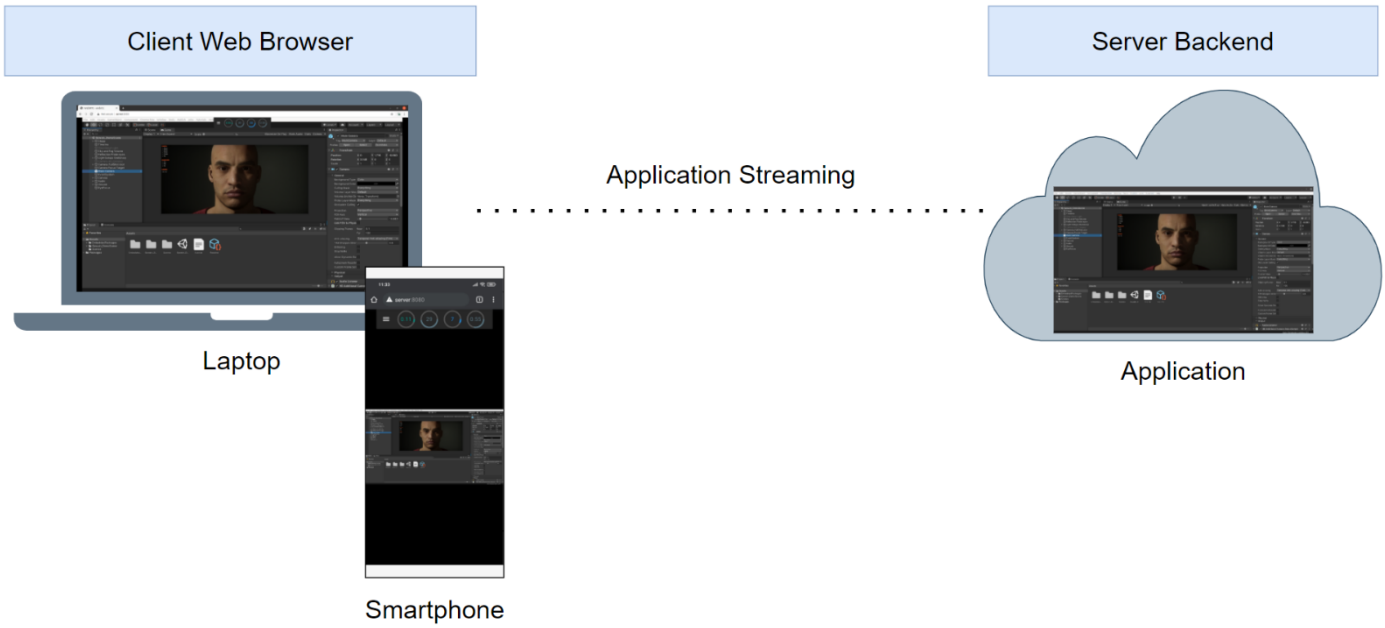


3D Visualizations in Web Based Environment

Ivo Marinho
Mestrado Integrado em Engenharias de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2020

Orientador
Verónica Orvalho, Professor Auxiliar, Faculdade de Ciências do Porto

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____

Abstract

We live in an era where there are multiple devices with different kinds of operating systems and hardware. Some of those devices have limitations in physical size and power consumption, making them incapable of delivering a 3D graphics rendering experience comparable to high-end desktops. Furthermore, many applications that render intensive graphics are unable to run on mobile platforms directly, due to incompatibilities in both hardware and software. This issue can be addressed with the idea of a web-based 3D application that uses streaming to display 3D content. The heavy 3D graphics rendering computation runs on a powerful server and the results are streamed to a web-based client.

This thesis presents a fully functional *remote streaming web-based 3D visualization application*, that will enable any device to render interactive 3D graphics in real-time. The application only requires a web browser to work and is not limited by the hardware/software and the physical size of the used device. Our application takes advantage of real-time communication technologies, like WebRTC, and multimedia frameworks, like GStreamer, to stream 3D content through the web browser. We also study the performance limitations of native rendering 3D content on mobile devices, by evaluating the number of objects various mobile devices are capable of rendering, concluding that mobile devices are less adequate to render a high number of polygons. In our experiments, the server completed the rendering, on average, 10% faster than our most powerful smartphone, and rendered 46 300 more objects.

Resumo

Vivemos numa era onde existem múltiplos dispositivos com diferentes categorias de sistemas operativos e equipamento físico. Alguns destes dispositivos apresentam limitações no seu tamanho e consumo de energia, o que faz com não sejam capazes de reproduzir uma experiência de gráficos 3D comparável com um computador de topo. Além disso, muitas aplicações que fazem *render* intensivo de gráficos, não são capazes de serem executadas diretamente num dispositivo móvel, devido às incompatibilidades em ambos, equipamento físico e *software*. Este problema pode ser resolvido, com a ideia de uma aplicação 3D baseada em *web* que utiliza *streaming* para visualizar conteúdo 3D. Devido à quantidade de recursos necessários, o *render* de gráficos 3D, é efetuado num servidor, e os resultados são posteriormente transmitidos para um cliente em *web*.

Esta tese apresenta uma aplicação para visualizar transmissões remotas de conteúdo 3D baseada em *web*, que faz com que seja possível qualquer dispositivo renderizar gráficos 3D interativos em tempo real. A aplicação apenas requer um navegador web para funcionar, e não está limitado pelo equipamento físico ou *software* e o tamanho físico do dispositivo utilizado. A nossa aplicação utiliza tecnologias de comunicação em tempo real, como WebRTC, e *frameworks* de multimédia como o GStreamer, para transmitir conteúdo 3D através de um *web browser*. Vamos estudar as limitações de desempenho da renderização nativa de conteúdo 3D em dispositivos móveis, através da avaliação do número de objetos que vários objetos são capazes de renderizar, chegando à conclusão que os dispositivos móveis são menos adequados para renderizar números altos de polígonos. Nos nossos testes, o servidor completou a renderização, em média, 10% mais rápido do que o nosso melhor *smartphone* e renderizou mais 46 300 objetos.

Contents

List of Tables

List of Figures

Listings

Acronyms

2D Two-dimensional

3D Three-dimensional

API Application Programming Interface

AVC Advanced Video Coding

CPU Central Processing Unit

CSV Comma-separated values

CUDA Compute Unified Device Architecture

ES Embedded Systems

FBX Filmbox

FPS Frames Per Second

GB Gigabyte

GCP Google Cloud Platform

GHz Gigahertz

GLSL OpenGL Shading Language

GPU Graphics Processing Unit

HTML5 Hypertext Markup Language revision 5

HTML Hypertext Markup Language

HW Hardware

ICE Interactive Connectivity Establishment

ID Identifier

JS JavaScript

LTS Long-term Support

MPEG Moving Picture Experts Group

ms Milliseconds

Mbps Megabits per second

NAT Network Address Translation

NVENC NVIDIA Encoder

OS Operative System

OpenCL Open Computing Language

OpenGL Open Graphics Library

QoS Quality of Service

RAM Random-access Memory

RDP Remote Desktop Protocol

RGB Red, Green and Blue

RTCP Real-time Transport Control Protocol

RTP Real-time Transport Protocol

RVS Remote Visualization Server

SDK Software Development Kit

SDP Session Description Protocol

SSH Secure Shell

STUN Session Traversal Utilities for NAT

SW Software

TCP Transmission Control Protocol

TLS Transport Layer Security

TURN Traversal Using Relays around NAT

UDP User Datagram Protocol

UI User Interface

URI Uniform Resource Identifier

VDI Virtual Desktop Infrastructure

VM Virtual Machine

VNC Virtual Network Computing

VTU Video Transcoding Unit

WebGL Web Graphics Library

WebRTC Web Real-Time Communication

Xpra X Persistent Remote Applications

Chapter 1

Introduction

This chapter provides an overview of the thesis by presenting and defining a web-based real-time streaming 3D visualization solution. We start by describing the motivation, the overview, the objectives, the contributions, and to finalize the organization of the chapters.

Since 2007, with the first release of the iPhone [1], the percentage of the population that is actively using a smartphone has been increasing exponentially, wherein 2021 according to Statista, approximately 48.34% of the population have a smartphone [37]. The increase in different types of mobile devices available, such as smartphones and tablets, has been changing the way people interact with their computing devices. Although the computational resources available on mobile devices has been improving throughout the years, they are more focused on lightweight graphics rendering, and due to the limitations and restrictions of physical size and power consumption, they are still far from being able to provide similar Three-dimensional (3D) rendering capabilities as a desktop. With the rapid advances in hardware capabilities and network transmission, streaming technologies have been a practical solution to a variety of tasks like video streaming, web conferencing, and cloud gaming.

1.1 Motivation

With the recent increase in computational resource demand, there has been a development in the availability of cloud-based services. Normally, graphics-intensive workloads do not run smoothly on low-power Central Processing Unit (CPU) and Graphics Processing Unit (GPU) devices, but, by utilizing a cloud-based service, we can run computational-intensive visualizations on any device. Most of the existent solutions are currently either proprietary or require additional software to be installed, and are not compatible with all the major platforms and devices. The most common open-source streaming solutions used to remotely control and visualize content have not seen a significant leap in performance in several years. Popular tools like Virtual Network Computing (VNC) and Remote Desktop Protocol (RDP) are still the most used remote

[1] More information at: https://en.wikipedia.org/wiki/IPhone_(1st_generation)

desktop solutions over the internet, but they are not well suited for the modern web browser, since they usually require you to install additional client software or plug-ins.

There are solutions like Guacamole, noVNC, and X Persistent Remote Applications (Xpra), that we studied with more detail in the state-of-the-art section 3, which already provide a web-based experience, but unfortunately they don't meet the performance and latency requirements for graphic-intensive workloads, like rendering high-fidelity 3D content. In the table 1.1, inspired by the one found in the Google cloud solutions article "GPU-accelerated streaming using WebRTC" [15], we can compare the most popular open-source streaming tools capabilities.

Most of the modern GPUs can encode real-time streams in hardware, which massively improves the performance of streaming technologies by decreasing the CPU usage and bandwidth required to deliver content. Some existent tools like Guacamole and noVNC don't support hardware encoding, which makes it hard when it comes to delivering good performance at a high resolution, other tools like Xpra have support for hardware encoding, but still suffer when it comes to optimizing bandwidth for graphic demanding tasks. One possible solution to the described problem is to stream content to a web browser by using Web Real-Time Communication (WebRTC) and GStreamer which enables you to perform low-latency communication in real-time and access any available GPUs for hardware encoding or decoding of real-time streams, through the web browser.

Table 1.1: Open-source streaming capabilities

Solution	Web based	Hardware encoding	Hardware decoding
VNC	No	No	No
RDP	No	No	No
Guacamole	Yes	No	`HTML5 Canvas`
NoVNC	Yes	No	`HTML5 Canvas`
Xpra	Yes	Yes	`HTML5 Canvas`

1.2 Overview

In this thesis, our focus was to implement a web-based 3D visualization application that took advantage of real-time streaming technologies to provide high-fidelity graphics-intensive content through the web browser, while maintaining good performance metrics, such as, latency and framerate.

With that in mind, we developed a full-stack application, which, as we can observe in the application overview figure 1.1, is composed of three main components, the client, the communication, and the server. The first component is the client web browser, which serves

the purpose of visualizing the content that is being transmitted from the server. Due to the importance of having multiple device compatibility, the client does not require any additional software or plug-in installation and the User Interface (UI) was implemented to be compatible with a variety of windows resolutions. The next component is communication, for the client and the server to be able to establish peer connection between each other we utilized the WebRTC protocol. With WebRTC we were able to include real-time communication capabilities in our application. The final component is the server, which contains the streaming engine and the application we want to stream. The streaming engine was developed using the GStreamer multimedia framework and its respective Python binding, by implementing a pipeline that was capable of capturing an application window, encode it through the usage of a hardware-based encoder, and send the video stream by utilizing WebRTC.

Figure 1.1: Application Overview

1.3 Objectives

The overall goal of this thesis is to provide a way for multiple people to visualize high-fidelity 3D content, without having restrictions on devices, operative systems, and available computational resources, a good example would be to use this system for visualization and interaction of 3D characters with a dense number of polygons and high resolution textures. We want the user to be able to stream any application with high resolution and low latency. As a result, we have a client and a server, where the client is a web browser page that is capable of receiving the video stream by establishing a connection with the server, and the server hosts the streaming engine, the application we want to broadcast, and also acts as an Application Programming Interface (API) to listen to a possible client request, such as, changing window resolution and video bitrate. For the experiments and tests, we created a fully functional web-based 3D application, using an efficient Web Graphics Library (WebGL) framework called three.js [24], which allows the usage of any device with a WebGL compatible web browser [45], without the need to install any additional software, enabling the evaluation to be performed in mobile devices and the streaming server in simultaneous.

To accomplish the described goals we have the following objectives throughout the thesis:

1. Study of the current state of the art for the existing frameworks and streaming technologies.
2. Develop a real-time web-based client that connects with the server and visualizes the streamed content.

3. Implement an API that receives the client requests and processes them on the server.
4. Develop the server streaming engine.
5. Perform tests regarding the streaming quality against native rendering on mobile devices.

1.4 Contribution

In the present thesis we have made the following contributions:

1. **Web Based Live Streaming Server:** By utilizing state-of-the-art technologies like WebRTC and GStreamer, we developed an open-source streaming application, that can stream and remotely control any 3D application with low latency and high rendering quality.
2. **Web Client:** We implemented a web-based client application, that communicates with a server, and can visualize streamed content from a cloud server.
3. **Evaluation tests:** Made experiments and tests that demonstrated the current limitations of mobile devices, by comparing the number of polygons that mobile devices are capable of rendering against a streaming server.

1.5 Outline

The present thesis is divided into six chapters, each describing a variety of work that was done.

Chapter 1 introduces our work by describing the context of our thesis and the problem we want to solve, what and why we want to do it, and how we achieved our objectives.

Chapter 2 gives a background by presenting the foundations of our work, giving theoretical aspects and background material that we need, to understand the terminology and expressions used in the following chapters.

Chapter 3 discusses the current state of the art related to our thesis, describing the existing frameworks, technologies, and related work previously done by others.

Chapter 4 describes the contribution of the work done and how our streaming application was implemented by outlining the different development phases for the server and client.

Chapter 5 presents the experiments and tests that were done to evaluate the benefits and performance results of our streaming solution.

Chapter 6 concludes the project thesis and describes the possibilities for future work.

Chapter 2

Background

This chapter presents the foundations needed to fully understand our work. We establish the context of our thesis, by explaining all theoretical aspects and background material that is used in the remaining chapters. We will start by explaining Web Graphics Library (WebGL), following by describing protocols such as Real-time Transport Protocol (RTP), Session Description Protocol (SDP) and Web Real-Time Communication (WebRTC).

2.1 WebGL

WebGL is a JavaScript Application Programming Interface (API) for rendering Three-dimensional (3D) and Two-dimensional (2D) graphics within any compatible web browser without the need to use additional plugins. It is developed and maintained by the Khronos Groups and is the standard for 3D graphics on the web. WebGL enables the developers to take advantage of the computer's graphics rendering hardware by only using JavaScript and a web browser [35]. Due to being based on Open Graphics Library (OpenGL), WebGL provided numerous advantages, such as [44]:

- WebGL API: the WebGL API is based on the most common and widely accepted 3D graphics standard.
- Cross-platform: WebGL is capable of running on any operating systems and devices, with the only limitation being the availability of a compatible web browser.
- Hypertext Markup Language (HTML) Content Integration: WebGL is closely integrated with HTML content, layering with other page content, interacting with other HTML elements, and makes use of the standard HTML event handling mechanisms.
- Dynamic Web Applications: the technology was developed with web delivery as the main focus. WebGL has OpenGL for Embedded Systems (ES) as its foundation, but adaptations were made with specific features to improve the integration with web browsers.

In this thesis, we used WebGL to develop a cross-platform 3D web application for executing experiments and tests. Because of the low-level nature of the WebGL API, we utilized a JavaScript framework to aid in the development. In the state-of-the-art chapter 2, we reviewed the most popular WebGL frameworks and performed an analysis to identify which one is the most suited for our needs.

2.2 RTP

RTP provides end-to-end network transport operations suitable for applications that require the transmission of real-time data, such as interactive audio and video or simulation data, over multicast or unicast network services [38]. The data transport is secured through the usage of a control protocol Real-time Transport Control Protocol (RTCP), which enables the monitoring of the data delivery in a scalable way to large networks. Additionally, some core definitions, that are used throughout the thesis, are the following:

- RTP payload: the payload contains the data transported by RTP in a packet, such as audio samples or compressed video data.
- RTP packet: RTP packet is a data packet that contains a fixed RTP header, a possibly empty list of contributing sources, and the payload data. An illustration of the RTP header, can be seen in the figure 2.1.
- RTCP packet: RTCP packet is a control packet that consists of a fixed header, in part similar to that of RTP data packet, followed by structured elements that may differ depending on the RTCP packet type. An illustration of the RTCP packet format, can be seen in the figure 2.2.

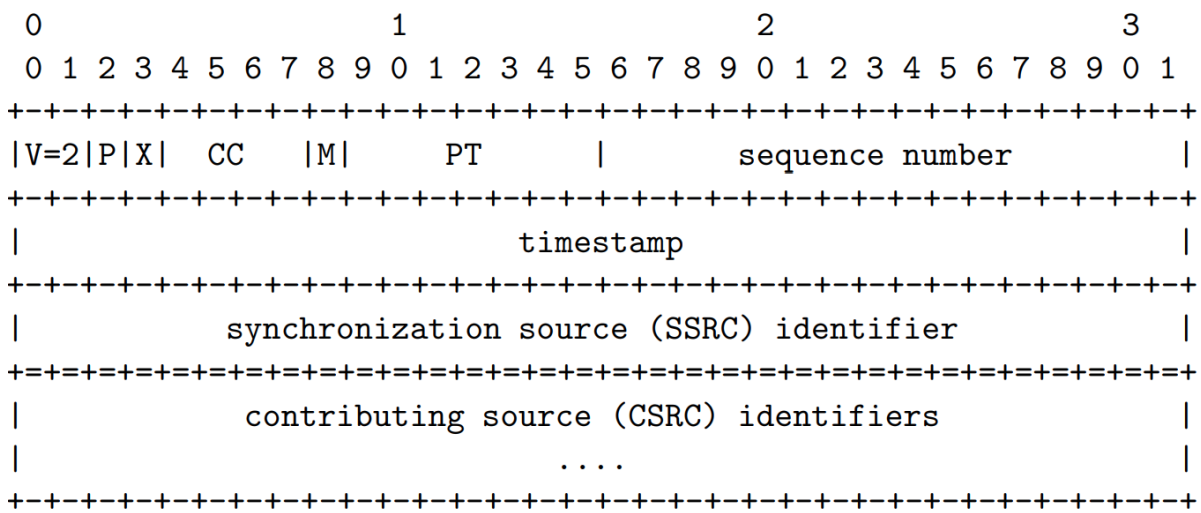


Figure 2.1: RTP Header

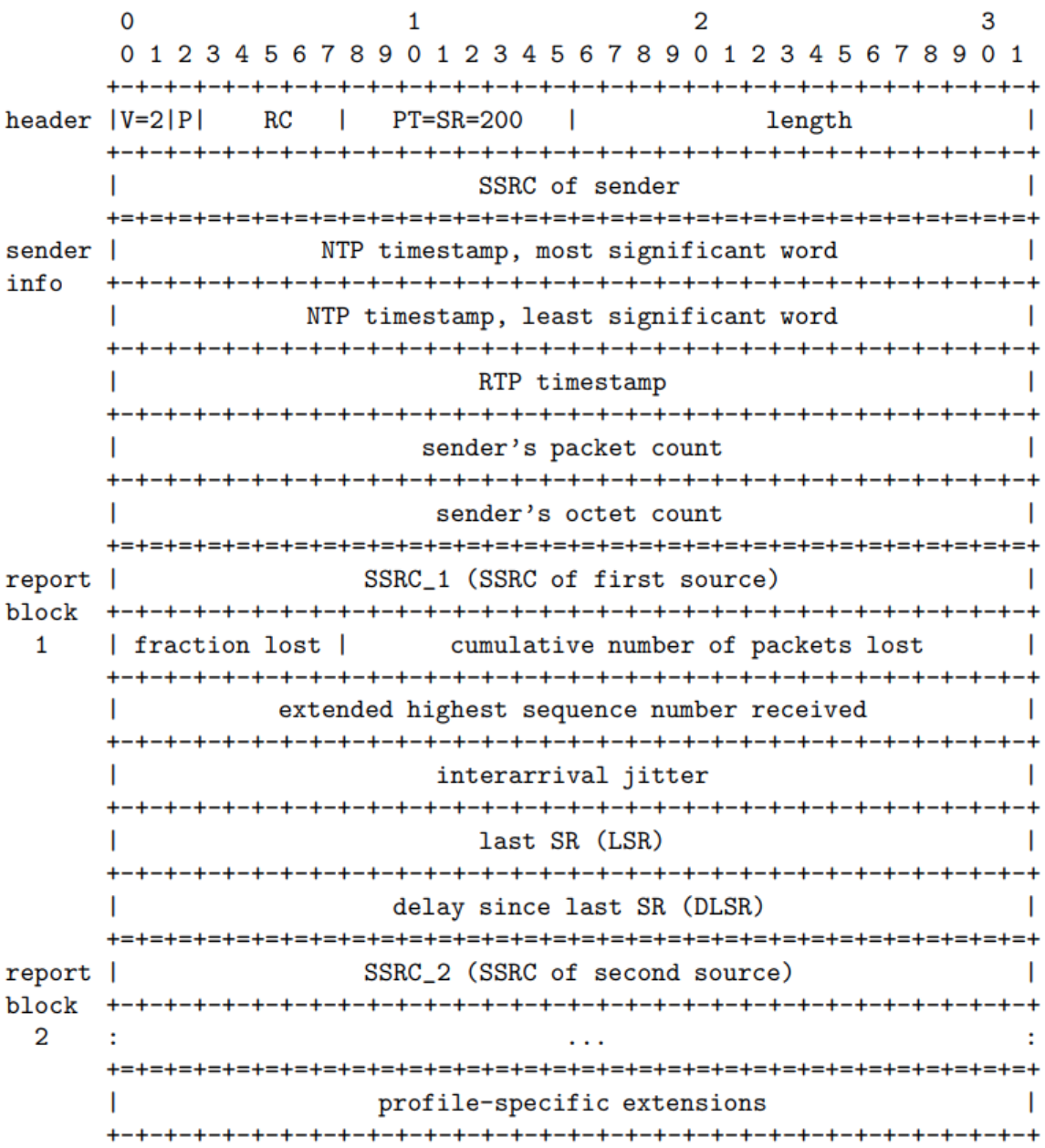


Figure 2.2: RTCP Packet Format

2.3 SDP

SDP supplies a way to convey information about media streams in multimedia sessions to enable the recipients of a session description to engage in a session [3]. The protocol provides a standard representation for information, such as media details, transport addresses, and other session description metadata that is required of the participants, which are necessary when initiating multimedia teleconferences, voice-over-IP calls, or streaming videos. In general, a SDP session description consists of the following information:

- The purpose of a session and its respective name.

- The time that the current session is active.
- Which media the session is composed of.
- Additional information needed to receive those media, such as addresses, ports, and formats.

For our thesis, the most relevant is the media and transport information, for that, the SDP session descriptions include the following media information:

- Type of media, for example, audio and video.
- Transport protocol, such as the previously discussed RTP.
- The format of the media, like H.264 video or Moving Picture Experts Group (MPEG) video.

2.4 WebRTC

WebRTC is the standard when it comes to peer-to-peer real-time communications on the web browser [25]. It enables Web applications and sites to exchange in real-time, audio, video, and data. The set of technologies that comprise WebRTC makes it possible to share peer-to-peer data, such as video-calling applications and screen sharing, without requiring the user to install additional plug-ins or third-party software.

The technology consists of numerous interrelated APIs and Protocols [53]. The protocol is a set of rules for two WebRTC agents to negotiate peer-to-peer secure real-time communication. The API enables the developers to utilize the protocol through the usage of JavaScript and must provide accordingly a wide set of functions, like connection's management, encoding/decoding negotiation capabilities, media control and firewall, and Network Address Translation (NAT).

The wide range of capabilities that WebRTC provides can enable us to implement a web-based streaming solution. In the following sections, we describe the three main processes necessary to establish peer-to-peer communication, and some technologies from the WebRTC protocol, that are relevant to understand the topics presented in our thesis.

2.4.1 Signaling

At the start, a WebRTC agent has no means to know who will communicate with and what they will communicate about. For multiple agents on different networks to be capable of locating each other, the discovery and media format negotiation needs to take place in a process called signaling. Signaling uses the previously discussed SDP to deliver the message. Each message is

conceived of key/value pairs that consist of a list of media descriptions. The SDP message can be used to exchange three types of information [1]:

- Session control messages: this is required to initialize or terminate the communication and can also be used to report errors that occurred during the session;
- Network configuration: when communicating outside our local network, signaling can give information about your computer's IP address and port.
- Media capabilities: determine what codecs and resolutions are compatible with your browser and the browser it wants to communicate with.

2.4.2 Connecting

After the two agents successfully communicate with the signaling service, they will be able to attempt to connect. Due to the constraints of the real-world networks, such as two agents not being on the same network, protocol restrictions, and firewall rules, WebRTC uses Interactive Connectivity Establishment (ICE) technology and Session Traversal Utilities for NAT (STUN) or Traversal Using Relays around NAT (TURN) servers, to provide a solution.

- ICE: ICE is a protocol that allows for the establishment of a connection between two agents, by finding the best way possible to communicate. Each ICE candidate publishes the way they can be reached, by providing information about the transport address of the agent. The protocol then determines which are the best pairing of candidates [2].
- STUN: STUN is a protocol that was created to aid in dealing with NAT traversal, and it can be used to help an endpoint that is behind an NAT to determine its respective address and port [3].
- TURN: One of the disadvantages of STUN is that it only works when direct connectivity is possible. When there are two NAT types that are incompatible or different protocols, a TURN server might be required. TURN is a dedicated server that supports STUN and is used to relay traffic if direct peer-to-peer connections are unsuccessful [4].

2.4.3 Communicating

The communication between two WebRTC agents is possible once the connection processes are completed with success. With WebRTC it is possible to receive and send unlimited amounts of audio and video stream while being codec agnostic. To communicate, two existing protocols

[1] More information at https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Signaling_and_video_calling
[2] More information at https://www.rfc-editor.org/rfc/rfc8445.html
[3] More information at https://www.rfc-editor.org/rfc/rfc8489.html
[4] More information at https://www.rfc-editor.org/rfc/rfc8656.html

can be used: RTP and RTCP. As we saw previously in the RTP section 2.2, the protocol was designed to enable the delivery of real-time video stream, and RTCP is focused on the communication of metadata.

In addition to media communication, another key feature of WebRTC is the existence of data channels for data communication. The Data Channel API was created to provide a generic transport service, enabling multiple clients to exchange bi-directional peer-to-peer data, with low latency and high performance.

Chapter 3

State of The Art

In this chapter we discuss the current state-of-the-art, starting with the analysis of the currently available Web Graphics Library (WebGL) frameworks, following with the most common remote desktop technologies, streaming multimedia frameworks, and cloud gaming services, concluding with the review of related work and articles.

In the last few years, there has been an evolution in the development of new technologies for the creation of Web-Based Three-dimensional (3D) Visualizations. Browsers have gotten more powerful and have become capable of delivering complex applications and graphics. The most popular browsers have adopted WebGL, which enables you not only to develop Two-dimensional (2D) applications but also 3D applications, by using the capabilities of the Graphics Processing Unit (GPU). WebGL-based technologies and frameworks are still dependent on the client and are limited by the computational resources available from the device that is being used for the rendering. One solution for this issue is to use a streaming-based service, where the 3D content is being rendered on a server, and frames are streamed through the web browser. To have a concrete notion of what is the current status of these concepts, we explore some of the work that has been done and compare it with what we want to accomplish with the dissertation.

3.1 WebGL Frameworks

Programming with WebGL directly, is very time-consuming and complex. Because of that, multiple frameworks have been developed to facilitate the development of Web-Based 3D Applications. Frameworks abstract the complexities of WebGL and help you become more productive and simplify the learning process. Below we talk about some of the most popular frameworks.

3.1.1 three.js

Three.js, in November 2020, was the most widely used JavaScript framework for displaying 3D content on the web [24]. The three.js library provides a very easy-to-use JavaScript (JS) Application Programming Interface (API) based on the features of WebGL. The JS framework allows you to create and display animated 3D applications in the web browser, without the need to learn the WebGL details [7]. Three.js provides numerous features and APIs that can be used to develop 3D scenes and includes a variety of examples and modules with very useful additions like for example an Orbital Camera and a Filmbox (FBX) Loader, which makes developing a 3D Visualization tool very easy and hassle-free. Some key features of the three.js framework are the following [6]:

- Built-in vector and matrix operators.
- API wrapper implementation of cameras, lights, materials, shaders, objects, and common geometries.
- Import and Export utilities (Supported formats can be seen in table 3.1).
- Great documentation and examples.

3.1.2 babylon.js

Babylon.js is a similar framework to three.js in that it allows for the creation of 3D applications and video games for the Web [2]. The babylon.js is a JavaScript framework developed using TypeScript, it can be used to create 3D applications and video games. The framework was built with the core focus around simplicity by adding tools like:

- Playground: is a live editor for Babylon.js WebGL 3D scenes. You can write code and simply see the result instantly.
- Sandbox: viewer tool, that allows you to import supported files and display them in the browser (Supported formats can be seen in table 3.1).
- Node Material Editor: allows you to avoid writing complicated shader language core, by instead replacing it with an interactive node-based User Interface (UI). It can also be used to create procedural textures, particle shaders, and post-process effects.
- Particle Editor: you can create and configure particle systems with the click of the mouse.
- Sprite Editor: enables you to create, control, and save sprite systems.
- Skeleton Viewer: makes it quick and easy to debug rigging issues inside your scene.
- Inspector: allows the analysis and debug of a scene.

With all the features presented, Babylon.js is one of the simplest frameworks to work with and learn, but that can also add some unnecessary overhead, which can produce worse performance results. We analyze the differences in performance from Three.js versus Babylon.js in 3.1.9 "Three.js vs Babylon.js".

3.1.3 PixiJS

PixiJS is a free open-source rendering library that allows the creation of interactive graphics experience, cross-platform applications, and games without having to directly deal with the WebGL API and facilitates multiple device and browser compatibility. One of the PixiJS strengths is the performance, but contrarily to three.js and babylon.js, it only allows for 2D graphics rendering [14].

3.1.4 PlayCanvas

PlayCanvas is an open-source game engine, that uses HTML5 and WebGL to run games and other interactive 3D content in any mobile or desktop browser. PlayCanvas is a fully-fledged game engine similar to Unity or Unreal Engine but for the web browser. The engine comes with an editor, that enables a drag-and-drop environment for building 3D scenes with the integration of physics, animations, audio engines, and a scripting interface. Although open-source PlayCanvas has some paid features, like for example private projects and team management features [36].

3.1.5 Clara.io

Clara.io is a full-featured cloud-based 3D modeling, animation, and rendering software tool that runs in the web browser. The tool shares similarities with Maya and Blender, it lets you create complex 3D models, make photorealistic renderings, and share them without the need to install any additional software. Clara.io supports Three.js and Babylon.js, but it is more suited to 3D modeling instead of the development of 3D applications [11].

3.1.6 Unity

Unity, in November 2020 was the world's leading platform for creating and operating interactive, real-time 3D content [47]. It provides tools for developing games and publish them to a wide range of different devices. Although Unity is not a framework like for example three.js, it is still relevant to evaluate since it has a WebGL build option. Some key features of Unity that differentiate it from the others are the following:

- Real-time 3D creation for everyone: you can create 2D or 3D scenes, animations, or cinematics directly in the Unity editor

- Create once, deploy anywhere: you can build your content once and deploy across 20 different platforms, one of them being WebGL. The WebGL build option allows Unity to publish content as JavaScript programs, using HTML5/JavaScript, Web Assembly, WebGL rendering API and other web standards to run Unity content in a web browser.

Compared to other game engines, Unity has a more user-friendly experience by providing a scripting API in C# and an easy-to-use interface with drag and drop functionalities.

3.1.7 Unreal Engine

Unreal Engine is the world's most open and advanced real-time 3D creation tool developed by Epic Games [9]. Unreal Engine is a real-time engine and editor that contains photorealistic rendering, dynamic physics, and effects. When it comes to the language used, the engine utilizes C++ and their Blueprints Visual Scripting, which is a complete gameplay scripting system based on the concept of using a node-based interface to develop within the Editor. Similar to Unity, Unreal Engine also supports HTML5 projects by utilizing the Emscripten toolchain from Mozilla to cross-compile UE4's C++ code into JavaScript [10].

3.1.8 Comparison between different frameworks

To find the most optimal framework for our project, we can look at the table 3.1 which compares different frameworks' features. The table was based on the "List of WebGL frameworks" found in [54].

3.1.9 Three.js vs Babylon.js

Three.js and babylon.js are the two most popular frameworks when it comes to developing web-based 3D applications. To find which one is the most adequate for our project, we evaluate the performance of both. Babylon.js with its playground and node material editor has the advantage when it comes to the creation of 3D applications easier and faster, but, as we will observe further if performance is a priority three.js might be a better option. Karlsson and Nordquist [22] made a performance comparison of three.js and babylon.js about rendering Voronoi height maps in 3D, although such rendering is not particularly useful to our project, their analysis is still relevant to our decision. In their analysis they ran performance tests for GPU, Central Processing Unit (CPU) and Random-access Memory (RAM):

- GPU: when it came to GPU performance they found that three.js was a better performer than babylon.js.
- CPU: similar to GPU results, three.js had less CPU usage when compared to babylon.js.

Table 3.1: Comparing framework features

	Scripting	Modeling	Animation	Import	License
three.js	JavaScript	No	Yes	glTF, DRACO, FBX, OBJ, STL, MMD, PRWM, PCD, PDB	MIT
babylon.js	JavaScript, TypeScript	No	Yes	OBJ, FBX, STL, Babylon, glTF	Apache License 2.0
PixiJS	JavaScript	No	Yes		MIT
Clara.io	JavaScript, REST API	Yes	Yes	OBJ, FBX, Blend, STL, STP	Freemium or commercial
PlayCanvas	JavaScript	No	Yes	DAE, DXF, FBX, glTF, GLB, OBJ	MIT (engine), proprietary (cloud-hosted editor)
Unity	C#	Yes	Yes	FBX, OBJ	Proprietary
Unreal Engine	C++, Blueprints Visual Scripting	Yes	Yes	FBX, OBJ	Proprietary

- RAM: for the RAM performance, three.js had slightly more RAM usage than babylon.js, but in their analysis, they concluded that the difference was not statistically significant.

When looking at strictly performance results, three.js used significantly less CPU and GPU than babylon.js, because of that if performance is a priority, three.js would be our best choice, however, as we saw in 3.1.2 babylon.js wins when it comes to features and simplicity. Because of the benefits in CPU and GPU performance, three.js is the best framework for the experiments and tests of our project.

3.2 Streaming Technologies

In this section, we go through some of the solutions that utilize the most common protocols to provide a web-based experience, multimedia frameworks that can be used to stream video, and also some of the most recent technologies that take advantage of protocols like WebRTC to enable real-time streaming of 3D content. Throughout the years there have been many streaming solutions available in the market, most of them are divided into two different approaches, the first one is to simply stream video and audio, this is commonly used for non-interactive video content on platforms like YouTube and Twitch, the other is based on remote-desktop technologies, where the user can utilize a client application to remotely connect to their desktop. The last approach is the one that is most related to what we want to achieve, but the most common protocols, like

Virtual Network Computing (VNC) and Remote Desktop Protocol (RDP), were not designed to work with a modern web browser, requiring the users to install additional software, which makes it difficult to support compatibility with multiple platforms.

3.2.1 xpra.org

X Persistent Remote Applications (Xpra) is an open-source multi-platform persistent remote display server and client for forwarding applications and desktop screens [55]. Xpra gives you remote access to the full desktop or an individual application by running an X client on a remote host, and direct their display to a local machine. The tool differs from others, in that it allows disconnection and reconnection without disrupting the forward application, and it also has an Hypertext Markup Language revision 5 (HTML5) Client which is particularly useful for our project.

3.2.2 Apache Guacamole

Apache Guacamole is a clientless remote desktop gateway. As opposed to, Xpra it supports multiple standard protocols like VNC, RDP, and Secure Shell (SSH). Apache Guacamole doesn't require any plugin or client software, since everything is accessible through an HTML5 web application. The software supports cloud computing, which means that we can access desktops that don't exist physically and use a desktop operating system hosted in the cloud. Apache Guacamole is free and open-source software and is licensed under the Apache License, version 2.0 [43].

3.2.3 NoVNC

noVNC is an open-source browser-based VNC client implemented using HTML5 Canvas and WebSockets [21]. noVNC is both a Hypertext Markup Language (HTML) VNC client JavaScript library and an application built on top of that library. The VNC client supports all modern browsers including mobile, which is particularly useful for our project, it also supports scaling, clipping, and resizing the desktop, local cursor rendering, clipboard copy/paste, and touch gestures for emulating common mouse actions. For the server component, noVNC follows the standard VNC protocol, but unlike other VNC clients, it does require WebSockets support. There are many VNC servers that already include support for WebSockets, but if there is a need to use a VNC server without the support we can use a WebSockets to Transmission Control Protocol (TCP) socket proxy, like websockify [31] which is developed by the same team as noVNC.

3.2.4 GStreamer

GStreamer is a multimedia framework for constructing graphs of media-handling components and creating streaming media applications [17]. The framework is based on plugins that can provide different codecs and functionalities. The various existing plugins can be arranged and linked in a pipeline, which defines the flow of the data. GStreamer framework is designed to easily implement applications that handle both audio and video. The pipeline design is made to have a low overhead above what is already induced by the applied filters, which makes GStreamer a good framework for the design of the high-end application that places high demands on latency. The GStreamer core function is to serve as a framework for plugins, data flow, and media type handling/negotiation. It also provides an API to write applications using the various plugins [41]. To assist in the development GStreamer has bindings for some of the most popular languages including Python, Perl, C++, .NET, Java, and many more [18]. The GStreamer plugins can be classified into [19]:

- protocols handling
- sources: for audio and video
- formats: formaters, muxers, demuxers, metadata, and parsers
- codecs: encoders and decoders
- filters: converters, mixers, and effects
- sinks: for audio and video

The "GStreamer Bad Plug-ins" has a Web Real-Time Communication (WebRTC) bin, that enables the connection of a streaming server with a web-based client by using the WebRTC protocol, which is particularly useful for our project.

3.2.5 FFmpeg

FFmpeg is a free and open-source solution to record, convert and stream audio and video. FFmpeg is used by well-known software and websites to read and write audiovisual files, for example, VLC, Google Chrome, and YouTube [13]. FFmpeg can be used as a command-line tool to perform tasks like transcoding or extracting metadata from files [12]. Like GStreamer, FFmpeg allows the user to live stream video and audio from a desktop, but it does not support WebRTC connections, which makes it harder to be used in a web-based application.

3.2.6 Furioos

Furioos is a cloud-based service that enables you to stream fully interactive 3D experiences from Unity, Unreal Engine, other real-time 3D platforms, and applications [48]. Their technology

makes it easy to share the interactive 3D applications on web browsers and embed them onto websites, and the cloud rendering service can be set up almost instantly with no download and low latency to any platform. Furioos can stream 3D applications to almost any device, with the only requirement being a device that is capable of receiving a video stream through a web browser. Some key benefits of Furioos are the following [49]:

- Easy upload: Furioos can automatically detect the corresponding executable file from a ZIP archive containing an application, by simply dragging and dropping.
- Easy sharing: the interactive 3D projects can be shared with the clients and collaborators through a generated URL or iFrame to embed on a website.
- Seamless end-user experience: their technology does not require account creating, downloading a plug-in, or a high-end computer, making it accessible to any user.
- Easy streaming: the cloud rendering service can be set up instant, independently of the end-user device, with almost no download time and low latency.
- Automatic scaling: Furioos can scale to accommodate various web traffic from anywhere in the world.
- Multiplatform power: the Furioos cloud rendering service, is compatible with multiple types of equipment capable of receiving a video stream, including computers and mobile devices.

Furioos was acquired by Unity in 2019 and has been integrated into the Unity Render Streaming technologies [26].

3.2.7 Unity Render Streaming

Unity Render Streaming provides Unity's high-quality rendering via web browser, by taking advantage of WebRTC technology. It was designed to provide a solution to viewing graphic-intensive tasks like car configurators or architectural models on mobile devices [50]. When compared to other streaming technologies like Furioos, Unity Render Streaming has the disadvantage of being limited to only streaming Unity-based projects. Nevertheless, the core features introduced by the Unity Render Streaming package are the following [51]:

- Video streaming: enable the broadcast of the video rendered on Unity to the web browser via the network.
- Audio streaming: allows the streaming of sounds generated by Unity, additionally it can also cast to multiple browsers simultaneously.
- Remote control: grants the ability to send input messages to Unity from the browser and supports sending inputs from multiple browsers. As for the input devices, the feature can handle mouse, keyboard, touch, and gamepad events.

Additionally, Unity Render Streaming is also natively supported by the Furioos platform, as a result, we can use the service to easily build a render streaming application, upload it to Furioos and take advantage of the features that were presented in the Furioos section.

3.3 Cloud Gaming

In recent years Cloud Gaming has been growing immensely, and although some products already existed years ago like for example OnLive, only since 2019 we have seen a rise in the quality and quantity of cloud gaming services available, with major companies, like Microsoft, Google, Sony, and Nvidia launching their own proprietary service. Cloud Gaming is very relevant for what we want to accomplish in our dissertation since the whole premise of the service is to stream 3D content that is running in a remote server into a user's device, a common cloud gaming workflow can be seen in figure 3.1, taken from Youhui Zhang et al. [57]. A more complete architecture and performance comparison can be seen in more detail in the Shea et al. [39] "Cloud Gaming: Architecture and Performance" article, where they conducted an analysis of the state-of-the-art cloud gaming platforms and measured their real work performance with a different type of games. Although very relevant, cloud gaming is still a difficult topic to research since every major development made through the years is proprietary.

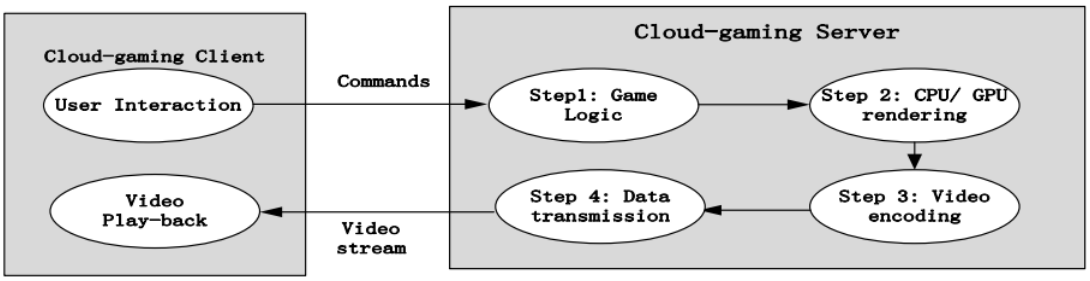


Figure 3.1: The whole workflow of cloud-gaming

3.3.1 OnLive Cloud Gaming Platform

OnLive was one of the first-ever game streaming services, which was founded in 2009 and later in 2015 was acquired by Sony [34]. Although the service wasn't particularly ground-breaking, the technology used to develop it can still serve as a base for our concept. Due to Onlive technology being proprietary, we need to use M. Manzano et al. [27] article to evaluate what kind of protocols were being used. They were able to identify the different flows that composed the OnLive traffic, and the protocols employed to transport it. The OnLive platform employs several controls and data protocols, transported using Transport Layer Security (TLS) over TCP connections and Real-time Transport Protocol (RTP)/User Datagram Protocol (UDP) flows, which are used for a variety of purposes during the different phases of an OnLive session. When it came to the different phases, they identified three main phases in an OnLive session. In the first phase, the

OnLive client authenticates and measures the latency and available bandwidth with different OnLive sites. In the second phase, once a suitable OnLive site is selected by the client, the servers start streaming the menu. Finally, in the third phase, the client selects a video game and starts playing. For the protocols, the OnLive client authentication uses TLS/TCP connection and the menu and the playing session are streamed, employing multiple RTP/UDP flows multiplexed over a single UDP port. Most of the articles had a big focus in the third phase corresponding to the gaming phase which employed the OnLive Streaming Protocol, which includes the Quality of Service (QoS) monitoring, control, and mouse pointer flows, they also explained the RTP flows. They found RTP flows like Monitor, Control, CBR-Audio, Cursor, VBRAudio, Video, and Chat in the downstream direction, and Keys, Mouse Control-ACK, and Mic in the upstream direction. And finally, they also found that the Video flow generates the largest network traffic load.

3.3.2 CloudRetro

CloudRetro is an open-source cloud gaming service for retro games. The application uses technologies like WebRTC,2.4 and libretro, which is a simple API that allows for the creation of games and emulators [42]. Like any cloud gaming, the game logic and storage of CloudRetro is hosted on a cloud service. Since it runs on a web browser, it is compatible with any platform and the most common web browsers .For further understanding of the system architecture we can observe both pictures 3.2 and 3.3, taken from their GitHub page [42]. CloudRetro also has features like collaborative gameplay, online multiplayer, and cloud storage to save your game state. Since most cloud gaming services are proprietary, CloudRetro is one of the most relevant applications for our project, due to being open-source and most importantly because it uses WebRTC, which is used in some of the most recent cloud-related services, like Stadia, Chrome Remote Desktop and Parsec Gaming.

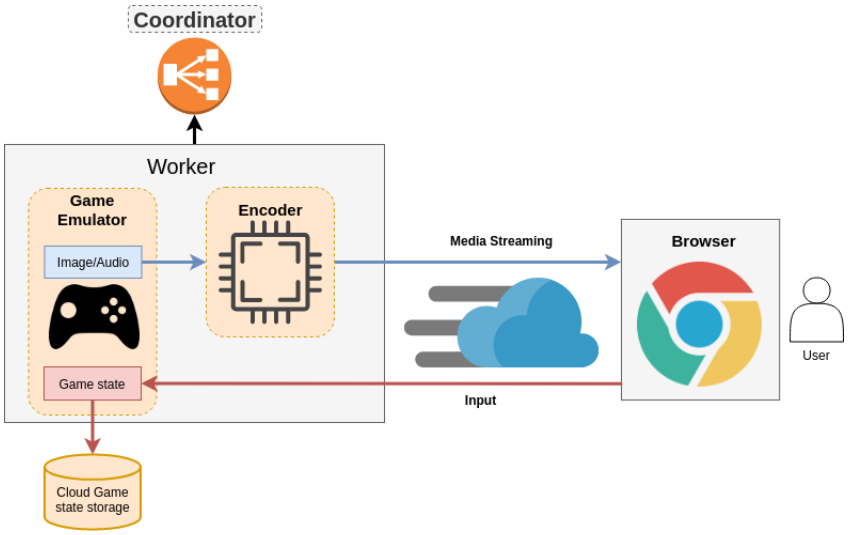


Figure 3.2: High level overview of the system architecture of CloudRetro

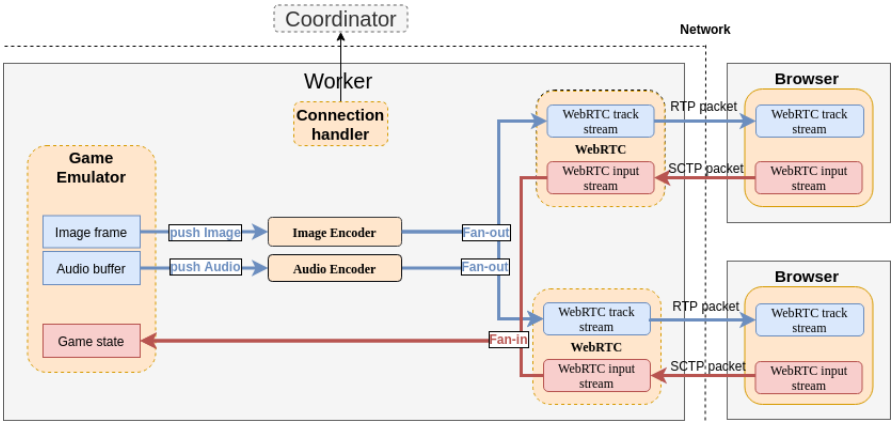


Figure 3.3: Worker internal for CloudRetro

3.4 Related Work

3.4.1 A Streaming-Based Solution for Remote Visualization of 3D Graphics on Mobile Devices

In 2007 Fabrizio Lamberti et al. [23] implemented a streaming-based solution to be able to visualize 3D graphics on mobile devices. In the paper, they used a system where clusters of computers equipped with GPUs managed by the chromium software, were able to handle remote visualization sessions based on Moving Picture Experts Group (MPEG) video streaming. Their proposed framework would allow mobile devices to be able to visualize 3D objects with millions of textured polygons at 30 Frames Per Second (FPS) or more, the frame rate is dependent on the hardware resources at the server-side and the client-side also. The way that they implemented the server-side also allow them to concurrently manage multiple clients, computing a video stream for each one, resolution and quality of each stream were also tailored according to screen resolution and bandwidth of the client. In the figure 3.4 taken from the article, we can better evaluate how their three-tier architecture works. As said before one or more clients can remotely control a 3D graphics application by interacting with Remote Visualization Server (RVS) based layer, that is responsible for handling distributed rendering on a cluster of existing PCs using Chromium, then the frames generated by the RenderVideo SPU are encoded into multiple video sequences by the Encode server and finally, they are streamed to the clients using multi-cast wireless channels by the Streaming Server components.

3.4.2 GamingAnywhere: an open cloud gaming system

GamingAnywhere was one of the first open cloud gaming systems developed in 2014 by Huang et al. [20]. GamingAnywhere, in contrast to OnLive 3.3.1, is an open system, in the sense that a component of the video streaming pipeline can be replaced by different components with different algorithms, standards, or protocols. By default, GamingAnywhere employs a highly optimized

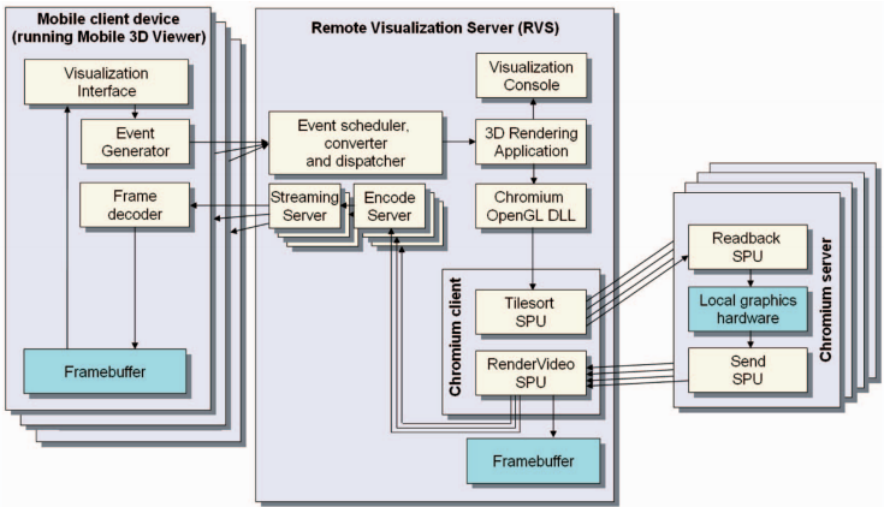
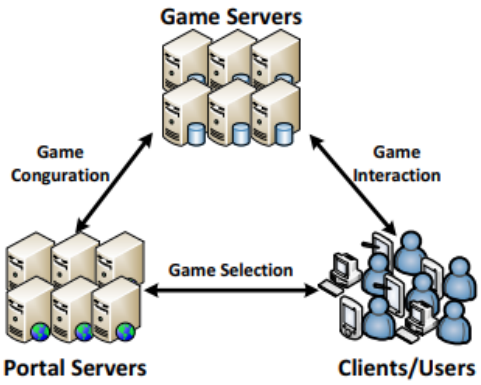
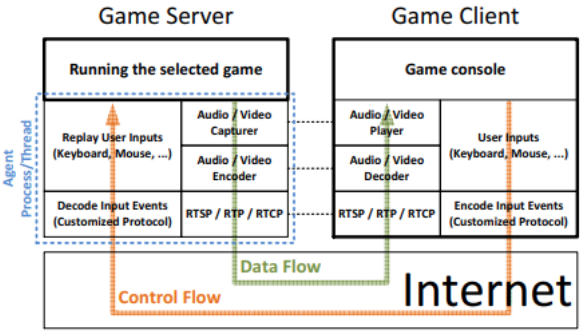


Figure 3.4: Fabrizio Lamberti et al. layout of the proposed three-tier architecture

H.264/AVC encoder [52], to encode captured raw videos. The system has been designed to be efficient, by minimizing the time and space overhead, using a shared circular buffer to reduce the number of memory copy operations. Such optimizations, allow GamingAnywhere to provide a high-quality gaming experience with a shorter response delay when compared to similar services. When it comes to the system architecture, as we can see in both figures 3.5 taken from the article, GamingAnywhere has two distinct components, the Game Server and Game Client. The user starts by logging into the system via the portal server for the game client, then selects the game and requests to play it. When the request is received, the portal server starts by finding an available game server and then launches the chosen game into the available server.



(a) The deployment scenario of GamingAnywhere



(b) A modular view of GamingAnywhere server and client

Figure 3.5: GamingAnywhere System Architecture

3.4.3 A Cloud Gaming System Based on User-Level Virtualization and Its Resource Scheduling

Due to the recent increase in cloud gaming popularity, there have been different takes on how to best implement such a service. In 2016 Zhang et al. [57] wrote an article on developing a cloud gaming system based on user-level virtualization. The article is particularly relevant for our project since in cloud gaming we need to have interaction between client and server and user-level virtualization is useful when trying to deal with the existence of multiple clients using the service at the same time. They proposed the design of a GPU/CPU hybrid system called GCloud, which used the user-level virtualization technology to implement a sandbox for different types of games, which allow them to isolate more than one game instance from each other on a game-server, capture the game video and audio outputs for streaming and handle the remote client device inputs, the proposed system can be seen in more detail in the figure 3.6 found in the paper. Additionally, they implemented a performance model, that analyzed the resource consumption of games and performance bottlenecks of a server, by performing experiments using a variety of hardware performance counters. When it came to games, they categorize them into two types: CPU-critical and memory-io-critical, because of that they also implemented several scheduling strategies to improve resource utilization. When compared to GamingAnywhere, GCloud differs from it since it starts by implementing a virtual input layer for each of currently-running instances, rather than a system-wide one, which enables them to support more than one Direct-3D games at the same time. GCloud also designs a virtual storage layer that stores each client configuration across all servers, which was not implemented in GamingAnywhere.

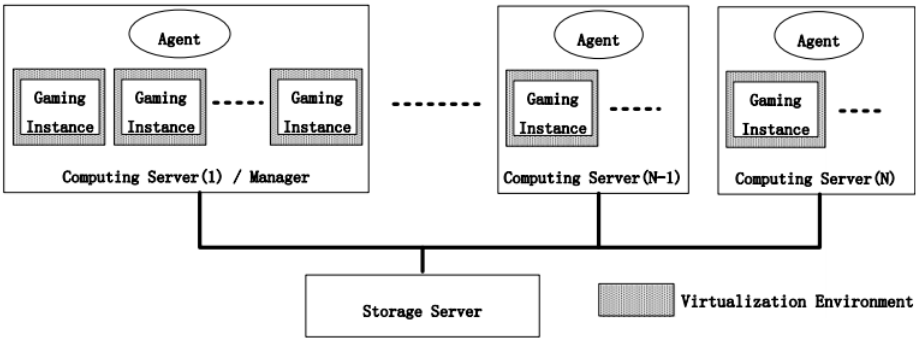


Figure 3.6: Zhang et al. user-level virtualization system architecture

3.4.4 GPU-based remote visualization of dynamic molecular data on the web

Mwalongo et al. [29] implemented an efficient web application for visualization of dynamic molecular data using WebGL. Although their focus was on streaming large amounts of data instead of 3D content, their approach to building an efficient web application is still relevant since we want to be able to use as few resources as possible. Their application implements a client-server architecture that can be seen in the figure 3.7 taken from the article. First, the

client starts by sending a request Uniform Resource Identifier (URI) to the server, then the server parses the request and sends a set of HTML files, JavaScript code, and the WebGL OpenGL Shading Language (GLSL) shaders required to use in the user interface and visualization. The client-side works by establishing a WebSocket connection to the server, which is later used to request and obtain the raw visualization. The server-side is implemented as part of a visualization framework that loads the client's requested molecular data, which contains atomic coordinates at specific time periods, than extracts the values for rendering, and transmits them through the already existing WebSocket connection. Finally, the server encodes the data, so that the client only needs as few operations as possible to obtain a directly renderable representation, the encoded data also needs to be as small as possible so that the message is quickly transferable to the client.

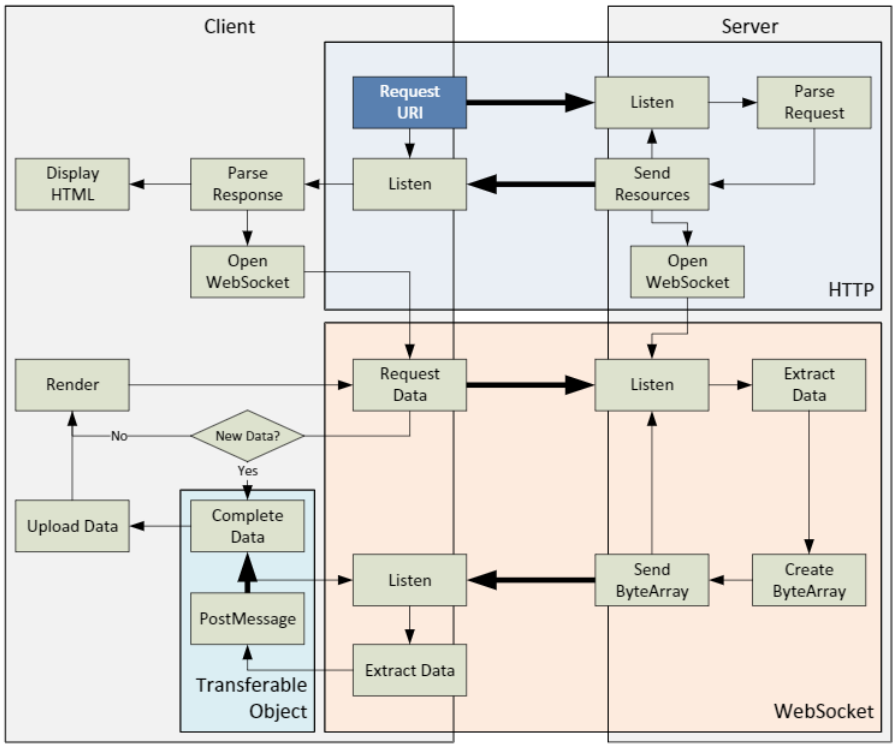


Figure 3.7: Mwalongo et al. remote visualization of molecular data system architecture

3.4.5 A real-time remote rendering system for interactive mobile graphics

In 2012, Shi et al. presented an advanced low-latency remote rendering system that assisted mobile devices to render 3D graphics in real-time [40]. They used a workstation as a rendering server, which rendered 3D content and transmitted the extracted result images to the mobile clients, the client was used to display the 3D image and didn't perform any rendering. To handle user interactions, like changing the rendering viewpoint, the mobile client runs 3D image warping with the received depth images to synthesize an image at the updated rendering viewpoint. As a result, the interaction latency of the remote rendering system is reduced to the time of image synthesis on mobile, which is independent of the network. The framework and an illustration of

the interaction latency can be seen in the figure 3.8, taken from the article.

When comparing the discussed proposed system with the one that we saw for example in OnLive 3.3.1, they differ by instead of sending one color image per frame to the client, they render the 3D scene from multiple rendering viewpoints and send multiple depth images to the client, which helps to reduce the interaction latency and keep the high rendering quality but has the cost of extra consumption on the server and more network bandwidth for streaming all the extra depth images.

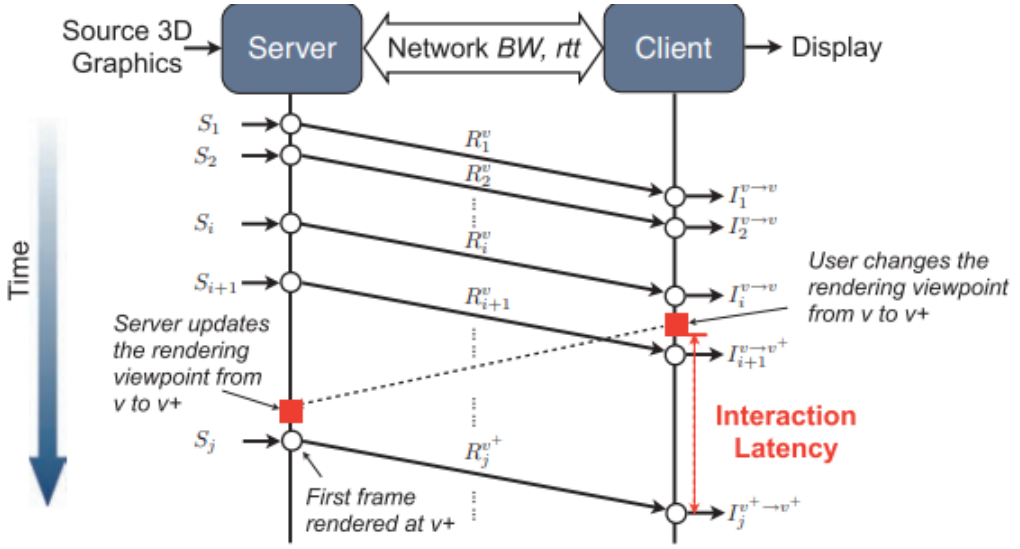


Figure 3.8: Shi et al. remote rendering system framework and an illustration of interaction latency

3.4.6 A Hardware-Accelerated System for High Resolution Real-Time Screen Sharing

In 2018, Yang et al. introduced a hardware-accelerated system for real-time screen sharing, that focused on streaming at ultra-high resolution, which decreased the encoding workload by taking advantage of content redundancies between successive screen frames [56]. Their approach made use of multiple codecs that were capable of utilizing various encoders with H.264 Advanced Video Coding (AVC) of different input sizes, as a result, they were able to save encoding time by always selecting the appropriate encoder for the specific updated screen content. Additionally, they proposed a frame split mode in metadata processing, that separated small screen updates into independent frames to obtain lower encoding complexity and better latency performance.

For the experiments, they compared the performance improvements of applying the multiple codec approach to the screen content compression and frame split mode to metadata processing against the basic single H.264/acAVC codec implementation. They concluded that the multiple codec approach outperforms the single codec in encoding time for common screen sharing scenarios, the presented frame split mode in metadata processing lowered the computational

complexity in interactive scenarios with minimal addition in network traffic usage. Lastly, when regarding the latency measurement, the proposed solution provided low end-to-end latency, ranging from 17 to 65 Milliseconds (ms).

3.4.7 GPU-accelerated streaming using WebRTC

GPU-accelerated streaming using WebRTC is an article found on the Google Cloud solutions page, where they describe the components of a web-based interactive streaming solution for graphic-intensive workloads [15]. The article works as a guide on how to develop the streaming solution by discussing every detail regarding the components necessary to the implementation. Their reference architecture can be seen in the figure 3.9, where they take advantage of three core technologies to create an individualized streaming application:

- WebRTC: WebRTC is the protocol utilized for adding real-time communication capabilities to the streaming application. With WebRTC they were able to establish a connection between the web client and the server backend and broadcast the video stream over the web with low latency.
- GStreamer: GStreamer was the multimedia framework, that served as the streaming engine. They implemented a GStreamer pipeline that captured an X11 window and encoded the buffer to H.264 on a GPU using Compute Unified Device Architecture (CUDA) and NVIDIA Encoder (NVENC).
- Google Cloud Platform: Google Cloud Platform (GCP) handled the individualization and scaling of the streaming application. They used GCP to deploy the WebRTC streaming stack and make it available to individual users, by authenticating requests and assigning a Compute Engine instance to each client.

Although the article mentioned is more focused on the implementation of the streaming solution, they also published a second article on "Orchestrating GPU-accelerated streaming apps using WebRTC" [16] that describe how the GCP can be used for building an orchestrated multi-tenant Virtual Desktop Infrastructure (VDI), which is particularly useful for understanding how Virtual Machines (VMs) can be used to scale a streaming application to be utilized by multiple users, additionally, the project behind this article is still in active development, and is now maintained by the Selkies Project [5].

3.5 Conclusion

Throughout this chapter, we researched and analyzed the existing frameworks, streaming technologies, and related work to our thesis. In this section, we will highlight the conclusions that were possible to be taken from the previous analysis.

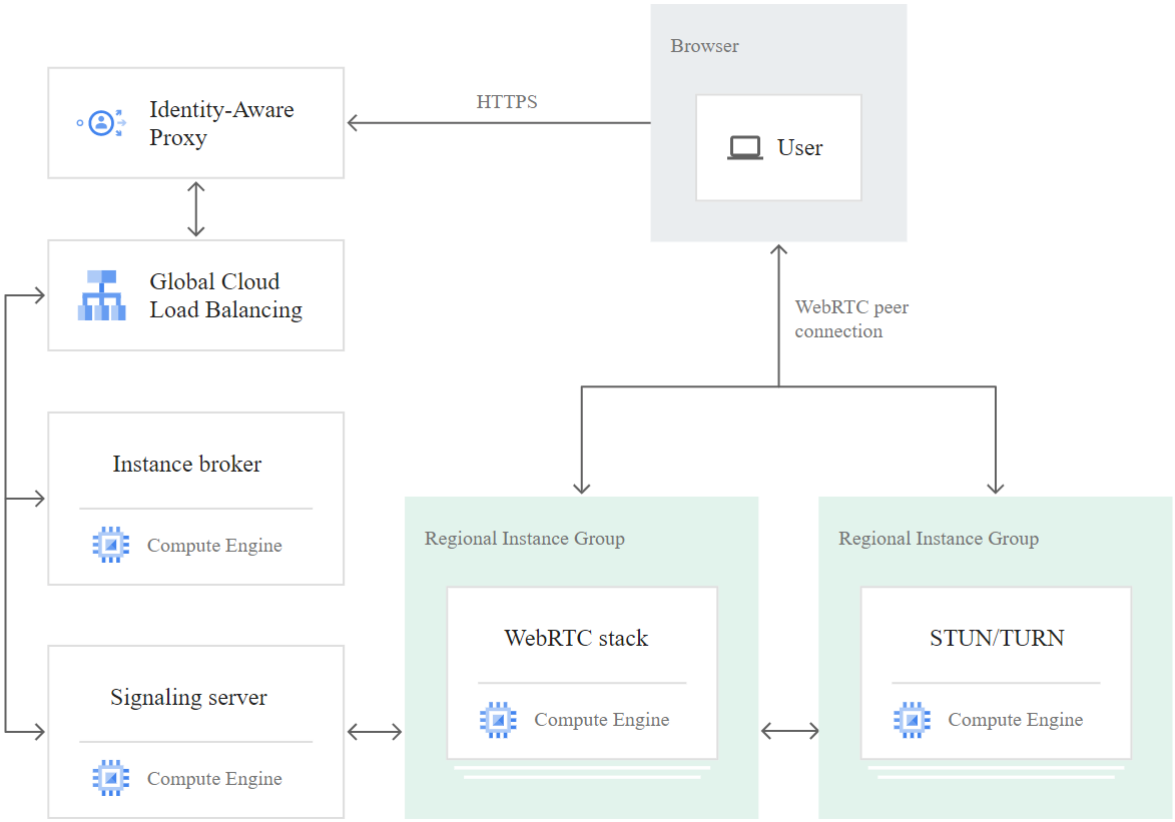


Figure 3.9: GPU-accelerated streaming using WebRTC Architecture

We started by exploring the current WebGL frameworks, to evaluate the most appropriate solution that could be used to develop the web application required to achieve the experiments and tests. During our research, we evaluated various frameworks, where we concluded that the most commonly used frameworks were Three.js and Babylon.js. In section 3.1.9 we compared both frameworks and evaluated the features and performance results for each, concluding that due to the simplicity, extensive documentation, and performance the framework Three.js would be the most optimal solution to be used in the development of our testing web application.

Following the WebGL frameworks, we proceeded with the research of the current streaming technologies. We started by researching the current web-based solutions used to remotely control and visualize an application. The most common solutions found were Xpra, Apache Guacamole, and NoVNC. These applications have support for web-browsers and make use of existing protocols like VNC and RDP. Although these are the most popular, we concluded that they are not appropriate to streaming graphically intensive 3D content due to o limitations in the encoder that is supported. Due to that reason, we proceeded with the research of the existing multimedia frameworks, that enables the streaming of video and audio which could be used during the implementation of our solution. With that in mind, we evaluated two frameworks, GStreamer and FFmpeg. Both of these frameworks, enable the creation of applications capable of streaming content with video and audio. FFmpeg was one of the most popular technologies, but it does not support WebRTC communications. Since our objective was to implement a web-based solution,

we concluded that GStreamer was the best multimedia framework, due to providing WebRTC support through the usage of a plugin.

To conclude the state-of-the-art, we studied the current related work and articles. One of the most relevant was the "GPU-accelerated streaming using WebRTC", which described in detail the components for a web-based streaming solution to visualized graphically intensive content through the usage of WebRTC, GStreamer, and the Google Cloud Platform. The first technology that their solution used was the WebRTC protocol for web-based communications in real-time. The next technology was the GStreamer multimedia framework, which was utilized to implement a pipeline capable of capturing a window and stream it to a web-based client. The final technology was the Google Cloud platform, which served the purpose of handling the client individualization and expanding the streaming application.

Chapter 4

Implementation

In this chapter, we describe the contributions and the development that was done. The implementation is divided into the server-side and client-side, where we structure and outline each component that was involved in the development phase. We start by explaining the setup required to develop our streaming solution, following by a general illustration of the system architecture, succeeding with the server backend, and conclude with the web client.

The overall goal of our thesis is to provide a streaming solution, where a user can visualize and interact with graphic-intensive applications on any device. Our implementation had as its focal point the user experience, by providing a simple to use User Interface (UI) and optimizing performance metrics, such as latency and framerate. With that in mind, we developed a simple and efficient web-based client that was capable of connecting with the server and visualize the content being transmitted, and a server backend that contains the streaming engine and a data communication Application Programming Interface (API).

In figure 4.1 we can observe the three main components required for the functioning of our streaming solution. For the server-side, we studied various solutions for the streaming component, concluding that the usage of a multimedia framework like GStreamer, would provide the best streaming engine for our application, due to the possibility of being able to create multiple pipelines with various encoders and the existence of a Web Real-Time Communication (WebRTC) plugin. Since we used WebRTC, the server-side also needs to provide a signalling service, required to establish WebRTC-based connections between the client and the server.

Regarding the client-side, we developed a web-based application, that allows the user to establish connections between the server backend, to visualize and interact with the content that is being streamed, by utilizing the JavaScript framework Vue.js. The web client interface was designed to be as simple and minimal as possible to provide the best streaming experience while being compatible with multiple devices.

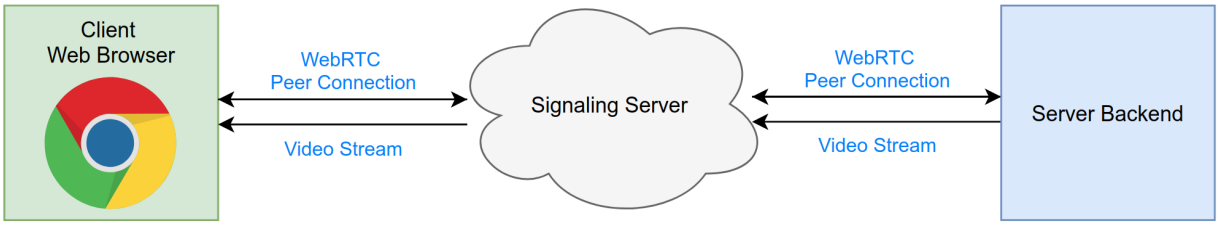


Figure 4.1: System Overview

4.1 Specification

Before proceeding with the development, we need to specify the set of requirements that our implementation needs to comply with.

Starting with the web browser client, the following set of assumptions were defined to ensure we achieve the best streaming experience:

- The first requirement is compatibility. We want the client to be agnostic to the device and platform. Due to that reason, the client must not require any additional software installation and the UI must support different resolutions and devices.
- Regarding the UI, we will want to display various metrics that can be used for debugging purposes and also provide useful information about the server. Besides displaying information, we will also need to provide a set of options, to enable the user to adjust the streaming experience.
- For the provided options, we want the user to be able to change the bandwidth usage, framerate, and dynamically alter the streaming application resolution.
- To conclude the client specification, we also need to handle the input events generated by the user, by capturing touch, mouse, or keyboard events and send them to the server.

To finalize the implementation specification, we set the following assumptions and requirements for the server backend:

- To commence, the server and the client need to establish a connection between each other. For that, the server will need to have a signalling server that will enable communication between both peers.
- For the streaming component, the server will need to have a GStreamer pipeline, that is capable of capturing an application window and broadcast the video stream to the client, by using WebRTC.
- Additionally, we want the client to be able to stream any application. With that in mind, the streaming component will need to support the broadcast of any application, through the inclusion of an application argument.

- We want the client to adjust the bandwidth usage, to accommodate different network conditions. For this to be possible, our streaming pipeline will need to support dynamically changing the bitrate values.
- To support the user-generated inputs incoming from the client, the server will need to emulate the mouse and keyboard client events.

4.2 Environment

We start our development phase by preparing our work environment. When it comes to operative systems, we utilized Ubuntu 20.04 Long-term Support (LTS) throughout the entire development phase. Another major component in our development is the web browser, although the most recent and wide available browsers already have available the necessary support for the technologies we want to use, there is still a need to ensure that the browser contains the full compatibility for WebRTC [1].

Additionally, even though some browsers may support WebRTC they might not have fully implemented all of its features, also, some browsers still have prefixes on some or all WebRTC APIs [28]. To mitigate some of the described issues, a `adapter` provided from the WebRTC organization was used. The adapter is a JavaScript library that allows your code to be written to the specification so that it works in all browsers with WebRTC support. To simplify the development of our application, only Google Chrome was used for the entire duration of the implementation phase. As future work, testing with different browsers should be made, to ensure compatibility with multiple browsers.

For the server backend, the environment is more complex, due to the dependencies and different tools that were necessary to be used to accomplish the defined objectives. Three key tools were necessary during the development of the server-side:

- Node.js: Node, was mostly used for the development of the web client and served the purpose of managing any package or dependency that was necessary for the implementation.
- Python: Python 3.8.5 was the language of choice when it came to the signalling server and the development of the server backend.
- Docker: Docker, was used for the server-side of our application, mainly on the development of our streaming engine. The tool facilitated the overall development and delivery of the application and helped to manage the different dependencies by allowing the creation of various containers.

A much more in-depth illustration of the system architecture and the reasoning behind the usage of each one of these tools is done in their respective following sections.

[1] A list of the currently supported browsers can be seen at https://en.wikipedia.org/wiki/WebRTC#Support

4.3 System Architecture

To fully understand what was involved in the development phase, we start by explaining the general system architecture of our application. An illustration of the system architecture can be observed in figure 4.2.

Our application is divided into a client-side and a server-side. The client is composed of a web browser that utilizes, WebRTC for real-time communication and Vue.js for UI development. The server-side of our application contains the streaming engine built using the GStreamer framework, the Signalling server, and the application we want to stream.

Regarding how the communication between the client and the server occurs, WebRTC requires the usage of a Signalling service for the negotiation and discovery process, the service allows for the multiple peers in a different network to find each other. Besides the signalling service, the WebRTC application can use the Interactive Connectivity Establishment (ICE) framework, to find the best path to connect peers, for ICE to make a connection, it needs to obtain an external address using a Session Traversal Utilities for NAT (STUN) server. Since, for our project, the connection with the client and server strictly occurs on a local network, only a simple, Google STUN server like "stun:stun.l.google.com:19302" was necessary. A detailed explanation about the communication is done in section 4.4.2, where we discuss how the Signalling server was implemented and illustrate how the communications take place.

For the application we want to stream, we decided to use Unity with one of the provided sample projects. Nonetheless, the streaming engine is completely independent of the chosen application, which means that any application can be used to stream and, it's not required by the server for the same application to be running on the same machine.

In the following sections, we discuss in detail each one of the elements presented in the architecture. We start, by detailing the implementation of the server backend, where we explain the development of the Signalling server and the GStreamer WebRTC application, and to conclude the web client.

4.4 Server Backend

The most common tools used to remotely control applications and desktops, like NoVNC and Apache Guacamole, are not suited for high graphical intensity tasks and low latency interaction through the web browser. WebRTC is the current state-of-the-art technology used in services that require the most graphics-intensive and low latency possible workloads, such as web conferencing and cloud gaming. Because of these reasons, we decided that WebRTC would be the best solution for our project, when it comes to stream and control Three-dimensional (3D) content from a server through the web browser.

WebRTC allows real-time, peer-to-peer media exchange between multiple devices. For the

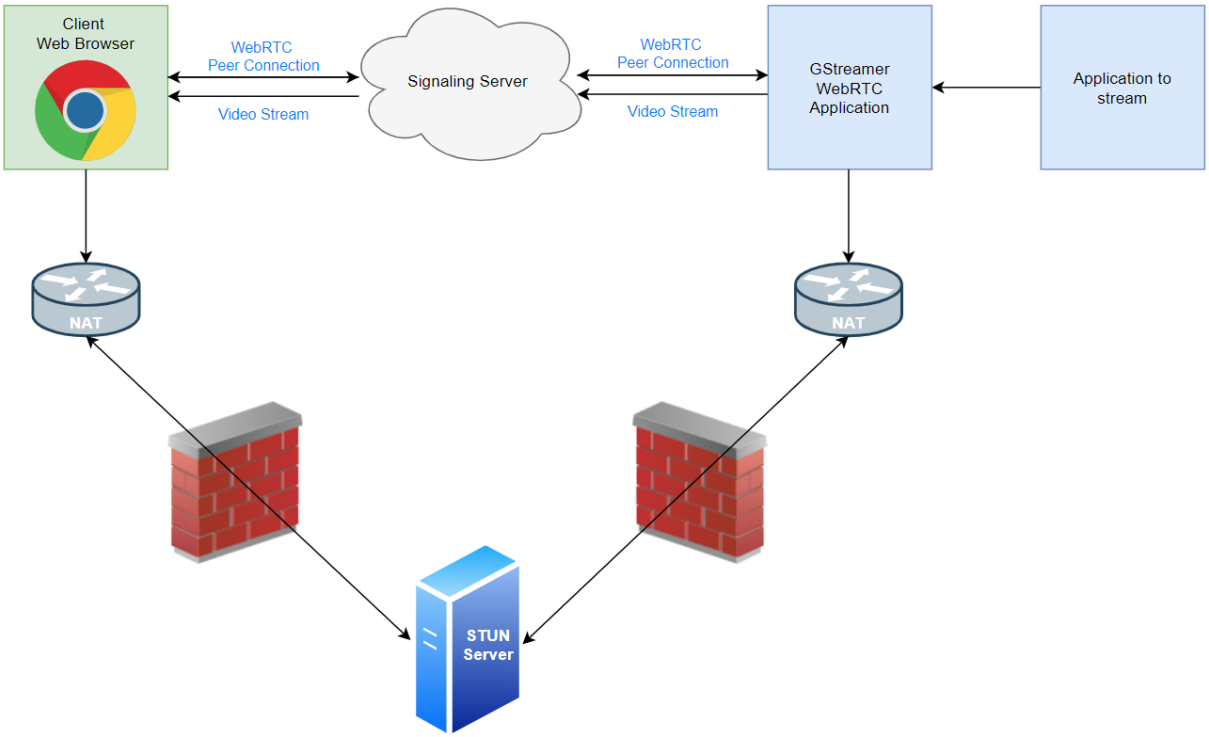


Figure 4.2: System Architecture

development of a WebRTC based application, we need to implement a client and a server, and the connection between these components is established through a discovery and negotiation process called signalling. Although WebRTC handles the real-time communication side of the application, we still need the major component that deals with capturing a window and stream it to a client. For that, we used the multimedia framework GStreamer, which provides a library to develop a high-performance streaming pipeline by making use of the current state-of-the-art video encoders.

One of the disadvantages of the most commonly used remote desktop tools is the lack of hardware encoding. Nowadays, most modern computing architectures already include dedicated chips designed for image and video processing. NVIDIA Graphics Processing Units (GPUs) contains a hardware-based encoder, called NVIDIA Encoder (NVENC), that supports accelerated video encoding and is independent of system graphics. When deciding the encoder we want to use for our implementation, there are two distinct types in which we can classify them, Software and Hardware-based encoders. Since our project involves streaming 3D content to the web, we need to choose the encoder that allows you to have the best performance possible in both latency and framerate. According to NVIDIA granted results [32], the NVENC H.264/AVC GPU-based video encoding can be 5 times faster, on average, than the CPU-based X264 [52] implementation. Furthermore, in 2017, Albanese et al. [1], evaluated the performance of various software and GPU accelerated video transcoding units for multi-access edge computing scenarios. They carried many tests, to achieve a full performance characterization of the Video Transcoding Unit (VTU), for both software and hardware only versions. In particular, they showed results

(expressed in frames per second), featuring the H.264 and H.265 transcoding with Software (SW) acceleration and with Hardware (HW) acceleration (using a GPU), where they decoded an input video file to four different video resolutions. They also, compared the computational resources used during transcoding for both SW and HW only, and to finalize, they compared the efficiency of these two solutions in terms of performance/watts. On all the tests discussed above, they concluded that hardware-accelerated implementation, presented superior results, not only in performance when regarding Frames Per Second (FPS), but also in power consumption, they also concluded that, when it came to performance, H.264 provided better results than H.265 for video encoding. When it comes to streaming high-resolution content, the utilization of a hardware-based encoder, such as the NVENC H.264, for our streaming engine pipeline, provides higher performance capabilities that can be utilized to boost screen frame processing and offload workload from the Central Processing Unit (CPU).

Our implementation was based on the article "GPU-accelerated streaming using WebRTC" [15], and also on the recent Selkies Project [5], that was forked from that same article. Although these projects were focused on the deployment of the WebRTC streaming stack to the Google Cloud platform, both their implementation and article described in detail how to create a general-purpose web-based streaming application.

In the following sections, we discuss what was involved in developing the backend of a WebRTC based application that utilizes GStreamer to access a GPU for hardware encoding, by discussing the system architecture and further explaining in detail each respective component.

4.4.1 Server Backend Architecture

In section 4.3, we saw a broader view of our system architecture. To have a better understanding of the server backend component, we start by explaining the respective architecture and follow up, by analyzing each component of the server architecture.

In our illustration of the system architecture 4.2, we observed that our server backend had three major components: the Signalling Server, the GStreamer WebRTC application, and the application to stream. For the simplicity of the development, all of these components are running on the same server, although this is not, by any means, a requirement, since the three components are independent of each other. As an improvement for future implementation optimization, the presented components should be in three separate servers or containers. These changes would ensure better expandability, system reliability and also would present benefits in performance, by not having the application to stream occupy computational resources that could be used by the streaming engine and vice-versa.

Regarding the purpose of each component, the signalling server is utilized for the discovery and negotiation of multiple peers, the GStreamer WebRTC application contains the streaming engine and also serves as an interface for various components, and finally, the application to stream is the application that we feed through the streaming engine to stream to the web browser.

As a more detailed system architecture of the server backend, we can observe figure 4.3. Although the three components are running in the same server, the GStreamer WebRTC application containing the streaming engine is functioning inside a docker container, due to the necessity of having an isolated environment from the rest of the components, with the docker container, we could more easily manage the different dependencies that were necessary to build the GStreamer application and test the application without the need to change the host system installation, which could affect other applications and make the host machine less stable. The docker container also brings benefits in the packaging and expandability of our infrastructure, since it makes it easier to expand by deploying our application in multiple servers. More in-depth information on the docker container and its receptive configuration are done in the following sections.

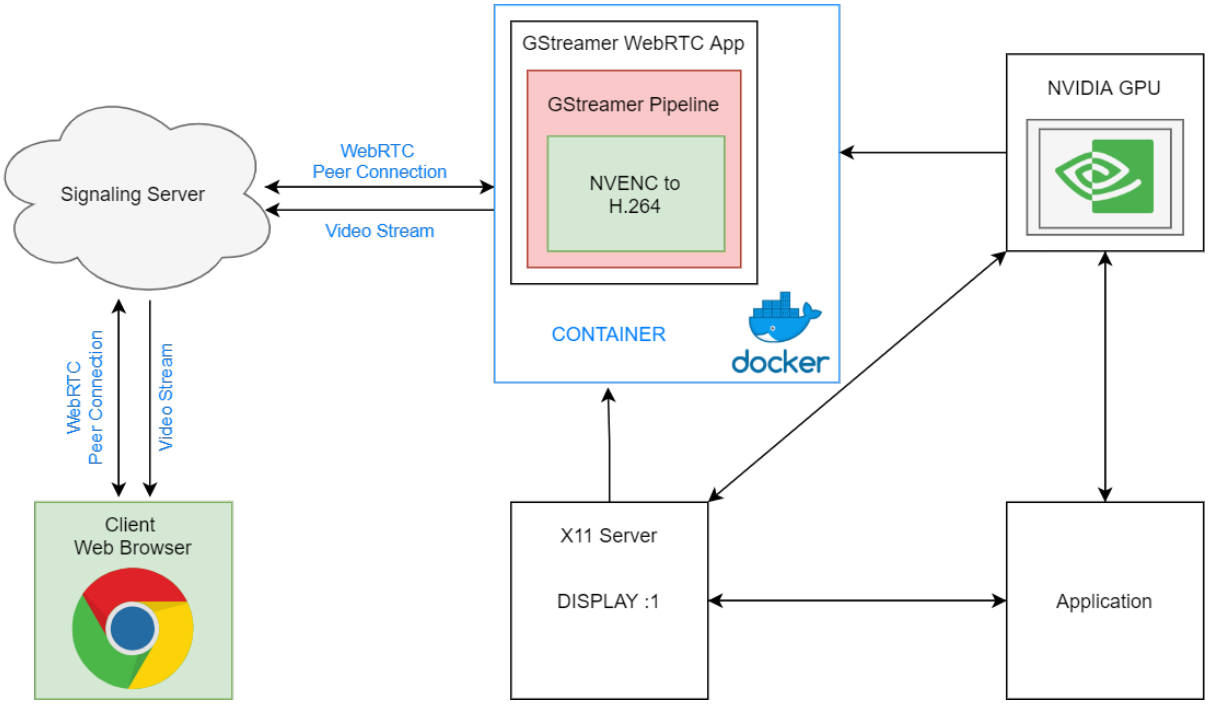


Figure 4.3: Server Backend Architecture

4.4.2 Signalling Server

For multiple devices on different networks to find each other, the discovery and negotiation need to be done through the signalling process. The process of signalling does not require you to use a single messaging protocol, instead, you have the choice to use the protocol that suits you best.

Our signalling server has to communicate with a GStreamer Pipeline to stream video. Since in our project the communication is only done between two devices, we can use as our signalling server the example provided by the GStreamer team [30].

Their implementation of the signalling server manages the peers and loads data between those peers, by utilizing the WebSockets protocol. First, the server registers the peers, which

is done by connection with the WebSockets server and sending and receiving "HELLO <uid>" messages with the unique identifier of the peer. For the second step, since our application only supports the connection to a single peer, the signalling server sends a "SESSION <uid>" message that identifies the peer we want to connect to and receives "SESSION_OK" once the connection is established. Finally, once the connection has been set up with the signalling server, the peers must negotiate Session Description Protocol (SDP) and ICE candidates with each other. Once the peer connection has been established, the signalling server is no longer required, and all further messages are made directly to the peer. The diagram 4.4, is based on the diagram found in [15] in section "Core Concepts", and shows the described process in more detail.

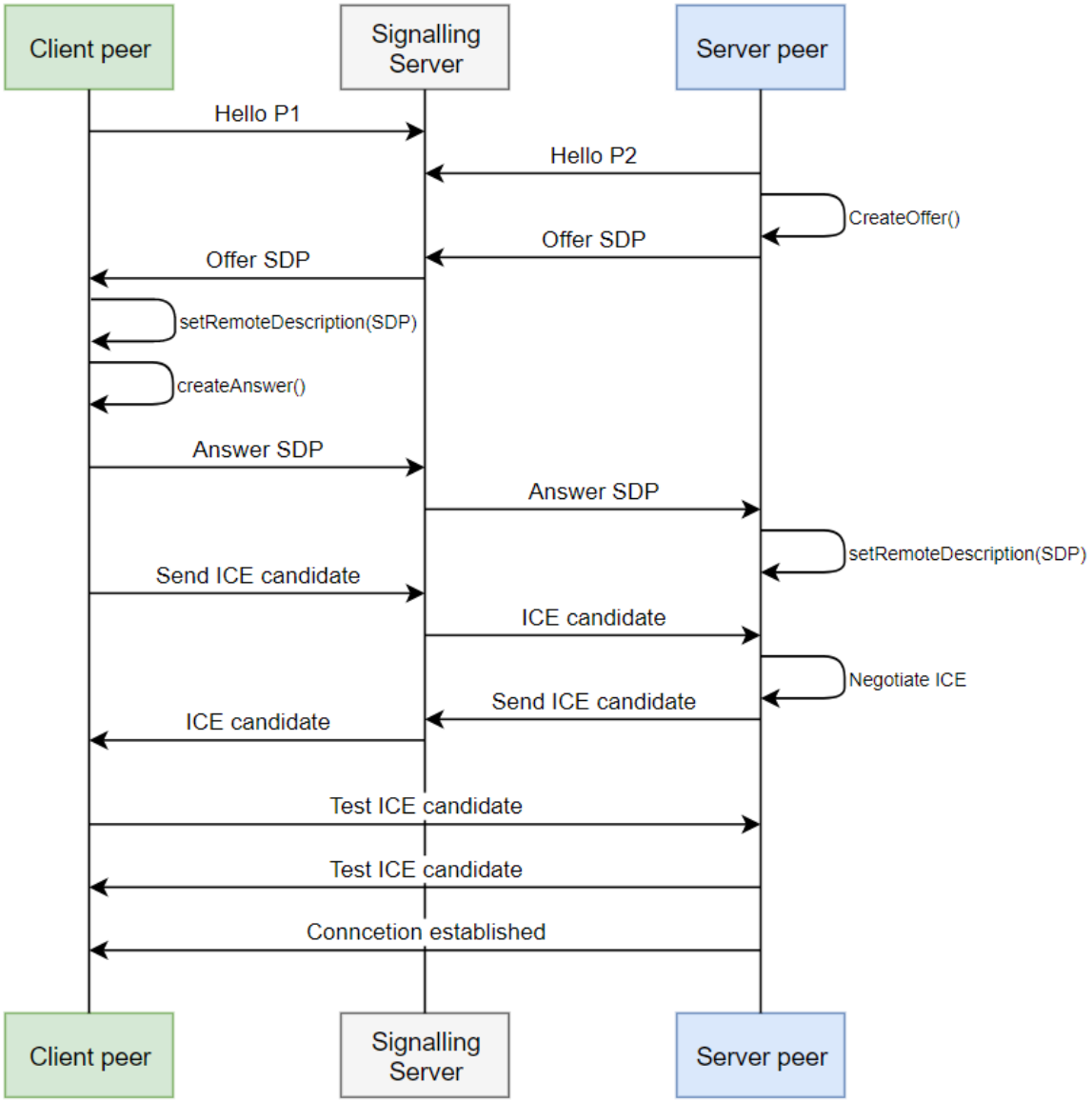


Figure 4.4: Signalling Server Connection Handshake

4.4.3 GStreamer WebRTC Application

The GStreamer Pipeline is the core of our streaming component, which enables us to stream 3D content through the web with high performance. GStreamer is a framework that links a variety of media processing systems, capable of handling multimedia streams with audio and video. In 2018, the support for WebRTC connections was added, with the release of GStreamer 1.14 [46]. The GStreamer WebRTC plugin facilitated the possibility of utilizing the framework in a web-based environment.

In our implementation, GStreamer is not only utilized as the streaming engine but also serves as an interface with the different required components, like, capturing an X11 window with a given Identifier (ID), communicate with the signalling server, handling the usage of the NVIDIA GPU and treating the user inputs received from the web client.

In figure 4.5, we illustrate in detail the architecture of the GStreamer WebRTC application. The development of the application was done using Python, which is one of the languages officially supported by the framework. Python brings some advantages due to the extensive library that is supported, which includes the `XLib` library, that allows the management of X Windows. In the following sections, we observe how the language and its libraries were applied during the development.

Regarding the organization and architecture of our application, as expected, the application is started by utilizing the "MAIN" component, which initiates each one of the required modules and processes the received arguments. The application has 3 core arguments:

- `Encoder:` the encoder argument allows you to choose the type of encoder you want to use on the pipeline, from a range of supported hardware and software-based encoders. The supported encoders are the following: `nvh264enc`, `nvh265enc`, `x264enc`, `x265enc`, `vp8enc` and `vp9enc`. Throughout the project, the implementation was mostly focused on the `nvh264enc`, since it was the one that showed the best performance.
- `Signalling Server:` the argument lets you choose the address of the signalling server, which might be useful in cases where the signalling service is located in a different server or machine. For the project, the signalling server defaults to the localhost.
- `App Name:` contrarily to the previous arguments, the app name is obligatory for correct usage of the application. With the argument, we can choose the application we want to stream by giving its name. After parsing the application name argument, the "getXWindowID" method is called, to convert the name to an ID, which can, later, be used for the GStreamer pipeline and input handling. In case, an application name is not provided as an argument, the GStreamer pipeline tries to stream the entire desktop.

As discussed earlier, the application is composed of three core modules, that are initiated by the "MAIN" component:

- `WebRTCSignaling:` which interfaces with the signalling server discussed in section 4.4.2.
- `GSTWebRTCApp:` the GSTWebRTCApp module contains the GStreamer pipeline necessary to create the streaming engine.
- `WebRTCInput:` the third and last module, handles the user input attained from the web client.

In the next sections, we start by explaining in detail the configuration required to run the GStreamer application inside a docker container, and we also further explain each module discussed in the list above.

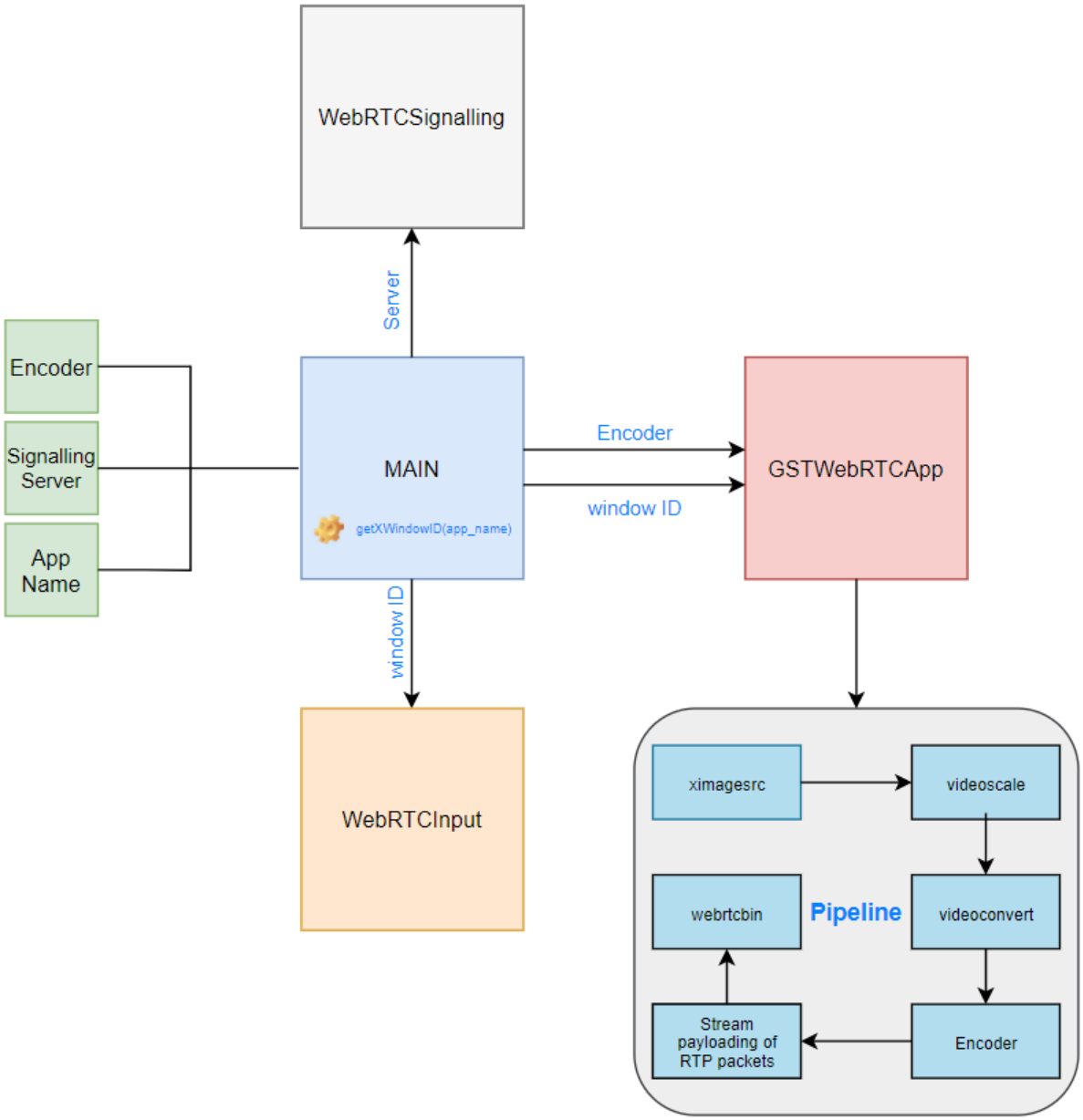


Figure 4.5: GStreamer WebRTC Application Architecture

4.4.3.1 Docker Configuration

In the following section, we discuss how docker was configured and its utilization. One of the key requirements to ensure we obtain the best performance possible in our container is the ability to pass through the host GPU to the docker container. To make that possible we utilized the Nvidia Container Toolkit [33], which allows the users to build and run GPU accelerated containers, by automatically configuring containers to leverage NVIDIA GPUs.

The first step was to package the GStreamer WebRTC application by creating a Docker image. The image is a template that contains a set of instructions for creating our Docker container, that simplifies the development and packaging of the application by having a preconfigured server environments with all the dependencies necessary to fully run the application without the need for further setup. Since we utilized our application, we created and built a custom image, which was done by implementing a Dockerfile. A Dockerfile is a text document that contains all the commands that a user could call on the command line to assemble an image [2].

To develop a Dockerfile, we need to decide our base image. The base image is the parent that our image is based on, and it can usually be a minimal Operative System (OS) based image or, in our case, a preconfigured NVIDIA image. For our base image, we chose the Ubuntu 20.04 based `NVIDIA CUDA` development image, that comes with the Compute Unified Device Architecture (CUDA) toolkit, and includes GPU-accelerated libraries, a compiler, development tools and the CUDA runtime, which are a dependency for one of the CUDA based GStreamer pipeline. The complete name of the chosen base image can be seen in the listing 4.1.

```
FROM nvidia/cuda:11.2.0-devel-ubuntu20.04
```

Listing 4.1: Dockerfile base image

The next step in development is to build GStreamer and install all the required dependencies. To ensure, we have the most optimal and up-to-date version of GStreamer we built it from the source. We start by cloning each necessary GStreamer module from the official repository and build it by following the instructions provided in the official documentation [8]. The modules that we built for our project were the following:

- `gstreamer`: GStreamer is the core in which all the modules resolve around. It includes the base functionality and libraries.
- `gst-plugins-base`: The "base" module contains collections of well-maintained GStreamer plug-ins and elements. Some useful plugins that we can find here, are the "videoscale" and "videoconvert" that we used in our pipeline.
- `gst-plugins-good`: The "good" module, includes a set of plugins that are considered to have good quality code, correct functionality, and preferred license. Here, we can find

[2] More information at https://docs.docker.com/engine/reference/builder/

most of the required modules for our pipeline, such as, "ximagesrc", "rtp" and "vpx".

- `gst-plugins-bad`: The "bad" module, includes plugins that aren't to par when compared with the ones included in the previous module. The provided plugins might require further testing, documentation, or code reviews. Here we can find some of the most important plugins for our application, the "webrtc" and "nvcodec", the first one contains the elements required to establish WebRTC based connections, and the last plugin contains the encoder required to encode video streams using NVIDIA GPUs.
- `gst-plugins-ugly`: The "ugly" module, consists of plugins that are up to par with the "good" module, in terms of code quality and expected functionality, but present problems regarding distribution due to their license. The module has the "x264" plugin, which is used for software-based accelerated pipelines.
- `gst-python`: this module, facilitates the development of the GStreamer WebRTC application utilizing the Python language.

All plugins and elements referred to in the list above, are explained in detail in the GStreamer Pipeline section 4.4.3.3. A more detailed example of the commands required to include in the Dockerfile, to build one of the modules can be seen in the listing 4.2. The following example starts by installing the meson build dependencies and builds the GStreamer core module.

```
RUN \
    apt-get update && apt install -y \
    python3-pip \
    python-gi-dev \
    ninja-build && \
    pip3 install meson

RUN \
    cd /opt/gstreamer && \
    meson build --prefix=/usr && \
    ninja -C build install
```

Listing 4.2: Dockerfile commands to build gstreamer from source

After all the required GStreamer modules are built, we can finalize our Dockerfile, by installing the application dependencies, copy the python source files, set environment variables, and set up the entry point script. When it comes to environmental variables, two variables were required for a fully functioning docker container:

- `NVIDIA_DRIVER_CAPABILITIES=compute, utility, video`: The NVIDIA environment variable controls which driver libraries or binaries are mounted inside the container. For our application there were three necessary driver capabilities [3]:

[3]More information at https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/user-guide.htmldriver-capabilities

  - `compute:` The "compute" driver is required for CUDA and Open Computing Language (OpenCL) applications.
  - `utility:` The "utility" diver provides command-line tools like "nvidia-smi" that can be used for management and monitoring of NVIDIA GPU.
  - `video:` The "video" driver is required for using the Video Codec Software Development Kit (SDK).

- `DISPLAY :1 :` the "DISPLAY" environment variable indicates the X Display server located in the host server, which can be necessary to stream the application window. Additionally, there is also the need to mount the X server volume, which is necessary for the X server to communicate with the X client.

An example of how we can set up, in a Dockerfile, these two variables and also mount the required volume for the "DISPLAY" variable, can be seen in the listing 4.3.

```
ENV NVIDIA_DRIVER_CAPABILITIES=compute,utility,video

ENV DISPLAY :1
VOLUME ["/tmp/.X11-unix:/tmp/.X11-unix:rw"]
```

Listing 4.3: Setup of environmental variables in Dockerfile

To conclude our Dockerfile, we developed an entry point script, which defines how the container will run. For our Dockerfile, the entry point needs to specify the startup of the GStreamer WebRTC application. An example of how we can include an entry point script to the Dockerfile can be seen in the listing 4.4 and the respective entry point script can be looked at in detail in the listing 4.5.

```
COPY entrypoint.sh /
RUN chmod +x /entrypoint.sh

ENTRYPOINT ["/entrypoint.sh"]
```

Listing 4.4: Adding entrypoint script to the Dockerfile

4.4.3.2 Signalling Interface

In the Signalling Server section 4.4.2, we discussed in specific how such a service is used in our application to establish peer connections between different devices. For the GStreamer WebRTC application to communicate with the web client, it needs first to establish a connection with the signalling server, which can be done with the signalling interface.

The class "WebRTCSignalling", interfaces with the WebSocket based signalling server. The class itself is very simple and abstract, and the interface isn't dependent on the usage of the

```
!/bin/bash

FRAMERATE=60
AUDIO=false

while true; do
    python3 /opt/app/main.py --debug \
        --app_name $APP_NAME \
        --framerate $FRAMERATE \
        --enable_audio $AUDIO \
        --encoder $ENCODER
    sleep 1
done
```

Listing 4.5: Entrypoint Script

signalling server, with that, the GStreamer WebRTC application can be independent of any signalling server, which allows us to use, if necessary, different servers with minimal changes in our signalling interface.

The signalling interface is composed of four main phases, which follow the principles discussed in the Signalling Server Connection Handshake figure 4.4:

- `Connect:` Connects to the signalling server with a given address and sends the "HELLO" command to the server with the attributed ID. When looking at the connection handshake sequence 4.4, we can see the method at the beginning of the connection with the delivery of the "HELLO P2" message.
- `Setup Call:` The "Setup Call" method is called after the "HELLO" message from the client is received and initiates the session with the peer by ID, through the delivery of a "SESSION <uid>" message.
- `Send SDP:` Once the connection is established by receiving a "SESSION_OK" message, the interface proceeds by sending the SDP to the peer.
- `Send ICE:` Once the connections are established we send the ICE candidates to the peer.

After following through these phases, the GStreamer WebRTC application can establish with success, when possible, a connection with a peer. In these cases, our peer is the web client, where the streamed content can be visualized. The client, and its corresponding implementation of the signalling interface, are discussed in its respective section.

4.4.3.3 GStreamer Pipeline

GStreamer utilizes a pipeline-based processing model. To fully understand the implementation of the streaming component some pipeline core concepts need to be explained, namely the concept

of "Elements". Elements are the core object when it comes to implementing a GStreamer pipeline, in general, elements receive, process, and output a result. For elements to communicate with each other, they utilize ports called "pads". Each element in the pipeline can have two different pads, the pad through which the data enters an element and the pad through which the data exits the element, these are called the sink and source pad respectively [41]. To further understand these concepts, we can observe the image 4.6, that demonstrates theoretically a simple pipeline example of a basic Ogg player, based on the Dynamic pipelines tutorial example found in the GStreamer documentation [19], and also the listing 4.6 which shows a real pipeline, using the command-line tool, to encode H.264 video streams using NVIDIA's hardware-accelerated NVENC.

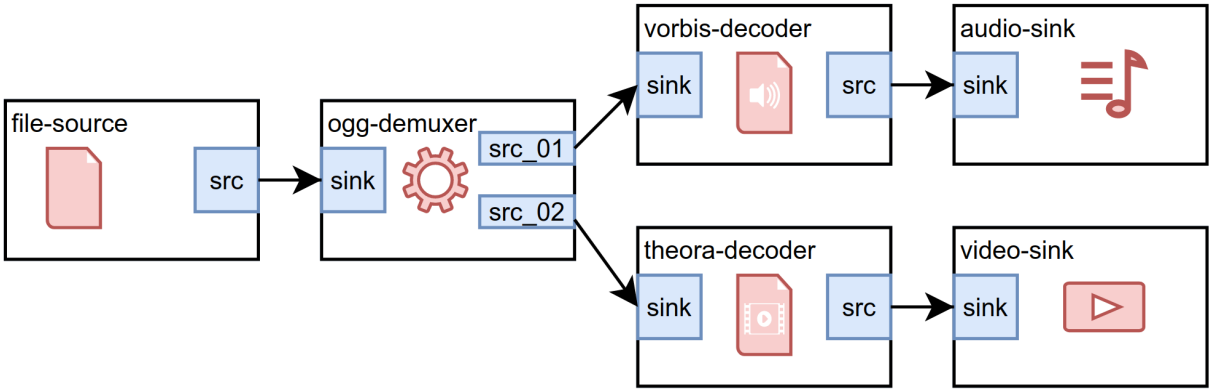


Figure 4.6: GStreamer Pipeline for a basic Ogg player

```
gst-launch-1.0 filesrc location=test.mp4 ! qtdemux ! h264parse ! nvh264dec !
    videoscale ! "video/x-raw,width=1280,height=720" ! nvh264enc ! h264parse !
    mp4mux ! filesink location=out.mp4
```

Listing 4.6: GStreamer pipeline to encode H.264 video streams

Now that some core concepts are explained, we can start exploring what was implemented into our own GStreamer pipeline. The class "GSTWebRTCApp" seen in the GStreamer WebRTC Application Architecture 4.5, contains the implementation of the video pipeline. Next, we describe the reasoning behind each element that was used, in our pipeline and their respective python code.

We start our class by initiating the pipeline and setting up the `webrtcbin` element, here we also set up some initial configuration properties like, connecting to the signalling handlers and adding the STUN server. The listing 4.7, illustrates the class setup, where the pipeline is initiated, by utilizing the "new" method from the Pipeline module. As for the elements that compose our pipeline, each one is created by utilizing the "make" method from the "ElementFactory" module, and their respective properties are configured by using the "set_property" method.

The next step is to assemble the video stream pipeline. For the video stream pipeline, several elements are required, to be able to stream an X11 display to the browser with WebRTC. As

```
self.pipeline = Gst.Pipeline.new()
self.webrtcbin = Gst.ElementFactory.make("webrtcbin", "app")
self.webrtcbin.set_property("stun-server", self.stun_server)
self.pipeline.add(self.webrtcbin)
```

Listing 4.7: Initiating GStreamer Pipeline

observed in the GStreamer WebRTC Application Architecture figure 4.5 our video pipeline has 6 elements:

- `ximagesrc`: The `ximagesrc` element is utilized to capture an X Display Window and creates raw Red, Green and Blue (RGB) video. Two key properties needed to be configured, the first one was the property "xid" which specifies the exact window we want to stream, the second property was the "framerate", by default the element fixates to 25 FPS, setting the framerate target to 60 FPS lowers the overall latency. Some code necessary to create and configure the described elements can be seen in the listing 4.8.

```
ximagesrc = Gst.ElementFactory.make("ximagesrc", "x11")
ximagesrc.set_property("xid", self.windowID)
ximagesrc_caps = Gst.caps_from_string("video/x-raw")
ximagesrc_caps.set_value("framerate", Gst.Fraction(self.framerate, 1))
ximagesrc_capsfilter = Gst.ElementFactory.make("capsfilter")
ximagesrc_capsfilter.set_property("caps", ximagesrc_caps)
```

Listing 4.8: Creating and configuring the `ximagesrc` element

- `videoscale`: The `videoscale` element is utilized to resize video frames. By default, the element always tries to apply the same size on the source and sinkpad so that there is no need to scaling. Because of that, it is always safe to include this element in a pipeline to achieve a more robust behavior without any cost if no scaling is needed. To create the element, we can utilize the element-making method defined earlier, and there is no additional configuration necessary to be done.

- `videoconvert`: Contrarily to the `videoscale`, the `videoconvert` element is mandatory in our pipeline. The element, converts video frames between a variety of video formats, for our pipeline. By default, it converts the RGB buffer coming from the `ximagesrc` element to a NVENC compatible format. Since our application includes multiples pipelines for different types of encoders, the video format for which the element converts might differ based on the chosen encoder. On the listing 4.9, we can observe how the element was created and configured for the H.264 NVENC.

- Encoder: In general, the encoder element, encodes the buffer to a specific video format. Our pipeline supports a variety of different encoders, including Software and Hardware-based encoders, like x264, x264 with NVENC and VP8. The encoder can be chosen by modifying

```
videoconvert = Gst.ElementFactory.make("videoconvert")
videoconvert_caps = Gst.caps_from_string("video/x-raw")
videoconvert_caps.set_value("format", "I420")
videoconvert_capsfilter = Gst.ElementFactory.make("capsfilter")
videoconvert_capsfilter.set_property("caps", videoconvert_caps)
```

Listing 4.9: Creating and configuring the `videoconvert` element

the encoder argument, by default we utilize the element `nvh264enc` which encodes H.264 video streams using the NVIDIA hardware-accelerated NVENC. The encoder element contains the most properties that can be configured to achieve different levels of performance, as expected, the properties vary from the type of encoder chosen. On the listing 4.10 we can observe how the `nvh264enc` was created and the properties that were configured. Two properties made the biggest difference regarding performance:

- `bitrate`: With the `bitrate` property we are able to manage the video quality. The higher the bitrate the sharper the video image can be, which is particularly important when dealing with a streaming service. The video bitrate defines the video data transferred at any given time, because of that, the bitrate value is highly dependent on the client bandwidth. By default, we set the bitrate to "2000", since, in our tests, it was the minimum value that showed good results. The `bitrate` property can be changed while the pipeline is running, which makes it possible for the client to adjust the bitrate value based on the bandwidth that he has available.
- `preset`: For the `preset` property, the "Low Latency High Quality" option was the one that exhibited the best results, a more in-depth explanation of the different presets possible and their results can be seen in the article "Using Netflix machine learning to analyze Twitch stream picture quality" [4], where they analyzed the picture quality of encoded video game footage, across different encoders and encoders properties.

```
nvh264enc = Gst.ElementFactory.make("nvh264enc", "nvenc")
nvh264enc.set_property("bitrate", 2000)
nvh264enc.set_property("preset", "low-latency-hq")
nvh264enc_caps = Gst.caps_from_string("video/x-h264")
nvh264enc_caps.set_value("profile", "high")
nvh264enc_capsfilter = Gst.ElementFactory.make("capsfilter")
nvh264enc_capsfilter.set_property("caps", nvh264enc_caps)
```

Listing 4.10: Creating and configuring the `nvh264enc` element

- RTP Payloader: In general, the Real-time Transport Protocol (RTP) payloader creates a RTP packet to be sent over the peer connection. The element varies based on the encoder and the video format, by default, since we are encoding H.264 video streams, our RTP Payloader is the element `rtph264pay` which payload-encodes the H.264 video coming

from the encoder into RTP packets. The code necessary to create the payloader for the nvh264enc, can be seen in the listing 4.11.

```
rtph264pay = Gst.ElementFactory.make("rtph264pay")
rtph264pay_caps = Gst.caps_from_string("application/x-rtp")
rtph264pay_caps.set_value("media", "video")
rtph264pay_caps.set_value("encoding-name", "H264")
rtph264pay_capsfilter = Gst.ElementFactory.make("capsfilter")
rtph264pay_capsfilter.set_property("caps", rtph264pay_caps)
```

Listing 4.11: Creating and configuring the rtph264pay element

- webrtcbin: The webrtcbin is the bin for WebRTC-based connections. It handles the WebRTC handshake and the contract negotiations, and it is required to be configured before initiating the video pipeline. To finalize the pipeline, the last element, namely the RTP Payloader is linked to the webrtcbin.

To finalize the construction of our pipeline, we need to add each element, that was previously created to the pipeline that was initiated in our class. Each element needs to be linked to each other following the order that was seen in the GStreamer WebRTC Application Architecture figure 4.5. The code necessary to add and link each element to the pipeline can be observed on the listing 4.12 and 4.13 respectively.

```
self.pipeline.add(ximagesrc)
self.pipeline.add(ximagesrc_capsfilter)
self.pipeline.add(videoscale)
self.pipeline.add(videoconvert)
self.pipeline.add(videoconvert_capsfilter)
self.pipeline.add(nvh264enc)
self.pipeline.add(nvh264enc_capsfilter)
self.pipeline.add(rtph264pay)
self.pipeline.add(rtph264pay_capsfilter)
```

Listing 4.12: Adding all elements to the pipeline

4.4.3.4 User Input Handler

To conclude the GStreamer WebRTC application, we have the module "WebRTCInput". The class has the purpose of handling the input commands from the client WebRTC data channel, by utilizing the Python XLib and the pynput libraries to change the window size of our application and also send x11 keypress and mouse events to the X server. To conduct these functionalities, there were five methods implemented:

- Connect: It utilizes the DISPLAY environmental variable, configured in the Docker

```
Gst.Element.link(ximagesrc, ximagesrc_capsfilter)
Gst.Element.link(ximagesrc_capsfilter, videoscale)
Gst.Element.link(videoscale, videoconvert)
Gst.Element.link(videoconvert, videoconvert_capsfilter)
Gst.Element.link(videoconvert_capsfilter, nvh264enc)
Gst.Element.link(nvh264enc, nvh264enc_capsfilter)
Gst.Element.link(nvh264enc_capsfilter, rtph264pay)
Gst.Element.link(rtph264pay, rtph264pay_capsfilter)
Gst.Element.link(rtph264pay_capsfilter, self.webrtcbin)
```

Listing 4.13: Linking all pipeline elements

Configuration section 4.4.3.1, and the `windowID` obtained from the application name argument, in order to establish connection to an existent X server.

- `Send x11 mouse:` Sends the mouse action events to the XServer, by utilizing methods like `mouse.move`, `mouse.scroll` and `mouse.press` from the `pynput` mouse class.
- `Send x11 keypress:` The keypress method sends the keyboard action to the XServer, but contrarily to the mouse method, the keysyms are converted to a keycode by using the `keysym_to_keycode` method from the `XLib` library.
- `Change Window Size:` Changes the size of the application window that is being streamed, to a given width and height. The size of the window is changed by utilizing the `configure` method from the `Window` object obtained from the `windowID`.
- `On message:` The fifth and last method, handles the input messages incoming from the web client data channel, with the format "<command>, <data>". The method supports the following message commands:
  - `kd:` Triggers the key down event by calling the `send_x11_keypress` method.
  - `ku:` Similar to the previous, differs only in that it triggers the key up event.
  - `m:` Handles the mouse events by calling the `send_x11_mouse` method.
  - `vb:` Allows the client to dynamically set the video bitrate while streaming. The change in the bit rate is immediate and does not require a pipeline restart.
  - `cws:` Enables the client to change the application resolution on demand, by calling the `changeWindowSize` method. Such a feature might allow the client to change the resolution to obtain a better performance or might simply be used to adjust the window resolution to match a particular device. The alteration of window resolution requires a full restart of the pipeline for the change to be applied.

A more detailed explanation of how the input is captured and how it is sent through the data channel is done in the next section regarding the Client Web Application.

4.5 Client Web Application

In the system architecture section 4.3, we discussed the existent modules of our application and reviewed the necessity of a client to visualize and interact with the application that is being streamed by the server. The client-side is a JavaScript frontend application that was written in Vue.js, based on the web application from the `selkies-project`, where the necessary changes were made both in the UI and in the application modules, to ensure the web client meets the functionalities we want to achieve. The client web application connects with the STUN server, communicates with the signalling server, establishes peer connection, captures user inputs, and receives and renders the stream.

4.5.1 Web Client Architecture

The web client architecture was based on the client architecture found in the article "GPU-accelerated streaming using WebRTC" [15] and can be seen in the figure 4.7. To ensure we maximize the expandability of the client, the development was divided into five modules, where each one concentrates specific functionalities necessary for the creation of the client:

- index.html: the index.html is the default page shown on the website and contains the various Vue HTML components necessary to compose the UI.
- app.js: consists of the core for the Vue application and encloses the data structures and methods necessary to have a functioning UI.
- signalling.js: contains the implementation for the signalling interface used to establish a connection with the signalling server.
- webrtc.js: encompasses the interface to WebRTC that handles the data channel.
- input.js: handles receiving and sending the keyboard and mouse input events through the WebRTC data channel.

In the following sections, we discuss in detail each one of these modules.

4.5.2 UI

Through the development of the UI, the main objective was to implement a straightforward interface, that was compatible with any device including mobile and desktop and that could demonstrate all the necessary information, such as performance metrics and some settings options, that exemplify possible usages on how to communicate with the server. Regarding the settings, we focused on options that could be used to expose how to possibly communicate with the streaming engine by requesting particular changes while the pipeline is running:

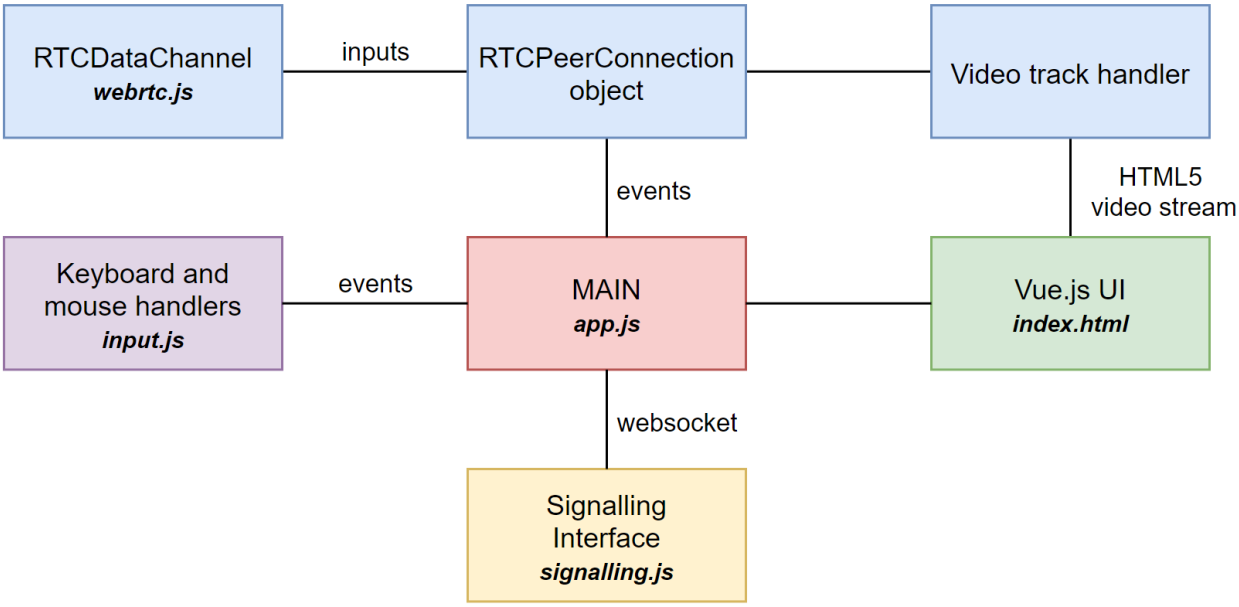


Figure 4.7: Web Client Architecture

- Video bitrate selection: dynamically changes the video bitrate for the hardware encoder on the server. By selecting the video bitrate we can give the user the option to actively change the video quality and bandwidth usage, which might be useful in cases where the internet connection is limited.
- Video framerate: selects the desired framerate limit for the streaming engine. Such options take no effect on the application that is being streamed, instead, it tells the streaming engine to limit the current transmission to the selected framerate. Although we can limit the FPS the performance is still dependent on the server hardware, additionally, the changes made after the selection of the new option can only be applied after a full restart of the pipeline.
- Window Resolution: actively changes the window resolution of the application that is being streamed, such feature can be used to adjust the resolution to match the one of a particular client device and also to regulate the streaming performance, by lowering the resolution we can increase the framerate performance but decrease the visual quality. Although this option takes immediate effect on the streaming engine a window reload is necessary for the changes to be applied to the client.

For the performance metrics, we center our data on the information that could be useful for testing, debugging, and comparing the performance with already existent technologies, therefore the focal point of these metrics are on retrieving data from the server running the streaming engine. The determined performance metrics were the following:

- Video bitrate: compares the current video bitrate in Megabits per second (Mbps) versus the selected bit rate in the options. The bitrate data was used to ensure the streaming

efficiency by making sure our engine wasn't using more bandwidth than necessary, and to aid in debugging the bitrate option setting.

- FPS: displays the current framerate the server is streaming at. With the FPS, we were able to test the performance of different encoders and evaluate how the change in window resolution affected the server performance.
- GPU usage: shows the current GPU load as percentage.
- GPU memory: presents the GPU memory used in Gigabyte (GB) of the total memory available for the GPU.

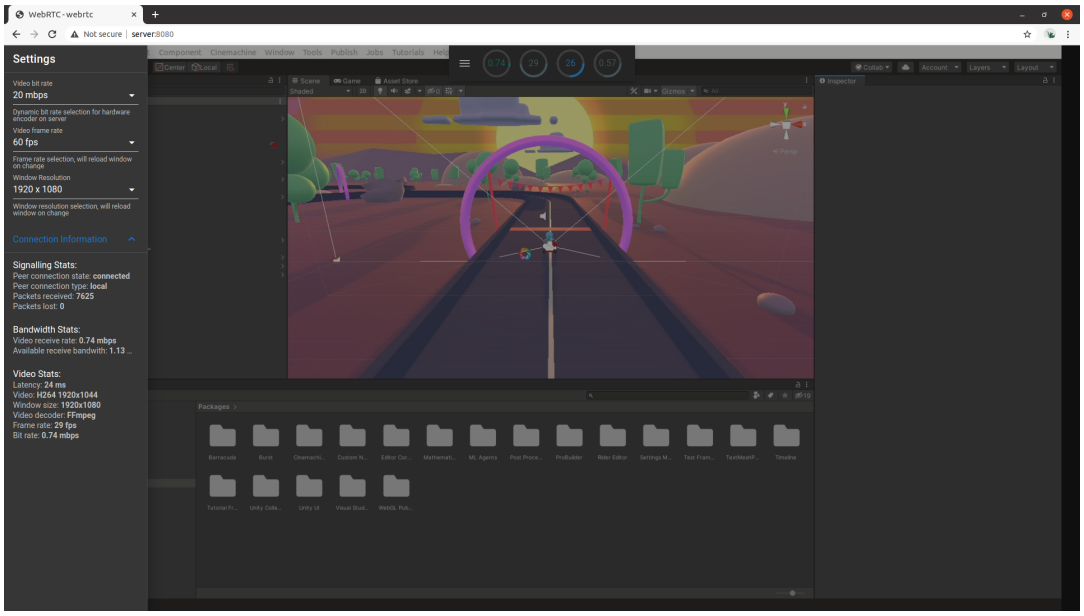
The settings options and the performance metrics were integrated into the interface of the client which is composed of three UI components:

- Toolbar: the toolbar can be observed in the figure 4.8a and 4.9a, for desktop and mobile devices respectively. The component contains a button that triggers the appearance of the navigation drawer, and various elements that show specific performance metrics. Such metrics are composed of tooltips and progress circular elements, the first one is used for conveying messages when a user hovers the element, like information of what metric is being displayed, the second element is used to transmit data circularly to users, the tooltip wraps the second element to integrate the progress circular with on hover messages, minimizing, the overall size of the toolbar by removing the necessity of labels.
- Navigation drawer: the navigation drawer can be observed in the figure 4.8b and 4.9b, for desktop and mobile devices respectively. The drawer component is accessed by utilizing the button presented in the toolbar and accommodates the settings options. The options were implemented by utilizing select field components which can provide the necessary information from a predefined list of options. To ensure compatibility with multiple types of devices, the drawer was configured to be expanded from the bottom of the screen when a smaller window or a mobile device is detected.
- HTML5 Video: the video element is utilized to visualize the video stream. The video stream object is received from the WebRTC connection and is then added to the HTML5 video element by assigning it to the `srcObject` property, which automatically loads the stream.

4.5.3 Client Signalling Interface

The client and the server need to establish a connection with the same signalling server, as a consequence, most of the client implementation for signalling interface and their respective methods share various similarities with the interface that was implemented for the server in the section 4.4.3.2. As seen in the figure 4.10, the core methods necessary in the interface, for the client to establish peer connection with the signalling server, are the following:

(a) Toolbar with performance metrics

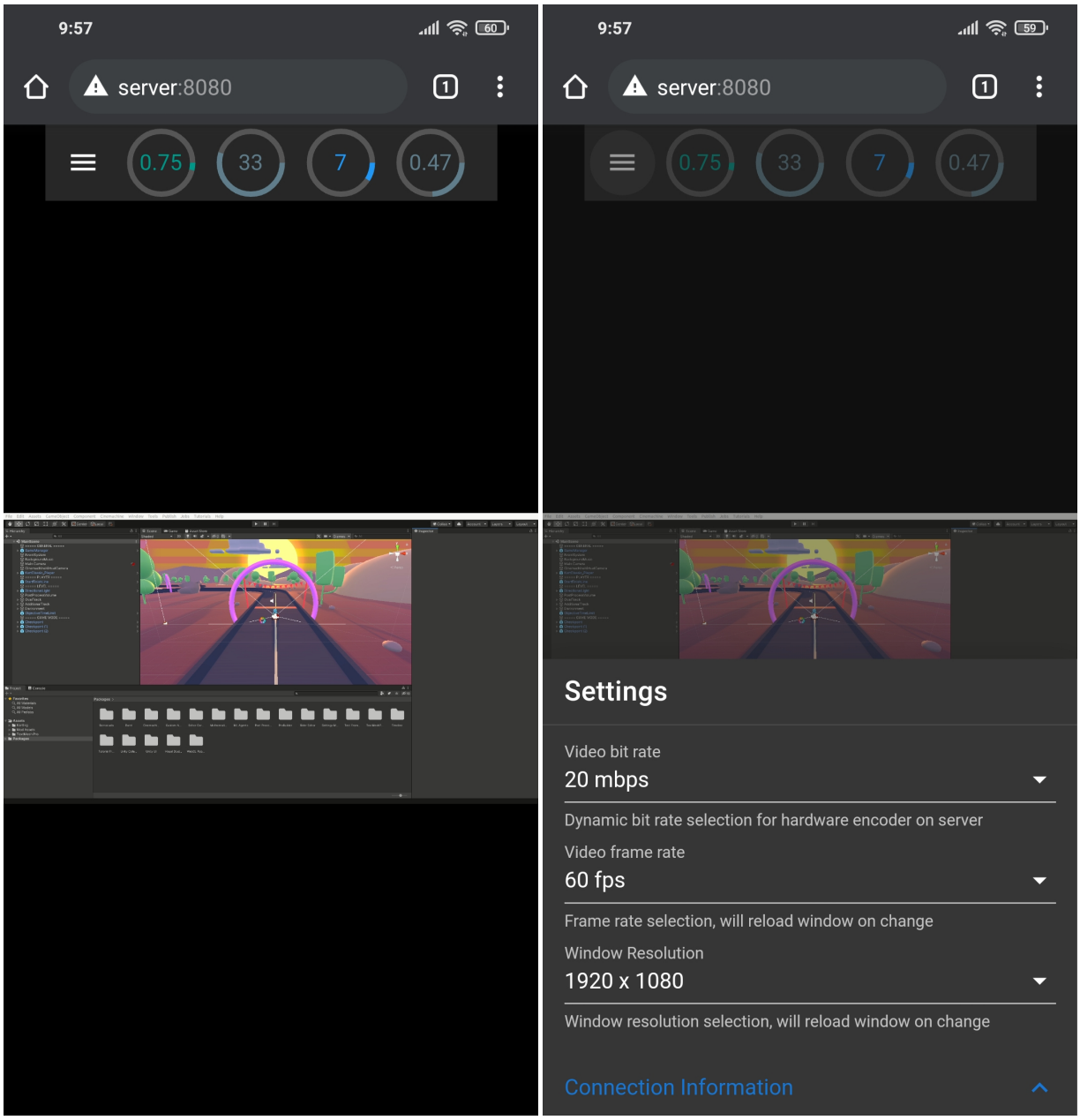


(b) Navigation drawer with settings options

Figure 4.8: Client UI in desktop devices

- `Connect:` initiates the connections to the signalling server with a given address and binds the event handlers. Consequently, a series of handshakes occur between the signalling server and the streaming server to negotiate ICE candidates and media capabilities.
- `Send SDP:` transmits the SDP to the peer.
- `Send ICE:` once the connection is established we send the ICE candidates to the peer.

After the execution of these methods, the client is ready to establish, when possible, communica-



(a) Toolbar with performance metrics (b) Navigation drawer with settings options

Figure 4.9: Client UI in mobile devices

tion with the signalling server. Once the connection is established with success, the web client is qualified to receive the Real-time Transport Control Protocol (RTCP) packets containing the video stream.

4.5.4 WebRTC Interface

The WebRTC class is a simple interface used for managing real-time communications between peers. It has as the main functionality, conducting the initial connection with the signalling

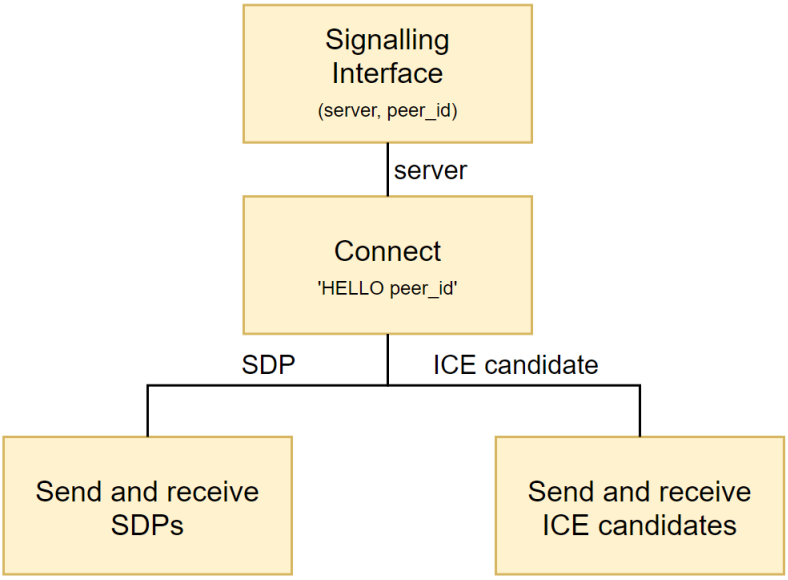


Figure 4.10: Client Signalling Interface

server, the creation of the `RTCPeerConnection` object, managing the receiving and sending of messages through the data channel, and handling the video stream. Such processes are achieved with the following methods:

- `Connect:` initiates the connection with the signalling server, creates the peer connection object, attaches the event handlers to local functions, and binds the callbacks.
- `Play Video:` once the signalling server is connected and the SDP offer is received, the stream is transmitted through the WebRTC connection, when the `RTCPeerConnection` object emits an `RTCTrackEvent` containing the video stream object. Once the stream object is received the `playVideo` method automatically loads the video stream.
- `Send Data Channel Message:` sends messages through the peer data channel. The data channel is used throughout the application for sending messages such as settings configuration and user-generated mouse and keyboard inputs.

4.5.5 User Input Handler

The mouse and keyboard inputs received from the user are managed by the "Input" class which implements the event handlers for capturing, processing, and transmitting input data to the server through the "RTCDataChannel". In the figure 4.11, we can observe the main methods implemented by the class to manage the inputs generated by the user:

- `Key:` the existing library from the `Apache Guacamole`, was used for capturing the keyboard events with the translated X11 keysyms. The keyboard symbols are then encoded as Comma-separated values (CSV) before being sent over the data channel.

- `Mouse Button Movement:` handles the mouse button and motion events and sends them to the server backend. Mouse events are more complex than the keyboard due to the possibility of receiving either relative position or absolute coordinates in mouse events mixed with multiple button presses in an individual event. Another key issue is the requirement of calculating the translation of the mouse position based on the window size, element offsets, and any scaling applied to the video element. Such calculation are done by the "_clientToServerX()" and "_clientToServerY()", which utilize the current window math to translate the pointer position. The x and y positions and the button masks are saved as local variables so that we can send the last know mouse position on the following events.

- `Touch:` manages the touch events generated from a mobile device and transmits the received data by using the data channel. The touch inputs are mapped to mouse events, and similar to the previous, the touch events also require special attention when transmitting the touched position relative to the server.

- `Window Math:` captures the display and video dimensions necessary for computing the mouse pointer position relative to the application being streamed by the server.

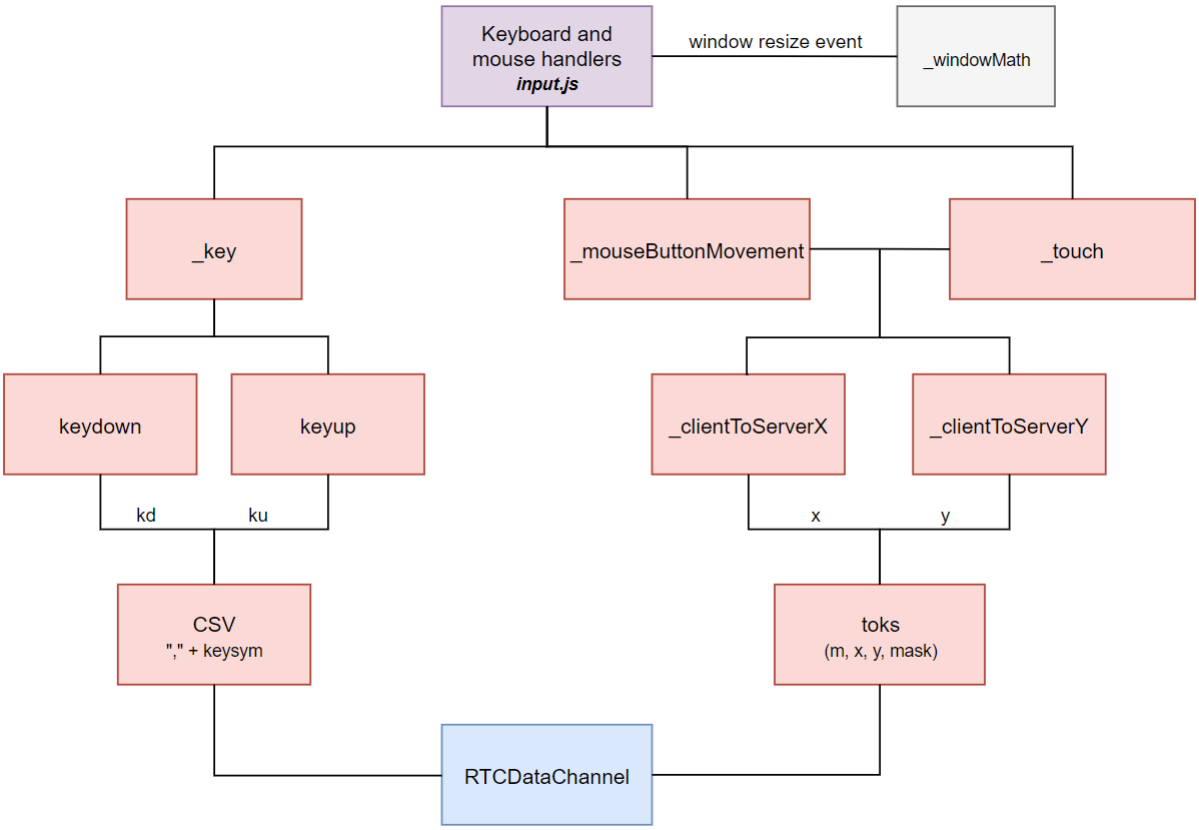


Figure 4.11: User Input Handler

4.6 Conclusion

Throughout this chapter, we specified and described the implementation steps that were necessary to develop a full-stack web-based real-time streaming application, that is capable of visualizing and interacting with an application that is being streamed by a server.

We started by making a specification of what were our requirements and assumptions for the application we wanted to develop. Next, we defined the system architecture and describe in detail the components that were implemented for the server backend. We successfully developed a server backend that was capable of establishing communication with a web client, then we implemented a streaming pipeline by using GStreamer, that can capture any application window, encode the buffer to H.264 by using the hardware-based NVENC and use WebRTC to transmit the video stream to the web client. To finalize the server implementation, we also added support for user-generated mouse and keyboard inputs from the client.

Proceeding the server implementation, we continued with the development of the web-based client. We were able to accomplish the previously defined specification, by developing a simple and efficient client, that does not require any additional software or plugin installation. Additionally, we built a UI that adapts to any device resolution, displays useful metrics, such as bandwidth, framerate, and GPU usage, and provides a set of options, like changing the bitrate and dynamically adjusting the application resolution. To conclude, we implemented methods for handling the input generated by the user, by capturing touch, mouse, and keyboard events and send them to the server.

Chapter 5

Experiments and Tests

In this chapter, we present the experiments and tests that were performed, by the end the reader will be able to fully understand the capabilities and the importance of the implemented application.

When discussing the importance of streaming services, one fundamental advantage is the ability to execute graphic intense applications in devices where otherwise would not be possible. To further evaluate such significance, we start our experiments and tests by comparing the performance of streaming against native rendering on mobile devices.

5.1 Specification

The following experiment is based on performing load balance tests to evaluate the limits of native mobile rendering against streaming and determine the overall drawbacks of rendering graphic-intensive workloads on mobile devices. The evaluation is executed in various mobile devices and in the server that was used throughout the implementation phase, the server experiment is also done while streaming since due to the available computational resources the obtained performance results might differ whether the server is or not currently streaming.

To conduct the analysis, we used Three.js [24] to implement a web application that utilizes WebGL and Instancing [1] to continuously draw the same 3D object indefinitely until the following parameters are met:

- Browser crash: the crash occurs once the page produces more resources than the device can handle, causing the browser to decide to tell all the pages that they lost the context. The crash can be determined once a context-lost message like "WebGL: CONTEXT_LOST_WEBGL: loseContext: context lost" is produced [2].
- Reach defined limit: the experiment for a specific device will stop once a prescribed limit

[1] More information at https://threejs.org/docs/api/en/objects/InstancedMesh
[2] More information at https://www.khronos.org/webgl/wiki/HandlingContextLost

of rendered objects is reached. Due to the available hardware, we specified the limit to 100000 objects, since most of the available devices crashed before reaching the limit.

The application continuously increases the number of rendered objects on the screen and logs the frame rate, number of polygons, and the time it took to draw each corresponding number of objects. When regarding the available devices for the evaluation, we used 4 different devices with various ranges of performance, for the hardware we focused on the CPU, RAM, GPU and the Chipset for mobile devices. The available devices and their specifications are the following:

- Xiaomi Redmi 4A
  - Chipset: Qualcomm MSM8917 Snapdragon 425
  - CPU: Quad-core 1.4 Gigahertz (GHz) Cortex-A53
  - RAM: 2 Gigabyte (GB)
  - GPU: Adreno 308
- Xiaomi Redmi 7
  - Chipset: Qualcomm SDM632 Snapdragon 632
  - CPU: Octa-core Max 1.8 GHz
  - RAM: 2 GB
  - GPU: Adreno 506
- Xiaomi Pocophone F1
  - Chipset: Qualcomm SDM845 Snapdragon 845
  - CPU: Octa-core Max 2.8 GHz
  - RAM: 6 GB
  - GPU: Adreno 630
- Server:
  - CPU: Intel(R) Core(TM) i7-6700HQ CPU @ 2.60 GHz
  - RAM: 16 GB
  - GPU: GeForce GTX 960M

5.2 Execution

Our experiments and tests were executed by developing the WebGL application necessary to evaluate multiple devices. As the base for the application, we took advantage of the Three.js framework and their Instancing Performance example [3]. To improve the efficiency of the experiment, the following adjustments were made to the example:

[3] More information at https://github.com/mrdoob/three.js/blob/master/examples/webgl_instancing_performance.html

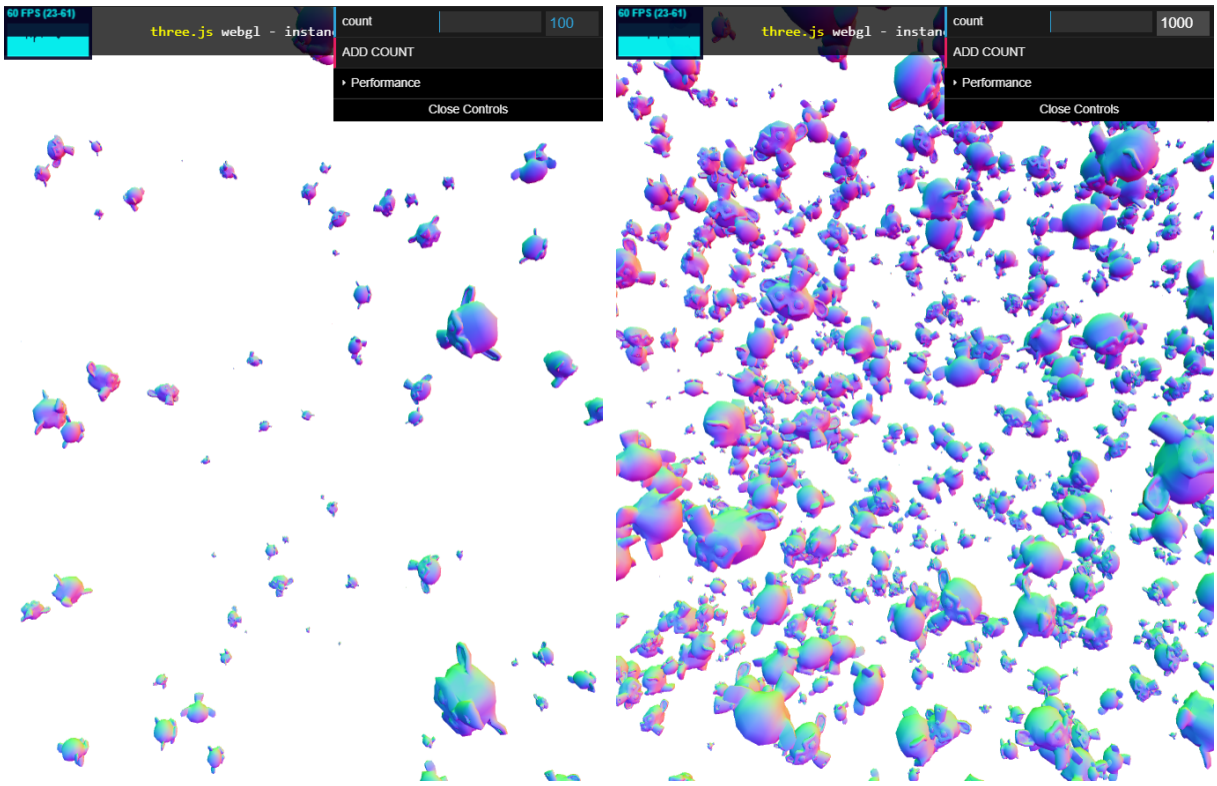
- We started by removing any unnecessary functionalities, to make the User Interface (UI) less cluttered and improve the evaluation on the mobile devices.
- Next, we implemented an "ADD COUNT" button, that incrementally increased the number of objects rendered. By default, the button works in increments of 100.
- For the final change, we added an output, that was necessary for retrieving data to analyze the results. The output implemented, was in Comma-separated values (CSV) format, and included metrics such as the number of objects, framerate, number of polygons, and the time it took to complete the rendering. An example of the output result can be seen in figure 5.1.

```
COUNT,FPS,POLY,TIME(ms)
0,59,0,1.0000000474974513
100,59,0,5.699999979697168
200,59,18179600,3.399999928660691
300,59,26689200,3.9000000106170774
400,58,39163500,11.400000075809658
500,59,57343100,14.700000057928264
600,57,78617100,7.7000000746920705
700,59,102985500,3.0999999726191163
800,59,133446000,2.5000000605359674
900,59,167484400,2.899999963119626
1000,58,204907300,3.2000000355765224
1100,59,244554300,3.9999999571591616
1200,59,293484500,4.799999995157123
1300,58,342221300,4.499999922700226
1400,60,398790800,4.100000020116568
1500,60,466480800,10.500000091269612
1600,58,540456300,2.9999999096617103
1700,59,611627500,4.199999966658652
1800,59,687246900,2.099999925121665
1900,60,762092700,2.4999999441206455
2000,60,859469600,4.499999922700226
```

Figure 5.1: WebGL Application Output

The developed WebGL application and its UI can be seen in the figure 5.2, where we can observe the results at a different number of objects, 100 and 1000 respectively.

After concluding the development of the application, we proceed with the execution of the experiments and tests. The tests were done with the same procedure for the entire set of mobile devices and the server, however, the server evaluation was done while the streaming engine was in execution. For the operation, we utilized the same browser (Google Chrome), and to record the output results and save the CSV file from the mobile devices, we utilized the Chrome



(a) 100 Objects (b) 1000 Objects

Figure 5.2: WebGL Application at different objects counts

remote debugging tool, that facilitated the inspection of a page running on an Android device [4]. By utilizing the discussed procedures, the execution was simple. We started by adding several objects in increments of 100 until one of the previously defined parameters was met. Once the execution was concluded, we utilized the remote debugging tool to save the produced output to an CSV file.

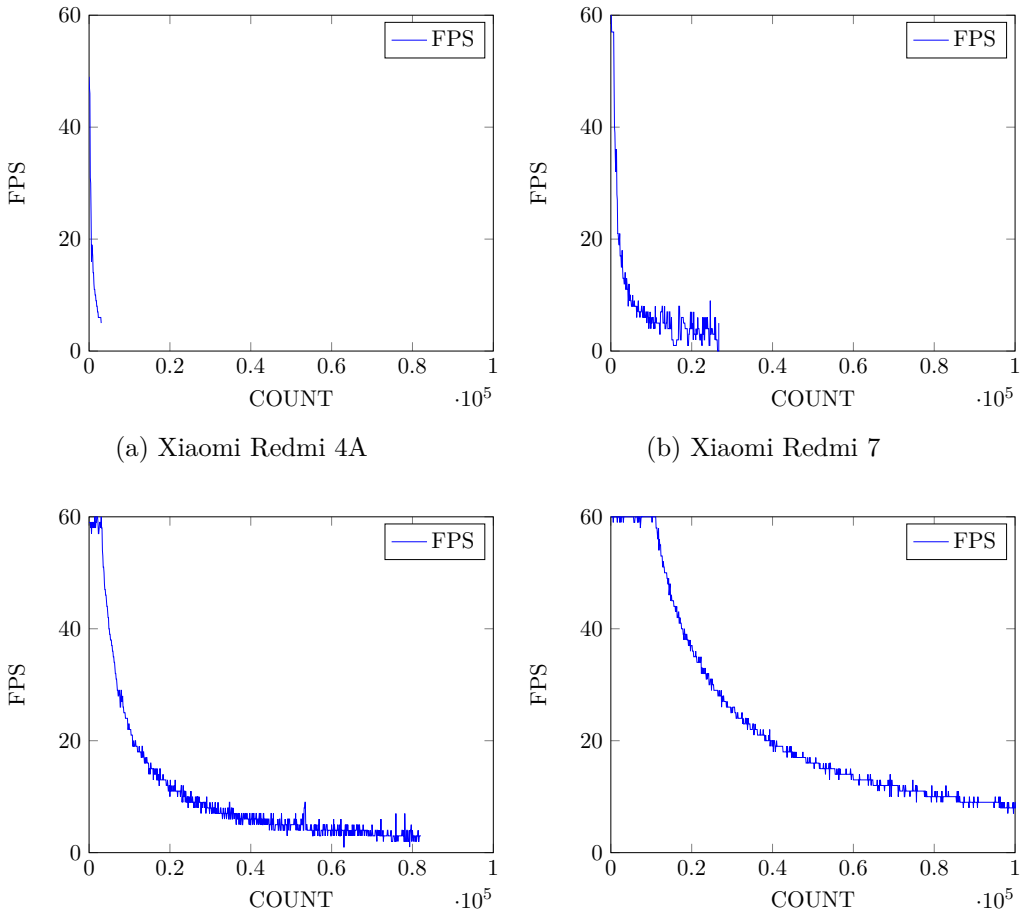
5.3 Results

The first result that we examine from the executed experiment, is the obtained frame rate for each number of instances rendered on the browser. In the figure 5.3, we can observe the results for each device defined in the specification, where the x-axis is the number of instances and the y-axis the frame rate. The results for each device were as following:

- Xiaomi Redmi 4A: the first experiment was done on the Redmi 4A, comparing to the devices on the list, the 4A is the cheapest and has the lowest hardware specification, in consequence, the results obtained were the worst, where the device was only able to render 3000 instances and was never able to achieve a stable framerate.

[4]More information at https://developer.chrome.com/docs/devtools/remote-debugging/

- Xiaomi Redmi 7: the Redmi 7, exhibited performance improvements when compared to the previous, it was capable of maintaining on average 60 Frames Per Second (FPS) until around 700 instances, but from there on, the FPS exponentially decreased as the number of instances increased, until the browser crashed when 26700 instances were reached.
- Xiaomi Pocophone F1: the Poco F1, showed clear improvements in performance from the previous, it was capable of maintaining a stable frame rate until 3500 instances, and rendered on a total of 55400 more instances than the Redmi 7. Still, it did not reach the specified limit, crashing at 82100 instances.
- Server: the server results were undoubtedly superior to the ones obtained from the mobile devices. It was capable of maintaining a stable FPS for a higher number of instances and no crashes attained, reaching the specified limit.



(a) Xiaomi Redmi 4A

(b) Xiaomi Redmi 7

(c) Xiaomi Pocophone F1

(d) Server

Figure 5.3: Frame rate per number of instances rendered

Additionally, the figure 5.4, displays the difference in performance between all devices, where we can observe in more detail the discrepancy in the number of instances that each device is capable of rendering, reinforcing the conclusion that we were able to obtain from the previously discussed plots.

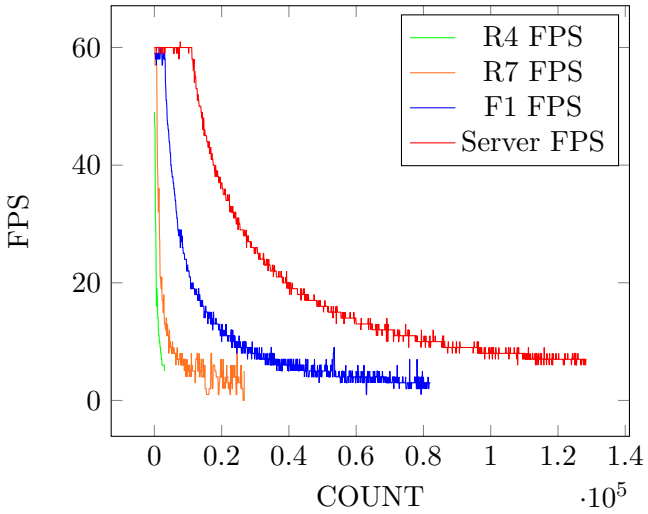
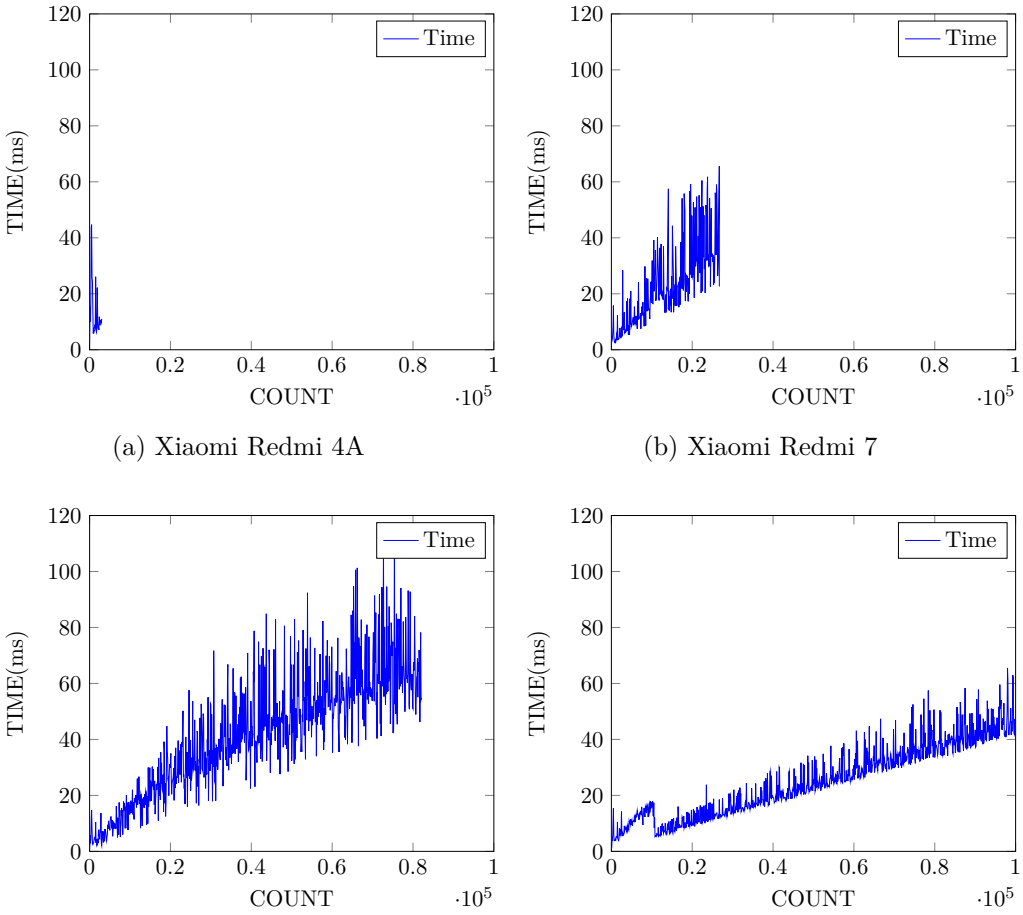


Figure 5.4: Comparison of frame rate per number of instances rendered for entire available devices

Another performance metric, retrieved from the experiment, was the time it took to draw each number of instances on the screen. On the figure 5.5 we can observe in detail the time for each device, where the x-axis contains the number of instances and the y-axis represents the time in Milliseconds (ms). Contrary to the previous metric regarding the frame rate, the results obtained from the current evaluation were considerably more inconsistent. Although there is a clear increase in the time it takes to render as the number of instances increments, we found spikes in performance where it took less time to render higher counts of instances on some particular number of occasions. An example of these issues can be seen in the results obtained from the Xiaomi Pocophone F1, where the time it took to render 82000 instances was lesser than the time it took to render 24600 instances, taking 54.3 ms and 57.6 ms respectively. The reasoning behind the event that might be causing the issue was not found, nevertheless, the data obtained can still be useful for the experiment.

When comparing the results between the available devices, with the figure 5.6, we can observe that for a smaller number of instances, all the devices take on average the same time to render, that said, as the number of instances increases, the server is the only device that can consistently render a higher number of instances in less time than any mobile device. The server was, on average, 30% faster at rendering the same amount of objects than the Xiaomi Pocophone F1.

As expected, devices with better hardware are capable of rendering more instances, through the discussion of the data and the observation of the plots we can observe that the server is much more capable than the mobile devices. Due to the browser crashing, the mobile devices used in the evaluation were not capable of reaching the specified instances limit, conversely, the server was not only able of reaching the limit but could also surpass it if needed. As a result of being apt of rendering a bigger number of objects in less time, the server is more suited for drawing higher counts of polygons than any of the mobile devices available in the executed test. By evaluating the time and the amount of instances that each mobile device is capable of rendering,



(a) Xiaomi Redmi 4A

(b) Xiaomi Redmi 7

(c) Xiaomi Pocophone F1

(d) Server

Figure 5.5: Frame time per number of instances rendered

we were able to conclude, that through the usage of our streaming application it is possible to display a higher number of polygons that would otherwise not be feasible for rendering natively on mobile phones.

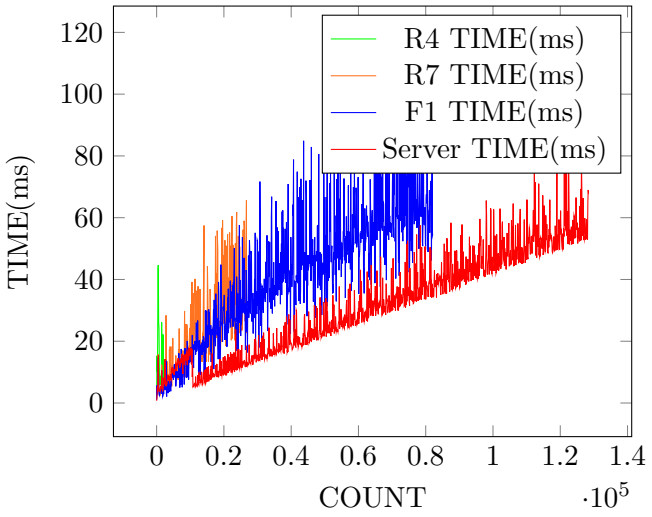


Figure 5.6: Comparison of the time it takes to render per number of instances on screen for the entire available devices

Chapter 6

Conclusion

In this chapter, we describe the problem and iterate through the solutions and their advantages, by the end the reader will be able to conclude what was accomplished, the limitations, and the possible development that can be done in the future.

Throughout the years, there has been an increase in the population that has access to mobile devices. As a result of the advancements made in the hardware, the rendering capabilities of the mobile devices have been increased, still, their performance results are limited on the size and power consumption. To solve such limitations, we can exploit streaming technologies, where the required computational resources are handled by a server and the result is transmitted to the client. Previous streaming solutions were limited by the available bandwidth, but with the increasing accessibility of 5G, mobile devices can accomplish faster internet speeds and lower latency.

Throughout the thesis, we presented a web-based application that is capable of streaming high-fidelity 3D visualizations to the web browser. When streaming high-fidelity content, latency, resolution, and framerate were key metrics that we utilized to determine the best user experience. To accomplish that, we implemented a full-stack application, containing a client and a server backend, where we utilized state-of-the-art technologies such as Web Real-Time Communication (WebRTC), GStreamer, and Vue.js.

The server backend was implemented using Python and GStreamer, the GStreamer multimedia framework served as our streaming engine, where we implemented various pipelines that were capable of utilizing a Graphics Processing Unit (GPU) for Hardware (HW) accelerated streaming workloads and establish a connection with WebRTC to communicate with the client in real-time. The client was developed using Vue.js and JavaScript, where we designed a minimal and functional User Interface (UI) that provided a prototype on how to implement interface components that could communicate with the server backend.

After the development phase, we specified evaluation tests, that showed the importance of streaming services, and the advantages in our implementation, by gauging the constraints of native rendering on mobile devices against our server. By utilizing state-of-the-art WebGL

frameworks, such as Three.js, we developed a web-based testing application that continuously rendered more objects and was compatible with any device that has a WebGL-supported browser available. As a result, we concluded that mobile devices with better hardware are capable of native rendering more objects, but due to the ability to render a much bigger number of objects in less computing time, a server is more suited for drawing higher counts of polygons than any mobile device available.

6.1 Future Work

The web-based real-time streaming application, presented in this thesis, enables the users to visualize and interact with graphic-intensive applications on any device. Nevertheless, throughout the development, there was a variety of improvements that could be done:

- **Streaming Engine:** When regarding the server backend, and in particular the streaming engine, although efforts were made to obtain the best performance possible, such results were still dependent on the hardware available at the time of writing. By updating the GPU used on the server, we could bring instant benefits on performance, and we could potentially enable different pipeline implementations, namely pipelines that take advantage of Compute Unified Device Architecture (CUDA) based workflows.
- **Communication Handling:** Continuing with the server backend, enhancements on the communication between the client and server could be achieved, at the moment, the server only supports streaming a single application to one client, by improving the Signalling server we could achieve the support for multiple clients streaming in simultaneous, as for streaming multiple independent application, the implementation is much more complex, and might require the usage of multiple systems for each user, which could be considerably more expensive, or, as we previously saw in the state-of-the-art with articles such as "A Cloud Gaming System Based on User-Level Virtualization and Its Resource Scheduling" [57] and "Orchestrating GPU-accelerated streaming apps using WebRTC" [16] the usage of virtualization and cloud computing services like Google Cloud Platform (GCP) could be a good solution for a more cost-effective implementation to enable a user-independent streaming application.
- **Device Compatibility:** We focused our implementation compatibility, to support the available devices that we had available during the development, as a consequence, our testing was limited to Android devices and an Ubuntu-based desktop. Although, theoretically, the technologies used are supported by the most popular platforms, and efforts were made to improve the compatibility such as using the WebRTC adapter.js [45], further testing in the future could be done to ensure every platform and web browser has the best performance and user experience possible.

6.2 Final Thoughts

The concept of remotely accessing an application is something that already existed, but, the most commonly used technologies are not suited to handle graphic demanding content. With the recent rise in the availability of Cloud Gaming services, such as Google Stadia [1], NVIDIA GeForce NOW [2] and Microsoft Project XCloud [3], we have seen a technology leap in the performance that it is possible to achieve through streaming, where games can be played with almost no latency and high rendering fidelity. Unfortunately, the advancements made are proprietary, and the services are limited when it comes to controlling what type of content you want to stream. Throughout this thesis, we tried to solve some of these issues, by describing the development required to implement a similar service to the ones described above, while mostly utilizing open-source technologies and being agnostic to the content that can be streamed.

According to GSMA real-time intelligence data, in 2021, 5.27 Billion people have a mobile device in the world [4], which is 67.03% of the world's population. The population that as mobile devices increased 14% since 2017, but the performance of each device is not equal, furthermore, due to increasing prices necessary to acquire a mobile device, not all populations have access to the same performance, in other words, not all people have access to the same content. By utilizing a streaming solution, such as the one we developed during this thesis, we will be able to ensure that a higher number of the population with mobile devices has access to more graphically demanding content. Additionally, with the network improvements, such as 5G, reaching a higher population, more companies will be able to adopt cloud-based applications. With the possibility of moving the most demanding applications to the cloud, mobile devices might require less computational resources, which can improve the costs in hardware manufacturing and the availability of mobile devices to a higher population.

[1] More information at https://stadia.google.com/
[2] More information at https://www.nvidia.com/en-eu/geforce-now/
[3] More information at https://www.xbox.com/en-us/xbox-game-pass/cloud-gaming
[4] More information at https://www.bankmycell.com/blog/how-many-phones-are-in-the-world

Bibliography

[1] Antonino Albanese, Paolo Secondo Crosta, Claudio Meani, and Pietro Paglierani. Gpu-accelerated video transcoding unit for multi-access edge computing scenarios. In *Proceeding of ICN*, 2017.

[2] Babylon.JS contributors. Babylon.js: Official babylon.js website. https://www.babylonjs.com/, 2020. [Online; accessed 09-November-2020].

[3] A. Begen, P. Kyzivat, C. Perkins, and M. Handley. Sdp: Session description protocol. RFC 8866, RFC Editor, January 2021. https://www.rfc-editor.org/rfc/rfc8866.html.

[4] Christian Moore. Using netflix machine learning to analyze twitch stream picture quality. https://streamquality.report/docs/report/index.html#1080p60-nvenc-h264-picture-quality, 2018. [Online; accessed 18-February-2021].

[5] Dan Isla. Selkies open source project. https://github.com/selkies-project, 2021. [Online; accessed 19-February-2021].

[6] Brian Danchilla. Three. js framework. In *Beginning WebGL for HTML5*, pages 173–203. Springer, 2012.

[7] Jos Dirksen. *Learning Three. js: the JavaScript 3D library for WebGL*. Packt Publishing Ltd, 2013.

[8] Edward Hervey. Building gstreamer from source using meson. https://gstreamer.freedesktop.org/documentation/installing/building-from-source-using-meson.html, 2020. [Online; accessed 23-February-2021].

[9] Epic Games, Inc. Unreal engine the world's most open and advanced real-time 3d creation tool. https://www.unrealengine.com/en-US/, 2020. [Online; accessed 16-November-2020].

[10] Epic Games, Inc. Unreal engine developing html5 projects. https://docs.unrealengine.com/en-US/Platforms/HTML5/GettingStarted/index.html, 2020. [Online; accessed 18-November-2020].

[11] Exocortex Technologies, Inc. Clara.io model. animate. render. online. https://clara.io/, 2017. [Online; accessed 16-November-2020].

[12] FFmpeg. Ffmpeg a complete, cross-platform solution to record, convert and stream audio and video. https://ffmpeg.org/, 2019. [Online; accessed 13-November-2020].

[13] FFmpeg. Some of the free projects and programs known to incorporate work from ffmpeg. https://trac.ffmpeg.org/wiki/Projects, 2019. [Online; accessed 13-November-2020].

[14] Goodboy Digital Ltd. Pixijs the html5 creation engine. https://www.pixijs.com/, 2020. [Online; accessed 12-November-2020].

[15] Google Cloud solutions. Gpu-accelerated streaming using webrtc. https://cloud.google.com/solutions/gpu-accelerated-streaming-using-webrtc, 2020. [Online; accessed 30-December-2020].

[16] Google Cloud solutions. Orchestrating gpu-accelerated streaming apps using webrtc. https://cloud.google.com/solutions/orchestrating-gpu-accelerated-streaming-apps-using-webrtc, 2021. [Online; accessed 30-December-2020].

[17] GStreamer contributors. Gstreamer: open source multimedia framework. https://gstreamer.freedesktop.org/, 2021. [Online; accessed 11-March-2021].

[18] GStreamer contributors. Gstreamer bindings. https://gstreamer.freedesktop.org/bindings/, 2021. [Online; accessed 11-March-2021].

[19] GStreamer contributors. Gstreamer documentation. https://gstreamer.freedesktop.org/documentation/, 2021. [Online; accessed 11-March-2021].

[20] Chun-Ying Huang, Cheng-Hsin Hsu, Yu-Chun Chang, and Kuan-Ta Chen. Gaminganywhere: An open cloud gaming system. In *Proceedings of the 4th ACM multimedia systems conference*, pages 36–47, 2013.

[21] Joel Martin. novnc: Html vnc client library and application. https://novnc.com/info.html, 2020. [Online; accessed 28-December-2020].

[22] Axel Karlsson and Oscar Nordquist. Babylonjs and three. js: Comparing performance when it comes to rendering voronoi height maps in 3d, 2018.

[23] Fabrizio Lamberti and Andrea Sanna. A streaming-based solution for remote visualization of 3d graphics on mobile devices. *IEEE transactions on visualization and computer graphics*, 13(2):247–260, 2007.

[24] Lewy Blue. Discover three.js: Welcome to the missing manual for three.js! https://discoverthreejs.com/, 2020. [Online; accessed 04-November-2020].

[25] Salvatore Loreto and Simon Pietro Romano. *Real-time communication with WebRTC: peer-to-peer in the browser*. O'Reilly Media, Inc., 2014.

[26] Luc Barthelet. Unity acquires obvioos, creators of furioos. https://blogs.unity3d.com/2019/11/01/unity-acquires-obvioos-creators-of-furioos/, 2019. [Online; accessed 08-April-2021].

[27] Marc Manzano, Manuel Uruena, M Sužnjević, Eusebi Calle, Jose Alberto Hernandez, and Maja Matijasevic. Dissecting the protocol and network traffic of the onlive cloud gaming platform. *Multimedia systems*, 20(5):451–470, 2014.

[28] MDN contributors. Improving compatibility using webrtc adapter.js. https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/adapter.js, 2020. [Online; accessed 01-March-2021].

[29] Finian Mwalongo, Michael Krone, Michael Becher, Guido Reina, and Thomas Ertl. Gpu-based remote visualization of dynamic molecular data on the web. *Graphical Models*, 88: 57–65, 2016.

[30] Nirbheek Chauhan. Gstreamer signalling example. https://gitlab.freedesktop.org/gstreamer/gst-examples/-/tree/master/webrtc/signalling, 2021.

[31] noVNC contributors. websockify: Websockets support for any application/server. https://github.com/novnc/websockify, 2020. [Online; accessed 28-December-2020].

[32] NVIDIA Corporation. Nvidia video codec sdk. https://developer.nvidia.com/nvidia-video-codec-sdk, 2021. [Online; accessed 08-March-2021].

[33] NVIDIA Corporation. Nvidia container toolkit. https://docs.nvidia.com/datacenter/cloud-native/index.html, 2021. [Online; accessed 23-February-2021].

[34] OL2, Inc. Onlive. http://onlive.com/, 2020. [Online; accessed 25-November-2020].

[35] Tony Parisi. *WebGL: up and running*. O'Reilly Media, Inc., 2012.

[36] PlayCanvas Ltd. Playcanvas the web-first game engine. https://playcanvas.com/, 2020. [Online; accessed 12-November-2020].

[37] S. O'Dea. Number of smartphone users worldwide from 2016 to 2023. https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/, 2021. [Online; accessed 03-May-2021].

[38] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. Rtp: A transport protocol for real-time applications. STD 3550, RFC Editor, July 2003. http://www.rfc-editor.org/rfc/rfc3550.txt.

[39] Ryan Shea, Jiangchuan Liu, Edith C-H Ngai, and Yong Cui. Cloud gaming: architecture and performance. *IEEE network*, 27(4):16–21, 2013.

[40] Shu Shi, Klara Nahrstedt, and Roy Campbell. A real-time remote rendering system for interactive mobile graphics. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 8(3s):1–20, 2012.

[41] Wim Taymans, Steve Baker, Andy Wingo, Rondald S Bultje, and Stefan Kost. Gstreamer application development manual (1.2.3). *Publicado en la Web*, 2013.

[42] Thanh Nguyen. Cloudretro: Open-source cloud gaming service for retro games. https://github.com/giongto35/cloud-game, 2020. [Online; accessed 28-December-2020].

[43] The Apache Software Foundation. Apache guacamole clientless remote desktop gateway. https://guacamole.apache.org/, 2020. [Online; accessed 13-November-2020].

[44] The Khronos Group Inc. Webgl public wiki. https://www.khronos.org/webgl/wiki, 2020. [Online; accessed 13-April-2021].

[45] The Khronos Group Inc. Webgl supported browser. https://get.webgl.org/, 2020. [Online; accessed 03-December-2020].

[46] Tim-Philipp Müller. Gstreamer 1.14 release notes. https://gstreamer.freedesktop.org/releases/1.14/, 2019. [Online; accessed 17-February-2021].

[47] Unity Technologies. Unity real-time 3d development platform. https://unity.com/products/unity-platform, 2020. [Online; accessed 16-November-2020].

[48] Unity Technologies. Furioos: Stream any 3d project anywhere, anytime, on any devices. https://www.furioos.com/, 2021. [Online; accessed 08-April-2021].

[49] Unity Technologies. Unity products: Furioos. https://unity.com/products/furioos, 2021. [Online; accessed 08-April-2021].

[50] Unity Technologies. Unity render streaming. https://github.com/Unity-Technologies/UnityRenderStreaming, 2021. [Online; accessed 09-April-2021].

[51] Unity Technologies. Unity render streaming manual. https://docs.unity3d.com/Packages/com.unity.renderstreaming@3.0/manual/index.html, 2021. [Online; accessed 09-April-2021].

[52] VideoLAN Organization. x264. http://www.videolan.org/developers/x264.html, 2020. [Online; accessed 30-November-2020].

[53] WebRTCforTheCurious contributors. Webrtc for the curious. https://webrtcforthecurious.com/, 2021. [Online; accessed 16-April-2021].

[54] Wikipedia contributors. List of webgl frameworks — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=List_of_WebGL_frameworks&oldid=999964717, 2020. [Online; accessed 12-November-2020].

[55] Xpra contributors. Xpra multi-platform screen and application forwarding system. https://xpra.org/, 2018. [Online; accessed 09-November-2020].

[56] Siyu Yang, Bin Li, You Song, Jizheng Xu, and Yan Lu. A hardware-accelerated system for high resolution real-time screen sharing. *IEEE Transactions on Circuits and Systems for Video Technology*, 29(3):881–891, 2018.

[57] Youhui Zhang, Peng Qu, Jiang Cihang, and Weimin Zheng. A cloud gaming system based on user-level virtualization and its resource scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1239–1252, 2015.