Bachelor's thesis

Information and communications technology

2024

Mika Ahlsten

Real-Time Motion Capture and Streaming for Remote Collaboration

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Information and communications technology

2024 | 46 pages

Mika Ahlsten

Real-Time Motion Capture and Streaming for Remote Collaboration

According to *Harvard Business Review*, remote work surged from 6% to over 50% during the pandemic but has declined since early 2023, now stabilizing at around 28%. This indicates that remote work has become a permanent part of working life, even though its prevalence has decreased from its peak levels. However, in certain fields, such as performing arts, remote work opportunities are limited, creating challenges for job continuity and accessibility.

The thesis examined the benefits of remote work in the arts by focusing on the transmission of motion data over the internet for virtual performances. A system was developed in which an animated avatar in Unity generated motion data, which was sent to a server and received by another application. The study analyzed the limitations of motion data transfer, such as latency and data integrity, and explored ways to optimize these factors. The result was a prototype that demonstrated the challenges and possibilities of real-time motion data transmission for virtual performances.

The solution is built with Unity and Node.js, incorporating motion detection, VR glasses, and a server-based interface to transmit movement data, speech, and control signals. A custom measurement tool and third-party software help assess challenges.

The results indicate that avatar-based live performances are technically feasible and could offer new possibilities for the performing arts. Based on the observed challenges and opportunities, further research would be warranted.

Keywords:

motion capture, node.js, unity, c#, server architecture, animation

Opinnäytetyö (AMK) | Tiivistelmä

Turun ammattikorkeakoulu

Tieto- ja viestintätekniikka

2024 | 46 sivua

Mika Ahlsten

Reaaliaikainen liikkeenkaappaus ja suoratoisto etäyhteistyöhön

Harvard Business Review'in mukaan etätyö kasvoi pandemian aikana 6 % – yli 50 %:iin, mutta on laskenut vuoden 2023 alusta lähtien ja vakiintunut noin 28 %:iin. Tämä kertoo siitä, että etätyö on jäänyt pysyväksi osaksi työelämää, vaikka sen määrä onkin vähentynyt huippulukemista. Tietyillä aloilla, kuten esittävässä taiteessa, etätyömahdollisuudet ovat kuitenkin rajalliset, mikä luo haasteita esimerkiksi työn jatkuvuudelle ja saavutettavuudelle.

Opinnäytetyössä tutkittiin etätyön hyötyjä taidealalla keskittymällä liikedatan välittämiseen internetin yli virtuaaliesityksiä varten. Työssä kehitettiin järjestelmä, jossa Unityssä animoitu avatar tuotti liikedataa, joka lähetettiin palvelimelle ja vastaanotettiin toisessa ohjelmassa. Tutkimuksessa analysoitiin liikedatan siirtoon liittyviä rajoituksia, kuten viivettä ja tiedon eheyttä, sekä pyrittiin löytämään keinoja niiden optimointiin. Tuloksena syntyi prototyyppi, joka osoitti, millaisia haasteita ja mahdollisuuksia liikedatan reaaliaikainen siirtäminen tuo virtuaalisiin esityksiin.

Ratkaisu hyödyntää Unitya ja Node.js:ää sekä liikkeentunnistusta, VR-laseja ja palvelinpohjaista rajapintaa, joka siirtää liike-, puhe- ja ohjausdataa. Haasteiden arviointiin käytettin räätälöityä mittaustyökalua ja kolmannen osapuolen ohjelmistoja.

Tulokset osoittavat, että avatar-pohjaiset live-esitykset ovat teknisesti toteutettavissa ja voivat tarjota uusia mahdollisuuksia esittävälle taiteelle. Havaittujen haasteiden ja mahdollisuuksien perusteella jatkotutkimus olisi perusteltua.

Asiasanat:

liikkeentunnistus, node.js, c#, unity, palvelinarkkitehtuuri, animaatio

Content

List of abbreviations (or) symbols 7

1 Introduction 8

2 Motion Capture and Theatre Platform 10

2.1 Introducing Theatre Program 11

2.2 General Structure 12

2.2.1 StageServer 12

2.2.2 VRClient 13

2.2.3 Node.js Server 13

2.2.4 Front-end User Interface 13

2.2.5 Voice Server 14

3 Explanation of the Structure and Functionality of the Theatre Platform 15

3.1 StageServer 15

3.2 VRClient 16

3.3 Node.js Server 17

3.4 Front-end control room 18

The control room is a web-based interface built using Node.js and running on port 80. It utilizes HTTP protocol for communication with the server and a separate websocket port (2999) to differentiate from the VRClient's websocket port (3000). This separation allows for greater control over the overall system architecture. 18

3.5 Voice Server Integration 18

4 User Interface and System Usage 20

4.1 StageServer User Interface 20

4.2 VRClient User Interface 22

4.3 Node.js System Usage 24

4.4 Front-end usage 24

5 Animations – Problems and solutions 26

5.1 Animation system in general 26
5.2 Problems 27
5.3 Solutions 27

6 Transfer speed Analysis 28
6.1 Research protocol 28
6.2 Data 29
6.3 Data Analysis 32

7 Discussion 36
7.1 Discussion 36
7.2 Avatar Transmission Limits 36
7.3 Precision and Optimization 37
7.4 Data Storage and Analysis 37
7.5 Future Considerations 38

8 Conclusion 39

References 41

Appendices

Appendix 1. Code Listing for SendData() Function

Figures

Figure 1: Datawrapper for storing animation data 15
Figure 2 Example control page containing possibility to open voice server connection 18
Figure 3 Stageserver Datasender script in Unity 20
Figure 4 Unity playfield with multiple character and different animations 21
Figure 5 Example of the .csv file 22

Figure 6 DataReceiver component in VRClient 23
Figure 7 Example of debug text fields 23
Figure 8 Node server starting 24
Figure 9 Node.js Control Room example 25
Figure 10 Node.js Express plugin usage 25
Figure 11 StageServer transfer speed at different FPS 29
Figure 12 VRClient transfer speeds at different FPS 30
Picture 13 Speed ratios on 30 FPS 31
Figure 14 Speed ratios on 60 FPS 31
Figure 15 Limitless speed ratio 32
Figure 16 Long-term dataset further analysis 33
Figure 17 Further analysis of 30 FPS dataset 34
Figure 18 further analysis of 60 FPS dataset 35

List of abbreviations (or) symbols

3D	3 Dimensional
API	Application Programming Interface
C#	C-sharp programming language
COVID-19	Coronavirus Disease 2019
CSV	Comma Separated Values
FPS	Frames Per Second
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JSON	Javascript Object Notation
RGB-D	Red, Green, Blue plus Depth Data
SDK	Software Development Kit
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VoIP	Voice over Internet Protocol

1 Introduction

In the year 2020, when the COVID-19 pandemic began to impose restrictions on societal activities, many artist in cultural field found themselves in dire straits. The majority of them were freelancers in the field of theater, and their sources of income came to a sudden halt. Witnessing artists making every effort to sustain their livelihood, many ventured into remote performances. However, this experience was by no means equivalent to the act of being physically present in the theater. Thus, the distinction between live and recorded presentations seems almost negligible.

This sparked the idea of how to construct an experience that could provide the most authentic sensation, both in real-time and remotely. In the overarching concept of the performance, it would unfold as follows:

Actors would use a Motion Capture suit or a similar system specifically designed to transfer their movements into the virtual realm, projecting them onto their respective avatars. This motion would be streamed live to a server, to which spectators would connect via their own virtual reality headsets. Given that the theatrical experience is interactive, actors would also need the ability to see their audience and hear their reactions.

In this context, there are several intriguing possibilities. Firstly, why a viewer would choose to attend remotely in a virtual world when, in today's world, they could attend a physical theater. The added value would derive from the virtual realm itself, which allows for the realization of elements unattainable under normal circumstances.

The primary focus of this research is to identify bottlenecks in creating a seamless performance and to explore suggestions for improvement. This requires a software stack to be created, where movement data is sent to a server and then picked up to another software which then moves an avatar in real-time.

Research will measure transfer speeds in different settings and how much that affects the movement of the avatar.

As secondary goals, there will be an exploration on feasibility in commercial use and feasible ways to save longer animation data for later use.

An integral part of this work involves striving to achieve an experience as close to the real thing as possible, focus of the research is on a technique that drives the most important part of the performance, which is to find out the limitations on data sent over the network.

2 Motion Capture and Theatre Platform

This thesis introduces a program that utilizes motion capture technology. The system is designed to be adaptable to various motion capture techniques, necessitating an examination of different categories within this field (Yang, 2024;Sabatier & Ekimov, 2006;Sandilands, Geol-Choi & Komura,2012). Motion capture techniques can be broadly classified into the following categories:

1. **Optical Motion Capture:**
    - **Passive Optical Systems:** This type uses reflective markers placed on the subject's body or object. Cameras track the movement of these markers to create a 3D representation of motion. Systems like Vicon and OptiTrack fall into this category.
    - **Active Optical Systems:** In this case, markers are equipped with their own light source, and the cameras detect the markers by capturing the emitted light. This method can work well in environments with varying lighting conditions.
2. **Inertial Motion Capture:**
    - **IMU Sensors (Inertial Measurement Units):** Inertial motion capture systems use accelerometers and gyroscopes to measure acceleration and angular velocity. These sensors are often attached to different body parts or objects. The data from multiple sensors are then combined to reconstruct motion in 3D space. Inertial motion capture is portable and doesn't require a large dedicated space.
3. **Mechanical Motion Capture:**
    - **Exoskeletons and Mechanical Sensors:** These systems use mechanical devices or exoskeletons attached to the subject's body to capture movement. The sensors on the mechanical devices record joint angles and movements. This type of motion capture is often used in biomechanics research.
4. **Magnetic Motion Capture:**
    - **Magnetic Sensors:** Magnetic motion capture systems use magnetic fields to track the position and orientation of sensors placed on the subject. Polhemus and Ascension are examples of magnetic motion capture systems. Magnetic systems can work well in situations where

optical systems might have limitations, such as in the presence of occlusions.

5. **Acoustic Motion Capture:**
   - **Ultrasonic Sensors:** Acoustic motion capture uses ultrasonic sensors to triangulate the position of markers on the subject. These systems emit ultrasonic pulses, and the time it takes for the signals to reach the markers and return is used to calculate their position. This type of motion capture is less affected by environmental lighting conditions.
6. **Depth Sensing (RGB-D) Motion Capture:**
   - **Depth Cameras:** Depth sensing technology, often utilizing infrared light, captures the spatial information of objects in a scene. Modern depth-sensing systems are widely used in applications such as motion capture, robotics, and augmented reality. Examples include Intel RealSense, Azure Kinect, and other 3D imaging solutions.

The prototype aims to capture the dynamic movements of a dancer. To achieve this, various motion capture methods were evaluated based on their suitability for fast-paced movements and the ability to avoid occlusions.

Optical systems, which rely on a constant line of sight to markers, are not ideal for this application, especially for dynamic dance movements (Yang, 2024). Mechanical systems, while capable of providing accurate measurements, can be impractical for dancers due to the requirement for exoskeletons and their limitations in capturing subtle movements (Rahul, 2018). Acoustic and depth-sensing systems may struggle in noisy or cluttered environments, and magnetic systems can be expensive and sensitive to electromagnetic interference.

Inertial motion capture emerges as the most suitable choice for this prototype due to its portability, accuracy, and ability to handle occlusions, which are critical factors for capturing dynamic dance movements.

2.1 Introducing Theatre Program

This thesis introduces a novel software platform, the Theatre Program or Theatre software stack, designed to facilitate the creation of real-time, remote theatre plays and similar presentations in a virtual space. The platform leverages three core components:

the StageServer, a backend server responsible for managing the virtual stage and coordinating interactions; the VRClient, a virtual reality client that allows users to immerse themselves in the virtual environment; and the Node.js Server, a middleware server that facilitates communication between the StageServer and VRClient.

The Theatre Program seamlessly integrates these components to create a unique and immersive live performance experience. By transmitting movement data to a server, processing it, and replicating the movements onto a remote avatar, the system enables real-time, remote performances. To further enhance the interactive nature of these performances, the integration of TeamSpeak provides robust voice communication capabilities, facilitating audience interaction and collaboration among performers.

To address the potential strain on internet traffic, careful consideration must be given to the selection of appropriate data transfer protocols. UDP protocol can be employed for efficient data transmission, while websocket communication can provide controlled reception. This combination ensures rapid data delivery and managed reception. (Ruohisto,2016;Luomala,2020).

In essence, the Theatre Program serves as a platform for streaming avatar movement data between machines, offering a new avenue for creating innovative and interactive virtual performances.To further enhance the interactive nature of these performances, the integration of TeamSpeak provides robust voice communication capabilities, facilitating audience interaction and collaboration among performers(TeamSpeak, 2024)

2.2 General Structure

2.2.1 StageServer

The StageServer, developed in Unity Engine, incorporates the motion capture SDK and necessary C# scripts. This enables the capture of movement data from the avatar, which is controlled by the motion capture system, and prepares the data for transmission to the server (Unity Technologies, 2024).

2.2.2 VRClient

The VRClient, also created in Unity Engine, operates independently from the StageServer. It functions as the recipient of movement data transmitted from the StageServer. The VRClient processes this data and applies it to the avatar within the virtual environment. As the end-user interface, the VRClient is designed to provide a comprehensive experience, including graphics, ambient sounds, and all elements essential for the user's immersion in the virtual world (Unity Technologies, 2024).

2.2.3 Node.js Server

The Node.js server is a web server constructed using Node.js, a JavaScript runtime. Node.js is renowned for its lightweight, efficient, and scalable nature, making it well-suited for developing server-side applications, including web servers (Wexler, 2019).

Key concepts related to a Node.js server include:

- Node.js: A JavaScript runtime built on the V8 JavaScript engine.
- Server: A computer or program that manages network resources and responds to requests.
- HTTP Module: A built-in module for creating HTTP servers.
- Event-Driven: Handles asynchronous operations efficiently.
- Callback Functions: Used for handling asynchronous operations.

The Node.js server was selected to manage the traffic between StageServer and VRClient due to its asynchronous nature and the need for a lightweight, high-performance solution. The anticipated volume of streaming data further supports this choice.

2.2.4 Front-end User Interface

Given the role of the Node.js server in managing traffic between StageServer and VRClient, the development of a control station front-end is feasible. This front-end could facilitate control over the origin and destination of movement data, as well as access

permissions. Additionally, Node.js libraries offer built-in statistics that can be leveraged to gather performance data for the server (Wexler, 2019).

2.2.5 Voice Server

TeamSpeak is a proprietary Voice over IP (VoIP) application widely used in the gaming community for real-time voice communication. It offers a platform for creating and managing voice communication servers, enabling users to connect and participate in voice conversations (TeamSpeak, 2024).

Key features of TeamSpeak include:

- Client-Server Architecture: TeamSpeak follows a client-server model, requiring users to install the client software and connect to a server.
- Channels and Permissions: Servers organize communication into channels, and administrators can control user permissions.
- Low Latency: TeamSpeak is known for its low-latency communication, making it suitable for real-time interactions.
- High-Quality Audio: TeamSpeak supports high-quality audio codecs and allows for customization of audio settings.
- Cross-Platform Compatibility: TeamSpeak clients are available for various platforms, ensuring accessibility.
- Security: TeamSpeak offers encryption and password protection to enhance security.
- File Sharing: Users can share files within the platform.
- Add-ons and Plugins: TeamSpeak supports customization through add-ons and plugins.

TeamSpeak operates on a client-server licensing model, requiring server licenses for hosting. It is frequently chosen for its stability, reliability, and efficiency in providing voice communication for gaming communities and other groups requiring real-time interaction (TeamSpeak, 2024).

3 Explanation of the Structure and Functionality of the Theatre Platform

3.1 StageServer

The StageServer captures avatar movements by recording Vector3 positions and rotations of specific joints within the avatar rig. The prototype utilizes 18 points of interest, capturing positions and rotations for each joint. These vectors are subsequently encoded into a JSON string. Given the nature of the data transmission and the tolerance for occasional packet loss, UDP was selected as the sending protocol. UDP's efficiency in streaming data to upstream aligns well with the requirements of this system (Ruohisto, 2016;Luomala, 2020).

The Datawrapper (Figure 1) serves as the foundation for data collected from the avatar. A coroutine function (Appendix 1) is essential for gathering, sending, and controlling the volume of data transmitted over the internet. A control timer will regulate data transmission at specified intervals.

```
private class SkeletonDataWrapper
{
    public int playerCount;
    public Vector3 worldPosition;
    public Vector3[] skeletonDataPos;
    public Vector3[] skeletonDataRot;
    public string playerName;

}
```

Figure 1: Datawrapper for storing animation data

UDP, or User Datagram Protocol, is a fundamental protocol within the Internet Protocol (IP) suite (Ruohisto, 2016). It operates at the transport layer, providing a connectionless and lightweight communication service. Unlike TCP, UDP does not establish a reliable, connection-oriented channel before data transmission. Instead, it sends data in discrete packets, known as datagrams, without guaranteeing delivery or order.

Key characteristics of UDP include:

- Connectionless: UDP does not require a connection before sending data.
- Unreliable: UDP does not guarantee data delivery or order.
- Low Overhead: UDP is faster and more suitable for applications requiring low latency.
- Broadcast and Multicast Support: UDP supports sending data to multiple recipients.
- Simple Header: The UDP header is relatively simple.
- Usage Scenarios: UDP is often used for real-time communication, such as streaming media and online gaming.

While UDP is suitable for certain applications, it may not be the best choice for scenarios requiring guaranteed delivery and ordered data transmission. Developers should carefully assess the specific application requirements to select the appropriate transport protocol.

3.2 VRClient

While VRClient shares a similar structure with StageServer in Unity's C# code, the communication protocol differs. VRClient utilizes WebSocket, a full-duplex protocol enabling real-time data transfer over a single, persistent connection (Luomala, 2020). This contrasts with traditional request-response protocols like HTTP.

Key characteristics of WebSocket include:

- Full-Duplex Communication: Both client and server can send messages independently.
- Persistent Connection: A continuous connection is established, reducing latency.
- Low Latency: WebSocket is suitable for real-time applications requiring low-latency communication.
- Efficient Use of Resources: WebSocket reduces overhead compared to traditional methods.
- WebSocket Handshake: A handshake establishes the WebSocket connection.
- WebSocket API: Developers can use the WebSocket API in web applications.

WebSocket is valuable for applications requiring real-time updates, live notifications, or collaborative features. It is commonly used for building interactive and dynamic web applications.

By using WebSocket, the VRClient can exercise greater control over the data received from StageServer. While StageServer primarily transmits position and rotation data, WebSocket allows for the selection of specific data to be used on the user side of the application stack. Additionally, advanced features of the Node.js server, such as user authentication, can be implemented to enhance security and prevent unauthorized access.

However, incorporating additional information, such as multiple avatars or character-specific details, into the data stream can potentially increase the data volume, potentially impacting efficiency.

3.3 Node.js Server

Node.js server is, in its prototype phase, very simple. It will receive JSON strings from StageServer and parse the string. Parsing is not necessary for the prototype but it will give more control over it. When parsing the JSON string in node server there is a possibility to add commands from the stageserver for the node server, if they are embedded in the JSON string. Avatar personalization data is needed to separate one avatar from another. This means that separation of multiple avatars from the stream can be achieved. Websocket server is running also on different port to avoid confusion with the webtraffic in control room that runs in the node.js server.This control room runs in the webserver itself and connects to another websocket port. This websocket connection is used by VRClient and it will listen to requests from VRClient.

3.4 Front-end control room

The control room is a web-based interface built using Node.js and running on port 80. It utilizes HTTP protocol for communication with the server and a separate websocket port (2999) to differentiate from the VRClient's websocket port (3000). This separation allows for greater control over the overall system architecture.

3.5 Voice Server Integration

TeamSpeak is a third-party VoIP application that operates on a client-server architecture, providing voice communication capabilities (TeamSpeak, 2024). The TeamSpeak server manages communication between clients, creates voice channels, controls user permissions, and facilitates the exchange of voice and text data.

To ensure efficient voice communication, TeamSpeak employs compression techniques and optimized voice transmission protocols. It also incorporates security measures to protect communication, offers text-based communication in addition to voice, and enables file sharing between users.

The TeamSpeak SDK provides developers with programmatic access to TeamSpeak servers. The Client API allows developers to send commands to TeamSpeak clients remotely, such as managing channels, interacting with users, and querying server information. Communication between the client and server is established using a TCP connection, adding an additional protocol to the program stack.

The Server API enables developers to perform administrative tasks, manage users and channels, and gather information about the server. For the prototype, a basic webpage can be created for testing purposes (Figure 2), and the server can be connected programmatically upon startup.

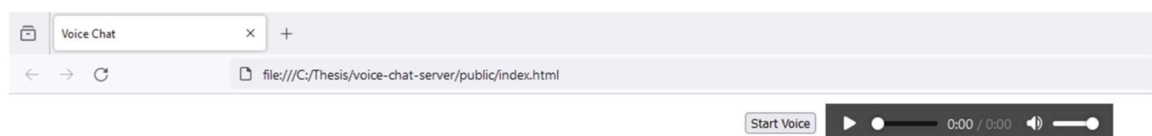


Figure 2 Example control page containing possibility to open voice server connection

By implementing this approach, a remote control webpage can be created for managing the server, allowing for convenient control from a distance. While transfer speed and data sent/received by TeamSpeak are not explicitly calculated in this prototype, TeamSpeak's settings for maximum voice quality can be used as a proxy for data transfer.

4 User Interface and System Usage

4.1 StageServer User Interface

The ServerURL variable stores the address of the Node.js server. In this prototype, the server is running locally, so the ServerURL will be set to "localhost" (Figure 3). The debugspeed test variable will display the actual data transmission speed.

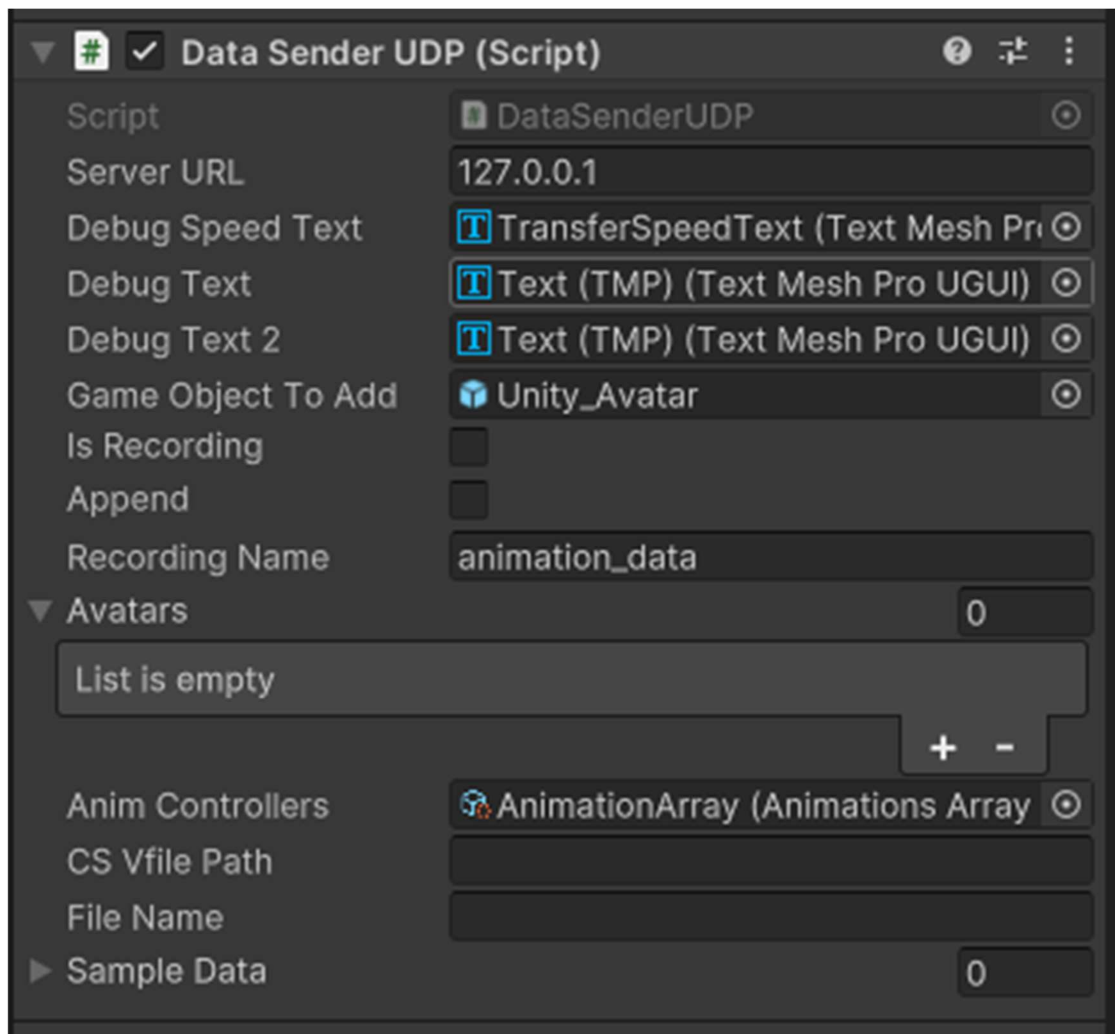


Figure 3 Stageserver Datasender script in Unity

To evaluate network traffic under stress conditions, multiple avatars can be added to the scene. Creating a prefab of the avatar can streamline this process, allowing for easy instantiation of additional avatars.

The system also includes the capability to save recorded data. A boolean variable, isRecording, can be used to trigger the recording of JSON strings to a file. The recording can be configured to append data to an existing file or create a new file based on the specified recording name (Unity Technologies, 2024).

A ScriptableObject was created to store a collection of different animation clips (Figure 4). By utilizing a ScriptableObject, we can centralize and manage these animations efficiently (Unity Technologies, 2024). To ensure variation between avatars, each instantiated avatar is assigned a randomly selected animation clip from the ScriptableObject. This approach guarantees unique movement patterns for each character, enhancing the diversity and unpredictability of the software's behavior.

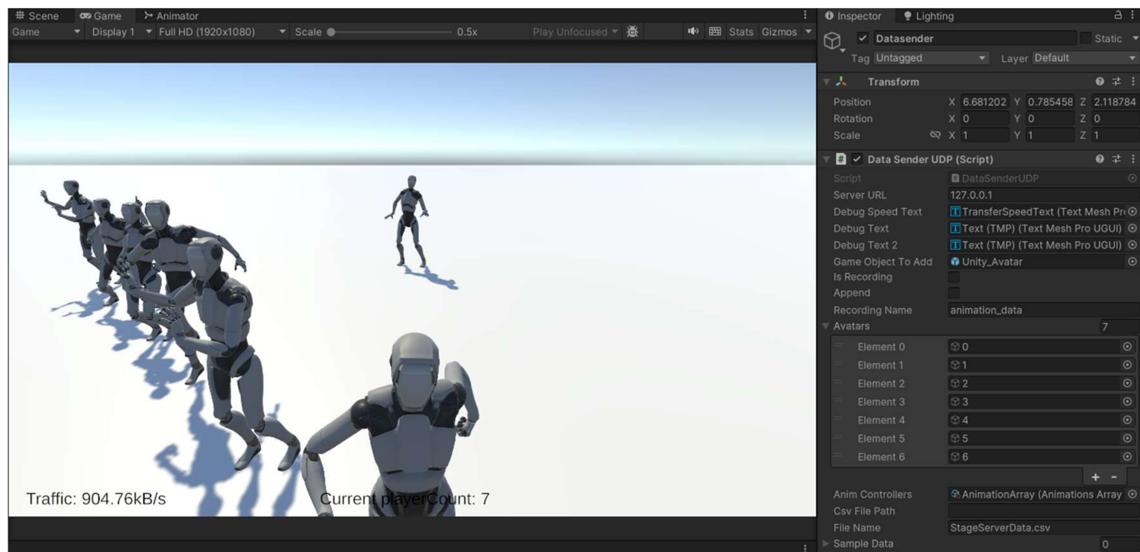


Figure 4 Unity playfield with multiple character and different animations

The StageServer is configured to automatically create a new character every 5 seconds, simulating a growing number of participants in the virtual environment. To monitor system performance and network traffic, the StageServer records the traffic volume, the current number of players in the field, and a timestamp. This data is then logged to a CSV file for analysis (Figure 5).

	A	B	C	D
1	Time,Character Count,speed			
2	5,1,73.29219			
3	10,3,148.074			
4	15,4,220.3613			
5	20,5,295.851			
6	25,6,364.5225			
7	30,7,441.0488			
8	35,8,518.1637			
9	40,9,584.0873			
10	45,10,657.0674			
11	50,11,725.4082			

Figure 5 Example of the .csv file

4.2 VRClient User Interface

VRClient uses similar methods. The ServerURL variable specifies the address of the Node.js server, and a port number must be defined (Figure 6) to establish the websocket connection for data retrieval. To introduce new players into the virtual environment, an avatar prefab should be instantiated each time a new player joins. Additionally, a

notification should be displayed to inform other players of the new arrival (Unity Technologies, 2024).

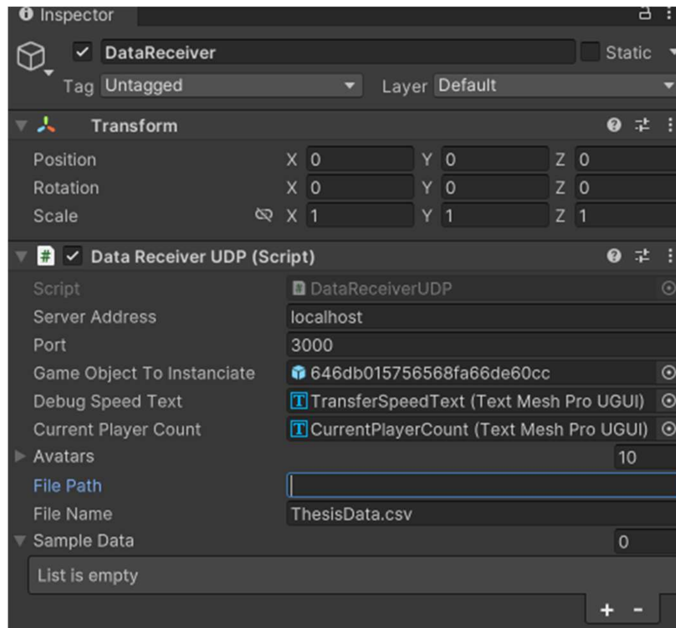


Figure 6 DataReceiver component in VRClient

The debug text fields (Figure 7) display real-time information regarding traffic speed and the number of players currently in the field. Upon launching the software, it establishes a connection to the Node.js server and continuously monitors for player activity. The traffic amount, player count, and timestamp are recorded in a CSV file for subsequent analysis.

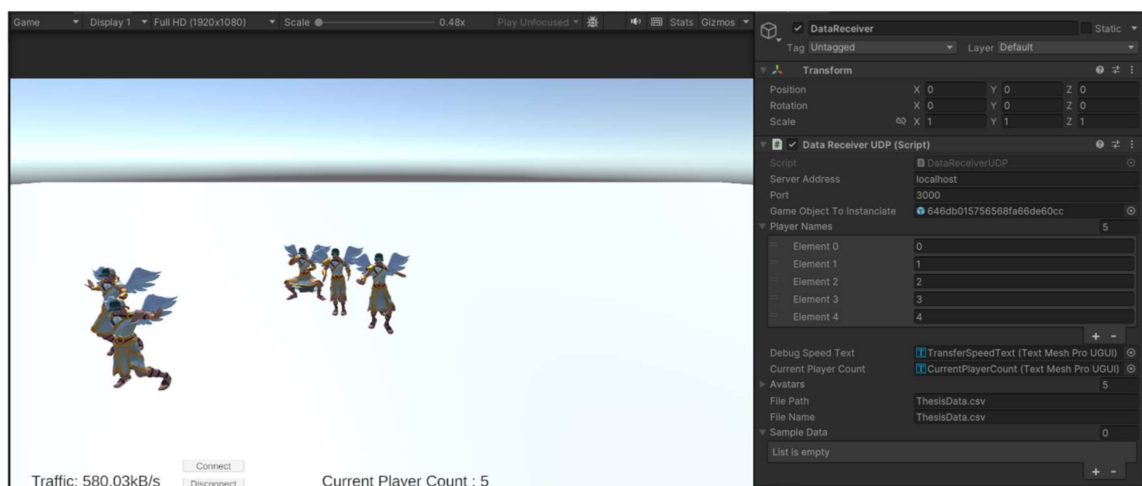


Figure 7 Example of debug text fields

4.3 Node.js System Usage

Upon startup, the Node.js (Figure 8) server will handle network traffic and listen for websocket connections on ports 2999 and 3000. Additionally, the server will monitor UDP traffic on port 7777 and TCP/IP connections on port 80 (Smith, 2023).

To inform the StageServer and VRClient of the server's address, a notification mechanism is implemented. This allows the client applications to establish connections and communicate with the server.

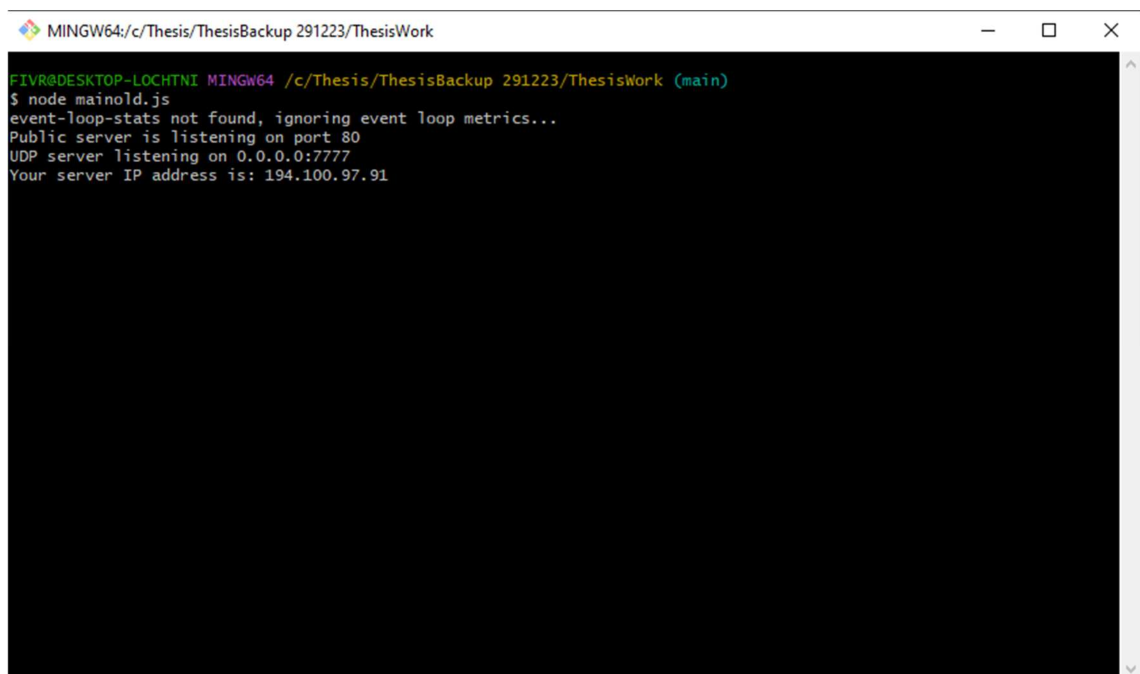


Figure 8 Node server starting

4.4 Front-end usage

The control room front-end (Figure 9) serves as a basic monitoring tool for the virtual environment (Wexler, 2019). It primarily functions as a ping tool to verify the integrity of the websocket connection. Additionally, the front-end displays the current number of players in the field and receives messages from the server.

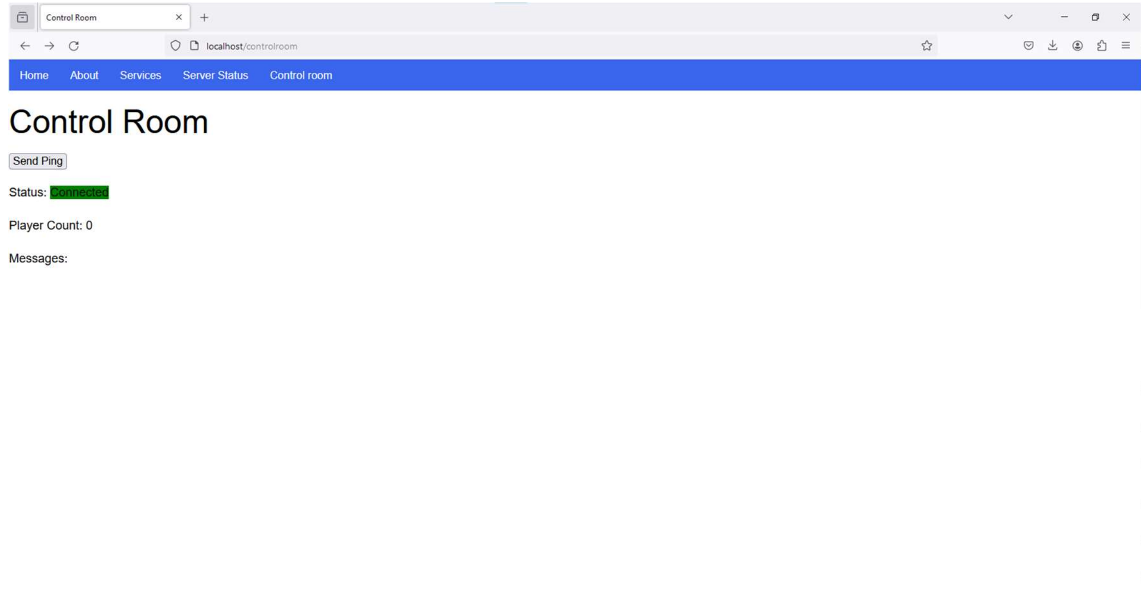


Figure 9 Node.js Control Room example

The received messages currently include notifications about new player arrivals, along with the name of the newly joined player.

The Node.js Express plugin (Figure 10) is utilized to monitor the server's status and performance. This plugin provides valuable statistics that offer insights into the overall server load and health (Express.js, 2024).

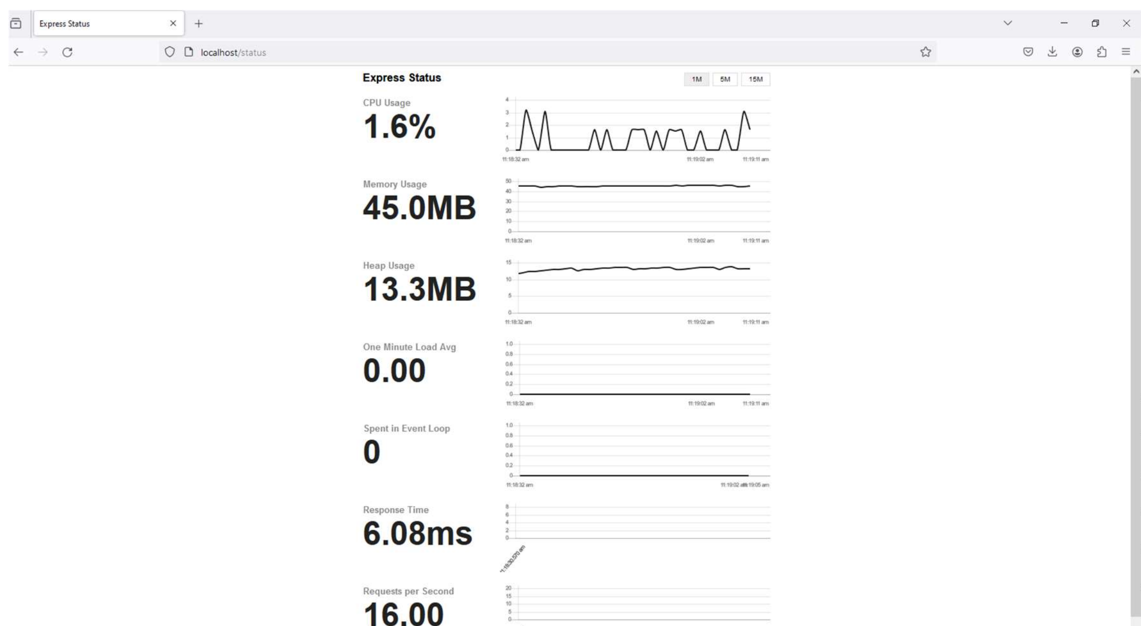


Figure 10 Node.js Express plugin usage

5 Animations – Problems and solutions

5.1 Animation system in general

Unity's animation system is a powerful tool that enables developers to create dynamic and interactive animations for characters, objects, and other elements within a game or application. As an integral part of the Unity game development engine, this system is designed to be user-friendly while offering a robust set of features. (Unity Technologies, 2024)

Key Components and Features:

- **Animator Component:** Controls the playback of animations for a GameObject.
- **Animator Controller:** Manages animations for a specific GameObject, allowing for visual design and organization.
- **Animation Clips:** Individual animation assets defining the motion of a GameObject.
- **States and Transitions:** Organize animations using states and transitions.
- **Blend Trees:** Create smooth transitions between multiple animations.
- **Parameters:** Variables that drive transitions or affect animation behavior.
- **Scripting Integration:** Control the animation system through scripts.
- **Mecanim:** Unity's animation technology.
- **Animation Events:** Trigger actions at specific points in an animation timeline.
- **Root Motion:** Animate the entire GameObject, including position and rotation.

Additional Considerations:

- **ScriptableObject:** Use ScriptableObjects to centralize and manage animations.
- **Randomization:** Randomly assign animations to avatars for variety.
- **Character Creation:** Add new characters at regular intervals.
- **Data Logging:** Record traffic, player count, and timestamps for analysis.
- **Control Room:** Implement a control room for monitoring and management.
- **Node.js Server:** Configure the server address and ports for websocket and other protocols.
- **TeamSpeak Integration:** Integrate TeamSpeak for voice communication.

- **Data Transfer:** Use appropriate protocols (e.g., UDP, WebSocket) for efficient data transmission.

5.2 Problems

While the StageServer leverages the full capabilities of Unity's animation system, the VRClient encounters limitations due to the remote nature of the animation control. The VRClient requires an Animator component to access the root motion system, but cannot utilize an Animator Controller. The remote control of animations overrides the local Animator Controller, preventing access to certain features such as State Machines and associated parameters (Unity Technologies, 2024).

5.3 Solutions

Root motion is a technique employed in animation to control a character's movement by animating its root bone, typically the hip or pelvis bone, and applying that motion to the entire character (Unity Technologies, 2024). In this project, root motion will be utilized to manipulate the character's movement by modifying the root bone's position and rotation through JSON strings.

The JSON strings, transmitted from the server, contain a single position and rotation value for all root bones. These strings are sent at the maximum rate supported by the system. Theoretically, transmitting 30 strings per second should align with a frame rate of 30 frames per second.

6 Transfer speed Analysis

6.1 Research protocol

The StageServer software calculates data transmission rates by converting JSON strings into byte arrays and measuring the length of the resulting data. This provides an estimate of the number of bytes sent per transmission cycle. By multiplying this value by 1024, the data transfer rate can be expressed in kilobytes per second (KB/s). This data is recorded at regular intervals to track changes over time.

A similar method is used on the VRClient software to calculate the amount of data received from the server, allowing for an assessment of the difference between the sending and receiving parties.

To evaluate the system's performance under stress, new characters are added to the scene every 5 seconds. This stress test helps determine the maximum data transfer capacity and identify any limitations. The VRClient receives information from the server regarding the number of players in the game field, and each character's movement data is transmitted accordingly.

Each JSON string has a specific length (X) and is sent through the system at predetermined intervals. By varying these intervals during testing, it is possible to identify the system's performance thresholds.

The system stack is configured for optimal data transfer within the local area network (LAN), ensuring the fastest possible transmission speeds. However, it is important to consider potential limitations within Unity itself that might affect the stress test. By introducing a sufficient number of characters into the playfield, it is possible to assess whether the animation performance degrades, which could impact the amount of data sent to the server.

Both the StageServer and VRClient implement a method to create a spreadsheet in CSV format. This spreadsheet records transfer rates, player count, and timestamps, enabling comparisons between different simulation scenarios.

In summary, the simulation adds a player to the field every 5 seconds, records transfer rates, and generates spreadsheets for data analysis. This approach allows for evaluating system performance under varying conditions.

.

6.2 Data

Two tests were conducted with limited frame rates of 30 FPS and 60 FPS, while the third test was performed without any software limitations. The first graph (Figure 11) illustrates the amount of data that StageServer can send under different settings. VRClient data is recorded for comparison. (Figure 12)

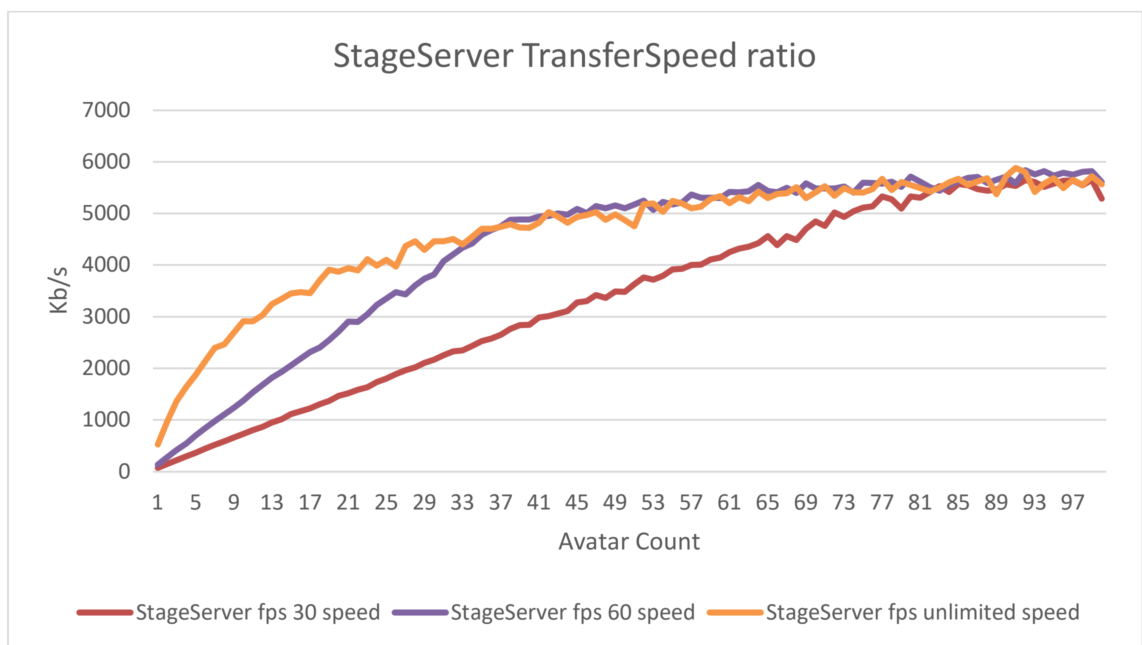


Figure 11 StageServer transfer speed at different FPS

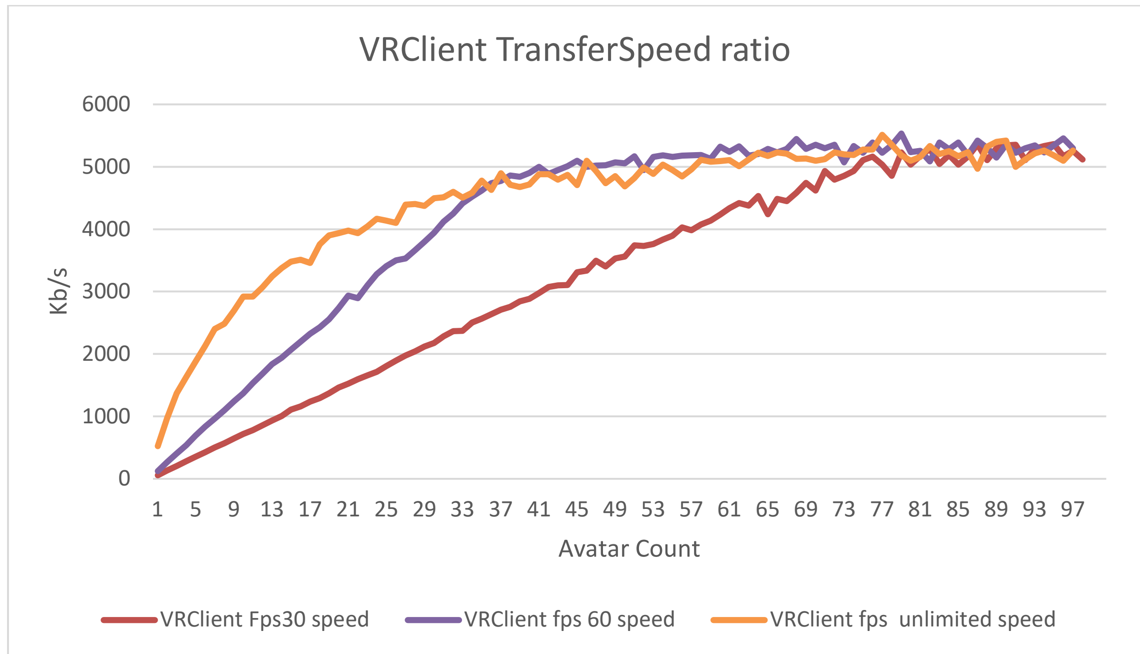
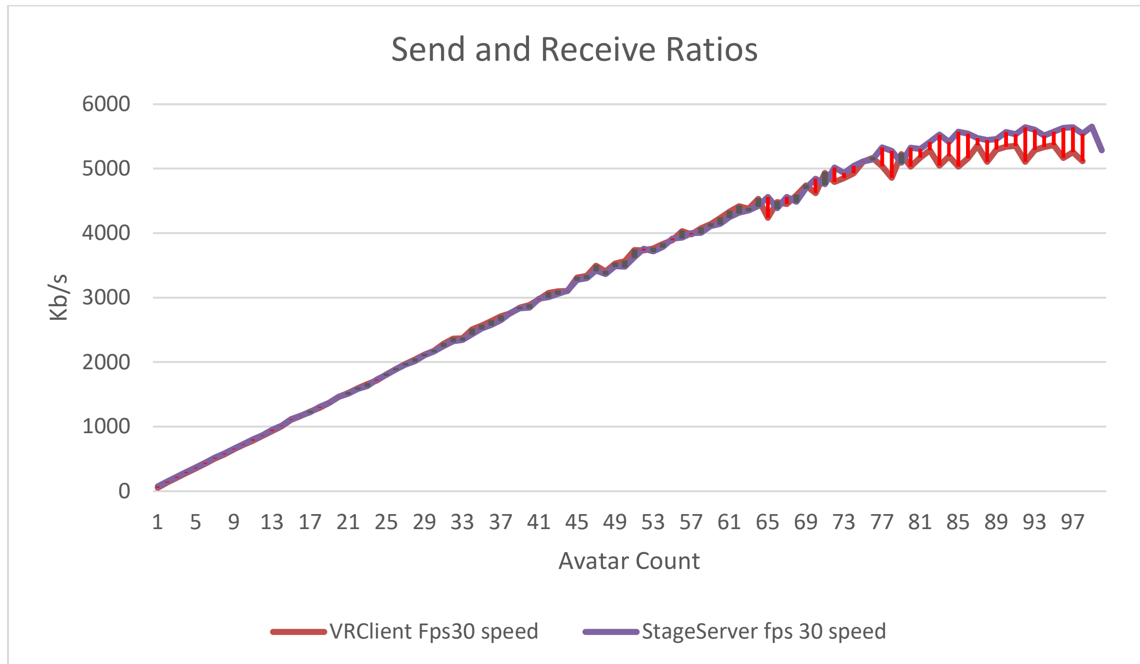


Figure 12 VRClient transfer speeds at different FPS

The data recorded from StageServer compares the amount of data sent by StageServer to the amount received by VRClient under different frame rate limitations (30 FPS and 60 FPS). The red horizontal lines in the graphs (Figure 13 and Figure 14) indicate when VRClient begins to experience data loss compared to StageServer.



Picture 13 Speed ratios on 30 FPS

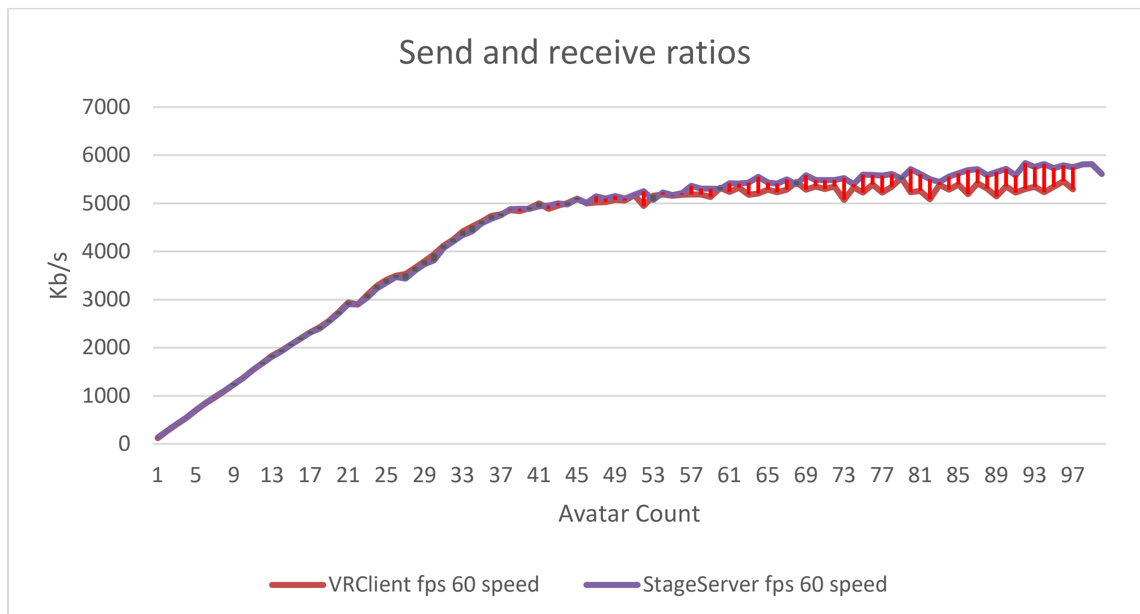


Figure 14 Speed ratios on 60 FPS

The final dataset (Figure 15) was collected using both StageServer and VRClient in an unlimited setting, with a significantly longer simulation runtime compared to the previous tests. In this scenario, Unity demonstrated its maximum data transmission capacity.

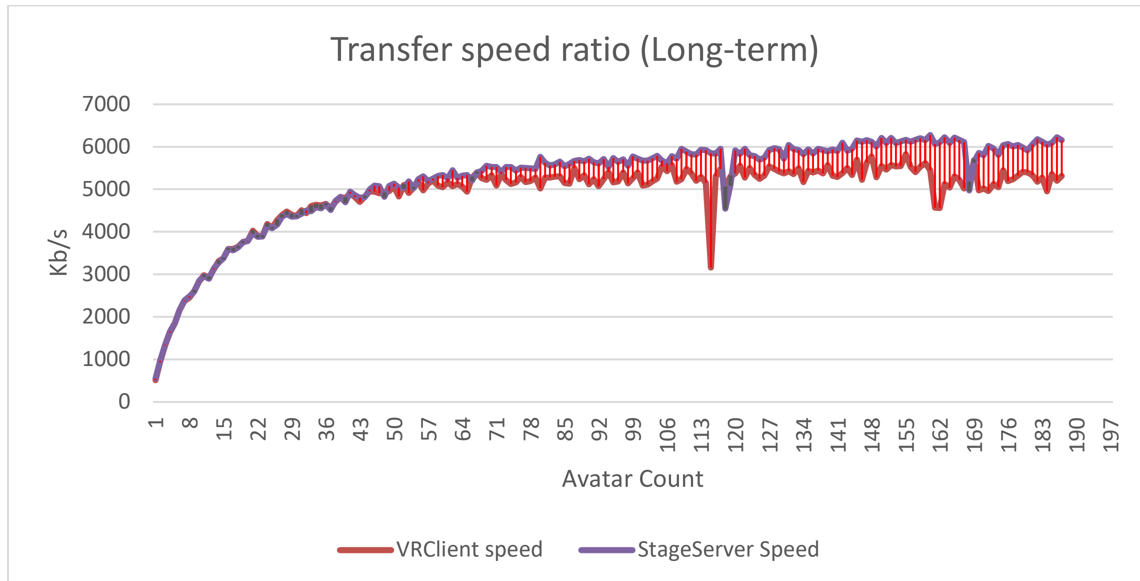


Figure 15 Limitless speed ratio

6.3 Data Analysis

Gathered data shows the limits on sending and receiving data on moving avatars in a Unity default setting where Unity is sending data on every frame it can.

Long-term data (Figure 16) suggests that everything will work normally, until 44-45 characters are on the field moving. At this threshold, VRClient is losing data from Stageserver.

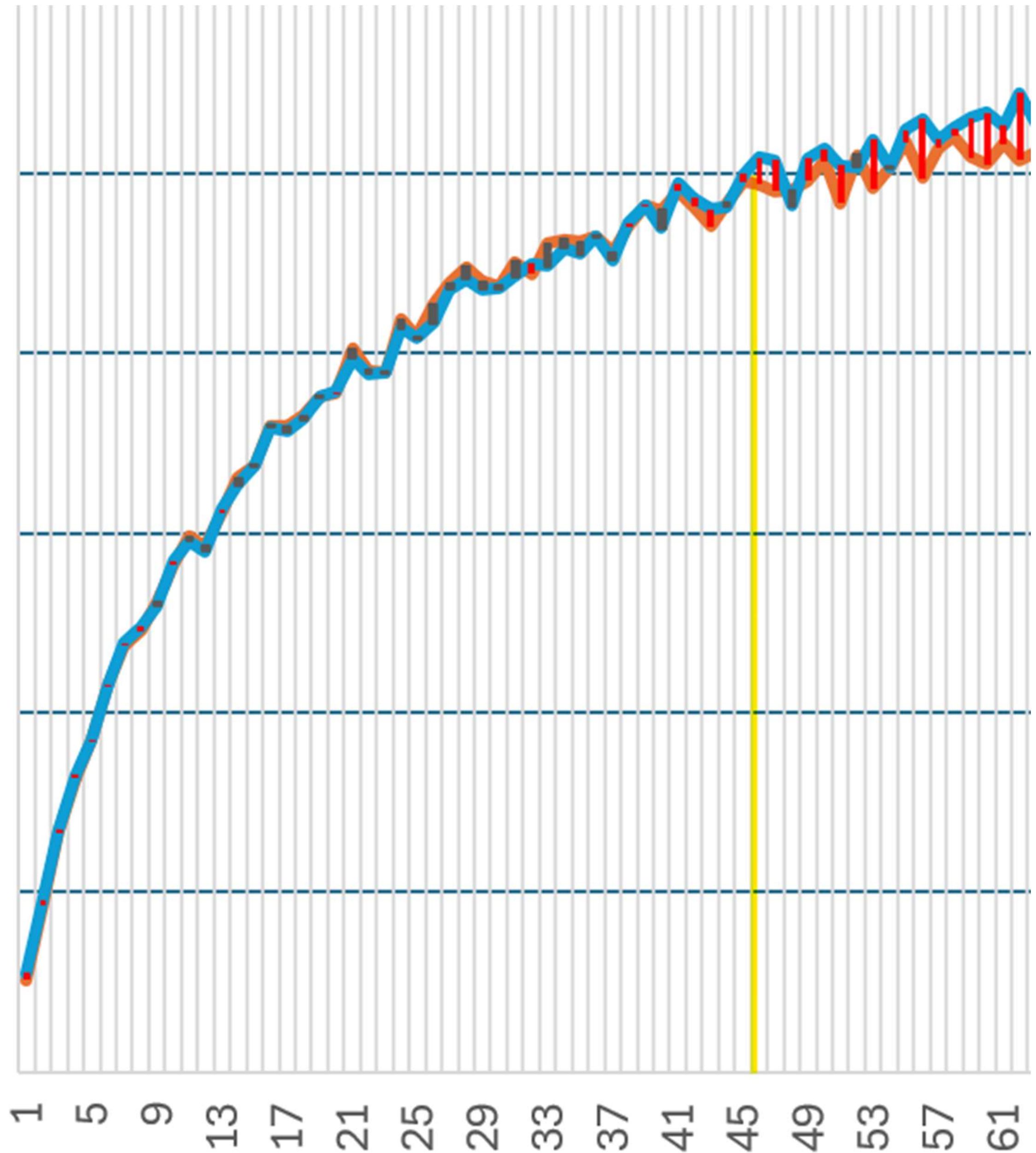


Figure 16 Long-term dataset further analysis

It is expected, when the FPS count is lower, that it is possible to move more characters than in a higher FPS count setting. On a 30 FPS setting (Figure 17) avatar data can be sent for 64 avatars safely, before losing data. The data gathered at a 60 FPS setting (Figure 18) indicates that it is possible to send 46 units of data without experiencing data loss. This suggests that a dedicated 60 FPS test may not be necessary, as the transfer rates and data transmission capabilities are comparable to those observed in the unlimited setting.

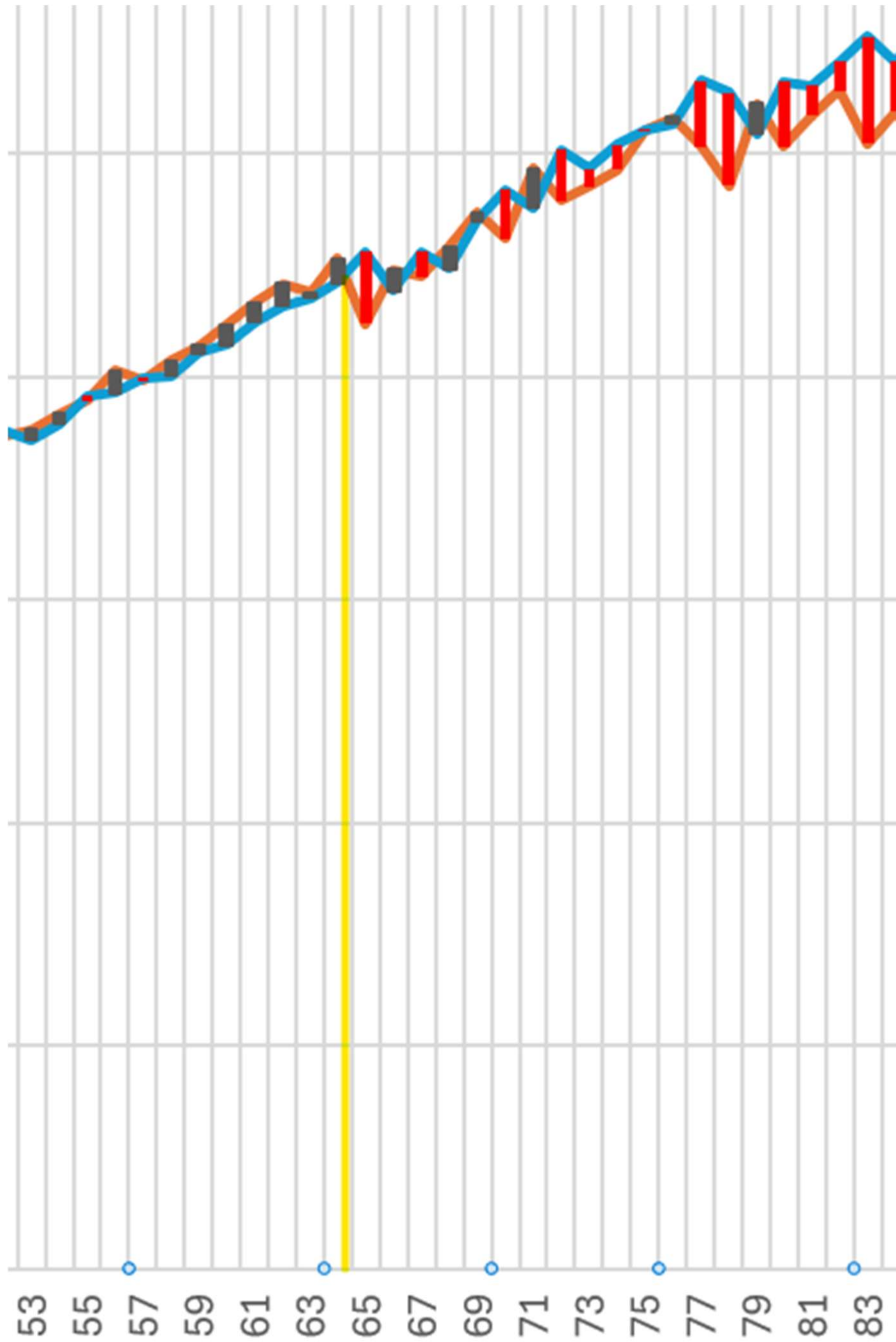


Figure 17 Further analysis of 30 FPS dataset

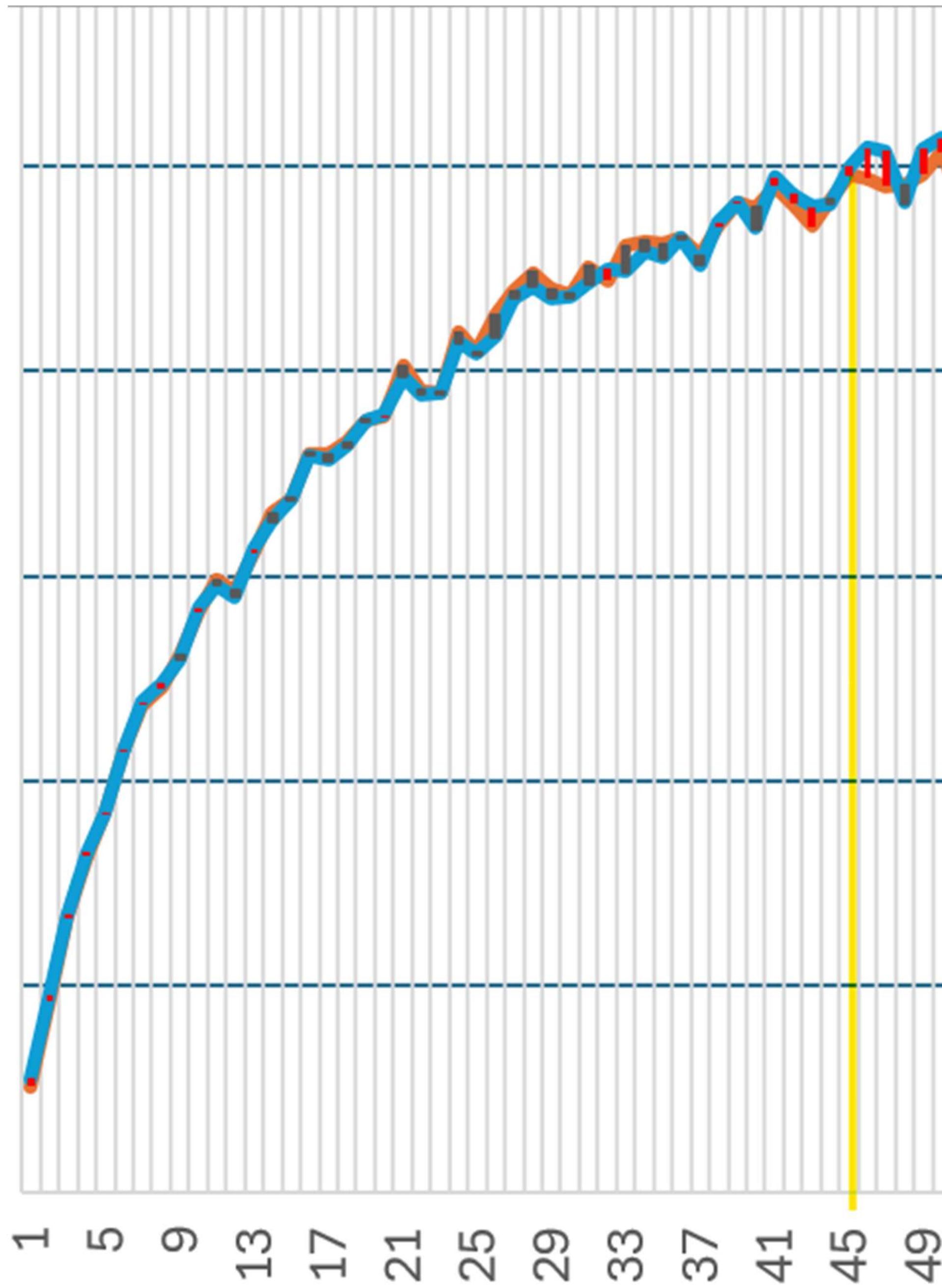


Figure 18 further analysis of 60 FPS dataset

7 Discussion

7.1 Discussion

The tests conducted within the local area network (LAN) environment provide valuable insights into data transfer capabilities under optimal conditions. These tests establish a baseline for understanding network performance, latency, and data transmission efficiency when handling real-time motion data. By using a controlled environment, the experiments ensure that external factors such as internet congestion, packet loss, or network throttling do not influence the results, allowing for a more accurate assessment of the system's capabilities.

One of the key findings is that the developed measurement tool can effectively calculate transfer speeds in various network configurations. This feature is crucial for understanding the limitations of different networking setups and optimizing the software stack accordingly. Whether used in a wired or wireless environment, the tool provides useful data that helps in fine-tuning performance and identifying potential bottlenecks. It can also serve as a diagnostic instrument when testing new implementations or troubleshooting performance issues in real-world scenarios.

7.2 Avatar Transmission Limits

The tests show that under optimal conditions, Unity can safely transmit movement data for approximately 45 avatars without any significant data loss. This indicates that the system is well-suited for medium-scale applications where multiple users interact in a shared virtual space. However, increasing the number of avatars to 65 may result in noticeable data loss, particularly when frame rates are constrained. This suggests that the system reaches a threshold beyond which data integrity becomes compromised due to bandwidth limitations or processing overhead.

The ability to accurately replicate movements in real-time is essential for ensuring a seamless and immersive virtual experience. When data loss occurs, animation quality may degrade, leading to issues such as jittery or inconsistent movement. Understanding these limitations allows for the development of strategies to mitigate performance issues, such as implementing data compression techniques, optimizing

packet transmission, or prioritizing critical data points to maintain smooth animation fidelity.

7.3 Precision and Optimization

Another important observation relates to the default float values used by Unity for representing movement data. While Unity's default floating-point precision offers high accuracy, reducing values to two or three decimal places can significantly improve data transfer speed without a noticeable impact on visual quality. This optimization reduces the overall data payload, leading to lower bandwidth consumption and improved transmission efficiency.

By refining the software stack and incorporating asynchronous functions, the system can process a larger number of reference points more effectively. This results in smoother animations with fewer interruptions or inconsistencies. Asynchronous data handling ensures that network communication does not introduce delays that could otherwise disrupt the user experience. Additionally, using efficient serialization techniques and minimizing redundant data transmission can further enhance performance, enabling real-time applications to scale more effectively.

7.4 Data Storage and Analysis

Beyond real-time data transmission, the ability to save motion data for later analysis or playback is a crucial aspect of the system. Data can be stored as JSON files, which offer a lightweight and easily readable format for further processing. This feature is particularly useful for reviewing past performances, conducting motion analysis, or debugging issues related to movement accuracy.

The JSON data can be written to text files by any of the core programs involved in the system, including **StageServer**, **VRClient**, or the **Node.js server**. This flexibility ensures that motion data can be captured at different stages of transmission, allowing developers to analyse potential discrepancies between sent and received data. Furthermore, storing the data enables researchers or performers to review and refine movements without requiring a real-time setup, facilitating iterative improvements and more efficient workflows.

7.5 Future Considerations

While the current implementation demonstrates promising results, further improvements can be made to enhance performance and scalability. One potential avenue for future development is the integration of more advanced compression algorithms to reduce the data footprint while maintaining movement fidelity. Additionally, implementing predictive algorithms based on machine learning could help interpolate missing data points in cases where minor packet loss occurs, leading to a more stable animation experience.

Another area for future work is optimizing the system for different network conditions. While LAN-based testing provides an ideal scenario, real-world applications often involve varying levels of latency and bandwidth constraints. Conducting tests in diverse network environments, such as cloud-based or peer-to-peer setups, would provide further insights into how the system performs under less predictable conditions.

Ultimately, the findings suggest that avatar-based live performances in virtual spaces are not only feasible but also scalable with the right optimizations. By leveraging efficient data handling, asynchronous processing, and network-aware optimizations, it is possible to create smooth and immersive experiences for performing arts and other interactive applications.

.

8 Conclusion

The ability to transmit motion and interaction data over the internet opens up a wide range of possibilities for real-time communication and immersive virtual experiences. While this thesis has primarily focused on developing a virtual theatre platform for actors to present plays in VR, the underlying techniques have far-reaching applications beyond performing arts. By enabling remote, real-time interaction in digital spaces, this approach contributes to the broader evolution of the Metaverse and virtual presence technologies.

One significant potential application is the use of remote helpers in environments where maintaining anonymity is essential. For instance, in sensitive settings such as prisons, rehabilitation centres, or therapy sessions, remote professionals could provide guidance, education, or mental health support without revealing their identities. This could help reduce bias, ensure security, and allow professionals to assist individuals in high-risk environments without being physically present. Similarly, in customer service or technical support, companies could use virtual avatars controlled by human operators to assist clients, offering a more engaging and interactive alternative to traditional call centres.

Beyond social services, marketing and sales could greatly benefit from these techniques. Virtual environments allow sales representatives to demonstrate and showcase products in a more immersive manner, making online shopping experiences more dynamic. For example, real estate agents could guide potential buyers through virtual property tours, while automobile sales could include live demonstrations of car features in a fully interactive 3D showroom. Businesses could create entire virtual storefronts, where customers interact with real salespeople represented as avatars, providing a richer and more personal shopping experience compared to static web pages or automated AI chatbots.

In the gaming industry, particularly in role-playing games (RPGs), traditional non-player characters (NPCs) could be replaced by real people controlling avatars remotely. This would introduce dynamic, unpredictable interactions, making game worlds feel more alive and immersive. For example, in a medieval RPG, a blacksmith, merchant, or innkeeper could be portrayed by a real remote actor, responding to players in natural, unscripted ways. This concept could extend to escape rooms, murder mystery games, and interactive storytelling experiences, where live actors guide players through unique narratives. Furthermore, live-hosted gaming experiences could provide employment

opportunities for performers, storytellers, and professional dungeon masters in tabletop-inspired digital worlds.

Education and training could also benefit from these advancements. Virtual classrooms and training simulations could feature real-time instructors interacting with students or trainees in realistic, immersive settings. Medical professionals could practice procedures in VR with guidance from remote experts, and corporate training programs could feature live mentors instead of pre-recorded lessons.

The versatility of the techniques outlined in this thesis highlights their value across multiple industries, from entertainment and commerce to education and professional services. As technology advances, further refinements in networking, motion capture, and AI-driven enhancements will likely expand these possibilities even further. Future research could explore latency reduction, enhanced avatar realism, and integration with artificial intelligence to create even more seamless virtual experiences.

In conclusion, the methodologies developed in this research are not limited to virtual theatre but represent a foundation for expanding digital interaction across numerous domains. The ability to control avatars in real time opens new doors for collaboration, commerce, entertainment, and education, making digital presence more immersive and engaging than ever before.

References

Yang, J. (2024). Analysis of Motion Capture Technology Research and Typical Applications. Applied and Computational Engineering, 112, 130-138.

Sabatier, J. M., & Ekimov, A. E. (2006). Ultrasonic Methods for Human Motion Detection. The University of Mississippi, National Center for Physical Acoustics.

Sandilands, P., Geol-Choi, M., & Komura, T. (2012). Capturing Close Interactions with Objects Using a Magnetic Motion Capture System and a RGBD Sensor. In *Lecture Notes in Computer Science* (Vol. 7660, pp. 205-214). Springer, Berlin, Heidelberg

Rahul M., 2018, Review on Motion Capture Technology. Available: https://globaljournals.org/GJCST_Volume18/4-Review-on-Motion-Capture-Technology.pdf [28.3.2019]

Ruohisto, Antti. (2016), Luotettavan UDP-pohjaisen protokollan toteutus, [Bachelor's degree, Metropolia Ammattikorkeakoulu].Retrieved from https://urn.fi/URN:NBN:fi:amk-201605168043

Luomala, Niko. (2020) , WebSocket-protokollan soveltuvuus tietoliikenteen kannalta reaaliaikaisten sovellusten tekemiseen. [Bachelor's degree, Haaga-Helia ammattikorkeakoulu] Retrieved from https://urn.fi/URN:NBN:fi:amk-2020120325869

TeamSpeak. (2024). https://www.teamspeak.com/

Unity Technologies. (2024). ScriptableObject. Retrieved from https://docs.unity3d.com/ScriptReference/ScriptableObject.html

Express.js. (2024). Express.js - Minimalist and flexible Node.js web application framework. Retrieved from https://expressjs.com/

Wexler, J. (2019). Designing and building web applications: A practical guide. O'Reilly Media.

```
private IEnumerator SendData(GameObject player)
    {

        Animator animator = new Animator();
        animator = player.GetComponent<Animator>();
        while (true)
        {

            SkeletonDataWrapper wrapper = new
SkeletonDataWrapper();
            wrapper.playerCount = avatars.Count;
            wrapper.worldPosition =
player.gameObject.transform.position;
            Debug.Log(wrapper.worldPosition.ToString());
            // Example skeleton joint data
            Vector3[] skeletonDataPos = new Vector3[18];
            // Populate the skeletonData array with your actual
skeleton joint positionvectors

            skeletonDataPos[0] =
animator.GetBoneTransform(HumanBodyBones.Hips).localPosition;
            skeletonDataPos[1] =
animator.GetBoneTransform(HumanBodyBones.Head).localPosition;
            skeletonDataPos[2] =
animator.GetBoneTransform(HumanBodyBones.LeftUpperArm).localPosi
tion;
            skeletonDataPos[3] =
animator.GetBoneTransform(HumanBodyBones.LeftLowerArm).localPosi
tion;
            skeletonDataPos[4] =
animator.GetBoneTransform(HumanBodyBones.LeftHand).localPosition
;
            skeletonDataPos[5] =
animator.GetBoneTransform(HumanBodyBones.RightUpperArm).localPos
ition;
```

```
            skeletonDataPos[6] =
animator.GetBoneTransform(HumanBodyBones.RightLowerArm).localPos
ition;

            skeletonDataPos[7] =
animator.GetBoneTransform(HumanBodyBones.RightHand).localPositio
n;

            skeletonDataPos[8] =
animator.GetBoneTransform(HumanBodyBones.LeftUpperLeg).localPosi
tion;

            skeletonDataPos[9] =
animator.GetBoneTransform(HumanBodyBones.LeftLowerLeg).localPosi
tion;

            skeletonDataPos[10] =
animator.GetBoneTransform(HumanBodyBones.LeftFoot).localPosition
;

            skeletonDataPos[11] =
animator.GetBoneTransform(HumanBodyBones.RightUpperLeg).localPos
ition;

            skeletonDataPos[12] =
animator.GetBoneTransform(HumanBodyBones.RightLowerLeg).localPos
ition;

            skeletonDataPos[13] =
animator.GetBoneTransform(HumanBodyBones.RightFoot).localPositio
n;

            skeletonDataPos[14] =
animator.GetBoneTransform(HumanBodyBones.LeftShoulder).localPosi
tion;

            skeletonDataPos[15] =
animator.GetBoneTransform(HumanBodyBones.RightShoulder).localPos
ition;

            skeletonDataPos[16] =
animator.GetBoneTransform(HumanBodyBones.LeftToes).localPosition
;

            skeletonDataPos[17] =
animator.GetBoneTransform(HumanBodyBones.RightToes).localPositio
n;



            wrapper.skeletonDataPos = skeletonDataPos;
```

```
            // // Populate the skeletonData array with your
actual skeleton joint rotation vectors
            Vector3[] skeletonDataRot = new Vector3[18];


            skeletonDataRot[0] =
animator.GetBoneTransform(HumanBodyBones.Hips).rotation.eulerAng
les;
            skeletonDataRot[1] =
animator.GetBoneTransform(HumanBodyBones.Head).rotation.eulerAng
les;
            skeletonDataRot[2] =
animator.GetBoneTransform(HumanBodyBones.LeftUpperArm).rotation.
eulerAngles;
            skeletonDataRot[3] =
animator.GetBoneTransform(HumanBodyBones.LeftLowerArm).rotation.
eulerAngles;
            skeletonDataRot[4] =
animator.GetBoneTransform(HumanBodyBones.LeftHand).rotation.eule
rAngles;
            skeletonDataRot[5] =
animator.GetBoneTransform(HumanBodyBones.RightUpperArm).rotation
.eulerAngles;
            skeletonDataRot[6] =
animator.GetBoneTransform(HumanBodyBones.RightLowerArm).rotation
.eulerAngles;
            skeletonDataRot[7] =
animator.GetBoneTransform(HumanBodyBones.RightHand).rotation.eul
erAngles;
            skeletonDataRot[8] =
animator.GetBoneTransform(HumanBodyBones.LeftUpperLeg).rotation.
eulerAngles;
            skeletonDataRot[9] =
animator.GetBoneTransform(HumanBodyBones.LeftLowerLeg).rotation.
eulerAngles;
            skeletonDataRot[10] =
animator.GetBoneTransform(HumanBodyBones.LeftFoot).rotation.eule
rAngles;
            skeletonDataRot[11] =
animator.GetBoneTransform(HumanBodyBones.RightUpperLeg).rotation
.eulerAngles;
```

```
            skeletonDataRot[12] =
animator.GetBoneTransform(HumanBodyBones.RightLowerLeg).rotation
.eulerAngles;

            skeletonDataRot[13] =
animator.GetBoneTransform(HumanBodyBones.RightFoot).rotation.eul
erAngles;

            skeletonDataRot[14] =
animator.GetBoneTransform(HumanBodyBones.LeftShoulder).rotation.
eulerAngles;

            skeletonDataRot[15] =
animator.GetBoneTransform(HumanBodyBones.RightShoulder).rotation
.eulerAngles;

            skeletonDataRot[16] =
animator.GetBoneTransform(HumanBodyBones.LeftToes).rotation.eule
rAngles;

            skeletonDataRot[17] =
animator.GetBoneTransform(HumanBodyBones.RightToes).rotation.eul
erAngles;



            wrapper.skeletonDataRot = skeletonDataRot;

            wrapper.playerName = player.name;

            string jsonData = JsonUtility.ToJson(wrapper);


            if (isRecording)

            {

                File.AppendAllLines(filePath, new[] { jsonData
});

                Debug.Log("JSON data saved to " + filePath);

            }



            // Convert the JSON data to bytes using UTF-8
encoding

            byte[] jsonDataBytes =
Encoding.UTF8.GetBytes(jsonData);

            totalBytesSent += jsonDataBytes.Length;
```

```
            // Send the data via UDP
            udpClient.Send(jsonDataBytes, jsonDataBytes.Length,
ServerURL, serverPort);


            // Wait for the specified interval before sending
data again
            yield return new WaitForSeconds(SendInterval);
        }
    }
```