FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Real Time Optimizations for a Web-based Telemedicine Platform

Paulo Renato Almeida Correia

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Daniel Sá Pina

Co-Supervisor: Renato Manuel Natal Jorge

July 28, 2020

Real Time Optimizations for a Web-based Telemedicine Platform

Paulo Renato Almeida Correia

Mestrado Integrado em Engenharia Informática e Computação

July 28, 2020

Abstract

At-home health monitoring devices can prove to be of great value both for patients and medical professionals in the process of making medical consultations more convenient and available. Although not a new field of study, recent advances in technology and infrastructure have made Telemedicine, the act of providing healthcare at a distance, more reliable and capable than ever before. Having numerous advantages over its traditional counterpart, this digital approach to medicine might be crucial to provide access to care to those who are most deprived of it.

While there is research being done in the area, the vast majority of consumer-available solutions still have many disadvantages, like excessive price, the lack of key features and inflexibility or the reliance on proprietary software and specific operating systems, which affect their overall adoption.

The present work will continue the team effort at INEGI and FEUP of building an affordable, robust and open prototype for telemedicine use. This device is capable of real-time Electromyography (EMG) monitoring and transmission so that a medical expert can analyse the sampled data in a remote consultation scenario.

The analysis of this data is made in a web-application context via a modern web browser, where the data from the device is received and displayed. The core of the work presented focuses on optimizing this platform so it is as performant and reliable as other available solutions, with the advantage of being easily accessible through the internet.

The solution entails a custom written a real-time charting library for the web capable of displaying thousands of values per second while maintaining performance. This is achieved through the usage of WebGL, a JavaScript API for rendering high-performance interactive 3D and 2D graphics, that provides a way to execute shader code in a device's graphics processing unit (GPU). It also takes advantage of other modern browser features, like WebSockets, IndexedDB and Web Workers, to provide all the necessary tools to analyse the dataset collected by the device.

The developed software achieves a significantly better performance than other available graphing solutions on the web and provides a feature full analysis platform that is available on any device with an internet connection.

Keywords: Telemedicine, Web Development, WebGL

Resumo

Dispositivos móveis de monitorização remota são uma mais valia tanto para os pacientes como para os profissionais de saúde no processo de fazer consultas mais convenientes e acessíveis. Avanços recentes em tecnologia e infraestrutura de comunicação fazem soluções de telemedicina, o acto de prestar cuidados de saúde à distância, mais capazes e seguras do que alguma vez fora possível. Além de ter várias vantagens em relação a métodos mais tradicionais, esta abordagem digital da medicina pode ser crucial para permitir o acesso a cuidados médicos àqueles que são mais privados deste.

A maioria das soluções de telemedicina disponíveis ao consumidor ainda te m vários problemas, como o elevado preço, a falta de certas funcionalidades ou a dependência de software proprietário e sistemas operativos específicos, que afectam a sua adoção.

O presente trabalho continua o esforço de equipa no centro de investigação INEGI para desenvolver um protótipo robusto, aberto e acessível para o uso em telemedicina. Este dispositivo é capaz de monitorizar e transmitir sinais eletromiográficos (EMG) para que um profissional de saúde possa analisar a informação recolhida, num contexto de consulta remota.

A análise destes dados é feita numa aplicação web acessível a partir de um web browser moderno, em que a informação é recebida do dispositivo e representada graficamente. O trabalho apresentado foca-se na otimização desta plataforma de forma a ser tão eficiente e fiável como outras soluções disponíveis, com a vantagem de ser facilmente acessível através da internet.

A solução requer a criação de uma biblioteca de desenho de gráficos em tempo real para a web, capaz de traçar milhares de valores por segundo mantendo uma boa performance. Isto é alcançado através do uso de WebGL, uma API de JavaScript para renderização de gráficos 2D e 3D de alta performance, que possibilita a utilização da unidade de processamento gráfico (GPU) do dispositivo para a execução do código de renderização. Também são utilizadas outras funcionalidades dos browsers modernos como WebSockets, IndexedDB e Web Workers, de maneira a implementar todas as ferramentas necessárias para analisar a informação recolhida pelo dispositivo.

O software desenvolvido alcança uma performance significativamente melhor que outras soluções de desenho de gráficos disponíveis online e proporciona uma experiência de análise completa, disponível em qualquer dispositivo com uma conexão à internet.

Keywords: Telemedicina, Desenvolvimento Web, WebGL

"A society grows great when old men plant trees
whose shade they know they shall never sit."

Greek proverb

Contents

List of Figures

List of Tables

Abbreviations

ICT	Information and Communication Technologies
EMG	Electromyography
RC	Remote Consultation
SBC	Single-Board Computer
CPU	Central Processing Unit
GPU	Graphics Processing Unit
RAM	Random-Access Memory
PCB	Printed Circuit Board
ECG	Electrocardiography
JS	JavaScript
NPM	Node Package Manager
TS	TypeScript
JSON	JavaScript Object Notation
FPS	Frames Per Second
DOM	Document Object Model
SoC	System-on-Chip

Chapter 1

Introduction

1.1 Context

In recent years the development of information and communication technologies (ICT) has revolutionized the way people live their lives in the contemporary world. The fast developments in the area have transformed many industries, birthed new and emerging ones and can prove to be of great benefit in the health space.

In the modern world, ICT is almost synonymous with the internet. A commodity most people in contemporary society take for granted, that provides access to almost anything with a simple press of a button. But in a time where access seems abundant, more than half of the world population still doesn't have access to essential healthcare services [2].

Recent world events have shown how technology not only can facilitate some of the interactions humans do daily but can also be crucial to provide said interactions at a distance, either when they are not easily accessible or have an associated risk for the intervening parties.

The act of providing healthcare, medical information or any health-related service at a distance using ICT is defined as Telemedicine [3]. This is a solution that can play a critical role in providing access to medical care to the parts of the world that are most deprived of it.

It can have a significant contribution in reducing hospitalization by better monitoring patient's status, it can reduce consultation time and burden for medical staff [4], reduce the cost for patients associated with travel to appointments [5], allow access to high-quality care to those who live in remote locations [6] and most importantly it often leads to an improvement in the quality of care and quality of life of the patient [7].

1.2 Motivation

At-home monitoring devices, most commonly in the form of wearable devices, have been a staple of telemedicine for a long time. They allow healthcare professionals to more accurately monitor patient progress from the comfort of their home [4]. However, most of these solutions are only available for people who already have ease of access to other medical resources like hospitals and

clinics. The availability of such telemedicine resources to people who would most benefit from it is still limited either by infrastructure or cost [8].

Advances in the fields of non-invasive sensors, that make them cheaper and more resource-friendly, in wireless technologies, that provide a cheaper and more broad access to the internet and in the production of inexpensive small embedded systems has made the viability of low-cost wearable devices a possibility [9].

While significant, affordability is not the only major hurdle in the adoption of these types of health enabling technologies. The lack of a solution that is easy to deploy and requires no additional software or service to the user is also one it's inhibiting factors [8].

A telemedicine system mustn't only provide the functionality needed to conduct medical consultations remotely, but also easily integrate with the existing organizational structure of the entity providing care. The ideal system offers a seamless experience of a traditional healthcare consultation in a user-friendly way so that both the caregiver and recipient feel comfortable while employing this new approach to medicine [10].

1.3 Objectives

The present dissertation will continue the effort of developing an affordable and open telemedicine device that measures electromyography (EMG) signals. These signals have a variety of uses in clinical and sports medicine, from diagnosis to monitoring and rehabilitation. This device aims to be used in a remote consultation setting, where a patient following a doctor's guidance can collect all the needed information without leaving their house.

One of the distinct characteristics of this device is the use of the web platform to conduct all the necessary interaction between both parties. In this way, the prototype can be used in tandem with any device capable of accessing the internet the user might have available.

The main objective of the work is to develop a web platform that is fast, robust, easy to use, and accessible on any modern web browser. This platform is then integrated with the existing prototype and is where the medical professional will analyse the patients' data during a consultation.

The software developed should be comparable in capabilities and reliability to other available solutions in the area, with the added benefit of being easily accessible through the world wide web. To achieve this the implementation should leverage the power of modern features available in the web browser like WebGL, Web Sockets and Web Workers.

1.4 Document Structure

The present document is organized into 7 chapters that detail the work done. The present chapter (Chapter 1) presents the context and motivation of the work, and briefly introduces the objectives of this dissertation.

A background on telemedicine solutions, followed by a review of currently available approaches for remote consultation and monitoring, like mobile applications, general use applications and specific use applications is presented in chapter 2.

Chapter 3 extensively details the project this work is part of, from it's overall goals to the existing prototype. It describes the implementation efforts previously done, both in the hardware and software experience, followed by an exploration of its software limitations. The chapter ends with a brief proposal of the work needed to solve or minimize said limitations.

In chapter 4 this proposal is discussed in more detail, with an overview of the goals and requirements for the present work. This is followed by an implementation proposal, a more technical description and justification of the technologies and approaches to be used during development.

Chapter 5 explores in great detail the challenges and solutions found during the development stage of the project. It justifies the technology stack used, describes the application architecture and explores implementation details for the most challenging features of the built software.

Following that, chapter 6 compares the built solution to the previously implemented one to outline the improvements made. Comparisons are also made to other state of the art graphing solutions to detail how the implemented software better handles the problem presented.

Finally, the document ends with a brief conclusion in chapter 7, that details the main contributions and difficulties of the project, followed by a description of future work that can be done to improve the current solution.

Chapter 2

Telemedicine solutions

Telemedicine is a broad area of study that encapsulates a lot of different techniques and approaches. The current work exists as a part of the development of a device for telemedicine to be used by physicians to deliver healthcare appointments at a distance. The present chapter offers a review of the state of the art of telemedicine solutions in the same field of remote consultation.

This chapter starts with a brief review of the history and market of telemedicine solutions in section 2.1. It's then followed by section 2.2, where an analysis of the presently available devices and services that provide access to remote consultation capabilities is done. This analysis is further divided into sub-sections that examine three distinct approaches to this particular type of telehealth.

2.1 Background

Telemedicine has a long history, dating as far as 1900 with physicians being amongst the first to adopt the telephone to deliver consultations at a distance. However, due to the limitations of the medium, it never reached a high rate of adoption [11].

Modern advances in ICT have made the disadvantages associated with telemedicine, such as the cost of communication and hardware, become less of an impeding factor while making its advantages increasingly apparent. The infrastructures in place that enable fast internet access to millions can also revolutionize communications between physicians and patients opening up the possibility for preliminary diagnosis before hospital arrivals, distance check-ups and remote medical consultations [12].

Besides solving limitations related to distance, computerized systems can also be used to facilitate other medical procedures like data collection and analysis. Health enabling technologies, such as wearable devices, are regarded as one of the primary means to support the maintenance of high-quality care in the coming years. With the increase of the elderly section of the population and average life expectancy [13], the monitoring need of patients also increases, and at-home monitoring solutions might be crucial in order to provide medical care to the growing amount of persons in need [8].

Consequently, the telemedicine market has been experiencing an unprecedented expansion, with a lot of entities interested in capturing this emerging industry. According to Statista, the global telemedicine market is valued at 30,5 Billion USD in 2019 and is expected to grow to 41 Billion USD by 2021. A 2018 European Commission market analysis [14] states that the demand for telemedicine solutions outpaces their supply, however, financial limitations on the part of the consumer or the medical institution can be prohibitive for the adoption of such technologies.

2.2 Remote Consultation Solutions

Telemedicine is a very broad area of focus, covering a wide range of products, services and techniques. Any part of the health sector affected by ICT can be defined as telemedicine, however, this project will focus on a specific subsection that will be referred to as Remote Consultation (RC), the ability to conduct a medical appointment at a distance. There are numerous ways of connecting patients and doctors using technology, it's been possible ever since the invention of the telephone, however, newer approaches present new opportunities for the field. The following subsections will detail some of these modern approaches and provide some context to what are their strengths and weaknesses.

2.2.1 Mobile applications

With the internet being so ubiquitous in modern society, its usage in communication is prevalent. Most modern approaches to connect users and medical professionals take advantage of this existing infrastructure, relying on the users' ability to connect to the internet.

The smartphone has also become a predominant part of modern life, in 2016, 79% of persons aged from 16 to 74 used a mobile phone to surf the internet [15], making it the primary device type used to do so. Considering this, a large number of RC services use mobile operating systems, such as Google's Android and Apple's iOS, as their only supported platforms.

Services like Push Doctor [16] from the UK, Swedish company KRY [17] with a service by the same name, Teladoc [18] operating in North America and the Portuguese startup knok [19] all offer a similar product. A mobile application that connects the user to a trained medical professional via video conference, using the device's built-in camera and microphone. After connected both parties can have a regular consultation via the internet, however, the physician has no means of evaluating the patient by doing any type of physical exam, making this type of service inadequate for cases where more information from the patient is needed.

These types of services are mostly offered by private companies that give consumers an alternative to going to a physical clinic. Their usefulness is more in preliminary diagnosis than patient monitoring and recovering.

Some services like Babylon Health [20] offer a way for the user to track their health using consumer-facing wearable devices like smartwatches. These devices can give a better context of the patient status in consultation but the physiological and environmental data that can be collected from them is still very limited and can be hard to verify its validity [21].

Despite the smartphone being the most used device type to access the internet, only being available for such platforms is still a limiting factor for some users. This is especially pronounced for the older range of the population, where is less likely for a person to own a modern and capable device, or simply less likely to be technologically apt to utilize one.

Some companies like Doxy.me [22] rely on open web standards to provide their platform to users. This service is identical to the ones described earlier but its availability through the browser, make it more accessible to the consumer since it can be used in any internet-capable device.

2.2.2 General Use Solutions

Another approach to RC is to offer a device with all the needed capabilities to provide a reliable diagnosis at a distance. These devices offer all the major sensors usually found in a general practitioners' office and allow the medic to evaluate a patient from a distance, based on their symptoms and the measured data.

Some companies offer all-in-one telemedicine kits so that no other equipment is required. These kits are complete with a computer device, like a small laptop or a tablet to connect to the internet and a large number of sensors and medical devices to allow for a thorough examination of the patient. Products like VSee's Telemedicine Kits [23] and Remote Health Solution's VER Line [24] are examples of this approach. They both offer two distinct lines of products, bigger and more capable models made for durability and use on the field and smaller consumer-facing solutions for at-home use.

A different and more recent solution is the Tyto device by TytoCare [25] that connects with a mobile phone for use in RC with medical professionals via their service. This small device has a large number of sensors like a high definition camera with an otoscope and tongue depressor adaptor, a stethoscope and a thermometer. This can be seen as a hybrid approach as it relies on the user having a smartphone but enhances its capabilities with additional hardware.

While these kinds of solutions are very versatile and allow for the testing of a significant amount of physiological parameters their main value is mostly on preliminary diagnosis. For use in follow up consultations or patient monitoring, a significant amount of sensors might prove to be redundant if most are not utilized, and not justify the high price of such devices. Another disadvantage of buying these types of equipment is that they are locked to a specific vendor platform, even if the user fully owns the device it might be useless if not paired with its required software service.

2.2.3 Telemedicine devices in academia

For use in follow up check-ups and patient monitoring, a custom purpose-built device that can measure the needed parameters with great precision can prove to be a better solution when only specific details of the patient status need to be checked. This scenario is common when the initial diagnosis is complete but routine check-ups have to be made in order to assess the development of the problem.

The large majority of the work and research related to telemedicine in academia falls in this category. Many researchers focus on a specific technology and try to improve upon it by making it smaller, more precise or work remotely in a consultation environment.

Examples of such devices include a heart rate monitor that wirelessly monitors a patient's heart rhythm in real-time with the aid of a smartphone [26], or another heart monitor that sends an SMS message to a doctor if any abnormality is detected [27]. Another example is of a low-cost digital stethoscope that interfaces with a communication device and can amplify, store and transmit the acquired signal to an expert for later analysis [28].

While specific use devices are the most common, research has also been done in general use solutions. An example of this is the device developed by Prodhan et al. [29] capable of collecting seven different vital signals from a patient such as blood pressure, oxygen and glucose levels in the blood, body temperature amongst others. This is all accomplished by a low-cost portable telemedicine kit that pairs with an Android enabled device to collect and transmit the acquired data.

While serving as proofs of concept, most of these devices are still in a prototyping stage and require further development to be finalized as end products that can be deployed and used by actual patients. Furthermore, they are also dependent on external communication devices, like smartphones or computers, that handle the communication with the physician in a telemedicine context.

2.3 Summary

Telemedicine is an old field of study that has recently become more relevant with the advances in ICT and infrastructure. Wearable devices capable of monitoring patient status might be crucial to provide care to ever-increasing demand. As such the market for said solutions is experiencing unprecedented growth.

The majority of consumer-available solutions for remote consultations are simple video conference applications that connect a medical professional to the patient in need. While convenient for the user, these solutions are limited as the physician can't properly evaluate the patient's status via a simple video call.

General use devices are an available solution to this problem. They come equipped with a large number of sensors and are capable of providing a reliable diagnosis at a distance. Conversely, they are often expensive and a large number of sensors might be redundant if most are not utilized.

In academia, the majority of research efforts done is in specific use devices. Custom-built hardware that monitors or records specific physiological signals that can be then transmitted and analysed by an expert remotely.

Chapter 3

Telemedicine Device for EMG capture

This dissertation exists as a part of the research being done in the Institute of Science and Innovation in Mechanical and Industrial Engineering (INEGI) and the Faculty of Engineering of the University of Porto (FEUP) on the development of a portable device for telemedicine capable of monitoring EMG signals.

In the present chapter an overview of the goals and previous work done on the device are given in section 3.1 and its hardware implementation are detailed in section 3.2. Section 3.3 specifies the prototype's software implementation and section 3.4 summarizes its current limitations.

3.1 Device Objectives and Background

As explored in the previous chapters there are many approaches to delivering medical consultations remotely. Some rely solely on devices a user might already own, like a smartphone, others have custom hardware but require integration with an existing platform or service.

As a contrast to this, one of the goals of the device being built is that it works as an independent self-contained unit. Everything from signal acquisition, video conferencing, data processing to data transfer is done in the device, with the only requirement being an internet connection. A user shouldn't need to worry about understanding or configuring the device, as all of this functionality should be abstracted from them while providing a ready to use experience with the press of a button.

Another goal is to be platform agnostic. As stated before, the user is only required to have an internet connection, but the professional conducting the consult is still required to have an external device to interact with the prototype. To maintain the platform open, all the data analysis and interaction done during the appointment are made via a modern web browser, found in any internet-capable device such as a laptop or smartphone. This allows easy access to the platform, no software has to be pre-installed or patched, and a session can be started instantly just by connecting to a web page. Developing for the web can also guarantee that the experience will always be the same, regardless of what device type or operating system the user has available.

In addition, the device should be affordable. All the parts used for its build are rather inexpensive with the goal of making the prototype accessible to a larger audience.

Since the device is being built as a collaboration between a multidisciplinary team, one of the core principles is also extensibility. The work done is expected to be expanded upon in the future to add new features to the device or fix any unwanted behaviour.

Work on the device was started in 2014 by Pina et al. [30] with the development of a portable prototype capable of capturing and processing EMG signals that could be monitored via the internet.

This work was further explored by Rafael [31] in 2019 with the design and development of a custom printed circuit board making the device smaller, cheaper and faster.

Currently, work on the project is being done in parallel by two dissertations. One is the work detailed in this document whose contributions will be proposed and detailed further ahead. The other is a re-implementation of the user interface according to feedback from medical professionals and the addition of a video module to the prototype, allowing for video conferencing to be done from the device.

3.2 Hardware Implementation

The prototype that currently exists and is being worked on already has a robust hardware implementation as a result of extensive work done by all the aforementioned projects. This section will provide a brief overview of this implementation to better contextualize the device's features.

The core component of the prototype is a small single-board computer (SBC) that manages and connects all the other components such as the signal acquisition module and the Wi-Fi transmitter along with processing the acquired signal and transmitting it. The software being run in this computer will be explored in the next section.

In its current iteration, the SBC used is an ODROID-C1, that features an ARM 1.5GHz quad-core central processing unit (CPU), an on-chip graphics processing unit (GPU) and 1GB of random-access memory (RAM). This model was chosen as it is smaller and cheaper than other available devices while still maintaining all features needed for the operation of the prototype, making it ideal for this use.

Connected to this SBC is a custom-designed printed circuit board (PCB) made for signal acquisition. This board includes an ADS1298 programmable integrated circuit that allows for up to 8 input/output channels with several possible configurations, making it flexible for the measurement of various signals. It should be noted that this allows for the capture of other biological signals, like Electrocardiography (ECG) to track cardiac activity, with minimal modifications to the hardware.

Electrodes used to attach to the patient's skin are connected to the PCB either via an available 3.5mm audio jack port, making the prototype convergent to similar products in the market, or directly to the board's inputs.

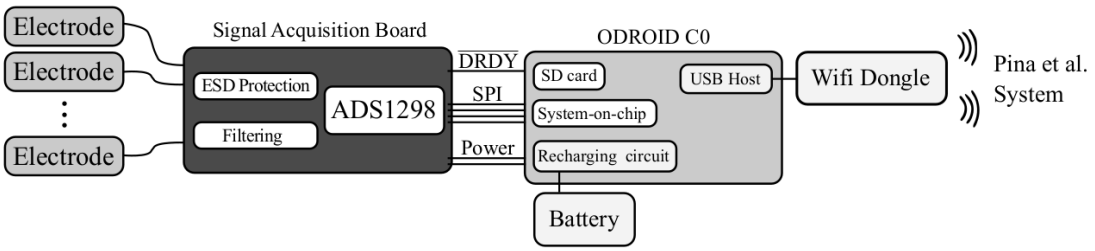

Figure 3.1: Block diagram of the prototype architecture.

A diagram of the described architecture can be found in Figure 3.1 and further details about the implementation and the decisions behind the used components can be attained in [31].

3.3 Software Implementation

One of the core principles of the project is to be easy to use, this has implications on the software side. The device should be able to easily work without requiring any technical knowledge from the user.

To achieve this, the SBC embedded in the device starts several functions when it is first powered on, that are run simultaneously on different threads of execution:

- **Starts the acquirement and processing of the data coming from the PCB:** Some of the data might be corrupted so a pre-processing is used to remove any unwanted values;
- **Stores the captured data:** The acquired data might be useful to analyse subsequently so it can be stored in an external SD card;
- **Starts a Web Server:** The professional conducting the consultation will then connect to this server to analyse the data that is being acquired.

The implementation for all of the functionalities described above is done using Python 2.7[1] the language chosen when the project was started. Although Python isn't the fastest language, compared to others such as C or C++, due to being an interpreted high-level language, it allows for faster prototyping, development and debugging. This gives it an advantage over faster but more restrictive languages. It was also chosen for having the status of being the *default* programming language for academia, making it ideal for teaching device capabilities and further extension and development in the future.

The Web server is built with a Python open-source web framework called Tornado [32]. This library was chosen because of its non-blocking network I/O making it ideal for applications that require a long connection with the user, such as a remote medical examination.

[1] At the time of writing this dissertation end of support for the language has been announced and it should be updated in future work.

The device serves a web page for analysis that can be connected via an internet connection. When the page is loaded on the client-side, the browser opens a WebSocket connection with the server through which further communication is made. The WebSocket API [33] is a communication protocol that allows a client and a server to exchange messages without any polling from either side over a TCP connection.

This protocol is used to transfer the live data being captured by the device to the web page where it is being displayed for real-time analysis. When a connection is established by the user the server begins to send, in regular intervals, packages of information with the captured data until the connection is closed or the device is deactivated. In Figure 3.2 a simple timing diagram of the detailed communication can be seen.

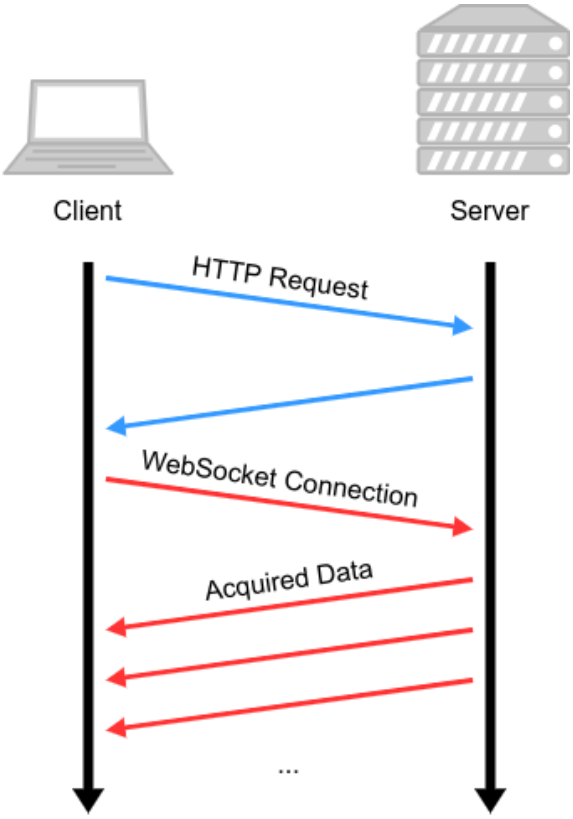

Figure 3.2: Time diagram of the communication between the client and the server.

The client-side of this exchange runs solely on a browser window and can be accessed in any modern internet-ready device, such as a smartphone or a computer. The implementation is a simple web page written in HTML and CSS that has a JavaScript (JS) component to initiate the WebSocket communication and properly format and display the arriving data. This is done with the aid of a charting library called CanvasJS [34] that takes advantage of the HTML5 Canvas browser API to plot the retrieved data to the screen.

The packages of data being periodically sent consist of 8 values one for each of the device's acquisition channels encoded in the JSON format. The user then has the ability to choose what

channel they want to plot on screen using the 2 available charts (Figure 3.3).

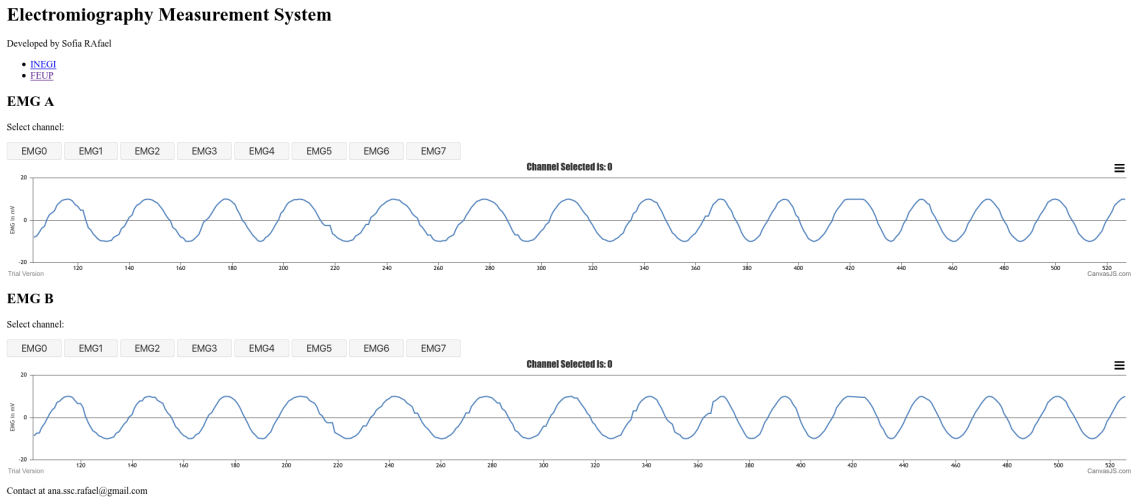

Figure 3.3: Screenshot of the current implemented interface.

3.4 Software Implementation Limitations

The detailed implementation works as a proof of concept for the device but has a lot of crucial flaws that would prevent it from being effective in a real-world telemedicine scenario. This section will highlight some of these shortcomings and detail some of the reasons why they are considered as such.

The first problem with this implementation is that the graph displayed on the web page does not fully depict the data being sampled by the sensors, the volume of data being transmitted is lower than required. The EMG signal ranges between approximately 50 and 500 Hz, so according to the Nyquist theorem the acquisition rate should be at least 1000 Hz, however, the device only communicates values to the client at a rate of about 30 Hz. Thirty times every second a message is sent to the client with the last registered values for each of the 8 acquisition channels, from which the selected ones are plotted to the corresponding graphs.

This discrepancy leads to a disregard of a large amount of data, about 97% of all values acquired are not shown on the client-side, and while this is enough to approximate the corresponding chart, a lot of precision is being lost in the transaction. If a spike was to occur in the interval between the values being transmitted by the device, this peak might be completely missed or otherwise lose some of its amplitude.

Latency is another problem, there is a noticeable delay between the acquisition of a value and its display on the client-side. For a device intended for real-time use, this might not only affect the overall user experience but also the user's trust in its underlying technologies [35]. This problem is

further aggravated as the latency increases with prolonged usage rendering the prototype unreliable and impractical for use in long consultations.

Another downside of this implementation using the CanvasJS library is data persistence. There is a maximum number of values displayed at any time per graph. When a value is added and the maximum threshold is already reached the left-most value being displayed is then removed. This results in the data only being available for analysis while it's not shifted outside of the window range by new incoming values.

Additionally, values are only added to one of the charts on display, therefore 6 of the 8 channels are not being displayed and their data is being completely discarded. In this iteration, there is no way to analyse more than 2 channels at once given that if the user changes a chart to plot a different channel the previous channel information will no longer be recorded.

A final limitation worth noticing is the usability of the web page itself. In its current form, the interface (Figure 3.3) consists only of the essential elements to test the prototype: 2 graphs and buttons to toggle between channels. However, for the device to meet its requirements the experience should be easy and seamless for the end-user and therefore the interface should be intuitive and accessible for even the less technologically competent users.

3.5 Proposal

The goal of the current work is to solve or minimize all of the limitations outlined in the previous section, in order to advance the prototype to a more complete state. This will be done with a special emphasis on performance and data fidelity to provide a professional experience, comparable to its competition, using only the browser.

To achieve this several parts of the experience will be re-written using different technologies and a custom solution will be built to solve specific challenges.

Ultimately the experience of analysing the data acquired from the device should be easy, efficient and powerful, providing the doctor all the necessary tools to conduct such analysis remotely.

In chapter 4 this proposal will be further detailed, with an in-depth analysis of the proposed methodologies and its corresponding justifications.

3.6 Summary

This chapter detailed the vision and work already done in the device being developed for, contextualizing future chapters. Being built as a collaboration in INEGI, the prototype aims to solve some problems detailed in the previous chapter, such as cost, ease of use and platform dependency.

The device aims to function as a simple, easy to use telemedicine kit, where a patient requires no additional equipment or software to take part in a remote consultation. The appointment is all done via the web so it's fast and easy to access by medical professionals regardless of their available devices.

The core of the device is an SBC that powers all other functionalities. It's connected to a PCB custom made to monitor EMG signals and a Wi-Fi dongle to deploy and transmit this information for analysis.

The bulk of the work previously done was on the hardware side with the software remaining in a very basic state. This is enough for a prototype phase of development but in order to be used for its intended purpose, some features and optimizations are required. Some of the flaws that need to be addressed are a lack of precision in the graphs being rendered, latency between signal acquisition and display and the inadequate analysis features such as having no data persistence and no ability to analyse multiple graphs.

This work proposes to solve all of these flaws and ultimately making the prototype and its software experience more robust, reliable and easy to use for the end-user.

Chapter 4

Proposed Solution

As seen in the previous chapter, the work being presented is contextualized in a team effort to develop a telemedicine device and platform that relies on open technologies to provide a robust experience. In a project of such scope, there are multiple different domains where work needs to be done.

The present chapter details what its focus is in the context of this project and expands upon what and why modifications were made. Firstly the objectives of the proposed solution are stated in section 4.1. Section 4.2 details all the functional and non-functional requirements of the proposed work and their significance. This is then followed by an implementation proposal done in section 4.3, that is divided into sub-sections regarding each facet of the proposed implementation, that tries to tackle all the requirements described previously.

4.1 Objectives and Focus

The main objective of this dissertation is to optimize the software experience delivered by the prototype so it provides a platform for telemedicine comparable to other more mature solutions in the area. The final product should be as reliable and performant for analysis as proprietary software with the added benefit of running entirely on the web.

A major focus will be on how to reliably send and display the data captured by the device in real-time. A re-write of the data displaying functionalities in the browser will be done, using a custom-built charting solution, so that performance is guaranteed regardless of the volume of data being displayed.

Changes to the way data is transmitted and encoded will also be made, some to complement the developments made in the client-side, such as how data is stored, some to simply reduce the volume and rate of unnecessary information being sent.

It should be noted that even when focusing only on the web experience of the platform there is a lot of other improvements needed in other to make it fully ready and usable. Areas like video conferencing, authentication and user interface all need to be either reworked or implemented, however they are not the focus of the current project and will instead be addressed by future work.

As a result, most of the contribution done is centred around the scripting part of the web page, changing how data is stored and represented, with additional work done in the server side to how it's transmitted and encoded.

The project is considered successful if a significant improvement is done in this area even if the full vision of what the platform is intended to be, a ready to use telemedicine platform, is not fully realised.

4.2 Requirements

With a better understanding of what is the main focus of the project, this section presents all the functional and non-functional requirements for the overall work being proposed.

4.2.1 Functional Requirements

For all the work related to the rewrite being done, the only user that is going to interact with the system in a meaningful way is the medical professional, since the patient only has to turn on the device and follow instructions. Table 4.1 details all the user stories for this particular user, illustrating the functionalities required for the system associated with this dissertation. A lot more features are needed for a complete remote telemedicine scenario but these are the ones that pertain to this project.

Table 4.1: User stories related to the proposed system.

ID	Name	Description
US01	*Data History*	As a medical professional, I want to be able to view data previously acquired for a channel so that I can go back and look at something I might have missed.
US02	*Switch Between Graphs*	As a medical professional, I want to be able to switch between which channel is being charted per graph, so that I have more control over what I'm analysing.
US03	*Zoom and Pan*	As a medical professional, I want to be able to pan and zoom the charted data, so that I can analyse it with further detail.
US04	*Observe Multiple Graphs Simultaneously*	As a medical professional, I want to be able to observe multiple graphs simultaneously, so that I can compare and relate data from different channels.

4.2.2 Non-Functional Requirements

The previously described functionalities are the ones needed for the system to be on par with what is expected. Non-functional requirements are constraints that guide the implementation of said functionalities. Table 4.2 details these requirements.

Table 4.2: Non-Functional Requirements of the proposed system.

Name	Description
Website Performance	The website experience while displaying, moving and adjusting the data should maintain a constant performance without noticeable frame drops or halts that might hinder the experience.
Graph Precision	The data being charted should match the data being sampled by the device, therefore no information should be sacrificed during the transmission between prototype and client.
Low Latency	The charting of the data should occur as close to real-time as possible, the latency between value acquisition and display shroud be negligible and not interfere with the overall experience.
Resource Efficient	The platform should be as resource and power efficient as possible and not require an overwhelming use of device memory or CPU to properly function.

These requirements are considered the biggest challenge of this project, and managing to fulfil them all, without greatly sacrificing each other, is the main contribution the present dissertation attempts to make.

4.3 Implementation Proposal

In order to fulfil all the requirements described above, this work proposes to completely rewrite the scripting part of the website experience. This section details the modifications proposed, some of the reasoning behind them and how their implementation is done.

4.3.1 Data Transfer

The biggest change to the data transfer layer is the volume in data transferred. To fulfil the *Graph Precision* non-functional requirement the amount of data transmitted between the device and the website will increase. Changing from sending 8 values at a 30 Hz rate (30×8 values per second) to transmitting all the acquires values by the device sampling up to 8 different channels at 1000 Hz (1000×8 values per second) would theoretically increase the needed bandwidth by about 97%.

To mitigate this increase, data compression will be applied to before transmission. Several compression methods will be compared and tested before one is chosen for the problem. The chosen algorithm not only has to have a good compression ratio but also has to be fast and efficient as to not put extra strain on the battery life of the device.

Another change to the data transfer layer is in the rate of messages being sent. Rendering the full frequency of the graph doesn't imply sending individual values at this higher rate. Values will

be bundled and compressed together, resulting in less number of calls needed to send the same amount of information. This results in less strain to the web browser, as it needs to handle fewer incoming messages, and also in reduced bandwidth usage since larger chunks of information are more easily compressed.

4.3.2 Data Visualization

The essential part of the platform is the data visualization charts. These are updated in real-time with the collected information and are where the medical professional can conclude on a patient's status.

A completely new solution for displaying this information on the screen will be written, with the intent of being both fast in rendering but also responsive for the user regardless of their device's capabilities. To achieve this, the proposed solution will take advantage of WebGL [36], a low-level browser rendering API.

WebGL is based on the OpenGL ES 2.0 API and provides a way to execute code on the device's GPU via the browser. It's also fully integrated with the rest of the web standards allowing for WebGL elements to be used in conjunction with more standard HTML elements found on a web page. This API is implemented by all major browsers and is both available on desktop and mobile devices with a coverage of about 97% of global devices [37].

Most commonly used for 3D rendering applications, like video games or scientific visualization [38], its benefits can also be used in the rendering of 2D scenes. Comparing it to other 2D rendering solutions available in the browser, like HTML5 Canvas and the now less utilized Flash player, it shows a remarkable decrease in rendering times because of the reliance on GPU acceleration as seen in Table 4.3.

Table 4.3: Simulation and rendering times per frame for 100,000 sprites being rendered in Chrome by different rendering technologies in [1].

Implementation	**Simulation Time (ms)**	**Rendering Time (ms)**	**Total Time (ms)**
Flash	56.3	174.4	230.7
HTML5 Canvas	59.3	391.0	450.3
WebGL	54.7	4.3	59.0

Another advantage of WebGL is that it allows for lower power consumption and CPU usage. When the rendering is being handled by the device's GPU, less strain is being put on the CPU consequently improving battery usage [39].

4.4 Summary

This chapter details the focus of the present work in optimizing the data visualization parts of the platform. It's expected that the implemented solution has all the features needed for a thorough analysis of the data in the browser while still maintaining performance, precision and being resource-efficient.

This will be accomplished by a full re-write of the charting code on the web using the WebGL API, a browser feature that allows deferring the rendering to the GPU to achieve better performance. Additionally, the transfer layer will also be overhauled, in order to transfer and analyse a larger volume of data. This will be achieved by bundling data and compressing it to reduce the number of sent messages and bandwidth.

Chapter 5

Development

During the course of this dissertation, a robust charting solution for the web was developed, meeting all the requirements outlined in the previous chapter. Throughout development, different technologies were employed, comparisons between similar approaches were carried out and decisions about the implementation were done in order to reach the final product.

The present chapter details all these challenges starting with section 5.1 that describes the technology stack of the project. Next, the primary library used to build the solution, Three.js, and its core concepts are explored in section 5.2 followed by an overview of the project's architecture in section 5.3.

Section 5.4, Implementation Details, is divided into multiple subsections related to different aspects of the final solution. Each of these subsections thoroughly explores its implemented functionality as well as the reasoning behind it, further exploring the difficulties faced when developing the software.

5.1 Development Infrastructure

The right development infrastructure is a critical element of any software project. Not only should the program fulfil its requirements and function as expected, but its implementation should be easy to understand, maintainable and easily scalable for any future work.

The current project is part of a joined effort with a multidisciplinary team and is expected to be built upon in the future. Accordingly, the choice of tooling and guidelines for the development process was made with this in mind to ensure the code is easy to understand and maintain after its original development period is over.

The bulk of the work proposed in this dissertation targets the web platform, narrowing the number of available solutions to choose from. All modern web pages are built on top of 3 standards: HTML, CSS and JS[1], thus a solution built for the web will consequently rely on these 3 languages.

[1] Recent developments to web standards allow the use of WebAssembly on modern browsers but that is outside of the scope of this work.

Since the work is mainly focused on displaying graphs using WebGL, a JS rendering API, the little need for markup and styling required was done using plain HTML and CSS files. However, for the scripting part, a more complex solution was employed.

The popularity of JS as a general-purpose language as risen ever since the release of Node.js in 2009, and with it, what is called the "JavaScript package ecosystem" [40]. Using the Node Package Manager (NPM) a user can install JS packages, pieces of software that implement particular features, that were developed by other people and are freely available to use in any software project. NPM makes the process of installing and maintaining these dependencies easier for developers.

The proposed solution takes advantage of several open-source and free to use libraries, such as, Three.js, localforage, zlib and threads.js that abstract certain browser APIs or implement needed features that are not available in the browser by default. These libraries will be further explored in the next sections when relevant.

To manage all these different dependencies without having to keep track of multiple JS files, the library packages are installed using NPM and the project is built using Webpack [41], an open-source JS module bundler.

Webpack concatenates all the written code, as well as any imported libraries, into a single minimized output file that is then included by the HTML file, removing the hassle of dealing with multiple imports, versions and downloads. This way the codebase can be broken down into files that serve a singular purpose and depend on one another, making the code easier to read and debug.

Webpack also offers other features useful in the development process, such as plugins for asset management, bundle optimization and variable injection, as well as a robust development server and source maps for easier debugging.

To further aid with code quality the project is written in TypeScript (TS) [42], an open-source language maintained by Microsoft that adds static typing to JS. TS was designed for the development of large applications for the web and offers a robust module system, classes, interfaces and other static type declarations. Features like these allow for development environments to better understand the code base and offer more information to the developer, even while it's being written, such as warnings and errors.

This provides a contrast to standard JS programming where problems are only encountered during runtime which can be an extra hassle during development. Another advantage of the language is making the code easier to read and understand for any future developers as the type declarations offer more information about functions and classes and how they interact with each other.

Ultimately the TS code written is transpiled to JS and bundled with all of the needed imported libraries by Webpack as stated above. A schematic of the system described can be found in Figure 5.1.

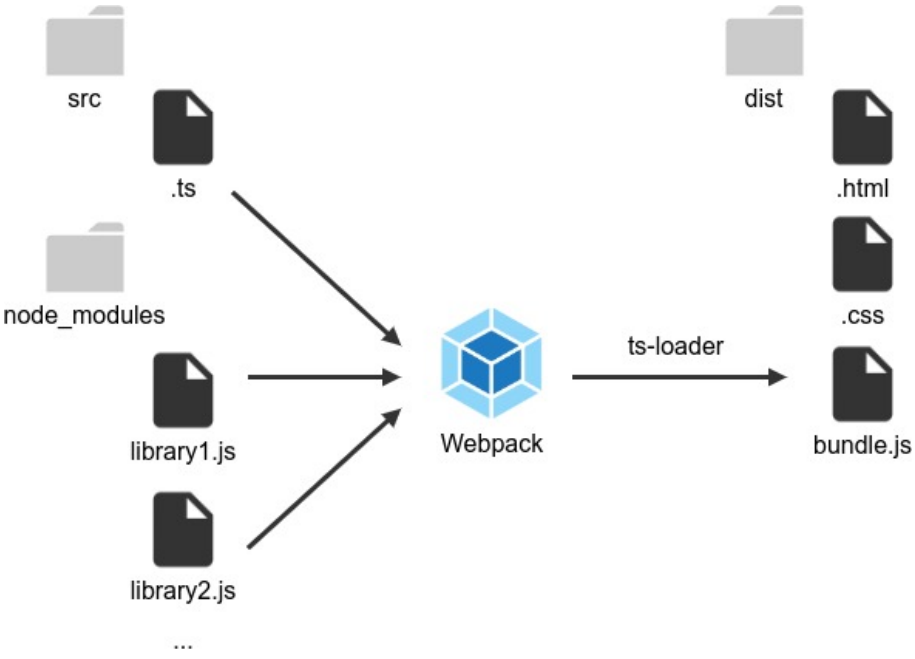

Figure 5.1: Diagram of the web part of the project.

5.2 Three.js

As stated in the solution proposal (Chapter 4) the built charting implementation will take advantage of the WebGL browser API to render to the screen. This API is effectively a port of OpenGL to the browser, so it provides the same low-level functionality as its C written counterpart but in a native JS interface. While powerful, this graphic programming approach can be very verbose and unnecessary for more simple rendering needs, as is the case with the present project.

In this way the Three.js library [43] was used to interface with the WebGL API. Three.js is a wrapper library that abstracts the low-level execution of WebGL and greatly eases the process of setting up the necessary code for an in-browser graphics rendering pipeline.

In this section, some key concepts and functionalities of Three.js are detailed as they are necessary for understanding the architecture and implementation of the developed solution.

As with most 3D rendering engines, there is the concept of a `Scene`, where all that is being rendered is stored. Much like a virtual environment, objects can be placed in 3D space within this scene, and subsequently moved, rotated or scaled in all 3 dimensions. Since what's intended to be drawn in this project is a 2D graph, all objects are placed in the XY plane of the scene and have no depth in the Z-axis.

The Three.js objects used in the implementation are the following: `Line`, `LineSegments`, `Group` and `Sprite` objects. Both `Line` and `LineSegments` are used to draw straight lines between points while applying a certain material. The difference between the two is that `Line` only draws a segment between 2 points while `LineSegments` can draw multiple lines connecting adjacent points in a geometry buffer. A `Sprite` object serves to render static images in a scene

and is used in the context of this project to display the numerical values associated with the scale of the graph. Finally, the `Group` object is used to group multiple objects and apply transformations, such as scale or translation, to the group instead of each object individually. This is used in several different ways throughout the implementation.

To know what portion of the 3D space is being projected to the user's screen and what objects are visible a `Camera` has to be defined. The `Camera` works just like a real-life camera does, it's placed in the scene and using parameters like Field of View, Aspect Ratio and focus planes, it projects what it *sees* to the screen. For this project, the camera is outside the XY plane while being focused on it in order to capture the objects there being placed.

Lastly, to finish the Three.js graphics foundation a `WebGLRenderer` needs to be defined. This is the object that handles all the rendering to the screen. It requires a `Scene` and a `Camera` and when called it renders what the camera is capturing to an associated canvas HTML element using the WebGL API.

5.3 Architecture

The tool built uses an object-oriented approach in its organization structure. With the help of TS static type declaration, the code is broken into different classes that each represent and implement a part of the system and rely on each other to build its complex functionalities.

These classes and the relationships between them can be seen in Figure 5.2. Their purpose and functionalities are as follows:

- **`index.ts`** - Represents the root of the file being called when the bundle is imported by the browser. It instantiates the needed components for the functioning of the project and handles the connections between them. It invokes the WebGL rendering pipeline when the browser's `requestanimationframe()` method is called;

- **`WebSocketManager`** - Handles all of the steps required to connect and read messages coming from the device via the WebSocket API. It has two callback functions `onMessage` and `onError` that are called when the respective action occurs. This class is also responsible for decompressing and decoding the messages being sent, a functionality that is described further ahead;

- **`TimedValues`** - Describes how the data sent from the device is modeled in the client. A single time floating-point number corresponds to an array of data numbers, one for each of the channels being sampled;

- **`Graph`** - Represents a single graph in the system. It has a `Scene` and a `Camera` associated with said graph and keeps it's own internal state. It also has distinct X and Y-Axis and zoom values allowing for different graphs to show completely different zoom levels and value ranges while being analysed;

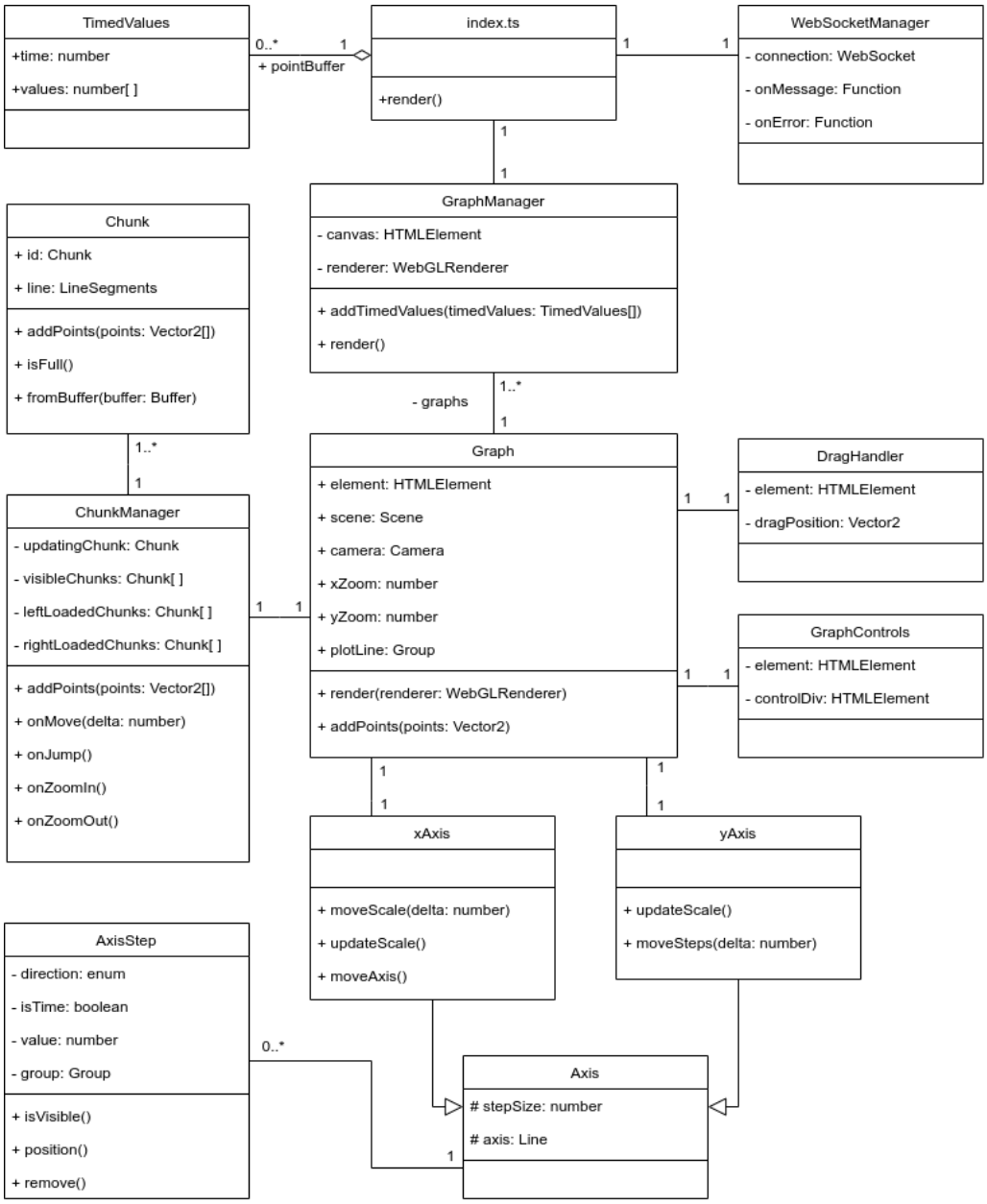

Figure 5.2: Class diagram of the project.

- **`GraphManager`** - Handles all the necessary integration with the multiple components of the system. It stores all the individual graphs and handles passing received values to each one. It also contains all the needed code to render the multiple graphs to the screen and is where the Three.js `WebGLRenderer` is defined;

- **`DragHandler`** - Instantiates and manages the event listeners required to handle user input. It supports both mouse and touch events, implementing the needed logic to calculate the drag velocity and move the associated chart according to the input;

- **`GraphControls`** - Creates and handles the input on a child element of the graph containing all the buttons needed to interact with the graph;
- **`Axis`** - Represents one of the axis of the graph, implementing all the shared functionality between both of them;
- **`xAxis` and `yAxis`** - Both inherit from `Axis` and implement the specific functionality for each axis, such as different unit types, step length and scale adapting logic;
- **`AxisStep`** - Handles all the necessary code to create a step in an `Axis`, both creating the sprite that displays the number associated with that step as well as the tick line where it intersects the axis. Also implements logic related to positioning and moving said steps according to the current graph's zoom values;
- **`Chunk`** - Represents a portion of the data being drawn by the graph, displayed as a collection of connected points stored in a `LineSegments` object. It has a maximum defined number of data points it can hold and can be instantiated from a previously stored buffer of information;
- **`ChunkManager`** - This class handles all the information, stored in chunks, associated with a graph. It contains an array of `visibleChunks` that are the ones currently displaying to the screen and 2 buffers where chunks are loaded to improve performance if the graph is panned or zoomed out. It also has all the code necessary to store and recall all of the graph's information from device local storage to keep a data history.

5.4 Implementation Details

This section focuses on specific parts of the implementation, detailing the challenges they introduced and how they were ultimately solved.

5.4.1 Transfer Layer

One of the main challenges of the project was the method to transfer the data from the device, in the possession of the patient, to the client-side, being used remotely by the medic. The transfer method used in previous implementations, WebSockets, is the most reliable approach to transfer real-time data to a website, therefore, the previous implementation was reworked.

The WebSocket API is built on top of the TCP protocol where, after a connection is established, messages between both parties can be exchanged freely without any polling from either side. The biggest issue faced for the final implementation was how to send the large amount of data being collected to the website in real-time, whilst being efficient in bandwidth usage.

To mitigate this problem two factors are taken into account, the frequency of messages being sent and the structure of the messages being sent. The details regarding the frequency are, however,

influenced by the message structure implementation, so the latter will be broken down and detailed with the former being addressed when relevant.

5.4.1.1 Data Formatting

The way the messages being sent are structured have a large impact on their size. The standard encoding for messages on the web is the JavaScript Object Notation (JSON) format, that is universally used and easy to understand for both humans and computers.

However, when transmitted via the web, a JSON object is converted to text which is then sent encoded in the UTF-8 format. Numerical values that were previously stored in a double-precision floating-point format, that occupied a total of 4 bytes, are now sent as text with each character occupying 1 byte, making numbers with high precision up to 4 times larger.

The solution is to send messages as binary data, that is then interpreted in the client-side. This requires a standard way to encode the message so that it can be properly decoded without any data loss.

The prototype device samples values at a rate of 1000 Hz and since it's impractical and unreliable to send each value individually, after a certain interval, values are bundled together and sent as a single message.

The data collected is stored as a relation between the timestamp when they were sampled and the sampled value itself. Since the device has 8 channels we can remove duplicate data by grouping 1 timestamp to 8 values, one for each channel being sampled. This way, the message would be structured as groups of 9 numbers, occupying 4 bytes each, with the first number of each group corresponding to the timestamp and the following 8 to the values associated with each channel in sequential order. Using this method, for n groups of values being sent per message the bundle size of the message is $n*9*4$ bytes.

This value can be optimized further when taken into account the fact that the sample rate of the device is always constant. Considering this, this rate can be closely approximated in the client-side just by knowing the first and last time stamps of the bundle and the total number of groups of values being sent. Using this optimized method, the message would consist of 2 initial values corresponding to the first and last timestamp respectively followed by groups of 8 values for the channels. This method would provide a bundle size of $(2+n*8)*4$ bytes for n group of values sent.

The efficiency of all the described approaches above can be seen in Table 5.1, where the size of the average data packet and the total bandwidth of a 30 second transmission of a generated signal sent at a rate of 6 Hz for each approach are compared.

Table 5.1: Comparison of average packet size and total bandwidth of the data format alternatives detailed.

	Avg. Size	Total Bandwidth
JSON	20237.86 B	120752.56 Bps
Binary Format w/ All Timestamp	5201.80 B	31037.40 Bps
Binary Format w/ 2 Timestamps	4645.93 B	27720.71 Bps

5.4.1.2 Data Compression

Another way to reduce bandwidth is to apply a compression algorithm to the bundle of data being sent to reduce its overall size. This, however, presents a trade-off that conflicts with one of the non-functional requirements of the system: low latency between capture and analysis of the data.

Both compressing and decompressing the data increases the delay between acquisition and display. The final solution should be one that has a good compression rate but also a minimal impact on the overall time it takes for the signal to be rendered.

Before doing a comparison of multiple compression libraries, some general principles of compression were studied and taken into account when choosing the frequency at which the messages are transmitted. Compression works by removing redundancies and repetition found in the data, that can be reverted afterwards to reconstruct the original file [44]. Thus, the smaller the data-set is the harder it is to compress since it has less redundancies.

Taking this into account it was decided to send bundles of information to the client at a frequency of 6 Hz, increasing the overall size of the bundle, making it better for compression. Transmission at this frequency adds an overall delay of 167ms to the system. This happens because after a value is sampled it's kept in the memory of the device for some time before the next message is scheduled to dispatch. This latency increase was considered necessary since it helped reduce bandwidth usage and improved performance as the client has to process less incoming messages.

For the compression solution, only lossless algorithms that were purpose-built for real-time usage were considered. The candidates found that met these requirements and the were compared further were: LZ4 [45], LZO [46] and zlib [47].

In order to compare them, both the final size of the bundles and compression time were taken into account. The test was done by calculating the average compression time, average package size and total bandwidth used during a 30 second data transfer test at a rate of 6 Hz. The data being compressed is formatted according to the standard described in the previous subsection that contains only the first and last timestamps for the bundled values. The values transmitted are a simulation of what the device would output, having 8 different generated channels at 1000 Hz, of both a sine wave signal and random values[2]. The results of these tests can be seen in Table 5.2.

[2]Due to the fact this dissertation was done during a period of global pandemic access to the device and therefore a reliable EMG signal was unfeasible.

Table 5.2: Comparison of compression libraries during a 30 second test of signal transfer.

	Compression Time	Sine Wave		Random Noise	
		Avg. Size	Total Bandwidth	Avg. Size	Total Bandwidth
Control	0 ms	4645.93 B	27720.71 Bps	4638.96 B	27679.12 Bps
LZ4	0.16612 ms	1444.78 B	8620.52 Bps	3046.75 B	18178.94 Bps
LZO	0.20170 ms	1519.16 B	9064.32 Bps	2164.21 B	12913.11 Bps
zlib	0.90467 ms	788.17 B	4702.74 Bps	1247.32 B	7442.34 Bps

Taking into account the results, the chosen algorithm used for data compression was zlib since it achieved a significantly better-compressed size, of about half, when compared to the other libraries. It obtained a compression ratio of 5.89 for the sine wave signal and 3.72 for the random signal. This better compression ratio came at the cost of performance, measuring over 4 times slower than the two other libraries, however, the average time was still bellow 1 ms and was considered insignificant in the overall functioning of the system.

5.4.1.3 Client Side

With the data sent being compressed and formatted as a binary buffer when it reaches the client device, it has to be decoded to be acted upon. This decoding happens in real-time in the browser while the charts are being updated with past received values. Since JS is a single-threaded language when a package was received and unbundled, the frame being rendered was delayed causing the experience to freeze for a noticeable fraction of a second. In order to remove this unwanted irregular frame every time a new message was received (6 times every second) all the decompressing and parsing logic was moved to a web worker.

The Web Worker API [48] is a part of the HTML standard, finalized in September 2015, that allows a website to spawn background workers that run scripts in parallel to the main page. These workers can run any type of code, including data storage mechanisms, and communicate with the main thread via a system of messages. This way, the implementation takes advantage of the multiple CPU cores and threads that are present in modern hardware of all classes, from smartphones to desktops.

The implementation of web workers for this project was done via the library threads.js [49] that integrates with Webpack to properly compile and bundle these workers for use with the rest of the project. When initiated `WebSocketManager` spawns a `readData` worker to which it forwards all the incoming message bundles. On receiving this binary data, the worker decompresses it using pako [50], a web implementation of the zlib compression algorithm, parses the uncompressed bundle and reconstructs the data by approximating all the unsent timestamps according to the standard described above.

After the data is reconstructed in the worker it's sent back to the main thread where it's kept in a `pointBuffer` before being added to any graph. Values are removed from this buffer and

used at the rate of the rendering of the client. If the device is rendering at 60 frames per second (FPS) this means each frame lasts about 16.66 ms of time on screen, and so, for that frame the top values in the buffer, that when combined, span the sampling time of around same amount are removed and added to the graphs. This ensures that even though values are sent as a big chunk of information they are displayed to the user as if they were being sampled in real-time, making the overall experience significantly smoother.

5.4.2 Graph

The final solution implements all the features expected from a charting library, allowing for a robust analysis of the data while maintaining high performance. A screenshot of this final implementation and its interface in use can be seen in Figure 5.3.

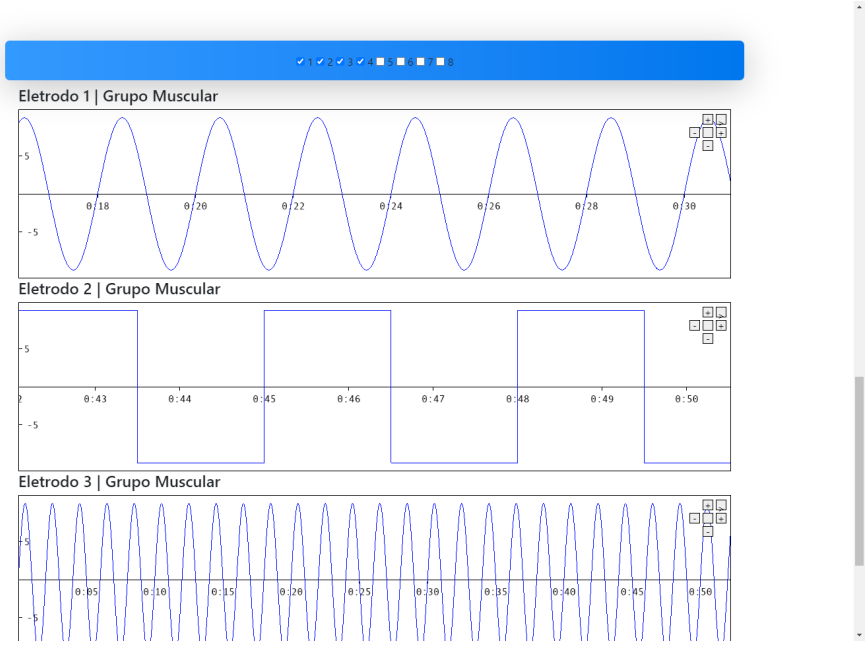

Figure 5.3: Screenshot of the current implemented interface

A graph in the system is modelled by the `Graph` class that holds all its logic, state and data in a way that multiple graph instances can work independently of one another. On the HTML Document Object Model (DOM), the website structure, the graph is represented by a `div` element with a `graph` class associated to it. When the `GraphManager` is initiated it parses the DOM for all elements that fit the above criteria and creates an instance of `Graph` for each one of them.

When a new `Graph` object is created the proportions of the associated HTML element are used to initiate a `Camera` with the appropriate aspect ratio to project onto the div. This camera is then placed in an empty `Scene`, with its capture planes parallel to the XY plane. When the camera is first positioned, it's ensured that the leftmost point being captured corresponds to the origin point of the scene, where values will be placed. After this, a `VisibleRange` is calculated, that corresponds to the highest and lowest point in both the X and Y directions the camera can

capture at a certain time. A top-down representation of the scene's initial state can be seen in Figure 5.4.

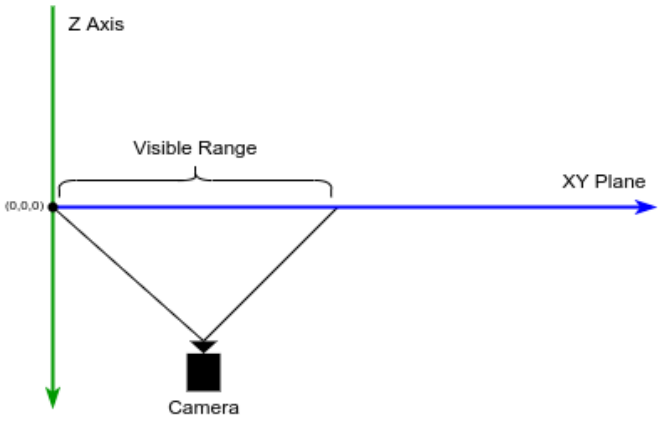

Figure 5.4: Diagram of the initial state of a Three.js `Scene` corresponding to a graph.

When points are added to the graph, they are added as lines in the XY plane, with its timestamp value corresponding to the X coordinate and the EMG value corresponding to the Y coordinate. Since the time is counted from the start of a consultation, no negative times exist so no values are plotted with negative X coordinates. When more information is added, it arrives sequentially so it's always placed to the right of existing data, where line segments are appended from the previously plotted values to the new ones.

These values, however, are not placed in their corresponding coordinates in the scene. A point with the timestamp of 1 second and the value of 10 is not plotted at the coordinates (1, 10, 0). This happens due to the way the graph handles zoom, allowing for independent zoom values for each of the two dimensions.

When points are added to the graph they are in fact added to a `LineSegments` object that is contained within a `Group` object called `plotLine`. This happens in order to apply transformations to the data set as a whole. Since all points are contained in this group, transformations applied to the object will affect the entirety of the data. This way if the user wants to increase the zoom in the X direction, a simple scale transformation is applied to that axis in the `plotLine` object, which in turns affects all the plotted points, making it extremely efficient when compared to applying said transformation to all points individually. Other scene parameters such as the scale of the axis and the camera's `VisibleRange` also need to be updated to reflect the change in zoom to match the performed transformation. Using this approach allows for the manipulation of a data set of thousands of information points without requiring too much computation power.

While data is being appended to the graph over time, the line being plotted might move outside of the camera's `VisibleRange`, forcing the graph to alter said range in order to fully show the new values. If the point added is outside of the camera's Y range, a simple zoom change is done in this axis to show the new value. Because instantly updating the zoom value would cause the graph

line to *snap* to this new position, the zoom operation is applied over several frames providing an ease animation as to make the experience smoother for the user.

If the incoming value is plotted outside of the X range of the camera, the camera is then moved in the horizontal axis to show this new point. This movement also occurs when a user interacts with the graph by dragging with the mouse, or using touch input, to pan the information left and right. The user drag velocity is inversely applied to the camera over time while the visible range is updated with the new displaying values. This camera movement is applied every frame to provide a smooth dragging experience allowing for better handling of the graph.

It's important to note that despite being built using the Three.js framework and using its scene paradigm, requiring the update of camera positions and applying 3D transformations to the data projected, all of this complexity is abstracted from the end-user. All a user sees when the camera is moved, are the charted values *moving* along with the screen. The interface and usability are indistinguishable from any other graphing solution.

5.4.3 Axis

Each graph has two axis, a horizontal one, that corresponds to the X coordinate on the Three.js scene, where time is plotted and a vertical one, corresponding to the Y coordinate, where the sampled values are plotted. Each axis has several markings along its length to convey the current scale. A screenshot of an empty graph, with the axis drawn, can be seen in Figure 5.5.

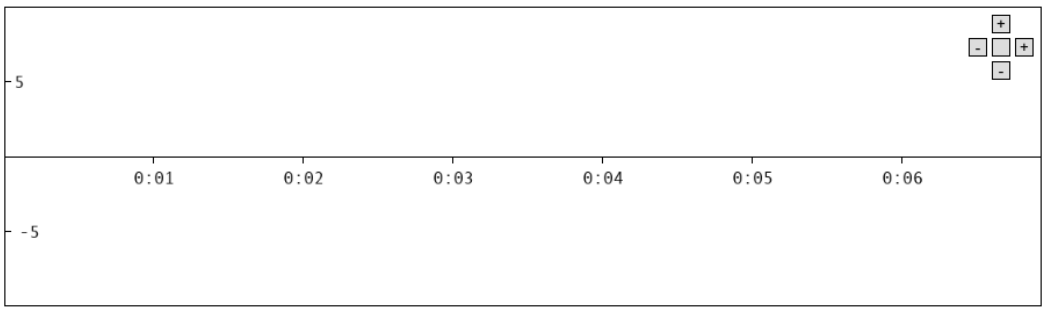

Figure 5.5: Screenshot of an empty graph.

Before exploring how the axis work individually we first need to describe what constitutes an `AxisStep`, the class that models each individual marking on an axis. Every step consists of a small line with a label associated with it that denotes its corresponding value. In the horizontal axis, this label is expressed in a time format while in the vertical axis it simply displays the actual value.

The main challenge when creating these steps was displaying this label. When text needs to be displayed to the screen the Three.js framework doesn't offer a convenient solution to use within a scene. The recommended approach is to create the text via HTML and CSS and have it overlay on top of the canvas element where the scene is being rendered. However, this solution was found to be too inconvenient and complex for this project, since the text would have to move synchronously

with the camera, making it necessary to update CSS positions in real-time according to movement happening within the scene. Since all the necessary logic to display the text and information about its position was already stored in the scene object, the `Texture` and `Sprite` components were used to draw the numbers instead.

When a graph is first created, a `Texture` object is also instantiated using an image of all the needed numbers and symbols to properly represent all possible text variations (Figure 5.6). The font chosen for this purpose was monospaced to help with the placement of the digits by making them all the same width. This texture is then broken into multiple `Sprites` one for each digit, that are placed in a `Map<string, Sprite>` where the key to each sprite is the character represented in its texture image. With all this setup done, when the label portion of the step is being constructed, the associated text is cast to a string and for each character in that string a clone of the corresponding `Sprite` is created and added to the step.

Figure 5.6: Texture image used to make the sprite digits to display the label text on an `AxisStep`.

The creation and placement of these steps is done by the `Axis` class and are handled differently depending on the axis. The X-axis is composed of a horizontal line with n equally spaced steps along its length. The line that constitutes the axis itself has the same length as the existing `VisibleRange` and moves synchronously with the camera creating the illusion of being infinite without having to update its size.

The steps are placed in their corresponding place in the scene according to the current zoom value for the graph. However, these steps do not stay permanently in the scene. To save memory, when a step is out of view of the camera it is removed from the scene, only being reconstructed again if necessary. This allows the graph to keep running indefinitely without having any memory issues.

The distance between steps is calculated based on the current zoom value of the graph. If the graph is zoomed out the markings update to a bigger interval to better convey the scale. This is accomplished by defining minimum interval width in pixels, and when a zoom action occurs, calculating what unit value, from a set of predefined values, would make the interval closest to the minimum width whilst surpassing it. The predetermined unit values, in seconds, for the X-axis are the following: [1, 2, 5, 10, 30, 60, 300, 600, 1800, 3600].

For the Y-axis, no vertical line is drawn since the graph can't be panned to negative time values and a line coinciding with the origin would never be seen. The vertical steps are placed at the leftmost position in the visible range and are moved with the camera so they always stay in view. The algorithm used to calculate the distance between steps is similar to the one previously described except with different minimum widths and unit values. The unit values for the Y-axis u are calculated according to the following formula $u = 10^n \times v, n \in \mathbb{N}, v \in [1, 2.5, 5]$.

5.4.4 Data Management

The continuous plotline represented by the graph that links all the information points collected is not an uninterrupted object. This line is made of segments of information that when together transition seamlessly and give the appearance of continuity. This stems from the fact that all data stored in a graph is divided and represented by `Chunks`, uniformly sized blocks of information that make displaying and managing a large number of data points easier.

5.4.4.1 Data Chunks

A `Chunk` is the class in the system that models these blocks of information. The core of the `Chunk` is a `LineSegments` object where the information is added and when drawn to the screen represents the multiple line segments connecting all the information points that make up the incoming data.

Rendering objects in Three.js is an expensive operation, and rendering thousands of lines at a high frame rate would cause performance issues on most systems. This was solutioned by using the `LineSegments` component, that draws lines between consecutive points represented in its given geometry.

Two distinct types of geometry can be used by objects in the framework, a standard `Geometry` and `BufferGeometry`. The difference between the two is that the latter has a fixed size and cannot be resized after its creation. This distinction exists because resizing buffers is a very costly operation, almost as costly as creating a new buffer, so there is a benefit to having a fixed size geometry that while more strict offers a significant performance increase.

This is one of the reasons data is divided into blocks. When initializing a new chunk, a `BufferGeometry` object is created with enough space to accommodate a pre-defined number of points. When data is added to the chunk, specific positions in this buffer geometry are updated with the added values, causing the object to update when rendered. This means all `Chunks` have the same size since their available space is allocated when they are created.

Since specific positions in a typed array are being accessed and updated, the `Chunk` class has helper methods that abstract all these low-level operations which make it simple to add points to the `LineSegments` object, check its capacity or return its first and last value.

5.4.4.2 Data Structures

With a structure to accommodate sections of data, the graph has to have a way to manage when chunks are created, displayed and destroyed. This is the job of the `ChunkManager` the class that handles how data is stored and displayed for a graph.

As information is added to the chart, the `ChunkManager` appends it to the last created chunk with available space. A reference to this chunk is always maintained by the manager and is called `updatingChunk`. When the current `updatingChunk` is full, it's stored in local memory and a new empty chunk is created and referenced as the new updating chunk. Every chunk has an

assigned ID number. These numbers ascend in order and are used to know the correct sequence of chunks that form the graph plotline.

Just like the steps in the X-axis, chunks are not permanently kept in the scene after they are created. They are only added to the scene when a section of the information in the chunk is contained within the camera's visible range. When a chunk is no longer visible, it is removed from the scene and kept either in memory or local browser storage.

The `ChunkManager` has three arrays to handle loaded chunks. `VisibleChunks`, where all the chunks containing data that is currently visible are stored, `LeftLoadedChunks`, where the first 3 unseen chunks to the left of the current window are held and `RightLoadedChunks`, that follows the same logic for the right of the window. When a chunk becomes out of view it's removed from the visible buffer and added to the appropriate loaded chunks buffer, according to the direction it exited the screen. That buffer, if it reaches its capacity of three chunks, then removes the furthest visible chunk from memory to keep its size constant.

The opposite also occurs, when a chunk that was in a side loaded buffer gets pushed to the visible array, if available, a chunk stored in browser memory is loaded and added to the now smaller buffer to keep the consistent size. These arrays are always kept in order according to the correct sequence of chunks, so they can be easily manipulated with simple and performant array operations such as `pop` and `push`.

An example of how the visible range would move after a pan and how that would affect the chunk structure described can be seen in Figure 5.7. Changing from frame (a) to (b) the last portion of the leftmost visible chunk stops being visible to the camera, and as a result, the chunk is shifted out of the visible buffer and pushed to the left buffer which at the same time shifts its left most chunk. The transition from (b) to (c) requires a new chunk from the right to be loaded, so it is pushed into the visible array. Since there are no more chunks to load past the updating chunk, the `RightLoadedChunks` buffer maintains its size.

This behaviour occurs to guarantee that when information needs to be loaded, either from a zoom out or a pan, the chunk loading happens fast and responsively. Because the chunks that need to be loaded are always the ones that are just outside of the visible frame, it's easier to keep them pre-loaded in a buffer and add them to the scene when necessary.

Only in exceptional cases, where the user does an exceedingly fast pan or has zoomed out to a degree where a large number of chunks need to be loaded at once does the graph not respond instantly. This is because if more than the 3 pre-loaded chunks need to be pushed to the visible buffer in a single frame, they have to be fetched from local memory.

5.4.4.3 Local Storage

One of the main requirements of the system is data persistence. All information in the graph should be available to be analysed during a consultation. The medical professional should be able to zoom the graph out to the first received values and have a complete history of the signal.

This is achieved by storing the information chunks in local storage and then recalling them only when needed, allowing for much more optimized usage of the device's limited RAM.

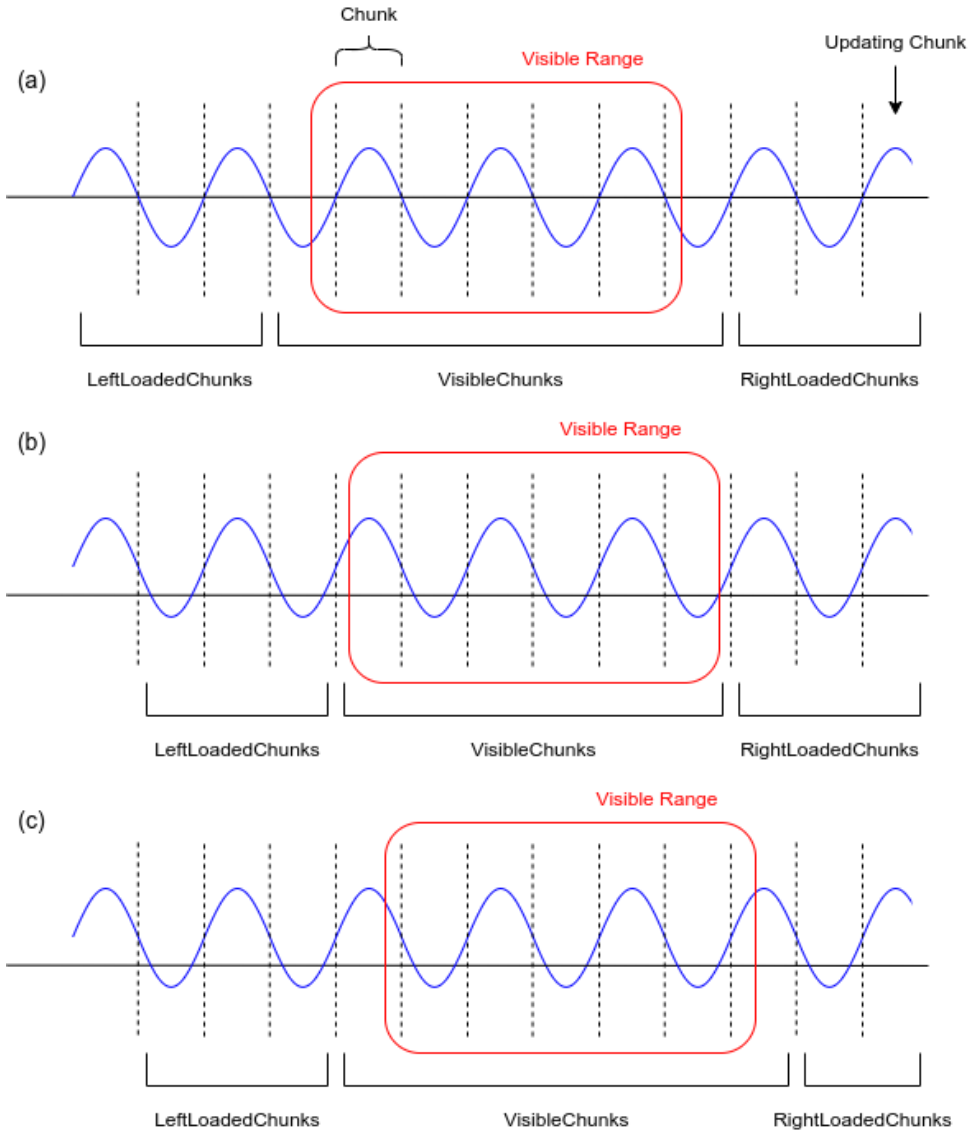

Figure 5.7: Example diagram of the changes in the graph data structures after 3 consecutive frames where the visible range is shifting.

Modern web browsers support multiple ways for web sites to store data on the device and retrieve it when necessary. Two examples of local storage APIs, and the ones used to achieve the wanted functionality for the application, are the IndexedDB API [51] and the session storage API [52]. They operate quite differently and have distinct capabilities. The session storage API provides a mechanism to store key/value pairs in the device during the duration of a session. The information, formatted as a string, is stored in a simple JS object that can easily be accessed during run time with a maximum capacity of 5MB of storage.

IndexedDB is a more low-level browser API that allows for the storage of a complex amount

of structured data of any type, including binary arrays. The data limit is much higher than the session storage API and depends on browser implementation. The information stored is indexed to enable high-performant searches of the data set. While more robust, this solution is, however, much more difficult to use, so the library localForage [53] was employed to abstract some of the more complex functionality.

The two described solutions are used in juncture to allow for the functioning of the history system for the graph data. When a chunk reaches its maximum capacity its geometry data is sent to a web worker to be stored. The web worker API is again used for heavy operations to be applied to the data without halting the main UI thread.

Firstly the X coordinate for last value of the chunk is stored with a reference to the chunk ID in the session storage instance. This is integral for the rebuilding of the chunk that will be explained further on. Following this, the geometry is formatted into a binary array, removing all unnecessary information. This includes the Z coordinate for all points that is always equivalent to 0 and all the duplicate points created to properly draw the line segments.

After this pre-processing occurs the buffer is further compressed by pako, the JS implementation of the zlib compression algorithm, in order to be as memory efficient as possible. Finally, this formatted and compressed binary data structure is indexed in the browser database with a key that pertains to the chunk and graph ID so it's easily recalled.

Since the IndexedDB can only be accessed asynchronously when a chunk needs to be recalled that process does not happen instantly. In a situation where the user zooms out to a great extent and multiple chunks need to be loaded instantly the `ChunkManager` needs to know the width of the needed chunks to only load the required amount of information to fill the visible space. This is where the session storage last value that was saved is used since it can be accessed in synchronously. Using the reconstructed chunk's last value and the last value of the previous chunk an empty `LineSegments` object with the correct width can be built and placed in the scene before any of its information is fetched and processed.

After this placeholder is created, a web worker is called to reverse the above-described process. The data is decompressed and subsequently formatted as a `BufferGeometry` before being transferred back to the main thread where it's set as the geometry for the empty spacer chunk. In most instances, this whole process happens while the empty chunk is still in one of the two loaded chunk buffers and is seamless for the user. In the case the empty chunk was already placed in the scene, in the next frame its information updates and the line segment is rendered.

5.4.5 Rendering

This solution is built around the Three.js library in order to benefit from the better performance that comes from using the WebGL browser rendering API. This API, however, has some drawbacks that had to be addressed. When rendering with WebGL, the browser has to fetch, from the canvas HTML element where it's going to render to, a `WebGlRenderingContext` that provides an interface to the underlying OpenGL graphics platform.

To render multiple charts on multiple canvases a new context would have to be created for each one. Apart from being memory-intensive, these contexts would also not share the same resources. All the low-level buffers, shaders and loaded textures needed for the rendering of a graph would have to be loaded for every different context.

This was solutioned by having a single context and canvas where all graphs would be rendered to in different positions, giving the illusion of multiple standalone graphs existing. To achieve this the canvas is the first element in the body of the website, having a height and width that matches the currently open window. This element is moved when the user scrolls the page and is resized when the window dimensions change, to always fully occupy the visible area of the page.

Subsequently, when a graph is about to be rendered, the position of its corresponding div element in relation to the browser window is measured. Using this positioning information the viewport of the renderer is set to only output to this defined area of the canvas. Since the canvas matches the area of the screen and the div element is transparent, it appears as if the information is being rendered inside the div when in reality it's being rendered to the canvas.

Using this method any number of graphs can be rendered without the use of extra resources. It also helps with performance since if a graph intended to be rendered and is currently outside of the pages visible frame, its rendering is skipped until a portion of it is visible.

In Figure 5.8 the fully rendered page can be compared to an example where only the underlying canvas is visible. In this instance, it's easy to understand how the canvas is being sectioned in order for the information rendered to match the corresponding elements.

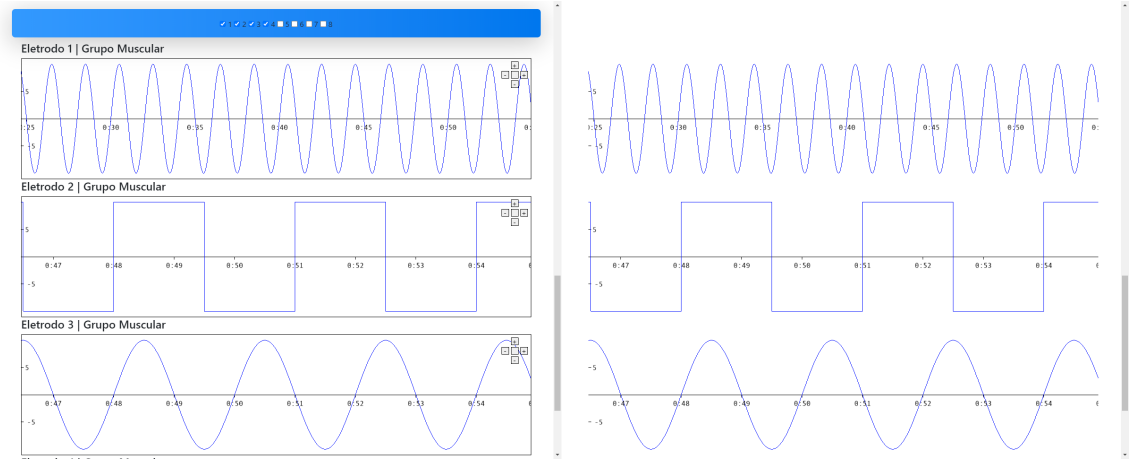

Figure 5.8: Comparison between fully rendered page (left) and only the canvas element (right).

5.5 Summary

This chapter explores in detail the implementation effort made for this project. It discusses the choice of using tools such as TypeScript and Webpack to facilitate the development process. It also details the architecture, implementation and key concepts used during development, not only for the achieved solution but also for its underlying technology Three.js.

The transfer layer is built using the WebSocket API through which binary bundles of formatted and compressed data are sent at a rate of 6 Hz. These bundles are decompressed and pre-processed by a web worker on a separate thread on the client-side as to not interrupt the main thread where the rendering is done.

The implemented solution has all the functionality expected from a charting solution, implemented in a way that is easy to use, fast and responsive. Features like data panning, zoom and the adaptive axis scales are all done taking advantage of several different Three.js components to maintain good performance.

All the incoming graph data is persistent and can be recalled at any time during a consultation. This is achieved by splitting data in multiple chunks of information that are stored locally via the IndexedDB API. By using a combination of methods while managing the information, like pre-loading data, using simple array operations and employing web worker threads, the overall experience of examining the graph is seamless and performant.

Chapter 6

Results

A big focus of the development throughout the project is in optimizing small details in the implementation of both the display solution and the transfer layer to achieve the best possible performance. The efficiency of managing device resources of the system was also an active priority during the development phase.

In this chapter, a set of measurements is presented to examine the performance of the developed solution and provide an assessment of the improvements the present work made in the overall usability of the prototype device.

Firstly, in section 6.1 a comparison between the new displaying approach and the previously implemented solution is conducted. This further demonstrates the improvements made to the website and the overall user experience with a greater emphasis on device performance.

Subsequently, in section 6.2 an analysis is done of some popular charting libraries for the web. These were tested in scenarios identical to the functioning of the device and then compared to the developed solution. This allows for the comparison between the developed solution to the state of the art approaches available on the internet.

6.1 Comparison with previous solution

The newly implemented approach differs from what was previously implemented in several ways. In order to understand how the changes made affect the overall functioning of the system, several criteria were taken into account starting with the efficiency of the transfer layer.

6.1.1 Transfer Layer

Multiple improvements were done in this section but the most important was the increase in the amount of data transmitted. With the previous implementation, fewer values were sent more frequently. This was enough to build a chart that would approximate what the device was sampling but a large amount of information was lost in the process. With the new approach, all values sampled are sent to the device after they are pre-processed and compressed so no information is lost.

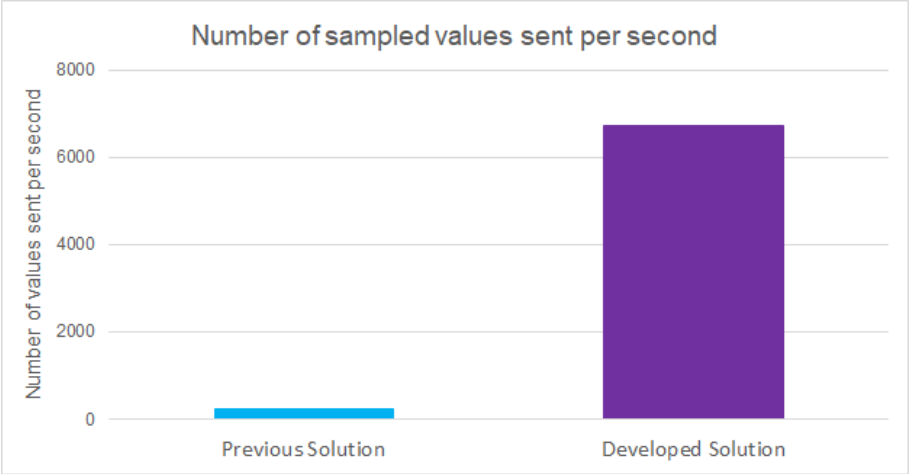

Figure 6.1: Number of sampled values sent per second.

Figure 6.1 shows a comparison between the amount of usable data being sent to the client every second. This number is calculated by a simple formula using the number of values sent per message and the frequency of transmission. For the previous solution, 8 values are sent every message, at a rate of 30 Hz, resulting in $8 \times 30 = 240$ values per second. For the developed solution, a buffer with an average size of 140 bundles of 8 values is sent at a rate of 6 Hz, resulting in $140 \times 8 \times 6 = 6720$ values sent per second. This corresponds to an increase of 2800% in information transmitted after the enhancements were done.

With the use of the network protocol analyser tool Wireshark [54], several tests were made with both approaches to compare them in terms of overall network traffic. Although much more information is being transmitted with the new solution, a substantial effort was made to minimize the impact of this increase in the overall network consumption of the system.

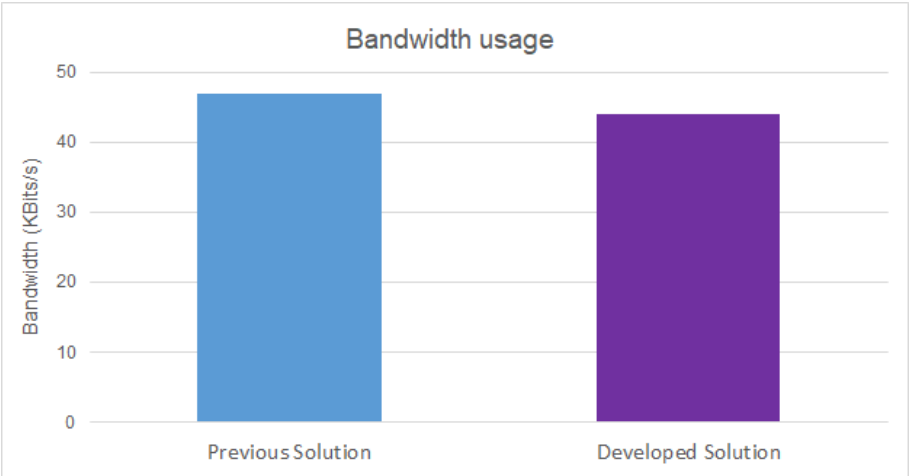

Figure 6.2: Comparison of bandwidth usage for both solutions.

Figure 6.2 compares the bandwidth usage of a 30 second transmission of data for both solutions. Despite the considerable increase in data transmitted the developed solution manages to achieve a lower network consumption of 44 kbits/s, a slight improvement over the old solution of 47 kbits/s. The improvements are better perceived in Figure 6.3 that compares the bandwidth necessary to send 10000 values.

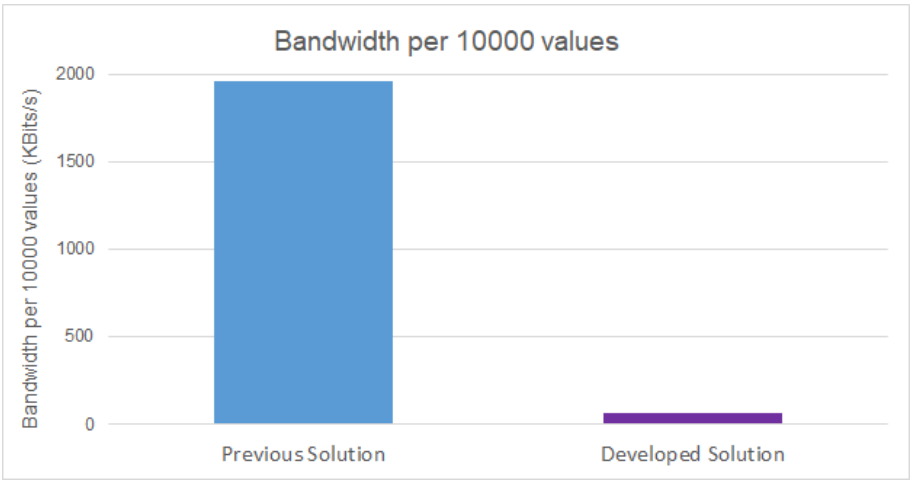

Figure 6.3: Comparison of bandwidth usage per 10000 values for both solutions.

Figure 6.4 helps to explain these findings. To transmit the higher amount of information the average packet length for the improved solution is about 4,5 times higher than its counterpart, even after the formatting and strong compression are applied, methods that were previously not used. Nevertheless, the new solution transmits packets of information at a smaller rate in order to improve compression so the overall bandwidth usage stays approximately the same.

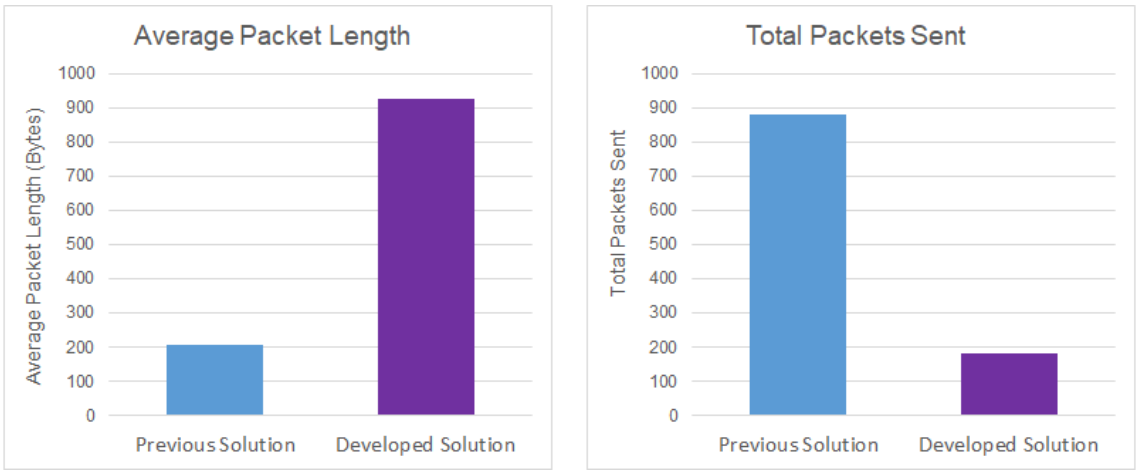

Figure 6.4: Comparison of average packet length and total packets sent for both solutions.

6.1.2 Performance

To evaluate and compare the performance of the two approaches they were tested and benchmarked in several parameters. These tests will provide a better understanding of how the implemented changes impact the usability and efficiency of the system.

The tests conducted consisted of a 30 second transmission between 2 computers: one emulating the prototype device by generating values and transmitting them over a LAN connection and the other receiving the data and plotting it to the screen. The computer running the benchmarks is using an Intel(R) Core(TM) i7-6700HQ CPU at a rate of 2.60 GHz and 8 GB of RAM. Although the device is equipped with a dedicated graphics card, both browsers tested used the processors' integrated Intel HD Graphics 530 GPU to render the web page. During the benchmarking, only the minimal software required for the computer's functioning was open to provide consistency between readings. The tests were conducted in both Google's Chrome browser and Mozilla's Firefox browser, the two most popular products for Windows and Linux.

The overall user experience of the website is hard to compare objectively, but a good indicator is the time that frames take to render along with the FPS. These benchmarks can reflect how efficient and smooth the overall experience is, with a high average FPS and low frame time variation generally reflecting a more fluid and responsive solution. Table 6.1 contains information about these measurements for each solution.

Table 6.1: Frame timing information for both solutions.

	Avg. Render Time	Slowest Frame	Fastest Frame	Avg. FPS
Previous Solution Firefox	28.35 ms	85.36 ms	4.85 ms	35.31
Previous Solution Chrome	31.74 ms	89.85 ms	13.40 ms	31.54
Developed Solution Firefox	16.67 ms	25.82 ms	11.15 ms	60.00
Developed Solution Chrome	16.67 ms	24.51 ms	8.66 ms	60.03

As evidenced, the improved solution has a significantly better average frame render time and FPS than the previously implemented approach despite handling and displaying over 28 times more information. In both browsers the developed solution runs at an average of 60 FPS, this is because the browser attempts to match the refresh rate of the device, in this case, 60 Hz. Any improvement above this threshold would not be noticeable since the device would not be able to display it faster.

Average FPS, however, does not strictly equate to good performance. If the average render time for the system is low, but periodically some frames take longer to process than consistency is lost and the user can encounter noticeable stutters in the motion that can undermine the overall experience. A good indicator for this type of problem can be found by analysing the overall number of frames, as it's done in Figure 6.5.

This graph indicates that the experience for the developed solution is consistent overall, with an overwhelming majority of the frames rendered, 89% on Chrome and 76% on Firefox, being below the needed threshold to maintain the desired frame rate of 60 FPS and never surpassing

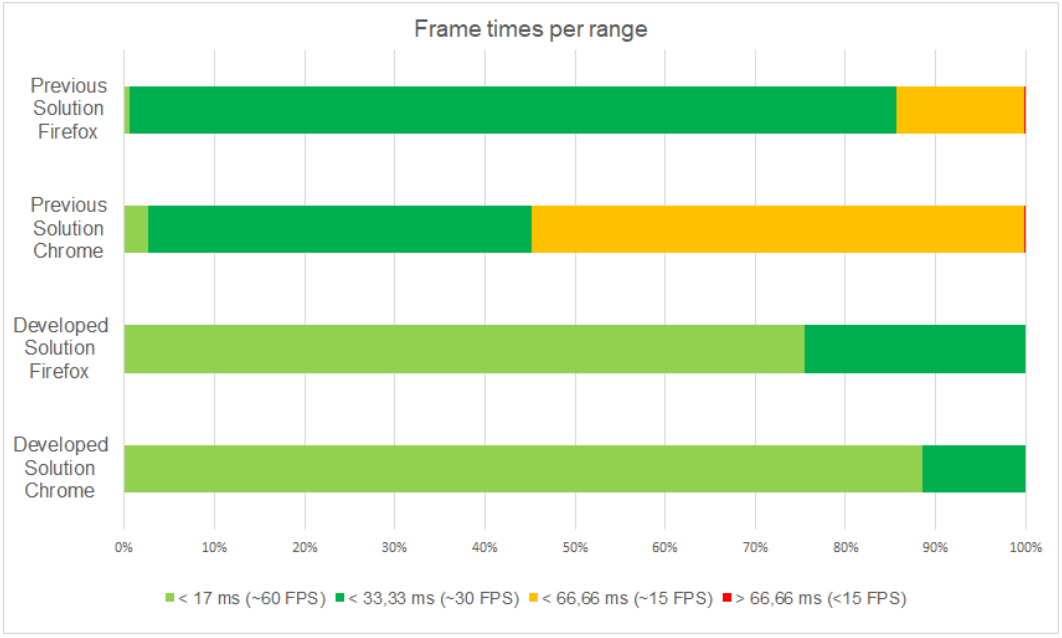

Figure 6.5: Comparison of the frame times by range for both solutions on Chrome and Firefox.

the range of 33,33 ms. This represents a significant improvement when compared to the previous solution, where only a few frames would reach the desired threshold and occasionally some would require up to 90 ms to render as evidenced in Table 6.1.

Other important information to benchmark is how efficient the new solution is. One of the non-functional requirements of the system was for it to be power and resource-efficient, allowing it to run easily and without any performance issues on lower-end devices.

Figure 6.6 depicts the improvements made concerning CPU usage for the new solution. As illustrated, the proposed work is unequivocal more performant than the previous solution, more notably on Google Chrome. This is greatly influenced by the fact that the GPU is tasked with the rendering part of the implementation through the use of the WebGL API. Even though the data set being displayed is significantly larger, it requires less processing power due to this implementation and therefore relaxes the total usage of the CPU. It should be noted that modern CPU's have dynamic clock speeds that increase, if possible after certain usage thresholds are crossed. However, this is not taken into account in the following comparisons.

Consequently, the developed solution is also less taxing in regards to power consumption, making it more convenient to use in all types of devices, especially ones that are battery powered like laptops or smartphones. A comparison of the power consumption of the System-on-Chip (SoC), which consists of the CPU cores, caches, GPU and memory controller for the conducted tests can be seen in Figure 6.7.

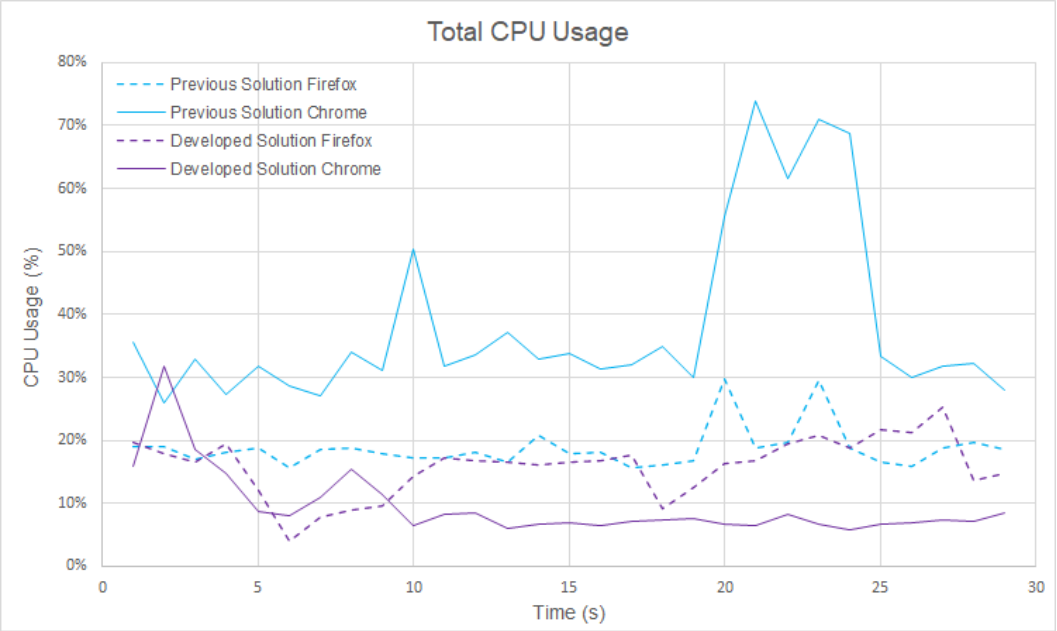

Figure 6.6: Comparison of CPU usage for both solutions on Chrome and Firefox.

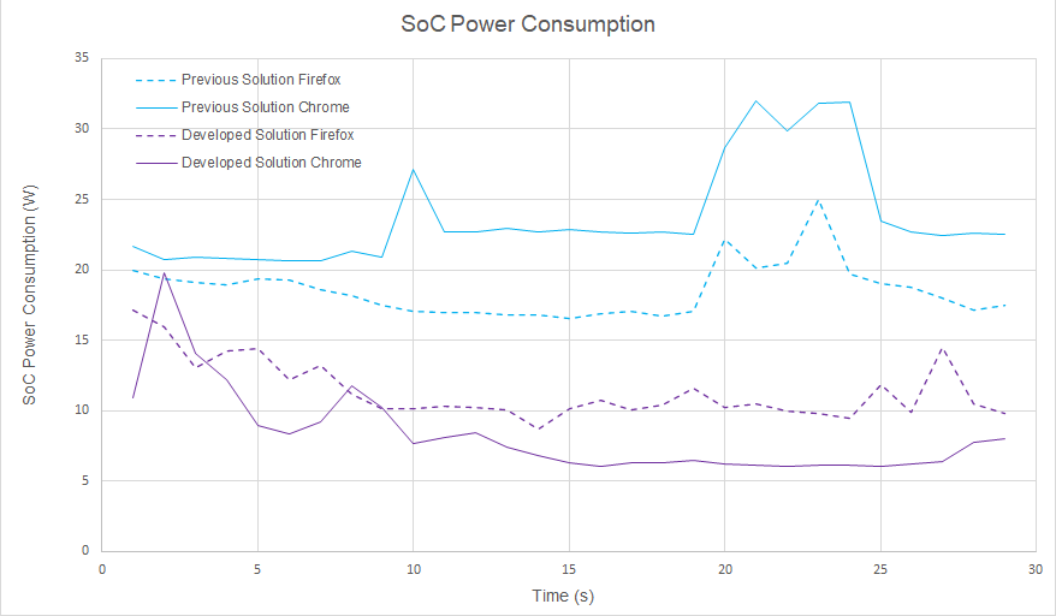

Figure 6.7: Comparison of SoC power consumption for both solutions on Chrome and Firefox.

In regards to GPU usage, Figure 6.8 demonstrates as expected an increased use for the new solution, since it takes advantage of this component to accelerate the rendering of the graph. This increase is more pronounced in Firefox since the WebGL implementation is apparently not as efficient as the one present in the Google browser.

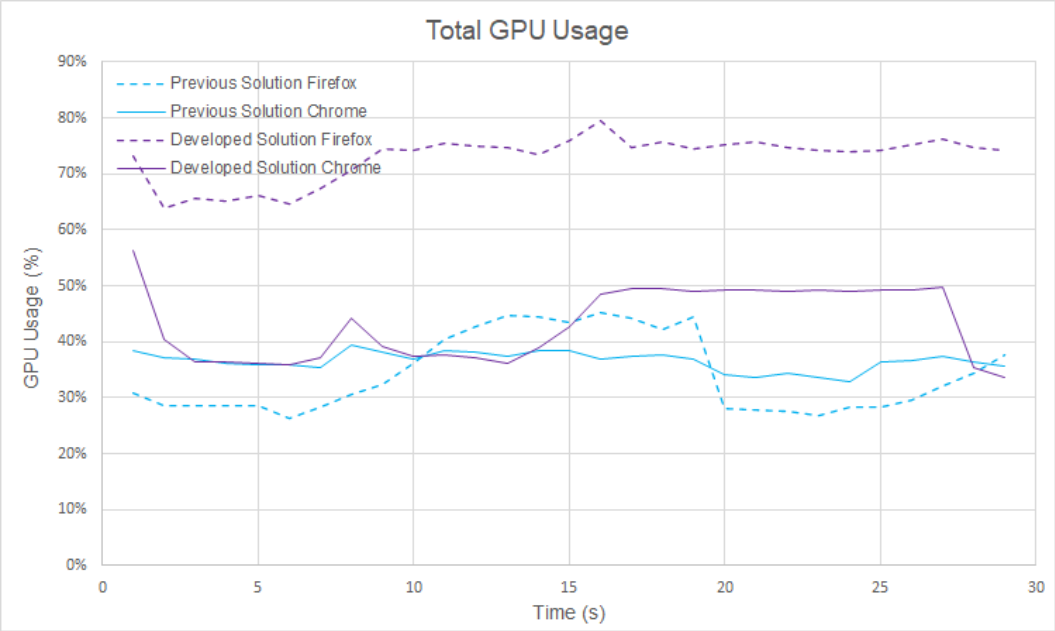

Figure 6.8: Comparison of GPU usage for both solutions on Chrome and Firefox.

6.2 Comparison with other charting solutions

A large portion of the work done was spent on implementing a charting solution that is capable of handling the specifics of the collected information. This solution has to be able to display thousands of data points precisely, have multiple graphs charting different information simultaneously and be easily manipulated by the user, all whilst being updated in real-time.

As with most software problems, there were already pre-built solutions ready to be used when this project started, as seen previously. While none of these ready-made solutions were considered ideal for the requirements of this project they are still an indication of the state of the art of what is expected from charting solutions for the web.

This section compares a few of these available implementations to the developed solution in order to assess if it is the best solution for the problem, and what are its strengths and weaknesses when compared to the prior art.

Many charting libraries were considered for this comparison test. The four chosen solutions and the reasoning behind the choice are the following:

- **Rickshaw and D3.js** - The Data-Driven Documents library or D3 [55] is one of the most popular charting solutions for the web, with over 90k stars on its GitHub repository and 1M weekly downloads on NPM. This library, however, is not a ready-made graph implementation, instead it offers the tools to easily manipulate data and create different visualization approaches using SVG. Consequently, for this comparison the Rickshaw library was employed, a ready to use real-time graph implementation built with the tools provided by D3.js by Shutterstock [56];

- **Chart.js** - A simple, clean and open-source charting library that uses the Canvas API to render to the screen. Was chosen mostly due to its popularity as a web charting solution, it counts with over 49k stars on GitHub and over 1M weekly downloads on NPM [57];

- **CanvasJS** - A robust and flexible graphing library used by multiple professional software companies for their data visualization needs. Also relying on the Canvas API for its rendering, it claims to have 10x performance over other solutions and was already in use by the previous implementation of the prototype [34];

- **Lightning Chart** - A new charting solution that claims to be the highest-performant charting library for the web. It allows for real-time scrolling line charts of up 1 million points. It differs from all the others in this list since it relies on WebGL to render to the screen, a characteristic that shares with the developed work [58].

The different libraries are all compared according to different parameters measured during the following scenario. During 30 seconds, the charting software has to render onto two distinct graphs the full information of two simulated channels with a sample rate of 1000 Hz that are transmitted via a LAN connection. All the tests were run in the same machine as described in the previous section.

This scenario closely emulates the default use case of the prototype, so the different solutions can be compared amongst each other to see which one would be better suited for this specific problem. These tests will also demonstrate how capable these solutions are when handling large amounts of incoming information.

As done before the first metric analysed will be frame render time and average FPS, since these measurements provide a better understanding of the overall usability and responsiveness of the solution. Table 6.2 contains a detailed breakdown of each solution's results for both Chrome and Firefox while Figure 6.9 presents a graph of the frame rendering duration over the 30 seconds of the test run on Chrome.

Table 6.2: Frame timing information for all libraries on both Chrome and Firefox.

	Avg. Render Time	Slowest Frame	Fastest Frame	Avg. FPS
Rickshaw Chrome	67.16 ms	123.59 ms	45.68 ms	14.92
Rickshaw Firefox	45.53 ms	97.09 ms	27.90 ms	21.98
Chart.js Chrome	341.69 ms	389.09 ms	327.41 ms	2.96
Chart.js Firefox	382.77 ms	455.23 ms	356.75 ms	2.64
CanvasJS Chrome	68.67 ms	96.90 ms	57.28 ms	14.59
CanvasJS Firefox	59.54 ms	221.25 ms	45.45 ms	16.79
Lightning Chart Chrome	54.11 ms	116.88 ms	9.13 ms	18.50
Lightning Chart Firefox	49.23 ms	64.88 ms	39.88 ms	20.31
Developed Solution Chrome	16.67 ms	24.51 ms	8.66 ms	60.03
Developed Solution Firefox	16.67 ms	25.82 ms	11.15 ms	60.00

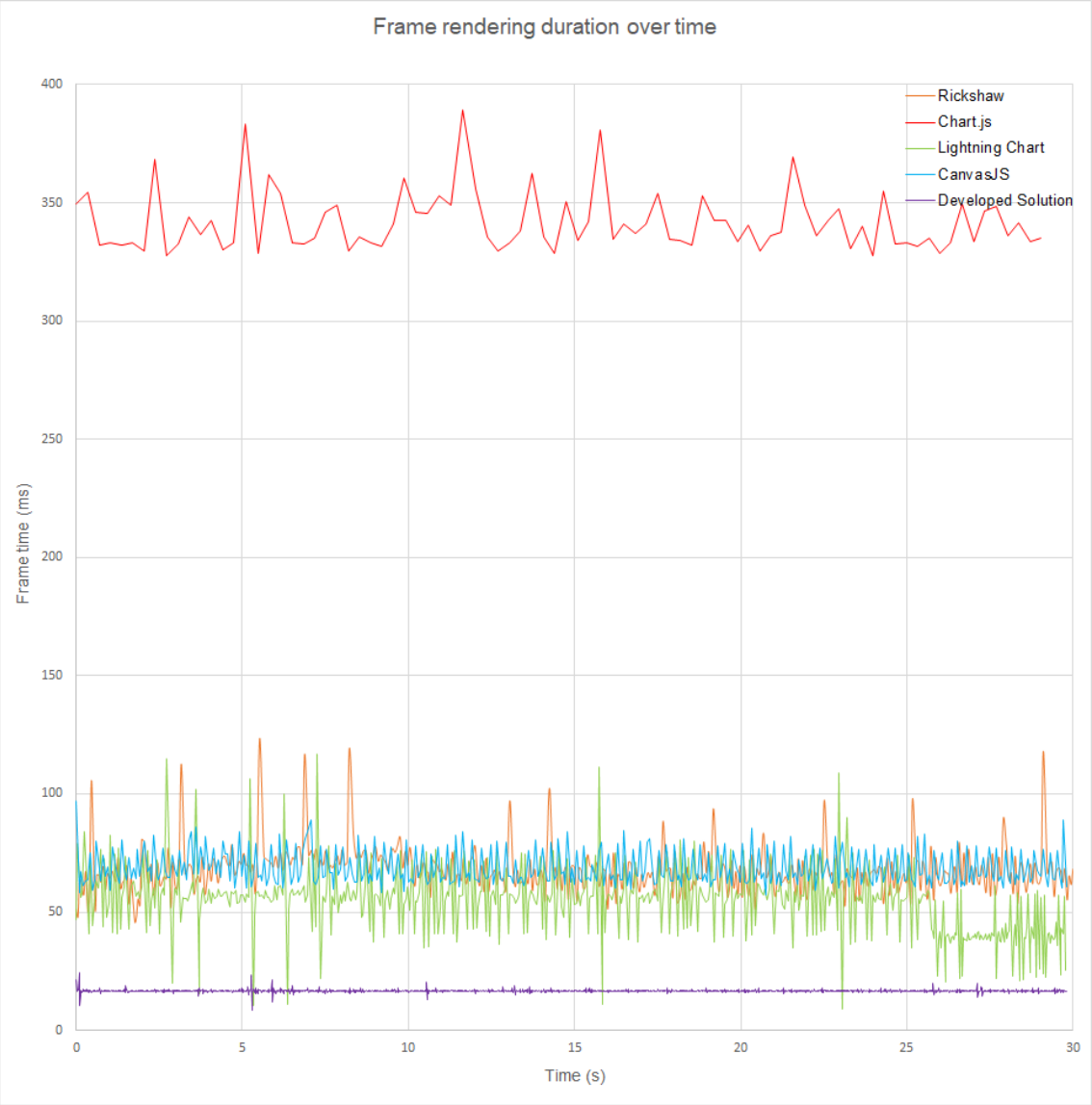

Figure 6.9: Frame rendering duration per library over time on Google Chrome.

As evidenced, the developed solution achieves a much better frame rate and, consequently, a better user experience than all the compared approaches. None of the mentioned libraries are capable of reaching an average of 30 FPS, the standard for most modern video and animation. Figure 6.10 again confirms the previous statement, showing the developed solution having a significant better distribution when compared to the other approaches.

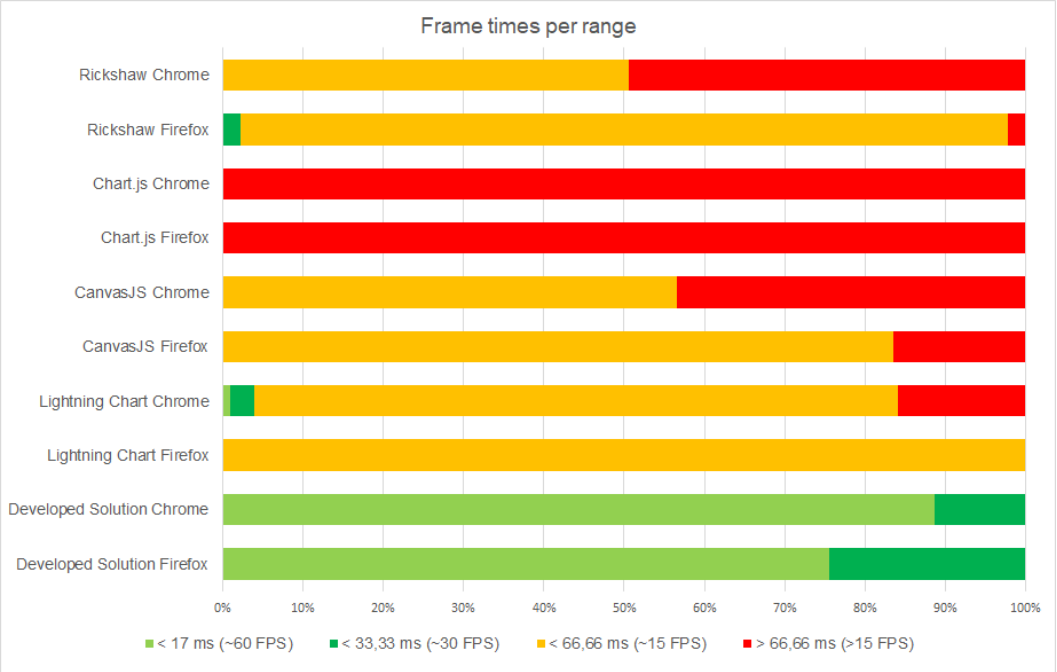

Figure 6.10: Comparison of the frame times by range for all libraries on Chrome and Firefox.

In terms of performance and resource usage, Figure 6.11 and Figure 6.12 show the CPU usage and the SoC power consumption for all the discussed libraries. In this comparison the developed solution also outperforms its competition, maintaining a much better CPU usage percentage of more than half usage when compared to all other solutions.

6.3 Summary

All of the metrics presented demonstrate the substantial improvement done by the present work to the overall experience of analysing the prototype's data. The new solution achieves better performance and lower resource usage whilst providing better analysis tools and more precise representation of the sampled data.

When compared to the what was previously implemented for the device it manages to use about the same total bandwidth to transfer 28 times more information, through the use of compression and less frequent messages. It also reaches twice as better of an average FPS improving overall performance and responsiveness of the system.

In regards to available graphing solutions for the web, an analysis was done with some of the most popular and performant solutions available. When tested for the volume of information needed for the project, the developed solution outperformed all of its competitors by a wide margin achieving about 3 times better average FPS than the second-best solution while using less total resources and power.

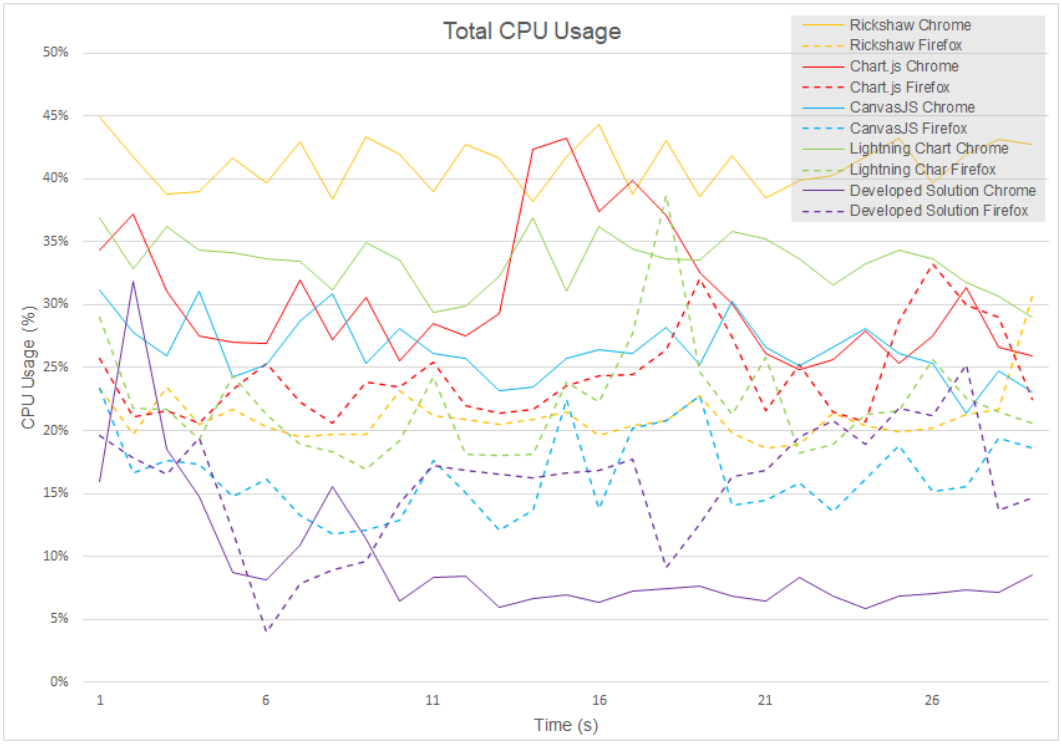

Figure 6.11: Comparison of CPU usage for all libraries on Chrome and Firefox.

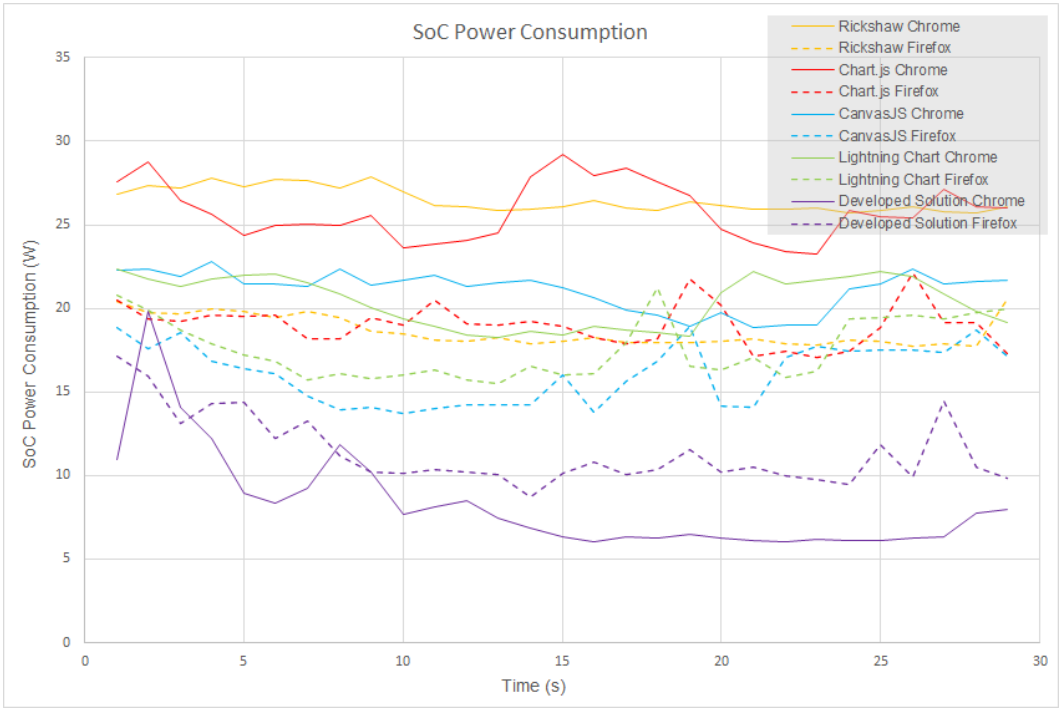

Figure 6.12: Comparison of Soc power consumption for all libraries on Chrome and Firefox.

Chapter 7

Conclusions and Future Work

The present chapter is a reflection of the work done throughout this dissertation. Firstly a summary of the project, its context, its importance and how it improved on the previously implemented solution will be done in section 7.1.

Section 7.2 details what are the main contributions done by the present work followed by the main difficulties it tackled and solved in section 7.3. Finally, at the end of the chapter in section 7.4 a description of possible future work is made.

7.1 Summary

Telemedicine can facilitate the way we provide care to those who need it, having a significant amount of benefits over traditional medicine practice in several scenarios. With technology ever-evolving and becoming more available, telemedicine devices can also evolve, aiding a larger number of people that previously had no access to such type of medical care. The severity of recent world events demonstrate the importance of access to health care and how that access should strive to be universal, not impeded by wealth, location or culture.

The majority of currently available solutions are provided by private companies, that offer expensive, platform-exclusive solutions that are tied to specific services that force the user to commit to a particular provider. Consequently, most solutions are only useful for early diagnosis or broad cases, leaving a scarcity on custom offers for certain instances where patients' needs are more particular.

This dissertation extended the work previously done on a prototype built to monitor EMG signals for use in remote consultations. The goal of this device is to be simple to use and deploy, affordable, compatible with all platforms via the web and be open and extensible for any further development.

The development work done was focused on improving the web platform where data from the device is analysed with the intent of providing a robust and performant experience, comparable to a standalone solution, but without the need to install or run any specific piece of software.

Ultimately the solution developed fulfils all of its requirements, improving greatly on what was already implemented, allowing the full amount of data collected by the device to be thoroughly analysed, with the use of a variety of features and tools, from any modern web browser. It also achieved the goal of making the platform more performant, through the use of modern web technologies, allowing for a better, faster and more responsive experience while using less of the system's resources.

7.2 Main Contributions

As stated, this dissertation achieved its main objective of creating a performant and viable solution for analysis of live sampled EMG data for the web. It addressed all the limitations outlined in 3.4 and further propelled the device and platform being developed to their intended goal.

Breaking down the presented implementation we can dissect the main aspects that made it possible. Firstly this dissertation proved that using the capabilities of WebGL for other types of rendering, where it's not traditionally employed, can greatly improve performance. The implemented solution greatly outperformed all the other tested graphing libraries, with the ability to maintain a steady frame rate while charting thousands of points at once by delegating the rendering part of the platform to the device's GPU.

The solution also takes advantage of new browser features, like Web Workers (parallel processing) and IndexedDB, that make it possible to write better and more ambitious applications. The ability to create separate threads to process and handle data without affecting the main updating thread was crucial to achieve the performance previously only found in standalone applications.

Ultimately the biggest contribution from the present work is proving that the web can be a reliable platform for applications that were previously considered too resource-intensive and required specific combinations of software and hardware to be accessed. With the implementation of new features and APIs as well as major improvements to their respective JavaScript engines, modern web browsers are now better equipped to enable feature-rich experiences that rival what was before only possible under said conditions. These better capabilities paired with the ease of access of the internet can greatly improve the reach of essential software to a much wider audience, that is currently already equipped with the necessary tools to access such a system.

7.3 Main Difficulties

Throughout development several technical difficulties were encountered, this section details the problems they presented and how those problems were solutioned. While most of these adversities were ultimately resolved or mitigated in the final version of the application they still required additional effort or research to solve during the development process.

One of the major problems encountered was how to render text inside the graph. As stated previously Three.js doesn't offer a convenient solution for creating text inside a scene. The previously implemented solution involved creating the text in a canvas element programmatically,

converting what was displayed on the canvas to a Texture, and finally applying that image to a Sprite element. The painting and conversion of the canvas every time a new step would have to render greatly affected performance and was later replaced by the solution described in 5.4.3.

Modelling the information received to the Three.js paradigm was also challenging. Several iterations were made to how the information was displayed to either improve performance or allow for the implementation of new features. Every iteration improved on the previous making the solution more performant in the process. Some of these changes include: adding all the individual Line elements to a group element to perform data set transformations easily, moving the information from individual Line elements to a single LineSegments element to reduce draw calls and using pre-allocated BufferGeometry objects to store the information as individual chunks that are more efficient to instantiate and manage.

Another problem was encountered when chunks had to be recalled from local storage after a zoom action. In order to keep the solution efficient only the required chunks of data needed to fill the visible range would need to be recalled. However, since this information was stored in the browser's indexed database recalling it didn't occur synchronously, so there was no way of knowing in run time how many data chunks were needed to fill the required space. Several solutions were implemented, from forcing chunks to have a fixed width instead of a maximum point limit, to requiring the zoom-out process to wait for all the information to be fetched from the database before affecting the interface. Eventually, these solutions were considered compromises to the overall experience, and the now implemented hybrid solution between the IndexedDB and the session storage APIs, detailed in 5.4.4.3 was developed.

7.4 Future Work

For the prototype being developed to be considered ready for release a lot of improvements still have to be made, like reworking the user interface, implementing communication methods between both parties and ensuring that all communication is secure and private. Some of these features are already being worked on by other projects and some will be done in the future.

Even after the device is considered ready and can be tested in real-world scenarios it can always be iterated upon making it smaller, more affordable or simply more efficient by using different components or re-implementing already available functionalities. The device can also be extended for use in other fields, expanding it's available capabilities like ECG monitoring.

There are also improvements that can be done that relate more directly to the software side of the project that was the focus of this work. A complete re-write or port of the software operating the device must be made in order to future proof the project. This is because the code base is written using Python 2.7, a version of the language that was discontinued in 2020 and will receive no more security or improvement updates.

On the web part of the implementation improvements also have to be made. Currently, the device only works if it's accessed through a local connection. A DNS solution has to be developed in order for the device to be accessed through the internet. This solution also has to provide a

secure connection through the modern HTTPS protocol since it's considered the standard for the web, with most web browsers blocking websites or the access to certain APIs, like Web Workers, if a connection is not secured.

Finally, regarding the developed graphing solution some extra features not covered by the present work can be implemented if they are deemed necessary. Functionality like the ability to export or import data to the medic computer for later consultation, the availability of a progressive web app companion of the website so analysis of saved data can be done offline and the creation of accessibility and customization options, to improve the analysis experience, such as plotting more than one line per graph, offering different types of graphs or providing different options to customize the fonts, colours and sizes of the elements in the graph. This concept of high performant plotting can also be extended and applied to other contexts such as the industry and financial sectors.

Bibliography

[1] Rama C. Hoetzlein. Graphics performance in rich internet applications. *IEEE Computer Graphics and Applications*, 2012.

[2] WHO and The World Bank. Tracking Universal Health Coverage: 2017 Global Monitoring Report. Technical report, 2017.

[3] Douglas A. Perednia and Ace Allen. Telemedicine Technology and Clinical Applications. *JAMA: The Journal of the American Medical Association*, 1995.

[4] Mirza Mansoor Baig and Hamid Gholamhosseini. Smart health monitoring systems: An overview of design and modeling. *Journal of Medical Systems*, 2013.

[5] David Hailey, Risto Roine, and Arto Ohinmaa. Systematic review of evidence for the benefits of telemedicine., 2002.

[6] Matthew Berman and Andrea Fenaughty. Technology and managed care: Patient benefits of telemedicine in a rural health care network. *Health Economics*, 2005.

[7] Bahram Delgoshaei, Mohammadreza Mobinizadeh, Reyhaneh Mojdekar, Elham Afzal, Jalal Arabloo, and Efat Mohamadi. Telemedicine: A systematic review of economic evaluations. *Medical Journal of the Islamic Republic of Iran*, 2017.

[8] Michael Marschollek, Matthias Gietzelt, Mareike Schulze, Martin Kohlmann, Bianying Song, and Klaus Hendrik Wolf. Wearable Sensors in Healthcare and Sensor-Enhanced Health Information Systems: All Our Tomorrows?, 2012.

[9] J. Puentes and B. Solaiman. Telemedicine in Perspective: Trends and Challenges. 2006.

[10] Md Zihad Tarafdar. *Software Development for a Secure Telemedicine System for Slow Internet Connectivity*. PhD thesis, University of Dhaka, 2019.

[11] Karen M. Zundel. Telemedicine: History, applications, and impact on librarianship, 1996.

[12] Ittipong Khemapech, Watsawee Sansrimahachai, and Manachai Toahchoodee. Telemedicine - meaning, challenges and opportunities. *Siriraj Medical Journal*, 2019.

[13] David E Bloom, Axel Boersch-supan, Patrick Mcgee, and Atsushi Seike. Population aging: facts, challenges, and responses. *Program on the Global Population Aging*, 2011.

[14] European Commission. Market study on telemedecine. *European Commission*, 2018.

[15] Eurostat. Almost 8 out of 10 internet users in the EU surfed via a mobile or smart phone in 2016.... *Newsrelease*, 2016.

[16] Push Doctor. Online Doctor & Prescription Services with a UK GP today | Push Doctor. https://www.pushdoctor.co.uk/, 2020. [Online; accessed 2020-07-02].

[17] KRY International AB. Kry – see a doctor by video. https://www.kry.se/en/, 2020. [Online; accessed 2020-07-02].

[18] Teladoc Health. The right care when you need it most | teladoc®. https://www.teladoc.com/, 2020. [Online; accessed 2020-07-02].

[19] Knok healthcare. knok. https://www.knokcare.com/, 2020. [Online; accessed 2020-07-02].

[20] Babylon Health. Babylon health uk - the online doctor and... | babylon health. https://www.babylonhealth.com/, 2020. [Online; accessed 2020-07-02].

[21] Christine E. King and Majid Sarrafzadeh. A Survey of Smartwatches in Remote Health Monitoring. *Journal of Healthcare Informatics Research*, 2018.

[22] Doxy.me. The simple, free, and secure telemedicine solution | doxy.me. https://doxy.me/, 2020. [Online; accessed 2020-07-02].

[23] VSee. Telemedicine kits, carts + digital medical devices + software. https://vsee.com/hardware/, 2020. [Online; accessed 2020-07-02].

[24] Remote Health Solutions. Virtual exam room | remote health solutions. https://rhsusa.com/virtual-exam-room, 2020. [Online; accessed 2020-07-02].

[25] TytoCare. Tytocare | on demand medical exams. anytime. anywhere. https://www.tytocare.com/, 2020. [Online; accessed 2020-07-02].

[26] Hun Shim, Jung Hoon Lee, Sung Oh Hwang, Hyung Ro Yoon, and Young Ro Yoon. Development of Heart Rate Monitoring for Mobile Telemedicine using Smartphone. In *IFMBE Proceedings*, 2009.

[27] B. Ramachandran and S. Bashyam. Development of real-Time ECG signal monitoring system for telemedicine appalication. In *Proceedings of the 3rd International Conference on Biosignals, Images and Instrumentation, ICBSII 2017*, 2017.

[28] Aparna Lakhe, Isha Sodhi, Jyothi Warrier, and Vineet Sinha. Development of digital stethoscope for telemedicine. *Journal of Medical Engineering and Technology*, 2016.

[29] Uzzal Kumar Prodhan, Mohammad Zahidur Rahman, Israt Jahan, Ahsin Abid, and Mohtasim Bellah. Development of a portable telemedicine tool for remote diagnosis of telemedicine application. In *Proceeding - IEEE International Conference on Computing, Communication and Automation, ICCCA 2017*, 2017.

[30] Daniel Sa Pina, Antonio Augusto Fernandes, Renato Natal Jorge, and Joaquim Gabriel Mendes. Development of a portable system for online EMG monitoring. In *exp.at 2015 - 3rd Experiment International Conference: Online Experimentation*, 2016.

[31] Ana Sofia Simões Correia Rafael. Development of a prototype for EMG in telemedicine. Master's thesis, Faculdade de Engenharia da Universidade do Porto, 2019.

[32] TornadoWeb. Tornado web server — tornado 6.0.4 documentation. https://www.tornadoweb.org/, 2020. [Online; accessed 2020-07-02].

[33] I. Fette, Inc. Google, A. Melnikov, and Isode Ltd. RFC6455 - The WebSocket Protocol. *Journal of Chemical Information and Modeling*, 2011.

[34] Fenopix. Beautiful HTML5 Charts & Graphs | 10x Fast | Simple API. https://canvasjs.com/, 2020. [Online; accessed 2020-07-02].

[35] S. Egger, T. Hossfeld, R. Schatz, and M. Fiedler. Waiting times in quality of experience for web based services. In *2012 4th International Workshop on Quality of Multimedia Experience, QoMEX 2012*, 2012.

[36] Khronos Group. WebGL, 2015.

[37] Fyrd. Can I use... Support tables for HTML5, CSS3, etc. https://caniuse.com/#feat=webgl, 2020. [Online; accessed 2020-07-02].

[38] Charles Marion and Julien Jomier. Real-time collaborative scientific WebGL visualization with WebSocket. In *Proceedings, Web3D 2012 - 17th International Conference on 3D Web Technology*, 2012.

[39] Xu Hui, Wei Lihao, Wang Tian, and Luo Xiaoben. WebGL based HTML5 application performance analyzer. *Journal of Convergence Information Technology*, 2012.

[40] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the javascript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 351–361, 2016.

[41] webpack. webpack. https://webpack.js.org/, 2020. [Online; accessed 2020-07-02].

[42] Microsoft. TypeScript - JavaScript that scales. https://www.typescriptlang.org/, 2020. [Online; accessed 2020-07-02].

[43] mrdoob. three.js – JavaScript 3D library. https://threejs.org/, 2020. [Online; accessed 2020-07-02].

[44] Debra A. Lelewer and Daniel S. Hirschberg. Data compression. *ACM Computing Surveys (CSUR)*, 1987.

[45] lz4. GitHub - lz4/lz4: Extremely Fast Compression algorithm. https://github.com/lz4/lz4, 2020. [Online; accessed 2020-07-02].

[46] Oberhumer, Markus. oberhumer.com: LZO real-time data compression library. http://www.oberhumer.com/opensource/lzo/, 2020. [Online; accessed 2020-07-02].

[47] Gailly, Jean-loup and Adler, Mark. zlib Home Site. https://zlib.net/, 2020. [Online; accessed 2020-07-02].

[48] Mozilla. Web Workers API - Web APIs | MDN. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API, 2020. [Online; accessed 2020-07-02].

[49] Wermke, Andy. Web worker meets worker threads - threads.js. https://threads.js.org/, 2020. [Online; accessed 2020-07-02].

[50] nodeca. pako 1.0.11 API documentation. http://nodeca.github.io/pako/, 2020. [Online; accessed 2020-07-02].

[51] Mozilla. IndexedDB API - Web APIs | MDN. https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API, 2020. [Online; accessed 2020-07-02].

[52] Mozilla. Window.sessionStorage - Web APIs | MDN. https://developer.mozilla.org/en-US/docs/Web/API/Window/sessionStorage, 2020. [Online; accessed 2020-07-02].

[53] Mozilla. localForage. https://localforage.github.io/localForage/, 2020. [Online; accessed 2020-07-02].

[54] Wireshark Team. Wireshark · Go Deep. https://www.wireshark.org/, 2020. [Online; accessed 2020-07-02].

[55] d3. D3.js - Data-Driven Documents. https://d3js.org/, 2020. [Online; accessed 2020-07-02].

[56] Shutterstock. Rickshaw: A JavaScript toolkit for creating interactive time-series graphs. https://tech.shutterstock.com/rickshaw/, 2020. [Online; accessed 2020-07-02].

[57] charjs. Chart.js | Open source HTML5 Charts for your website. https://www.chartjs.org/, 2020. [Online; accessed 2020-07-02].

[58] Arction. Javascript High Performance Charts | WebGL Charts Library. https://www.arction.com/lightningchart-js/, 2020. [Online; accessed 2020-07-02].