Aalto University
School of Science
Master’s Programme in Computer, Communication and Information Sciences

Robert Lee Seligmann

Web-based Client for Remote Rendered Virtual Reality

Master’s Thesis
Espoo, June 29, 2020

Supervisor: Professor Antti Ylä-Jääski, Aalto University
Advisor: Teemu Kämäräinen, Aalto University
Although the concept of Virtual Reality (VR) has existed since the 1960s, it is only in recent years that it has gained a significant amount of attention from consumers, businesses, and researchers. High-end VR headsets offer the best visuals to the users; however, the high cost of running a high-end VR setup limits the consumers’ adoption which often fall back on less expensive mobile or standalone headsets. With the mobile hardware included in these devices, the graphics capabilities are limited compared to high-end devices. Streaming VR content over the network can potentially provide high-quality experiences to headsets with limited power. Moreover, the streaming service can be used to provide the same rendering experience to different platforms, regardless of the client’s hardware. However, any remote rendering solution unavoidably suffers from latency caused by the network, the amount of latency may not be acceptable in latency sensitive applications such as VR.

The goal of this thesis is to develop a cross-platform VR application that delegates the rendering process to the server, the rendering results are then sent back to the client application for display. Unlike most of the previous systems, the client and server should be designed to support multiple headsets and controllers, the design should also allow to easily add the support for new devices in the future. More importantly, the client application should fulfill the basic VR requirements and ensure low latency to provide the best user experience. The server was built with Unity, and the client, with A-Frame. The streaming components were handled by using the WebRTC open standard.

Although the implemented system in this work was functional, the performance evaluation showed that the client application could not maintain the minimum of 60 Frames-per-Seconds (FPS) which is required to provide a fluid playback. Moreover, the high stale frame count suggests that additional latency is added when displaying the frame. Optimization in the decoding speed and shader are necessary to improve the frame rate.
Acknowledgements

First, I would like to express my gratitude to my advisor Teemu Kämäräinen for his valuable advice, support, and guidance throughout the entire process. I would also like to thank my supervisor Professor Antti Ylä-Jääski for supervising this master’s thesis.

I am also grateful to my family and friends who have supported and encouraged me throughout my studies. Additional thanks to John Cheng for proof-reading and giving me feedback on written parts of my thesis.

Espoo, June 29, 2020

Robert Lee Seligmann
### Abbreviations and Acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application programming interface</td>
</tr>
<tr>
<td>AR</td>
<td>Augmented Reality</td>
</tr>
<tr>
<td>CPU</td>
<td>Central processing unit</td>
</tr>
<tr>
<td>DOF</td>
<td>Degree Of Freedom</td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model</td>
</tr>
<tr>
<td>FoV</td>
<td>Field of View</td>
</tr>
<tr>
<td>GLSL</td>
<td>OpenGL Shading Language</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>HMD</td>
<td>Head-Mounted Display</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>HTTPS</td>
<td>Hypertext Transfer Protocol Secure</td>
</tr>
<tr>
<td>IBR</td>
<td>Image-Based Rendering</td>
</tr>
<tr>
<td>ICE</td>
<td>Internet Connectivity Establishment</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>LDI</td>
<td>Layered Depth Images</td>
</tr>
<tr>
<td>LOD</td>
<td>Level of Details</td>
</tr>
<tr>
<td>NAT</td>
<td>Network Address Translator</td>
</tr>
<tr>
<td>ND</td>
<td>Network Delay</td>
</tr>
<tr>
<td>OD</td>
<td>Playout Delay</td>
</tr>
<tr>
<td>PD</td>
<td>Processing Delay</td>
</tr>
<tr>
<td>QoE</td>
<td>Quality of Experience</td>
</tr>
<tr>
<td>RTT</td>
<td>Round-Trip Time</td>
</tr>
<tr>
<td>SDP</td>
<td>Session Description Protocol</td>
</tr>
<tr>
<td>STUN</td>
<td>Session Traversal Utilities for NAT</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>VR</td>
<td>Virtual Reality</td>
</tr>
<tr>
<td>XR</td>
<td>Extended Reality</td>
</tr>
</tbody>
</table>
Contents

Abbreviations and Acronyms 4

1 Introduction 7
   1.1 Problem Statement ....................................... 8
   1.2 Structure of the Thesis ................................... 9

2 Background Study 10
   2.1 eXtended Reality ......................................... 10
      2.1.1 Virtual, Augmented and Mixed Reality .......... 10
      2.1.2 Hardware and Other Limitations .................. 11
   2.2 Remote Rendering ........................................ 12
      2.2.1 Graphic Data ........................................ 15
      2.2.2 Data Compression .................................... 18
      2.2.3 Latency Reduction Techniques .................... 19
   2.3 Standardization of XR .................................... 22

3 System Overview 24
   3.1 System Design ............................................ 24
   3.2 System Overview .......................................... 25
      3.2.1 The Server .......................................... 25
      3.2.2 The Client .......................................... 26
   3.3 Technologies .............................................. 26
      3.3.1 A-Frame ............................................ 26
      3.3.2 Unity ............................................... 27
      3.3.3 WebRTC ............................................ 28
   3.4 Hardware ................................................ 28

4 Implementation 30
   4.1 Signaling Server ......................................... 30
   4.2 WebRTC ................................................ 32
   4.3 Camera Setup ........................................... 33
4.4 Remote Input ................................................. 36
4.5 Client Setup .................................................. 40

5 Evaluation .................................................. 45
  5.1 Testing Procedure ........................................... 45
  5.2 Server Performance ......................................... 46
  5.3 Client Performance .......................................... 48
    5.3.1 Frame Rate ............................................ 49
    5.3.2 Bandwidth .............................................. 57

6 Discussion .................................................. 59
  6.1 Limitations ................................................ 60
  6.2 Future Work ............................................... 61

7 Conclusions ................................................. 62
Chapter 1

Introduction

During the recent years, Virtual Reality (VR) has gained a significant amount of attention from consumers, businesses, and researchers. Although the first notion of the technology dates as far back as the 1960s, it is only after the announcement of the Oculus Rift that it became popular. With the advancement of mobile hardware, it was made possible to create affordable VR headsets for the consumers. Thanks to the release of the Oculus Quest in 2019, the revenue for eXtended Reality (XR) climbed 26% to $6.3 billion during the same year [44]. The market for VR headsets has greatly increased over the years to satisfy the various demands, e.g. the Varjo VR-2\(^1\) is a high-end headset with human-eye resolution that is best suited for industrial corporations, and the Oculus Quest is an all-in-one VR headset best suited for mainstream consumers. As the research in [12] shows, the main barrier to consumer adoption is the price of the hardware, high-end consumer-grade headsets are high-priced, but they also require to be connected to a powerful computer to function. The price of a high-end headset and computer could discourage the consumer who will then fall back on lower-end alternatives such as standalone or mobile headsets. These do not need to be connected to a computer, all the required hardware is mounted directly in the headset, or it may be powered by a smartphone. These grant the user additional comfort and freedom when moving as the headset is not tethered. However, these headsets are limited to mobile device hardware and cannot provide the same level of quality as high-end headsets. Using a remote rendering server could potentially serve as a solution to provide high-quality graphics to low-end headset through a wireless connection. This has been a popular topic recently with the releases of new Cloud Gaming services. In a remote rendering system, the server will handle all the heavy graphic computations

\(^1\text{https://varjo.com/}\)
and forwards the rendering results to the client device. The client will only have to render the received results on screen and will not need any dedicated graphic hardware. However, using a remote server to render graphics will cause a delay before the frame is displayed on the screen. This could be harmful in a low-latency system such as VR applications. Therefore, developing a VR application that relies on a remote rendering server will have to be designed in a way that ensures that the perceived latency is minimal and does not deteriorate the experience of the user. The rendering service is not tied to a unique device, it can provide the same rendering experience to different headsets. However, if the client application has to support different headsets, the application will have to be written multiple times. Although the connection to the streaming server is the same, each headset has its own proprietary set of APIs and cannot be used with a different headset. This is a major drawback for developers as they will spend additional time adapting the application for each platform, this increase the project's cost, decreases the speed at which they can deploy new applications, and limits their potential market reach. Some solutions provide a cross-platform support; however, none has focused on adding a streaming server to the application. Successfully developing a cross-platform application that connects to a rendering server could greatly increase the market reach and user experience, as well as reduce the development costs.

1.1 Problem Statement

A lot of solutions that provide streaming service already exists on the market, and a lot of research has been done to lower the network bandwidth and latency without deteriorating the user experience. Most of the existing solutions are platform-specific and only works on a limited selection of headsets. Therefore, we plan to provide a similar system that is compatible with different headsets and that can be extended to support new headsets in the future. The goal of this thesis is two-fold. The first goal is to use modern solutions to develop a cross-platform application that can receive a video stream from the server. The second goal is to optimize the rendering process to mitigate the latency of the system.

Thus, the goal of this thesis is to provide a cross-platform VR solution that can leverage the dedicated hardware of a rendering server to offer high-quality graphics to hardware-limited devices. Although the streaming system is functional and the cross-platform support is enabled, the evaluated performance of the client application does not achieve the expected performance. The evaluation demonstrated that the video shader and the video decoding
process take a significant amount of time. This causes the frame rate to decrease drastically and the stale frame to increase, thereby adding additional latency when displaying new frames.

1.2 Structure of the Thesis

This thesis is divided into 6 main chapters. In Chapter 2, we will introduce the concepts of XR technologies, remote rendering systems, and the standardization of XR. Then, in Chapter 3, we will overview the main components of the system and list the tools that will be used to develop it. The detailed explanation of the implementation is presented in Chapter 4. Then, the performance evaluation of the implemented system is presented in Chapter 5, and the results, limitations, and improvements are discussed in Chapter 6. Lastly, Chapter 7 concludes this thesis by summarizing the main points of this work.
Chapter 2

Background Study

This chapter presents all the relevant information and related work on remote rendered applications and eXtended-Reality (XR) technologies. Section 2.1 will overview the different technologies that XR consists of, such as Virtual and Augmented Reality (VR and AR respectively). Then, Section 2.2 will focus on the remote rendering paradigm and its consequence on latency critical applications. Lastly, Section 2.3 will review the current standards for XR technologies.

2.1 eXtended Reality

2.1.1 Virtual, Augmented and Mixed Reality

The concept of Virtual Reality has existed since the 1960s [10]; however, it is only in recent years that it has gained a significant amount of attention and popularity. A Virtual Reality head-mounted display (HMD) renders a computer-generated virtual environment or a 360° degree footage (video or image) where the user can visualize the virtual world and interact with it using an input device such as a controller. In Augmented Reality (AR), the user is not projected in a virtual world; instead, the user views the real world which has been augmented with virtual objects. Mixed Reality (MR) is a hybrid of VR and AR, allowing to merge the real and the virtual world. These technologies are often referred to by their umbrella term, i.e. eXtended Reality (XR).

The development of VR products was made possible thanks to the improvement of mobile phone hardware and resolution display. The level of immersion depends partly on the possible movements achievable, these are referred to as Degree Of Freedom (DOF). A basic VR headset allows rota-
tional movements of the head (i.e. 3DOF), while a headset with more advanced tracking hardware includes rotational and translational movements (i.e. 6DOF).

Currently, the market can be divided into two categories: tethered and mobile headsets. The former requires to be connected to a high-end computer with a powerful Graphics Processing Unit (GPU). Thanks to the greater computing power, a higher visual quality and performance can be achieved at the cost of limited mobility from the cable connection. On the other hand, mobile headsets can be either powered by a smartphone, or have the necessary hardware built-in (referred to as a standalone headset). In both cases, they are limited to mobile hardware that is less powerful; therefore, the application can be less immersive. For a smooth experience, a mobile VR application will require to be optimized.

2.1.2 Hardware and Other Limitations

The survey in [12] reports that the main barrier to mass consumer adoption is the price of VR devices, this is mostly due the high cost involved with running a high quality VR experience. The price for a high-end tethered headset and a VR desktop computer with minimal VR capabilities would amount to at least 1'500.- euros. If the user wants even better performance, they will need to invest in more expensive hardware. In the same survey, it was noted that the headset comfort and usability needed to be improved in order to increase the consumer adoption. This is referring to tethered devices as the cable connection restricts the user’s movement and can potentially be hazardous to the user [22].

Based on this information, mobile VR could prove to be the ideal solution as the cost is lower and there is no tethered connection. Furthermore, the sales of standalone headsets have more than doubled in 2019, following the release of the Oculus Quest in May [44], and represented 49% of VR sales in the same year. The Oculus Quest is an affordable, easy to use, and capable headset, which recently provided hand-tracking features, and the possibility to connect to a computer in order to achieve higher performance.

A high-performance VR application requires a heavy computing power, a performance degradation can negatively impact the quality of experience (QoE), possibly resulting in discomfort or motion-sickness, i.e. mismatch between the sensory vestibular system and the visual senses [10]. In order to have a satisfying experience, a VR application should meet specific requirements regarding the visuals, the mobility, and the latency [22].

One of the major requirements is the need for low motion-to-photon latency, also shortened to latency. It can be described as the time occurring
between the moment the headset or controller moves and the moment the frame with the new position and orientation is rendered to the HMD. The latency should not surpass 20 ms to guarantee a good experience [2, 4], any excess up until 50 ms will still feel responsive, yet noticeably delayed [28]. This is the strictest requirement since a high latency throughput will also expose the user to motion sickness. However, it should be noted that people may react differently to latency, a few may not notice a 100 ms delay while others may perceive a latency even lower than that [30].

To offer the best possible experience, the system should also provide high-quality visuals to offer abundant and photo-realistic scenes to the user. Moreover, to provide a smooth playback, a minimum of 60 frames-per-seconds (FPS) should be targeted [4]. Regarding mobility, the hardware should not be restricting the user’s actions. Tethering a headset can prove to be hazardous or will, at the very least, hamper mobility and decrease the QoE. Providing untethered experience offers a more immersive and comfortable experience [22, 24]. To reach a stunning level of visual detail, it is possible to run the graphics rendition on a remote server. This practice is well-known for gaming and is referred to as Cloud Gaming. It provides devices with limited hardware a way to run video games smoothly, as long as it is connected to the network.

2.2 Remote Rendering

A remote rendering system is a server-client system, where the computation of the 3D graphics is shifted from the client to the server. The main role of the client is to display the received data; however, it must also listen to the user’s input and forwards the new state to the server. The client might have to do additional processing depending on the data format, e.g. decoding of the data or rendering simple structure with a low-end GPU. The role of the server is to listen to the input forwarded by the client and update the simulation appropriately, a new frame is then rendered and delivered to the client. By doing so, the server will do the heavy computation which should normally be done on the client (which is then referred to as thin client). Accessing the services of a remote server can be achieved through a native application or a web browser [6].

Rendering on a server instead of the client holds many advantages. Unlike the client, the server’s resources (e.g. CPU, GPU, and memory) are flexible and are usually equipped with better hardware than the client device, thereby offering a better rendering performance. The computing resources of the server can be efficiently shared between multiple clients; moreover, it of-
fers a cross-platform solution where different platforms can receive the same rendering experience, granted the client program has been developed with the server in mind. Furthermore, it can offer additional security features for the service provider as it can prevent piracy or theft of content [37]. This is achievable when only sending an image frame of the scene instead of the 3D model itself; therefore, the model is never on the client device. The different types of content will be discussed in Section 2.2.1.

![Diagram of basic remote rendering system](image)

Figure 2.1: An example of basic remote rendering system. Image from [37].

Remote rendering technologies have been receiving an increasing interest since the penetration of broadband internet and cloud computing technologies [6]. A study made by Cisco predicts that 79% of the mobile network traffic in 2022 will be related to videos, and XR traffic will have increased 12-folds since 2017 [8].

Cloud Gaming is another type of remote rendering experience that has gained popularity. The players would only require a thin client and a broadband internet connection to play. Using a cloud gaming server bypasses the tedious setup and the need to constantly upgrade their hardware in order to play the latest video games. This also removes the risk of hardware or software compatibility issues as a game supported by the service provider is guaranteed to run [6]. The Cloud Gaming market is getting larger with big companies now providing these services, e.g. Shadow\(^1\), Nvidia Now\(^2\), PlayStation Now\(^3\), and Google Stadia\(^4\).

Most of the research on the topic of remote rendering revolves around the challenge of reducing the latency and optimally transferring data from server to client to ensure low bandwidth usage. Alike VR, the latency, or interaction latency, is the time from when the user interaction is processed by the application until the new frame with the updated state is displayed to the user [37]. Some applications may be latency-sensitive, and a high latency can greatly impair the enjoyment and experience of the user. In

\(^1\)https://shadow.tech/int/
\(^2\)https://www.nvidia.com/en-us/geforce-now/
\(^3\)https://www.playstation.com/en-us/explore/playstation-now/
\(^4\)https://stadia.dev/
cloud gaming for instance, the highest acceptable latency for a first-person shooter is 100 ms, whereas for a remote VR system, the same delay of 20 ms or less as mentioned previously should be targeted [2, 4]. As in the work in [6, 14, 19], we define the latency, or response delay, as being fragmented into 3 components: the Network Delay (ND), the Processing Delay (PD), and the Playout Delay (OD).

The network delay takes into consideration the time needed to deliver a user command and the time to deliver the frame back to the client. It is also referred to as the network round-trip time (RTT) [6]. The processing delay represents the time from the server’s reception of the input command until the time it processes the input and responds with a new rendered frame. The playout delay accounts for the time between the user’s reception of the frame until it is decoded and shown on the display. The ND will also depend on the network conditions and on the data sent, as larger data will take longer to transfer over the network.

For this reason, a vast amount of research has focused on ways to optimize the bandwidth consumption rate, e.g. by compressing the data or by splitting the workload. However, to solve the bandwidth challenge, it is also important to consider the constraints (e.g. limited bandwidth, channel condition, and other real-time requirements) and limitations of wireless connections which can depend on mobility, signal strength, and interference [37]. Moreover, the servers could be located far away from the end-user, causing unacceptable delays. Moving the servers closer to the end-user is a method of achieving low latency, this practice is referred to as edge computing or fog computing [2]. However, this report will not discuss this topic into further depth. Device limitations such as power, memory, and screen size should also be considered [40]. Low-latency requirements for cloud gaming and VR accounts for added complexity, rendering complex 3D graphics requires a lot of computation, and rendering new frames occur more often since the client’s display usually has a higher screen refresh rate. Different schemes of compression, data, and streaming may be necessary depending on the application.

There are two streaming strategies: progressive and adaptive. With progressive streaming, the server will first send the graphic at a low level of quality, then another is sent at a higher level of quality until the final quality is achieved. As data of higher quality is received, the graphics will progressively look better and sharper [13]. The data representation will remain the same throughout its run time. On the other hand, adaptive streaming will change the data representation based on different factors, the network performance being the most influential one. This strategy will attempt to offer the best achievable quality based on the current network conditions. If over time, the network deteriorates and it is not possible to continue offering the highest
quality, the streaming service can dynamically update the bit rate of the
encoder to offer a lower quality video that is adjusted to the new network
conditions [40].

2.2.1 Graphic Data

Depending on the context and application, the data transferred between the
server and the client may differ. The study in [39, 40] classifies the type of
data transferred under two distinct categories: model-based and image-

based. In a model-based approach, the server does the data processing and
generates the 3D models, e.g. point clouds or polygon meshes. The models
are then sent to the client which will proceed to render them on the screen. In
this case, the client will require dedicated and, possibly, powerful hardware
to render the 3D graphics.

In the model-based classification, the first approach consists of sending
the original model; hence, every model is sent to the client. This will
require a lot of computation from the server. Furthermore, a scene with
complex geometry and texture will consume a lot of network bandwidth and
take a substantial amount of time to transfer to the client.

To reduce the size and bandwidth consumption, another approach sends
a partial model with only a portion of the original model. The subset can
be selected depending on different factors such as the region or the user's
attention. In [35], the regions in direct sight of the user are transmitted in
full resolution with high priority. The nearby regions can be sent at lower
resolution, and far-away regions can be sent later if the bandwidth is limited.

Another approach is the simplified model. This model is aimed towards
clients with limited graphics computing hardware. The server generates a
simplified model, as well as an image containing the difference between the
rendering of the abridged and standard model [23, 40]. The client will render
the model and add the image to the rendering output. In this scenario,
the client has less workload, but the difference is being compensated by the
server. The bandwidth load is also lessened in this approach.

The last approach is a point cloud representation based on the original
model. The original model may be highly detailed, meaning the triangle
count in the mesh may be tremendous, possibly higher than the number of
pixels on the display device. The work in [11] generates a point cloud of the
original model where the size is proportional to the resolution of the display
screen, reducing the possible number of points.

All these approaches still require the client to have dedicated 3D graphics
hardware in order to properly render the models. This can be abstracted in
an image-based system where all the rendering is done on the server and
an image of the scene is sent to the client.

In an image-based system, most remote rendering systems take an image imposter approach [40]. The server renders the 3D scene and projects it into a 2D image that will be sent to the client. This reduces the network bandwidth usage, and further reduction can be achieved by limiting the image size to match the screen resolution of the display device. Thus, the required bandwidth is limited despite the level of complexity of the virtual scene. Furthermore, these images can be compressed to reduce the streaming bandwidth. Even if image compression tools are rather advanced, it will cause an increase in the processing delay and the playback delay, thereby adding extra latency.

Using an environment map to render distant background objects is a common practice in 3D game development to reduce interaction latency [22]. The server, using a given viewpoint, captures a panoramic 360° image of the surroundings, and sends it to the client which will wrap the image around the viewpoint. With the panoramic image rendered around the user, they are able to freely look around the scene without any support from the server and with very low interaction latency. However, the server must send a new environment map whenever the viewpoint changes position or when the state is updated.

The last approach is a depth image [5, 40] where the server generates two different images and sends them to the client. The first is a color map which simply stores the pixel color value. The second is a depth map that holds information regarding the distance between an object’s surface and the point of view. On the client side, if the viewpoint has not changed since the request, it will render the color map. On the other hand, if the viewpoint has been modified, the client will run an image warping algorithm on the depth map and render a new frame with the appropriate viewpoint by re-projecting every pixel [42]. Similarly to a point cloud, a depth map is a set of 3D points in space; however, running an image warping algorithm is far less computationally expensive. Figure 2.2 showcases the different data types sorted by the amount of bandwidth used and load on the client side.

Some of these approaches are ideal when the models remain unchanged. However, some applications can have part of the data altered, either by the user or by the simulation itself. For instance, this is a common occurrence in games, the virtual world is updated over time with or without any input from the user. Depending on the virtual scene and the user interaction, the workload of the server will change. The survey in [37] classifies them according to two dimensions. The first dimension describes if the nature of the 3D graphic models is dynamic or static. The second defines the quality of the user interaction, it depends on how freely the user can change the
virtual camera and world. The interactions are then described as restricted or unrestricted.

With a static model and restricted interaction, the viewpoints are either pre-defined in advance or limited in range. This is often used in visualization systems to visualize complex data, such as medical imagery, scientific data, or industrial design. In this case, the model can be pre-computed on the server and a rendering of the viewpoint is sent when required.

A dynamic model and restricted interaction is often related to 3D videos or animations which requires a real-time rendering of frames. A single frame must be rendered before the next sequential frame; therefore, no time-consuming computation can be applied, and no pre-computing can be done in advance. Some existing research proposed different alternatives to ensure low latency, such as using Image-Based Rendering (IBR) frameworks [40] and splitting the rendering workload [7].

With a static model and unrestricted interaction, the user interactions are unregulated, they can modify the viewpoint of the virtual camera as well as the data in the scene. An example of real-life application is a virtual walkthrough which is common in online games and VR experiences. The main challenge is the complexity of the virtual environment which is often too large to be transferred efficiently for real-time applications. A few research focuses on streaming the scene from a central server to the client as efficiently as possible. Different systems have been proposed and many techniques rely on generating models at different resolutions. The work in [18] renders at lower quality the objects that are viewed in the user’s peripheral vision while prioritizing the objects directly in the line of sight. Another work [35] suggested a pipeline where the server acts as a database that stores 3D models and exclusively sends the visible models. All the models are arranged with different Level of Details (LODs). Depending on the area
where the user is located, the models will be transmitted at different LODs. Other systems rely on an image streaming approach with panoramic images or depth images.

Virtual Reality and cloud gaming are example applications of Dynamic model and unrestricted interaction. This is the most challenging one as the scene is usually comprised of complex 3D elements such as avatars, simulations, and animated effects. The state of the virtual world is often updated and the user has a vast choice of interactions that will affect the game state. The scene will therefore have to be constantly rendered to offer the latest frame to the user. Moreover, video games and VR must render the graphic data at much higher frame rate than other applications (e.g. 100 fps for a first-person shooter) and with very low latency [2, 37].

2.2.2 Data Compression

Data compression tools are often utilized in remote rendering systems to alleviate the network load. This is also crucial for latency sensitive applications since large data will take a relatively longer time to be transmitted. This section presents the different tools for compressing data.

There are many common tools used to compress images, videos, and graphics; the best-known methods are lossless and lossy compression. On the one hand, a lossless compression tool removes any unnecessary metadata in the image and compresses it to a reduced bitrate. Decoding the encoded data concludes to a perfect reconstruction of the original data. Texts, images, binaries, audio, and graphics can be encoded with a lossless compression tool. This can be an important factor in critical applications such as the viewing of medical images. On the other hand, a lossy compression tool drops details in the image which are unnoticeable to the human eye. Additionally, videos can also be compressed, but redundant information across frames cannot be leveraged. Hybrid video coders, such as MPEG-4 or H.264/AVC, are lossy compression tools that integrates motion compensation in order to take advantage of temporal redundancies [37].

Compound image encoders can be used for data with less motion, comprising of texts, graphics, and images [37]. The data is segmented into 2 parts which are then compressed with a different generic compression tool. The texts and graphics are compressed using a lossless encoding, whereas the natural images use a lossy compression tool (usually JPEG). Graphics encoders are utilized to encode geometrical graphics data such as volumetric data and triangle meshes. For progressive streaming of 3D content, a spectral compression tool can be used to project the 3D data onto an orthonormal basis to retrieve a compact representation [20]. Depth compression tools have
been well studied in the literature. Some methods rely on regions-of-interest, dynamic range adjustments, or Layered Depth Images (LDI). Research on the topic is vast, but goes beyond the scope of this work, interested readers are referred to a survey presented in [41].

Some researchers discovered that improved encoding performances can be reached by jointly encoding metadata extracted from the rendering engine to the rendered frame [37]. The metadata can serve multiple purposes, the work in [27] uses the information to divide the background and foreground into macroblocks. Different quantization parameters are then applied to each block and the less important regions are compressed at a higher ratio while maintaining the appropriate quality for important regions. Motion information from the engine was also analyzed to speed up the motion estimation of the encoder. Another work in [38] uses the run-time game engine context to improve the efficiency of the video encoder. By leveraging the context information such as the pixel depth, render viewpoints, camera motion patterns, as well as supplementary frames that will not be streamed, they are able to boost the compression ratio of their cloud gaming streaming service and decrease the interaction latency.

2.2.3 Latency Reduction Techniques

Every remote rendering system suffers from an amount of latency, the maximum amount deemed acceptable for a comfortable user experience will depend on the type of application, e.g. a VR experience favors a maximum of 20 ms [4], and at most 100 ms for an FPS game [37]. Current optimization techniques can decrease latency, but they cannot reduce it lower than the minimum generated by the network round-trip delay. The only way to bypass this delay is to generate a frame on the client while waiting for the response frame from the server.

**Local Rendering** is the easiest way to generate new frames locally. The 3D assets are stored on the client which also renders the graphics to the screen. Systems based on mesh streaming may also render the scene locally, reducing the interaction latency to the minimum delay required to generate the visuals on the display. The main issue is to properly synchronize the local rendering with the server. For a simple application such as a model visualization, only the viewpoint may need to remain consistent. This becomes harder for complex applications such as video games, where it is crucial to understand the logic of the game and user data to synchronize the server and client properly [37]. **Dead Reckoning** follows a similar approach to local rendering. This strategy is mainly used in multiplayer games to reduce the aforementioned synchronization issue. The premise of this strategy is to
define beforehand a series of algorithms that the client can use to predict the behavior of the game’s entities. Moreover, it is important to decide a threshold for every set of algorithms to issue a correction to the server whenever the prediction is too far from the expected behavior [29].

For systems that do not possess the sufficient hardware to render the graphics locally, an efficient strategy to reduce latency is to prefetch the content. Using a prefetch strategy, a client (or server) tries to predict the user’s future movements, then request the frames corresponding to all the predictions. If a prediction is successful, the client displays the matching prefetched frame with no latency. If all the predictions miss, the prefetch buffer is cleared and the client awaits the response frame from the server. The type of content to be prefetched can vary, panorama frames can be used as well to reduce the latency of rotational movements. However, prefetching frames come at the cost of extra network bandwidth usage in order to prefetch all the predicted frames, and most of the frames may never be utilized (if the prediction never hits). This strategy works best with a static environment and is unsuitable for an application with a dynamic scene such as a video game. As the data in these applications updates frequently, each frame will expire before being displayed and requesting a new frame will be necessary. Using this strategy in a dynamic application would require prefetching every prediction for every frame, subsequently increasing the bandwidth usage and the number of unused frames [37].
Another strategy that fits between local rendering and prefetching is **Image-Based Rendering (IBR)**. IBR requires the server to send to the client additional data along the rendered frame. When the client changes its viewpoint, it can use the extra data to run an IBR algorithm to generate a new frame at the updated viewpoint. By using an IBR strategy, the latency is shortened to the time necessary to execute the IBR algorithm locally. A notorious algorithm is **3D Image Warping** where the server sends a depth image. Using the depth image (which includes a depth and color map) and the corresponding current viewpoint, the 3D warping algorithm can output a new frame for a targeted viewpoint; thus, reducing the latency to the time needed to run the warping algorithm. Embedded CPUs/GPUs can efficiently run warping algorithms, making it ideal for mobile-oriented devices. Hole artifacts are likely to occur in warping algorithms since the output image depends on the pixels of the input. Artifacts can transpire due to the object being occluded or outside of the input image [40].

Many methods have been suggested to solve the issues of hole artifacts, with some working better in certain scenarios. For example, the *splat warping* in [15] is effective when dealing with small hole artifacts. The *wide field-of-view warping* [51] and *Super view warping* [3] use large reference frames to mitigate the artifacts that are caused by occlusion. The work in [3], uses the server to calculate the difference between the rendered and predicted image, the result is sent to the client to compensate for any warping error.

**Double Warping** and **Layered Depth Images (LDI)** are two notorious methods to cover exposed holes. Compared to a normal depth image where every pixel has a depth value and a color value (referred to as depth pixel), a LDI can potentially contain multiple depth pixels per pixel. They can be used to fill the occluded holes when the viewpoint is moving. The LDI can be computed by using multiple depth images from different viewpoints, using a custom ray tracing algorithm, or by doing a 3D reconstruction from multiple images [36]. Double Warping is a similar approach as it requires the server to create various depth images at different viewpoints. The client runs the warping algorithm for every depth images. Each reference viewpoint represents a prediction for a future camera movement. The challenge is to select the optimal reference viewpoints in order to increase the coverage to the maximum. This can greatly reduce the interaction latency at the expense of extra network bandwidth due to the additional depth images sent to the client [37, 39, 40].

The work in [7] proposes a distributed system to reduce interaction latency. In this system, the server generates the depth images and the references background images. On the client, the background is rendered using the data from the server, and the foreground layer is rendered using the GPU.
When the user viewpoint is changed, the 3D warping is only executed on the background layer. The foreground objects are therefore directly responding to the user input and the latency is reduced to the time needed for the GPU to render and display them.

2.3 Standardization of XR

With the increasing interest in XR, the hardware in headsets has greatly improved over time. The most popular devices are being supported by well-known corporations such as Facebook, Microsoft, and HTC. Although the hardware is improving, the releases of new development kits and frameworks to develop XR applications are not progressing at the same pace. The most approachable way to develop a VR or AR application is to use a video game engine such as Unity or Unreal Engine. However, the ability to interface with the devices from the game engine depends solely on the tools provided by the manufacturer. Each of them provide their proprietary APIs which are platform-specific.

This leads to increased complexity when developing an application for multiple devices, as each targeted application will involve re-writing part of the code base. Moreover, not all provided APIs offer the same features and will therefore require further work and optimization to offer the same features on every platform. Porting an application to another game engine will also require an extensive overhaul of the code. This fragmentation increases the time and resources spent on development, thereby increasing the project’s cost, slowing the speed at which they can deploy new applications, and limiting their market reach.

These challenges arise due to the lack of common standard towards XR. Each company provides the features they deem best suited to their product, there are no strict rules. To solve the challenge of fragmentation, different groups have been looking to provide a standard approach to develop a cross-platform XR solution. The two most notable are the OpenXR open standard and the WebXR standard.

OpenXR, by Kronos Group, is a royalty-free, open standard aiming to simplify the VR/AR development process. It offers a way to develop applications for multiple platforms without the drawback of re-writing code; thus, offering a simpler way to provide cross-platform experiences [21]. Their solution offers a single common API and a single device interface, shifting the workload from a specialized support for each platform, to a single support for all platforms. Their API provides the necessary features for any XR application, such as spatial tracking, input processing and binding, and
viewpoint control. Adding a new functionality is simple due to its extensible design. Figure 4.3 illustrates the fragmentation of the XR market before and after using OpenXR.

Another way to provide a cross-platform XR experience is through a web browser which is accessible on most XR devices. Serving XR content on the web comes with its own challenges and risks, notably low-latency and security concerns. The Immersive Web Working Group is currently working on a group of standard under the name WebXR, formerly WebVR. At the time of writing, WebXR is currently a working draft, but is intended to be recommended as a World Wide Web Consortium (W3C) [16]. The WebXR Device API provides the required interface to build persuasive, comfortable, and safe experience for both VR and AR. Unlike OpenXR which provides native access to an engine, WebXR does the 3D rendering of the scene by relying on graphics APIs like WebGL (Web Graphics Library) and WebGL2.
Chapter 3

System Overview

In this chapter, we will describe the overall design of the application, including the software and hardware used. Section 3.1 will describe the scenario of the application and list the different requirements that the system should fulfill. In Section 3.2, we will describe the typical workload of our server and client. Lastly, Section 3.3 will describe the different technologies used to implement the system, and Section 3.4 will present the hardware where the system will run on.

3.1 System Design

Although the objective is to develop a working prototype of a remote rendered web application for VR, there is no specific scenario that the experience should follow. In order to provide some sort of interactivity, the simulation will allow the user to navigate around the virtual scene much like a virtual walkthrough. While this does not provide an accurate representation of what can be achieved, it does provide a fundamental base to demonstrate the potential of the application, and to test the performance of the system.

To create a prototype application that can be easily extended in the future, we have defined a set of requirements that will guide us during the development process. The requirements will affect the choice of tools that will be used to implement the system. Moreover, they will help us evaluate if the implementation was successful. The requirements are defined as follows:

- The application is usable across different headsets, and support different input schemes.

- The input system should be built in a modular way, allowing to add new scenes and interactions with relative ease.
• The system should implement a way to mitigate interaction latency.

• The system should run at least at 60 FPS on the server and client.

3.2 System Overview

3.2.1 The Server

The server in our system will have the same basic functionality as any rendering server as described in Chapter 2. Interactions are supported in the client application, meaning that their every action must update the game logic on the server side. The typical task that the server will execute in a loop are:

1. **Listens for a user input and tracking data.** Whenever the user triggers an action, e.g. pressed a button on the controller or their viewpoint has changed, the state of the virtual world needs to be updated. To match the user's action in the server, the client must send the new state to the server over the network. Once the new state is received on the server side, the server will process the data to update the game logic and state accordingly, resulting in both the user and the server's representation of the user to be synchronized.

2. **Renders the virtual scene using a virtual camera.** Once the game state has been updated, the system will need to render a new frame with the updated logic. It should be noted that VR experiences and games are usually dynamic, the game’s state will likely update regardless of whether the user changed the state. This means that the server will always be rendering new frames for the client. The virtual camera will match the position and orientation of the user, so the scene will be captured from essentially the same perspective. In this work, the camera is modified to adapt to the work in [18, 48]. The final output will render a customized cube map. Similar to a panoramic frame, a cubemap can be used to project a 360° scene around the user.

3. **Encodes the frame and sends it to the client.** The custom cubemap will be rendered into a temporary texture. The texture resolution will be quite large to match the high-resolution of current VR headsets. Therefore, the texture will be compressed using a specific video encoder. With the frame now encoded, the server can send the frame over the network to the client. While the texture is being encoded, the server will have already started the next cycle loop.
CHAPTER 3. SYSTEM OVERVIEW

3.2.2 The Client

The client application will be the only interface the user will have access to. The client will request the connection to the rendering server. Once the connection is successful and the stream is established, it can start its general update cycle which may look like this:

1. **Receive a video frame from the server.** When the video frame is received on the client, it will first need to be decoded before being displayed. Both the server and the client should support the same video codec in order to work. Once the frame is encoded, the client will be able to display the new frame to the client. Some adjustments are necessary to display the custom cube map correctly.

2. **Update the game logic.** In a VR experience, the user is highly prone to rotate their head to observe around them. The application needs to update accordingly to match the user. The local state must also be updated whenever the user presses an input on their controller. In our system, the client will have to perform a local rendering of some elements, e.g. the controllers or some user interface.

3. **Update the server with the new state.** Whenever the user changes their viewpoint or presses a button on the controller (when at least one is available), the local state changes and that change must be reported to the server. The new state will be sent to the server which will use the new information to render the next frame. As mentioned on the server description, this part may not always occur in every cycle.

3.3 Technologies

3.3.1 A-Frame

There are quite a few tools available if we want to create a VR application. Since supporting diverse devices is a requirement as defined in Section 3.1, it is necessary to use a cross-platform tool for the client application. To ensure that most VR headsets are supported, the application will run on a browser, limiting the requirements to only an internet connection. To develop the web application for the client, we will use the framework A-Frame [1] which enables the creation of WebVR/WebXR applications (uses WebXR unless unsupported) using plain HTML without any necessary installation. It runs on most VR headsets, smartphones, and even desktops, and is supported by most major browsers.
A-Frame is built on top of Three.js, a JavaScript 3D library that facilitates the creation of 3D content on a web page. Three.js rely on WebGL, a low-level JavaScript API compatible on a browser without any plugins. WebGL draws points, lines, and triangles based on supplied code, requiring a significant code base to render a simple 3D scene. Three.js will simplify the process by handling the setup that we would normally have to code ourselves, such as the scene, lights, shadows, materials, textures, and 3D math operations. Although it would be possible to create our VR application with only Three.js, it is a generic framework and is used for all purposes of 3D graphics rendering, while A-Frame is focused on XR experiences and therefore offers even more interesting functionalities than Three.js. Furthermore, A-Frame provides unrestricted access to JavaScript, the DOM API, Three.js, WebXR, and WebGL for a full control of the application. New features are easily integrated using their component systems and many community components can be found online. A-Frame is a scalable and capable framework that has been used by companies such as Google, Disney, CERN, and NASA.

3.3.2 Unity

The render server will be based on a game engine, the best-known engines for VR development are Unreal Engine and Unity. The latter was chosen as it is the most accessible of the two, Unity [46] uses C# as a scripting language while Unreal Engine uses C++. Moreover, Unity has a more exhaustive catalog of assets on their asset store, and the user community/support/documentation is arguably larger. In our context, we do not look at the support for XR that the engines provide. The client will take care of the XR interactions instead and a virtual player will be mimicking the user. This means that the server will run a standard 3D game with no XR components. Unity is always being in constant development, new features and packages are often created to offer new services. During the past year, Unity has been developing a package that provides a peer-to-peer high quality streaming solution. The broadcast takes full advantage of WebRTC, allowing to stream directly to any browser that supports it, i.e. Firefox, Chrome, Safari, and Microsoft.

The Unity Render Streaming package [47] provides three main features: video and audio streaming, as well a remote control system. Although it is a peer-to-peer system, broadcasting is only done from Unity to the browser, and streaming audio and video from the browser to Unity is not yet supported. The package provides different sample projects to offer a variety of applications, including the support of real-time ray tracing. Unity supports
different render pipelines \footnote{https://docs.unity3d.com/Manual/render-pipelines.html} which have different characteristics and performances, and are generally suitable for different target applications, games, and platforms. The standard sample project utilizes Unity's High-definition Render Pipeline which is used to create cutting-edge graphics for high-end platform. This version was first used but offered a frame rate significantly lower than the ideal 60 FPS. A new package update (version 2.0) offered a new sample project using the Universal Render Pipeline which allows to create optimized graphics and get a more balanced performance. Additionally, to reduce the latency on the client side, the custom cubemap system presented in [18, 48] is integrated to the Unity project.

\section*{3.3.3 WebRTC}

WebRTC [49] is a powerful real-time audio and video communications solution and will be the bridge between our server and client application. The Unity Render Streaming package is built on top of their own proprietary WebRTC package and offers access to its main features. On the client side, WebRTC is an open web standard that stands to offer real-time communication between browsers without requiring any external plugins. It is supported on major browsers via a regular JavaScript API and on native platform via a library. Due to its vast support, it has become a popular choice for video conferencing, gaming, and file transfer [17, 43].

To achieve low latency performance, WebRTC transmits the data through UDP by default instead of TCP [43]. This feature comes with its benefits and drawbacks. Unlike TCP, when UDP sends a packet (or datagram), it then sends the next one and does not wait to verify that the recipient has indeed received the packet. Since the overhead of error checking and correction is removed, the communication can be done rapidly and is therefore more suitable for time-sensitive applications such as video or game streaming. There is, however, a risk that datagrams are not delivered, or are not delivered in a sequential order. For a streaming service, this would mean slightly decreasing the quality or skipping some frames which could cause some short disturbance. For increased security, several protocols are added on top of UDP in order to encrypt the data.

\section*{3.4 Hardware}

This work has a few requirements tied to the hardware necessary to run the system. The client can be run on any headset that would ideally use at least
one controller to support additional interactions. The Oculus Quest was our choice as it is currently one of the most popular headsets. Moreover, it is a mobile headset, meaning that the rendering capabilities are limited and can be improved with the use of the rendering server. For the A-Frame application, the latest version has migrated to the WebXR standard which requires the content to be served over HTTPS. For this reason, the application will be hosted on Heroku\textsuperscript{2} and will be accessed using Firefox Reality 9.1.

As the application is running on HTTPS, the request for the server must also be made to an HTTPS address. The Unity server is launched locally on a laptop. The local server is then exposed to a public internet address by a secure tunnel created using ngrok\textsuperscript{3}. The tunnel will forward all of the client’s requests to our local server. Finally, the Unity Render Streaming package requires a Unity Engine version 2019.3 or higher, and necessitate a NVIDIA graphics card that supports NVCodec\textsuperscript{4}. Table 3.1 shows the main hardware specifications of the devices used.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
Hardware & Oculus Quest (Client) [31] & MSI 63VR Stealth Pro (Server) \\
\hline
CPU & Snapdragon 835 @2.45 GHz & Intel Core i7-7700HQ @ 2.80GHz \\
\hline
GPU & Adreno 540 & NVIDIA GeForce 1070 Max-Q \\
\hline
Memory & 4 GB & 16 GB \\
\hline
Refresh & 72 Hz. & 60 Hz. \\
\hline
\end{tabular}
\caption{Used hardware specification.}
\end{table}

\textsuperscript{2}https://www.heroku.com/
\textsuperscript{3}https://ngrok.com/
\textsuperscript{4}https://developer.nvidia.com/video-encode-decode-gpu-support-matrix
Chapter 4

Implementation

This chapter describes how the overall system was implemented. Section 4.1 will introduce the signaling server that will help peers establish a connection. Section 4.2 will describe how WebRTC uses the signaling server to create the connection between peers and the different channels. Section 4.3 will detail how the video frame is rendered before it is sent to the client. Then, in Section 4.4, we discuss the interfaces used to remotely control the server application from the client. Lastly, Section 4.5 will contain the information on how the client application was setup.

4.1 Signaling Server

To connect two peers through webRTC, they must first be aware of each other. A peer can communicate directly to another peer; however, it doesn’t know how to discover that peer. This discovery process is handled by a signaling server. This server will coordinate the communication between our peers, i.e. the rendering server and the client application. Henceforth, we will simply refer to the Unity rendering server as "server", and the client A-Frame application as "client".

Conveniently, Unity’s render streaming package provides a sample web application that will serve as the signaling server and a client web page. Slight modifications were made to the web application to support cross-origin HTTP request. There are a few information that needs to be exchanged during the signaling process [25, 34]:

- Session control messages used to prepare, open, and shut the communication channel, as well as to handle possible error messages.
• Connection information needed to enable the peers communicate with each other, e.g. IP, port, and session information.

• Media information such a media type and codec. These need to be agreed before the webRTC session starts.

The information is defined in a Session Description Protocol (SDP) message. Figure 4.1 showcases the different steps that occur before establishing the connection. First, the client browser will initiate the negotiation by sending an offer SDP message to the signaling web server which will in turn forward the offer to the server. The server will then create an answer based on the received offer, or it also may reject the offer. If the server accepts the offer, it will create an answer SDP message and send it back to the client via the signaling server.

![Diagram](image.com)

Figure 4.1: Overview of the signaling mechanism. Image from [46].

After the offer has been answered, the client and server must exchange information about the IP and port that will be used to communicate the data. This is done by exchanging a candidate Internet Connectivity Establishment (ICE) configuration. The peers will use the ICE candidates to query a Session Traversal Utilities for NAT (STUN) server to discover the public IP that the Network Address Translator (NAT) has allocated for the exchange [32]. In our system, the STUN server is a public server provided by Google. Generally, a number of ICE candidates are suggested by both peers until they mutually agree on which one they think would function the best.
CHAPTER 4. IMPLEMENTATION

Once both peers are aware of each other’s connection information, the connection can be set up and the different medias and data channels can be created [25]. At any point, either peer can send a new offer to update the session, e.g. to fix a network congestion. A new offer can be created only if there are no offers that have not been answered or rejected [34]. It is worth noting that the signaling server does not interact with the data in any capacity, it does not need to know the content of the messages as it mainly serves as a relay point between the two peers.

4.2 WebRTC

The signaling process is not defined in the webRTC standard; however, creating the SDP message description for the offer/answer process is handled by webRTC. Setting the ICE candidates is also handled by the specification [26]. Although the server and client access the webRTC features using a different medium, i.e. respectively the C# webRTC package from Unity and the WebRTC JavaScript API, both use a common set of interfaces.

The most important interface is the RTCPeerConnection. It represents the connection between the local and remote peer. It provides the methods to setup, maintain, and close the remote connection when it is no longer needed [26]. The RTCPeerConnection possess methods (such as: createOffer(), createAnswer(), addIceCandidate, and setLocalDescription) that will be used during the signaling process to setup the communication to the peer. Once the connection is established, new media streams and/or data channels can be added to the connection.

A stream of media content is represented by the MediaStream interface. A MediaStream object consists of a number of MediaStreamTrack interfaces where each of the tracks represents a single type of media data, e.g. video, audio, or text (i.e. for subtitles or chapter names). A media stream can be used to send data from a live or stored media (e.g. a movie). The render streaming package provides two additional components to specify a video and audio track in Unity. In every scene, a Main Camera object is added in the Unity scene. It is from this camera that the player will view the world. The camera also holds an AudioListener that acts like a microphone that listens to nearby audio sources in the scene and replays the sound to the user, this allows to create an aural experience. By adding the Camera Streamer and Audio Streamer components [46] on the camera, we specify that we want to stream the rendering result of the camera as a video track, and the audio listener’s output as an audio track. The camera streamer component can be used to set the streaming resolution; however, it
CHAPTER 4. IMPLEMENTATION

should be noted that the resolution will ultimately depend on the network quality.

At the moment, the render streaming package does not possess the ability to add a text track. If we want to exchange data between two peers, it must be sent through a network channel created using the RTCDataChannel interface. A data channel is a bi-directional channel that can transfer arbitrary data, such as metadata, game updates, and even files. An RTCDataChannel is tied to a specific RTCPeerConnection; however, a RTCPeerConnection can have multiple data channels, the maximum amount depends on the browser used.

The creation of a data channel and media stream is handled by the RTCPeerConnection through their respective methods createDataChannel() and addTrack(). Creating the peer connection, the related media stream and data channel is done by the Render Streaming component provided by the package. The component is easily edited to access all peer connection functionalities. To function correctly, the component needs the address of the signaling server to establish a connection. Once the address has been set to the correct web socket or HTTP server, the render streaming component will handle the offer/answer process, the ICE messaging, and the media stream and/or data channel creation.

On the client, the same interfaces are used to setup the connection to the server. A-Frame's architecture is also based on components; however, there is no integrated webRTC component included. Since A-Frame components are developed using JavaScript, it is simple to create a new component that will perform the same work as Unity's render streaming component. A part of the code included in the web application can be re-purposed into a new component. The only difference is that the client cannot send a media stream to the server as the render streaming package does not support this feature yet. Figure 4.2 presents the overall architecture of the system. The various components in the system will be presented in the next sections.

4.3 Camera Setup

In the previous section, we explained that the video stream is defined by the camera streamer component. While this works fine for a standard browser application, we want to integrate a way to mitigate interaction latency on the client side. For that purpose, we implemented the customized cubemap defined in [18, 48]. A cubemap will serve a similar purpose as a panoramic image as it captures a 360° Field of View (FoV), it can be used to create a surrounding environment around the user and they will be able to look
around freely with low interaction latency.

Unlike a standard cubemap which has 6 faces of equal size, the cubemap in our system will only have 5 faces. We make the assumption that the user will never see the back face of the cube and can thereby discard it. The reason we assume that the user will never see the back face is because the server will send additional information to the client to correct the cubemap’s orientation. This adjustment ensures that the front face is aligned with the user’s forward viewing direction. While waiting for a new frame and its orientation to arrive, the user may start viewing a part of the side faces (i.e. left, right, top, and bottom faces) before the cubemap is adjusted. It is unlikely that the user will turn fast enough to view the back face before the correction arrives.

Since the user will be seeing the front face most of the time and the FoV of a headset is generally limited to 95°-110° horizontally, the side faces will either be seen partially or in the peripheral vision. Therefore, we can reduce the size of the side faces, leaving more pixels in the frame to be dedicated to the front face. If the orientation is not compensated before the next frame arrives, the unrendered area out of the cubemap will appear as black. This can possibly break the user’s immersion and negatively impact the QoE. By reducing the size of the side faces, we assume that the latency of the streaming service is low enough to rapidly update the cubemap’s orientation.

Unity offers a native method to create a cubemap; however, if we want to reduce the size of the side faces, we need to crop the texture image after the rendering has been done. This leads to additional rendering that will never be seen. To avoid the unnecessary rendering and avoid possible distortions,
we use a different setup that consists of 10 cameras, 5 per eyes to support stereoscopic rendering. Each camera is turned to face towards a specific direction in order to capture its designated face of the cubemap. To increase the resolution of the front face, we enlarged every face to the requested resolution. As the video frame size remains the same and the front face is located at the center of the frame, the enlarged side faces will no longer fit entirely inside the frame; thus, the out-of-bound area will not be rendered. In order to only render the correct parts, the projection matrix of each side camera is modified to solely capture the portion of its FoV that is adjacent to the front face. When the rendering pipeline begins, each camera will render the scene from its perspective into a temporary texture. Subsequently, these temporary textures will all be copied into a common texture to output the final cubemap for each eye into a single texture. The resulting texture (containing all 10 faces) will then be encoded by the webRTC plugin before sending it to the client.

However, the implementation in this work requires additional work. In [18, 48], the resulting frame is being sent over the network alongside side supplementary control data (i.e. orientation corresponding to the video frame, and size of the front and side faces). Since it is sent over TCP, the correct order for each video frames and control data is maintained. However, webRTC uses UDP which does not guarantee that the data is received in the same order as it was sent [43]. The control data is being sent over the RTCDataChannel, but since the data is paired with a specific frame that is sent through the RTCPeerConnection, we need to implement a way to synchronize the data and the frames received on the client.

In order to achieve some sort of synchronization, an additional texture is
added in the final cubemap output texture. The texture fills the gap located in conjunction to the left and bottom face of the left eye. This texture is filled with a simple color value that is based on an identifier that loops continuously between 3 and 50. This value is then interpolated between 0 and 1 to create a grayscale color that will be set to the new texture’s color. Figure 4.4 showcase a single frame containing both cubemaps and the frame identifier texture.

![Figure 4.4: Final rendering output. The red and blue borders represent the cubemap for left and right eye respectively. The green border depicts the grayscale value used as a frame identifier.](image)

Once the identifier has been defined, all the information needed by the client is known. Therefore, the data can be sent through the data channel while the final texture is being copied and encoded. The client will store the information and will be able to process the frames and data to correctly compensate the cubemap’s orientation. When the texture is being encoded, the next frame can already start the rendering process.

### 4.4 Remote Input

It is crucial to keep the server up to date with the user’s data to ensure a smooth experience. In order to update the server state, the client will send messages containing a new data state through the RTCDataChannel. The message will typically hold information such as the user’s position and orientation, or a new input state. Processing a message is done in the channel’s `onMessage` event callback where it will be handled by the **Remote Input** interface. With this interface, the render streaming package supports browser inputs coming from a connected input device, including a mouse, keyboard, touchpad, and gamepad. The remote input interface leverages
Unity’s new Input System package [45], this system provides a standardized way to correctly and conveniently map the received inputs across different devices. For instance, a gamepad device can be used to map the inputs coming from a PlayStation, Xbox, or Switch controller. Whenever an input device is physically attached to the computer, the input system will, if the device is supported, choose which layout matches the device; a layout represents a combination of input controls (e.g., a button, key, or stick) and describes the layout of the memory holding the input data. Once the layout has been selected, the input system will instantiate an `InputDevice` and fill it with the appropriate controls described in the layout. The InputDevice can be used to easily access the internal data such as the current state and value of a control. This makes it easier to define which actions are executed when a certain input is triggered. However, the controllers in our application are physically located on the client side and the server is therefore unaware of them. Fortunately, the input system offers a way to manually create a virtual input device and update its state by using the input system’s `QueueStateEvent(Device, DeviceState)` function.

We previously mentioned the devices supported by the render streaming package; none of them are VR specific, which means that we must create our own device. It should be noted that the input system does offer some XR specific interfaces; however, they cannot be used as virtual devices as they do not provide a way to update the state manually. Moreover, the provided interfaces are platform dependent; therefore, if we want to enable cross-platform support, we need a device that can be used to support different controllers’ inputs. By designing our own device, we will also be able to add additional controls to support new controllers in the future.

For a single user, the server must listen to the inputs coming from at most two controllers, i.e., the left and right hand controllers. Although most controllers differ in ergonomy, design, and input labels, they possess a relatively similar input layout. For instance, a controller for a 6DOF headset typically has a trigger and grip button, a few standard buttons to access the system’s options and other functions, and a thumbstick. We will create our device to match this description. This similarity will allow us to use a generic notation when describing our inputs. Other than the inputs, our device will also store the controller’s position and rotation.

The first step to create our custom device is to define its state structure. We create a new structure that will derive from the `InputStateTypeInfo` interface. In this structure, we will define all the different data formats of our controls. The input system typically already has existing data formats that can be used to describe our data. We can use this at our advantage by using their simple `InputControlAttribute` annotation. The attribute
is used to identify and represent a single input control, and can be used to provide additional details on the input, e.g. the name, display name, aliases, layout, usage, and default value. At runtime, the attribute will cause the input system to create the related input control which will be used to read the value of the input state. Each input control must have a unique name and layout, and a display or short name can be added to get more readable names for display purposes, e.g. an Oculus Touch and a Vive Knuckles controller both have two vertically aligned buttons with different names, we will generalize the nomination by setting the display name to Primary and Secondary button. The most important attribute is the layout, as it will define what type of input control will be generated. We can, for instance, set the layout to be a button or stick depending on the input. If a layout is not specified, the input system will automatically try to refer to the value type of the field or property when generating the input control. An attribute can be reapplied multiple times to create numerous input controls, this facilitates the mapping process of our buttons as we can declare a single integer variable that represents a bit field in which every bit defines the state of a different button.

Figure 4.5 shows a simple example of a device state with different input control attributes. The primary, secondary, and stick buttons are represented by the buttons field, the state of the button is either up or down, and is represented by their respective bit defined in the input control attribute. The data for the thumbstick is stored in a 2D motion vector. The two values of the vector describe the horizontal and vertical direction respectively. If a new controller reaches the market and has additional buttons, they can easily be mapped by adding another attribute and specifying the related name and bit.

```csharp
// ButtonControl
[InputControl(name = "stickPress", displayName = "Thumbstick Press", layout = "Button", bit = 0)]
[InputControl(name = "primary", aliases = new[] { "A", "X", }, layout = "Button", bit = 1)]
[InputControl(name = "secondary", aliases = new[] { "B", "Y" }, layout = "Button", bit = 2)]
public uint buttons;

// Vector2Control
[InputControl(name = "thumbstick", layout = "Stick", usage = "Primary2DMotion")]
public Vector2 thumbstick;
```

Figure 4.5: Example of a basic device state

In the same file, we can create a few methods that will allow us to update the state more conveniently, e.g. a method `WithButton()` that takes a button identifier and a value, and returns a new state with the updated value. The button identifier mentioned is an enumeration that associates a button with
CHAPTER 4. IMPLEMENTATION

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>Source (e.g. keyboard, mouse, A-Frame)</td>
<td>byte</td>
</tr>
<tr>
<td>input</td>
<td>Input type (i.e. button, stick, or camera)</td>
<td>byte</td>
</tr>
<tr>
<td>btn</td>
<td>Button ID specifying the bit location</td>
<td>byte</td>
</tr>
<tr>
<td>evt</td>
<td>Event type (state up or down)</td>
<td>byte</td>
</tr>
<tr>
<td>hand</td>
<td>Controller identifier (right or left)</td>
<td>byte</td>
</tr>
<tr>
<td>axis</td>
<td>Direction along an axis of a stick</td>
<td>float</td>
</tr>
<tr>
<td>position</td>
<td>Position of the headset or the controller</td>
<td>Vector3</td>
</tr>
<tr>
<td>rotation</td>
<td>Rotation of the headset or the controller</td>
<td>Quaternion/Vector3</td>
</tr>
</tbody>
</table>

Table 4.1: Description and format of the data extracted from the data channel’s message.

its corresponding bit location in the bit field. The same enumeration is stored on the client side to send the correct button identifier when the button is pressed.

With the state finished, we must now implement the device layout that will represent the virtual input device that will be used to read the value of each input state and map them to their designated actions. To create a device layout, we need to create a class that inherits from `InputDevice` and add an `InputControlLayout` attribute. The controller layout attribute is used to define a readable display name and more importantly, to set the state to our custom device state. In this class, all the variables that will be used to retrieve a specific input state are defined. After the variables have been listed, they need to be assigned to the correct input control as specified in the state. This is done by overriding the `FinishSetup()` function and mapping each property with its respective input control using `GetChildControl<ControlType>(ControlName)`. Finally, the device needs to be registered as a new input device using the input system’s `RegisterLayout<Layout>()` function. The device is now fully configured and can be used as a virtual device. The device creation process is repeated to create a new device that will store the state of the user’s headset (position and orientation). Table 4.1 lists the different information that our devices will process from the client’s messages.

The server now has a way to remotely keep track of the state of the user’s headset and controllers. The virtual devices (one headset and two controllers) are then added to the remote input component to assign the devices to the user connection. Whenever a message coming from the client user is received from the data channel, the remote input will update the state of the appropriate device. Finally, we add a `PlayerController` script to update the
controllers and the camera in our scene. The script will have references to all three virtual devices and will be able to access their current state. We can then implement an update function that will check on every cycle the state of each device, and call a specific function whenever an input has been triggered. Since the application will be a virtual environment walkthrough, the player controller will only have two actions that are defined. The first action will correct the position and orientation of the camera to match the headset device state. The second will respond to thumbstick inputs to move the player in desired direction.

4.5 Client Setup

The A-Frame application can be developed without installing any external libraries. The application starts from a plain HTML file where a script tag pointing to the content delivery network of the latest A-Frame build is added. With this setup, we have, as mentioned in Subsection 3.3.1, full control of the application using A-frame’s custom components, JavaScript, the DOM API, Three.js, WebXR, and WebGL [1]. Additionally, another script for webRTC (adapter.js [50]), is added to avoid issues due to specification changes and prefix differences in new webRTC releases. The webRTC functionalities are implemented using JavaScript and follows the same procedure as the server side. The client will start the offer/answer process until the connection is established, then the **RTCpeerConnection** will setup the *MediaStream* for the video and audio stream, and the *RTCDatChannel* for the arbitrary data exchange.

To display anything on A-Frame, it is necessary to first create a scene, a camera and renderer will be created automatically if not specified. The scene will contain all the entities and assets in the application. Here, an entity is a container that will have components attached to it, therefore every object in a scene is an entity, e.g. the player is an entity comprised of the components: position, camera, and controller. Since the background scene will be rendered on the server and streamed to the client, our scene will only render a few important components on the foreground.

First, we must add a camera entity to our scene and unlike the default camera, additional components will be added on it. Whenever a user enters VR mode, the camera will turn into two cameras to switch to stereoscopic rendering. Each camera will render the scene from the perspective of one of the eyes. The standard behavior is that every other objects in our scene will render to both eyes. By adding the **stereocam** component developed by
oscarmarinmiro \footnote{https://github.com/oscarmarinmiro/aframe-stereo-component}, this allows us to show different objects to each eye. With this component, each eye will be assigned to a different layer mask, and when another entity is set to one of those layers, it will only be seen by the eye on the same layer. If the entity is one neither layer, it will be rendered to both eyes. We will see later where this is useful. The second component is used to constantly track the position and the orientation of the camera. When the position and orientation has been calculated, the information is sent through the data channel to update the server’s viewpoint.

The next entities are the controllers, there will be one for the right hand, and another for the left hand. Depending on the device, the controllers may not exist, or maybe only one is required. The controller options provided by A-Frame are device-specific, so to ensure cross-platform usage, a custom component is necessary. Much like the provided controller options, the custom component will leverage the browser’s GamePad API which is used to access signals emitted by a gamepad or game controller. There is no magical solution, to support every controller, we need to listen to every event generated by each different controller that we want to support and map each input with the callback function that they should execute. The controllers will be rendered locally by A-Frame, the 3D models used are provided by A-Frame, but any external 3D model can be used as long as its format is supported.

Each input is matched to a unique identifier that will be used to identify the appropriate input to update on the sever. The identifiers used on the client must also be present on the server to ensure correctness. Each input also holds a state, unlike a standard button which has an up and down state, some VR controllers additionally have buttons which are touch-sensitive, i.e. they have 2 supplementary states: touch start and touch end. This would mean that some buttons may have 4 different states and it would not be possible to store them as bits on the server side. To handle these touch event, we do not treat them as being another state of the button. Instead, we treat them as being a separate button entirely. On the Oculus Quest, the trigger is touch sensitive which means that instead of having one button with four states, we would have:

- A \textit{trigger} button with the standard \textit{up} and \textit{down} state.

- A \textit{triggerTouch} button with an \textit{up} and \textit{down} state which matches the \textit{touch end} and \textit{touch start} event respectively.

Whenever an input is registered, the client must notify the server of the action by sending a message through the data channel. When an input
is pressed, the client will send: a source identifier (to indicate the source of the update), the input type (button, thumbstick, or headset), the button identifier, the event state, the controller identifier (left or right), the rotation, the position, and axes if the source is a thumbstick. All the information will be stored in a data view which provides a low-level interface to read and write into a raw binary data buffer. The data buffer will then be sent to the server and processed by the remote input system.

The next entity in the scene is a virtual button that will serve as a scene selection option. The main component on this entity will establish the connection to the remote rendering server, and connect to the data channel and media stream. Components can be assigned parameters when instantiating the entity; in this case, the address of the rendering server is specified. This leaves the possibility to add new virtual buttons to connect to different rendering servers and explore different scenes. The user will then have to choose which scene they wish to connect to by pointing to the appropriate button with the controller or cursor (if no controllers are available) and clicking it.

The last entity is a skybox, which can be seen as a giant sphere with the textures rendered on the inside to serve as a background. The skybox can be assigned a simple color or a texture from an image or video. The scene will have an asset of type video and its source will be set to the video feed of the server when the media stream is received. This asset will serve as the texture for the skybox. Typically, to display an image correctly, it should be a panoramic 360° image. Although Unity can use panoramic images and standard cubemaps on a skybox, A-Frame handles standard cubemaps differently and has a component dedicated for that. However, the layout of the cubemap received from the server is different, therefore we can use the skybox component to display it. To display the custom cubemap properly on the skybox, we need to change the way how the skybox is rendered by using a custom shader. A-Frame supports the usage of OpenGL Shading Language (GLSL) shaders. A GLSL shader can be written using the ShaderMaterial provided by Three.js. Although we need only one texture, we need two different shader type: a vertex and a fragment shader.

1. The vertex shader is the first shader to run. It will calculate and likely manipulate the position of each separate vertex (3D point in a model). It can receive variables (attributes) that are only accessible in the vertex shader, as well as other variables (uniforms) that can be accessed by the vertex and fragment shader. This shader can also perform additional computations to pass data (varyings) to the fragment shader.

2. The fragment shader is the second shader to run. It receives the
**varyings** data from the vertex shader and the shared **uniforms** data, then it set the color value of each individual pixel rendered on the screen.

Unlike attributes and varyings, uniforms variables have the same value for every vertex. We will use uniforms to pass our data, i.e. the video texture, the orientation of the cubemap, size of the side faces, and other options, to the GLSL shader. The uniforms will be defined using JavaScript, which we can leverage to conveniently update it when new data is received from the server. Editing the uniforms variable will automatically update the value in the shader. The most important variable that must be updated is an array of cubemap orientations. Since the server sends the orientation of a specific frame through the data channel, they must be stored and passed to the shader to select the correct orientation of the of video frame on the skybox. The array has a length of 50, which represents the maximum frame identifier on the server (starts over at 3 when it reaches the maximum).

As the messages containing the orientation and other data are received in an out-of-order basis, we must ensure a synchronization mechanism. The GLSL shader will also decode the color of the additional texture added to the bottom left section of the frame and extract the frame identifier which will be used to select the appropriate orientation in the array. There are different ways to retrieve a specific color pixel value outside of the GLSL shader as well; however, this solution proved to be the most efficient one. Although the pixel value may sometimes be incorrect (the value may be one/two values higher or lower), the frame identifiers are incremented, therefore it would select the orientation matching the previous or next frames which should not cause any consequential jumps. It should also be noted that this is not the ideal solution, since the color extraction is done inside the vertex shader, the computation of the same pixel color value is executed for each vertex in the skybox. Moreover, the array must be iterated over until the correct orientation is found as arrays does not provide a mechanism to directly access one of its members without a constant index. Ideally, it should only be computed once since it will always yield to the same result.

The cubemap layout is divided into two halves to provide a stereoscopic rendering, therefore the shader will only project one half of the video frame onto the skybox. Another skybox with the same shader is added to the scene to project the remaining half of the video frame. As is, the two skyboxes will be rendered one on top of the other, ultimately causing the last rendered skybox to hide the previous one. This means that both eyes would be seeing the same half of the video frame on the skybox. It was mentioned earlier that the camera holds a **stereocam** component that would add each eye into
a separate layer mask. To solve this issue, each skybox must be added on the same layer as the eye it must render to. Consequently, only the corresponding skybox will be rendered on each eye, solving the overlapping issue and enabling a stereoscopic rendering of the video frame.
Chapter 5

Evaluation

In this chapter, we evaluate the performance of our system, focusing mostly on the client side application. To evaluate the performance, we will run a sample scene and measure various metrics to assess the success of the system. Section 5.1 will describe the testing procedure used. Section 5.2 will present a brief explanation of the server's performance, then, Section 5.3 will provide a performance analysis of the client application to show if it fulfills the real-time performance requirements for VR.

5.1 Testing Procedure

To test the performance of our system, we will measure the metrics that we deemed the most important. Since our focus is on the client application, we will not perform a thorough performance test on the streaming server. Instead, the server will only be tested to see if it can support a video stream at a steady frame rate of 60 FPS. The server will calculate the frame rate and output the result when the server closes. On the client side, we will have a large number of metrics to analyze. First, we will analyze the frame rate as computed by the A-Frame application. Then, we will check the headset's hardware metrics to see how the application affects the hardware usage. Finally, we will check the statistics provided by the RTCPeerConnection and analyze the data that is exchanged with the server.

The flow of the test will be executed as follows:

- The server is started, it will calculate the current frame rate, and will listen for the client connections.
- The client application is launched on the browser and requests a connection to the rendering server.
• Once the connection is established and the video feed is visible to the user, we will use the headset and the controller to look around and navigate through the scene.

• After one minute, the metrics are collected and the applications are closed.

Depending on the metrics tested, we might repeat the test under different conditions. For instance, we will often test the system using different streaming resolutions to see how the quality affects both parties. The tests are performed using the hardware presented in Table 3.1. The Unity and A-Frame application will both calculate the frame rate and output it. To analyze the webRTC statistics and the headset’s hardware usage, we will use external tools.

To analyze the webRTC statistics, we could manually check the RTCPeerConnection and extract the data. A few libraries are available to retrieve a more readable set of data, we will use a tool that is provided by Google Chrome. By using a Chromium-based browser (on the Oculus Quest, we will use the Oculus Browser), we can access the webrtc-internals tool by accessing the internal URL: chrome://webrtc-internals/. On another tab, we will have our client application running, when the connection with webRTC starts, the webrtc-internals will start recording the metrics. The tool will let us access information regarding the session information, data channel, and the video and audio tracks.

To gather the headset hardware metrics, we will use the OVR Metrics Tool [9] provided by Oculus. The metrics tool is an application that will be running on the background. It can display a head-up display overlay that provides a real-time performance graph and information. The metrics can also be exported into a file when a session has concluded, i.e. when the browser closes. It can provide information about the average frame rate, usage of the GPU/CPU, battery temperature, and more.

5.2 Server Performance

To test the server, it is important to see how the frame rate is affected during runtime. Once the testing session ends, the server will export the accumulated logs. The log includes the last frame’s render time which can be used to calculate the average frame rate. For testing purposes, we compiled four different versions of the server, each running at a different resolution, Table 5.1 displays the different resolutions used. Henceforth, the qualifier
Table 5.1: Streaming resolution size.

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Video Frame size</th>
<th>Front cube size</th>
</tr>
</thead>
<tbody>
<tr>
<td>UHD</td>
<td>2880 x 1440</td>
<td>1024</td>
</tr>
<tr>
<td>HD</td>
<td>2160 x 1080</td>
<td>900</td>
</tr>
<tr>
<td>SD</td>
<td>1440 x 720</td>
<td>560</td>
</tr>
<tr>
<td>LD</td>
<td>720 x 360</td>
<td>300</td>
</tr>
</tbody>
</table>

will be used to mention a specific resolution size. Ideally, the Ultra High-Definition (UHD) would have a resolution that matches the headset’s display; however, the current highest resolution achievable with the render streaming package is 2880 pixels wide and 1440 pixels high. The render streaming package does not appear to fully support higher resolutions when using the universal render pipeline. Note that the width will always be twice the size of the height. Therefore, the UHD will be the main resolution to test. The High-Definition (HD) resolution will have lower size to compare how the size affects the performance on both server and client. In some cases, we may use a Standard Definition (SD) and Low Definition (LD) resolution if the results of the previous two are not sufficient. Furthermore, the server should ideally have a frame rate that matches the refresh rate of the display device, in our case it is the Oculus Quest which has a refresh rate of 72 Hz. Therefore, the server should ideally run at 72 FPS. Unfortunately, the render streaming package did not function with that frame rate, likely due to limitations on the webRTC package. As a result, the server will run at 60 FPS and our tests will see how the streaming service will affect the rendering time.

The application is virtual environment walkthrough, therefore we need to add an environment that the user can navigate through. The default scene on Unity only provides a singular plane which is not interesting. Ideally, the environment would be of high detail and quality to offer great visuals to the user. To that end, we chose an asset already owned by the author on the Unity Asset Store. The Sci-Fi Facility asset¹ made by Triplebrick is a highly quality environment of a base station on the moon. The models and textures are highly detailed and are stated to be perfect for virtual reality, which fits our target application. Figure 5.1 showcases one of the different rooms in the station, with the project running on the High-Definition Render Pipeline. The asset provides a version using the universal pipeline that our

¹https://assetstore.unity.com/packages/3d/environments/sci-fi/sci-fi-facility-116916#description
project uses; therefore, the graphics will look slightly less impressive as the pipeline focuses more on performance.

Figure 5.1: Demo of the Sci-Fi Facility asset. Image from the asset store page.

The tests on the server revealed that the server can maintain a steady frame rate most of the session. In Figure 5.2, we can observe that the server can handle 60 FPS a majority of the time. Note that the average FPS is calculated based on the time it took to render the last frame. In UHD resolution, the server frame rate decreased around the time the connection with the user was established, but it quickly recovered and maintained the maximum frame rate through the rest of the session. In HD resolution, the frame rate lightly decreased around the time the user disconnected. Since both resolutions managed to mostly preserve 60 FPS, the lower resolutions were not tested since it would not give us any substantial information.

We can observe that the connection to the user can impact the performance of the server; however, the effect does not deteriorate the performance drastically. Since the maximum degradation is less than four frames, the impact is not likely to be perceived by the user.

5.3 Client Performance

The performance of the client application will be the main focus of the analysis since it will help us evaluate the user's QoE. As mentioned earlier, we will perform an analysis on the A-Frame application, the webRTC statistics, and the Oculus Quest's hardware.
5.3.1 Frame Rate

To evaluate the performance on A-Frame, we use the statistics component provided by A-Frame which will display a few performance metrics on the UI. The most important metric is the frame rate, the rest of the metrics includes the number of triangles, shaders, textures, and entities, which are not important since their values are relatively low. A-Frame aims to provide up to 90 FPS with the WebVR API. Ideally, since the Oculus Quest has a refresh rate of 72 Hz., A-Frame will manage to aim for a steady 72 FPS. At the very least, it should run at least at 60 FPS to fulfill the basic VR requirement. We will use the browser Firefox Reality 9.1 as it currently provides the best experience. With this version, A-Frame defaults to the WebVR API as it does not yet fully support the WebXR API.

Figure 5.3 presents the results obtained by testing the client and the server at all four resolutions. At the beginning of the session, A-Frame can more or less reach 72 FPS when the scene is loaded. However, once it connects to the server, the frame rate drops drastically in all four cases. As the frame rate dropped, the total number of frames in one session is also decreased. At UHD resolution, the client struggles to keep the frame rate above 40 FPS, dwindling as low as 33 FPS which is less than half of the targeted frame rate. At HD resolution, the frame rate increases to 40-50 FPS and would at time decrease under 40 FPS. Since neither resolution succeeded in maintaining at least 60 FPS, additional tests using the SD and LD resolution
Figure 5.3: Client average frame rate at all tested resolution.

were conducted. Note that these resolutions should not be used as the textures become pixelized, they are only used as we want to see if 60 FPS can be achieved by lowering the quality. Surprisingly, neither achieved a steady frame rate over 50 FPS. A performance drop occurred in both cases, the same decline was also noted on the hardware metrics which is presented next. The overall frame rate improved at a lower resolution, but the effect appears to be minimal when running below SD resolution.

The remote stream greatly impacts the frame rate of the application. However, projecting the video texture onto the skybox does also reduce the average frame rate. In this case we are referring to the shader which requires additional computation, the pixel reading process is also repeated multiple times in the vertex shader, despite the fact that the result is the same. Moreover, the processing is done twice since the shader runs on two skyboxes to offer a stereoscopic rendering. An extra test with the UHD version was performed to see the impact the shader has on the application. In this test, the texture was simply projected onto two skyboxes using the default behavior. Figure 5.4 shows the frame rate improves considerably, the client manages to maintain over 55 FPS for the majority of the time after the connection was established. The average frame rate throughout the session is 59.6 FPS, which is a considerable increase compared to the version with the shader which runs on average at 42.4 FPS. Although the frame rate is higher, the performance is not ideal, we will detail later why that is the case.

Although we have seen that the shader and the remote stream impact
the frame rate, it is necessary to check the hardware performance to evaluate any issues it may cause the application. The OVR Metrics Tool was used during the same session that evaluated the frame rate achieved on A-Frame. Therefore, we will be able to see how the hardware performed during that particular session. There are numerous metrics available, we will consult most of the important metrics recommended on the Oculus developers page [9]. First, we will check the frame rate measured and compare it to the versions measured on A-Frame. As seen in Figure 5.5, the frame rate is more or less similar to the one estimated on the browser, the same performance losses are visible on both figures.

Although this does not give us a better idea of the performance of the application, the metrics tool provides us a better metric to evaluate the application’s performance. One of the metrics provided is the stale frame count, which is different from the frame rate count. Before explaining what a stale frame is, it is important to understand that the rendering pipeline on Oculus devices is different. Oculus added a system called Asynchronous Time-Warp to their headsets, this system expects the last application-rendered frame to be displayed at a specific time [9]. On the Quest which has a refresh rate of 72 Hz., the frame should be rendered in less than 13.89 ms to ensure 72 FPS. If the frame is not ready on time and misses the expected display time, it will resort to using the last frame which is outdated or ”stale”. Thereby, the same frame would be displayed twice. In normal games, missing
Figure 5.5: Average frame rate of the headset.

a frame can pass unnoticed; however, with VR, missing a frame could have serious repercussions on the user’s QoE as the world would not feel synchronized. To mitigate this effect, TimeWarp takes the latest rendered frame and alters it before displaying it. Once the scene is rendered, the frame is submitted to TimeWarp system along with the corresponding viewpoint, and the frame will be modified at the last moment to match the viewpoint. This is beneficial only when a frame is missed, instead of the display being locked to the last rendered frame, TimeWarp will update the frame so it seems to match the latest viewpoint even if the frame is stale. Whenever a frame is missed, the stale frame count goes up, and the frame rate goes down. However, they are not the inverse of each other. The GPU and CPU can work in parallel, once the CPU requests the GPU to render the scene, the CPU can already work on the next frame. Therefore, the application can run at a full frame rate and have stale frames.

Figure 5.6 displays the stale frame count. At all four resolution, the stale frame count is almost always at the maximum of the 72. This means that the TimeWarp system is interrupting the application 72 times a second to check whether a frame was rendered, and every time the answer was negative; thereby, the last rendered frame was displayed again. After missing two consecutive frames, the CPU will block the request for the next frame and let the GPU finish rendering the frame, but, in the meantime, the previous frame that was reused will be displayed again before displaying the currently rendered frame. This means the same frame can be shown 2-3 times in
a row, causing additional latency. Therefore, the stale frame count is a more important metric than the FPS when evaluating the user’s QoE. It is important to note that the stale frame count for the version with the standard skybox without the shader (i.e. Figure 5.4) also capped at 72 stale frames throughout the session while the average frame rate cycled around 50 FPS. Even if the frame rate was higher the stale frame count did not decrease.

Since we know that the frame rate drops since the CPU and GPU cannot render a frame under 13.89 ms, we can check the other metrics provided by the OVR metric tool to determine on what side the bottleneck is. One of the metrics provided is the application GPU time, i.e. the time spent by the GPU rendering a frame. The time it takes can indicate if the application is CPU or GPU bound. If the time is higher than 13.89 ms, it signifies that the application is GPU bound. However, as seen in Figure 5.7, the render time is less than 13.89 ms. This means that the application can potentially be CPU bound, we will need to look at other performance metrics to evaluate that. Next, we can check the hardware usage for the CPU and GPU.

In Figure 5.8, the results for the UHD version are displayed on the left in blue, and the HD version is displayed on the left in orange. In all four graphs, the brown line represents the GPU or CPU level, where the maximum is level 4. Whenever the application does not hit the targeted frame rate, the GPU or CPU will increase the level to adjust the clock speed.

The CPU and GPU levels for both resolutions indicate that the application runs at the maximum level after the scene has been loaded, which is not
an ideal scenario since it may overheat the device. Looking at the usage, the
CHAPTER 5. EVALUATION

GPU utilization is very high after the remote stream was rendered on the screen, frequently passing past 90% yet never reaching 100%. This is good as it means the GPU has a high throughput; however, exceeding 90% may cause performance issues due to scheduling [9]. Ideally, the system tries to keep the utilization around 75% [33]. The system is not likely GPU bound since the utilization is not peaking at 100% and the application GPU time previously mentioned is low; therefore, there is a possibility of a CPU bottleneck since we are not reaching the full frame rate. By checking the bottom row of Figure 5.8, we can see that the CPU is also running at the maximum level in both resolutions. The utilization is not as high as the GPU, but it fluctuates between 75% and 90% on average. The GPU and CPU are used extensively which will drain the battery and heat the device.

Since both the GPU and CPU are almost fully utilized, it should ensure that the application would run at full frame rate or at least at 60 FPS. Loading the scene already pushes the CPU and GPU to the maximum level, but a high frame rate and low stale frames are still maintained until the connection to the streaming service is established. Once the stream is connected, the frame rate drops and the stale frame count peaks; therefore, processing the video stream is causing significant delay which would cause the frame to miss the 13.89 ms time limit.

We will check a few metrics provided by the webRTC stats API; however, we will be using it on a Chromium based browser since it provides additional statistics, and a more convenient way to collect and export the data. Therefore, the results are recorded on another test run and will not be tied to the previous metrics. The statistics provide various information about events (ICE candidates, offer/answer SDP, and channel creation), and details about data channels and media streams. We will first check a few statistics related to the video stream track. To collect the data, we will run the application at UHD, IID, and LD resolution. We discard the SD resolution as the results are similar to the LD version, and because we will be mainly focusing on the higher resolution since they provide the quality that would normally be used for the application.

Figure 5.9 presents the important statistics regarding the video frames decoding. In the first part (a), we can observe the number of frames received each second. As the server manages to run at 60 FPS, the client receives more or less the same number of frames. The amount may vary depending on the network condition. After the frames are received, they must be decoded before being displayed. The next part (b), showcases the number of frames that was decoded, the number appears to mostly fluctuate around 60 frames. Significant drops occur on the UHD and LD resolution, these are mostly due to the number of frames received being reduced at the same time. Both
Figure 5.9: Number of frames decoded per second.

The UHD and HD versions appear to handle a steady decoding with the HD version being slightly more stable and the LD being less steady.

Approximately 60 frames are being decoded each second, yet the application cannot maintain a high frame rate since the frame render time exceeds the 13.89 ms limit to ensure 72 FPS. The statistics provide the number of frames decoded and the total decode time; thereby, we can calculate the average time used to decode a frame. The last part (c) displays a frame's average decoding time. The decoding is remarkably high, on average the UHD version would take around 17 ms to decode while the HD version would take 11 ms and the LD version around 9 ms. At higher resolution, adding the decoding time to the application’s GPU time would already exceed 13.89 ms without even considering the additional CPU processing time. The LD version would only leave a narrow window of time for the CPU to work when aiming for the target render time. This high decoding time would explain why the application’s frames are not rendered on time, causing the stale frame count to rise and the average frame to decrease.

Looking at the collected data, reducing the video frame resolution under the HD variant would not grant any major benefits. The slight increase
in FPS would hardly justify the decrease in quality. The major reduction obtained in SD and LD resolution is the decrease in the application’s GPU render time which cuts the time in half compared to the UHD resolution. In other metrics, lower resolutions do not have any major increase in performance, the hardware is being utilized at the same extent as higher resolutions, and the decoding time only decreases slightly when comparing the average time on the LD and HD versions.

5.3.2 Bandwidth

![Image](image-url)

Figure 5.10: Video track bandwidth consumption.

For future reference, we will also look at the network bandwidth consumption. This will help us compare this system with other implementations once a stable version with a high frame rate is achieved. Figure 5.10 showcases the average network bandwidth used by the video stream service in MBit/second. The UHD version has a higher frame resolution and therefore uses on average 7-15 Mb/s more than the HD version. Surprisingly, the LD version uses slightly more bandwidth than the HD version, the encoder used by Unity’s webRTC package may be more efficient in higher resolutions. The number of packets received is also proportional to the Mb/s received. Reducing the quality from UHD to HD does reduce the bandwidth requirement; however, reducing it below the HD resolution seems to slightly increase the bandwidth usage.

In Figure 5.11, the bandwidth is displayed for both incoming and outgoing data. The bandwidth is shown in Kbit/second and is pulled from the UHD
resolution. Since the data channel is another channel and has no correlation to the video stream resolution, the other resolutions were not shown as they would be similar. The received Kbit/s are sent from the server, the messages are sent on each update cycle; therefore, the incoming bandwidth is more or less steady. Regarding the outgoing data, the application is always sending the information regarding the headset’s position and orientation. The spikes occur when we are using the controller to move the viewpoint’s position, the new input state is sent along with the headset’s data. The fluctuations also vary depending on the frame rate, a lower frame rate application will update less frequently than a higher one, causing the data to be sent less often.
Chapter 6

Discussion

The results of the performance provide us with some valuable insights. In Section 3.1, we listed a few requirements regarding the application. The first was the cross-platform support, which we have enabled by using a browser as the client application. Although, the evaluation was conducted on an Oculus Quest, an early version of the updated remote input system was successfully tested on the HTC Vive and the Vive controllers. The second requirement focused on the modality or the input system. The custom virtual devices and custom controllers developed in this work can easily be extended to support other inputs not yet supported. On the server side, the new inputs also need to be added to the custom player controller where we can easily develop new actions to perform when the inputs are triggered.

The remaining requirements are crucial for the user’s QoE. The third specified that the system must implement a mechanism to mitigate interaction latency on the server side. To that end, we used the work in [18, 48] to modify the camera to render a custom cubemap that will be projected around the user, thereby reducing the latency of rotational movements. Although the mechanism is functional, it does not compensate for the poor performance on the client side. As the last requirement states, the client and server should run at least at 60 FPS to fulfill the basic VR requirement. Although we were not successful in running the server at 72 FPS (i.e. to match the refresh rate on the Oculus Quest), we were able to maintain a steady 60 FPS using a complex and high-quality scene. However, the client application suffered from a significant performance loss. The video decoding process appears to take a significant amount of time, causing the rendered frame to go stale and the frame rate to drop. Reducing the resolution of the video frame does not sufficiently improve the performance of the application. The performance loss is also due to the complexity of the shader. Ideally, extracting the frame identifier can be done outside of the shader to avoid
repeating the same process multiple times.

To run at full frame rate, it will be necessary to reduce the decoding time and optimize the shader in order to reach the 13.89 ms time limit. However, even if further optimization is made and the application runs at the maximum frame rate, it is crucial to also decrease the stale frame count. Even if the application runs at 72 FPS, if it has 72 stale frames, the application will be rendered on the display with extra latency which will degrade the QoE.

In the current state, we have demonstrated that we could build a VR cross-platform application that uses a remote server as a main source of graphics rendering. However, the frame rate reached on the client side does not reach the expected performance. The low frame rate and the added latency from the stale frames can cause the user to suffer from motion sickness. As long as the frame rate does not improve, the usability of the application will be greatly limited.

6.1 Limitations

The evaluation reported in this work has a few limitations. The most important limitation is the lack of tested headsets. The Oculus Quest was used as it was the only headset the author possessed. Although, the selected headset can be considered representative of its target group, mobile headsets in general possess less powerful hardware and are the most likely to benefit from a remote rendering server. Moreover, the application was tested as a virtual walkthrough, the results are therefore not generalizable for other types of applications with additional interactions. Furthermore, the system was tested using a free hosting service for the client and on the author’s computer for the server. Ideally, the server should be executed on a dedicated server as well. As the client did not reach the expected performance, this work did not study the perceptible latency and comfort of the application. Once the frame rate issues has been fixed, it will be necessary to conduct user tests to truly evaluate the QoE.

Although we have a full control of the client application using A-Frame, the packages used in the server side do not allow a complete access to their content and forces us to depend on the feature they provide. For instance, the frame rate, maximum resolution, and decoder configuration depends on the webRTC package. Since the packages are still currently in development, a future version could improve the performance of the client or server, and offer new functionalities that the application could benefit from.
6.2 Future Work

Since the application does not perform as well as expected, there is a lot of room for improvements. Some optimizations will be necessary in order to improve the frame rate. Since the decoding process appears to be the main reason the frames are not rendered on time, future implementations can focus on optimizing the encoding performance or editing the webRTC package to test codecs other than the H.264 codec (AVC) used in this implementation. The render time should be low enough to support a full frame rate with ideally zero stale frames to ensure low latency. Moreover, the shader could be optimized to lower the necessary computation and remove the repeated pixel extraction process.

Even without connecting to the remote streaming service, the application already pushes the hardware to the maximum level. This remains true even when using the default example scene provided by A-Frame. A higher utilization level may cause the device to overheat and the battery to drain quicker. Testing another VR game like Vader Immortal: Episode 3 showed that the game well optimized and does not use as much resources as our application. Therefore, switching A-Frame for another cross-platform framework like OpenXR could be less demanding on the hardware.

Other than improving the frame rate and the headset performance, the latest version of the render streaming package supports streaming to multiple users. The system was modified with a single user in mind, but the it can be easily adapted to support additional users (the package support a maximum of 5 devices). However, the quality may be reduced for each additional user connected. Moreover, the server uses Unity’s universal render pipeline to get a balanced performance. For higher graphic quality, we could revert to the high-definition pipeline to obtain stunning graphics. This would require running the server on a more powerful machine to ensure a high frame rate.

Finally, we implemented the solution presented in [18, 48], but there are other existing solutions that have similar ambitions. Combining their solution to our work could prove worthwhile if we desire to achieve the best performance possible. Furthermore, the hardware used and the network quality greatly influence the performance of the application as well, but, as viewed in Chapter 2, each application will perform differently and there is no unique solution to fit every system.

Chapter 7

Conclusions

The goal of this thesis was to design and develop a cross-platform VR application that leverages a rendering server to offer a high-quality experience to the user. The main goal for the application was to improve the developer and user experience by simplifying the development process while still providing the same high-quality experience to users with different headsets. To achieve this, the thesis presented the problems and the background study in Chapter 2. In Chapter 3, we presented the different components of our system, and listed the selected technology that will be used in the implementation. In this work, we used a browser-based solution to provide access to the application using only an internet connection. The graphics rendering were executed on a Unity game server and the real-time media and audio communication was handled by the WebRTC standard. The detailed explanation of the implementation was presented in Chapter 4, followed by a thorough performance evaluation.

Although the underlying system we wanted to implement is functional, the performance achieved is not sufficient to provide a good user experience to mobile headsets. The frame rate achieved at high-resolution streaming did not meet the 60 FPS requirement to provide a fluid playback. Moreover, the video decoding process is too time-consuming, causing the frames to go stale, thereby adding additional latency when displaying new frames. This degrades the user experience and increases the risk of the user to suffer from motion-sickness. Although the system is under-performing, the bottlenecks of the system have been listed in Chapter 6, along with the limitations of this work and suggestions for future improvements. Before the application is suited for real-life cases, the shader and the video decoder will need to be properly optimized to provide the frame rate and latency requirements necessary in a VR application.
Bibliography


[23] Levoy, M. Polygon-Assisted JPEG and MPEG Compression of Synthetic Images. 8.


[35] Schmalstieg, D. The Remote Rendering Pipeline. MANAGING GEOMETRY AND BANDWIDTH IN DISTRIBUTED VIRTUAL ENVIRONMENTS, 92.


[45] UNITY TECHNOLOGIES. Input System | 1.0.0. Available at https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/api/index.html, version 1.0.


