnegie Mellon University," 2004.

[24] Y. Boger, 2 june 2014. [Online]. Available: http://www.roadtovr.com/overview-of-positional-tracking-technologies-virtual-reality/.

[25] C. Anthes, "LMU VR lecture "Tracking and Input Devices"," 2016.

[26] L.Rosenblum, "The Virtual Reality Responsive Workbench: Application and Experiences," 1997.

[27] Virtual Reality Society, [Online]. Available: http://www.vrs.org.uk/virtual-reality-environments/cave.html.

[28] D. K.Keller, "What is it in Head Mounted Displays (HMDs) that really make them all so terrible?," 1998.

[29] C. Anthes, "LMU VR lecture "Stereoscopy and Output"," 2016.

[30] S. Buckley, "gizmodo," 19 may 2015. [Online]. Available: http://gizmodo.com/this-is-how-valve-s-amazing-lighthouse-tracking-technol-1705356768.

[31] HTC Corporation, "HTC Vive user guide," 2016. [Online]. Available: http://www.htc.com/managed-assets/shared/desktop/vive/Vive_PRE_User_Guide.pdf.

[32] Dassault Systemes 3DExcite GmbH, "3dexcite," -. [Online]. Available: http://www.3dexcite.com/de/angebot/software/author/3dexcite-deltagen.

[33] Autodesk Inc., "autodesk," 2017. [Online]. Available: http://www.autodesk.de/products/vred/overview.

[34] ESI Group, 2017. [Online]. Available: http://virtualreality.esi-group.com/.

[35] Epic Games Inc., "UE Documentation," 2017. [Online]. Available: https://docs.unrealengine.com/latest/INT/Gameplay/Networking/Actors/RPCs/.

[36] Epic Games Inc., 2015. [Online]. Available: https://wiki.unrealengine.com/Network_Replication,_Using_ReplicatedUsing_/_RepNotify_vars.

[37] P. S. a. A. Mahanti, "Observations on Round-Trip Times of TCP Connections," 2006.

[38] "clumsy net limiter," -. [Online]. Available: https://jagt.github.io/clumsy/.

[39] M. Fabri, "the emotional avatar: non-verbal communication between inhabitants of collaborative virtual environments," 1999.

[40] J.-L. L. a. M. E. Latoschik, "Avatar anthropomorphism and illusion of body ownership in VR," 2015.

[41] G. M. Lucas, "Do avatars that look like their users improve performance in a situation?," 2016.

[42] M. Slater, "Transcending the self in immersive virtual reality," 2014.

[43] D. Chung, "Influence of avatar creation on attitude, empathy, presence, and para-social interaction," 2007.

[44] D. Freeman, "Height, social comparison, and paranoia: an immersive virtual reality experimental study," 2013.

[45] D. Anguelov, "SCAPE: shape completion and animation of people," 2005 .

[46] A. Feng, "Avatar Reshaping and Automatic Rigging Using a Deformaable Model," 2015.

[47] A. Shapiro, "Rapid avatar capture and simulation using commodity depth sensors," 2014.

[48] Wang, "Accurate Full Body Scanning from a Single Fixed 3D Camera," 2012.

[49] Tong, "Scanning 3D Full Human Bodies Using Kinects," 2012.

[50] Microsoft, 2017. [Online]. Available: https://msdn.microsoft.com/en-us/library/jj131033.aspx.

[51] Occipital Inc., 2017. [Online]. Available: http://structure.io/support/what-are-the-structure-sensors-technical-specifications.

[52] S. Lim, "The Effect of Avatar Choice and Visual POV on Game Play Experiences," 2006.

[53] A. Vasalou, "Constructing my online self: Avatars that increase self-focused attention," 2007.

[54] J. Lanier, You are not a gadget:A manifesto, 2010 .

[55] Oculus VR, LLC, "Oculus Blog," 9 november 2016. [Online]. Available: https://www.oculus.com/blog/a-closer-look/.

[56] Morph 3D, "morph3d," 2015. [Online]. Available: https://www.morph3d.com/.

[57] A. P. a. B. Pease, "The Definitive Book of Body Language: How to read others' attitudes by their gestures," in *The Definitive Book of Body Language: How to read others' attitudes by their gestures*, 2007.

[58] M. Mori, "translation of original paper regarding uncanny valley," 12 june 2012. [Online]. Available: http://spectrum.ieee.org/automaton/robotics/humanoids/the-uncanny-valley.

Images