FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

VRNet - A framework for multi-user Virtual Reality

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Michael Leopold Wagner, BSc.
Matrikelnummer 00827376

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Priv. Doz. Mag.rer.nat. Dr.techn. Hannes Kaufmann
Mitwirkung: Iana Podkosova, BSc MSc

Wien, 25. Jänner 2019

Michael Leopold Wagner	Hannes Kaufmann

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

VRNet - A framework for multi-user Virtual Reality

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Michael Leopold Wagner, BSc.
Registration Number 00827376

to the Faculty of Informatics

at the TU Wien

Advisor: Priv. Doz. Mag.rer.nat. Dr.techn. Hannes Kaufmann
Assistance: Iana Podkosova, BSc MSc

Vienna, 25th January, 2019

Michael Leopold Wagner | Hannes Kaufmann

Erklärung zur Verfassung der Arbeit

Michael Leopold Wagner, BSc.
Veitscher Straße 29/5, 8663 Skt. Barabara im Mürztal

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 25. Jänner 2019

Michael Leopold Wagner

Danksagung

Ich möchte mich bei meiner mitwirkenden Betreuerin Iana Podkosova bedanken, die mich durchgehend bei der Umsetzung dieser Arbeit unterstützt und ermutigt hat. Weiters möchte ich mich bei meinem Betreuer Prof. Hannes Kaufmann für die Unterstützung und Ermöglichung dieses spannenden Projekts bedanken.

Teil dieser Arbeit wurde im Rahmen eines Forschungsprojekts in der Zusammenarbeit vom CeMM Forschungszentrum für Molekulare Medizin und TU Wien erstellt. Ich möchte mich herzlich bei dem CeMM Team unter der Leitung von Jörg Menche bedanken, besonders auch bei Sebastian Pirch für die gute Zusammenarbeit. Ich bin sehr froh Teil dieses spannenden Projekts gewesen sein zu dürfen und wünsche weiterhin noch viel Erfolg.

Weiters möchte ich mich an dieser Stelle auch bei meiner Familie bedanken, besonders meinen Eltern Leopold und Hermine Wagner, welche mir mein Studium ermöglichten und mich immer unterstützt haben.

Ich bedanke mich auch bei Fernanda und ihrer Familie, sowie allen meinen Freunden und Kollegen die mich während und außerhalb des Studiums unterstützt haben. Besonders Lukas Meindl, der sich bereit erklärt hat meine Diplomarbeit durchzulesen.

Abstract

Multi-user VR applications are challenging to develop and maintain. They are required to be highly responsive to not cause discomfort to users. Potential network issues, like delays and loss of connection, need to be taken into account.

The goal of this thesis is to design and implement a framework to provide developers with the necessary tools to aid networked VR application development. Game engines like Unreal Engine 4 (UE4) offer tools and a development environment to create complex multiplayer 3D applications. However, the underlying networking system was not designed to be used in local walkable VR scenarios. Our approach is to build the VRNet framework on top of the engine to enhance it with important functionalities. VRNet is implemented as an UE4 code plugin that extends core classes in C++. As a plugin it can easily be integrated and used in UE4 projects. It adds capabilities for networked user movement, interaction and representation in VR. Client-authoritative replication is implemented to ensure consistency between machines with optimal user experience, guarded from networking issues. Additionally, interpolation mechanisms are provided to smoothly simulate other clients.

Furthermore, as a proof of concept, VRNet is integrated into a VR application that is developed in the scope of a research project by the CeMM Research Center for Molecular Medicine in cooperation with TU Wien. CeMM Holodeck is an UE4 application for visualization and interaction with biological data-networks. Another goal of this thesis is to extend the functionality of the application to allow multiple users to move around in a shared virtual environment and collaboratively interact with the data-network. The integration of VRNet was successful and showed that VRNet provides a robust and useful foundation for the creation of multi-user VR applications.

Evaluation tests were conducted to investigate the performance efficiency of the VRNet framework, as well as the CeMM Holodeck. The results of the network traffic measurements show that the used bandwidth is manageable even with high network update rates. The results also show that resource usage is not noticeably affected by the integration of VRNet. VRNet offers simplified and efficient development of multi-user VR applications that are performant and provide smooth user experience.

Kurzfassung

Multi-User VR Applikationen sind herausfordernd zu entwickeln und zu warten. Sie haben die Anforderung, dass sie höchst reaktionsschnell sein müssen damit kein Unwohlsein für Nutzer erzeugt wird. Über das Netzwerk können zusätzlich Probleme auftreten, wie Verzögerungen und Verbindungsabbrüche, welche berücksichtigt werden müssen.

Das Ziel dieser Arbeit ist es ein Framework zu designen und zu implementieren, mit dem nötige Werkzeuge um Multi-User VR Applikationen zu entwickeln bereit gestellt werden. Game Engines, wie Unreal Engine 4 (UE4), eignen sich gut um komplexe Multiplayer 3D Applikationen zu entwickeln, jedoch ist das zugrundeliegende Networking-System nicht für lokale Multi-User VR Applikationen ausgelegt. Unsere Methode ist es, VRNet auf der Basis von UE4 zu entwickeln und um wichtige Funktionalitäten zu erweitern. VRNet ist als Code-Plugin konzipiert, welches Kernklassen von UE4 in C++ erweitert und einfach in UE4-Projekte integriert werden kann. Die bereitgestellten Funktionalitäten umfassen die effiziente Übertragung von User-Daten, wie Positionsdaten und Repräsentation, sowie Interaktionen mit der virtuellen Umgebung. Als Kernelement in der Umsetzung wird Client-Authoritative Replikation eingesetzt um die Konsistenz zwischen Maschinen sicher zu stellen, während User bestmöglich vor Netzwerkeinflüssen geschützt werden. Zur flüssigen Simulation anderer Clients und Objekten werden Interpolierungsmechanismen bereitgestellt.

Als zweiten praktischen Teil dieser Arbeit wird VRNet im Rahmen eines Forschungsprojektes in eine VR Applikation integriert um sie vollständig Multi-User-fähig zu machen. CeMM Holodeck ist eine UE4 Applikation zur Datenvisualisierung und Interaktion mit biologischen Datensätzen in Netzwerkform und wird vom CeMM Forschungszentrum für Molekulare Medizin entwickelt. Die Integration von VRNet erwies sich als erfolgreich und zeigte das VRNet eine robuste und nützliche Grundlage zur Erstellung von Multi-User VR Applikationen bietet.

Weiters wurden Evaluierungstests durchgeführt um die Leistungsfähigkeit von VRNet, sowie CeMM Holodeck, zu untersuchen. Die Ergebnisse zur Netzverkehrsanalyse zeigen, dass die benutzte Bandbreite trotz hoher Aktualisierungsrate relativ niedrig ist. Zusätzlich ist kein Overhead in der Ressourcenverwendung bei Integration von VRNet bemerkbar. VRNet bietet eine vereinfachte und effektive Entwicklung von performanten Multi-User VR Applikationen.

Contents

CHAPTER 1

Introduction

Virtual reality (VR) is a continuously growing field in which the scientific community has already been working for decades. VR offers many new possibilities of interaction and communication and has been applied in several domains, with new application fields still emerging. Recent advances in hardware development, including the increase of the available graphics processing unit (GPU) computing power and subsequent appearance of several consumer-priced head mounted displays (HMDs) made VR technology accessible to a broader public. At the same time, better accessibility of the technology has opened potential for collaborative VR scenarios where users can explore virtual worlds together.

Multi-user VR applications are software systems that support multiple users and provide interaction possibilities with each other and with the virtual environment in real time and aim to provide a highly realistic shared sense of space, presence and time [54]. Multi-user VR can be used in applications ranging from entertainment, to industrial design, to systems aimed for training, research and education. It offers capabilities for collaborative and immersive exploration of scientific simulations, making abstract data more tangible and understandable [30].

This thesis focuses on using the VR technology for collaborative scientific visualization. In particular, we consider a VR setup that provides immersive experience through the use of an HMD (HTC Vive [8]) and allows users to navigate a virtual environment by real walking. In this case, navigation by walking is enabled by the Lighthouse position tracking technology that is built into the HTC Vive functionality and provides a room-scale interaction space.

The CeMM Holodeck is an example of a useful collaborative application for scientific visualization. It is currently developed by the CeMM Research Center for Molecular Medicine of the Austrian Academy of Science [3] in cooperation with the TU Wien Interactive Media Systems Group [16]. This application, which is made with Unreal Engine 4 (UE4), allows a user with an HTC Vive to explore and interact with complex

biological datasets in the form of networks, so called *interactomes*. In its initial state, CeMM Holodeck was a prototype application that could be used by only a single person at a time. Our goal was to extend its functionality to support multiple users.

1.1 Problem Statement

High responsiveness is a crucial requirement for VR. The delay between a user's movements and interactions and the resulting visible changes in the virtual environment has to be brought to the minimum. Drops in frame-rate can cause noticeable discomfort in users which has to be avoided at all costs. While this is often already challenging to achieve with today's graphical computing power, introducing additional latency and possible network problems (e.g. loss of connection) of distributed systems further increases the difficulty. A real-time update rate of player positions is also very important to prevent the possibility of collisions when users are walking in the same real space.

On the other hand it is important to assure the consistency of the shared environment on each machine. Proper replication techniques have to be applied to assure quick update distribution for high responsiveness and guarantee that the state of the application stays synchronized [49].

These low latency requirements and the fact that a large amount of user tracking- and object data may have to be distributed can lead to high network throughput. To match the high refresh rates of today's HMDs e.g. the HTC Vive with 90 Hz [8], the network update rate has to be high as well. Techniques for network latency masking, such as client-side prediction and interpolation have to be used, which can prove very challenging to implement for irregular player movements and physics-based movements [29].

Furthermore, the networked nature of multi-user systems adds a lot of complexity to the software development and maintenance process. Especially in VR it is important to be able to do rapid prototyping for experiments.

Game engines like Unreal Engine [18] and Unity [17] are often used for developing desktop-based multi-player computer games. Both engines provide means of including tracking input from VR hardware devices and setting the rendering target to an HMD. The combination of being VR-ready and providing the framework for networking makes them a good base for multi-player VR applications. However, the networking systems of Unreal Engine and Unity are designed to be used in multi-player games played in a desktop environment and possibly over Internet but not specifically in walkable VR scenarios. Therefore, multiple adjustments and enhancements to the built-in networking frameworks of these two game engines are necessary to develop a collaborative application where users can freely move around in VR.

Such enhancements could be provided by a specialized framework on top of the game engine, which adds capabilities for networked user movement, interaction and representation, and which can be easily integrated and used in projects. Apart from

the functional requirements, such a framework has to fulfill certain non-functional requirements, like being highly flexible and extendable, as well as offering the satisfactory performance for usage in VR. UE4 provides extension mechanisms in the form of plugins that can be easily integrated into projects [20]. Therefore implementing such a framework as an UE4 plugin constitutes a suitable approach.

1.2 Aim of the Work

The aim of the thesis is to design and implement a flexible and extendable framework in the form of an UE4 plugin that can be used to develop immersive walkable multi-user VR applications. Together with its UE4 basis it should provide the tools necessary for an easier development process. Furthermore, this plugin will be integrated and used as a foundation for the ongoing scientific visualization project CeMM Holodeck to make it fully multi-user capable.

1.3 Methodological Approach

The methodological approach to achieve the expected results is based on the following steps that will be worked on in an iterative process:

- *Literature review:*
  Background research on multi-user VR and networking architectures will be conducted and used as a basis for the evaluation, analysis and design of the software architecture and implementation.
- *Analysis and design of the software architecture:*
  Requirements of a networking architecture for VR will be analyzed. The software architecture of VRNet will be designed based on requirements of VR scenarios and the evaluation of constraints imposed by the networking conventions of UE4.
- *Implementation of the VRNet plugin:*
  The UE4 plugin according to the established architecture will be implemented in C++.
- *Implementation of the project template:*
  A project template will be implemented, which integrates the VRNet plugin and provides a framework for testing the plugin functionality and a starting point for new projects. This part will be written partly in C++ and UE4 Blueprints.
- *Analysis and design of the CeMM Holodeck networking architecture:*
  The project's initial state will be analyzed to gain a deep understanding of its functionalities. Integration of VRNet plugin functionality will be investigated and the networking approaches for the project-specific functionality will be designed (e.g. navigation, replication of the data-network state and interaction with it).

- *Implementation of the networking of CeMM Holodeck:*
  The VRNet plugin and its relevant functionalities will be integrated. Project specific networking, as previously designed, will be implemented, tested and optimized. This part will be mostly written in UE4 Blueprints.

1.4 Structure of the Work

The thesis is divided into a theoretical and a practical part. The theoretical part includes the analysis of related work in Chapter 2, the presentation of the methodology in Chapter 3 and the overview of the designed software architecture in Chapter 4. The practical part consists of the implementation of the VRNet plugin according to the software architecture presented in Chapter 5 and the integration of the plugin into the CeMM Holodeck prototype and the networking of all its features, described in Chapter 6. Furthermore, the outcomes are presented and evaluated in terms of the previously analyzed requirements in Chapter 7. The thesis is concluded in Chapter 8.

CHAPTER 2

Related Work

Multi-user VR research is closely related to distributed systems research, from which it has taken many research outcomes and concepts. Section 2.1 highlights some of those concepts that are relevant to immersive multi-user VR and this thesis. The state-of-the-art of multi-user VR systems and applications is presented in Section 2.2.

2.1 Distributed Systems

Networking communication architectures range from simple client-server architectures with either a single or distributed servers to completely decentralized peer-to-peer solutions. To ensure the consistency of the distributed system there are commonly two types of *replication protocols* used: Primary-based and replicated-write protocols [56].

Primary-based replication protocols [31] are simple and popular approaches where there exists one primary replica and the other machines hold backups of it. Updates are sent to the primary, which then locally performs the update and notifies the backup replicas to update their values. The primary can either reside on a remote server, or migrate between machines.

The opposite approaches are *replicated-write protocols* where updates can be carried out at several replicas instead of just one like with primary-based replicas [56]. A prominent example of such a protocol is active replication, where the operations are sent to each replica via atomic multicasts that update those replicas at the same order on all computers. Active replication relies on the determinism of the system: all replicas have to produce the same outputs with the same inputs to stay synchronized.

Primary-based replication and a strict client-server architecture is what the Unreal Engine 4 networking framework is built on [20]. While the server is intended to be

authoritative, it is possible to implement a system that changes the primary replica to reside on clients and use the server to multicast updates to other clients.

To assure the important high responsiveness of networked VR applications several techniques are used to minimize the inherent network latency. Dead Reckoning[46] predicts movements and positions of objects and players by extrapolating the next move based on the previous one. It is mainly used to be able to reduce the network update rate and for lag compensation. Another technique for smoothing movements is interpolation between the last updated position and the one updated before that. Clients render other users or objects at the position where they were in the past. The time is equal to their latency plus the amount over which it is interpolating [29].

2.2 Multi-User VR Systems

2.2.1 Early Multi-User VR Systems

First prototypes of multi-user VR systems had existed long before the advancements in hardware development and the increase in available computing power made the recent consumer VR boom possible. For example, the DIVE[32] system was presented in 1993 and provided an architecture and software environment for developing interactive multi-user VR applications. An example created with the DIVE system can be seen in Figure 2.1. The networking architecture of this system was based on the active replication protocol and communicated via peer-to-peer multicast, but evolved to additionally use client-server connections to a name server.[35] Interactions and modifications to the state, which was held in the form of a shared database, were first applied locally and then replicated to all connected peers. The system tolerated an initial inconsistency between the peers but used a service that ensured their equality over time. Additionally it provided mechanisms to divide the virtual world into sub-hierarchies that were only replicated to applications currently interested in them to cut down on network traffic.

Another very similar software platform for implementing multi-user interactive environments that got presented in 1995 was SPLINE [27]. It had a networking architecture similar to the one used in the DIVE system, using a multicast approach mixed with a client-server approach.

Within the scope of the COVEN project [48] the COVEN (Collaborative Virtual Environments) platform was developed, which was based on two separate VR systems. The primary development system was the dVs [53], a commercial VR software system at the time, and was intended as the final delivery system. This system was developed by the British company Division to realize distributed virtual environments where users can interact witch each other in the same virtual world at the same time. As a research system for exploratory prototyping the DIVE was used.

Since then other large-scale multi-user systems got developed, such as MASSIVE-3 [38]. The heart of the system consisted of distributed databases that were logically multicast, but realized as client-server systems. It employed several primary-based consistency

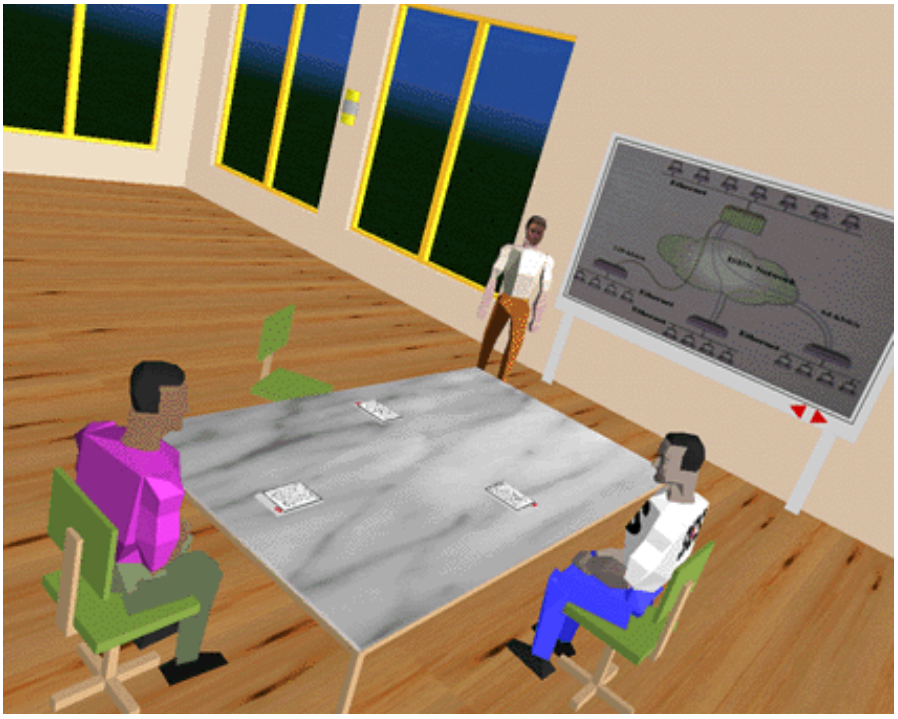


Figure 2.1: DIVE system - a virtual environment with three people simultaneously connected [12].

mechanisms, which could be used for different objects, either data ownership transfer, centralized update or a combination of them. Also, like DIVE, MASSIVE-3 allowed objects and interactions to be applied locally only.

ANTS [36] was a framework designed to aid in the development of computer-supported cooperative work environments. While not specifically designed for multi-user VR it was used to build a three-dimensional collaborative virtual environment called *Move!* that let users communicate and interact with each other. It was built as part of the Catalonian *internet2* project [10], using Virtual Reality Modeling Language (VRML). ANTS used a distributed collaboration event bus as a key part of the architecture, where events were triggered by components and listening services getting notified, as well as a distributed model-view-controller pattern to develop different views, e.g. 3D and text-based views of the same shared scenarios.

AVANGO, formerly known as *Avocado*, was a framework that supported the development of distributed and interactive VR/augmented reality (AR) applications. It was initially presented in 1999 [57] and later revised in 2008 [44]. In contrast to the former presented frameworks it especially focused on rapid prototyping using Python as its scripting language and on shielding developers from the distribution-specific aspects. The underlying networking architecture, similarly to the DIVE, built on the concept of distributed shared memory with active replication and reliable multicast communication between peers.

Figure 2.2: ImmersiveDeck - Two users interacting during an immersive VR experience [50].

In terms of networking, AR systems are very similar to VR systems. One such a collaborative AR system was Studierstube, which was initially presented in 1998 [55]. It was built on top of the Open Inventor toolkit and focused on local collaboration, where all players are physically in the same location while using the virtual environment. Construct3D [42] is an example of an educational collaborative multi-user AR application that was built with the Studierstube system.

2.2.2 Immersive Walkable Multi-User VR Systems

Since then several multi-user VR systems that focused on real walking in a large physical space have been published. For example, the HIVE presented in 2007 [58] is a huge immersive virtual environment system that enabled users to move realistically inside large physical spaces. This was achieved with a mobile computer backpack attached to a HMD and outside-in optical position tracking. While it was designed to support multiple users and tracked objects simultaneously this multi-user aspect was not initially focused on.

A similar low-cost multi-user immersive VR system called ImmersiveDeck was presented in 2016 [50]. It used inside-out optical head tracking and full body tracking via a motion capture suit. Figure 2.2 shows the real walkable area and tracking, as well as the virtual representation. The networking architecture was built on top of the Unity3D networking framework, where clients communicate with the server and request ownership of GameObjects to locally update them and multicast the replication via the server.

Today there also exist several immersive walkable multi-user VR systems deployed in VR entertainment centers. An example for such an entertainment center is Zero Latency

VR [26], which offers immersive VR experiences in a large walkable space for multiple players simultaneously.

2.2.3 Other Collaborative Multi-User Virtual World Systems

Multiple virtual world platforms exist that, although not entirely focusing on, are getting increasingly used for immersive VR. For example, Second Life (SL) [14] got launched in 2003 and is still in development today. It provides the infrastructure and development tools to build VR applications in a persistent virtual world. In 2007 the open source server platform OpenSimulator [13] emerged as an alternative to SL and was compatible with the SL client software. These platforms were used for collaborative VR experiments in the scientific community, as for example in the field of scientific visualization in VR [34].

Another similar distributed and open source VR platform recently appeared called High Fidelity [7]. It is a social VR platform where users can collaboratively create and explore virtual worlds.

2.2.4 Modern Multi-User VR Frameworks

Formerly, most of the productively used VR applications were developed with VR platforms that emerged from the scientific community, as the ones previously described, but with the rapid growth of the VR market more and more VR software platforms emerged. VR developers today can additionally draw from a range of native software development kits (SDKs), take advantage of feature-rich game engines, as well as develop using web technologies for browsers as a platform.

2.2.4.1 Web frameworks

Web frameworks that build on top of technologies like WebXR [25] are offering powerful development tools for browser-based VR applications. For example the A-Frame [1] open-source VR framework is a good and easy-to-use way to create WebXR applications. Together with WebXR such frameworks have the potential to lead to further adoption of VR in the consumer market and use the development resources of invested web developers to drive content creation [33].

2.2.4.2 Game Engines

Game engines provide several robust tools and development environments that enable the quick development of complex applications with state-of-the-art graphics and VR integration. Unreal Engine and Unity are popular game engines for developing desktop and mobile games, but are becoming increasingly popular for developing VR applications as well. Due to their mature state and free licensing options they become more and more state-of-the-art for developing high-end VR applications. They were used before for

multi-user VR in the scientific community, as for example by Jacobson [41] who modified the first-person shooter Unreal Tournament, which was based on the Unreal Engine.

An exploratory study [37] on technologies used for developing open-source VR applications showed that the usage of game engines is becoming more and more popular. It also found that there is a strong preference for developers to use Unity for developing VR applications, and less dominantly Unreal Engine. This difference is also reflected in the size of their communities. The reason for this difference might include that UE4 was made available for free not so long ago, while Unity3D had a free version for a long time, but also that Unreal Engine has a steeper learning curve and less popular programming language. The study found that a majority of the open-source projects were implemented in JavaScript and C#, both of which are supported in Unity.

Similarly, another empirical study investigating the growth of the number of open source VR projects made with Unity was conducted by Rodriguez et al. [52] and also showed that their number is steadily growing.

Both Unreal Engine and Unity provide sophisticated networking frameworks to develop multiplayer games, which are conceptually very similar to each other. They build on a client-server architecture with primary-based replication.

CHAPTER 3

Methodology

This chapter presents the approach for analysis, design and implementation to achieve the previously introduced goals of the thesis. The used methodology includes the analysis and design methods to develop the proposed VRNet software architecture. Furthermore, a software solution based on this architecture is implemented and then integrated in a real world project. The underlying technologies and our hardware setup for the implementations will be explained in detail.

3.1 Analysis Methods

Requirements Analysis is an important aspect of Software Engineering and includes the tasks necessary to define the needs for new or existing software [43]. The process significantly differs from project to project because problems and the approach to solve them are different. The results of a requirements analysis should include the required characteristics, functional and non-functional requirements of the system.

According to ISO/IEC 15288 [40] one of the tasks is the identification of the important stakeholders that are involved with a system, their needs, expectations and desires. The stakeholder expectations need to be analyzed and formalized into a requirements specification, including the description of functional and non-functional requirements. Requirements are used as inputs for the design and also for the verification process of software. These requirements will be used as the basis of the design of the VRNet software architecture, as well as used for reflection on the achieved software implementation.

Functional requirements describe the precise functionality that a system should provide. They need to specify how the system should react to certain input and situations, as well as what the output and behavior is. They should be written in a consistent and clear form to make them easily understandable and avoid ambiguity [51].

Figure 3.1: ISO/IEC 25010 defined quality characteristics and sub-characteristics [11].

In contrast, non-functional requirements (NFRs) are qualities describing the operation of a system, ensuring its effectiveness and usability. ISO/IEC 25010 [39] defines eight main categories of NFRs: **Functional Suitability, Performance efficiency, Compatibility, Usability, Reliability, Security, Maintainability and Portability**. This categories, as shown in Figure 3.1 include sub-requirements, like for example Reusability, Accessibility.

Good requirements should be as complete and consistent as possible, i.e. there should be no conflicts and contradictions. They need to reflect what the relevant stakeholders want from the project.

3.2 Software Architecture Design Methods

For the VRNet plugin the software architecture is designed with the requirement analysis outcome in mind and further refined during implementation in an iterative and incremental process. This thesis presents the final architecture solution outcome after its steady evolution.

The software architecture is an abstraction from the implementation and shows how elements are arranged, how they are composed and how they interact with each other. One of the goals of the software architecture design is to provide the documentation and schemas that can be shared with the involved stakeholders for discussion and for future developments. But even though the architecture evolved during implementation the implementation details are still detached from it and could be changed, making the architecture a general description for a multi-user VR framework built in the form of an UE4 plugin.

The architecture descriptions consist of the overview of the framework (e.g. what type, core architectural design styles, etc), constraints and the formal requirement descriptions.

The software architecture is constrained by the UE4 platform and its architecture, of which the relevant parts will also be analyzed and shortly presented in Chapter 4.

The understanding of UE4's capabilities is important for the design of the classes and subsystems of the VRNet framework.

3.2.1 Design Tools

Additionally to the methods for the written architecture description it is important to choose appropriate tools for creating the diagrams and schemas. Unified Modeling Language (UML) is the de facto modeling language for software development [45]. It is chosen for this master thesis to present module structures and parts of the plugin on a high abstraction level for the sake of presentation.

There are countless UML tools available for free. Visual Paradigm [23] proved to be a good choice in terms of ease of usage and pleasantness of the visual presentation. It fits the purpose to create the necessary diagrams in a visually pleasing way and make them easily maintainable in the future. The free community edition restricts some of its functionalities, but none that were necessary for this thesis, except the extraction of diagrams without a watermark, for which the trial is used. Version 15.1 is used in this thesis.

3.3 Implementation Tools

In this section the implementation methods are described. The main technologies used are the Unreal Engine in combination with the C++ programming language.

3.3.1 Unreal Engine

Unreal Engine is a proprietary game engine developed by Epic Games [4]. It is mainly focused on the creation of first-person shooter games, but it supports a wide variety of games and applications.

It provides several important features for the creation of multi-user VR applications, most importantly a very robust and efficient networking framework, as well as a specifically developed rendering solution for VR. It also provides many development tools, forming a feature-rich development environment. Unreal Engine also provides an extensive official documentation [20]. This documentation was used to gain deeper understanding of the engine and helped in development of the knowledge necessary of writing this thesis. The following subsections in particular are written with the documentation as its source.

It is free to use with a royalty system on gross product revenue and has shared source code access to its full C++ Source Code, with the possibility to modify and customize the engine source.

The current major version as of writing this thesis is Unreal Engine 4.

3.3.1.1 Programming Languages

With its current major release UE4 supports developing in two languages. The main language the engine is programmed in is C++, which can also be used to develop UE4 projects.

It also supports visual scripting through its *Blueprints Visual Scripting* system. Blueprint Visual Scripting is a robust, designer-friendly tool which enables gameplay classes to be created visually in the editor. It is very useful for rapid prototyping without the need to modify and compile C++ source code. While not all engine functionality is exposed to Blueprints yet, it is currently still heavily under development with more access and useful features coming out with each UE4 release.

With C++ it is possible to write new gameplay classes, user interface elements and editor functionality and compile it with Microsoft Visual Studio [24]. Apart of the compiling, for the development in this thesis the Visual Studio 2017 integrated development environment (IDE) is used.

Unreal C++ differs from regular C++ with many additional language features. It uses a reflection system to power many systems like detail panels in the editor, serialization, garbage collection, network replication and communication between Blueprints and C++ [19]. To use this reflection it is important to annotate any types or properties that should be visible to the system. The Unreal Header Tool (UHT) is a standalone program that is harvesting and using that information on compile time to generate C++ code containing the reflection data. It works in concert with the Unreal Build Tool (UBT) that scans the headers and detects any modules that contain headers with at least one reflect type and then invokes the UHT.

Annotations are provided in the form of C++ macros that can contain several keywords defined by Unreal Engine, e.g. UCLASS(Abstract). Necessary boilerplate code is included also in the form of C++ macros, like GENERATED_UCLASS_BODY().

For this thesis we will manly focus on using Unreal C++ to develop the VRNet framework, but provide accessibility and example implementations for Blueprints.

3.3.1.2 Tools and Editors

UE4 comes with a powerful editor that includes several different tools and editors.

The most important ones for the thesis are briefly described here:

- The *Level Editor* is the primary editor used to construct scenes for the game. In UE4 these scenes are generally referred to as *levels*. The Level Editor defines the play space by adding objects, geometry and functionalities. It is the main editor window that comes up when starting the UE4 editor, with all its panels and viewport and the other editors are opened through it.

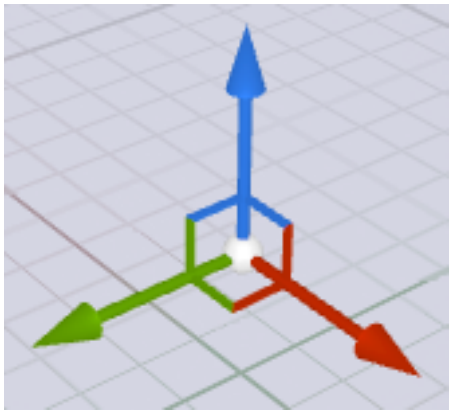


Figure 3.2: Coordinate cross in Unreal Engine 4 (X=red, Y=green, Z=blue).

- The *Blueprint Editor* is a node-based graph editor and used to open and modify Blueprints. It is focused to provide a context sensitive design to access functionality when it is relevant.
- The *UMG UI Editor* is the tool to develop Unreal Motion Graphics (UMG) user interface (UI) elements. At the core of UMG are *Widgets* consisting of different components and functionalities. They include interactive components such as buttons, checkboxes, sliders, progress bars. Widgets are edited in a specialized *Widget Blueprint*. It consists of a *Designer* tab, which allows the design of the visual layout of the UI and a *Graph* tab which defines the functionality of the widget in Blueprint code.
- The *Material Editor* is used to create and edit Material shaders to define the visual look of meshes. It allows the visual programming with of a custom shader, which gets translated into High-Level Shading Language (HLSL) code, which is a proprietary C-like shading language developed by Microsoft.

UE4 uses a left-handed and Z-up world coordinate system which can be seen in Figure 3.2. The red X axis is the forward axis, the green Y axis sideways and the blue Z axis up. Rotations are named *Roll* for the rotation around the X axis, *Pitch* for the rotation around the Y axis and *Yaw* for the rotation around the Z axis.

3.3.1.3 Platforms

One of the main benefits of using an engine like Unreal Engine is that a wide variety of platforms is supported and maintained, minimizing the effort for application developers to distribute to different platforms.

It is currently possible to develop on Windows, OS X and Linux with the Unreal Editor.

It supports several Virtual Reality platforms like SteamVR/HTC Vive, Oculus Rift, PlayStationVR, Google Daydream, Samsung Gear VR and OSVR.

3.3.2 Version Control

In software development using a version control system (VCS) is crucial, especially when multiple developers work on a project. Source control is supported inside the UE4 editor with an interface serving as an abstraction from the particularly used VCS. UE4 currently supports *Git*, which was chosen for the projects in this thesis.

Git [5] is a distributed VCS for tracking and managing changes in source files. Because the entire history of the repository is transferred to cloning clients due to the distributed nature, it can take a huge amount of time for projects that have a lot of large files that are frequently modified throughout its history. UE4 projects typically contain many large binary assets, like Blueprints, Levels, 3D models and other asset files. Because of this it is important to use a git extension called *Git LFS (Large File Storage)* [6] to reduce the impact of the large files, which is done by only downloading the files lazily when a certain revision is checked out.

The UE4 integrated VCS implements a diffing and merging tool that can be used to see changes between Blueprint versions. Other than with this tool it is not possible to see changes between Blueprints, due to them being binary files.

3.3.3 Hardware Setup

Two setups are used for developing and testing the implementations. The *development setup* is minimalist with only one machine and intended to allow development with quick testing possibilities for one developer. The *testing setup* represents the targeted main usage setup of multi-user VR for projects using the VRNet framework.

The HTC Vive is a VR headset developed by HTC and Valve Corporation. The HMD has a resolution of 1080x1200 pixels per eye and a refresh rate of 90 Hz, with a field of view of about 110 degrees. Two tracked controllers are provided for user input, and optionally Vive Trackers [9] can be used to track additional objects or to capture player movements of previously untracked body parts. To power the headset it is necessary to use a computer with a strong GPU.

Development Setup For the development of the UE4 plugin, primarily a custom-built desktop PC is used in conjunction with an HTC Vive. The tracking is configured in a standing area due to size limits of the available testing space. The computer has an Intel Core i7-6700 processor, 16 GB of RAM and a Radeon RX 480 graphics card with 8 GB GDDR5 memory.

Testing Setup For testing during and after development we use a setup with room scale tracking and several fist generation HTC Vives. Each of these Vives is connected to a backpack laptop, offering the users a possibility to walk around without restricting their movements. The backpacks are XMG laptops with an Intel Core i7-6700HQ CPU, 16 GB of RAM and a NVIDIA GeForce GTX 1070 graphics card with 8 GB of GDDR5 graphics memory. The testing area consists of a roughly 4x5 m large tracked room.

3.4 Terminology

A large part of terminology used in a game engine refers to games, even though the presented VRNet platform does not focus on VR games, but rather generic VR applications. Terms like users and players will be used interchangeably in this thesis, because while we do not actually see the target users of the VRNet platform to be players in its commonly understood way, it is the main term used in UE4 and its components. Similarly, *game* and *application* are used synonymously.

As for the implementation class names, the name convention in UE4 is to use a single letter prefix for the class name. For example classes deriving from *Actors* are prefixed with A, e.g. *AActor, APawn*, etc. and other classes deriving from *UObject* are prefixed with U, e.g. *UActorComponent*, etc. This prefix for the class names will be omitted in the thesis description to improve readability. Similarly, UE4 terms will be capitalized, so when using e.g. Actors and Components, we refer to UE4 Actor and Component classes.

CHAPTER 4

VRNet Design

VRNet is a framework which is intended to provide core and utility functionalities for developing networked multi-user VR applications. It is proposed as an UE4 plugin and therefore many design decisions are already made and a lot of the necessary functionality for applications is already provided by the engine.

This chapter presents the design results for the VRNet plugin, following the methodology described in Chapter 3. The identified target users and their expectations, as well as the extracted functional and non-functional requirements are described in section 4.1. Furthermore, it presents the relevant architectural foundation of the Unreal Engine in Section 4.2 on which the VRNet plugin and its parts are built on. The proposed software architecture descriptions of the VRNet plugin are presented in Section 4.3.

4.1 Requirements

The first step of the design process involved the analysis of the involved groups of persons, their expectations and requirements and the formalization to the functional and non-functional requirements of the VRNet framework. We consider developers to be the main target user group of the VRNet plugin.

4.1.1 Target Users

The main target users described here are two types of software developers: developers that use the VRNet plugin and developers that maintain the VRNet plugin.

Project Developers These are developers that will use the VRNet plugin to create multi-user VR applications. They can have different levels of programming proficiency and understanding of UE4. Since UE4 provides the Blueprints visual scripting system it is also possible to create UE4 applications without conventional programming knowledge.

The main expectations for this type of developers are useful functionalities for multi-user VR that are easy to use, configure and extend, and the stability of these functionalities.

Plugin Developers Plugin Developers are software developers that maintain the VRNet plugin code, for example to improve, fix or add new functionalities. Of course developers can be Project developers as well as Plugin developers at the same time. The main expectations for this type of developers are an understandable and maintainable code base with possibilities to modify it without breaking other software components, as well as a straightforward way of deployment of such changes.

4.1.2 Functional Requirements

- *Player movement distribution:* When the position of a player changes, all the other players should see this position change. The player handling involves movement of different kinds, from walking around inside the available play area, to teleporting, to moving with controllers and the HMD, or similar. All these types of movements should be handled by the VRNet plugin.

- *Avatar distribution:* Additionally to the player movement distribution the VRNet plugin should have proper player avatar handling and provide the possibility to customize the type of player representation inside the world. The avatars, as well as their distribution can be very different, ranging from simple static meshes to complex skeletal meshes with several distributed sockets.

- *Interaction distribution:* The VRNet plugin should provide a solid foundation for distinctive interactions to build on. Many of such possible interactions do not need a custom networking implementation when using UE4's replication and physics systems directly, but more specific ones might need complex logic with a common core including player input handling and ownership control and its networking. Because of this the foundation and common, ready-to-use interactions serving as showcases should be provided with the VRNet plugin:

  - *Grabbing interaction:* Grabbing objects is one of the most basic and intuitive controls in Virtual Reality, so it is important that the plugin offers a robust implementation with distribution for this type of interaction in particular.

- *Game Management and Connection Handling:* The VRNet plugin should include functionality for players to connect to each other either at game start or through interfaces inside the world at runtime. It should provide networked functionalities for controlling the current game session, like changing level and changing the player avatars and input methods.

- *Blueprint Access:* All the relevant functionality should be accessible in Blueprints as well as C++. Projects using the VRNet plugin should not be forced to use one over the other. For making the platform Blueprint-able it is necessary to identify

the important functions and properties of the classes which should be exposed to Blueprints and provide helper functions for easier Blueprint development.

4.1.3 Non-Functional Requirements

Maintainability This characteristic includes the effectiveness and efficiency with which a system can be modified to change it for improving, correcting and adapting it. Some generally important tactics to fulfil this group of non-functional requirements are to increase cohesion and reduce coupling between software components.

The main identified sub NFRs are listed as follows:

- *Modifiability:* Modifiability specifies the degree to which the functionality can be efficiently modified. Considering that VRNet should be a platform for developing different projects it needs to be easily changed and extended to fit their specific requirements. Many of the provided functionalities should be changeable by configuration and setup, without the need to change the source code.
- *Reusability:* Another important quality attribute for VRNet is Reusability, which is the degree in which the platform and its components can be used and reused in different solutions. For this independent software components that can be combined together are important.
- *Testability:* It is also important for the modules and the platform as a whole to be testable, to verify that the implemented functionalities work as intended and can be trusted to use. For debugging network applications it is necessary to provide ways to verify state and provide a proper logging framework that allows to analyze proper execution order on different machines and their states. The modules and components should be testable with automated tests.

Performance Efficiency The primary relevant performance characteristics are *Time Behavior* and *Resource Utilization*. It should be possible to use the functionalities of the VRNet plugin with a sufficiently high degree of performance efficiency so that it can be used in VR. Also, the integration of the plugin should not lead to a degradation of overall performance.

- *Time Behavior:* For VR, an absolutely mandatory quality attribute is smooth performance, since latency and non-consistent frame rates can rapidly lead to motion sickness in users. Unreal Engine provides tools to properly maintain and measure the time behavior. Tactics used by VRNet to improve the Time Behavior performance include the use of these tools to minimize network delay and introduce smoothing in case of interruptions.
- *Resource Utilization:* The amount of the used resources when performing its functions should be at acceptable levels. The network bandwidth should not exceed

unsustainable values and the implementation therefore should try to reduce it whenever possible. VRNet uses as a tactic to offer configuration possibilities to define the update rate and control network bandwidth in conjunction with network smoothing that is necessary on lower bandwidth. Additionally the CPU and memory utilization when using the plugin should not have a noticeable effect on projects.

Usability Another important NFR is *Usability*, which, although not a main focus, has to be kept in mind when developing the plugin. The intended users of the project are experienced developers of VR applications, but UE4 in general has a steep learning curve and because of that it is almost impossible to create a framework on top of it without requiring its users to understand its fundamentals. But apart from that, tactics in the design and implementation of the VRNet platform include making the source code understandable and providing documentation. Blueprints offer an easier to use and more abstract way to develop gameplay functionality without understanding most of the engine C++ code. Because of this Blueprint Accessibility is a big focus of the VRNet plugin as was already formalized as a functional requirement.

Other quality attributes Other usually important qualities for distributed systems like *Security* were not focused on, but can be added and extended via the provided means of UE4. For example, for our purposes client-tampering should not be a problem since the VRNet plugin is mainly meant to be used by projects that are used in controlled environments. Another important requirement is *Portability*, which is already taken care of implicitly by using UE4 with its support for the most important platforms and therefore not further taken into consideration for the VRNet plugin.

4.2 Unreal Engine Foundations

This section describes the UE4 gameplay classes and components that are relevant to the VRNet plugin. The presented engine classes are written in C++ and the source is openly accessible for registered developers. The following descriptions are based on the official documentation [20], the source code and our gained understanding during the development.

4.2.1 General Architecture

The most important entities in UE4 are the so-called *Actors*, which represent any object that can be placed or spawned in the world, and Components, which cannot exist by themselves but can be added to Actors and extend them with functionality.

Components are further split into three important types:

- *ActorComponent*, which is the base class of all Components that can be added to different types of Actors. It provides lifecycle functionality that is called by

the owning actor when it is for example spawned or destroyed. ActorComponents further can be updated each frame via their overridable TickComponent() function.

- *SceneComponent* (deriving from ActorComponent), which has transforms (i.e. location, rotation and scale) and can be attached to each other in the scene graph of an actor. Attachment is generally handled only on the Component level via SceneComponents. This way, if two Actors need to be attached to each other both need to contain at least one SceneComponent. Each SceneComponent has a FTransform struct that describes its world-relative location vector, rotation quaternion and scale vector. Additionally it has a relative transform describing the location, rotation and scale relative to its parent component and the possibility to also force these values to be relative to the world (i.e. with setting any of bAbsoluteLocation, bAbsoluteRotation and bAbsoluteScale to true). This allows for example to make the location independent, but keep the rotation and scale relative to the component parent.
- *PrimitiveComponent* (deriving from SceneComponent) contains or generates some kind of geometry, which is generally used to be rendered or for collision-related data. Many physics settings are found in this class. For example CapsuleComponents generate a capsule that is used for collision detection while StaticMeshComponents contain rendered pre-built geometry that can also be used for collision detection.

4.2.1.1 Actor

An Actor is a generic class that is a container holding ActorComponents. It provides functionality for replication of properties and function calls over the network and for handling the lifecycle of the object.

Actors do not have transformation information itself but if their RootComponent is a SceneComponent they use its transform.

Unreal Engine and Actors provide a sophisticated lifecycle framework to hook into if necessary. Most importantly, this framework defines how an Actor is instantiated (Spawned) and how it is removed (Destroyed). While there are several different ways for Actors to get instantiated there is only one path to their destruction.

For instantiation Actors can be loaded from disk, which occurs for Actors that are already placed in the level or in case of "Play in Editor", which is mostly the same as "Load from Disk" but instead of the Actors being loaded from disk they are copied from the Editor. Actors can also be spawned on runtime, either directly or deferred to initialize the instance before being spawned in the world. The most important lifecycle event being called on Actors and ActorComponents by the engine in the last step of the Actor spawn process is BeginPlay.

Respectively EndPlay is called when an Actor gets destroyed either manually, on level transition, lifetime expiration or if the application is shut down. When an Actor is destroyed it gets marked for the garbage collector to clean up in the next collection cycle.

Actors and their components can be set to update (so-called "Tick") each frame, at a set minimum time interval or not at all. They can be put into Tick Groups which define when in the frame it should tick, relative to other in-engine frame processes (e.g. physics simulation).

Notable Actors from the game framework are:

- *PlayerController:* represents high-level player input towards the game. Controllers are non-physical Actors and can possess a Pawn and set rules for its behavior. While a PlayerController is human-controlled there also exists an AIController class which implements the artificial intelligence for controlling Pawns.

- *Pawn:* represents a player or AI entity in the World and needs to be possessed by a PlayerController for a player to control it. By default only one Controller controls one Pawn at any given time. A notable and widely used subclass is *Character*, which extends the Pawn class with multiple functionalities designed for horizontal-oriented movement like walking, running, jumping, flying and swimming through the world. The class also contains sophisticated movement replication with client prediction and simulation.

- *GameMode:* controls the game flow, rules and logic. It also handles client connections, spawning their PlayerControllers and default Pawns and transitions between levels. It only exists on the server and does not get replicated to clients for security reasons. In the recent UE4 version 4.14 the *GameModeBase* class was extracted to a more simplified and stream-lined base class that omits specific and legacy code that is not necessary for most projects.

- *GameState and PlayerState:* hold state information for the game and each player respectively. GameState is closely related to the GameMode class and contains the state that needs to be replicated to all machines. Similarly, PlayerState holds the state belonging to a player.

- *GameInstance:* is a singleton, high-level manager object for a running game instance. It gets spawned at game creation and destroyed on game shutdown. Each machine running the game has one instance of this object, but it is not replicated. It provides helper functionalities, access to OnlineSessions and can be used to store data that needs to be carried between levels without serializing it to disk.

4.2.2 Networking Framework Architecture

Before examining the specific VRNet architecture it is important to take a closer look at the networking framework provided by UE4 and to consider its restrictions and consequences for VRNet.

The UE4 networking architecture is very similar to Unity HLAPI networking architecture.

It is a strict client-server architecture. The server is authoritative and data must be exchanged with the server to affect game state which gets communicated with other clients. The server side is designed to offer possibilities to further validate client requests before applying them as a security measure. One constraint is that the UE4 network architecture does not support server to server or client to client communication, making it impossible to exchange information between clients directly or having multiple servers for scaling purposes.

As mentioned in the previous section, the Actor class contains the main functionality for replication. Actors can be configured to exist on the clients as well as on the server with the engine maintaining the low-level connection between them, or not to be replicated at all.

There are two main mechanisms for the network communication between Actor instances:

- *State replication from Server to Client(s):* Every Actor has a list of marked properties that should be replicated. Whenever the value of the variable changes on the server side it will get send by the server to the clients and overwrite any value that may have been changed on the client side. This type of replication is reliable: the values marked for replication inside an Actor will eventually reflect the values on the server. It is also possible to control certain replication behavior, for example, to skip replication to certain clients. The Actor class provides many optimizations and settings for networking, as for example setting a maximum update frequency or "Adaptive Network Update Frequency" functionality that reduces the attempts to replicate an Actor when nothing is really changing in order to improve replication performance.
- *Remote Procedure Calls:* remote procedure call (RPC) are a useful abstraction where a function gets called locally, but executed on a different machine without the programmer needing to handle the necessary low-level communication. RPCs allow sending messages between the Actor instance of the client to the server, or the server to the client. Important requirement for RPCs is that the Actor needs to be replicated and the ownership of the actor needs to be considered. Clients have to own the Actor (default for example their PlayerController or Pawn) to be able to send a RPC to the server or to receive the correct RPC from the server.

  Functions can be declared as an RPC by adding keywords to the UFUNCTION declaration or setting them inside the Blueprint editor.

  - Client: RPC that will be called on the server and executed on the client that owns the Actor. If the server owns the Actor it runs on the server instance.
  - Server: RPC that will be called on the owning client and executed on the server. If the server calls the function it runs on the server instance.
  - NetMulticast: RPC that will be called on the server and then executed on all connected machines, including the server itself. If called on the client it will only execute on the calling client.

By default the RPCs are unreliable, but it is possible to declare an RPC as reliable to ensure that the RPC call is executed on the remote machine. This is handled with the engine implementation on top of UDP that resends lost packets and therefore can come with a big performance hit when compared to unreliable RPCs.

Networking settings can be changed per Actor. The update rate can be optimized by a careful setting of replication properties. It is also possible to configure an Actor to not be replicated and spawned on clients at all. This means that when an Actor is spawned on the server it will not be spawned on the clients. Similarly, Actors spawned on clients are never replicated and only exist on the local client.

In general, the instantiation of Actors can be grouped in the following categories:

- *Server only:* e.g. GameMode.
- *Local client only:* e.g. UIs and Actors that are visible and relevant only to the local player.
- *Server & local client only:* e.g. PlayerController.
- *Server & all clients:* most other replicated Actors, as well as Actors placed in the Level from the start but not being replicated, i.e. their instances not having a connection between machines.

The GameMode exists only on the server and handles game logic and integrating connecting clients and spawning their pawns. This is primarly a security measure to not give unnecessary access to game controlling logic to the clients.

The PlayerController exists only on the client owning it and the server. This means, while the server has a list of all connected players' PlayerControllers, each client only has access to its own PlayerController. This is also a security measure, as well as a reduction in unnecessary overhead.

To share the game and player related state to all machines the GameState and PlayerState Actors are used and set to replicate.

Special effects and UIs that need to be only visible to the local player are preferred to be spawned only on the machine of this player, instead of replicated directly. This reduces the replication overhead.

4.2.3 Plugins in UE4

Another relevant architectural aspect of UE4 is its plugin-based architecture. Many of the subsystems of Unreal Engine are designed to be modifiable and extendable via plugins.

UE4 plugins are flexible in regards to their distribution. They can be packaged and installed as engine plugins, or directly added to the project's directory. Engine plugins

are available for all projects, while project plugins are only available in the project they are put into. The main benefits of engine plugins are that once they are installed they can be used in multiple projects and centrally maintained, but leads to developers having to install the plugin before the project gets usable. Additionally the plugin needs to be packaged and installed for each engine version. UE4 looks for a missing plugin dependency in its provided marketplace [21] and offers to install it automatically to use the project, but this requires the plugin to be published on the marketplace. The marketplace in general greatly simplifies the management, integration and update process for plugin users, as well as plugin developers. When installed on the project level, it suffices to include the packaged plugin with the project source code and thus leads to no fruther installation process before the project gets usable. It compiles with the project and it is possible to change code directly in the plugin's source files. How plugins are distributed and integrated into projects does not directly effect their content, though engine plugins may have further restrictions like for example no directly defined dependency to other engine plugins.

Even though VRNet does not plan to change core engine functionality, it extends UE4 provided classes to add networked VR capabilities. For this purpose a runtime code-plugin is proposed to include and link C++ code into a project, where the exposed classes can be used or derived to further customize them. It is also possible to include Blueprints and other assets in plugins, but through their relatively big size they might make the plugin's size requirement bigger than it has to be.

In the scope of this thesis the VRNet plugin is intended to be distributed as a project plugin. Especially for the integration into the CeMM Holodeck project it is important to make the project independently usable without a central plugin maintainer, as well as allowing changes to the plugin's source code directly if necessary. But the presented design of the VRNet plugin is not influenced by its later distribution.

4.3 VRNet Plugin Architecture

The VRNet Plugin provides subclasses to UE4 game framework classes enhancing them with crucial and useful capabilities for multi-user VR with regard to the requirements described earlier in this chapter. It is an UE4 runtime plugin providing C++ classes that can be easily integrated into projects by linking the plugin and deriving the classes or using them as-is.

The description of the software architecture of the plugin is divided into Subsection 4.3.1 describing the core functionality, including client-authoritative replication and classes that provide distributed handling of player functionality and Subsection 4.3.2 describing useful game management functionality. This includes online session handling that can be used optionally.

4.3.1 Core Functionality

The core functionality of the platform provides classes necessary for distributed multi-user VR applications. The focus lies on the handling and replication of user movement and representation, as well as on the replication of their interactions.

4.3.1.1 Client-authoritative Replication

An important core concept of the VRNet plugin is the usage of client-authoritative replication rather than the server-authoritative default implementation in UE4. It is used to assure the latency free movement of users in VR, which importance was described earlier for the *Performance Efficiency* NFR. This approach is a trade of security for avoiding potential movement corrections from the server, which can be very disturbing for humans in VR. For our purposes we assume that clients are trustable, so that client-tampering is not an issue. Hence such security concerns are negligible for the intended application field of the plugin and the advantages of better user experience outweigh them.

In UE4 replication works server-authoritative, meaning that the server is in control of game logic including movement application and updating the other clients. The idea of the client-authoritative approach is to apply the movement directly to the instances on the controlling client and send the position updates to the server, which then further distributes it to other clients. This means that movement smoothing and position updates only need to be used for the other clients instead of the controlling one. The server simulation on the other hand has the same updated state as with server-authoritative position updates. This is important for gameplay logic handled on the server side.

In VRNet client-authoritative replication is used for all the movement updates related to the player, i.e. HMD and controller movement updates, but also for held objects. The logic to efficiently handle this type of replication is one of the contributions of the plugin to multi-user VR application developers. How the position data is updated and replicated is handled on an instance basis. This means that different Actors and Components can be replicated differently, depending on the situation and requirements, because both approaches replicate the state to all machines.

4.3.1.2 VRNet Player

As described earlier Actors that can be controlled by players (or more specifically Player Controllers) are called Pawns in UE4. The VRNet plugin provides two Pawns:

- VRNetBasePawn - a base Pawn providing only the most important base functionality and Components.
- VRNetPawn - a more specific Pawn deriving from VRNetBasePawn, which adds additional functionality and Components.

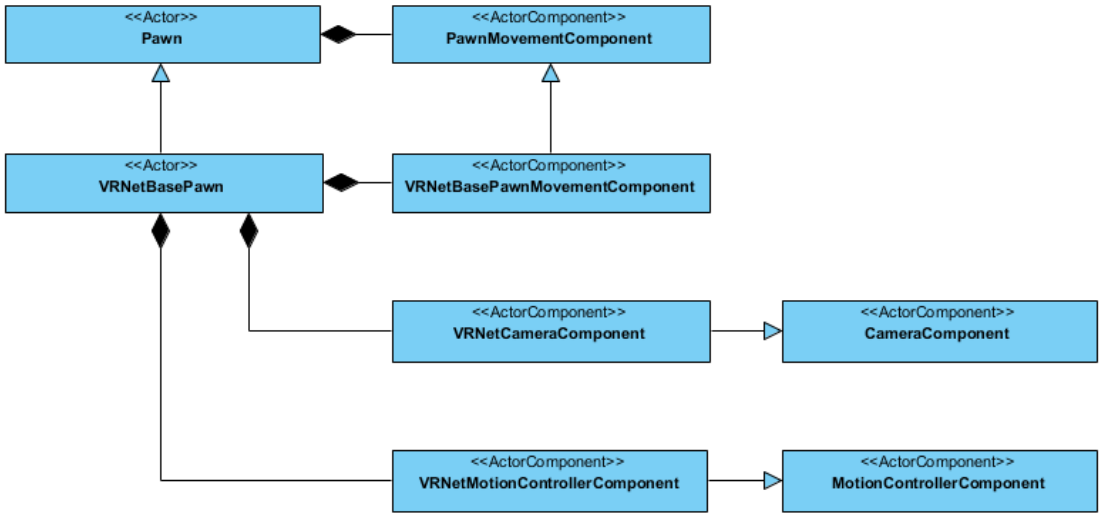


Figure 4.1: Class diagram of the VRNet Base Pawn and its Components.

Either of these Pawns can be used for multi-user VR applications, depending on the amount of required features. For many projects it might be sufficient to derive from the base class rather than the VRNetPawn class. Or to create a specialized Pawn adding certain Components that are included in the VRNetPawn. Additionally, it is possible to disable Components when deriving an Actor in UE4, if they are marked as optional.

VRNetBasePawn The VRNetBasePawn is a controllable Pawn that includes the base functionality and components necessary for multi-user VR. It contains several loosely coupled and exchangeable Components (presented in an UML diagram in Figure 4.1):

- **VRNetMotionControllerComponent:** The UE4 *MotionControllerComponent* represents a tracked controller and is abstracted to support various VR platforms and their controllers. The VRNetMotionControllerComponent class extends the Motion Controller and adds client-authoritative replication of the tracked controller location and rotation.
- **VRNetCameraComponent:** The *CameraComponent* provided by UE4 represents a camera viewpoint and settings (e.g. projection type, field of view and post-process settings). The VRNetCameraComponent sets the CameraComponent to lock to the tracked HMD and adds the client-authoritative replication of the tracked position data.

  Attached to the CameraComponent is a SceneComponent named *Head Origin*, and further to this SceneComponent an optional *Head* StaticMeshComponent. These Components are used to provide and control the visual presentation for the player's head and to define an offset for the head to the tracked HMD.

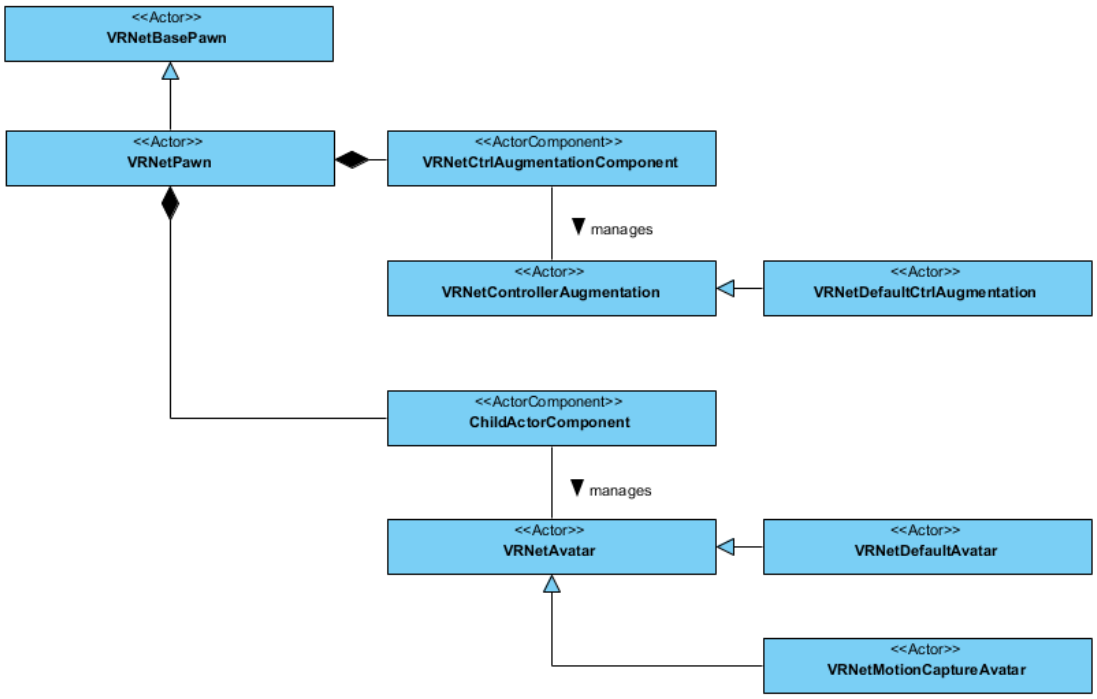


Figure 4.2: Class diagram of the VRNetPawn and its Components.

- **VRNetBasePawnMovementComponent:** UE4 *MovementComponents* provide a form of movement to the Actors *RootComponent*, or alternatively another selected component. The types of movement supported by the VRNetBasePawnMovementComponent are teleportation and simple translation/rotation of the Pawn. The component can be used in conjunction with VRNetBasePawn or any of its subclasses, including VRNetPawn, but also can be extended for a specific implementation.

VRNetPawn Extending VRNetBasePawn, this subclass provides further functionalities and components. It includes the handling of generic *Player Avatars*, representing the body of players, and *Controller Augmentations*, augmenting the controllers with additional logic and subcomponents. Avatars as well as ControllerAugmentations are intended to be easily exchangeable and reusable entities that encapsulate and add logic and Components. Because in UE4 SceneComponents can not contain child SceneComponents themselves it is necessary to use Actors for this entities and attach them to the Pawn. How this attachment works in detail is described in the implementation details for VRNetAvatars and VRNetControllerAugmentations. Figure 4.2 shows the class diagram of VRNetPawn and its Components.

- **VRNetAvatar:** While the VRNetBasePawn already provides optional visual

player representations with static meshes simply following their tracked controllers and the HMD, the VRNetAvatars add the tools for handling more complex body representations. The *VRNetAvatar* class is an Actor that contains a CapsuleComponent as its RootComponent, which can be used for collision queries for the rough player position or as a simple geometrical representation of the player. It handles the default movement of this RootComponent relative to the HMD position, but the specific movement logic can be overwritten in subclasses.

This base class can be easily extended either via Blueprints or in C++, for example to add a StaticMeshComponent or SkeletalMeshComponent. The VRNet plugin provides such a subclass including a basic StaticMeshComponent called *VRNetDefaultAvatar*. Additionally, VRNet plugin includes a more complex subclass of VRNetAvatar called *VRNetMotionCaptureAvatar*. This class provides the handling and replication of multiple tracked objects, like HTC Vive trackers or a motion capture suit.

- **VRNetControllerAugmentation:** VRNetControllerAugmentations are exchangeable Actors that provide controller specific functionality and subcomponents. They can be used to add visual representations for controllers and to augment controllers with functionality like object interaction, movement or UI interaction. By default, there is one ControllerAugmentation for each controller and they are attached to its VRNetMotionControllerComponent, but alternatively they can be attached to the VRNetPawn root if required. Additionally to the necessary boilerplate functionalities assuring the functioning of these Actors, it provides optional input handling of their respective controllers input.

  The VRNet plugin includes a default class deriving from VRNetControllerAugmentation called *VRNetDefaultCtrlAugmentation*. This class serves both as an example on how an implementation of ControllerAugmentation can look like, as well as a ready-to-use Controller Augmentation for projects. It provides logic and components for interacting with objects through grabbing, including the collision detection, as well as the input bindings and handling. It also provides basic movement of the VRNetPawn through teleportation. The grabbing interaction and its principal Components will be further described in the Subsection 4.3.1.3.

  Other examples of Controller Augmentations could be one that spawns a ring menu around the controller acting as a UI for the user to interact with, or one that simply changes the type of interactions that are possible with objects in the world (e.g. instead of grabbing to paint or scale objects). Due to this entity's swappable nature it is possible to dynamically change the behavior and the appearance of the controllers without having to define multiple instances deriving from the core classes.

4.3.1.3 VRNet Interactions

Player interaction with the environment can be very diverse and is hard to generalize. It can mean simple exchange of information, or the alteration of the state of another object, e.g. changing the pose, scale, color, physics properties or something else. UE4's Actor replication and physics system provides support for touching and moving game objects and replicate this movement, but complex interactions with objects call for specific implementations. The VRNet plugin provides the foundation and example interactions that can be used out-of-the-box and should be used as a reference for the implementation of other interaction types.

The main class layout for the interactions management and replication consists of three entities:

- An interaction interface, which needs to be implemented by the Actor that should be interactable.
- An ActorComponent containing the object behavior logic for the implementing Actor.
- And an ActorComponent that provides the management logic on the player side interacting with the interface.

Designing this functionality as Components and not as a part of Actors themselves offers the advantage of easily augmenting Actors with the necessary functionalities leading to better usability and reuseability. Actors containing objects to interact with only need to provide the interface methods and wire them up with the behavior Component. The Pawns or Controller Augmentations on the player side only need to include and setup the interaction Component.

Since the kind of input that initiates the interaction and the collision detection or collision shape can differ significantly these aspects are decoupled from the Components handling the management of interaction requests and their replication.

Grabbing Interaction Grabbing objects and holding them is an intuitive way to interact with the world in VR. Since grabbing is an important example of interaction VRNet provides classes to handle it. The grabbing interaction with an object is the changing of the position of the object according to the player movement when the player is picking it up. For the optimal user experience it may be beneficial to replicate this movement in a client-authoritative way. Similarly to player movement it is important that the movement of the grabbed object has no delay for the person grabbing it and that there are no sudden server position corrections. But additionally to the client-authoritative mode, the provided components also support server-authoritative and local-only grabbing (i.e. without movement replication). Figure 4.3 shows the UML class diagram for the described entities involved in the grabbing interaction.

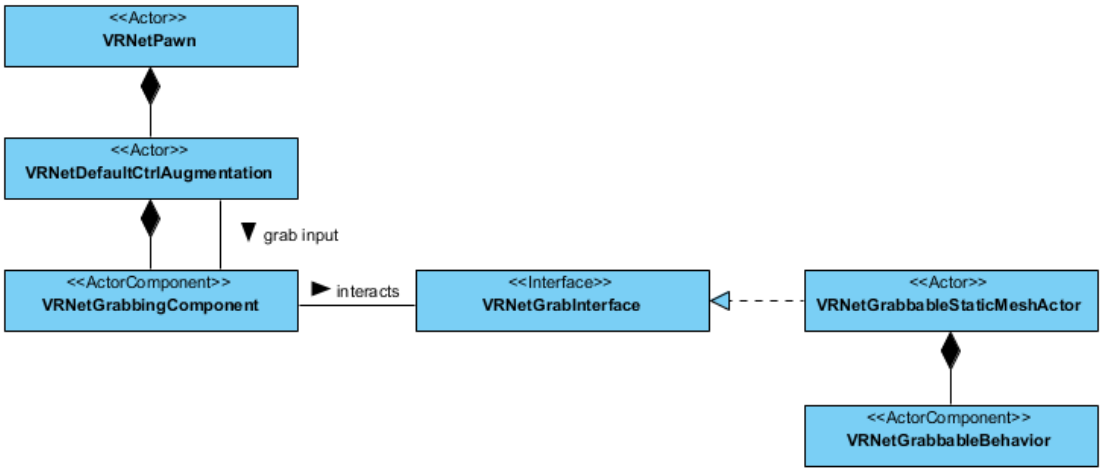


Figure 4.3: Class diagram for VRNet grabbing interaction.

VRNetGrabInterface The interface for the grabbing interaction contains methods for starting and stopping the interaction, as well as functions for retrieving the state of the interaction. It is important to query if the Actor is currently grabbed or if it is currently grabbed by a certain player. The *VRNetGrabbableStaticMeshActor* is an example Actor implementing this interface. It can however be replaced with any other Actor implementing the interface.

VRNetGrabbableBehavior VRNetGrabbableBehavior is an ActorComponent that provides the grabbing behavior needed to implement the IVRNetGrabbable interface. In this case the behavior logic could be attaching the object to the controller, attaching PhysicsHandles, or other types of locking the object to the player. It is designed to be set up with PrimitiveComponents, so that the Actor implementing the VRNetGrabInterface can define which of its PrimitiveComponents should be interacted with. Additionally, the replication handled by this component includes interpolation to smooth server position updates of grabbed objects. There are three different replication modes:

- *Default Server-Authoritative:* With this mode the actual movement of the object happens on the server and then replicates to the clients, including the client currently grabbing the object.
- *Client-Authoritative:* With this mode the movement happens on the controlling client, which updates the server via server RPC calls. The server then further updates the other clients.
- *No replication:* Additionally a mode is provided for client side grabbing without any kind of replication. This type of grabbing can be used for objects that other clients cannot see.

VRNetGrabbingComponent This component handles the grabbing logic on the player side and provides a simplified interface to the Actor containing it. The setting of the previously described type of replication is handled here as well. It is possible to set the replication type to either *ServerAuthoritative*, *ClientAuthoritative* or *OnlyLocalClient*. The VRNetGrabInterface is called by this component and the replication type is passed to the implementing object and its VRNetGrabbableBehavior Component. The grab request is distributed depending on the replication type.

As mentioned earlier, the VRNetGrabbingComponent includes the logic for managing grab requests and replicates them if applicable, but does not include any logic for initiating the grabbing, i.e. the input handling and collision detection. Input handling and collision detection are to be defined by the parent Actor. In VRNet it is included by default in the VRNetDefaultCtrlAugmentation.

Grabbing Communication This subparagraph presents the high-level communication that happens between client and server instances of the relevant Actors for the grab interaction. First the server-authoritative replication mode of the interaction is illustrated, followed by the client-authoritative one. The rectangles represent Actors and connections between them are messages with the direction, a sequence number and a label. Sequence number 4 shows ongoing, asynchronous messages (i.e. for property replication) The dropping interaction with the object has the same communication sequence and is left out in these examples.

- **Server-authoritative grabbing communication:** The communication diagram for server-authoritative communication is shown in Figure 4.4. The main idea is that the grabbed object is attached to the grabbing Actor and moved on the server, with its position being then replicated to all clients. Sequence 1 illustrates the communication happening on user movement and the collision with IVRNetGrabbableImpl (Seq. 1.2 and 1.3), which represents any object implementing the IVRNetGrabbable interface. Sequence 2 describes the user pushing the trigger button (in this example, it is the input action necessary to grab an object while the MotionController is colliding with it). The client then notifies the server via a server RPC to start grabbing the object (Seq. 2.2). The actual interaction is then executed on the server, which first calls the *Grab()* method of the IVRNetGrabbale interface. Afterwards the server instance of IVRNetGrabbableImpl is attached to the server instance of VRNetPawn and the movement is replicated to all the clients, including the owning client. (Seq. 4)
- **Client-authoritative grabbing communication:** The client-authoritative variant is shown in Figure 4.5. The main idea is that the object is attached to the Actor and moved on the controlling client directly, while the position updates are then sent to the server and consequently replicated to all other clients. Sequence 1 shows the exact same behavior as with the previously described server-authoritative mode, i.e. the user movement and collision events. Sequence 2 describes again the

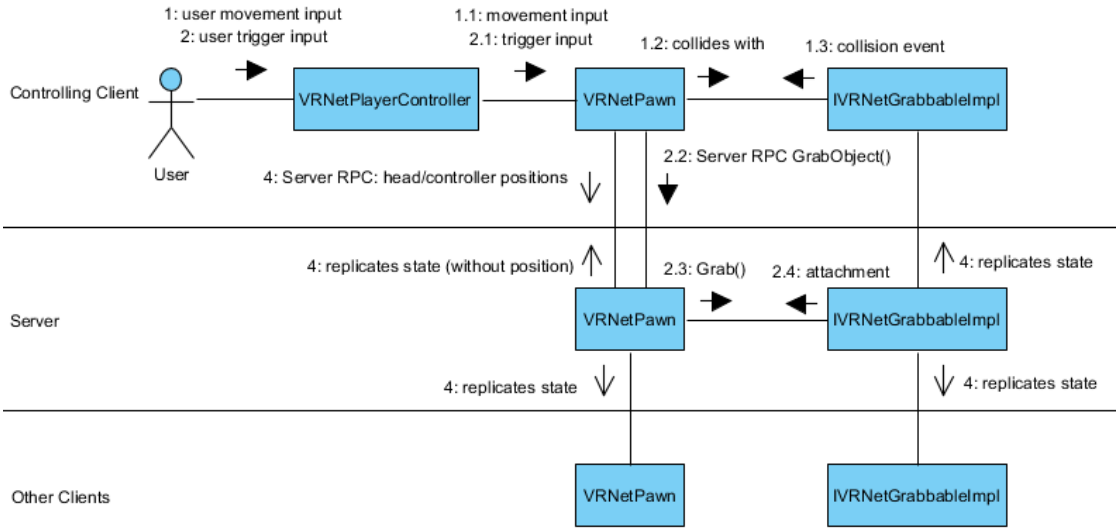


Figure 4.4: Communication diagram for server-authoritative VRNet grabbing replication.

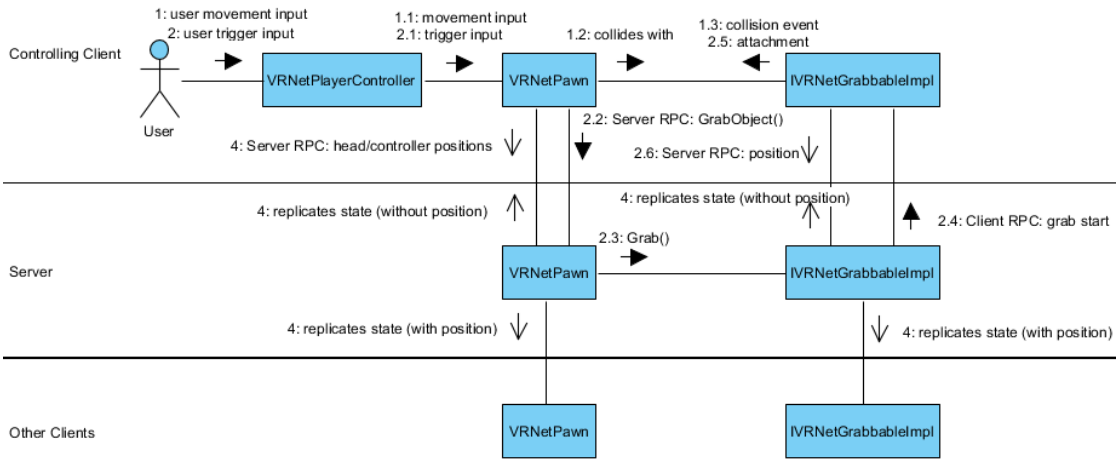


Figure 4.5: Communication diagram for client-authoritative VRNet grabbing replication.

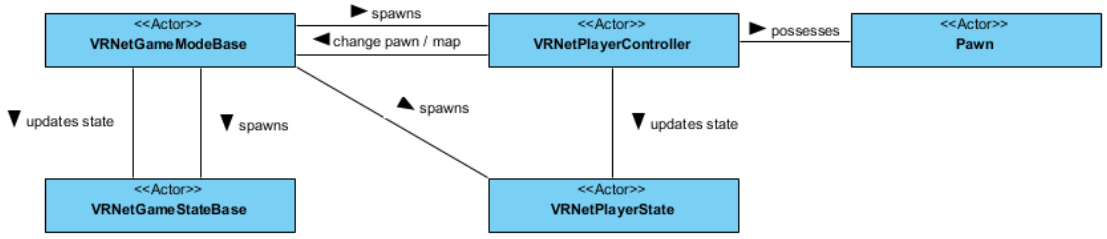


Figure 4.6: Class diagram for VRNet Game Management Functionality.

user pushing the trigger while overlapping with the object. Furthermore, there is also a server RPC (Seq. 2.2) to initiate the grabbing on the server, where the *Grab()* method of the IVRNetGrabbable interface (Seq. 2.3) is called. The main difference to the server-authoritative grabbing lies in the sequence of events that happen after successfully calling the Grab() method. The server instance then notifies the controlling client instance via a client RPC to start grabbing, at which point in time the actual attachment between the object and the player happens on the client side (Seq. 2.4 and 2.5). The object then starts to update the server with the object position via frequent server RPCs (Seq. 2.6). These position updates are further replicated from the server to all the clients except the controlling client. (Seq. 4)

4.3.2 Game Management Functionality

Besides the previously presented core functionalities for multi-user VR the VRNet plugin also includes important functionalities that help the creation of networked UE4 applications.

The UE4 default implementation of *GameModeBase* handles the automatic spawning of Pawns, which are specified inside the engine through a *DefaultPawnClass* property. The *PlayerController* class already includes the functionality for Pawn possession. For many multi-user VR applications a more sophisticated management of Pawn spawning will be necessary, including the possibility for players to change their Pawns during runtime.

The VRNet plugin includes logic allowing players to specify a certain Pawn they want to initially spawn and to change Pawns on runtime. Which Pawns are selectable can be specified in the settings. Additionally players can change or reset the current level they are in.

Those functionalities are implemented in the following classes, which are working together. Figure 4.6 shows the UML diagram that illustrates the relationship between them.

VRNetGameModeBase The GameModeBase Actor governs the game logic, controls who may enter the game and is only instanced on the server. The VRNet derived version

adds additional logic for Pawn spawning and for handling the *PreferredPawn* request parameter of connecting clients that specifies the Pawn they would like to spawn as. This preferred Pawn needs to be a part of a configurable list of selectable Pawns contained and managed inside the VRNetGameModeBase with the *VRNetGameSettings*. It also handles changing Pawns on runtime to one of the selectable Pawns. Similarly, the Game Settings include a list of selectable maps and the Game Mode provides the functionality to handle map change requests from players.

VRNetGameStateBase This Actor stores the game state data and replicates it to all clients. It works in conjunction with VRNetGameModeBase. The VRNet game state includes the Session Infos (Session Name, Session Status, Max Players) and the Game Settings (Selectable Pawns, Selectable Maps) that are set in the Game Mode. This Actor is easily accessible and can be used to display the information to users, for example.

VRNetPlayerController This VRNet-specific subclass provides players with the means to initiate the changing of its currently possessed Pawn. This change request is passed to the VRNetGameModeBase, which then handles it. Furthermore, it adds switching between Spectator mode and re-possessing its previously controlled Pawn, as well as adding basic testing capabilities to emulate movements via keyboard and mouse input.

VRNetPlayerState This class works in conjunction with VRNetPlayerController and holds the replicated state of the player. The state includes the currently selected Pawn, which gets set by the Game Mode, and the preferred pawn the player wants to spawn with initially which can be set directly. The selected Pawn can differ from the preferred Pawn, because the intended behavior for the preferred Pawn is that it will get used on any level reset or on connection of the player for the initial spawning, and therefore can be set independently.

4.3.2.1 Online Session Management

The Online Subsystem (OSS) of UE4 provides an abstraction for online functionality like session management and matchmaking across a wide range of platforms. A session is a running instance of the game. It can be advertised so other users can find and connect to it or private so only invited users would be able to join. The process of matching users with sessions is called matchmaking.

VRNet makes use of this online subsystem to host, find and connect to sessions. It provides the necessary functionality that can be used out of the box in applications integrating the VRNet plugin. Since the usage of the OSS happens solely through its interfaces and no platform specific functionalities are needed for our purposes the VRNet implementation should be fully platform independent.

The relevant interfaces of the OSS used by VRNet are *IOnlineSubsystem* and *IOnlineSession*.

IOnlineSubsystem is the main interface to get a reference to the correct OSS instance and its sub services, like for example the ones implementing the Session Interface (IOnlineSession), Friends Interface, Leaderboard Interface and others.

IOnlineSession is the interface for online session services and used to manage sessions. It has methods for getting or removing named sessions, creating, starting, updating, destroying, finding and joining sessions, and for determining if a certain player is currently connected to a session. These actions are asynchronous and therefore have to be dealt with in an asynchronous fashion.

VRNetGameInstance The VRNet overridden *GameInstance* singleton object takes care of handling the interaction with the OSS interfaces. It provides methods for initiating session management actions like hosting, starting, finding and joining sessions in C++ as well as for Blueprints. Blueprint support here is especially important since this functionality will be needed in user interfaces, which are usually created in UMG Widget Blueprints. Additionally this class also responsible for handling connection functionality without using the OSS, for example to connect directly to an IP address.

4.4 Summary

During the design process the formal functional and non-functional requirements were obtained and refined. Furthermore, using these requirements a software architecture was developed for a UE4 plugin. The underlying UE4 constraints were presented and shown how the VRNet architecture builds on top of them. Many new classes deriving from the UE4 game framework core were introduced to provide easy-to-user multi-user functionality. The main contributions of this plugin include the client-authoritative VR user movement, avatar and interaction replication. Additionally many crucial elements were introduced that are necessary for networked applications in general, i.e. online session management and other game management functionalities.

The presented design results were further used to implement one of the practical parts of this thesis, the VRNet plugin, which will be described in the following chapter.

CHAPTER 5

VRNet Implementation

This chapter describes the implementation of the software architecture presented in Chapter 4. The implemented VRNet framework entities are presented in detail and important decisions and workarounds are described.

First, the plugin structure and an overview of its contents are described in Section 5.1. Then the core replication implementations of the VRNet plugin are described in Section 5.2. Section 5.3 presents the implementation details of the functionalities and components related to the players. The implementations related to the game management and online session management are described in Section 5.4. Furthermore, the plugin implementation includes several convenient utility tools for debugging multi-user VR applications, which are presented in Section 5.5. In addition to the functionality provided in the plugin, the implementation part also contains a template project that can be used as a starting point for new projects. This template project includes several example implementations that either can be used as-is or discarded and newly implemented. This project template as well as general descriptions on how to use the plugin are described in Section 5.6.

5.1 Plugin Structure

UE4 plugins have to follow a certain file structure and provide multiple specific files. The file structure of the VRNet plugin is shown in Figure 5.1. The main folder of the plugin is named after the plugin name and contains all plugin files.

Code plugins have a *Source* folder, which contains one or more directories with source code for the plugin. Plugins can include Blueprint and other asset files in its *Content* folder. Compiled code is generated in a *Binaries* folder, as well as temporary project files being stored in an *Intermediate* folder. The FilterPlugin.ini in the *Config* folder lists additional files which will be packaged along with the plugin, additional configuration files are also placed in this folder. The *Resources* folder can include additional resources

```
VRNetPlugin
├── Binaries
├── Config
│   └── FilterPlugin.ini
├── Content
│   └── (..blueprint files and assets)
├── Intermediate
├── Source
│   └── VRNetPlugin
│       ├── Public
│       │   ├── Interaction
│       │   │   ├── VRNetGrabbableBehavior.h
│       │   │   ├── VRNetGrabbableStaticMeshActor.h
│       │   │   └── VRNetGrabInterface.h
│       │   ├── Player
│       │   │   ├── Avatars
│       │   │   │   ├── VRNetAvatar.h
│       │   │   │   ├── VRNetDefaultAvatar.h
│       │   │   │   └── VRNetMotionCaptureAvatar.h
│       │   │   ├── ControllerAugmentations
│       │   │   │   ├── VRNetControllerAugmentation.h
│       │   │   │   ├── VRNetCtrlAugmentationComponent.h
│       │   │   │   └── VRNetDefaultCtrlAugmentation.h
│       │   │   ├── VRNetBasePawn.h
│       │   │   ├── VRNetBasePawnMovementComponent.h
│       │   │   ├── VRNetCameraComponent.h
│       │   │   ├── VRNetGrabbingComponent.h
│       │   │   ├── VRNetMotionControllerComponent.h
│       │   │   ├── VRNetPawn.h
│       │   │   ├── VRNetPawnMovementComponent.h
│       │   │   ├── VRNetPlayerController.h
│       │   │   └── VRNetPlayerState.h
│       │   ├── Testing
│       │   │   ├── TestPawn.h
│       │   │   ├── TestVRNetPlayerController.h
│       │   │   └── VRNetDesktopPawn.h
│       │   ├── Utils
│       │   │   ├── LoggingUtils.h
│       │   │   ├── VRNetBlueprintFunctionLibrary.h
│       │   │   └── VRNetStaticUtils.h
│       │   ├── VRNetDatatypes.h
│       │   ├── VRNetGameInstance.h
│       │   ├── VRNetGameModeBase.h
│       │   ├── VRNetGameStateBase.h
│       │   ├── VRNetReplicationComponent.h
│       │   └── VRNetPlugin.h
│       ├── Private
│       │   ├── (..cpp files)
│       │   └── VRNetPlugin.cpp
│       └── VRNetPlugin.Build.cs
├── Resources
│   └── Icon128.png
└── VRNetPlugin.uplugin
```

Figure 5.1: File structure of the VRNet plugin.

and has to provide an Icon128.png file, which is used as the plugin icon to display inside the UE4 editor.

In UE4, projects and plugins can be split into multiple modules. Each of those modules needs to have a C# file that controls how the module is built, including build options for defined module dependencies, include paths and so on. The VRNet plugin only consists of the *VRNetPlugin* module, which includes its *VRnetPlugin.Build.cs* file, where the dependencies to other modules are defined, e.g. the *"Core"*, *"OnlineSubsystem"* and many more UE4 modules. This build file is necessary for the UBT's build process.

The VRNet plugin source files and class names have to be prefixed with *VRNet* to avoid naming conflicts as UE4 does not support C++ namespaces. The *VRnetPlugin.h/.cpp* source files include the *VRNetPluginModule* class, which implements the *IModuleInterface* that can be used to change behavior on module startup and shutdown. The header files of the VRNet classes are separately placed in the *Public* folder, which can be seen in Figure 5.1. The cpp files on the other hand are placed in the *Private* folder with the same structure as the header files (hence they are omitted in the Figure). The class files are further structured for clarity. Classes related to player functionalities reside inside the *Player* folder. These include the Pawns, PlayerController and all their subcomponents. The *Interaction* folder contains the classes related to interactions. In the scope of this thesis only the grabbing is provided, but further interactions can be implemented here in the future. The player and interaction implementations are described in Section 5.3. Furthermore, utility and testing classes are provided in the *Testing* and *Utils* folders respectively. These classes will be explained in detail in Section 5.5. The remaining classes that do not fit into the mentioned categories are placed in the root directory. For example *VRNetDatatypes.h* contains datatype defintions that are used in several VRNet classes.

The *VRnetPlugin.uplugin* plugin descriptor file for the VRNet plugin is shown in Listing 5.1, with some of the unimportant entries omitted. This file describes information like the plugin name, description, category, version and author informations and is discovered and loaded automatically by the Engine. Additionally *CanContainContent* specifies if asset files are included and *EnabledByDefault* specifies if it should be enabled inside the editor by default. *Modules* contains one or several game modules that are included with the plugin, their name, type and when they should be loaded. The *Type* of the Module determines which types of applications this module is loaded in. In this case, Runtime modules will be loaded when the editor is running, as well as in the final shipped application, but it is also possible to specify a module to be loaded only in editor or development builds. *WhitelistPlatforms* can be used to restrict the platforms for which this module will be compiled for, which in the case of VRNet plugin is used to speed up the compilation process. In the case that packaging for other platforms is necessary this can be changed in the future. *Plugins* lists other engine plugins this plugin is depending on. VRNet plugin has a dependency to the *OnlineSubsystem* and *OnlineSubsystemUtils* plugins for the online session management functionality. If the plugin should be installed as an engine plugin it might become necessary to provide these dependencies in the

```
{
  "Version": 1,
  "FriendlyName": "VRNetPlugin",
  "Description": "VRNet Platform Code Plugin",
  "Category": "VR",
  "CreatedBy": "mwagnurr",
  "CanContainContent": true,
  "EnabledByDefault": true,
  "Modules": [
    {
      "Name": "VRNetPlugin",
      "Type": "Runtime",
      "WhitelistPlatforms": [
        "Win64"
      ]
    }
  ],
  "Plugins": [
    {
      "Name": "OnlineSubsystem",
      "Enabled": true
    },
    {
      "Name": "OnlineSubsystemUtils",
      "Enabled": true
    }
  ]
}
```

Listing 5.1: VRNet Plugin Descriptor File - VRNetPlugin.uplugin.

project's *.uproject* file instead of inside the plugin directly.

5.2 Replication

In this section, the important core replication implementations that are used and reused throughout the VRNet plugin are presented.

UE4 already provides **server-authoritative** movement replication for the RootComponents of Actors if the exposed *bReplicateMovement* flag is set to true. This leads to the location and rotation of the RootComponent to replicate to all clients, including the owning client. This default functionality can be used for example for the server-authoritative movement replication of StaticMeshActors representing movable objects in the world that are set to simulate physics, and do not need a custom implementation in the VRNet plugin.

In contrast, the necessary replication logic for **client-authoritative** replication is not provided by the engine and needs to be implemented in several VRNet classes. Because the core replication functionality is the same it makes sense to extract it into a generalized and reusable ActorComponent. This Component is called *VRNetReplicationComponent* and is described in detail in this Section.

5.2.1 VRNetReplicationComponent

VRNetReplicationComponent includes the core replication logic and can be easily integrated into Actors as well as Components. It provides optimized client-authoritative replication of location and rotation data of the specified SceneComponent and different interpolation implementations for smoothing out server updates. It can be activated and deactivated on runtime to start and stop the replication process, or set to auto activate for cases where it should be active for all its lifetime. From outside a SceneComponent that needs to be replicated has to be set. Furthermore, several configuration settings are exposed to customize the replication. These settings include the update rate and interpolation settings among others, which will be explained in detail later.

The data which gets replicated consists of a struct including the location *FVector* and rotation *FRotator* with different serialization compression options. UE4 allows to define custom network serialization for the data of USTRUCTs. Compression can be used to reduce the bandwidth but might come with visually noticeable inaccuracies. The location vector values can be specified to be rounded on serialization to two, to one, or no decimals. Alternatively the vector values can be sent without rounding, which provides the highest precision but needs the most bandwidth. For the FRotator serialization UE4 provides functions for compressing the FRotator values to either 8 bits or 16 bits.

The replicated Component's location and rotation can either be sent and updated in world space, meaning that they are relative to the world origin, or in relative space, where they are relative to their parent to which they are attached to. To control this a flag called *bRelativeSpace* is provided, which directs what methods are used to get and set the respective SceneComponent position. While using world space is the most straightforward, the relative space replication can be useful when the replicated Component moves independently of its parent.

Replication dataflow The majority of the replication logic is executed on each Tick, as can be seen in Listing 5.2. The following enumeration describes the dataflow of position replication of VRNetReplicationComponent:

1. **Position update on the local client:** On the controlling client the position updates (e.g. through tracking input) get applied directly to their Components. The client is authoritative of such position updates. It owns the Actors that contain the Component that needs to be replicated. The VRNetReplicationComponent is not concerned with how the position of its replicated Component are updated, but solely is responsible for their replication.

2. **Position replication to the server:** To replicate the position data the VRNetReplicationComponent sends it first to the server. The owning client is excluded by the server from receiving the position data again. On each Tick the Component calls on the owning client the *ReplicateMovementToServer()* method, which can be seen in Listing 5.3.

```
1  void UVRNetReplicationComponent::TickComponent(float DeltaTime, ELevelTick TickType,
       FActorComponentTickFunction* ThisTickFunction)
2  {
3     SCOPE_CYCLE_COUNTER(STAT_Tick);
4     Super::TickComponent(DeltaTime, TickType, ThisTickFunction);
5     if (!ShouldPerformTick()) return;
6
7     if (IsLocallyControlled())
8     {
9        ReplicateMovementToServer(DeltaTime);
10    }
11    else
12    {
13       if (ClientInterpolationMode == EClientInterpolation::NoInterpolation || IsNetMode
           (NM_DedicatedServer))
14       {
15          SetReplicationComponentPosAndRot(ReplicatedTransform.Location,
              ReplicatedTransform.Rotation);
16       }
17       else
18       {
19          InterpolateClientPosition(DeltaTime);
20       }
21    }
```

Listing 5.2: VRNetReplicationComponent TickComponent implementation.

In this method the server RPCs for sending the movement data get called depending on the specified *NetReplicationRate*, which controls the frequency of how often updates get send, similarly to the *NetUpdateRate* provided by UE4 Actors for their property replication. Furthermore, the server RPCs only get sent if there was a difference in the position data compared to the last frame to avoid unnecessarily wasted bandwidth (see line 12). The data sent to the server consist of the previously described *ReplicatedTransform* struct holding the location and rotation data. The server RPC execution is then simply updating the *ReplicatedTransform* property on the server instance which then gets replicated by UE4's property replication to all clients except the owning client.

In case the described owning client is also the server, i.e. the host, nothing gets actually sent over the network, but the execution sequence stays the same.

3. **Position replication to other clients:** The other machines, which can be clients and the server, update the replicated SceneComponents movement based on the server updates and the client interpolation settings.

   On each Tick of each of the other machines the local SceneComponent's position gets either set directly to the replicated values (i.e. when the server is a dedicated server or if no client interpolation is configured) or the *InterpolateClientPosition* method gets executed. In this method, if there is a change between the local and the replicated position data, a method implementing the respective set client interpolation mode gets called. How these methods' execution sequences look like

```
1  void UVRNetReplicationComponent::ReplicateMovementToServer(float DeltaTime)
2  {
3     NetReplicationTime += DeltaTime;
4     if (NetReplicationTime >= 1.0f / NetReplicationRate)
5     {
6        NetReplicationTime = 0.0f;
7        FVector CurrentPosition;
8        FRotator CurrentRotation;
9        GetReplicationComponentPosAndRot(CurrentPosition, CurrentRotation);
10
11       // check if change happend, if not -> don't send Transform
12       if (!CurrentPosition.Equals(ReplicatedTransform.Location) || !CurrentRotation.
             Equals(ReplicatedTransform.Rotation))
13       {
14          SCOPE_CYCLE_COUNTER(STAT_ReplicateMovementToServer);
15
16          ReplicatedTransform.Location = CurrentPosition;
17          ReplicatedTransform.Rotation = CurrentRotation;
18          ServerSetTransform(ReplicatedTransform);
19       }
20    }
21 }
```

Listing 5.3: VRNetReplicationComponent ReplicateMovementToServer implementation.

will be explained in the following subsection.

5.2.2 Client Interpolation

The Component provides different client interpolation modes that can be set to change the interpolation behavior for simulating clients. They are intended to smooth the movement of the replicated Component when the server updates come sparsely or when packets are lost. The different client interpolation modes are:

- No Interpolation: Without interpolation the replicated component position and rotation get updated directly on each Tick. This can lead to noticeable jittering if the network delay is high, the net update rate is low, or packets are lost.
- InterpToLastUpdate: This interpolation mode makes use of the UE4 *FMath::VInterpTo* and *FMath::RInterpTo* functions. These functions interpolate from the current to the target value, scaled by distance to the target so that it has a strong start speed and ease out. On each Tick the current value responds to the SceneComponent location and rotation value. The target value is the last received server position update. The results of the interpolation from the current to the target values are set as the new values for the SceneComponent. A parameter for the interpolation speed is provided and can be changed for configuration.
- LerpToLastUpdate: This interpolation mode provides linear interpolation between the SceneComponent's position and the last position update. The implementation can be seen in Listing 5.4. The interpolating alpha value is calculated in line 6 taking

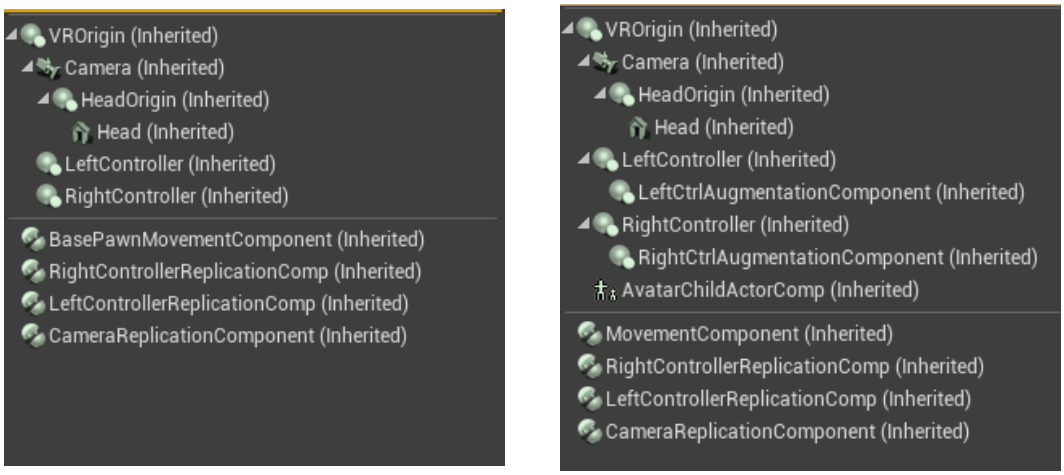
```
1  void UVRNetReplicationComponent::LerpToLastUpdate(float DeltaTime)
2  {
3     NetReplicationTime += DeltaTime;
4
5     const float LerpLimit = 1.0f;
6     float Lerp = FMath::Clamp(NetReplicationTime * NetReplicationRate, 0.0f, LerpLimit);
7
8     if (Lerp >= LerpLimit)
9     {
10       SetReplicationComponentPosAndRot(ReplicatedTransform.Location,
             ReplicatedTransform.Rotation);
11       LerpStartTransform = FTransform(ReplicatedTransform.Rotation, ReplicatedTransform
             .Location);
12       NetReplicationTime = 0.0f;
13    }
14    else
15    {
16       SetReplicationComponentPosAndRot(FMath::Lerp(LerpStartTransform.GetLocation(),
             ReplicatedTransform.Location, Lerp), FMath::Lerp(LerpStartTransform.Rotator
             (), ReplicatedTransform.Rotation, Lerp));
17    }
18 }
19
20 void UVRNetReplicationComponent::OnRep_ReplicatedTransform()
21 {
22    FVector CurrentPosition;
23    FQuat CurrentRotation;
24
25    GetReplicationComponentPosAndRot(CurrentPosition, CurrentRotation);
26    LerpStartTransform = FTransform(CurrentRotation, CurrentPosition);
27    NetReplicationTime = 0.0f;
28 }
```

Listing 5.4: VRNetReplicationComponent LerpToLastUpdate implementation.

the currently passed time since the last received server update and the exposed configuration variable *NetReplicationRate* into account. This value denotes the position between the start and end values of the linear interpolation. On each Tick the *LerpToLastUpdate* method is called where the local SceneComponent position is set to the next interpolated value until it reaches the time limit specified by the *NetReplicationRate* (see line 16). When this *LerpLimit* is reached the position gets immediately updated to the replicated server position and the *NetReplicationTime* is reset.

Each time UE4 successfully replicates new position data the *OnRep_ReplicatedTransform()* method gets called (see line 20). This denotes the starting point of the interpolation and resets the start position, as well as the *NetReplicationTime*. This interpolation procedure results in the linear smoothing of the current SceneComponent's position towards the updates received from the server thus leading to reduction in the visible jitter.

Although other, more sophisticated interpolation approaches exist, the implemented ones offer a sufficiently smooth experience. An example of another interpolation mode that



(a) VRNetBasePawn derived Blueprint

(b) VRNetPawn derived Blueprint

Figure 5.2: Screenshot of the Blueprint Component view of the VRNetBasePawn and VRNetPawn.

could be added to the VRNetReplicationComponent in the future is an interpolation that takes timestamps of the received position updates into account and replays movement slightly delayed on simulating clients.

5.3 Player Functionalities

In this section, the implementation details of the classes and concepts directly related to users are described.

VRNetBasePawn The implementation of *VRNetBasePawn* primarily includes the creation and configuration of its Components. This Pawn defines its Actor scene graph with a simple origin SceneComponent as its RootComponent and attached to it the *VRNetCameraComponent* and *VRNetMotionControllerComponents* for the left and right controllers. The Component hierarchy can be seen in Figure 5.2a. Each of the *MotionControllerComponents* is set up during the construction process by defining their *MotionSource* to be *FXRMotionControllerBase::LeftHandSourceId* and *RightHandSourceId* respectively.

For a visual representation of the player head the VRNetBasePawn provides an optional *Head* static mesh. This mesh is attached to a *HeadOrigin* SceneComponent that is further attached to the CameraComponent and can be disabled when deriving the class. The *Head* mesh is set to render only for other players, not the owner itself to not obstruct their vision and to avoid self-collisions. The HeadOrigin can be used to add an offset of the Head position to the actual tracked HMD.

VRNetBasePawn implements networked possession and unpossession events that are not provided by UE4 out-of-the-box. The implementation can be seen in the code

```
1  void AVRNetBasePawn::PossessedBy(AController* NewController)
2  {
3     Super::PossessedBy(NewController);
4     MultiPossessed();
5  }
6
7  void AVRNetBasePawn::UnPossessed()
8  {
9     Super::UnPossessed();
10    MultiUnPossessed();
11 }
12
13 void AVRNetBasePawn::MultiUnPossessed_Implementation()
14 {
15    VRN_LOG_I(LogVRNetBasePawn, Log, HasAuthority(), GetUniqueID(), "
          MultiUnPossessed_Implementation() ...")
16    K2_ClientUnPossessed();
17 }
18
19 void AVRNetBasePawn::MultiPossessed_Implementation()
20 {
21    VRN_LOG_I(LogVRNetBasePawn, Log, HasAuthority(), GetUniqueID(), "
          MultiPossessed_Implementation() ...")
22    K2_ClientPossessed();
23 }
```

Listing 5.5: VRNetBasePawn possession implementation.

Listing 5.5. This means that on the server's *PossesdBy* event, which gets executed during the PlayerController possession process, the VRNet implementation calls a NetMulticast RPC called *MultiPossessed*. This NetMulticast RPC gets executed on all machines, including the server itself, and then further calls an VRNet provided event that can be implementable in either C++ or Blueprints. This leads to the possibility to implement this event similarly as one would implement the actual UE4 provided possession event, but assuring that it gets executed on each machine. This logic is the same for the unpossession event.

The implementation of **VRNetMotionControllerComponents** extends the UE4 MotionControllerComponent by including VRNetReplicationComponent and setting it to auto activate and to use relative space.

The **VRNetCameraComponent** implementation also includes the VRNetReplicationComponent with auto activation and using relative space set to true. Additionally, the UE4 flag is set to lock to the HMD if its containing Pawn is currently locally controlled. This assures that only on the locally controlling client the camera is locked to the HMD, but not on the other machines. This setting is important since simulated clients receive their position updates through replication instead of tracked data.

VRNetPawn The *VRNetPawn* implementation additionally to its VRNetBasePawn parent class creates and sets up the ChildActorComponent for *VRNetAvatar*, and the *VRNetCtrlAugComponent* for each controller. Both components are described in detail

in following subsections. The *VRNetPawn* class overrides the possession/unpossession multicast of the VRNetBasePawn class to additionally notify the Controller Augmentations and Avatar about these events. The Component hierarchy as shown in the UE4 Blueprint editor can be seen in Figure 5.2b.

5.3.1 Input Handling

Input handling is an important aspect in the creation of VR applications and therefore also had to be considered in detail for the VRNet plugin implementation. Defining input in UE4 is primarily done through user defined input bindings in the application settings. There are two types of input bindings: *Action Mappings*, which stand for key presses and releases, and *Axis Mappings*, which are for inputs that have a continuous range. In code it is possible to bind delegate functions to *InputComponents* to these defined input bindings.

The input is managed in UE4 in the PlayerController class, where a stack of InputComponents is built that consume input depending on their position in the stack, with the highest priority on the top.

The built stack looks as follows from top to bottom:

1. Custom input stack - manually added InputComponents. e.g. Actors with Input enabled
2. PlayerController InputComponent
3. LevelScriptActors InputComponents (possibly several)
4. Pawn sub Components that are InputComponents (possibly several)
5. Pawn InputComponent

This offers multiple possibilities to control input consumption and the definition of input bindings. By default input gets consumed and not passed down the stack if an input binding is defined in one of the InputComponents, but it can explicitly be set at the binding to not consume it and pass it further.

In the VRNet plugin the provided default input bindings are primarily bound and handled in the ControllerAugmentation Actor of the respective controller, but they can also be defined in the Pawn or PlayerController itself. For example, the *TestVRNetPlayerController* binds and overrides input to emulate the movements of the normally tracked Components with keyboard and mouse inputs. The input binding mappings are set through the configuration file *DefaultInput.ini* or inside the UE4 editor, which manipulates this file.

5.3.2 Controller Augmentations

The VRNet plugin includes an abstract base Actor called *VRNetControllerAugmentation*. This class adds additional controller-specific functionalities to the tracked MotionController, which were described in the VRNet plugin architecture description in Section 4.3.

ControllerAugmentations are implemented as Actors, because including a hierarchy of potentially multiple SceneComponents is not possible inside a Component. The instantiation and attachment of this Actor to the VRNetPawn is handled inside a Component called *VRNetCtrlAugmentationComponent*.

VRNetCtrlAugmentationComponent This SceneComponent is responsible of the management of the spawning, setting up, attaching and destroying the ControllerAugmentation. It is attached to the VRNetMotionControllerComponent inside the VRNetPawn.

Contrary to the UE4 provided *ChildActorComponent*, which conceptually does a similar thing, it spawns the Actor on BeginPlay, additionally provides logic for setup of the Actor and simplifies access to specific functionality. These benefits and the necessity for custom attachment logic, as well as the Actor not really needing to be spawned before BeginPlay, are the reasons why a custom SceneComponent is chosen to be implemented in the VRNet plugin. Necessary information can be passed from the VRNetPawn to the contained VRNetCtrlAugmentationComponent and further to the VRNetControllerAugmentation through a *ControllerAugmentationSettings* struct. This information includes which hand it represents, how it should get attached and if input should be enabled, consumed and default input bindings should be used.

Attachment of the ControllerAugmentation can either be set to its managing VRNetCtrlAugmentationComponent or to the RootComponent of the parent VRNetPawn. This offers the possibility to make the ControllerAugmentation independent of the movement of the MotionControllerComponent if necessary.

On BeginPlay the *CreateControllerAugmentation()* method gets called on the server instance of the Component, where the previously set ControllerAugmentation class is passed to the *World->SpawnActorDeferred()*. This method will spawn the Actor without running its construction and BeginPlay scripts to give the opportunity to change settings beforehand. In this initialization period the ControllerAugmentationSettings are passed to the freshly created instance. Then the *UGameplayStatics::FinishSpawningActor()* method is called to finish the spawning of the Actor. Afterwards, the attachment logic is executed depending on the passed attachment settings described earlier.

Before the creation process a check executes to check if a creation cycle exists, to avoid an infinite cycle when the selected class to be spawned is already an owner of the VRNetCtrlAugmentationComponent.

This VRNetCtrlAugmentationComponent can be extended to change and enhance the ControllerAugmentation functionalities, for example to implement the handling of multiple different ControllerAugmentations per controller.

Input Handling in ControllerAugmentations: The VRNetControllerAugmentation base class handles the initiation of the default input bindings on each possession event if the *bUseDefaultInputBindings* option is set and defines the *AVRNetControllerAugmentation::GetInputBindingSuffix()* function, which returns the input binding suffix depending on the ControllerAugmentationType (Left or Right controller) of the current ControllerAugmentation. While the base class provides the handling logic, actual input bindings need to be defined through overriding the *AVRNetControllerAugmentation::BindDefaultInputBindings()* method. An example listing of an input binding in C++ can be seen in listing 5.6.

```
1 InputComponent->BindAction(*FString("VR_DefaultCtrlAug_Grab_" + InputBindingSuffix),
      EInputEvent::IE_Pressed, this, &AVRNetDefaultCtrlAugmentation::OnGrabPressed).
      bConsumeInput = Settings.bConsumesInput;
2
3 InputComponent->BindAction(*FString("VR_DefaultCtrlAug_Grab_" + InputBindingSuffix),
      EInputEvent::IE_Released, this, &AVRNetDefaultCtrlAugmentation::OnGrabReleased).
      bConsumeInput = Settings.bConsumesInput;
```

Listing 5.6: Input Binding for Grabbing in VRNetDefaultCtrlAugmentation.

5.3.2.1 VRNetDefaultCtrlAugmentation and BP_VRTemplateCtrlAug

The plugin includes the default implementations of the ControllerAugmentation showcasing the usage and providing some useful VR functionalities like grabbing interaction, teleportation and UI Interaction. The *VRNetDefaultCtrlAugmentation* class contains a StaticMeshComponent representing the visual mesh of the controller, and a sphere collider Component attached to it. To implement the grabbing interaction functionality the *VRNetGrabbingComponent* is integrated. This component works in conjunction with the collision detection and the bound input.

Additionally to the VRNetDefaultCtrlAugmentation class we implemented a Blueprint subclass called *BP_VRTemplateCtrlAug* which further enhances the base implementation with visual representations, e.g. for the teleportation. Some assets and parts of the teleportation logic from the official UE4 VR project template are taken and integrated into the VRNetDefaultCtrlAugmentation.

The Components of this Blueprint subclass are shown in Figure 5.3. On the left hand side of this figure is the SceneComponent hierarchy and on the right side the viewport with the Component meshes is shown. The *HandMesh* SkeletalMeshComponent is set to the UE4 default robot hand. The inherited ControllerMesh StaticMeshComponent is not used in this case, but could be used to for example add a controller mesh inside the hand. The red sphere indicates the debug view of the collider sphere.

The VRNetDefaultCtrlAugmentation and BP_VRTemplateCtrlAug classes can both be used as-is or as a foundation for further functionalities. The BP_VRTemplateCtrlAug Blueprint class and its assets are provided in the VRNet project template instead of directly in the plugin, to not clutter up the core plugin with showcase example assets.

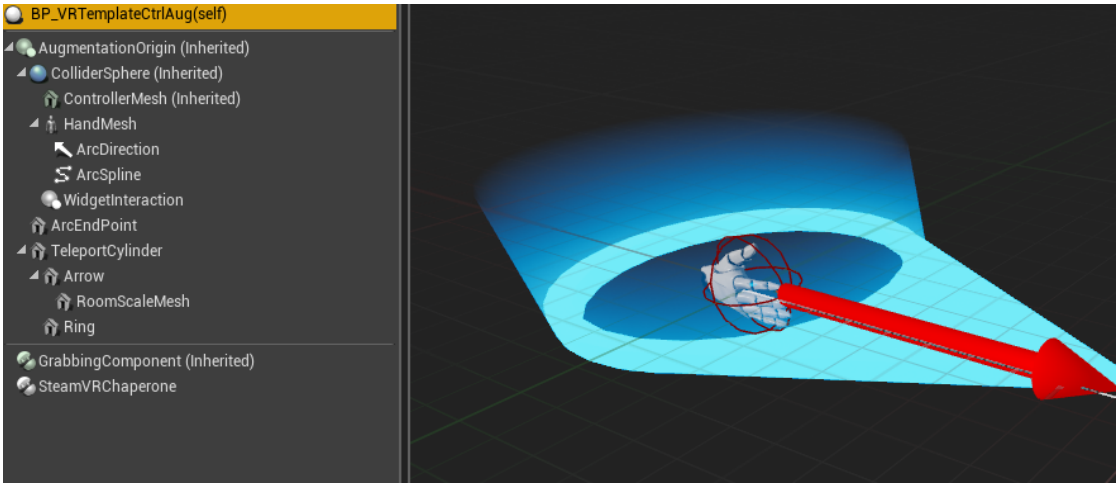


Figure 5.3: Screenshot of the VRNet BP_VRTemplateCtrlAug deriving from VRNetDefaultCtrlAugmentation.

Grabbing The grabbing interaction logic is completely implemented in the plugin classes, i.e. the VRNetDefaultCtrlAugmentation and its contained VRNetGrabbingComponent. The implementation works as follows. When another Actor in the world begins to overlap with the collider SphereComponent and the VRNetGrabbingComponent is not currently grabbing something, it checks if the overlapping Actor implements the IVRNetGrabInterface. If it does implement the interface the reference to this Actor is set as the currently overlapped Actor. When the input button for grabbing is pressed it initiates the *GrabObject()* request through the GrabbingComponent, and *DropObject()* when it is released. The implementation logic for the grabbing inside the VRNetGrabbingComponent will be explained in more detail in the subsection 5.3.4. The BP_VRTemplateCtrlAug additionally provides animation of the hand bones depending on the current grab state.

Teleportation: The teleportation logic is partly implemented in the VRNetDefaultCtrlAugmentation and its Blueprint subclass BP_VRTemplateCtrlAug. The process of teleportation as seen by the player is illustrated in Figure 5.3.

To start teleportation, a player presses the touchpad button on a controller. A teleportation arc is displayed upon the touchpad button press. The future destination is indicated by the oriented circle (see Figure 5.3) on the floor at the end of the arc. When the player releases the touchpad button they get teleported to the location of the circle. The new orientation of the player's viewport is determined by the direction of the arrow of the circle.

The ControllerAugmentation includes an *ArrowComponent* attached to the controller mesh, which indicates the direction for determining the teleport destination. The tracing works via simulating throwing a projectile in this direction. This simulated projectile traces against only the static world and then uses the hit against a *NavMesh* (Navigation

Mesh) to find a place for the player to stand on. It is required that such a NavMesh is placed inside the World to define the areas the players are supposed to be moving on.

The teleportation movement is handled in conjunction with the *VRNetBasePawnMovementComponent* of the VRNetBasePawn, where the *TeleportPawn()* method gets called with the previously determined teleportation destination as the parameter. This method executes the teleportation on the local client immediately and then sends a server RPC to execute the teleportation on the server. The server then replicates the new teleported position to the other clients.

Furthermore, for the visual respresentation of the teleportation the ControllerAugmentation includes a *SplineComponent* that uses the tracing projectile path and morphs it into a curve and spawns cylinder meshes. The result of this visual representation of the teleportation can be seen in Figure 5.4. On each Tick this arc spline is first cleared and its meshes destroyed, then the teleportation destination is traced as described earlier via simulation of a projectile. Then after the tracings and settings of the booleans controlling the visibility of the teleportation meshes the *UpdateArcSpline* function gets called with the teleportation destination and the series of positions of the traced projectiles path as the parameters. If a valid destination was found previously this method is then building a curved spline from all the trace path points and adding cylinder SplineMeshComponents for each of them. This then creates an animated and smoothly curved teleportation arc.

Additionally, the BP_VRTemplateCtrlAug adds arrow, ring and room scale meshes for the visual representation of the teleportation destination as shown in Figure 5.4.

UI Interaction: Basic UI interaction is provided through the inclusion of a WidgetInteractionComponent which allows interaction with WidgetComponents. From the player's perspective, the interaction with widgets is achieved through the use of a yellow ray that comes out of the virtual hand (controller representation). Internally, WidgetInteractionComponent sends interaction signals to the widgets that simulate key presses, e.g. mouse clicks when pressing the bound "VR_DefaultCtrlAug_UI" input. On each Tick the implementation draws a yellow line starting from the controller if the WidgetInteractionComponent is currently hovering over an interactable UI Widget.

5.3.3 Player Avatars

Similarly to ControllerAugmentations, VRNetAvatars need to be implemented as Actors to allow sub hierarchies of SceneComponents. But in contrast to ControllerAugmentations, the coupling of VRNetAvatar to VRNetPawn is achieved through the UE4-provided *ChildActorComponent*.

ChildActorComponent: ChildActorComponent spawns any specified generic Actor when it gets registered and destroys it when it is unregistered. Registration of a Component in UE4 is executed to register it inside its owning Actor where it will be set up and added to its Components array and is automatically done when the Components

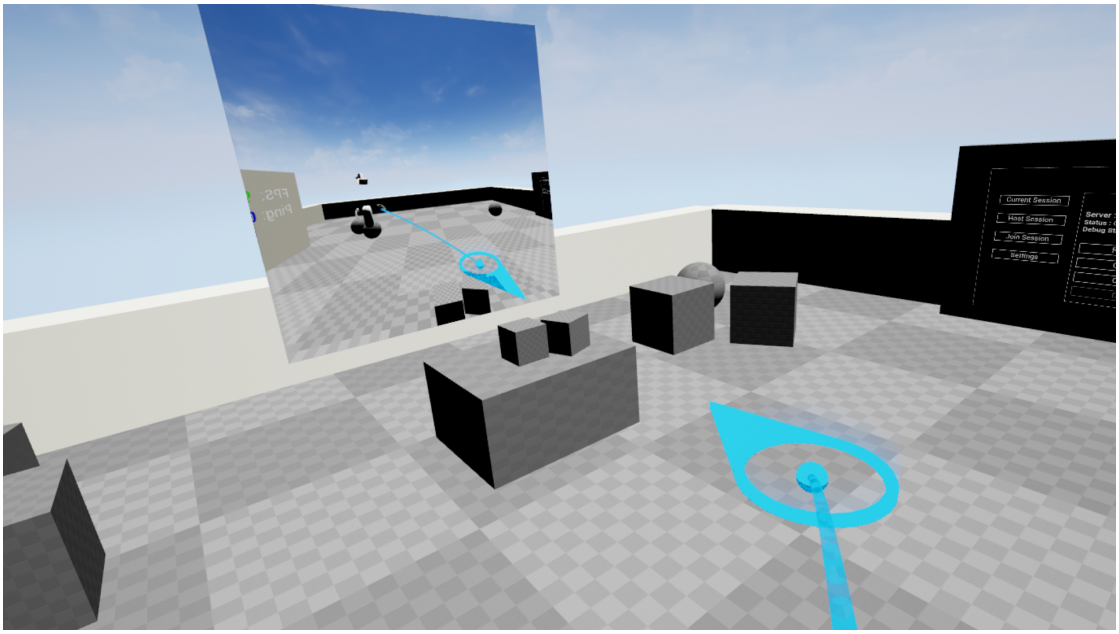


Figure 5.4: Screenshot of the Teleportation implemented in VRNet BP_VRTemplateCtrlAug.

are created in the construction phase of the Actor. The construction of Actors happens already at engine startup, where they also get cached. Because of this the ChildActor is already spawned when opening the Blueprint containing the ChildActorComponent and it is visible in the Blueprints viewport as if it were a Component. The access to the exposed variables of the ChildActor is provided through a template Actor instance, whose set values are copied on the creation of the real ChildActor instance.

Using this UE4-provided Component brings a lot of complex code related to the handling of ChildActors after the registration process, which is important for VRNetAvatars due to their visual nature and importance to see them in the viewport during development. They also bring some issues with them though, some of which are due to engine bugs that are still unresolved and others due to corruptions happening occasionally to Blueprints. For example, packaging the project with ChildActorComponents sometimes produces hard to resolve errors and additionally they may cause a minor hit on performance. Despite those issues we still decided to use this construct for the VRNetAvatar implementation because it worked stable most of the time and we think that many of the issues that occur every now and then will be fixed in future Unreal Engine releases.

VRNetAvatar structure: The *VRNetAvatar* Actor is a base class that contains a CapsuleComponent as its RootComponent, which can be used for collision queries or as a simple geometrical representation of the player. An ArrowComponent is attached to it to indicate the direction in which the Avatar is facing.

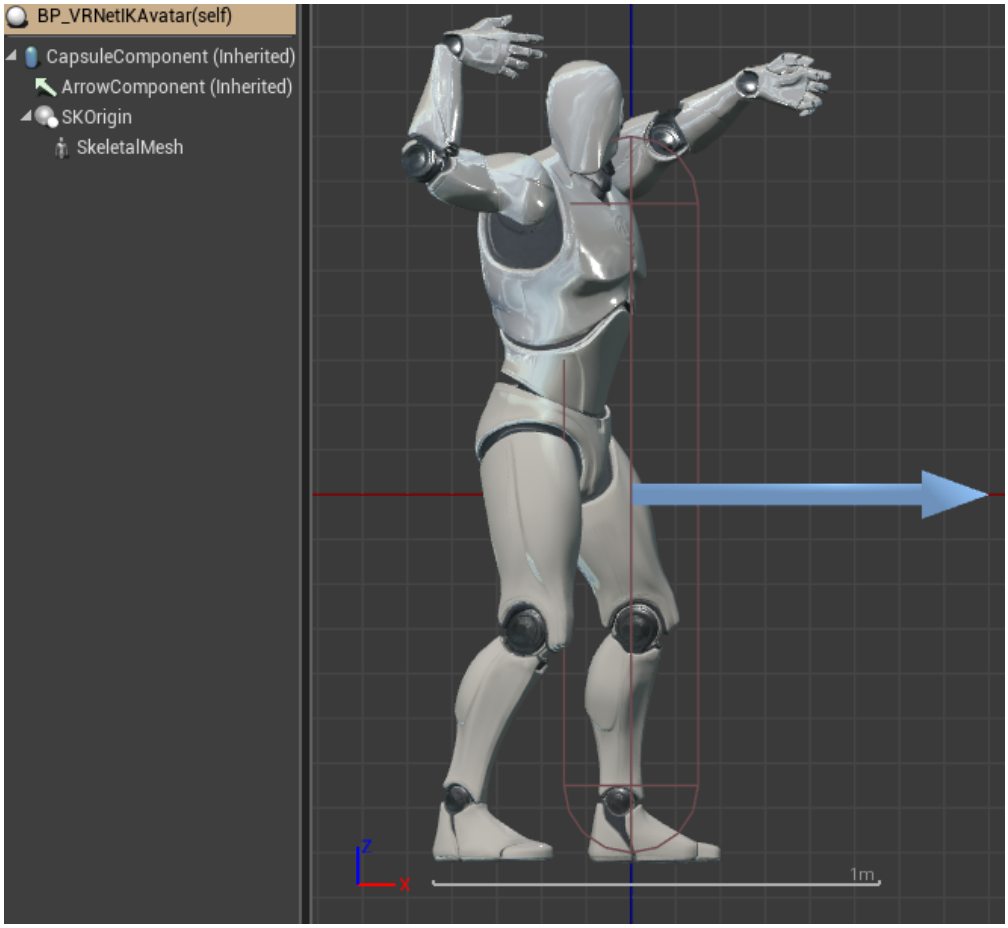


Figure 5.5: Screenshot of an example VRNetAvatar with a skeletal mesh.

Figure 5.5 shows the structure and viewport of a Blueprint example deriving from VRNetAvatar.

Apart of its own Components, the VRNetAvatar base class holds references to important tracked Components like the controllers and the HMD to make use of them. These references need to be set from outside the class on its initialization. The VRNetPawn implementation sets default references to the respective controllers and sets the head to follow the HMD. It is possible though to override this, for example in blueprints, through the unique *FName* of a certain Component. If specified and valid, this reference name gets used to query the parent Actor and return the actual Component reference if it exists. This functionality exists to provide developers with the means for example to change the head pivot reference to a SceneComponent attached to the tracked HMD with an offset, making it easy to tweak the head position without changing the implementation.

Furthermore, the VRNetAvatars get notified by the VRNetPawn on possession and unpossession events to offer the possibility to react to them. For this the *OnParentPawn-*

```
1  void AVRNetAvatar::ApplyMovementToRoot(float DeltaTime)
2  {
3     SCOPE_CYCLE_COUNTER(STAT_ApplyMovementToRoot);
4     if (GetHeadComponent())
5     {
6        FVector PivotLocation = GetHeadComponent()->GetComponentLocation();
7        float CapsuleHalfHeight = (PivotLocation.Z + 5.0) / 2.0;
8
9        FVector NewCapsuleLocation = FVector(PivotLocation.X, PivotLocation.Y,
             CapsuleHalfHeight);
10       CapsuleComponent->SetCapsuleHalfHeight(CapsuleHalfHeight);
11       CapsuleComponent->SetWorldLocation(NewCapsuleLocation);
12
13       FRotator CurrRotation = CapsuleComponent->GetComponentRotation();
14       FRotator PivotRotation = GetHeadComponent()->GetComponentRotation();
15
16       FRotator NewRotation = FRotator(CurrRotation.Pitch, PivotRotation.Yaw,
             CurrRotation.Roll);
17       CapsuleComponent->SetRelativeRotation(NewRotation);
18    }
19 }
```

Listing 5.7: VRNetAvatar ApplyMovementToRoot implementation.

Possessed() and *OnParentPawnUnpossessed()* methods get called inside the VRNetPawn.

VRNetAvatar movement handling The default movement handling implementation of the VRNetAvatar handles the movement of its RootComponent relative to the player's HMD position. On each Tick the *ApplyMovementToRoot()* method is called, which can be seen in Listing 5.7. The movement application inside this method takes the HMDs world location as its pivot and calculates the CapsuleComponents half-height depending on it. Furthermore it sets the new world location of the CapsuleComponent center (being the horizontal and vertical center of the capsule) to the X and Y location of the HMD and for the Z axis the calculated capsule half-height. The Yaw rotation of the HMD is used as the new rotation for the CapsuleComponent, making it rotate around the up axis only. Since the movement gets applied to the root CapsuleComponents, these movements get also applied to its children SceneComponents attached to it, which can be StaticMeshComponents, SkeletalMeshComponents or any other SceneComponents.

However the specific movement logic can be overwritten in subclasses to customize the movement behavior for a specific Avatar.

5.3.3.1 Example VRNet Avatars

VRNetDefaultAvatar The plugin includes a default implementation of a VRNetAvatar, which simply adds a StaticMeshComponent attached to the RootComponent. This class is sufficient for simple static body representations. An example of the default VRNetPawn that is used in the template is shown in Figure 5.6. The arrow illustrates the forward direction of the avatar, the red capsule indicates the CapsuleComponent used in collision detection. The HMD mesh follows the head rotation of the user.

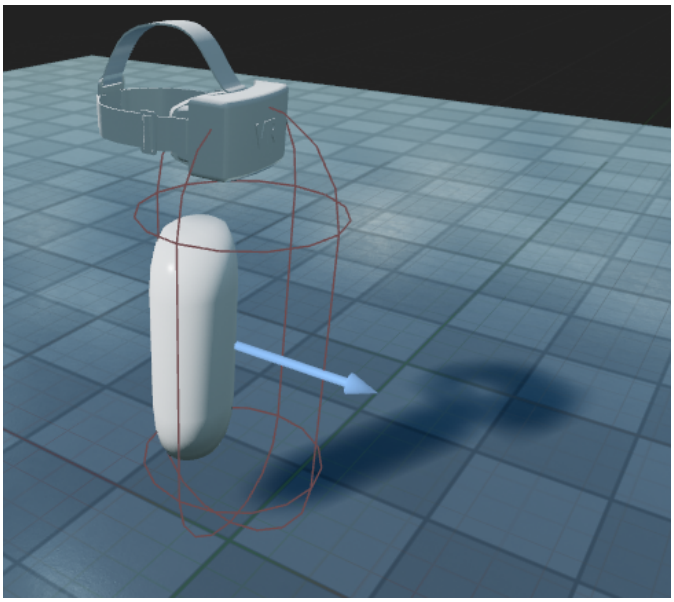


Figure 5.6: Screenshot of a VRNetPawn with a simple VRNetDefaultAvatar.

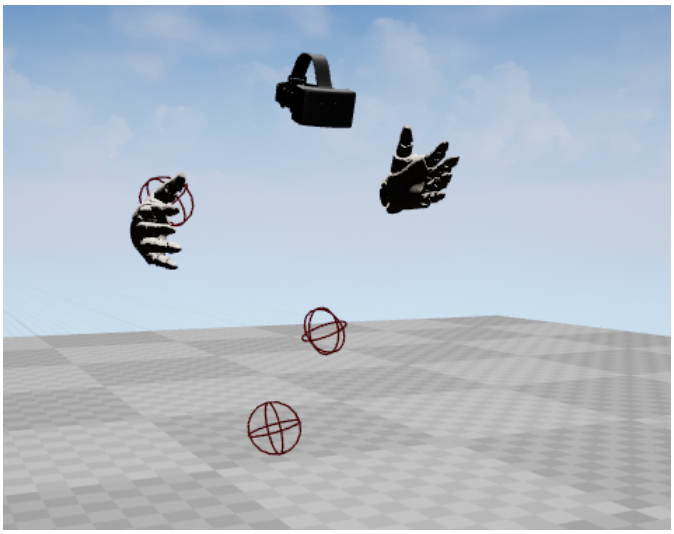


Figure 5.7: Screenshot of an example VRNetMotionCaptureAvatar with three Trackers in debug view.

VRNetMotionCaptureAvatar The VRNetMotionCaptureAvatar is a more complex example that adds functionality for replicating the tracking input of motion capture suites or other techniques providing full-body avatar animation. Contrary to most other Avatars this Avatar needs to be set to replicate, since it contains networking functionality of the Tracker position updates. It provides handling and replication of multiple additional Trackers that can be used as reference points in the Avatar representation, for example in the animation of a SkeletalMesh.

The data structure used for the Trackers is a struct containing the replicated data, i.e. their unique tracking device ID and their tracked transform as well as non-replicated data that needs to be set up and updated locally on each machine, i.e. a SceneComponent, for representing the Tracker in the UE4 level.

Access to the tracking platform is provided in UE4 via the *IXRTrackingSystem* interface. For initializing the Trackers the *IXTrackingSystem::EnumerateTrackedDevices()* method is used, which populates an array of tracked device IDs of the type *EXRTrackedDeviceType::Other*. With these Tracker device IDs the previously described Tracker struct gets initialized and stored in a map for quick access. Additionally, an array of the replicated Trackers gets created that then gets sent via server RPC to initialize the setup process on the server and the other connected clients. As soon as the server initializes its Tracker map and array, the array gets replicated to all clients not owning the Actor through property replication and on its completion the Tracker map gets initialized on those clients. At this point the data structures are initialized on each machine.

Listing 5.8 shows the implementation of the local Tick execution and the *GetTrackerDevicePose()* method. On each Tick, the locally controlled Avatar collects the Tracker transform data via its *GetTrackerDevicePose()* method, which wraps the interface call *IXTrackingSystem::GetCurrentPose()*. The benefit of wrapping this call is that it is easy to override this method in subclasses to provide randomly generated Tracking position data for testing purposes, for example. After the position data got collected for each Tracker an array of transforms get send via a server RPC. The server then updates its replicated Tracker array and the UE4 property replication notifies the other clients. The updates to the replicated Trackers then get incorporated on each machine into the Tracker map to set the location and rotation of their respective SceneComponent.

To showcase and test this functionality Blueprint Actors deriving from the VRNetMotionCaptureAvatar are provided in the VRNet template. *BP_MotionCaptureAvatar* adds SphereCollisionComponents to each Tracker, with their collision profile set to respond to physics bodies and the shape visualized for debug view. Figure 5.7 shows this Avatar with its visualization. The *BP_TestMotionCaptureAvatar* further replaces the real tracked data with generated trackers that randomize their tracking positions. This overridden emulating behavior was useful in the development and debugging of the Avatars functionality.

```
1  void AVRNetMotionCaptureAvatar::TickLocalUpdateTrackers()
2  {
3     SCOPE_CYCLE_COUNTER(STAT_TickLocalUpdateTrackers)
4     if (ReplicatedTrackers.Num() > 0)
5     {
6        for (FReplicatedTracker& currReplTracker : ReplicatedTrackers)
7        {
8           FQuat Orientation;
9           FVector Position;
10          if (GetTrackerDevicePose(currReplTracker.TrackerID, Orientation, Position))
11          {
12             currReplTracker.Transform.Position = Position;
13             currReplTracker.Transform.Rotation = Orientation;
14          }
15       }
16       ServerUpdateTrackers(ReplicatedTrackers);
17    }
18 }
19
20 bool AVRNetMotionCaptureAvatar::GetTrackerDevicePose(int32 DeviceId, FQuat&
       OutOrientation, FVector& OutPosition)
21 {
22    if (GEngine && GEngine->XRSystem.IsValid())
23    {
24       return GEngine->XRSystem->GetCurrentPose(DeviceId, OutOrientation, OutPosition);
25    }
26    return false;
27 }
```

Listing 5.8: VRNetMotionCaptureAvatar local server updates implementation.

5.3.4 Grabbing Interaction

In the previous Controller Augmentation Section 5.3.2 we described how the VRNetDefaultCtrlAugmentation implements the input and collision handling to initiate the grabbing interaction. This section takes a more detailed look onto the interaction components that implement the grabbing.

When a user is touching an object he can press and hold the trigger button to initiate the grab. Then the object gets attached to the controller and stays attached until the trigger button is released and the drop mechanism is called. Through this attachment the object is following the movements of the controller. Figure 5.8 shows the grabbing in action.

The implementation of this functionality is split into the logic on the grabbable object side, as well as the side of the grabbing player.

Grabbable object Objects, or Actors to be precise, need to implement the *IVRNetGrabInterface* to be grabbable. As soon as they implement this interface they will be considered for the grabbing interaction. To showcase the grabbing interaction a StaticMeshActor that implements the *IVRNetGrabInterface* called *VRNetGrabbableStaticMeshActor* is included in the plugin. This Actor contains a StaticMeshComponent, which can be set to any static mesh, and the VRNetGrabbableBehaviorComponent.

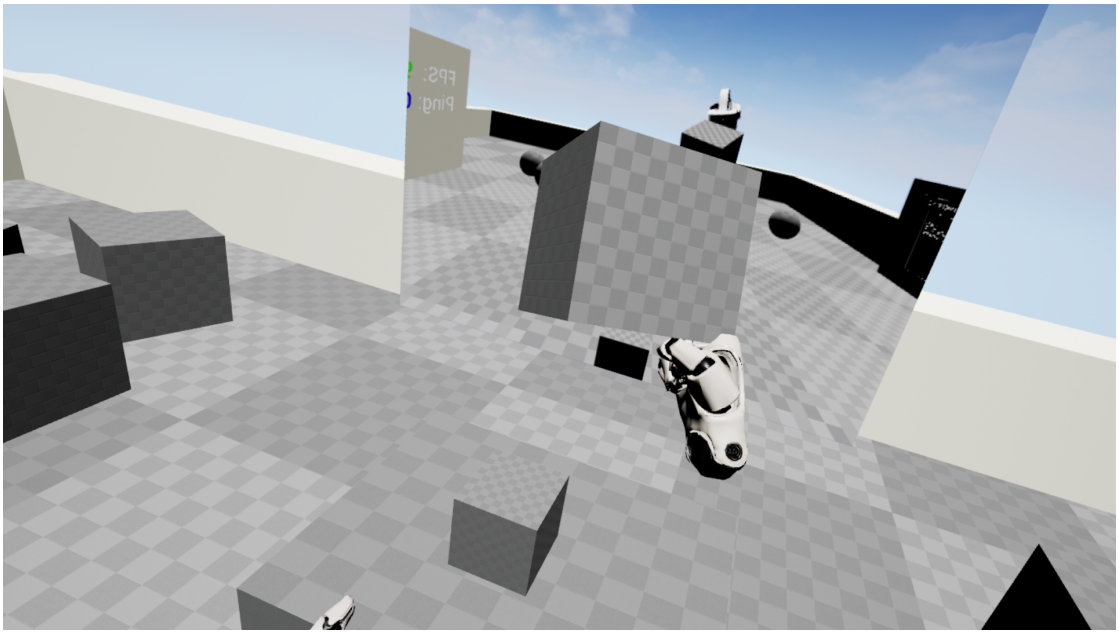


Figure 5.8: Screenshot of the grabbing interaction with objects in VRNet.

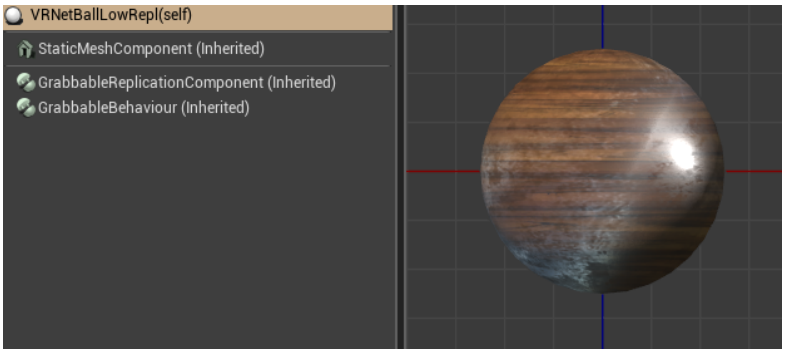


Figure 5.9: Screenshot of an example VRNetGrabbableStaticMeshActor.

Figure 5.9 shows the Components and viewport of an example VRNetGrabbableStaticMeshActor that can be placed in the level and is grabbable by players.

The VRNetGrabbableBehaviorComponent needs to be initialized with a reference to a PrimitiveComponent it is defining the behavior for, which is done here with setting the StaticMeshComponent in the constructor. For the replication the VRNetGrabbableBehaviorComponent includes a VRNetReplicationComponent, which was previously described in detail in section 5.2. Depending on the interaction replication mode this Component is used to start client-authoritative replication when the object is being grabbed, and turned off to switch to the default server-authoritative replication with simulated physics when it is dropped.

```
1  bool UVRNetGrabbableBehaviour::OnGrab(USceneComponent* InGrabbingController, bool
       clientAuthoritative)
2  {
3     if (!bIsGrabbed)
4     {
5        bIsGrabbed = true;
6        GrabbableComponent->SetSimulatePhysics(false);
7        GrabbingController = InGrabbingController;
8
9        if (clientAuthoritative)
10       {
11          ReplicationComponent->Activate(true);
12          MultiOnRequestCASuccess();
13       }
14       return true;
15    }
16    return false;
17 }
```

Listing 5.9: VRNetGrabbableBehaviour OnGrab implementation.

When the player initiates the grab interaction the *IVRNetGrabInterface::Grab()* method of the object is called. In the implementation of the interface then solely the *OnGrab* method of the GrabbableBehaviorComponent gets executed. This method's implementation can be seen in Listing 5.9. If the object is not currently grabbed it deactivates the physics simulation of its grabbable Component, stores a reference to the grabbing Controller, sets its Pawn as the owner of the VRNetGrabbableStaticMeshActor and then activates the VRNetReplicationComponent if the Grab was requested in the client-authoritative mode. It returns a boolean whether the grab was successful or not.

On calling the *IVRNetGrabInterface::Drop()* method, similarly the call gets passed to the Behavior Component. It then deactivates the Replication Component, removes the previously set references, sets the physics simulation for the StaticMeshComponent to true again and if successful the *OnDrop* method returns true.

The described behavior is important for the default implementation of the VRNet grabbing interaction to work, but could look differently depending on the interaction type and the desired behavior on the object side.

Grabbing player On the player side the collision detection and grabbing initiation via input is handled in the *VRNetDefaultCtrlAugmentation* as described in subsection 5.3.2.

When the *GrabObject()* method gets executed in the VRNetGrabbingComponent it sends a server RPC with the Actor reference of the Actor that should get grabbed. The *ServerGrabObject()* method, which is executed on the server, can be seen in Listing 5.10. If the passed Actor implements the *IVRNetGrabInterface* the *IVRNetGrabInterface::Grab()* method is called with its controller reference and the configured replication type (see line 5 in the Listing). Then the previously explained logic on the object side is executed. The replication type is set in the VRNetGrabbingComponent at runtime or before to either

```
1  void UVRNetGrabbingComponent::ServerGrabObject_Implementation(AActor* objectToGrab)
2  {
3     if (IVRNetGrabInterface* Grabbable = Cast<IVRNetGrabInterface>(objectToGrab))
4     {
5        bool grabRequestSuccessful = Grabbable->Execute_Grab(objectToGrab,
             ParentSceneComponent, ReplicationType == EGrabReplicationType::
             ClientAuthoritative);
6
7        if (grabRequestSuccessful)
8        {
9           GrabbedObject = objectToGrab;
10          if (ReplicationType == EGrabReplicationType::ClientAuthoritative)
11          {
12             objectToGrab->SetOwner(GetOwner());
13             ClientOnRequestCASuccess(Cast<UPrimitiveComponent>(objectToGrab->
                   GetRootComponent()));
14          }
15          else
16          {
17             objectToGrab->AttachToComponent(ParentSceneComponent,
                   FAttachmentTransformRules::KeepWorldTransform);
18          }
19       }
20    }
21 }
```

Listing 5.10: VRNetGrabbingComponent ServerGrabObject implementation.

Server Authoritative, *Client Authoritative* or *Only Local Client*. Depending on if the grab was successful on the object side, it then attaches the object Actor to its controller either on the server in the case of server-authoritative replication (Listing line 17), or after a client RPC on the owning client if the replication type is client-authoritative (Listing line 13). If the mode is *Only Local Client* it attaches on the local client without replication, but tells the server to stop replicating the movement data.

The dropping logic work analogous to the grabbing, but instead of attaching the Components it detaches them.

Though, the grabbing interaction could be implemented in several different ways. An implementation different from the one described above could provide a better match for the requirements of a specific project other than our template project. To enable an easy replacement of the grabbing interaction implementation, the attachment logic is handled completely inside the VRNetGrabbingComponent on the player side. This way, only the player side logic would need to be replaced to change the actual attachment. As an example of an alternative implementation of the grabbing functionality, UE4 PhysicsHandles could be used. PhysicsHandles offer the benefit of keeping physics simulation enabled during interaction as well as the possibility of grabbing an object with both controllers at the same time.

Figure 5.10: Screenshot of the VRNetGameSettings inside the VRNetGameModeBase.

5.4 Game Management Functionalities

The VRNet plugin provides extended UE4 game framework classes for basic game management functionalities as well as blueprint-accessible implementations for matchmaking and session management. UIs are also provided that can be used as-is or extended for VR and desktop projects. The relevant OSS platform configurations and considerations are described in Subsection 5.4.3.

VRNetGameModeBase is responsible for defining and managing *VRNetGameSettings*, a class containing arrays of *VRNetSelectablePawns* and *VRNetSelectableMaps*. The UE4 editor's exposed *VRNetGameSettings* and DefaultClasses for spawning are shown in Figure 5.10. A VRNetSelectablePawn consists of a key, display name and Pawn class.

```
1  void AVRNetGameModeBase::ChangePawn(FString NewPawnKey, AController* Controller)
2  {
3     // 1. destroy previous pawn (if exists)
4     if (Controller->GetPawn()) Controller->GetPawn()->Destroy();
5
6     // 2. select pawn class to spawn and spawn the new pawn at the found PlayerStart
7     TSubclassOf<APawn> PawnClassToSpawn = SelectPawnClass(NewPawnKey);
8     APawn* SpawnedPawn = SpawnPawn(PawnClassToSpawn, FindPlayerStart(Controller));
9     if (SpawnedPawn)
10    {
11       // 3. possess spawned pawn (if spawning was successful)
12       Controller->Possess(SpawnedPawn);
13
14       if (AVRNetPlayerState* VRNetPlayerState = Cast<AVRNetPlayerState>(Controller->
             PlayerState))
15       {
16          FVRNetSelectablePawn FoundSelectablePawn;
17          if (UVRNetStaticUtils::GetSelectablePawnByKey(NewPawnKey, VRNetSettings.
                SelectablePawns, FoundSelectablePawn))
18          {
19             // 4. set SelectedPawn in VRNetPlayerState
20             VRNetPlayerState->SetSelectedPawn(FoundSelectablePawn);
21          }
22       }
23    }
24 }
```

Listing 5.11: VRNetGameModeBase ChangePawn implementation.

VRNetSelectableMaps include the map name, which is the name of the level asset file, and display name. These settings also get passed to *VRNetGameStateBase* to be replicated to all machines.

Preferred Pawn and Change Pawn The VRNetGameModeBase overrides the *GameModeBase::InitNewPlayer()* method to parse the incoming clients connection options to extract the preferred Pawn key, e.g. a string looking like this: *?preferredPawn=myPawnKey*. This preferred Pawn string is then stored into the players *VRNetPlayerState*.

The GameModeBase spawns Pawns of players in the *GameModeBase::RestartPlayer()* method. This either happens when a player is connecting, or when a restart is triggered differently. In the *VRNetGameModeBase* class, the *RestartPlayer()* method is overwritten. The overwritten method executes *VRNetGameModeBase::ChangePawn()* if a valid pawnKey is provided. Listing 5.11 shows the *ChangePawn()* implementation. The *VRNetGameModeBase::ChangePawn()* method first destroys any previous Pawn if it exists. Then the class of the Pawn to spawn gets selected by matching the passed Pawn key to the SelectablePawn in the game settings' *SelectablePawns* array. With the Pawn class reference the *VRNetGameModeBase::SpawnPawn()* method gets invoked, which spawns the Pawn at a spawn position defined by a passed Actor that is the designated player start. After the successful spawning the Pawn gets possessed by the players PlayerController and the VRNetSelectedPawn gets set in the VRNetPlayerState. The

ChangePawn method can also be called independently and is executed by clients with the method *VRNetPlayerController::ChangeSelectedPawn()*, which does so through a server RPC.

The preferred Pawn key can be passed on the connection either through passing it with the server IP address when starting through the command line, or in the connections handled through *VRNetGameInstance* which is further described in 5.4.1.

5.4.1 VRNet Online Session Management

The majority of the VRNet session management implementation lies in the VRNetGameInstance class, which is an application-wide singleton object without internal replication, meaning that the client and server instances are independent of each other and cannot communicate.

VRNetGameInstance provides methods for hosting, starting, finding and joining Server Sessions. Internally these methods first get a reference to the correct OSS instance via *IOnlineSubsystem::Get(FName SubsystemName)* and afterwards a reference to its session management service, which implements the *IOnlineSession* interface. Since all of the online session interface actions are asynchronous the implementation makes heavy use of delegates when calling functions. Delegates are function pointers that can get executed at a later time, i.e. when the actions finished. It is important to wait for delegates before calling functions further down the line, since failing to do so can cause crashes and unexpected behavior, for example in timeout cases. The delegates are added right before calling the appropriate function and cleared from within itself. Additionally VRNetGameInstance provides its own delegates that can be registered in blueprints and get called on the completion of each respective internal delegate call, which can be used by developers using the VRNet plugin to write additional logic for handling session functionalities in Blueprints. These delegates are named *OnVRNetHostOnlineSessionDelegate, OnVRNetStartOnlineSessionDelegate*, and so on.

Hosting and Starting Sessions For hosting sessions a set of properties are put into a *Session Settings* struct that determine its characteristics, like the number of players, if it is a LAN session, if it should be advertised for matchmaking and allow joining. For simplicity some of those settings are already preset by the VRNet implementation to default values, e.g. *shouldAdvertise = true*. The others can be changed by the provided method parameters, i.e. *IsLan, MaxNumPlayers, ServerName*, etc. These settings get passed to the *IOnlineSession::CreateSession()* function, which asynchronously takes care of the session creation. Additionally, another VRNet method parameter is the *MapName* string, which does not get passed to the OSS but gets stored to be later used for opening this Map. After the session got successfully created the *OnCreateSessionComplete* delegate gets executed and the session is created but not started.

Since the session is created but not started it is then necessary to call *IOnlineSession::StartSession()* to mark it as being in progress. VRNet implementation provides an

option to set the *directlyStartSession* to true to start the session directly after the session got created. Alternatively, *StartSession()* can be called manually if Lobby functionality is needed. After the session got successfully started the previously specified Map gets opened and relevant Actors get notified (e.g. the VRNetGameState is notified to update its replicated session infos).

Finding Sessions The session finding works with defining session search parameters and calling the *IOnlineSession::FindSessions()* method. The actual available functionality depends a lot on the used OSS platform and its implementation of the IOnlineSession interface. The UE4 provides the default *OSS Null* platform that only supports LAN queries. For using other OSS platforms like the *OSS Steam* platform and its matchmaking it may be needed to provide additional configuration settings to make the search functional, as will be described further in section 5.4.3.

The VRNet implementation provides parameters to specify *IsLanQuery* and *MaxSearchResults* that are populated in *OnlineSessionSearch*, which is an OSS object that stores all the session search query settings. The *FindSessions()* method is then called with *OnlineSessionSearch* as a parameter to start the search. After the search is completed the respective delegate is called, where the search results are iterated by the VRNet implementation and each result is wrapped to a Blueprint useable struct that exposes useful session informations like the owning username, the number of currently connected players and the still available player slots, as well as storing the internal C++ session result object to easily connect to the found session at a later point. Then the *OnVRNetFindOnlineSessionDelegate* broadcasts these Blueprint wrapped search results.

Another important aspect during development and testing with the OSS Steam platform is setting the *MaxSearchResults* parameter. Steam test matchmaking is shared by developers around the world, and this fact can affect the number of session results found. UE4 filters out sessions not belonging to this exact UE4 application, but it could happen that actual sessions of this application are not found if the maximum number of session search results is too low. Because of this the *MaxSearchResults* parameter that is exposed in the VRNet method should be used with a sufficiently high number to assure to actually find relevant sessions. For an example, we were consistently testing successfully with a value of 100.

Joining Sessions Finally, for joining a session the previously found session search result object gets passed to the *IOnlineSession::JoinSession()* method. On its successful completion the client has joined the session, but is still not physically connected to the Server and its current level. For this the resolved connect string provided by the session interface gets used to connect to the server via the UE4 *PlayerController::ClientTravel()* function.

OSS independent connection Additionally the VRNetGameInstance provides OSS independent hosting and connection functionality, which uses directly the engine provided

hosting and connection functionality via IP.

In UE4, game instances can turn into *Listen Servers* through loading the level with an appended *?listen* parameter, either at startup or through the method *UWorld::ServerTravel()*. The VRNetGameInstance provides a Blueprint exposed *ServerTravel* method with *MapName* and *AsListenServer* parameters that calls the internal ServerTravel function with its provided parameters. Additionally a *BeginHostingWithoutSession()* method is provided further streamlining the hosting process, which calls the *VRNetGameInstance::ServerTravel* method.

Joining by IP, similarly to a successfully completed Join Session request, is handled via calling the *PlayerController::ClientTravel()* method with the server's IP as its address. This IP parameter string value is validated for a minimal length of 7, to avoid unnecessarily trying a connection attempt to an invalid address. For all joining connections it also appends previously specified option query parameters to the address, as for example the previously explained *preferredPawn* parameter.

5.4.2 User Interface

A user interface created with the UMG UI designer tool is included in the VRNet template project to access the game and session management functionalities. It can be taken as-is, modified or simply seen as a showcase of how to use these plugin functionalities.

The VRNet UI is made with *Widget Blueprints* and consists of a main widget and several sub widgets. It can be added to the view port to access it like a traditional desktop menu or can be placed in the world, e.g. attached to the provided *BP_WidgetUIActor*. In the case that it is placed in the world the Pawn needs to include a *WidgetInteractionComponent* to interact with the UI. When using the VRNetPawn it is possible to use the VRNetDefaultCtrlAugmentation, which readily implements this UI interaction.

The main screen of the VRNet interface is illustrated in Figure 5.11. This UI widget is split up into a main section on the right side where the sub widgets are displayed depending on the control widget switcher buttons on the left side. The switchable sub widgets are *Current Session*, *Host Session*, *Join Session* and *Settings*.

Current Session The Current Session widget shows information about the current session, including the session name and status, the connected users and their user informations. Figure 5.11 shows the widget on the right side. Debug stats display the current Ping time and frames per second (FPS) on the machine.

The UI contains several buttons for basic game management control of the current session:

- *Restart Session* reloads the current map in a way so that connected clients stay connected, leading to a reset of the level.
- *Change Pawn* opens a popup that contains a VR friendly way to select a new Pawn that the player wants to change to. Only Selectable Pawns defined in the

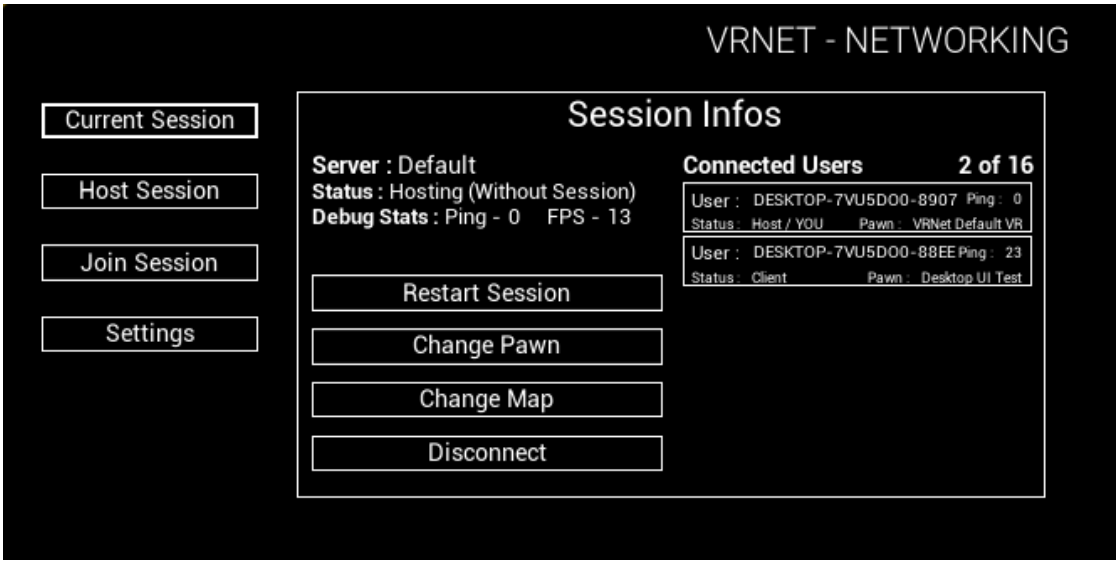


Figure 5.11: Screenshot of the current session menu of the VRNet networking user interface.

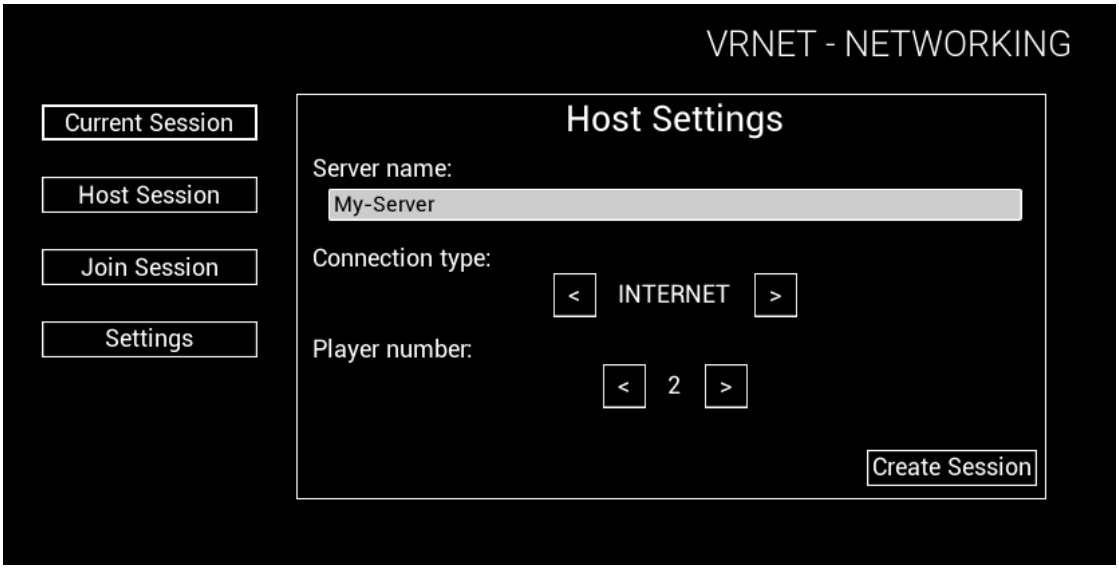
VRNetGameModeBase's VRNetGameSettings are listed here. This popup window is another sub widget.

- *Change Map*, similarly to Change Pawn, opens a popup where it is possible to change to a map contained in the list of Selectable Maps.
- *Disconnect* closes the session and disconnecting all connected clients if the host. As the client it is simply disconnecting from the server.

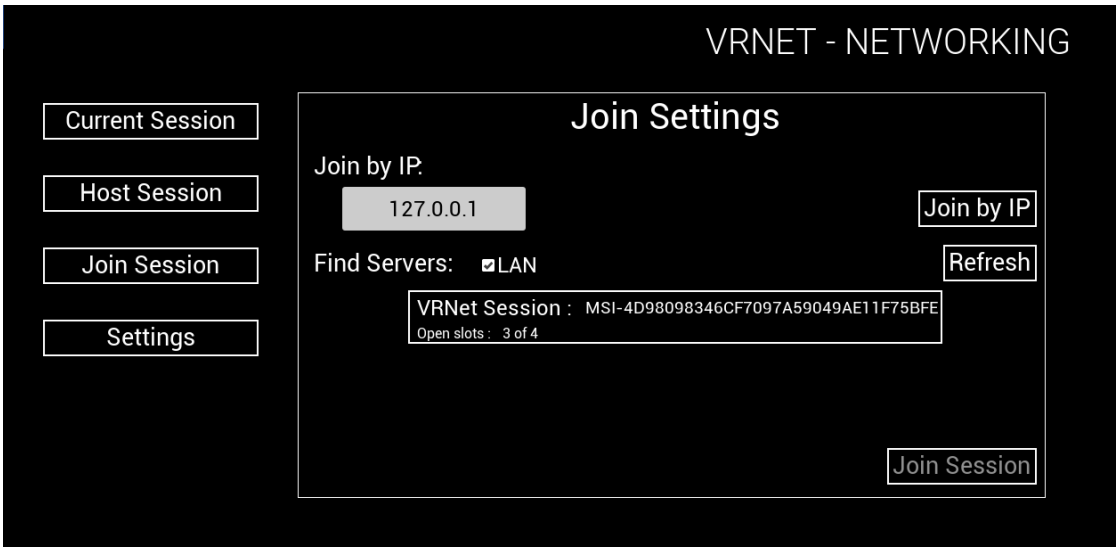
Host Session The Host Session widget allows users to enter a server name, specify if it should be an online or LAN session and define the maximum number of clients that can connect. The Host Settings tab is shown in Figure 5.12a. The server name gets automatically populated with the computer name or the Steam account user name when Steam is used.

Join Session The Join Session widget, which is shown in Figure 5.12b, includes a field to enter an IP address and a "Join by IP" button to start the connection process to the server with the provided address. This uses the direct connection functionalities without the OSS of the VRNetGameInstance.

Additionally it is possible to look for sessions either over Internet or LAN, which is controlled by the LAN checkbox and the "Refresh" button that initiates a search. When sessions are found they are listed with their session name and how many open slots for clients are available versus the maximum number of clients that can be connected. It is



(a) Host session menu



(b) Join session menu with its list of found sessions

Figure 5.12: Screenshots of the host / join session menus of the VRNet networking UI.

```
[/Script/Engine.GameEngine]
+NetDriverDefinitions=(DefName="GameNetDriver",
DriverClassName="OnlineSubsystemSteam.SteamNetDriver",
DriverClassNameFallback="OnlineSubsystemUtils.IpNetDriver")

[OnlineSubsystem]
DefaultPlatformService=Steam

[OnlineSubsystemSteam]
bEnabled=true
SteamDevAppId=480
bRelaunchInSteam=false
bAllowP2PPacketRelay=true

[/Script/OnlineSubsystemSteam.SteamNetDriver]
NetConnectionClassName= "OnlineSubsystemSteam.SteamNetConnection"
```

Listing 5.12: Configuration entries for Steam OSS in the DefaultEngine.ini file.

then possible to click on one of the results and then press the "Join Session" button to start the connection to the session process.

Settings The settings widget offers the possibility to specify the *Preferred Pawn* a user would like to spawn as initially after connection or level restart. Any other potential game settings can be added in this widget.

5.4.3 OSS Platforms

Due to the abstract nature of the OSS it is possible to use different platforms through configuration and without changing the implementation in the VRNet plugin. Using the UE4 provided default platform *"OSS Nullplatform"* already offers simple session functionalities and matchmaking that works for LAN.

For matchmaking over the Internet it is necessary to use a different OSS platform, like for example the *OSS Steam* platform. The OSS Steam API provides most of the functionality offered by the Steamworks SDK [15] from Valve. To use this platform and its functionalities it is important to provide additional configuration settings and that the application meets Valve's requirements. Applications using the Steam OSS must have a previously registered, valid application ID. For testing purposes it is possible to use the app ID 480, which is a test application ID that is shared by all Steamworks SDK developers. Certain additional configuration settings need to be provided in the *DefaultEngine.ini* file, as shown for example from the VRNet project template in listing 5.12.

Another important step for defining the OSS platforms is to include the module dependencies in the *.Build.cs* files, e.g. for Steam OSS it is necessary to add "OnlineSubsystemSteam" to the *DynamicallyLoadedModuleNames*. To use any OSS functionality the "OnlineSubsystem" needs to be added to the *PublicDependencyModuleNames*, which is done in the *VRNetPlugin.Build.cs*.

5.5 Debugging Utilities

Distributed applications can be very hard to debug. Due to code being executed on different machines it is important to have good logging output to analyze and detect problems in the implementation and understand the state the machines are in. The Logging Subsection 5.5.1 describes the tools provided by UE4 and the VRNet plugin to help understand the application state and control the output logs.

Additionally, being able to quickly test code changes is crucial for developing and debugging software. VR applications have to be tested with a sufficiently spacious setup of HMD, controllers and base stations, but sometimes it is beneficial to emulate the tracking with easily available input devices like keyboard and mouse. The Subsection 5.5.2 describes the implemented tools in the VRNet plugin to help with testing.

5.5.1 Logging

UE4 provides a sophisticated logging framework, with possibilities to control which logs get printed, as well as to filter the output log inside the engine. Log verbosity levels can be used in each log statement to define on which configurable level it should be printed. This allows to keep very detailed log statements in the code and activate them only when necessary. The log verbosity can be configured on compile-time, either in the code itself or the engine configuration files, and can also be changed at runtime with console commands.

UE4 provides a C++macro for printing logs in C++called *UE_LOG*:

```
#define UE_LOG(CategoryName, Verbosity, Format, ...)
```

VRNetPlugin defines additional logging macros to add more information that is useful for distributed applications, as shown in Listing 5.13. The *VRN_LOG* macro adds an *IsServer* parameter to print whether the log is executed on a machine with either SERVER or CLIENT authority. The *VRN_LOG_I* macro further adds a *UniqueID* to discern between different Actor instances.

```
1 #define VRN_LOG(LogCat, Verbosity, IsServer, FormatString, ...)
2 UE_LOG(LogCat,Verbosity,TEXT("%s: %s"), (IsServer ?TEXT("SERVER") : TEXT("CLIENT")), *
      FString::Printf(TEXT(FormatString), ##__VA_ARGS__ ) )
3
4 #define VRN_LOG_I(LogCat, Verbosity, IsServer, UniqueID, FormatString, ...)
5 UE_LOG(LogCat,Verbosity,TEXT("%s [%d] - %s"), (IsServer ? TEXT("SERVER") : TEXT("CLIENT
      ")), UniqueID, *FString::Printf(TEXT(FormatString), ##__VA_ARGS__ ) )
```

Listing 5.13: Logging macros defined in the VRNet plugin.

Furthermore it is possible to define custom Log Categories to easily identify to which category a log statement belongs to. In the VRNet plugin this is used to define categories for each class. An example of the definition and the usage is shown in Listing 5.14.

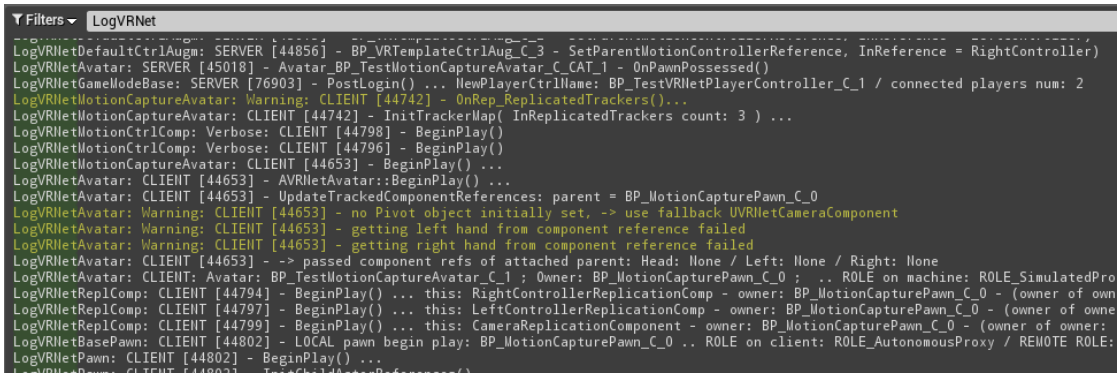


Figure 5.13: Screenshot of the output log filtered with "LogVRNet".

```
1 DEFINE_LOG_CATEGORY_STATIC(LogVRNetCtrlAugmentation, Log, All)
2
3 void AVRNetControllerAugmentation::BeginPlay()
4 {
5    VRN_LOG_I(LogVRNetCtrlAugmentation, Log, HasAuthority(),
6     GetUniqueID(), "%s-BeginPlay()...", *GetName())
7
8    Super::BeginPlay();
9 }
```

Listing 5.14: Log category definition and usage example in VRNetControllerAugmentation.cpp

It is possible to filter the output log for the category or part of the category, e.g. *"LogVRNet"* to show only VRNet logs. An example of logging output filtered to "LogVRNet" is shown in Figure 5.13.

5.5.2 Testing

Testing multi-user VR applications brings several difficulties. SteamVR is designed to run a single VR application at a time and automatically closes additional applications or instances of the same application. This makes it hard to test networked functionality on the same local machine. One workaround to solve this issue is to start a server instance in the Unreal Engine editor and another standalone clientafterwards. In this case, the standalone client connects directly to the localhost IP address. This works because SteamVR does not close the UE4 editor instance and offers the possibility of testing simple networked functionalities on the same machine. This method proved to be very useful for quick debugging. For more thorough testing it is necessary to deploy the project on multiple machines and connect them via local network or Internet.

To debug applications without using the VR devices the VRNetPlugin provides a *TestVRNetPlayerController*, which consumes previously defined input bindings for moving

the most important tracked SceneComponents: the camera and the motion controller components. The example testing setup looks like this:

- Modifier keys are pressed to control which Component should be moved (i.e. Shift for the camera, Ctrl for the left motion controller and Alt for the right motion controller).
- Keyboard buttons control the translation of the respective Component (i.e. WASD to move on the X-Y plane and QE for translating along the Z axis).
- Mouse movements change the rotation of the respective Component.

The input bindings define input actions for the modifier keys, e.g.:
"VR_Test_Modifier_MC_L", and input axis for the movement directions, as for example:
"VR_Test_forward".

Automated Testing The UE4 *Automation System* allows to create and execute various automated tests. This system provides several out-of-the-box utilities to help write tests and easily execute them in the *Session Frontend* tool of the editor and export the data.

Designing automated tests for UE4 applications or in game development in general is challenging, especially for experimental gameplay code. Even though many tools are provided it can be hard to implement tests that have dependencies to other engine parts, which can lead to having to mock away many of the sub systems in a time consuming way. When implementing experimental prototypes it may not be applicable to invest time in automated testing. However, automated testing can be useful in the development of a framework that is intended to be used as a base for many projects and to include stable functionality. In this case, automated testing can give assurance that crucial parts stay functional when changes are made. Another important benefit of writing automated tests is that to make the code more testable it is often necessary to refactor the code, making it also more readable and maintainable.

For the VRNet plugin the implementation of automated tests sadly was out of scope of this master thesis, but should be definitely considered to be implemented in the future to improve the code base.

5.6 Plugin Usage

This section explains how developers can integrate and use the VRNet plugin for UE4 projects. A UE4 project template is also provided. The manual setup steps described in this section are readily implemented in the template. Additionally, the template includes example usages and a working project that can be used as a starting point of development. This template and its content is further described in subsection 5.6.1.

The VRNet plugin needs to be packaged with the same engine version that the project is using.

To include the plugin it is necessary to provide either the packaged plugin in the projects *Plugin/* folder or to the *Engine/Plugins* folder if it should be included with the engine rather than the project. This is mostly a question of preference and depends on the specific situation. Theoretically the plugin could also be published and provided through the UE4 marketplace, which installs its plugins as engine plugins. If a plugin is not installed through the marketplace not having the plugin included in the project folder might add additional overhead for developers, who need to make sure the plugin gets installed on their developer machine before the project becomes usable. If it is included in the project itself and checked into version control of the project, developers just need to check out everything and it will work. This latter variant has the disadvantage that it adds duplicate code to the version control.

Usage of the Core Functionalities The following list describes the necessary steps to use the core functionality of the VRNet plugin:

1. The packaged VRNet plugin needs to be made available for the project. It is necessary to include it either in the projects *Plugin/* folder, or in the *Engine/Plugins* folder.

2. The VRNetPlugin needs to be enabled, either in the UE4 editor, or including the following entry in the *.uproject* file:

```
"Plugins": [
    {
        "Name": "VRNetPlugin",
        "Enabled": true
    }
 ]
```

3. A Blueprint or C++ class has to be created deriving either from *VRNetBasePawn* or *VRNetPawn*. In this class the defaults like static meshes for the players can be set. This step can range from simple player representations to also creating a *VRNetAvatar* or *VRNetControllerAugmentation* for including more complicated visual representation and logic.

4. In a GameModeBase Actor (created either through C++ or Blueprints) the *DefaultPawnClass* needs to be set to the previously created Pawn class. When starting the game it automatically spawns this Pawn for each player and if SteamVR is correctly set up and the HMD and controllers tracked it should be possible to start in VR mode and work.

Usage of Game and Session Management Functionalities To use the game and session management functionalities of the VRNet plugin the Project Settings (accessible through the *Editor->Project Settings*, or through *Config/DefaultEngine.ini* [/Script/EngineSettings.GameMapsSettings]) have to set the *GameInstanceClass* and *GlobalDefaultGameMode* to the VRNetGameInstance or VRNetGameModeBase respectively, or any sub classes of them. Inside the VRNetGameModeBase the following default classes, or any of their subclasses need to be selected:

- *Game State Class* -> VRNetGameStateBase
- *Player Controller Class* -> VRNetPlayerController
- *Player State Class* -> VRNetPlayerState

The VRNetGameModeBase then spawns the VRNet classes, which can be used to access the game and session management functionalities. Furthermore, in the Game Mode it is possible to set Selectable Pawns in the *VRNetGameSettings*: e.g. a Pawn with the key *myPawn*, as was shown in an earlier Figure 5.10.

The game can then be started via a shell command looking similar to this to connect to the IP address of a running server (in this example Localhost) and spawn as a myPawn Pawn:

```
UE4Editor.exe MyProject.uproject 127.0.0.1?preferredPawn=myPawn
```

Overwriting ActorComponent Classes When deriving from one of the VRNet Pawns it is useful to understand how ActorComponents can be overwritten or deactivated in the subclass.

In UE4 for object creation the *ObjectInitializer* is used to, for example, create the Components in the constructor of an Actor. Through the ObjectInitializer it is also possible to override Components with sub classes deriving from them. If the Components are marked at their creation as optional they can also be disabled. This happens with passing an ObjectInitializer instance that got set up accordingly to the Super constructor. Listing 5.15 shows an example of a chained ObjectInitializer setup.

```
1 ObjectInitializer.DoNotCreateDefaultSubobject(AExampleActor::ComponentName).
      SetDefaultSubobjectClass<UMyExampleComponent>(AExampleActor::AnotherComponentName)
```

Listing 5.15: ObjectInitializer example for disabling and overwritting ActorComponents.

5.6.1 Project Template

The VRNet template includes the previously described steps, as well as adds example implementations and sets up scenes for testing and showcasing. This project template can be used as a starting point when creating a multi-user VR project.

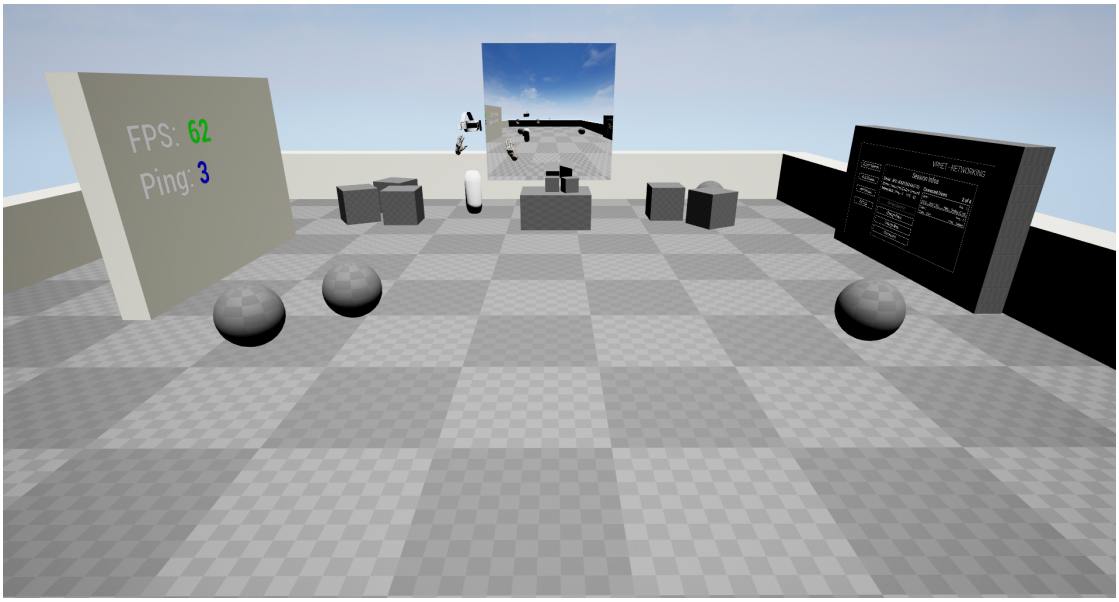


Figure 5.14: Screenshot of the main VRNet project template level.

A default Level is provided to showcase the functionalities and objects of the VRNet plugin. Figure 5.14 shows this Level with a user. The small map includes a mirror for players to see themselves and their avatars. On the right side of the figure the VRNet networking UI is placed, and on the left side a wall with FPS and Ping statistics are shown for debugging. The scene also includes a NavMesh setup placed on the floor to allow teleportation. There are different objects placed throughout the Level that can be grabbed and moved. These objects are based on the *VRNetGrabbableStaticMeshActor* class, with different static meshes. Figure 5.15 shows a user grabbing one of such object.

There are multiple different Pawns included that the users can change to.

- *BP_DefaultVRNetPawn:* Is a VRNetPawn with simple visual representations and the *BP_VRTemplateCtrlAug*. This Pawn is the default Pawn that is being spawned for each player if nothing else is selected. It can be seen in Figure 5.15 grabbing a cube.

- *BP_VRNetBasePawn:* Is the simplest Pawn, deriving from VRNetBasePawn, which only adds visual representation for controllers and HMD, without any interaction possibilities like grabbing or teleporting.

- *BP_VRNetIKPawn:* Is a VRNetPawn that adds a VRNetAvatar with a skeletal mesh and basic inverse kinematics (IK) for the animation of the arms related to the controller movement and animation of the head related to the HMD movement. It also includes BP_VRTemplateCtrlAugs for the controllers.

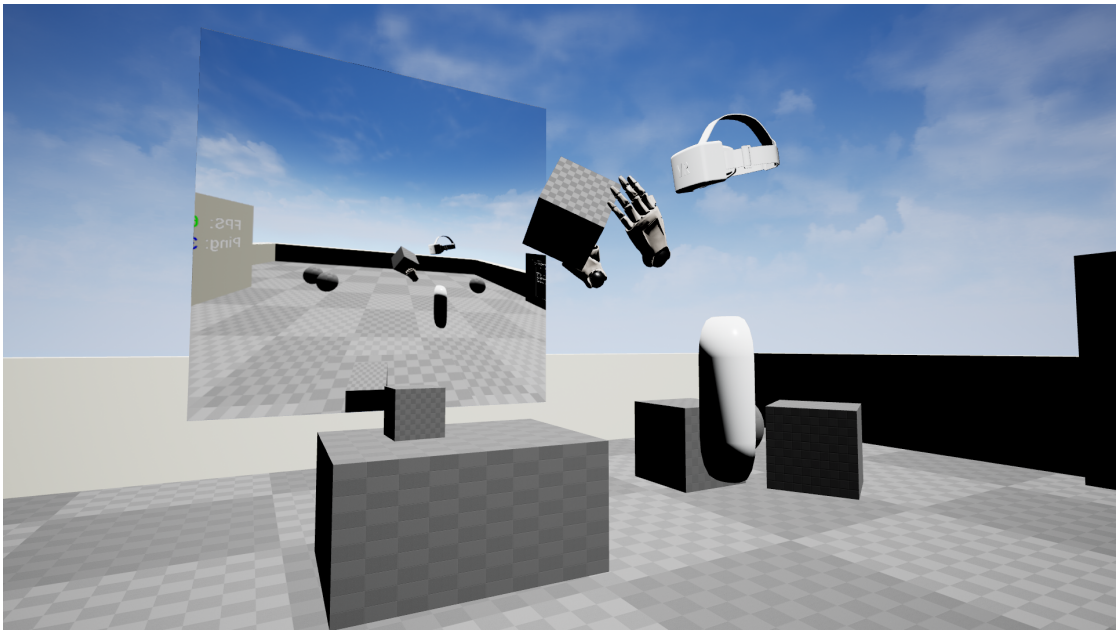


Figure 5.15: Screenshot of a user grabbing an object in the VRNet project template.

- *BP_MotionCapturePawn:* Is a VRNetPawn including the previously described *VRNetMotionCaptureAvatar*. It also includes BP_VRTemplateCtrlAugs for the controllers.
- *BP_DesktopPawn:* Is a non-VR Pawn that is controlled with keyboard and mouse used to fly around for movement. It integrates the grabbing and UI interaction Components to allow for using those functionalities without VR.

CHAPTER 6

CeMM Holodeck

In this chapter we present the integration of the VRNet plugin in an ongoing real-world multi-user VR application called *CeMM Holodeck* to make it multi-user capable.

The context and requirements of this project are presented in Section 6.1. The initial prototype, which was developed prior to this thesis without networking in mind, is analyzed and described in Section 6.2. The implementation steps to achieve the defined requirements are described in Section 6.3.

6.1 Introduction and Context

CeMM Holodeck is a data visualization platform developed by the Jörg Menche Group [2], which is part of the Center for Molecular Medicine (CeMM), a research institute of the Austrian Academy of Sciences. A year-long cooperation between CeMM and the VR Group at TU Wien should amount to a functionally usable and fully networked multi-user VR prototype.

CeMM Holodeck is intended to provide researchers with a platform to collaboratively explore and interact with complex biological data networks in VR, as well as on desktop.

These biological data networks are called *Interactomes*. They represent a set of physical interactions among molecules in the human cell, e.g. protein-protein interactions. Interactomes can be used to discover and classify diseases, as well as help in the development of treatments to prevent them [28].

The disease-associated proteins tend to cluster in the same area of an Interactome. The accurate identification of this area is a first step in understanding the molecular mechanisms happening with a complex disease and done by a network-based framework introduced in 2015 [47]. Figure 6.1 shows an example of how network-based relationships between disease genes look inside a interactome.

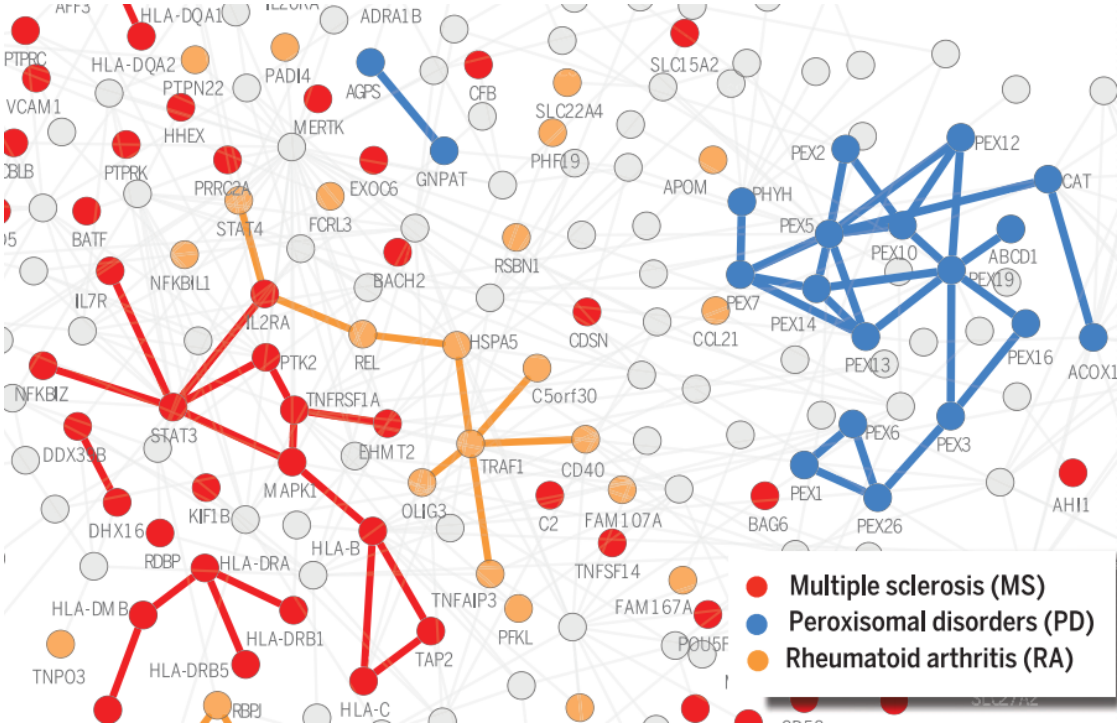


Figure 6.1: A subnetwork of the interactome showing network-based relationship between disease genes [47].

The VR platform should be used in the future to visualize different interactomes and diseases within them to investigate possible relationships of diseases. Since the human mind is excellent for pattern recognition a visualization tool like this might asssist the improvements of the research in this area.

The target audience of the platform primarily consists of biologist researchers. Apart from the main use case of research for biologists, the platform is also intended to be used to show and educate people about the research.

CeMM Holodeck is developed with UE4. Prior to this master thesis, the initial prototype had been in development for around half a year. The prototype did not have networking taken into consideration yet, but already provided the rendering of a biological data network and basic interactions with it. This initial prototype will be described in detail in Section 6.2

In the scope of this thesis a fully networked prototype with priorly defined functional requirements for the networking was developed with the help of the VRNet plugin.

Target Users of the Project There are several types of target users for this application with differing levels of biological background understanding and varying VR

proficiency. Users in VR most importantly want to use the application without getting uncomfortable, so the experience has to be smooth for them.

Biologists are the main type of target users and are supposed to use the platform for research and presentation. This group of users have a background understanding of the biological data networks and mainly expect to use the application for collaboratively exploring, interacting with and demoing the data networks.

Other users unfamiliar with Biology but interested in the research are also an important target user group. This group of users expect to use an application mainly for viewing purposes and understand the content with the guidance of experienced users or researchers.

6.1.1 Implementation Goal

The goal of this part of the thesis is to make CeMM Holodeck project fully networked. Furthermore, the VRNet plugin should be used to achieve this goal.

The nature of the project is very experimental, with functionalities that are added, discarded and changed constantly. To reach the goal it is necessary to define and agree on a set of functional requirements for the resulting prototype in cooperation with the CeMM development team, which was done and improved iteratively during implementation.

The non-functional requirements for the implementation are very similar to the ones identified for the VRNet plugin in Section 4.1.3. Performance efficiency and maintainability are the most important NFRs for the prototype and its networked functionality. It is crucial to provide a smooth and performant experience for the users of the platform in VR. Furthermore, the implementation of the functional requirements should be as maintainable and extendable as possible, especially since the development of the project is ongoing, with many additional features being planned in the future.

6.1.1.1 Functional Requirements

The functional requirements for the implementation that have to be networked are listed in this subsection.

- Movement and representation of users should be replicated.
- Session management functionality with possibility to connect in different ways, also over the Internet should be provided.
- It should be possible to select and change Pawns.
- Several different Pawn implementations should be provided, e.g. desktop Pawns, spectator Pawns, VR Pawns, a static camera Pawn following a Tracker, etc.
- Data-network interactions should be replicated, which include:
  - Moving, scaling and rotating the data network.

  - Node selection handling visible to all users. This includes adding and removing selected nodes, either directly through user interactions or through other interaction actions that involve displaying certain nodes from the database (e.g. loading gene nodes related to diseases and adding them to the selection).
  - Node manipulations like hiding, showing, isolating node selections or hiding and showing all the nodes.
  - Node information highlighting functionality, where users can highlight nodes and their linked neighbors and show information displays with the node names.
  - Switching functionality that allows users to change between node selection and node information highlighting.
  - Displaying a browser window with detailed informations when users request more information about a highlighted node.
  - Loading the node layouts of data-networks from files into two different "channels A/B" that are held in memory, only one being visible at a time. Additionally loading of link layouts.
  - Blend functionality that makes the nodes transition from the data-network layout in channel A to the one in channel B.
  - Loading and saving node selections to files.
  - Molecular Cluster selection that requests nodes from the CeMM database and adds it to the node selection and isolates it.

- The UIs for the world and data-network interactions should be replicated:
  - All users should see the changes made to the UIs, including UI element states, like sliders and buttons.
  - It should be possible to move and attach UIs in a designated area and to hide, remove or minimize parts of them.

6.2 Initial Single-User Prototype

The initial prototype of CeMM Holodeck was developed during the cooperation between CeMM and the VR Group at TU Wien before the start of this thesis. In its initial state the prototype was only usable by a single person at a time and networking was not taken into account during its development process.

The project at first was also in an early experimental state with implemented approaches to solve the visualization problem as well as basic interaction methods.

This section presents *two different versions of initial prototypes*. One was an early version before the implementation of the VRNet plugin was started, where we experimentally implemented parts of the replication to evaluate their feasibility. The second initial prototype was developed shortly after we finished the networking of the first one, where

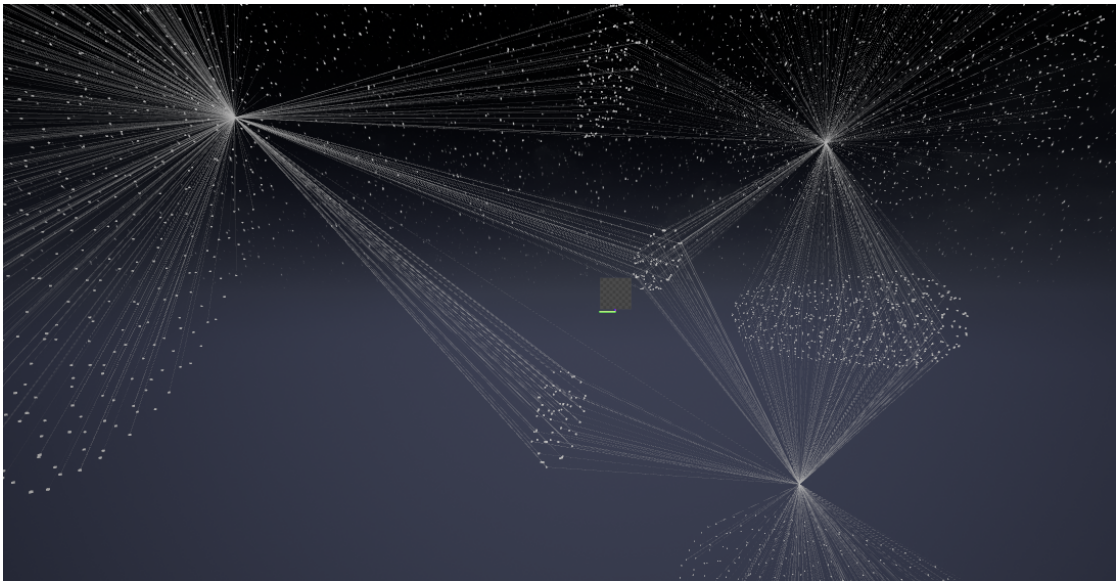


Figure 6.2: Screenshot of the first initial CeMM Holodeck prototype.

an additional range of crucial features was introduced to the prototype, including a room, many user interfaces with data-network interaction possibilities and node selection functionalities.

This second prototype is the main focus of the implementation state presentation in this section, but the functionalities of the first one are also briefly described.

First Version In the first version of the initial prototype the data-network was loaded from files on the disk, containing the node and link data in comma-separated values (CSV) file format. This loaded data was rendered with small cubes for the nodes and lines between them for the links. The screenshot in Figure 6.2 shows how the world and data-network layout visualization looked like in the initial prototype.

A user could seemingly manipulate the location, rotation and scale of the data-network indirectly via moving their Pawn, i.e. translating, orbiting around the data-network and moving closer and farther for a zooming effect respectively.

With a laser-pointer-like ray coming out of one of the controllers it was possible to select and highlight a node, where then an info annotation with the node's name was spawned.

Second Version While the first prototype only consisted of the data-network and an empty world surrounded with a night sky, the second one introduced a room with UIs and the data-network placed inside it. Figure 6.3 shows a screenshot of this Level.

The user interfaces introduced a way for the user to interact in VR with the data-network, e.g. to change the scale, the thickness of the nodes and links or loading different data-

Figure 6.3: Screenshot of the second initial CeMM Holodeck prototype.

network layouts. Some of those UI parts were displayed, but not yet functional, for example the node selection file loading and saving. User interaction with the UIs worked with a laser-pointer type of selection coming out of the left controller, whereas the right controller's red ray was used for node selection.

Adding and removing nodes from the current selection set was done with the controllers' left and right face buttons when pointing at a certain node. The selection then could be isolated from the rest of the data-network, meaning that all nodes except the selection were hidden. Additionally it was possible to hide and show all nodes, as well as hide and show the current node selection. The color of the selected nodes could be changed through the *color picker* UI.

6.2.1 Implementation Details

The amount of data-network nodes and links between them by far exceeded the number that could be represented inside UE4 by game objects. The approach to render the data-network focused on vertex displacement and was handled mainly in the material shader. In this case this meant that the node position and link data was stored in textures and displaced by the implemented shader to render them where they should be. Shared global material parameters like the network scale were used to control the displacement and scale the data-network. To interact with certain nodes the implementation had to take these shared parameters and the interaction position into account to trace and calculate which node was being selected.

With this rendering technique the handling of the translation and rotation of the data-network inside the shader would have been difficult and complex. Because of this the main approach for the data-network movement was to keep the position unchanged at the world origin and instead move the player around it to give the illusion that the player is moving the data-network. This allowed that complicated rotation calculations were not needed to be done inside the shader, but simply the input for the tracing interaction with the data-network nodes changed to calculate from a different player position and rotation.

6.2.1.1 Important Classes

The initial prototype was partly implemented in C++ and partly in Blueprints. The handling of the visualization of the data-network, its state, file loading and http request for getting node information were implemented in C++. The remaining functionality, including player and interaction functionality was written in Blueprints. The material shader implementations were written in a special type of Blueprint called *Material Blueprint*.

The world outline in the UE4 editor of the Level and its Actors is shown in Figure 6.4. The Actors with white text were placed statically inside the Level, while the ones with yellow text were spawned on startup.

Data-Network core classes The main logic directly related to the data-network was handled in the following classes.

- **MyStaticMeshNodeActor:** The core C++ class for the data-network visualization was mainly responsible for the data-network state, manipulating and querying it to interact from the outside. It also implemented the logic for the file loading and parsing of the CSV data.

- **BP_texture_interface:** This was the main Blueprint Actor for interacting with the MyStaticMeshNodeActor, responsible for most of the line tracing logic, for data-network loading and node selections. It was strongly coupled with the main *VivePawn* implementation and intended to be an interface for access to the data-network, but in many classes also the MyStaticMeshNodeActor was referenced directly instead.

- **M_DynamicTextureNodes & M_DynamicTextureWires:** These UE4 Material files contained the shading code, which takes the textures held by the MyStaticMeshNodeActor and renders the displaced nodes and wires. Global "blend" and "scale" parameters were used to manipulate the calculated positions.

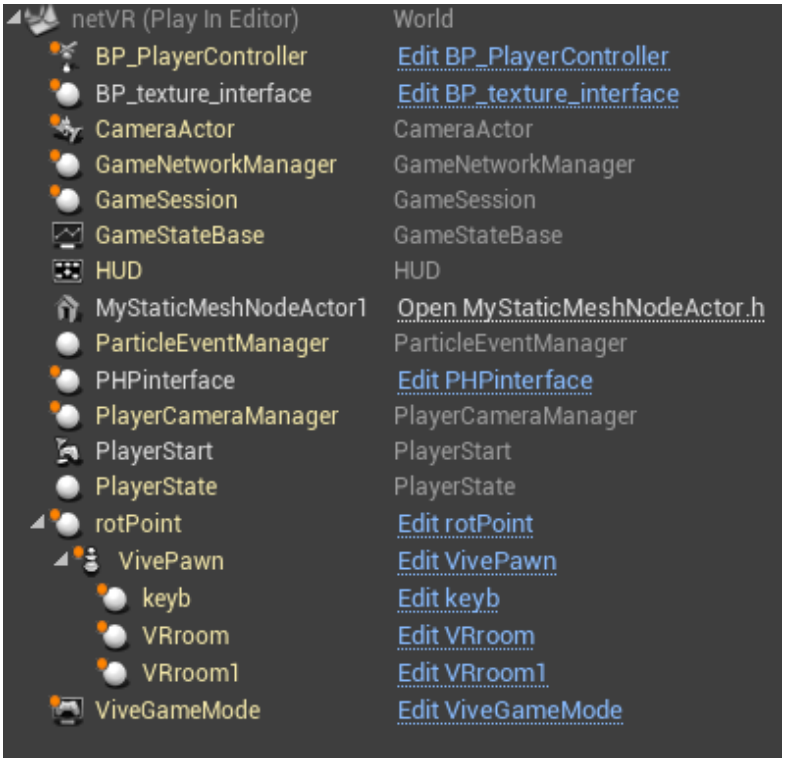


Figure 6.4: Screenshot of the world outline of the second version of the Holodeck initial prototype.

RotPoint: The RotPoint Actor was the central point where the player, the room and UIs were attached. The main function was to act as the center for orbiting rotation calculation, as well as moving and scaling. These movements applied to the RotPoint translated down its attachment hierarchy and were also applied to attached Actors.

VivePawn: The Pawn implementation included all the input handling, data-network navigation and UI interactions, as well as all visual Components related to the player and for spawning data-network informations. At first the room and the UI Actors were all part of this class and attached to its RootComponent. This was due to the navigation logic of the data-networking only moving the *RotPoint*.

UI Widgets: There were several UI Widgets providing the interfaces for interaction with the data-network that were previously explained. They were mostly interacting with either the BP_texture_interface, the MyStaticNeshNodeActor or the VivePawn.

6.2.2 Problems and Challenges for Networking

While the described implementation was in an already fairly functional and advanced state, it was implemented in a way that it could only work for one user at a time. Hence for the implementation of the networking functionality we faced the following challenges.

- **Data-network movement:** The used approach for navigating the data-network and how the room attachment was handled initially neither worked conceptionally nor practically for multiple users. Most of the Actors were placed either globally into the Level or attached directly to the Pawn. Additionally references to Actors were usually queried with an UE4 provided function that returns an array of Actors inside the Level and then simply taken the first entry, which does not work if there are multiple relevant instances. To accommodate multiple users it was necessary to change much of the spawning and attachment logic, as well as the structure of the Level scene.

- **Data-network state replication:** Similarly, the data-network's state with its multiple textures meant that it could not be simply replicated through UE4's property replication system. To implement proper distribution of this data-network state a different networking approach needed to be developed. The constraints for such an approach included that the underlying UE4 engine and some of its systems could not be assured to be deterministic on different machines. For example, while sending only user inputs and movement data to update the state on each machine without its actual replication would be easy to implement it could lead to inconsistency problems making such an approach unfeasible.

- **Challenges with the code base:** The initial prototype was in an experimental state that resulted from rapid prototyping and it was still heavily under development. Most of the implementation logic was in Blueprints, which, while being an excellent tool for rapid prototyping, can present challenges for understanding and clearly structuring classes. We realized that we would have to be extra careful and employ tactics to assures that newly introduced Blueprint code would be sufficiently clean and particularly the networked parts would be easier to discern. The screenshot in Figure 6.5 shows an example of how some of the Blueprint code was looking before the refactoring process, to give an idea of the complexity of the initial prototype.

To summarize, the networking aspect not being taken into account from the beginning, the difficulties of dealing with the data-network state and replicating it, as well as the prototype nature of the code base presented a challenging starting point for the networking integration. The crucial first steps of the networking implementation would be the analysis and reworking of the initial functionality while coming up with a viable networking approach. These steps are described in detail in the following sections.

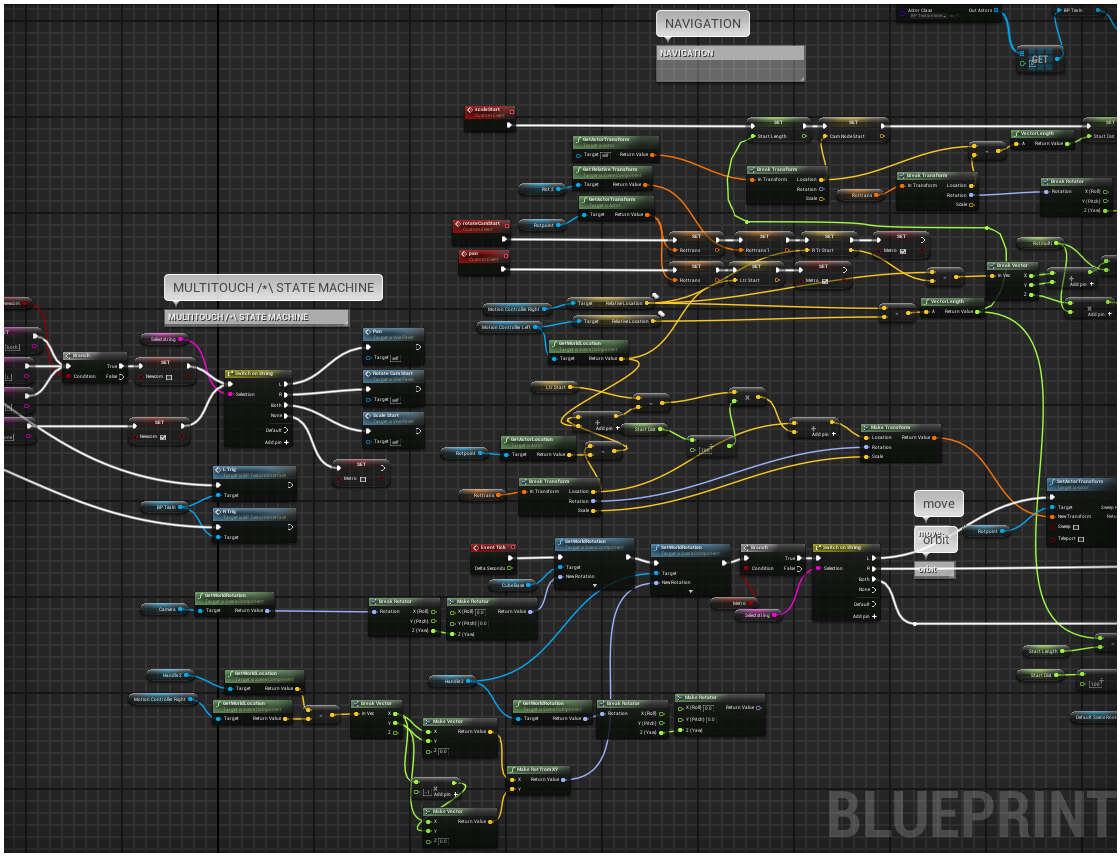


Figure 6.5: Overview of an example Blueprint class in the initial prototype.

6.3 Implementation

6.3.1 Refactorings

6.3.1.1 VivePawn

Starting with the Pawn implementation, called *VivePawn*, first cleanups and reworking of its functionalities needed to be done.

In the VivePawn most of the player interaction logic was contained, the navigation logic and input handling of the player to move and orbit around the data-network, the widget interactions with the user interfaces, as well as collision events. The principal actions we took to refactor the code were the following:

- **Restructuring of the code base:** This step included renaming some variables and functions and removing the unused parts of the code. Additionally, Blueprint comment boxes were introduced to contain the Blueprint function nodes, describe them and visibly mark them. This can be seen in Figure 6.6, which shows the

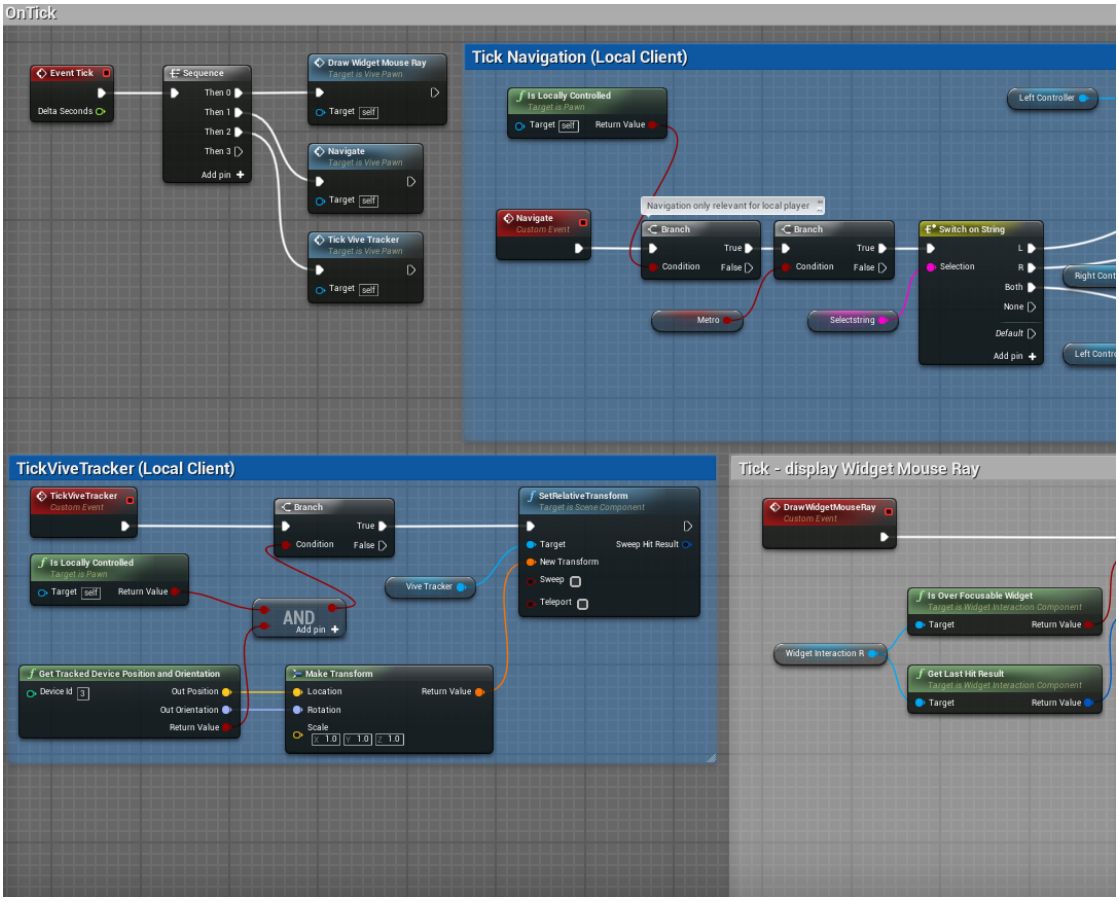


Figure 6.6: *OnTick* segment of the refactored VivePawn Blueprint.

OnTick segment of the VivePawn where nested comment boxes were introduced to structure the code. The meaning of the different colors will be explained later.

- **Extracting code from VivePawn into specialized Components:** Furthermore, parts of the code were identified that should be extracted from the VivePawn class and put into contained and independent ActorComponents. This approach to extract functionalities and encapsulate them in ActorComponents rather than in the Actors themselves was similar to the design philosophy pursued in the VRNet plugin. The result of these steps was that most of the navigation logic inside the VivePawn got extracted to a *NavigationComponent*, and the UI interactions to a *UIInteractionComponent*. The VivePawn defined and managed the player input for the navigation and called exposed navigation functions of the NavigationComponent, i.e. Pan, Orbit or Zoom. The specific navigation implementation was contained in the NavigationComponent, making it independent from the VivePawn and exchangeable. Similarly the UIInteractionComponent was intended to act as the interface between player and the UIs and to handle the logic and replication

of the user initiated UI actions. Most of the VivePawn's contained UI interaction logic got moved to this Component.

Creating these generalized Components had the big advantage to control which Pawns have capabilities for navigating or UI interaction and to make it easy to add the capabilities to new Pawns. Another benefit of this approach was that splitting the code into several files leads to less merge conflicts during the development process, since it is easier to coordinate and focus on developing inside fewer Blueprint files. This was important because Blueprints can be hard to merge, the problem that we faced multiple times during the development process.

6.3.1.2 Data-Network Interaction Actors

Data-network interaction logic that was previously handled in the *BP_Texture_Interface* Actor, which was placed statically into the Level got converted into an ActorComponent as well. This Component was named *TextureInteractionComponent* and can be added to any player Pawn to enhance it with data-network interaction capabilities, e.g. VivePawn. This would change the concept of the class to being a player Component instead of being a single instance holding global state. This approach was chosen because this Actor held a lot of state and logic that is relevant to only one interacting user, and the state that is meant to be global should be handled inside the *GameState* Actor.

In the final implementation, this class is the main interface to the *MyStaticMeshNodeActor*, which was completely unchanged in the refactoring process and continued to be a single instance without directly replicated data-network state, i.e. the node and link textures.

This Component allows to easily add and remove data-network interaction capabilities to any Pawns. For example, VivePawn includes it, since it is the main Pawn with all the functionalities, but other Pawns like spectator Pawns do not include it, since their users are not supposed to interact with the data-network.

To implement such a generalized Component it was important to rework the logic contained in the *TextureInteractionComponent*.

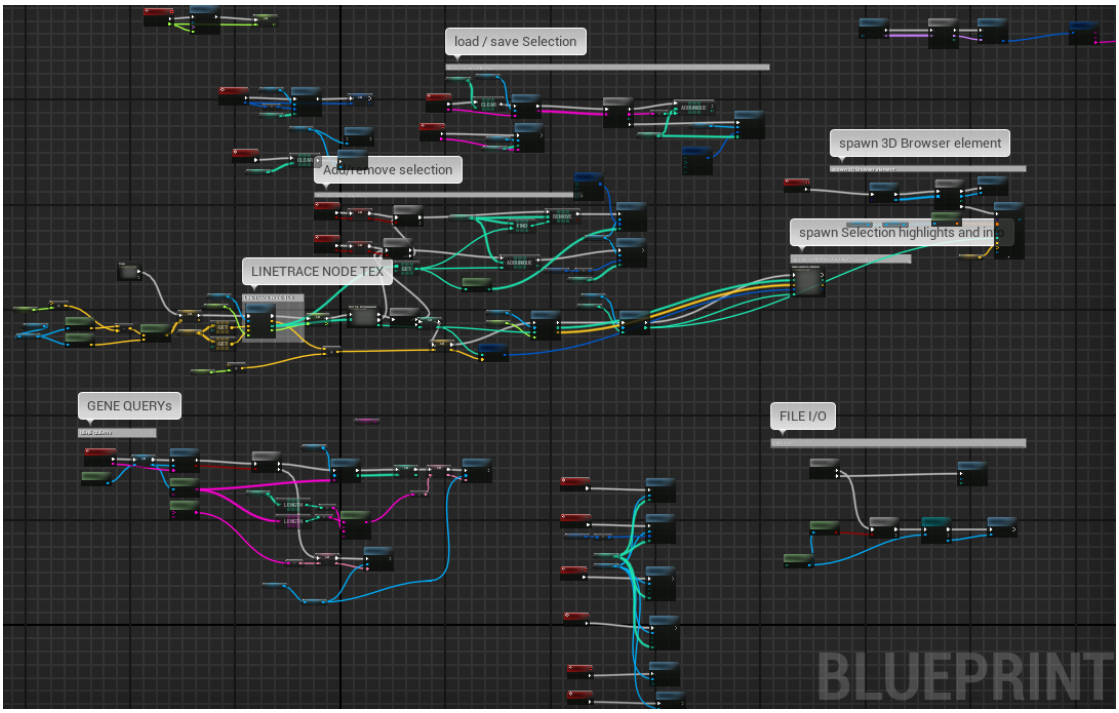
TextureInteractionComponent Refactorings The TextureInteractionComponent extraction and refactoring was one of the main undertakings in the refactoring process, with probably the biggest impact on the codebase of the project.

- **Restructuring Blueprint code:** This step included renaming, removing and restructuring variables and functions, and additionally the introduction of colorized comment boxes. Some long functions that contained a lot of code (e.g. the line tracing logic) was refactored into multiple functions. This was used to make the code more clear and to extract separate functionalities. For example the spawning of the Mesh Actors for the visual representation was detached from the line tracing logic.

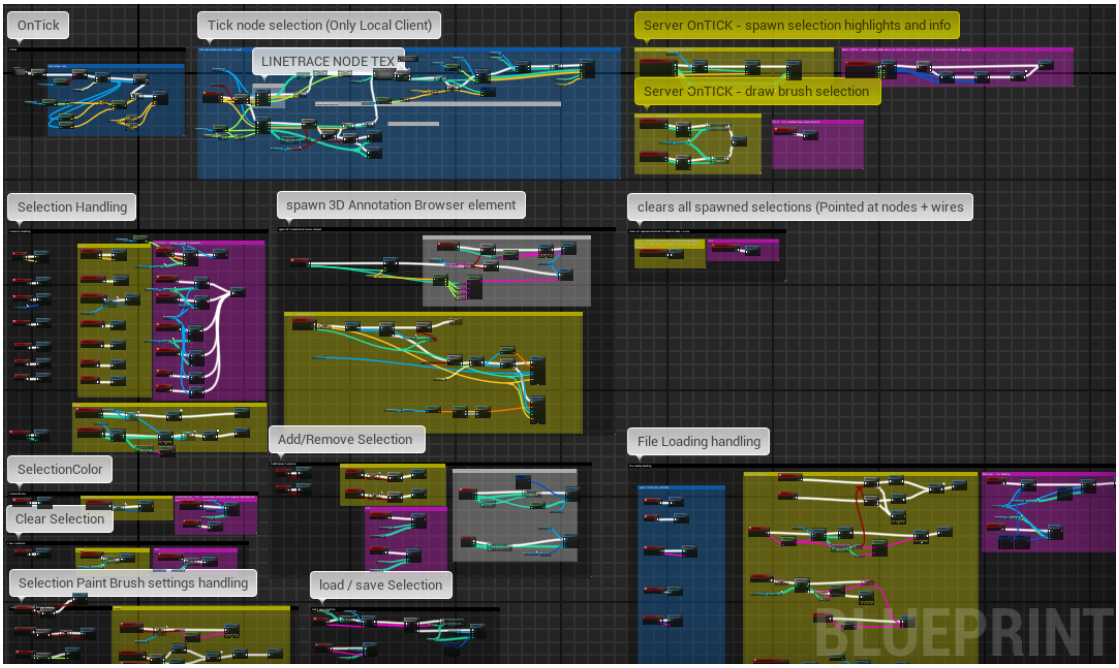
- **Removal of direct dependancies to VivePawn and its Components:** This step was made to make the TextureInteractionComponent a generalized and independently usable ActorComponent that could be used by various types of Pawns and was followed by the creation of generalized methods to interface with the outside world. Instead of the VivePawn reference an *OwnerPawn* Pawn reference was used in combination with an *ITextureInteractablePawn* interface defined to provide some necessary functions that a Pawn wanting to use the TextureInteractionComponent needs to implement. Specifically, the interface included functions that returned the necessary reference SceneComponents of the Pawns, i.e.:
  - *GetLineTracingComponent* - which returned the SceneComponent used as a starting point for the line tracing algorithm E.g. a tracked MotionController-component.
  - *GetINFOAnnotationHandle* - which returned the SceneComponent that determined the position of the information displays that spawn when highlighting nodes
  - *GetSelectionBall* - which returned the SceneComponent holding the ball that was used for selecting several nodes at a time.
  - *GetBrowserHandle* - which returned the SceneComponent that got used to attach the spawned browser Widgets.
- **Moving the code describing the state of the data-network to GameState:** This step was made to allow for the easy access and replication of the data-network state. This state includes the scale value of the data-network which is passed to the shader to scale the rendering of the data-network and is also used to calculate the node position for the line tracing.

  The other state inside this Component was meant to be only valid for one instance of the Component, i.e. for one player Pawn containing it. This state included the currently highlighted node, the selected nodes, the color of these selected nodes, etc. Also the spawned Actors for the currently highlighted node, its node neighbors and the wires between them was changed to be managed exclusively inside the Component. While the replication of this state was not the focus in this section's description yet it was important to structure it accordingly during refactoring to prepare it for the replication implementations.

The result of the described changes made during the refactoring process can be examined in the comparison of the Blueprints of the initial prototype and the final prototype code of the TextureInteractionComponent logic shown in Figure 6.7. The Subfigure 6.7b for the final prototype also includes the code necessary for making the project fully replicated, which are explained in detail in the Section 6.3.3. While far from being perfect we think that the usage of the colored comment boxes and the other explained refactoring steps makes the final prototype's code clearer and easier to maintain.



(a) Initial prototype Blueprint code overview



(b) Final prototype Blueprint code overview

Figure 6.7: Blueprint overview comparison of the TextureInteractionComponent between the initial Holodeck prototype and the final result.

6.3.1.3 UI widgets

- **Interaction through UIInteractionComponent:** The UI widget blueprints were refactored to not directly access the VivePawn, the TextureInteractionActor, or the MyStaticMeshNodeActor, but instead access the UIInteractionComponent of their locally controlled Pawn. This step was taken to make the UIInteractionComponent the only Component interacting with user interfaces to ease the management and replication of the UI interactions.

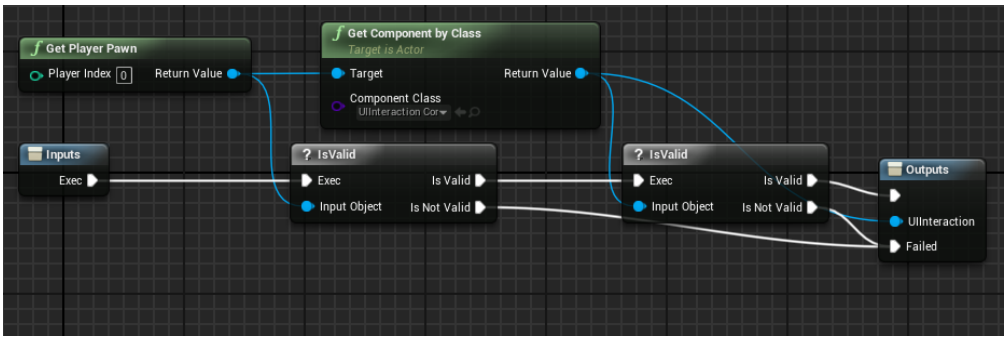
- **Improvements in reference handling:** Getting the references inside the UI widgets was problematic initially. A common used pattern in the initial prototype was to have globally existing instances of the Actors placed in the world and getting the reference to them inside the *BeginPlay* events to store and access them. The references were retrieved there via the UE4 Blueprint function *GetAllActorsOfClass* which returns an array of all the found Actors and then the first entry chosen without further checks.

  This was especially problematic for getting the reference to the VivePawn and the BP_texture_interface, since both naturally can not be singleton objects but rather have to be spawned dynamically for each player. UE4 stores internally references to all current players with their relevant Actors, i.e. PlayerController and Pawn. It is possible to access the local player on each machine via the provided *Get Player Controller* functions and the player index of 0, which the engine ensures to be the local player on each machine. To refactor the reference access this function was used, and the returned class was either casted or containing Components queried to get the specific references.

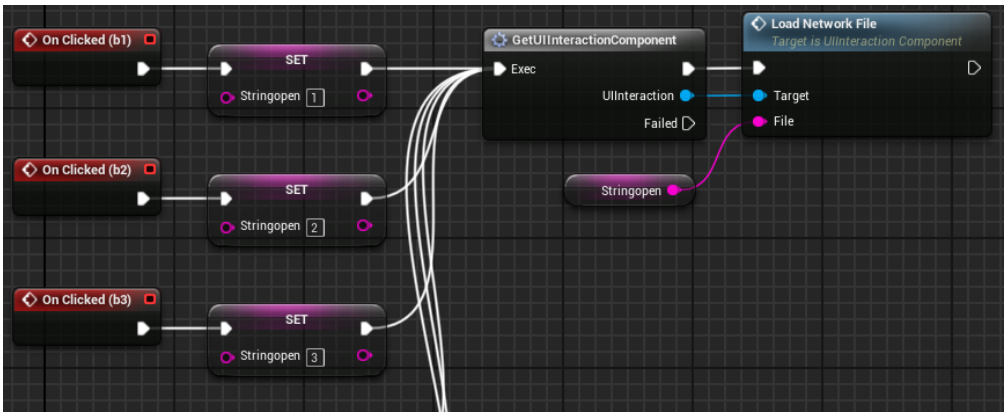
- **Definition of Blueprint Macros:** To further simplify this access Blueprint Macros were defined to provide reusable functions for getting the local player Pawn, and if valid, get their UIInteractionComponent, TextureInteractionComponent, etc. Blueprint Macros are simply reusable code snippets that get replaced with the actual code on compilation. Getting the references in the widget blueprints were refactored to use this kind of macros instead of the previous methods. Figure 6.8a shows the implementation of the *GetUIInteractionComponent* Blueprint Macro, which queries the local player Pawn and gets a single UIInteractionComponent if it is contained in the Pawn. If there exists no UIInteractionComponent the *Failed* execution line is called, which means that the direct execution output line will not be executed. An example usage of the Macro is shown in Figure 6.8b. In this example the *LoadNetworkFile* function is only called if the local Pawn exists and if it contains a UIInteractionComponent.

6.3.2 VRNet Plugin Integration

Preliminary integration of classes The first integration of some of the relevant VRNet plugin classes was done through copy and pasting the classes directly into the



(a) Implementation of the Blueprint Macro



(b) Usage of the Blueprint Macro

Figure 6.8: Blueprint Macro GetUIInteractionComponent.

project instead of integrating the whole plugin. This was done in the first version of the inital Holodeck prototype, even before the refactoring phase. The goal of this direct integration was to evaluate the VRNet classes in the Holodeck context and to create a working proof-of-concept version with player replication. Additionally we hoped to integrate a VRNet Pawn into the relatively sizeable VivePawn as soon as possible to avoid major divergences later between the structure of the Pawns.

Integration of the plugin After the preliminary integration and the refactoring phase of the initial prototype the VRNet classes were provided, as it was initially intended, with the integration of the VRNet plugin into the project. With the actual plugin being integrated into the project it was significantly easier to maintain the updates of the classes with new versions during development.

Used VRNet classes

- **VRNetBasePawn** was used as the base for the VivePawn and other VR Pawns.

Figure 6.9: Screenshot of the VRNet networking UI integrated into Holodeck.

With the VRNetBasePawn the functionalities and VRNet Components for the player movement replication of the controllers and HMD were included. The VRNetBasePawn rather than VRNetPawn class was chosen because its minimal functionalities were ideal for the Holodeck project and the VRNetPawn with its introduced Components were not implemented yet during the initial integration of the VRNet plugin.

- Additionally, the **VRNetReplicationComponent** was used for the replication of the RotPoint Actor's movement. Since the navigation logic solely manipulates the RotPoint's RootComponent position and rotation to move the attached room and players around the data-network replicating the movement data via the VRNetReplicationComponent was a working approach. This approach will be described in further detail in Section 6.3.3.1.

- The classes handling the game management functionalities were extensively used to manage the connection process, access to the OSS and pawn changing functionality. The **VRNetGameModeBase**, **VRNetGameStateBase**, **VRNetPlayerController** and **VRNetPlayerState** were used as base classes for the respective existing Actors. Additionally the Game Instance class was set to be **VRNetGameInstance**.

- To access the game management functionalities the **VRNet networking UMG** example implementation was integrated and changed to appear in the Holodeck UI

style. Interacting with this UI was handled in the same way as the interaction of the other Holodeck UIs. Figure 6.9 shows the VRNet networking UI inside Holodeck and a user interacting with it in VR.

Other VRNet plugin functionalities like the ones introduced with VRNetPawn, including the VRNetControllerAugmentation and VRNetAvatar classes were not used in the final prototype of the Holodeck project. For example, the grabbing interaction simply was not needed for this type of project. The teleportation on the other hand would be theoretically useful for the application, but was also not included at this time. While out of scope for this thesis, we think it would make sense in the future to further use the VRNet's capabilities and invest time to refactor the Holodeck Pawns to make use of the swappable VRNetControllerAugmentations and VRNetAvatars, since they would be applicable for at least some of the Holodeck's use cases.

6.3.3 Networking Implementations

This section describes our iterative implementation process to find working replication approaches for the different parts of the prototype. Some of these approaches had to be changed multiple times during the development, due to the Level setup and core functionalities changing significantly.

Structuring of the networked Blueprint code In general, a chosen tactic for making the Blueprint code more understandable, especially for the replicated parts, is to color the comment boxes containing replicated code to certain colors. The used color designates RPC code and on which machines it will be executed. The comment colors are:

- *Blue:* for only locally executed code (e.g. player input) or client RPCs.
- *Yellow:* for server RPCs and server-only executed code.
- *Purple:* for NetMulticast RPCs, i.e. code that is executed on each machine

An example of the used colored comment boxes can be seen in the previously introduced Figure 6.7b showing the overview of the TextureInteractionComponent of the final prototype.

6.3.3.1 Replication of the Data-Network Movement

The replication of the data-network movement logic is challenging, because of the constraint that the data-network is placed statically in the world and not movable directly. To give the illusion of moving it the environment including the players has to be moved around it. Figure 6.10 shows a screenshot of a user moving the data-network inside

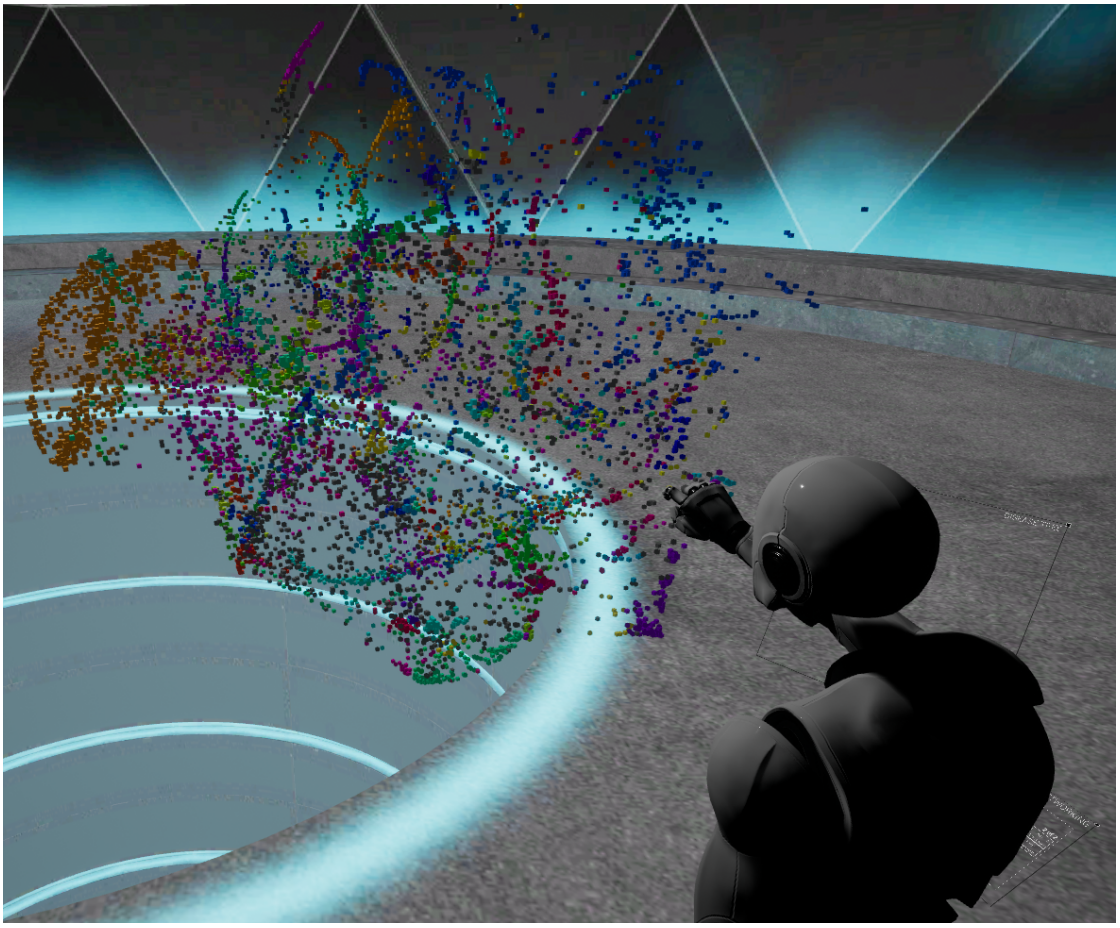


Figure 6.10: Screenshot of a VivePawn inside the Holodeck moving the data-network.

the room with its right controller, pressing the trigger button and dragging the data-network around. In their perception when initiating the movement only the data-network is moving, the world around them is static and still.

To achieve the proper replication of these movements with multiple users the following steps are described:

- **Replication:** To move around the data-network the attachment logic is changed to accommodate multiple users. Additionally, each of them potentially can move the data-network.

  Figure 6.11 shows the concrete attachment hierarchy. The platform (i.e. the enclosing room) is attached to one RotPoint. Furthermore, to the platform all spawned players, as well as other scene objects like UIs and lights are attached to the platform. Every rotation and translation of the RotPoint moves the whole scene around.

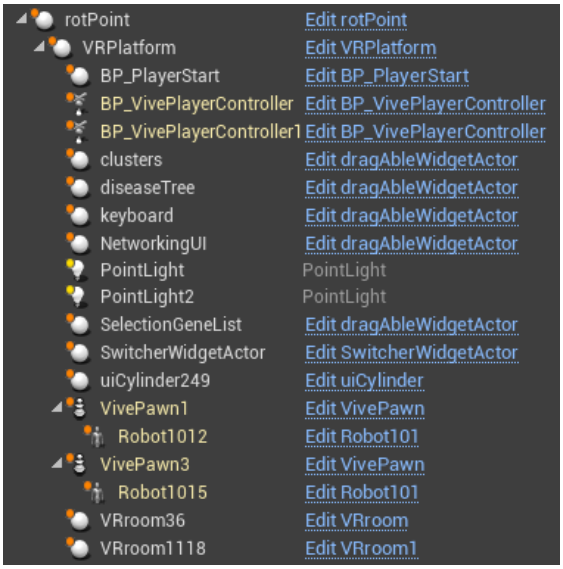


Figure 6.11: Screenshot of the RotPoint attachment hierarchy of the spawned Actors in the world.

This RotPoint, as described earlier in the Refactorings section, is manipulated through the navigation logic that resides in the NavigationComponent, which is contained in the user's Pawn. To replicate the relevant RotPoint the *VRNetReplicationComponent* of the VRNet plugin is used. The RotPoint contains multiple SceneComponents that can be manipulated. Each of them is replicated with a VRNetReplicationComponent, which is set to auto-activate to replicate for all its lifetime.

- **Issues due to the replication:** While this approach successfully replicates the movement there arose issues with noticeable jitterings for the users. Especially for the users not currently moving the data-network there appeared significant jerkings of their controllers and HMD representations.

  This issue happens due to a conflict between the player movement replication and the RotPoint movement updates that get passed through the attachment hierarchy. Because the position updates are sent and updated in world space coordinates there is a race condition happening between replicated RotPoint updates and the player's replicated position updates. This is likely due to the position updates overwriting such updates initially, but then getting corrected. Therefore it is important to use the VRNetReplicationComponent's relative space setting for users' movement replication. Then the problem does not exist, because only the relative position to the Pawn RootComponent gets transmitted. This relative position is not changing when the Pawn gets moved from outside the Actor, i.e. through the attachment to the RotPoint.

- **Performance optimization measures:** Another, though smaller drawback of this data-network movement replication approach is that the rotation of the users and their perceivable world around the data-network makes it more challenging to provide a smooth replicated movement. For the locally controlling client the movement is perfectly smooth, due to the client-authoritative replication. Other clients though could notice small jitterings due to network performance. Especially because the data-network movement is one of the most critical parts of the application to provide a smooth experience for users, due to slight delays and jitterings were effecting the world and other players and easily noticed.

  Our tactic to resolve these was to use the replication optimization utilities that are provided by UE4 and the VRNet plugin.

  The RotPoint Actor's *NetUpdateFrequency* is set to a value of 90 (times per second the Actor and its properties will get considered for replication). This roughly matches the targeted VR frame rate. Such a high value ensures that the movement updates of the room are fine-grained and smooth, but comes with the cost of higher bandwidth while moving. The VRNet plugin allows to control the rate of sending the server RPCs with the position updates in the VRNetReplicationComponent as well, which is also set to 90 to match the *NetUpdateFrequency* of the property replication.

  High update rates can still lead to jerking movements if there are unexpected network delays and packet loss. To circumvent such issues the client interpolation provided by the VRNetReplicationComponent is used.

  Additionally to the high update rates, which still could lead to jerking movements if there were unexpected network delays and packet loss, the client interpolation provided by the VRNetReplicationComponent is used. For the RotPoint movements the *LerpToLastUpdate* client interpolation mode is configured, with the *Lerp Time* matching the *NetUpdateFrequency*. While not perfect, this combined measurements leads to a decently smooth user experience with only some minor disturbances due to network lag. For the most part such disturbances get mitigated by the interpolation, but sometimes they were still noticeable when rotating or moving the room around.

  Future improvements are already planned beyond this thesis that will change the described setup and movement approach. The rotation of the data-network This would lead to the world and its contents to not be affected so strongly by minor jitterings, making it less noticeable and more forgivable.

6.3.3.2 Replication of the Data-Network State

This section describes our approaches to assure the consistent state of the data-network. The data-network state that is relevant for replication includes the node and link positions, which are held in textures. Furthermore, the state consists of the loaded data-network layouts and other state related to the data-network's content like its scale, node and

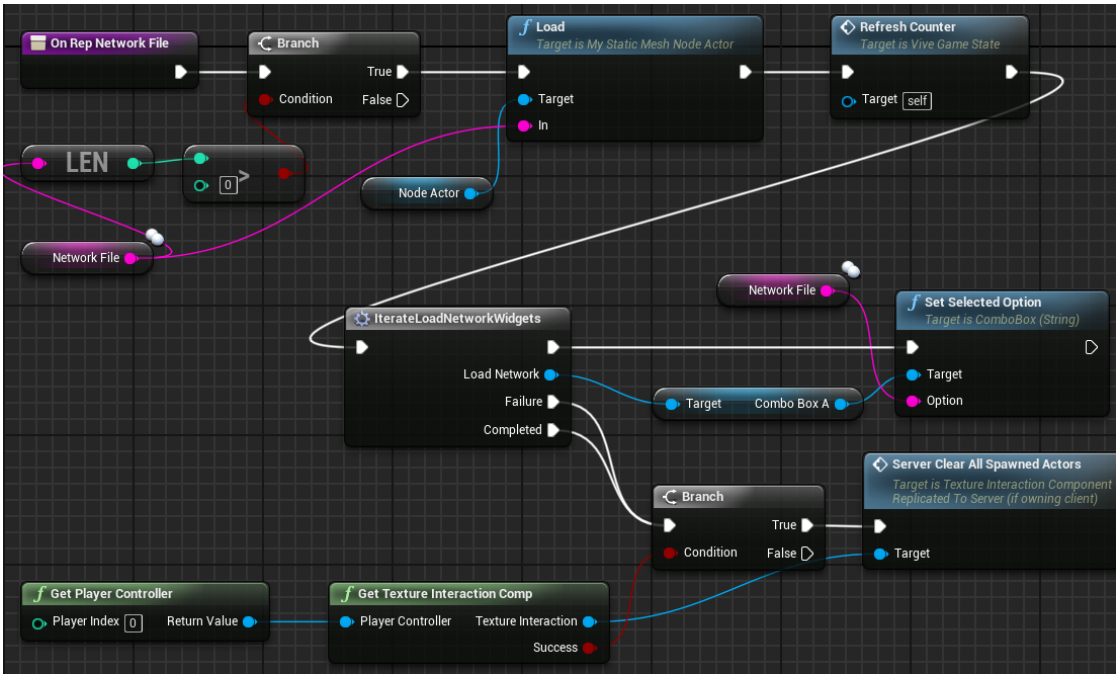


Figure 6.12: Blueprint of the *OnRep_NetworkFile* function that is called on replication of the *Network File*.

line thickness and so on. Interactions with the data-network also manipulate certain state that needs to be replicated, for example node selections. Property replication is not possible for certain parts of the state, e.g. the node/link position data that is encoded in textures. Textures' values are not possible to be replicated internally in UE4.

Data-network loading and creation On *BeginPlay* the files for the default data-network layout are loaded to display the data-network initially. Similarly, this process happens when the data-network layout is changed (e.g. by a user). The data-network layout files include two independent node layouts that are stored in channel A and B, as well as the link layout. The loaded layout files are parsed and then the data stored inside the textures held in the MyStaticMeshNodeActor, which further will be used by the Material shaders. This happens on each machine, which means that the same data-network layout files are provided for everyone at startup. The size of such files is typically several MBs, containing the relevant data encoded in text form. For our purposes it is sufficient to include these files in the packaged application instead of sending it over the network. In the future, however, the application could be further augmented with the funtionality of dynamically loading these files on the server machine and then distributing them to clients before starting an online session.

The *ViveGameState* Actor includes string variables denoting which data-network layouts are currently loaded. These strings contain the respective filenames of both channel A&B

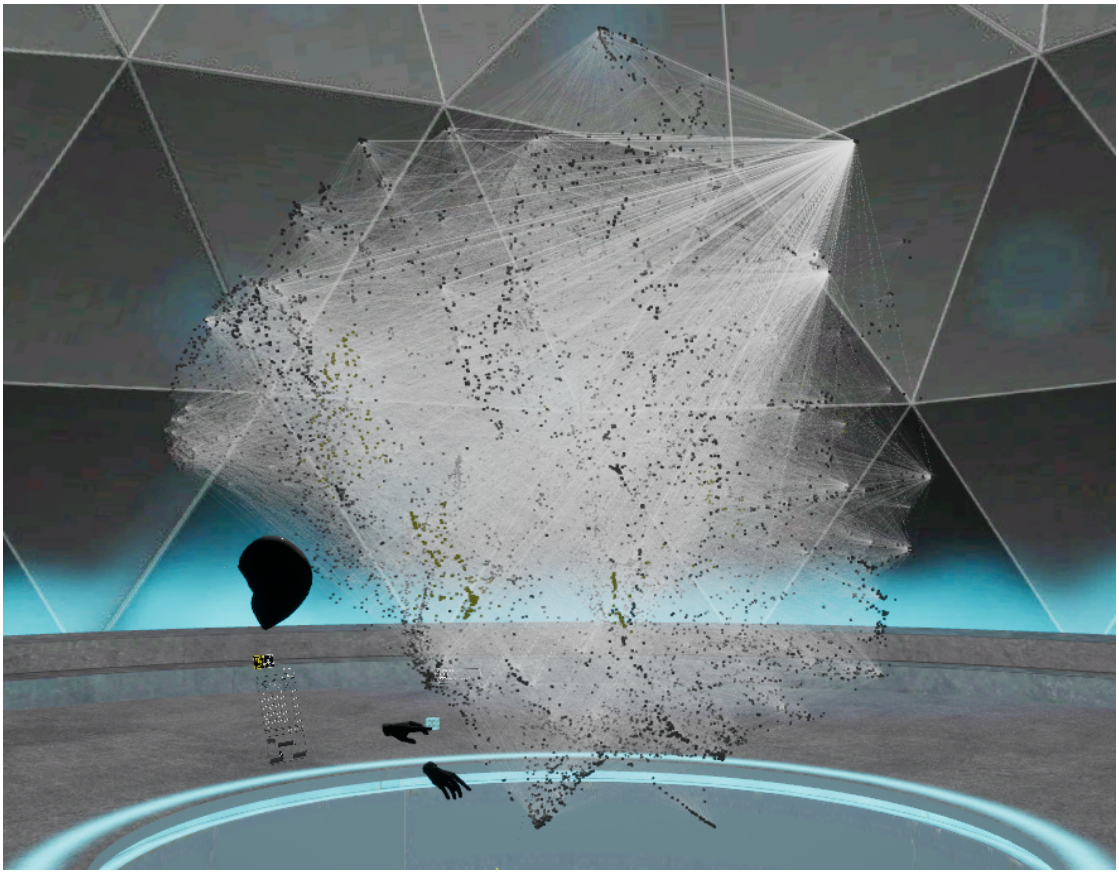


Figure 6.13: Screenshot of the data-network with the node links being visible.

node layout files and the link layout file. They are set to replicate via property replication. Then, on successful replication the necessary logic to load the file and process it on each machine is loaded. Figure 6.12 shows the Blueprint function for the call sequence after the *Network File* got changed and successfully replicated. After checking the newly replicated string to be valid it calls the MyStaticMeshNodeActor function that loads and parses the file changing the textures for the the data-network rendering. Then the relevant user interfaces are updated, e.g. refreshing the counter that contains the number of nodes. Finally, any currently spawned Actors related to the highlighting nodes are cleared since they are not up to date with the new layout anymore.

This sequence of actions is the same for the loading of the channel B node and the link files.

Replication of the data-network's internal state The state of the data-network is also held, managed and replicated in the ViveGameState. This state includes the scale, node thickness, link thickness, link transparentness of the data network and the light inside the room. They are stored as float variables. Similarly to the data-network

loading replication it is also necessary here to create *OnRep* callback functions for each property. These functions are called on each machine to assure that the newly replicated values get applied to the data-network of each machine. But instead of manipulating the textures directly, here the values get set into *Material Parameter Collections*, which are UE4 assets where scalar or vector values can be stored to globally affect materials using them, i.e. the materials for rendering the nodes and links. The sequence of changing and replicating these values is the following:

1. First the user initiates the change of a value, e.g. changing the scale slider in the UI.
2. On changing this slider the values is then sent via server RPC of the UIInteraction-Component to the server and the value then updated in the ViveGameState.
3. The server then replicates the changed value to each client.
4. Afterwards, the corresponding *OnRep* function is called that updates the values for the rendering leading to a change of the scale.

Figure 6.13 shows the data-network rendered with a small node thickness and link transparentness being set to a value that makes the links visible. The screenshot is from the view of an invisible spectator Pawn, which shows that this data-network state is shared and replicated to all users.

6.3.3.3 Replication of the Data-Network Interactions

Interactions that are slightly more complex than the previously described state changes are described in this section. These include highlighting and selecting nodes in the data-network, manipulating their visibility, etc.

There are two main interaction modes with the data-network that can be changed in the UI, affecting each user. One for highlighting single nodes to see their directly connected neighbors (depending on the currently loaded link layout) and to spawn a display that shows additional node informations. Another one for selecting multiple nodes at a time and add them to a selection set, which then could be further interacted with.

Node tracing for highlighting and information Node tracing functionality is implemented inside the TextureInteractionComponent. A user can highlight a node by touching it with the index finger on their virtual hand displayed at the position of a Vive controller. This interaction can be seen in Figure 6.14, where a user is highlighting a node and the red highlight Mesh Actors and the Hover Annotation Actor with additional information to the node appear.

The complex tracing logic and the amount of data would suggest an approach where the tracing detection is executed locally on each machine for each Pawn. This is not feasible though, because of potential inconsistencies due to non-determinism.

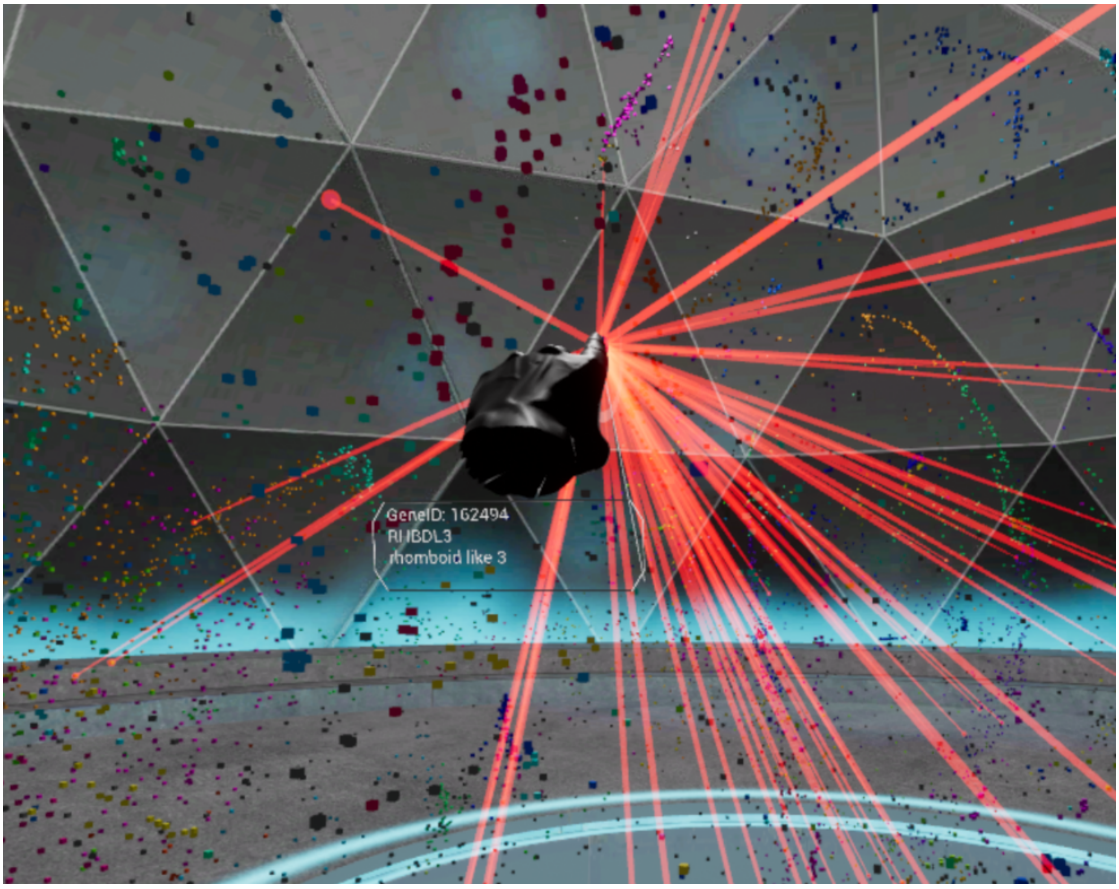


Figure 6.14: Screenshot of highlighting nodes and show a node information display.

Our solution to this problem is to only execute this tracing calculation logic on the owning client and then to replicate the results to update the server. These results include the traced node ID, and the ones of its linked neighbours. This makes only the local client authoritative of the tracing calculation and its results. The drawback of this approach is that it needs more networking bandwidth, since the amount of information sent over the network can be fairly large.

The sequence of replication is the following:

1. On each *Tick* the calculated node with its ID and position, as well as its neighbors are sent to the server. This sending is done via unreliable server RPC. It is only sent if there are tracing results and if the values changed since the last frame to reduce some bandwidth.

2. On the server this information is then used to spawn the Actors related to the visual representation. These Actors are then replicated normally through UE4's Actor replication. The *Hover Annotation* Actors consist of a widget blueprint displaying

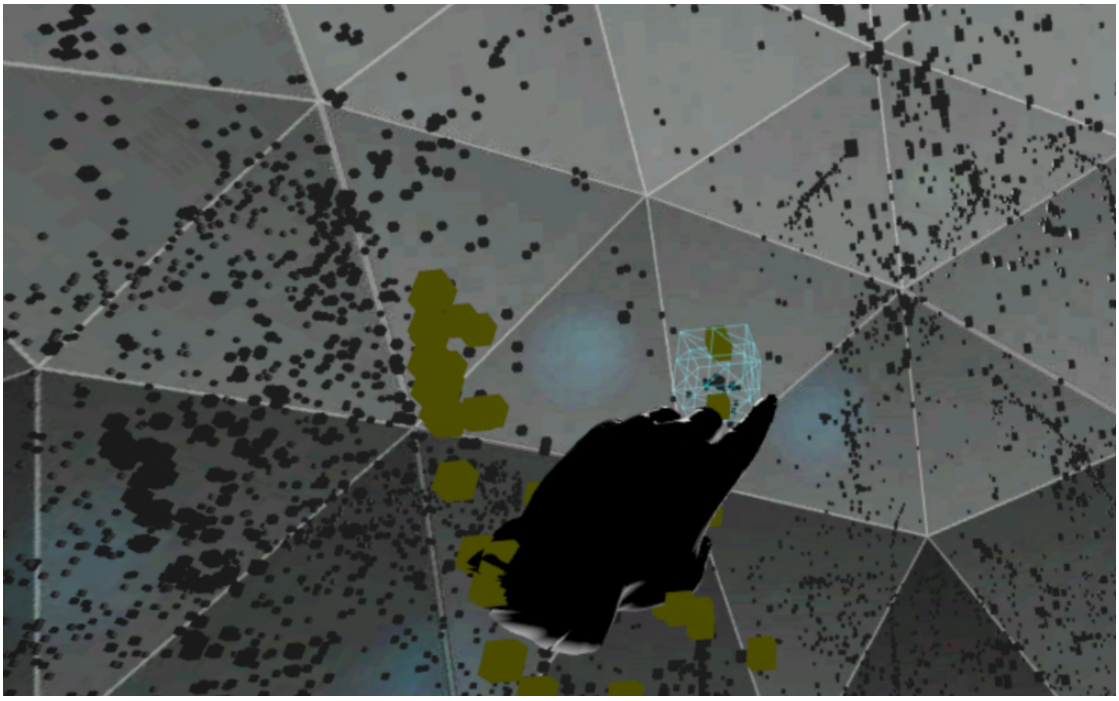


Figure 6.15: Screenshot of the brush selection of several nodes inside the cube area.

the information for the selected node, in this case the gene id and name. The other spawned Actors like the red node highlighting spheres and wires as seen in Figure 6.14 are also spawned on the server and then replicated to clients. If a new node selection is detected these Actors for the previous one got destroyed, which was also handled on the server instance.

3. Finally, related to the replication aspect of this functionality, it is possible for users to click on the Hover Annotation display to spawn another Widget Actor with a browser that loads a web page with gene information to the currently highlighted node. Due to the necessary node information being available on the server because of the previously described replication it is easy to spawn the browser Actor and simply set it to replicate.

Node selection The brush selection mode allows the selection of nodes through a controllable cube area displayed at the finger tips of the left hand, as can be seen in Figure 6.15. It works similarly to the single node tracing, except that multiple nodes can be selected. In this mode, the color of the data-network nodes that are not selected changes to gray, while the selected nodes appear in green.

This functionality is implemented in the TextureInteractionComponent. The replication sequence is as follows:

1. On each *Tick* the local client executes the tracing for the node brush selection that determines the nodes locally. Then the selected nodes are sent via unreliable server RPC to the server.

2. On the server instance these nodes are added or removed from a replicated *Selected Nodes* set. This Selected Nodes set is stored and replicated inside the ViveGameState to make it shareable between users, meaning that each user manipulates the same set of selected nodes. Similarly to the other replicated values, this set is only changed on the server instance. This selection set is also manipulated through other kinds of interactions, e.g. nodes retrieved from a database. If the nodes should be added or removed is decided by a previously set and replicated boolean value.

3. Afterwards, on each machine the changed selected nodes are recolored to make the change visible.

Interactions with selected nodes There are further interactions possible with the set of currently selected nodes. In the *Actions* part of the UI are the previously described interaction buttons for changing the brush size of the selection cube, as well as adding and removing nodes from the selection. Figure 6.19b shows the UI interactions related to node selection. The button with the symbol X clears the current selection, emptying the Selected Nodes set.

Additionally it is possible to interact with the currently selected nodes:

- *Show or hide selected nodes* The isolation of the selected nodes from the rest of the data-network is an important functionality of Holodeck, e.g. to select interesting parts of the data-network and hide the rest. It is possible to show and hide the selected nodes, or all the nodes of the data-network. Figure 6.16 shows the isolation of the previously selected nodes.

  This visibility of the selected nodes is not a clearly defined internal state of the data-network. Because of this our approach is to simply use reliable Multicast RPCs to communicate the change requests to all machines. This was works sufficiently well for our purposes but has the disadvantage that this state may be getting lost for later connecting clients leading to possible inconsistency of the current node visibility. In the future, a clearly defined and replicable state would have to be introduced to eliminate this problem.

  The sequence of actions for the implementation of the networking for this interactions is:

  1. First, the user initiates the action in the UI, e.g. *hide selection*.
  2. Then a reliable server RPC is called to communicate the action to the server.
  3. Then a reliable Multicast RPC that calls the *SetOpacity* method on the MyStaticMeshNodeActor is executed on each machine.

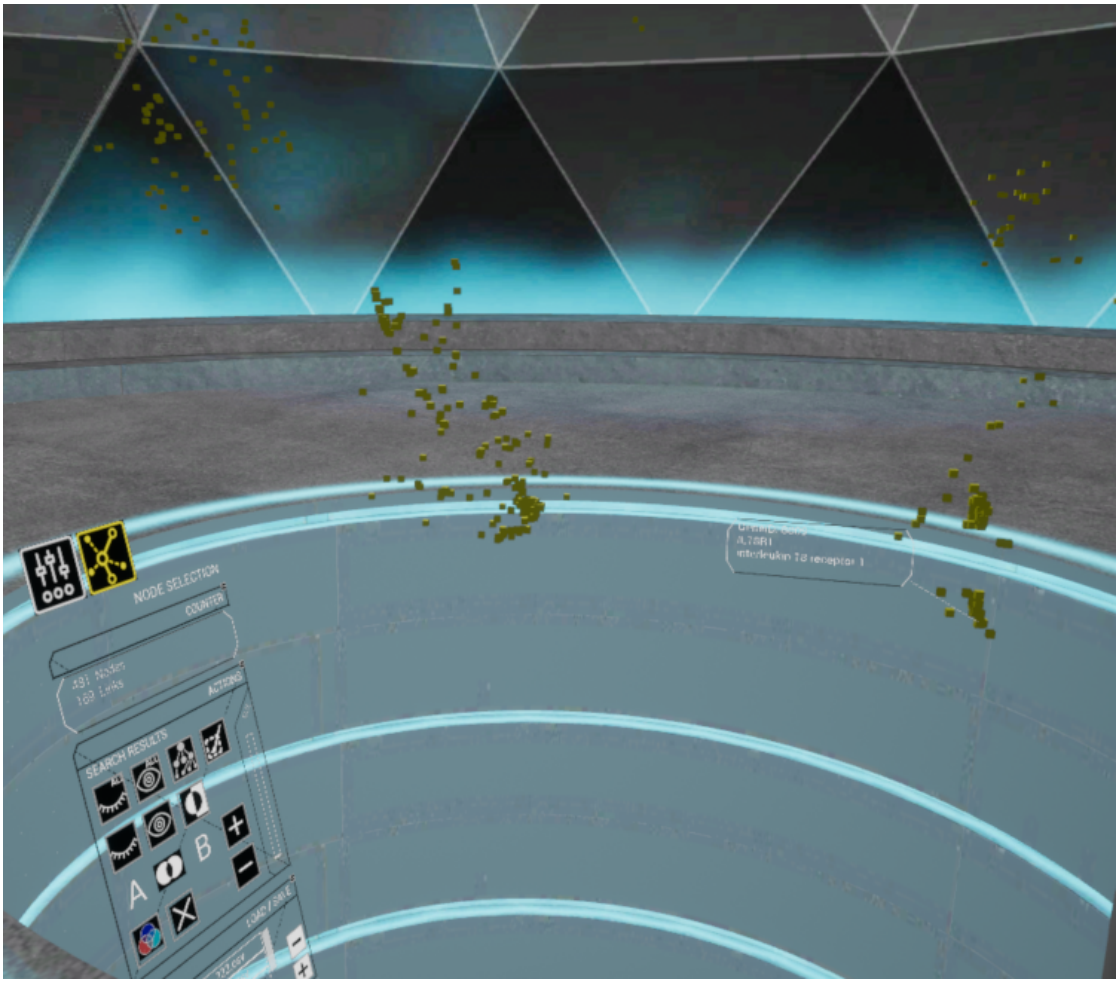


Figure 6.16: Screenshot of the selected nodes being isolated from the rest of the data-network.

- *File loading and saving of node selections* The currently selected nodes can be saved into a file and at a later time loaded again. These files are CSV files containing the node IDs. The logic for the final prototype is that files are saved and loaded only on the server. The file list display of the available files on the server is replicated to the clients.

  1. To save the current node selection a file name has to be typed into the keyboard Widget, which is shown in Figure 6.17. Then the user initiates the saving action via the *Actions* UI.
  2. This filename is then sent to the server via RPC. On the server the currently selected nodes are iterated through and the CSV file created with their IDs as its content. For saving and loading the file the *MyFileActor* C++ class is

Figure 6.17: Screenshot of the UI keyboard for text input.

used, which contains static functions for generic file loading and saving.

3. The file list of the server is then refreshed and replicated to the clients. On the clients it is then displayed in the file list view of the UI.

Database nodes The *Selected Nodes* set is also manipulated through interactions involving nodes retrieved via database access. This access happens through web services that are only reachable when using the CeMM virtual private network (VPN). Without the VPN the relevant UIs are not being rendered and not usable.

The *Disease Tree* UI contains buttons for diseases to retrieve the nodes from the CeMM web service related to them. This buttons are dynamically generated and presented a hierarchical tree of diseases and disease groups, as can be seen in Figure 6.18. Clicking on a button sends a request to get its sub diseases and display them as buttons, as well as send a request with all nodes related to the current disease or disease category including the sub diseases. It is then possible to isolate these nodes in the currently displayed data-network to show them in the current node layout. As an example, the screenshot in Figure 6.18 shows the displayed buttons after clicking the *Cardiovascular Diseases* button and then *Heart Diseases*. At that point the nodes related to diseases contained in the *Heart Diseases* group are retrieved and can be added to the *Selected Nodes* set to isolate them.

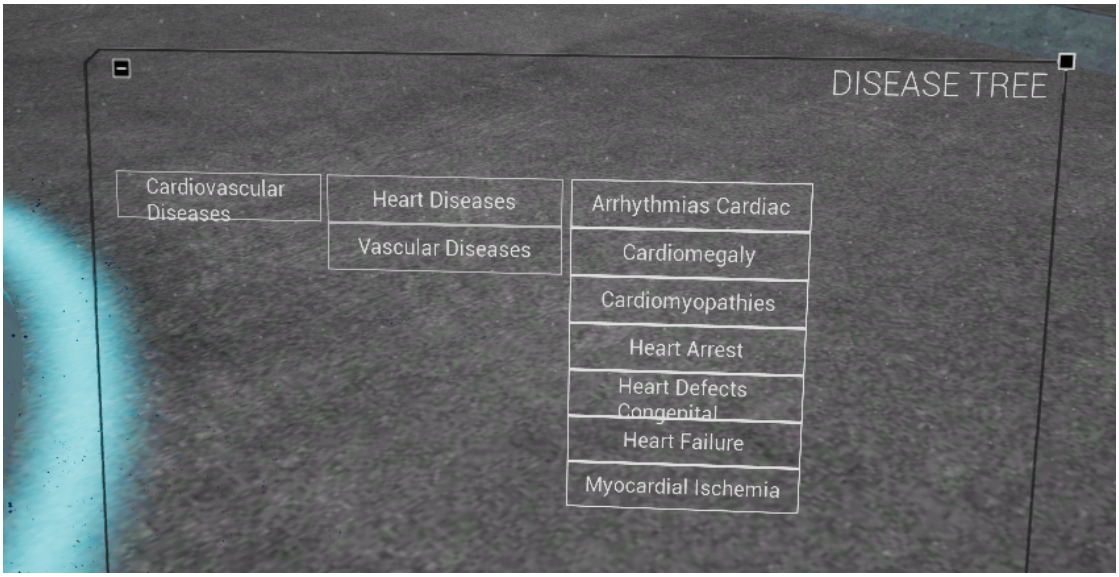


Figure 6.18: Screenshot of the diseases UI showing a hierarchical tree of diseases that can be isolated inside the data-network.

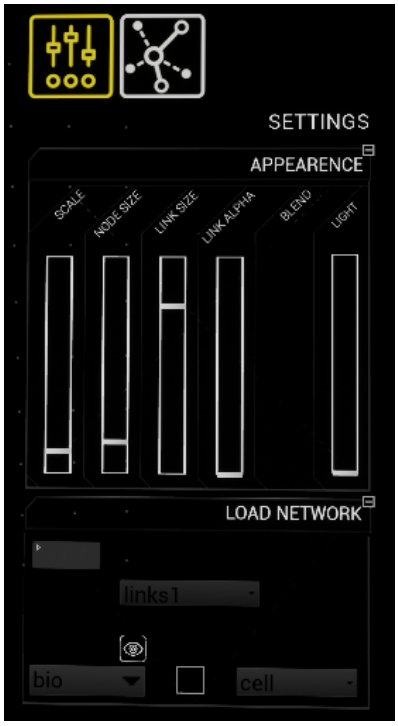
Web service communication Responsible for calling the CeMM web services is primarly the *PHPInterface* Actor, which is a singleton object placed into the world that can be accessed to request the backend calls. This web services provides simple Representational State Transfer (REST) endpoints to access the CeMM database. For this communication the *VaRest* plugin [22] is used, which is available in the UE4 Marketplace for free. It is designed for REST communication and includes the HTTP request and response handling, parsing of JSON values and has full accessibility through Blueprints.

The main relevant aspect for the networking implementation in this thesis is that the request handling logic needs to be changed to only be executed on the server. To achieve this, the PHPInterface Actor is set to only exist on the server. This means that this REST communication happens only between the server instance and the web services. To initiate the interaction from the client it is necessary to call a server RPC to start the requests and then the response is handled on the server side and the results then replicated. The server RPCs are handled by the UIInteractionComponent.

As a sequence example for *RequestDiseaseQuery*:

1. The UIInteractionComponent sends the ID of the pressed disease button via server RPC to the server. Then it calls the *RequestDiseaseQuery* function of the PHPInterface, which defines the web service endpoint and parameters.

2. This web service request is then sent in this method and a callback event defined to handle the response.



(a) Settings User Interface

(b) Node selection User Interface

Figure 6.19: Screenshot of the main switchable User Interface in the final Holodeck prototype.

3. On response of the web service the callback event is called, depending on the target request (i.e. in this case *RequestDiseaseQuery*.
4. The response JSON is then processed. The extraced values are then replicated inside the UIInteractionComponent and the UI updated on each machine.

6.3.3.4 UI Widget State and Interaction Replication

All UI Actors are placed into the world-space and can be seen by all the connected users. Because of this the interactions with the UIs also need to be visible and therefore the UI state replicated, e.g. slider positions, button press and hover animations, etc. Further, the registered UI actions need to be sent from client to server where the interaction logic is then executed and the state updated. UMG Widgets are by design not replicable and the Widget Actors containing them are not owned by any particular user. They are intended

to be used freely by users that have the interaction capabilities. Because of this it is not possible to define server RPCs there, since as observed earlier this type of RPC can only be executed by Actors that the client owns. Instead the UIInteractionComponent is used, which is contained by certain player Pawns that should have UI interaction capabilities, e.g. VivePawn. Other Pawns, like the spectator Pawns, are not supposed to interact with these UIs and therefore do not have a UIInteractionComponent. This Component is mainly intended to handle this type of UI action replication from client to server to change the state on the server, e.g. in ViveGameState.

Movement of the UIs The UMG Widgets are contained in so called *DragAbleWidgetActor*s, which are placed in the world and attached to a cylindric shape that is attached to the room. This cylindric Actor denotes the area where UI Widgets can be moved and sticked to. The movement of a Widget is initiated when a user touches the upper-most part of it. Then the Widget sticks to the finger, following the player's controller movement. To let go of the Widget it is necessary to move it to some part of the cylinder area where it gets attached again.

The movement logic to realize this is included in the DragAbleWidgetActors, which also handle the replication of the movement through UE4's default Actor movement replication. A collider box fires an event when the user's finger touches it on the server instance of the Actor, which then attaches itself to the users's controller. Similarly the ending of the movement happens when a collider overlaps with the cylinder Actor and detaches itself from the user and re-attaches to the cylinder.

SwitcherWidgetActor The main data-network interactions UMG Widgets are contained by the *SwitcherWidgetActor*. It consists of two sub Widgets that can be switched when pressing their respective buttons. Figure 6.19 shows two screenshots of each of them active. The SwitcherWidgetActor also holds the state of its UI that is directly related to the UI components and replicates it. This state includes booleans for the visibility of each sub part of the widget, which controlls if a certain area is expanded or not expanded.

The *Settings* part of the SwitcherWidgetActor Widget contains the data-network state settings. This state is replicated by the ViveGameState, which is also responsible for updating the UIs elements. It does so by calling the provided methods inside the UMG Widgets that further set the newly replicated values into the UI elements. For example the scale slider update happens with ViveGameState calling *SetScaleSliderValue()* with the new value, and this method then converts the value and sets it in the scale slider.

The *Node Selection* Widget contains the counter Widget for the currently visible nodes and links, as well as the node selection interaction possibilities that were already explained in detail in the previous section 6.3.3.3. Figure 6.19b shows the Widget. Apart of the already mentioned Widget visibility state there is no further replication necessary.

CHAPTER 7

Evaluation

In this chapter, the results of the implementations are evaluated and reflected upon in regards to the previously identified requirements and thesis goals.

For the performance efficiency NFR evaluation tests with measurements are presented and evaluated in Section 7.1. The critical reflection of the VRNet plugin and CeMM Holodeck implementations is described in Section 7.2.

7.1 Performance Evaluation

To investigate the performance efficiency of the VRNet plugin we mainly focus on networking bandwidth tests and the subjectively experienced smoothness during testing. We also look at the general resource usage of the plugin.

7.1.1 Resource Usage Evaluation

For the evaluation of the runtime resource usage we used an early CeMM Holodeck prototype version right before the plugin integration and compared it to the version after the plugin integration. The only difference between this versions is the inclusion and loading of the VRNet plugin and that the relevant gameplay classes are parented to the VRNet classes. For the measurements we used the *Performance Monitor* tool provided by Windows with counters for processor time and committed bytes in use percentages.

As we initially expected no significant differences in CPU time or RAM used are observable. The integration of the plugin, at least by itself, is not leading to a noticeable degradation of the runtime performance.

The storage space of the packaged VRNet plugin amounts to around *65 MB*. The majority of this storage space is made up by the compiled binary files.

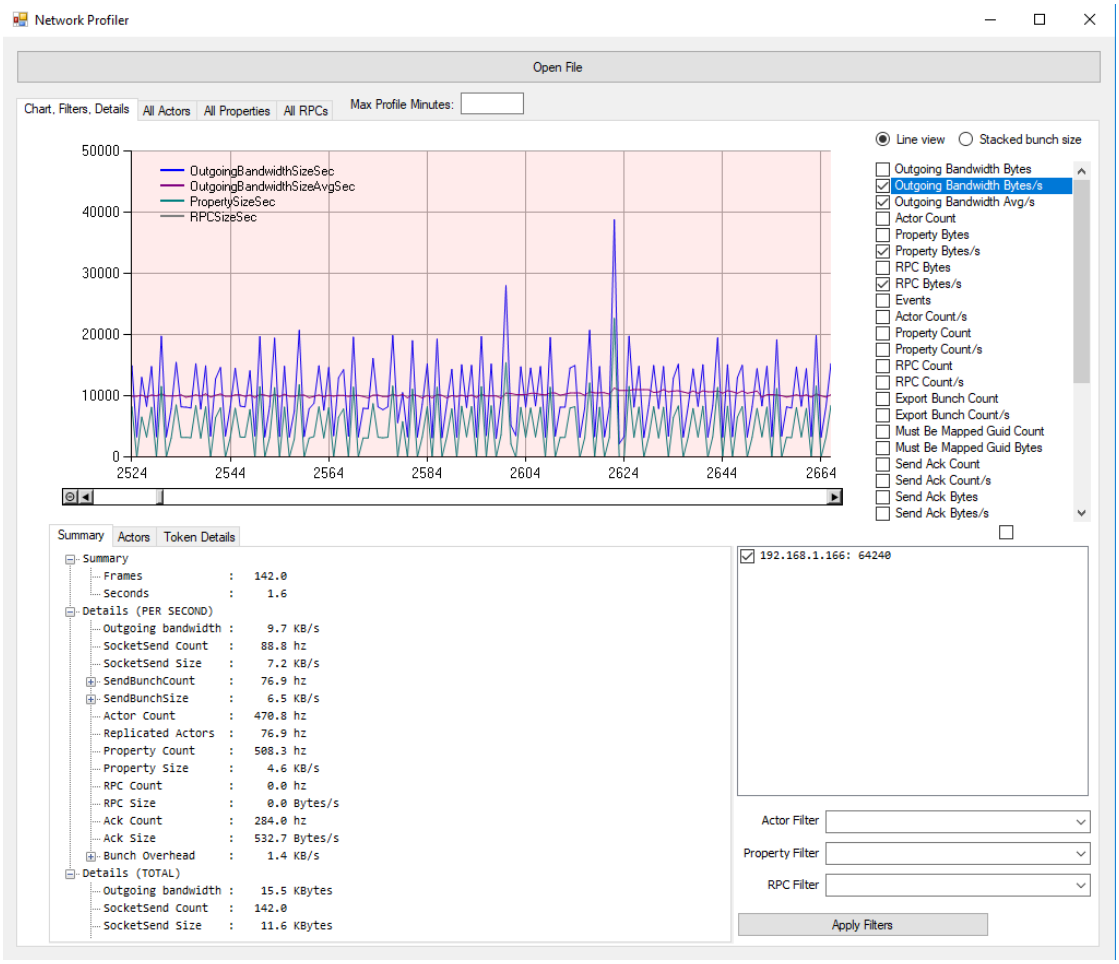


Figure 7.1: UE4 Network Profiler tool for analyzing networking performance efficiency.

7.1.2 Network Evaluation

To evaluate the networking related performance efficiency of the VRNet plugin functionalities we used the *Network Profiler* provided by UE4. This standalone tool can be used to investigate network traffic and performance. Figure 7.1 shows a screenshot of the UE4 Network Profiler tool. It is possible to display the overall statistics and how much bandwidth individual actors, RPCs and properties contribute to the total amount. Additionally it is possible to investigate selectable portions of the session and see the values in graph form.

Before using the network profiler it is necessary to record a profiling session file. To do this the engine needs to be built with stat tracking enabled and either *"networkprofiler=true"* passed as a command-line argument on engine startup, or with console commands at runtime *"netprofile enable/disable"*. The profiling data file will then be saved to *<project directory>/Saved/Profiling/<project name>-<timestamp>.nprof*, which can be loaded

with the network profiler tool to display the data.

When viewing a profile session it is important to take into consideration that only outgoing bandwidth is recorded. This means that on clients there is only shown RPC data that is sent by the client. On the other hand, when recording on the server the actor and their property replication, as well as RPCs sent to the clients can be seen.

Several tests were conducted with different investigation goals in mind. For VRNet in the first evaluation we analyzed the bandwidth of a host and a client and further describe it in Subsection 7.1.2.1. The next VRNet plugin test described in Subsection 7.1.2.2 investigates the bandwidth increase with each additional connected clients. For the CeMM Holodeck project we also conducted a detailed performance evaluation test on the final prototype demo version, which can be seen in Subsection 7.1.2.3.

7.1.2.1 VRNet - Client and Host

	Host	Client
Outgoing Bandwidth (KB/s)	9.8	5.9
Property Count (hz)	507	0
Property Size (KB/s)	4.6	0
RPC Count (hz)	0.8	104.2
RPC Size (KB/s)	0.003	2.5

Table 7.1: Results of the VRNet client and host comparison.

This VRNet plugin evaluation test investigates the network bandwidth with a test setup of two users, one being the host and the other a client connected to them. For this test we used the VRNet plugin test Level that was used for developing and testing the plugin functionalities and provided in the VRNet template. The contents and functionalities provided in this Level are described in the VRNet plugin implementation chapter 5. Specifically for this test, apart from moving around the player Pawns we used grabbing functionalities and throwing objects around, as well as teleporting. The test time was five minutes long.

Table 7.1 shows the outgoing bandwidth per second and the replication differences between host and client. Property replication is only happening on the server side in UE4, as can be also seen in the profiled data. RPCs are used on each machine, but they are only really impacting the total bandwidth on the client, because they have to send their movement data in relatively high frequency to the server. The host's player movement data is directly replicated via property replication. Other client's player movement data would also be replicated via the server's property replication after their RPC update if further clients would be connected.

While reducing the update rate of the client's RPC replication or the server's property replication could be a first point for optimization efforts, reducing it might lead to visibly

decreased smoothness. More sophisticated interpolation and extrapolation techniques might be needed to mitigate these effects.

After the five minute test the total outgoing bandwidth reached around *3.1 MB* on the host and *1.8 MB* on the client.

7.1.2.2 VRNet - Changes with Connected Clients

For this evaluation we investigate the change in network performance efficiency for different numbers of connected clients to try and extrapolate the results to speculate about how the plugin will perform with higher numbers of simultaneously connected clients.

Again, the VRNet plugin default test Level was used to record for five minutes a networking profiler session. Each consecutive test added another connected client which was moved through an improvised contraption used by one tester to control 2 clients at a time, alternating between grabbing through one client to the other while constantly moving around both. We tried to keep the actions inside the tests as similar as possible. They included moving around, grabbing objects and teleporting around.

	Test 1	**Test 2**	**Test 3**
Connected Clients	1 Client	2 Clients	3 Clients
Outgoing bandwidth (KB/s)	9.8	19.6	29.3
Property Count (hz)	507.2	1064.2	1801.3
Property Size (KB/s)	4.6	9	14.4

Table 7.2: Results of the host for the VRNet tests with different amount of connected clients.

The outgoing networking bandwidth per second results for the host can be seen in Table 7.2. In this comparison we omitted the RPC data of the host, due to it being negligible. Interestingly, each additional client seems to add a little bit less than *10 KB/s* with apparently linear growth. On the other hand the property count and size comparison show that the amount of property replication did increase significantly in *test 3*. Possibly just not enough to have a noticeable effect on the total bandwidth, which might also be due to UE4's replication optimizations. The hosts FPS was constantly around 90 FPS.

While not directly observed in our test, each connected client is also adding CPU overhead for the host, so the limiting aspect might become FPS instead of network bandwidth. Especially for LAN only projects where network bandwidth is not a very limiting factor.

A host has to handle all the player input and rendering additionally to the replication requirements and connection handling. A first optimization step could be to deploy a dedicated server on a different machine that focuses on serving the clients without having the overhead of rendering and handling a host player.

As for the clients' bandwidth, there is no significant observed difference that cannot be attributed to different usage behavior. The bandwidth is mostly independent to the amount of clients connected at the same time. (except minor replicated client information)

7.1.2.3 CeMM Holodeck - Client and Host

	Host	Client
Outgoing Bandwidth (KB/s)	6.4	5.7
Property Count (hz)	252	0
Property Size (KB/s)	2.2	0
RPC Count (hz)	4.7	101
RPC Size (KB/s)	0.025	2.3

Table 7.3: Results of the CeMM Holodeck client and host comparison.

For the CeMM Holodeck evaluation we conducted a ten minutes long test using most of the functionality provided in the final prototype. The version of the project for the test was the same that was used for the final presentation version.

The test consisted of a list of following interactions, first executed by the host user, and then by the client user.

1. move the data-network to the center of the room
2. change the scale, node size and light
3. load another network in channel B and use Blend functionality
4. rotate and move data-network around and select nodes
5. switch to selection mode and select nodes with the selection brush
6. isolate, hide and show selected nodes and all nodes
7. move UIs around and expand/hide parts

Table 7.3 shows the overall results per second for the host and the client. The values are similar to the VRNet plugin results, with the main difference that the host is sending less data in total. This can be lead back to the fact that less objects in the world are replicated constantly than in the VRNet template test map. Mostly the player Pawn movements are replicated, as well as the RotPoint transform changes and data-network state values changes.

In this evaluation we also had a closer look at the more detailed replication to see which parts of the project are responsible for what bandwidth. Table 7.4 shows the replication data of each Actor as listed by the glsue4 Network Profiler tool. This data makes it

clear that most of the replication is happening in the Pawn, mostly because of the player movement replication, but also because of its data-network interaction, which is a part of the Pawn as the *TextureInteractionComponent*. Other Actors like *RotPoint* are not moved as much, and contrary to the player movements, the check to only replicate when the transform changes is very effective, since movement is only occurring if one of the players is initiating the navigation. The *GameState* Actor handles the global data-network and other game state, and especially values changed by sliders are replicated here and add up to the value of *13.2 KB*. The so called *3DwidgetHolderINFOhover_C* Actors are the ones spawned for additional information to selected nodes and due to being frequently shown on normal usage they add up *9.1 KB* of network bandwidth. The rest of the Actors, including the *VRPlatform* and *uiCylinder* that are shown in the table and many UI Actors that were omitted, are already pretty low contributors to the overall bandwidth.

Total Size (KB)	**Count**	**Average Size (Bytes)**	**Actor Name**
1344.2	38517	35.7	VivePawn_C
32.8	28175	1.2	rotPoint
13.2	24258	0.6	ViveGameState_C
9.1	6809	1.4	3DwidgetHolderINFOhover_C
0.2	1608	0.1	VRPlatform
0.1	1302	0.1	uiCylinder

Table 7.4: CeMM Holodeck host Actor replication network data.

Total Size (KB)	**Count**	**Average Size (Bytes)**	**Property Name**
1377	153244	9.2	ReplicatedTransform
6.4	1633	4	Blend
5.8	300	19.8	geneName
4.9	1244	4	SelectedGenes
0.6	166	4	NodeThickness

Table 7.5: CeMM Holodeck host property replication network data.

Furthermore, Table 7.5 lists the most replicated properties, categorized by the property name. As already noted earlier the notion that mostly movement data is responsible for the networking bandwidth gets reinforced, because *ReplicatedTransform* is by far the highest contributor to the total with *1377 KB*. This includes the player movement data, as well as the data-network movement data. Other than that the replicated properties mostly relate to the data-network and selection state.

As for the host's RPCs the results are shown in Table 7.6. They are not contributing much to the total bandwidth as observed earlier in the overall statistics. Many of the

Total Size (KB)	Count	Average Size (Bytes)	RPC Name
56.4	101	572.1	MultiSpawnSelectionIndicatorsOnTick
7.4	4647	1.6	MultiTickBrushSelect
0.4	245	1.8	MultiClearAllSpawnedActors
0	3	4.1	MultiIsolateSQLGenes
0	2	4.1	MultiShowAll

Table 7.6: CeMM Holodeck host RPC network data.

Multicasts were removed from the table due to not having noticeable effect on the total size, since they are only called a few times and have around *4 Bytes* each call.

Total Size (KB)	Count	Average Size (Bytes)	RPC Name
1445.2	61813	23.9	ServerSetTransform
48.8	82	609.3	ServerSpawnSelectionIndicatorsOnTick
10.4	119	89.8	ServerTickBrushAddSelect
2.6	327	8	ServerSetNodeThickness
1	125	8	ServerSetLight
0.9	146	6.2	ServerSetLinkAlpha

Table 7.7: CeMM Holodeck client RPC network data.

On the other hand, for the clients the RPCs make up most of the bandwidth, due to the client-authoritative movement updates for the player movement data and the data-network movement data. The table 7.7 shows an extract of the client's RPCs. The *ServerSetTransform* RPC is the one called in the *VRNetReplicationComponent* and amounts to *1445 KB*. The *ServerSpawnSelectionIndicatorsOnTick* is called much less often, but due to there being a list of nodes and other data sent each call, the average size is relatively high with *609.3 Bytes*. Similarly *ServerTickBrushAddSelect* is also used for the node selection where a lot of client data needs to be sent to the server. User interactions through the UIs' sliders also lead to many server RPCs to be sent to update the values, though they contribute relatively little to the total amount due to the small size of the values.

7.1.3 Performance Conclusions

Overall the networking bandwidth requirements are already relatively low for multi-user VR applications. Also, we expect that a sufficiently high number of simultaneously connected users are supported before noticing limits.

It is important to remember that for these tests only minor bandwidth optimizations were used and that the smooth player experience was the priority. If reduced band-

width becomes a priority the VRNet framework and UE4 provide many optimization configuration possibilities.

Another evaluation aspect for the performance evaluation tests was the subjective smoothness for each player. While naturally hard to quantify we can say that it was satisfactory. Especially for the demonstration of the final prototype of the CeMM Holodeck the reactions of the users, i.e. the developers and other stakeholders of the project of the CeMM institute, was very positive.

The storage, CPU and RAM usage of the plugin is very low, which is especially important for projects that only intend to use certain functionalities of the plugin.

Because of these results we think that the goals that we set for the performance efficiency are reached by the efforts of this master thesis.

7.2 Critical Reflection

7.2.1 VRNet Discussion

The main defined functional and non-functional requirements were fulfilled for the VRNet implementation.

The VRNet framework provides the necessary tools for user movement replication. The plugin also provides the foundation and some basic examples of user avatar representations, including a complex avatar that replicates the position data of a dynamic number of tracked points. It is possible to swap these avatars dynamically during development and at runtime. The provided examples are still basic though, and may need to be improved before using them for more complex player representations.

User input and interaction handling and replication is another major contribution of the VRNet framework. The provided system provides a default implementation for user interaction with objects via grabbing. It can be further extended with new interaction types.

VRNet provides diverse game management functionalities that are important for any kind of multi-user application. This management facilitates connection handling, with the possibility for different users to connect over LAN using a known IP address or over Internet using Steam matchmaking. Simple UIs to access these functionalities are also available in the VRNet project template. Additionally, VRNet provides some useful utility classes for helping in the development, debugging and testing of networked VR applications. The introduced logging macros for UE4 logging framework can be used to identify problems on the correct instance and machine. The provided logs in the VRNet classes at crucial events also help with keeping an overview of the application state. These management and utility classes proved to be valuable during development and testing of the VRNet plugin, as well as the CeMM Holodeck implementation.

Many measures were taken in the design process to make VRNet as easily usable as possible. Functionality of the plugin is structured into independent and easy-to-add

Components, providing a flexible foundation for the development of multi-user VR projects. The VRNet framework further enables the development of multi-user projects entirely through Blueprints. All the crucial VRNet functionality can be easily accessed via Blueprints. The CeMM Holodeck implementation is an example of how to use the framework exclusively in Blueprints.

Another important part was the inclusion of a project template that works as a starting point for developers. When starting with this project template the effort that is necessary to provide the base networking functionality for multi-user VR is low. Naturally, the implementation of project specific interaction logic still needs to be taken into consideration when planning to develop such an application.

Furthermore, *maintainability* NFR was a major focus for the VRNet design and implementation. Many of the Components and Actors in the plugin can be easily exchanged, extended or reused. Application developers can also modify the VRNet classes themselves if necessary, because the plugin can be integrated directly to projects, including its source code. But for most scenarios, using the provided configuration and extension capabilities should be sufficient.

As for *testability*, we see this requirement partly fulfilled. While we provided debugging convenience tools and implemented the classes with testability in mind, the design and implementation of automated tests like unit tests or integration tests were left for future work to be done.

7.2.2 CeMM Holodeck Discussion

CeMM Holodeck's networking implementation requirements were fulfilled with the help of VRNet. The integration of the VRNet plugin into CeMM Holodeck enables the replication of users' movement and avatar. It also provides the necessary game and session management functionality to use and test the application.

Interactions with the data-network, like its movement, were implemented with the help of the client-authoritative replication implemented in *VRNetReplicationComponent*. Further development of the application-specific functionality required the implementation of replicated interactions that go beyond the general functionality of the VRNet framework. These include interactions with the UIs that change the state of the data-network, like its scale, visibility, node layout, and so on. Node selection handling and its replication were implemented in a way to make it possible for users to collaboratively select nodes and modify their visibility and color.

The integration and implementation was challenging due to the initial prototype being already in a fairly advanced state without networking in mind. Extensive refactoring of the code base was necessary to prepare for the networking implementations. These challenges could have been mitigated by taking networking into consideration earlier, ideally from the beginning.

The fully networked final prototype presented in this master's thesis was demoed and presented at the end of the project. It was also shown on multiple occasions by CeMM to biologists and the general public, e.g. at the the Austrian *Long Night of Research 2018*[1] The reception of the final multi-user prototype was positive. The informal feedback of several researchers who tested the system throughout the development is encouraging.

Future plans include testing the system with more researchers to gather more feedback. The project is still ongoing and further functionalities will be added in the future.

7.2.3 Problems During Implementation

During the implementation of the projects we encountered time consuming problems that we did not anticipate before. UE4 is a very large game engine and naturally brings with its many features a lot of issues and bugs with it. In our development we encountered many of such issues that we needed to resolve with workarounds.

- **Corruption problems with Blueprints:** Blueprints were often making problems when developing in conjunction with C++. One issue was that Blueprint classes frequently got corrupted in certain ways, for example by Component references being overridden with null references. To fix this it was necessary multiple times to change the Component reference access to be public in the source code. Otherwise the editor does not allow to change it (in this case, to set it back to the default Component reference). But since it is not desirable to be public, because neither a developer nor the engine should actually change the reference, it needed to be changed back. Other Blueprint corruption issues were encountered when trying to change the parent class of Blueprint classes to other C++ classes, which basically is not possible without recreating everything from scratch.

- **Abstraction and interface problems in UE4:** Another problematic aspect during implementation was how UE4 implements and uses interfaces. It is very complicated and error prone to include even simple interfaces in C++ code. Also, verbose and hard to read boilerplate code is necessary to create them. Furthermore, it is not possible to keep a typed reference to the interface. Rather it is necessary to have e.g. a reference of type Actor and then dynamically cast it to the interface every time to access it, making it cumbersome to use. Because of this reasons we decided to only make limited usage of the UE4 interfaces in the C++ classes of the VRNet plugin, even though conceptually it would have made sense to use more. On the other hand, Blueprint interfaces proved to be more useful and were used a lot in the refactorings and networking implementation of CeMM Holodeck.

- **Testing problems:** The testing of VRNet, as well as the CeMM Holodeck project was another challenging aspect during implementation and evaluation. Ensuring

[1] https://www.imba.oeaw.ac.at/about-imba/general-news-press/long-night-of-research-2018/

that the infrastructure for the test was set up accordingly was hard, because of often having to change the available hardware and dealing with connection and installation problems. Additionally SteamVR was sporadically making problems where either controllers needed to be repaired or the headset rebooted. This lead to most of the testing sessions taking additionally one to two hours longer for getting the infrastructure up and running before starting the actual testing. To mitigate these problems, instead of changing the testing hardware setup, it would be helpful to use dedicated hardware that is only used for the project's development and test setup and to keep the software up-to-date.

- **Version control and merging problems:** The UE4 Git support proved to be unstable at times, making it often impossible to use the merging tools efficiently. Merge conflicts, especially in the Holodeck project where constantly divergences happened were very hard to merge at times. The merging had to be manual with having to take care to not break the functionalities or to not forget changes. This is another big part to be considered when planning to use Blueprint heavy implementations when developing with several developers. The text based C++ source code is much easier to automatically merge and see changes between revisions. To mitigate this issues we tried to split up the Blueprints into several files to limit the potential for conflicts happening and to ease the coordination while developing.

7.2.4 Comparison to Related Work

Compared to the frameworks and systems discussed in Chapter 2, UE4, combined with the VRNet framework, offer a modern and powerful tool set for developing multi-user VR applications that is relatively easy to use. After the experience with developing in UE4 and the previously described problems we think that one of the reasons of the current dominance in usage for VR of Unity over UE4 is that UE4 has a steeper learning curve, requires the usage of C++ and to dive into the engine source code occasionally. We think that with providing the VRNet framework on top of UE4 some of these problems will be less striking for developers implementing multi-user VR applications. Especially with the help of UE4's Blueprints and the focus of VRNet to expose its functionalities and make it configurable makes it much easier to quickly build applications. Furthermore, also other useful engine functionality that is only accessible through C++ is exposed in Blueprints by VRNet.

UE4 and especially Blueprints are still under development and get improved with each UE4 release. Because of this it will be necessary to also maintain the VRNet framework to have an up-to-date development environment. One of the benefits is that new or old shortcomings of UE4 in general and in regards to multi-user VR development can be improved inside the VRNet framework.

Using the VRNet framework and UE4 offers a great, feature-rich and potentially better alternative to Unity and other state-of-the-art frameworks for developing multi-user VR applications.

CHAPTER 8

Conclusion

This master's thesis focused on the design and implementation of a framework that eases the development of multi-user VR applications. Challenges for multi-user VR applications are identified, including assuring high responsiveness in a networked environment while also satisfying consistency requirements. Aiding rapid prototyping and dealing with the complexity of the software development and maintenance of multi-user systems is also challenging to achieve.

Many state-of-the-art tools for multi-user VR application development were researched and analyzed. The UE4 game engine was chosen as the foundation of the VRNet framework. It offers a modern toolset and development environment that is necessary to develop complex and high-end 3D applications. The underlying networking system however, was not designed to be used in local walkable VR scenarios. Therefore, extensions to this system were necessary.

VRNet was designed and implemented in the form of a UE4 plugin, which is easy to integrate into new or ongoing UE4 projects. The major contributions of the framework are the implementations of user movement, user interactions and user representation replication. Client-authoritative replication was chosen as the main replication type to provide the smoothest possible experience to users, unhampered by potential network delays. VRNet provides a solid implementation with several configuration possibilities. Furthermore, VRNet introduces crucial application management functionalities and easy access to UE4's online session system. To help with development debugging utilities are provided.

CeMM Holodeck is an application focused on the visualization and shared interaction with Interactoms - large data-networks representing collections of protein molecules. Another goal of the thesis was to integrate VRNet into the CeMM Holodeck project to make it fully multi-user capable. CeMM Holodeck brought additional challenges and requirements that had to be overcome and lead to the improvement of the VRNet plugin

as well. Apart of that it was necessary to take further steps to implement the networking functionalities specific to CeMM Holodeck to allow collaborative interactions with the data-network.

The successful integration of the VRNet plugin showed that it contributes with a useful foundation for the creation of multi-user VR applications. VRNet, as well as the underlying UE4 engine provide many configuration possibilities for networking, offering developers control for networking performance optimizations. Furthermore, evaluation tests for the performance efficiency of the VRNet and CeMM Holodeck implementations were conducted. The results for the networking traffic show that the used bandwidth is acceptable for providing smooth user experience. There is still a high potential for further networking performance optimizations. The results for the resource usage show that the integration of VRNet has no performance degrading effect on projects.

To conclude, we think the VRNet plugin in combination with UE4 provides a good foundation for developing modern multi-user VR applications with high-end graphics. It enhances the development efficiency and provides many functionalities supporting the development.

8.1 Future Work

In general, further manual testing and refinement of functionalities will be the logical next steps in the future. Apart of that there are many possibilities to enhance and further develop the VRNet plugin to make it more useful and stable.

- **Introduction of new avatar types:** While the plugin provides the foundation and basic implementation for player avatars, more sophisticated avatars would be useful for multi-user VR projects. For example introducing a SkeletalMesh avatar that utilizes IK or an improved implementation of VRNetMotionCaptureAvatar.
- **Introduction of more interaction types:** In its current state, VRNet only provides grabbing as a default user interaction. Further useful interaction types could be developed to increase the VRNet's interaction repertoire.
- **Introduction of automated testing:** The VRNet plugin is currently completely untested by automated tests. Considering that UE4 provides a feature rich automation framework to help with the creation and execution of automated tests an important next step would be to develop such tests. This would ensure a stable and well documented code base that is easier to maintain and enhance.
- **CeMM Holodeck future directions:** Future plans for CeMM Holodeck include testing with more researchers to gather more feedback. This test may be executed as formal user studies. Furthermore, the project is still ongoing and more functionalities will be added in the future. The introduction of such features need to have replication in mind from the beginning.

List of Figures

List of Tables

List of Listings

Acronyms

AR Augmented reality. 7, 8

CSV Comma-separated values. 83, 85, 106

FPS Frames per second. 67, 76, 114

GPU Graphics processing unit. 1, 16

HLSL High-Level Shading Language. 15

HMD Head mounted display. 1, 2, 8, 16, 20, 28, 29, 31, 47, 48, 55, 56, 71, 74, 76, 95, 98

IDE Integrated development environment. 14

IK Inverse kinematics. 76, 124

NFR Non-functional requirement. 12, 21, 22, 28, 81, 111, 119

OSS Online Subsystem. 37, 38, 63, 65, 66, 68, 70, 95, 127

REST Representational State Transfer. 108

RPC Remote procedure call. 25, 26, 33, 34, 36, 44, 48, 53, 58, 61, 62, 65, 96, 99, 102, 103, 105, 106, 108, 110, 112–114, 116, 117

SDK Software development kit. 9, 70

UBT Unreal Build Tool. 14, 41

UE4 Unreal Engine 4. 1, 3, 4, 10, 12–17, 19, 20, 22, 24–30, 32, 36–39, 41–51, 53, 54, 58, 62, 63, 66, 67, 70–75, 80, 84, 85, 87, 93, 99, 100, 102, 103, 108, 110, 112–114, 118, 120, 121, 123, 124, 126

UHT Unreal Header Tool. 14

UI User interface. 15, 26, 31, 53, 63, 67, 76, 77, 82–84, 86, 89, 90, 93, 95–97, 102, 105–107, 109, 110, 115–119

UMG Unreal Motion Graphics. 15, 38, 67, 95, 109, 110

UML Unified Modeling Language. 13, 32, 36

VCS Version control system. 16

VPN Virtual private network. 107

VR Virtual reality. 1–3, 5–10, 12, 13, 16, 17, 19–22, 27–29, 32, 36, 38, 39, 49, 51, 63, 67, 71, 72, 74, 75, 77, 79–81, 83, 94, 96, 99, 117–119, 121, 123, 124

VRML Virtual Reality Modeling Language. 7

Bibliography

[1] A-Frame - Javascript web framework. https://aframe.io/. Accessed: 2019-01-21.

[2] CeMM - Jörg Menche Group. https://cemm.at/research/groups/joerg-menche-group/. Accessed: 2019-01-21.

[3] CeMM - Research Center for Molecular Medicine of the Austrian Academy of Science. http://cemm.at/. Accessed: 2019-01-21.

[4] Epic Games. https://www.epicgames.com/. Accessed: 2019-01-21.

[5] Git. https://git-scm.com/. Accessed: 2019-01-21.

[6] Git Large File Storage. https://git-lfs.github.com/. Accessed: 2019-01-21.

[7] High Fidelity. https://highfidelity.com/. Accessed: 2019-01-21.

[8] HTC Vive. https://www.vive.com/. Accessed: 2019-01-21.

[9] HTC Vive Tracker. https://www.vive.com/us/vive-tracker/. Accessed: 2019-01-21.

[10] i2cat - The Internet Research Center. http://www.i2cat.net/en/. Accessed: 2019-01-21.

[11] ISO/IEC 25010 - Non-functional requirement model. http://iso25000.com/index.php/en/iso-25000-standards/iso-25010. Accessed: 2019-01-21.

[12] Lennart Fahlén - DIVE online article. https://www.ercim.eu/publication/Ercim_News/enw31/fahlen2.html/. Accessed: 2019-01-21.

[13] OpenSimulator. http://opensimulator.org/. Accessed: 2019-01-21.

[14] Second Life. https://secondlife.com/. Accessed: 2019-01-21.

[15] Steamworks SDK. https://partner.steamgames.com/. Accessed: 2019-01-21.

[16] TU Wien - Interactive Media Systems Group. https://www.ims.tuwien.ac.at/. Accessed: 2019-01-21.

[17] Unity3D. https://unity3d.com/. Accessed: 2019-01-21.

[18] Unreal Engine. https://www.unrealengine.com/. Accessed: 2019-01-21.

[19] Unreal Engine blog: Unreal Property System. https://www.unrealengine.com/en-US/blog/unreal-property-system-reflection. Accessed: 2019-01-21.

[20] Unreal Engine documentation. https://docs.unrealengine.com/. Accessed: 2019-01-21.

[21] Unreal Engine Marketplace. https://www.unrealengine.com/marketplace/store. Accessed: 2019-01-21.

[22] VaRest Unreal Engine plugin. https://www.unrealengine.com/marketplace/varest-plugin. Accessed: 2019-01-21.

[23] Visual Paradigm. https://www.visual-paradigm.com/. Accessed: 2019-01-21.

[24] Visual Studio. https://visualstudio.microsoft.com/. Accessed: 2019-01-21.

[25] WebXR Device API. https://immersive-web.github.io/webxr/. Accessed: 2019-01-21.

[26] Zero Latency. https://zerolatencyvr.com/. Accessed: 2019-01-21.

[27] D. B. Anderson, J. W. Barrus, J. H. Howard, C. Rich, C. Shen, and R. C. Waters. Building multiuser interactive multimedia environments at merl. *IEEE MultiMedia*, 2(4):77–82, 1995.

[28] A.-L. Barabási, N. Gulbahce, and J. Loscalzo. Network medicine: a network-based approach to human disease. *Nature reviews genetics*, 12(1):56, 2011.

[29] Y. W. Bernier. Latency compensating methods in client/server in-game protocol design and optimization. In *Game Developers Conference*, volume 98033, 2001.

[30] S. Bryson. Virtual reality in scientific visualization. *Communications of the ACM*, 39(5):62–71, 1996.

[31] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. *Distributed systems*, 2:199–216, 1993.

[32] C. Carlsson and O. Hagsand. Dive a multi-user virtual reality system. In *Virtual Reality Annual International Symposium, 1993., 1993 IEEE*, pages 394–400. IEEE, 1993.

[33] C. Dibbern, M. Uhr, D. Krupke, and F. Steinicke. Can webvr further the adoption of virtual reality? *Mensch und Computer 2018-Usability Professionals*, 2018.

[34] C. Donalek, S. G. Djorgovski, A. Cioc, A. Wang, J. Zhang, E. Lawler, S. Yeh, A. Mahabal, M. Graham, A. Drake, et al. Immersive and collaborative data visualization using virtual reality platforms. In *Big Data, 2014 IEEE International Conference on*, pages 609–614. IEEE, 2014.

[35] E. Frécon and M. Stenius. Dive: A scaleable network architecture for distributed virtual environments. *Distributed Systems Engineering*, 5(3):91, 1998.

[36] P. García and A. Gómez-Skarmeta. Ants framework for cooperative work environments. *IEEE Computer*, 36(3):56–62, 2003.

[37] N. Ghrairi, S. Kpodjedo, A. Barrak, F. Petrillo, and F. Khomh. The state of practice on virtual reality (vr) applications: An exploratory study on github and stack overflow. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 356–366. IEEE, 2018.

[38] C. Greenhalgh, J. Purbrick, and D. Snowdon. Inside massive-3: Flexible support for data consistency and world structuring. In *Proceedings of the third international conference on Collaborative virtual environments*, pages 119–127. ACM, 2000.

[39] ISO. *ISO-IEC 25010: 2011 Systems and Software Engineering-Systems and Software Quality Requirements and Evaluation (SQuaRE)-System and Software Quality Models*. ISO, 2011.

[40] I. ISO. Iec 15288: 2008. systems and software engineering-system life cycle processes. *2nd International Organization for Standardization/International Electrotechnical Commission, ISO/IEEE/IEC*, 2008.

[41] J. Jacobson and M. Lewis. Game engine virtual reality with caveut. *Computer*, 38(4):79–82, 2005.

[42] H. Kaufmann, D. Schmalstieg, and M. Wagner. Construct3d: a virtual reality application for mathematics and geometry education. *Education and information technologies*, 5(4):263–276, 2000.

[43] G. Kotonya and I. Sommerville. *Requirements engineering: processes and techniques*. Wiley Publishing, 1998.

[44] R. Kuck, J. Wind, K. Riege, M. Bogen, and S. Birlinghoven. Improving the avango vr/ar framework: Lessons learned. In *Workshop Virtuelle und Erweiterte Realität*, pages 209–220, 2008.

[45] C. F. Lange, M. R. Chaudron, and J. Muskens. In practice: Uml software architecture and design description. *IEEE software*, 23(2):40–46, 2006.

[46] K.-C. Lin. Dead reckoning and distributed interactive simulation. In *Distributed Interactive Simulation Systems for Simulation and Training in the Aerospace Environment: A Critical Review*, volume 10280, page 4. International Society for Optics and Photonics, 1995.

[47] J. Menche, A. Sharma, M. Kitsak, S. D. Ghiassian, M. Vidal, J. Loscalzo, and A.-L. Barabási. Uncovering disease-disease relationships through the incomplete interactome. *Science*, 347(6224):1257601, 2015.

[48] V. Normand, C. Babski, S. Benford, A. Bullock, S. Carion, Y. Chrysanthou, N. Farcet, E. Frécon, J. Harvey, N. Kuijpers, et al. The coven project: Exploring applicative, technical, and usage dimensions of collaborative virtual environments. *Presence: teleoperators and virtual environments*, 8(2):218–236, 1999.

[49] J. Pečiva. *Active transactions in collaborative virtual environments*. PhD thesis, Doctoral dissertation. Brno University of Technology, Brno, Czech Republic, 2007.

[50] I. Podkosova, K. Vasylevska, C. Schoenauer, E. Vonach, P. Fikar, E. Bronederk, and H. Kaufmann. Immersivedeck: A large-scale wireless vr system for multiple users. In *SEARIS, 2016 IEEE 9th Workshop on*, pages 1–7. IEEE, 2016.

[51] S. Robertson and J. Robertson. *Mastering the requirements process: Getting requirements right*. Addison-wesley, 2012.

[52] I. Rodriguez and X. Wang. An empirical study of open source virtual reality software projects. In *ESEM, 2017 ACM/IEEE International Symposium on*, pages 474–475. IEEE, 2017.

[53] M. Rygol, S. Ghee, J. Naughton-Green, and J. Harvey. Technology for collaborative virtual environments. In *Proc. of Collaborative Virtual Environments*, volume 96, pages 19–20, 1996.

[54] S. Singhal and M. Zyda. *Networked virtual environments: design and implementation*. ACM Press/Addison-Wesley Publishing Co., 1999.

[55] Z. Szalavári, D. Schmalstieg, A. Fuhrmann, and M. Gervautz. "studierstube": An environment for collaboration in augmented reality. *Virtual Reality*, 3(1):37–48, 1998.

[56] A. S. Tanenbaum and M. Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[57] H. Tramberend. Avocado: A distributed virtual reality framework. In *Virtual Reality, 1999. Proceedings., IEEE*, pages 14–21. IEEE, 1999.

[58] D. Waller, E. Bachmann, E. Hodgson, and A. C. Beall. The hive: A huge immersive virtual environment for research in spatial cognition. *Behavior Research Methods*, 39(4):835–843, 2007.