FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Virtual reality web application for automotive data visualization

Tomás Sousa Oliveira

DISSERTATION

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Jorge Alves da Silva

Company Supervisor: João Manuel Leite da Silva

July 31, 2020

Virtual reality web application for automotive data visualization

Tomás Sousa Oliveira

Mestrado Integrado em Engenharia Informática e Computação

July 31, 2020

Resumo

A indústria automóvel permanece em constante desenvolvimento, com um olhar concentrado nas novas tecnologias a serem produzidas no mercado. A investigação em veículos autónomos continua em evolução e centenas de empresas mantêm o foco na investigação de novas formas de aperfeiçoar estes automóveis. A segurança na estrada é crucial na elaboração destes veículos, portanto é necessário que testes de segurança e desempenho na estrada sejam realizados regularmente. Aplicações de visualização de dados destes testes são fundamentais para ajudar a progressão da condução autónoma.

Atualmente, a Altran possui uma aplicação que ajuda a visualizar estes dados através de uma plataforma *web*, denominada *Kratos Visualization*. Esta aplicação foi desenvolvida com recurso à biblioteca *WebGL*, para ajudar as empresas que realizam experiências com veículos autónomos a visualizar e validar os dados que essas empresas recolhem. Estes dados podem ser relativos a experiências de condução autónoma, e podem conter bastante informação sobre o veículo e os objetos no ambiente ao seu redor. Além disto, os dados podem incluir imagens tiradas de câmaras posicionadas no veículo e nuvens de pontos a representar o espaço abrangente. O trabalho que conduziu à presente dissertação tem como objetivo explorar esta formas de visualização destes dados com recurso a técnicas de realidade virtual.

A realidade virtual é uma tecnologia capaz de criar uma experiência semelhante ao mundo real, ao usar técnicas que iludem os sentidos do utilizador. Para isto é necessário que o utilizador tenha aparelhos específicos, como um *Head-Mounted Display* que ajudam a simular essa experiência.

Para fortalecer a visualização de dados de condução autónoma foi criada uma aplicação denominada *Kratos VR*. Criada através da *framework A-frame* para funcionar em qualquer *web browser*, esta aplicação *web* mantém as funcionalidades do *Kratos Visualization* e foi adaptada para funcionar com qualquer *headset* de realidade virtual através de um *web browser*.

Foram realizados testes de desempenho e de memória para avaliar a aplicação e foi feita uma comparação com a aplicação original. Estes testes permitiram concluir que a realidade virtual pode ser utilizada para melhorar o progresso da condução autónoma, e que a aplicação desenvolvida providencia uma base sólida para futuras aplicações de realidade virtual que possam vir a ser realizadas dentro da Altran.

Abstract

The automotive industry remains in constant development, with a special look at the new technologies being developed in the market. The autonomous vehicle continues in its evolution and hundreds of companies keep the focus on investigating new ways to improve these cars. Road safety is crucial in the design of these vehicles, so safety and performance tests must be carried out regularly. Visualization applications of data based on these tests are critical to helping the continuous progress of autonomous driving.

Currently, Altran has an application that helps in visualizing this data through a web platform, called Kratos Visualization. This application was developed using the WebGL library, to help companies that carry out experiments with autonomous vehicles to visualize and validate the data that these companies collect. These data represent different sequences of autonomous driving experiences and can contain a lot of information about the vehicle and objects in the environment around it. Besides, the data may include images taken from cameras positioned in the vehicle and point clouds in the wide space. The work that led to this dissertation aims to explore this form of visualization of these data with the help of virtual reality.

Virtual reality is a technology capable of creating an experience similar to the real world, using techniques that deceive the user's senses. For this, the user must have certain devices, such as a head-mounted display that help to simulate that experience.

To strengthen the visualization of autonomous driving data, an application was created named Kratos VR. Created with the A-frame framework to work in any browser, this application maintains the functionality of Kratos Visualization and has been adapted to work with any virtual reality headset on the market, through a web browser.

Performance and memory tests were carried out to evaluate the application and a comparison was made with the original application. These tests allowed us to conclude that virtual reality can be used to improve the progress of autonomous driving and that the developed application provides a solid basis for future virtual reality applications that may be created within Altran.

Acknowledgements

I would like to express my gratitude to everyone that contributed to the completion of this dissertation.

To my supervisor in FEUP, Prof. Jorge Silva, for his exceptional amount of guidance, patience, and support, and especially giving his best to ensure this dissertation was well prepared. To all other professors in FEUP and Ghent University for their competence and determination, whose teachings have accompanied me in my academic life.

To my colleagues in Altran, for their quick and warm integration and encouragement. A special thanks to my supervisor in Altran, João Silva, for always guiding and helping me in every way possible, and being present every single time I needed.

To my friends, who motivated me in finishing projects on sleepless nights and helped me in having a great time on multiple occasions.

Finally, to my family for their encouragement, love, and support, and especially to my parents for helping me achieve my goals over the past twenty-plus years, always believing in me, and supporting me unconditionally in all the decisions of my life.

Thank you.

Tomás Oliveira

"The most important thing about a technology is how it changes people."

Jaron Lanier

Contents

List of Figures

List of Tables

Abreviaturas e Símbolos

2D	Two-Dimensional
3D	Three-Dimensional
AD	Autonomous Driving
ADAS	Advanced Driver-assistance Systems
API	Application Programming Interface
CPU	Central Process Unit
DK	Development Kit
ECS	Entity-Component-System
GIS	Geographic Information Systems
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HCI	Human Computer Interaction
HMD	Head Mounted Display
HTML	Hypertext Markup Language
IMU	Inertial Measurement Unit
JPEG	Joint Photographic Experts Group
JS	JavaScript
JSON	JavaScript Object Notation
LiDAR	Light Detection And Ranging
PNG	Portable Network Graphics
RAM	Random-access Memory
R&D	Research and Development
RF	Radio Frequency
ROS	Robot Operating System
RViz	ROS Visualization
TAM	Technology Acceptance Model
UI	User Interface
VR	Virtual Reality
WebGL	Web Graphics Library
WGS	World Geodetic System

Chapter 1

Introduction

This dissertation focuses on the field of Virtual Reality and Autonomous Driving, in which both Human-Computer Interaction (HCI) and Computer Graphics maintain a huge presence. This chapter will provide an overview of the motivation, context, and objectives of the project that originated in this dissertation.

1.1 Motivation

An autonomous vehicle is a vehicle that is capable of moving by itself, with no human help, by using a collection of sensors, such as radars, sonars, Light Detection and Ranging (LiDARs), GPS, and inertial measurement units (IMU), to identify its environment. Due to its enormous capability of improving driving safety, numerous companies already started investing in these technologies.

Advanced Driving Assist Systems (ADAS) are systems created to improve, automate, and adapt road and car safety, by minimizing the main cause of road accidents, the human error. By providing the driver with crucial information about traffic and giving suggested routes to avoid congestion, blockages, or closures, an increase in traffic performance is expected. These systems can also evaluate the fatigue or performance of the driver and thus make deliberate suggestions, as well as taking over the control from the driver to perform difficult maneuvers, like parking [1]. Developing an ADAS system requires state-of-the-art yet cost-effective Radio Frequency (RF) technology that can be embedded in the vehicle for exterior object detection and classification [2].

However, web applications that can assist the developers in creating these autonomous systems are incredibly scarce and not generic enough. One example is deck.gl, a WebGL framework to visualize large data sets of geographic data, created by Uber. It can render data about a vehicle on top of each other like its trajectory or figure, and present point clouds and polygons of other elements (vehicles, people, or objects) situated in the environment around it[3]. As such, a generic web platform for ADAS and Autonomous Driving (AD) visualization was created with Kratos Visualization, a WebGL application that provides and analyses results of autonomous driving experiments [4].

Still, there is an important element missing in the application, that could provide a better experience to the user. William R. Sherman and Alan B. Craig define virtual reality as a medium composed of interactive computer simulations that sense the position and actions of the participant and replace or augment the feedback to one or more senses, giving the feeling of being mentally immersed or present in the simulation [5]. Through the combined use of ADAS and VR technologies, we can expect another step in the impact of the results of AD experiments, promoting a better perception and interest in the application. With the constant increase of market competitiveness, the usability of these technologies can help in captivating an audience and relatively increase the value of a company.

1.2 Context

Kratos Visualization is a project created by Altran Portugal, in its research and development (R&D) department [6]. The purpose of this WebGL application is to show AD data about the trajectory of a certain vehicle, obtained through electronic systems (such for example, ADAS). Among many other functionalities, it can handle real-time viewpoints and perspectives, control the animation, and personalize the vehicles shown during a certain sequence of an AD experiment. An example of a sequence can be seen in Figure 1.1. The sequences always contain one main vehicle, also known as the ego, represented as the green element.

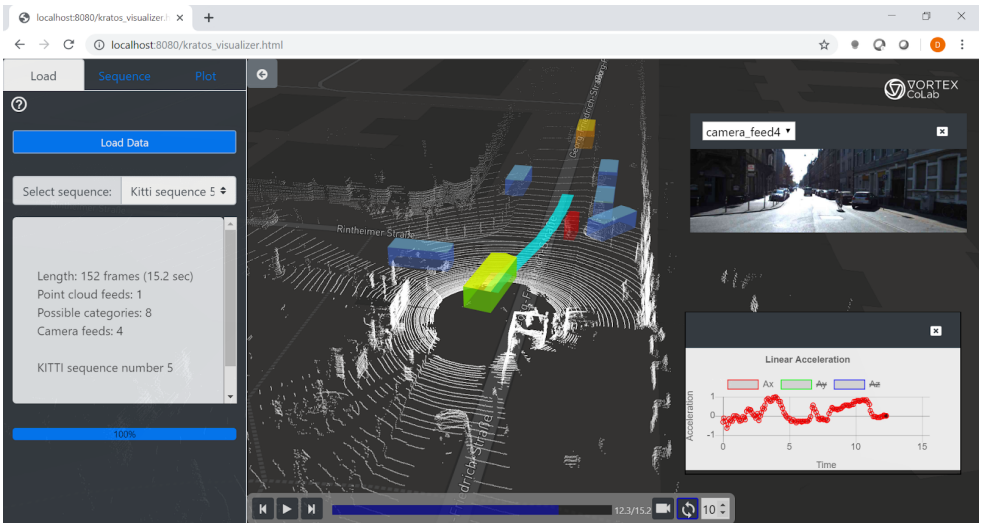

Figure 1.1: Current interface of Kratos Visualization

However, this application faces numerous restrictions and problems that should be solved by using VR technologies. For example, the user can only see a very limited number of perspectives, and the zoom tool has limitations in Mapbox, a plugin in WebGL. This form of visualization is also not so straightforward and easy to interpret since the perspectives of the camera are scarce and there is a need to see multiple sides of the sequence. That is why the use of virtual reality

technologies are being studied, to take advantage of their immersion and simple interaction. This application will be more strictly detailed in chapter 2.1.

This is why the usability of VR devices and techniques are being explored by developing Kratos VR. With VR, one can create a simulated environment, with the help of a headset and with or without hand controllers [7]. Kratos VR will serve as a complement to Kratos Visualization, with both tools providing different perspectives for the future of autonomous driving.

This thesis was proposed by Altran Portugal in the scope of assisted and Autonomous Driving (AD) research projects, developed within the company in collaboration with the Vortex-CoLab association [8]. This virtual reality visualization module is one of the modules composing the company's initiatives for creating a software infrastructure to drive forward the development of AD platforms. Modules already tackled include Perception, V2X communication, World Modeling, data labeling, decision & control, infotainment, data visualization. The work developed during this thesis is integrated mainly in the data visualization module, further extending the ecosystem of the project towards an integrated and harmonized set of solutions.

1.3 Goals

The main goal of Kratos VR is to facilitate the visualization of AD data obtained by ADAS, or other electronic systems, from autonomous driving experiments, using Virtual Reality techniques. As such, the objective is to develop a VR scene capable of showing information about that data, as well as being possible to interact with it. The user must be able to easily view, understand, and control all the elements in his surroundings while using a VR headset and one or two controllers. This data can have a large variety of formats including point clouds and trajectories.

As such, a plan with the following set of tasks to complete for the application was developed:

1. Allow the user to choose any ADAS sequence to visualize;
2. Present all the data provided by that ADAS sequence, including vehicles (cars, vans, trains, bicycles...) in the form of polygons, trajectories and point clouds;
3. Show information about each vehicle, including its category and position, when the user hovers over it with the VR controller;
4. Allow the user to control the point of view of the camera using VR controllers;
5. Allow the user to customize each element, including changing its visibility, color, and opacity, or even toggle between display its wireframe;
6. Allow the user to control the animation of the sequence, including pausing the animation, changing the frame, fast-forwarding or even going backwards, using VR controllers;
7. Show statistics about the ego vehicle, including linear acceleration and velocity and angular velocity, while the animation is running;

8. Display camera feeds of the sequence, while the animation is running;

9. Allow any VR headset with 1 or 2 controllers to fully interact with the application.

One of the main objectives of Kratos VR is that the application can be fully used by all the major VR devices in the market. As such the application will not limit the number of VR controllers or types of controls that each of them has.

1.4 Contributions

This application will serve as one of the first VR additions to the autonomous driving continuous progression. The possibility of showing polygons, point clouds, data plots, and cam feeds in the same VR environment demonstrates the real advancements that this field can have.

The possibility to create a publication based on this thesis is being explored, to demonstrate the contributions of this project to the area. Besides this, Altran is considering to start a new virtual reality development area inside its R&D department, and this application will be a base ground for all the following ones, as well as a new addition to the projects that Altran currently possesses.

1.5 Organization of the Dissertation

Besides the introduction, this document contains five chapters. An introduction to the project was already made, and its goals were already specified. In chapter 2 a study on the current state of the art and the areas related to this dissertation is presented. All the necessary background and concepts will be defined and the tools needed for the development of the application will also be explored.

Chapter 3 will provide a detailed implementation of the visualization component of Kratos VR. All the methods and the discussed approaches that were tested and examined will be evaluated. Chapter 4 will focus on the interaction component. This part will mostly specify the interface development and the progress made while implementing the VR controllers and input.

The results of the application will be presented in chapter 5. In this chapter, a brief analysis of the performance and a comparison between Kratos VR and Kratos Visualization and the usability of different VR devices will be demonstrated. An experiment with participants that tested this application will also be analyzed.

Finally, chapter 6 will present the conclusions made while developing this project, and the future work that will be developed for the application.

Chapter 2

State of the Art

In this chapter, the history and state of the art of the developments made in the areas of autonomous driving and virtual reality will be analyzed. The technologies that made significant advancements in these areas will also be addressed and for some of them, a detailed discussion on how they could be integrated into the project will be provided.

Section 2.1 introduces Kratos Visualization and its features and contributions to the AD area. The infrastructure made in ROS will also be clarified and the data set used in the application will also be examined. Section 2.2 gives a brief introduction to the history of virtual reality and an analysis of its frameworks, devices, and main technologies. Section 2.3 provides a precise discussion of the tools and libraries related to the features that need to be implemented in Kratos VR. Finally section 2.4 presents the notable projects that contributed to the progress of AD in virtual reality scenarios.

2.1 Kratos Visualization

Returning to the application mentioned before, a screenshot of the application is shown in Figure 2.1. This application was made using Deck.gl, a large-scale, WebGL-based data visualization library, with the goal of presenting all the AD data provided by an ADAS, and allow the user to customize and control it [9].

On the top left of the screen, there are three different categories of menus. The Load menu is where the user selects a sequence to load. This sequence appears in the middle of the screen. The Sequence menu is where the personalization of the elements occurs. It is possible to change the color, visibility, and the opacity of the elements and the width of the trajectories. The elements, point clouds, and camera feed images that are presented to the user originate from data sets that represent objects or space. Since the sequences can have an abundant number of these types of elements, and with point clouds included, there is a demand for high storage capacity.

In addition to this, feeds of the cameras positioned along with the car are exhibit on the right of the screen, obtained by the Kitti Vision Benchmark Suite, described in chapter 2.1.2. The third

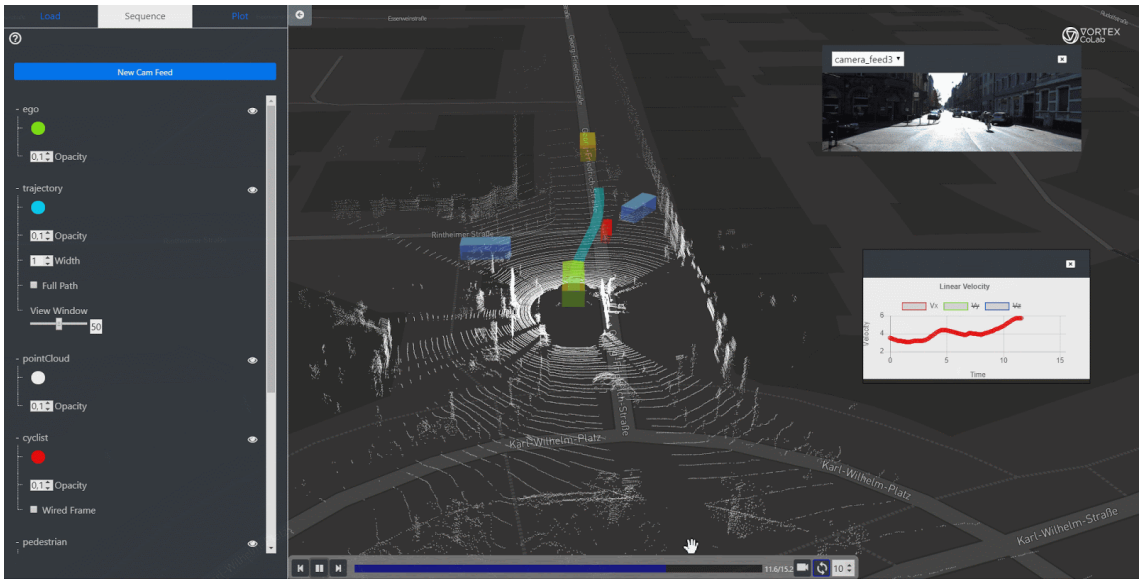

Figure 2.1: WebGL application functionalities

and final menu, the Plot menu, is where one can select data plots to analyze. For the moment there are only graphs for the linear and angular velocity and linear acceleration of the main vehicle.

On the bottom of the screen, one can pause and change the frames of the animation, the frame rate, and the perspectives of the camera. The map presented on screen was created by Mapbox, an online provider for real-world maps [10].

The data visualized in the application were collected from several sources, including video, point clouds, location, map, velocity, labels, or bounding boxes all of which originate from multiple sources such as LiDARs, cameras, inertial navigation systems(GPS/IMU), etc [9]. This data set can be viewed simultaneously and easily controlled via the platform interface. It is expected that the VR application will contain these current functionalities and many more features that will be discussed in Chapter 3.

In sections 2.1.1 and 2.1.2, the infrastructure of Kratos Visualization and its data set will be reviewed. Most of it will be reused in Kratos VR.

2.1.1 ROS

ROS is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms [11], that provides a structured communications layer above the host operating systems of a heterogeneous compute cluster [12]. The supplied data is based on ROS, through rosbags (logs) or in real-time.

This data can be accessed and visualized with RViz (ROS Visualization), which is a 3D visualization tool for ROS. RViz has multiple built-in functionalities that help visualize elements like cameras, point clouds, or an entire map [13].

This tool called Visualization node for ROS using OpenVR allows the visualization of data through ROS, however it is still in development and only has been tested with the HTC Vive [14].

Roslibjs is the main JavaScript library to interact with ROS. This library is completely open-source and provides a lot of essential ROS functionalities to work with JS [15].

2.1.2 Data Set

Kratos Visualization utilizes data sets from the KITTI Vision Benchmark Suite, with some plans of using other data structures in the future like the Oxford Car data set [16]. The structure of the data set is also an integral part of the application, with the files in JSON format.

2.1.2.1 Source

The data set used is part of the KITTI Vision Benchmark Suite, provided by the Karlsruhe Institute of Technology. These data sets were taken from a normal station wagon that contains two high-resolution video cameras, represented in Figure 2.2. It includes camera images, laser scans, high-precision Global Positioning System (GPS) measurements, and Inertial Measurement Unit (IMU) accelerations from a combined GPS/IMU system. The main sensors used are the following [17]:

- One Inertial Navigation System (GPS/IMU): OXTS RT 3003
- One Laser scanner: Velodyne HDL-64E
- Two Grayscale cameras, 1.4 Megapixels: Point Grey Flea 2 (FL2-14S3M-C)
- Two-Color cameras, 1.4 Megapixels: Point Grey Flea 2 (FL2-14S3C-C)
- Four Varifocal lenses, 4-8 mm: Edmund Optics NT59-917

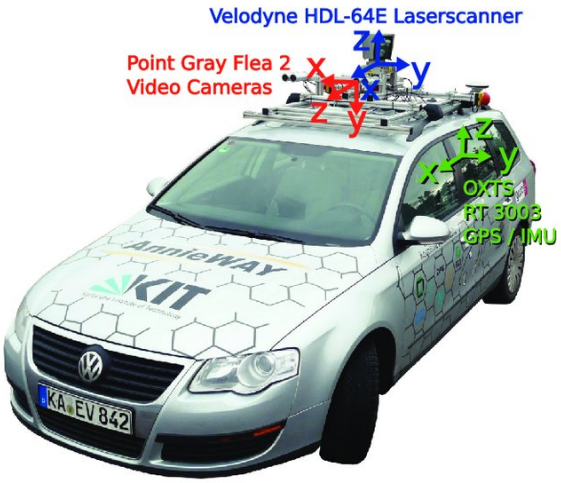

Figure 2.2: Station wagon from the KITTI Vision Benchmark Suite

The laser scanner works at 10 frames per second, capturing approximately 100k points per cycle, forming a point cloud. The vertical resolution of the laser scanner is 64. The cameras are approximately leveled with the ground plane and its images are cropped to a size of 1382 x 512 pixels. After rectification, the images get slightly smaller. The cameras are triggered at 10 frames per second by the laser scanner (when facing forward) with shutter time adjusted dynamically (maximum shutter time: 2 ms).

The data set for a certain sequence consists of the following information:

- Raw (unsynced and unrectified) and processed (synced and rectified) gray scale stereo camera feed images (0.5 Megapixels, stored in png format);
- Raw (unsynced and unrectified) and processed (synced and rectified) color stereo camera feed images (0.5 Megapixels, stored in png format);
- 3D Velodyne point clouds (100k points per frame, stored as binary float matrix);
- 3D GPS/IMU data (location, speed, acceleration, meta information, stored as text file);
- Calibration (Camera, Camera-to-GPS/IMU, Camera-to-Velodyne, stored as text file);
- 3D object tracklet labels (cars, trucks, trams, pedestrians, cyclists, stored as xml file).

Here, "unsynced and unrectified" refers to the raw input frames where images are distorted and the frame indices do not correspond, while "synced and rectified" refers to the processed data where images have been rectified and undistorted and where the data frame numbers correspond across all sensor streams. For both settings, files with timestamps are provided [17].

The main objective of this data set is to add an extra step to the development of robotics algorithms and computer vision targeted at AD.

2.1.2.2 Structure

This structure was created for the previously mentioned application by its creators since the files provided by the KITTI Vision Benchmark were not prone to direct loading and mapping with the desired technologies and functionalities.

This section will present the structure of these files, and how they were integrated inside the application. All the files are in JSON format, inside the data folder.

This folder always contains a directory configuration file and the folders of the available sequences. This file is loaded right after running the application. It contains information about the possible sequences to visualize and their name, path, and database table, as seen in Listing 2.1.

```
1 {
2     "seq05":{
3         "sequenceName":"Kitti sequence 5",
4         "sequencePath": "seq05/",
```

```
        "sequenceTable":"inserttest14"
    },
    "seq14":{
        "sequenceName":"Kitti sequence 14",
        "sequencePath": "seq14/",
        "sequenceTable":""
    }
}
```

Listing 2.1: Directory configuration file

Each sequence folder contains a configuration file that provides information about the names of the files of the point clouds and each category of elements, including the ego (main vehicle). Besides this, the file also provides knowledge about the frame rate, the number of frames in the sequence, the provider of the sequence, the available camera feeds, and the labels of the data plots. The structure for this file can be seen in Listing 2.2.

```
{
    "pCloud" : ["pointCloud"],
    "category" : ["pedestrian","car","sit","van","cyclist","tram","
       misc","truck"],
    "frameRate" : 10,
    "nframes" : [0,152],
    "cameraFeed":["camera_feed1","camera_feed2","camera_feed3","
       camera_feed4"],
    "description" : "KITTI sequence number 5",
    "plottable" : ["vx", "vy", "vz", "ax", "ay", "az", "wx", "wy", "
       wz"]
}
```

Listing 2.2: Sequence configuration file

Besides this configuration file, the sequence folder also contains a file, for each frame, for each category of elements, ego, and point clouds. This means that the name of the file for all vans in frame 10 would be "van10". Since the vehicles are created in the form of polygons, the structure for each category file contains the id for the element, and the position and height of the polygon, as shown in Listing 2.3.

```
[{
    "id":19,
    "polygon": [
        [18.9612,26.872,-1.52947],
        [15.4334,23.7353,-1.49593],
        [14.762,24.3404,-1.45957],
```

```
        [14.24,25.0783,-1.42464],
        [17.7679,28.2149,-1.45818]
    ],
    "height":1.87298
}]
```

Listing 2.3: Category file

However, the file structure for the ego vehicle is very different, since the other elements are positioned like the ego is in the origin point. As such, the file contains information about its geographic coordinates, principal axes, and linear acceleration and linear and angular velocity, as illustrated in Listing 2.4.

The point cloud files are in DRC format, a binary format internally used by the Draco compression library and extremely large and unreadable. There is a point cloud frame for each file, and methods for its decompression will be strictly detailed in section 3.5.

```
[{
    "lat":49.015786,
    "long":8.436498,
    "alt":115.646141,
    "roll":0.030034,
    "pitch":0.001851,
    "yaw":0.548948,
    "vx":8.787442,
    "vy":-0.009430,
    "vz":-0.006625,
    "ax":1.624350,
    "ay":0.176316,
    "az":9.607892,
    "wx":0.006231,
    "wy":-0.007153,
    "wz":0.009431
}]
```

Listing 2.4: Ego file

Besides this, the folder also contains sub-folders for each group of camera feeds, with as many images as the number of frames. Each image can be in JPEG or PNG format and can be with or without color. These images were taken by the cameras mentioned in section 2.1.2.1. An example of an image can be seen in Figure 2.3.

Since most of the infrastructure of the Kratos Visualization application can be reused, Kratos VR can work with the same data structure.

Figure 2.3: Camera feed image

2.2 Virtual Reality

There are countless definitions of the term Virtual Reality. Gary Bishop and Henry Fuchs describe it as "a real-time interactive graphics with three-dimensional models, combined with a display technology that gives the user the immersion in the model world and direct manipulation" [18]. On the other hand, Michael Gigante provides a more detailed explanation in his book: "VR is characterized by the illusion of participation in a synthetic environment than external observation of such an environment. It relies on three-dimensional, stereoscopic, head-tracked displays, hand/body tracking, and binaural sound (...)" [19]. Both definitions seem to agree that VR can simulate a real-world scenario, which is a key factor that originated the Kratos VR idea.

According to William R. Sherman and Alan B. Craig, there are four key elements of a virtual reality experience [5]. These elements are the following:

- Immersion - This element is arguably the main reason VR became so successful. Characterized as the perception of being physically present in a non-physical world, it can gather the attention of the user by using a display that allows the user to look in any direction and tracking his motion [20]. The displays mostly used in VR applications are head-mounted displays (HMD), like the Oculus Go or the HTC Vive Pro (described in section 2.2.4).

- Interactivity - Inside the VR experience, interactivity provides the possibility to interact with and modify the virtual world. Jonathan Steuer claims that interactivity depends on factors like speed (the rate that the actions of the user are integrated, reflected and perceived), range (how many outcomes could happen from any action) and mapping (ability to produce natural results in response to the actions of a user) [21].

- Sensory feedback - The role of this element is to present information of the virtual world to the senses of the user. The most common is the visual sense, with the audio and touch sense not falling behind. One obvious example is the position of the user. By moving their heads or bodies, using HMD with or without controllers, the participants will be given the illusion that they are moving inside the virtual world.

- Virtual World - The virtual world or medium can be described as an imaginary space made of elements rendered by computer graphics. The creator of a certain virtual world can define the relations and interactions between each of these elements.

One can see how VR can be immersive and interactive, however according to Grigore Burdea and Philippe Coiffet, people are not aware of the imagination feature. This feature is important for how well an application can perform, and as such completes the three I's of VR, as seen in Figure 2.4. The triangle in the figure can be perceived as a triangle, however, it only exists in the imagination of the reader [22].

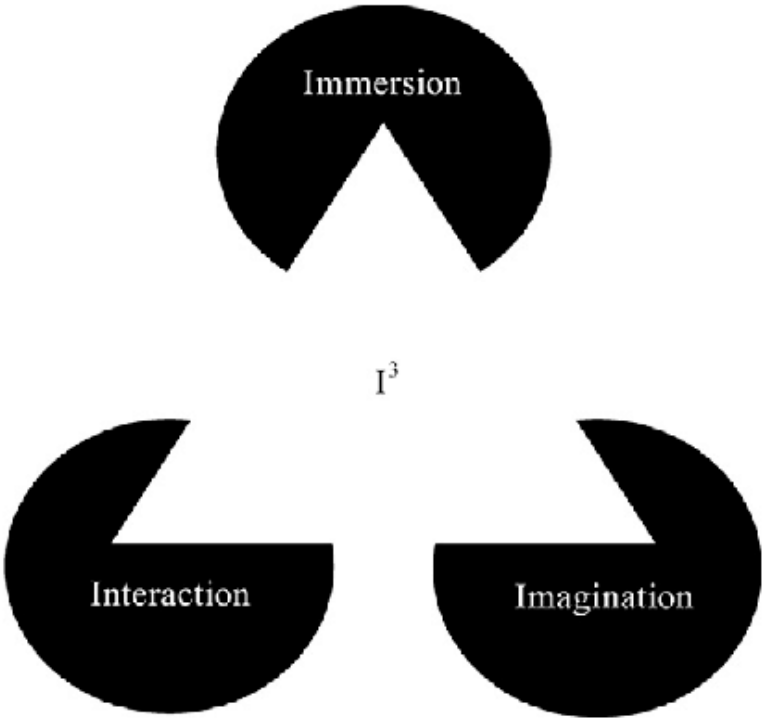

Figure 2.4: Three I's of VR

2.2.1 History

There have been some advancements in the area of VR that date back to the middle part of the 20th century. The use of the term "virtual reality," was first introduced around 1985 when Jaron Lanier and Thomas Zimmerman, founded VPL Research. They developed a range of VR equipment, including goggles and gloves, needed to experience what he called "virtual reality", as seen in Figure 2.5.

Despite this, there were already multiple devices created to simulate virtual world experiences like the Sensorama in 1956 or the Telesphere Mask in 1960, both created by Morton Heilig, as seen in Figure 2.6. Other milestones were the Until Headsight, created by Comeau and Bryan and the Ultimate Display, invented by Ivan Sutherland [23]. It was not until the 2010s when the first

Figure 2.5: VR Equipment by VPL Research

high-quality VR headsets became available to the public, with the Oculus Rift. Later on, many companies started developing more VR devices and types of equipment like the Oculus Quest, HTC Vive and Google Cardboard gain huge success.

Figure 2.6: Sensorama

2.2.2 WebXR

WebVR is an open specification that allows anyone to easily develop VR applications in any web browser. This API is a complement to WebGL, a Javascript API, specially designed to enable the web browser to display 3D graphics without additional plug-ins [24]. This API was mainly developed to target the first wave of VR headsets that were released to the public, like the Oculus Go, the HTC Vive, and the Daydream phones. WebVR provides support for a huge variety of VR devices to use in a browser. One key element that makes WebVR a great addition to the web, is that any developer can create a VR application without the need for any special software [25].

The new and experimental WebXR is an extension of WebVR, which also supports augmented reality and has better rendering and controller management, in addition to other important optimizations. The main reason for its creation was to support a larger range of immersive equipment, like augmented reality devices such as the Microsoft HoloLens and Magic Leap One and any mobile phone. WebVR had many issues implementing these new additions, and as such, was deprecated in favor of WebXR. One of its main goals is to allow these technologies to be added to new or existing web sites [26]. Both WebVR and WebXR can be used with any VR headset and contain suitable frameworks to display 3D graphics. All major VR enabled browsers to support this API, including Google Chrome, Oculus browser, Firefox Reality, and Samsung Internet. One example provided by the Mixed Reality Blog details the level of design sketches created for the XR store application, in Figure 2.7. These were focused on the four possible user experiences, using only the desktop, augmented reality, and VR with three or six degrees of freedom (DoF) [27].

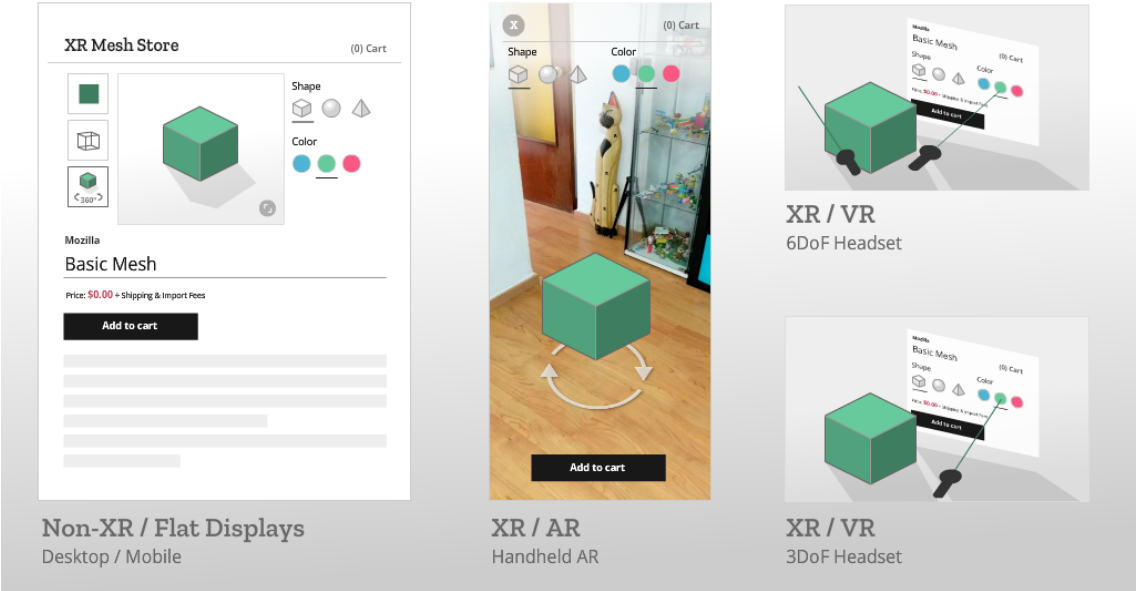

Figure 2.7: XR Store

Despite the significant difference between WebGL and WebVR or WebXR, there have been some successful attempts to convert applications between them, however, this depends on their size and number of features [28]. Converting an application like Kratos Visualization to WebXR would take a lot of time, using the same technologies. However, using different libraries or frameworks that are more suitable to the VR environment for each feature can be easier to implement.

2.2.2.1 Frameworks

Since the development of WebXR, multiple frameworks have been adapting their code to support this new API, namely A-Frame, three.js, and babylon.js. In the following sections, these

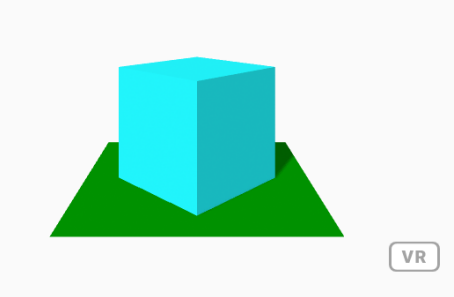

Figure 2.8: A-Frame scene example

frameworks will be discussed and a comparison between them will be analyzed in section 2.2.2.4.

2.2.2.2 A-Frame

A-Frame was released in December 2015, as an object driven framework to run with only HTML. One of the main reasons for its simplicity is the fact that any prior installation or build steps to run the code are not needed [29]. A-frame was based on three.js and build as an open-source Domain Specific Language (DSL) for generating in-browser VR content and there is a curated set of new artifacts and properties that can be used by any user via a simple import [30].

It involves just an HTML file and opening it in a browser, and because it is built on top of HTML, almost all of the libraries and tools of three.js work with A-Frame. If one has experience in HTML, close to none learning is needed for A-Frame [31]. For these reasons, A-Frame is the obvious choice for people who have little experience in VR. To be considered an A-Frame scene, the entities have to be surrounded by the tag <*a-scene*>. There can only be one scene in a file. In Listing 2.5 and Figure 2.8, there is a small example of an entity in A-Frame, representing a box on top of a plain.

```
1 <a-scene background="color: #FAFAFA">
2       <a-box position="0 0.5 -4" color="#1CD3D9" shadow></a-box>
3       <a-plane position="0 0 -4" color="green" shadow></a-plane>
4     </a-scene>
```

Listing 2.5: A-Frame code example

Even though the previous example is only a glimpse of the amazing concepts A-Frame can create, there have been many recent developments surging in this area. One example is the creation of a virtual museum, with 360º images, where its creators demonstrated that, despite some issues, developing in this framework can be quite simple [32].

Additionally, A-Frame contains an entity-component-system (ECS) architecture. Entities are objects representing containers, where components can be attached. Components are data or modules containers that can implement functionalities, appearances and behaviours to these entities. On the other hand, systems are an optional way to handle management, logic and a global scope

to a class of components. The ECS architecture is very common in 3D and game development and includes the following advantages [33]:

- Greater flexibility when defining objects by mixing and matching reusable parts;
- Eliminates the problems of long inheritance chains with complex interwoven functionality;
- Clean design by decoupling, encapsulating, modularization and re-usability;
- Scalable way to build a VR application in terms of complexity;
- Proven architecture for 3D and VR development;
- Extension of new features.

2.2.2.3 Three.js

Initially released in 2010 as an API to develop 3D software using WebGL, it was not until 2019 that this framework started developing for WebXR. This framework is more complex than A-Frame and contains a lot of extra functionalities for VR implementations [34]. However, since A-Frame is based on three.js, it provides full access to its API [35].

To draw the elements, like vehicles, pedestrians, or trajectories, the three.js feature *Object3D* is needed. To create a polygon, one can create an A-Frame entity and then register it with the *Object3D* component. Afterward, the object can be customized, like changing its position, color, or material. An example with the component *obj* registered in entity with id *box* can be seen in Listing 2.6.

```
1  <script>
2      AFRAME.registerComponent('obj', {
3        init: function () {
4          let boxObj = this.el.object3D;  // THREE.Scene
5        }
6      });
7  </script>
8  <body>
9      <a-scene>
10         <a-entity id="box" obj></a-entity>
11     </a-scene>
12 </body>
```

Listing 2.6: A-Frame component registration example

2.2.2.4 Framework Comparison

Other frameworks like babylon.js, a widely known JavaScript library to easily render 3D primitives in a browser, are more specifically used in game development [36]. It also has a more active

community [37]. There are multiple VR frameworks like PlayCanvas and Clara.io, but for performance, fundamentals, and financial reasons, only the other three were considered. The table 2.1 provides a detailed comparison between some of the major frameworks, as well with game engines, mentioned in Section 2.2.3 [38]:

Framework	Scripting	Integrated Networking	Integrated Physics	WebXR
A-frame	Javascript	No	No	Yes
Babylon.js	Javascript / Typescript	No	Yes	Yes
Clara.io	Javascript	No	Yes	Yes
PlayCanvas	Javascript	Yes	Yes	Yes
Three.js	Javascript	No	No	Yes
Unity	C++ / C#	Yes	Yes	No
Unreal Engine	C++	Yes	Yes	No

Table 2.1: Frameworks comparison

2.2.3 Game Engines

Considering that the previously discussed frameworks were created for web applications, there was a need for analyzing technologies that could handle high-storage demand more efficiently. As such, game engines like Unity and Unreal Engine were also studied. The main reason for their involvement is the fact that web browsers do not support large files with a huge load of data. As such, using a game engine could help in reducing those concerns.

2.2.3.1 Unity

Unity is a game engine that allows the use of VR specific functionalities. There are countless VR games and applications created in Unity all over the market, and since it is free to use, lots of projects and tutorials are open to the public all over the Internet. Various conference papers provide full support on how to use VR components in Unity [39]. With its easily understandable tools and well simpler interface, creating a VR application would not take much work.

Unity provides full support for all major VR headsets and controllers on the market. Creating the necessary elements as well as a real-world map would not take much effort. A suitable menu would also be simple to customize and implement since there are countless free community projects with these features [40].

However, issues like adding point clouds and trajectories would take a larger study. Working on the server-side of the project and at the same time using data that should not be available to the public would require a lot of time and effort. ROS already contains integration tools for Unity, however, its development is still on its early stages and countless issues and bugs could occur [41].

2.2.3.2 Unreal Engine

The other studied game engine, Unreal Engine, is a well-known platform developed by Epic Games. Both Unity and Unreal are powerful engines, however Unreal are well known to have more 3D features than the former, while Unity is more suitable to 2D applications. Nonetheless, Unity is still easier to use and contains a more active community [42] [43]. Both of these engines are free if the application does not profit with its sales, which would also be a concern in the development of the project [44].

Concerning VR development, both engines have made significant advancements in the area. Table 2.2 presents a detailed comparison for both engines, regarding VR functionalities.

Functionality	Unity	Unreal Engine
Percentage of VR games in market	60%	25%
Number of supported platforms	28	15
Number of free assets	50000	10000
Assets quality	Lower	Higher
Number of courses	5500	2200
VR topics in forums	6100	4600
Programming language	C++ or C#	C++
Open source	No	Yes

Table 2.2: Game engine comparison

In conclusion, Unreal Engine is designed for more experienced programmers, especially with C++ knowledge. Starters on the VR development area have more to gain by using Unity [45].

However, to contradict the presented advantages for Unity and Unreal Engine, a new infrastructure would have to be created, since no developed feature created for Kratos Visualization could be reused. Using a game engine would require creating an application, and as such, people would have to install software, instead of only accessing a website. As such, using web frameworks would simplify the development process.

2.2.4 VR Devices

This section will explore the VR devices that are currently on the market. One of the main goals of Kratos VR is that this application can be responsive to any available HMD. For this to happen, there was a need to have the necessary knowledge about what type of headsets and controllers exist in the market. These devices can have one, two, or even none controllers, and can have three or six DoF. The controllers are even trickier since they hugely differ on the number and type of buttons and switches.

2.2.4.1 Current Headsets

Currently, there are dozens of VR devices in the market, each of them unique in their own way. A head-mounted display (HMD), worn on the head with lenses and semi-transparent mirrors

embedded in eyeglasses, displays images in front of each eye. There are visual, haptic, and multi-sensor types of HMDs but for terms of simplicity only the visual HMDs were considered since the other ones will not be used. Figure 2.9 shows the headset and controller of the Oculus Go.

Figure 2.9: Oculus Go

Visual HMDs are usually divided into two categories: mobile and wired. All of the mobile devices are wireless and do not need any external computer to be used. Some of these headsets have support for positional tracking, in which a precise position and movement of the head, controllers, and body parts of the user can be detected in the Euclidean space, which means devices with six DoF. Others only support orientation tracking, which means only the rotation of the head is detected, and as such, with three DoF. Some known examples are the Google Cardboard, Samsung Gear VR, and the Oculus Go and Quest. Also, a few headsets, like the Google Cardboard, are dependent on a smartphone and its technology [46].

The second category is the wired HMDs, where usually a powerful PC is needed to support the device. Typically using positional tracking, notorious examples are the Oculus Rift S, Playstation VR, and the HTC Vive Pro and Cosmos. Table 2.3 presents key factor differences in the available headsets as of right now [47].

Headset	Price	Tracking	DoF	Number of Controllers
Google Cardboard (Phone needed)	13€	Orientation	3	0
HTC Vive Cosmos (Full)	829€	Positional	6	2
HTC Vive Pro	679€	Positional	6	2
Oculus Go (32 GB)	159€	Orientation	3	1
Oculus Quest (64 GB)	449€	Positional	6	2
Oculus Rift S	449€	Positional	6	2
Playstation VR	300€	Positional	6	2
Samsung Gear VR (Phone needed)	90€	Orientation	3	1
Valve Index	799€	Positional	6	2

Table 2.3: Headset comparison

2.2.4.2 Controller Input

When studying the VR controllers, one can notice a lot of differences regarding the aspect, input, and type of buttons. An example of this can be seen in Figure 2.10, where the Oculus Go controller (on the right) has two buttons, a touchpad, and a trigger, while the Oculus Quest controller contains three buttons, two triggers, and a thumbstick.

Figure 2.10: Difference between VR controllers: left: the Oculus Quest controller; right: the Oculus Go controller

A-Frame already provides full support for each of these different input options. With already created components, one only needs to create an entity for each hand and specify which type of controller will be used. Regarding the different inputs, a listener for each type of click, trigger, or touch needs to be created. This can be implemented by adding the listener event to an already established component. Listing 2.7 demonstrates the event when someone presses on a certain touchpad on the right-hand controller [47].

```
1 AFRAME.registerComponent('right-hand-controls', {
2     init: function () {
3         const { el } = this;
4
5         el.addEventListener('touchpadchanged', function () {
6             console.log('Touch pad pressed');
7         });
8     }
9 });
```

Listing 2.7: A-Frame controller input example

In the meantime, the application will first be tested with the Oculus Go and with the Oculus Quest, with plans of testing with other headsets shortly.

2.2.4.3 Effects of using HMDs

While VR can simulate a wonderful experience to some, others might feel some discomfort while wearing the headset. This can be very common, since wearing HMDs for some time can cause motion sickness, nausea, fatigue, and other symptoms.

Motion sickness is a condition in which a conflict exists between visually perceived movement and the vestibular system's sense of movement. Scene motion in VR is a visual movement that occurs when seeing the virtual world, that does not happen in the real world. Currently, there are three types of motion sickness: cybersickness (when inside a virtual reality environment); gaming sickness (when playing VR video games); simulator sickness (when using driving or flight simulators). Some frequent symptoms that users may experience include dizziness, eyestrain, fatigue, headaches, and nausea [48].

Besides this, there are some effects of the weight of VR headsets. Creators of VR devices must be aware of the pressure and weight of the HMD when developing these headsets [49]. Table 2.4 details the causes that may increase or decrease motion sickness.

Option	Less Sickness	More Sickness
Frequency	Experience with VR	No experience with VR
Position	Sitting	Standing
Screen Size	Small Viewing Area	Large Viewing Area
Speed	Walking	Running
Textures	Realistic	Generic or Repetitive
Time	Short Duration	Long Duration

Table 2.4: Motion sickness causes

Besides VR developers having to be aware of these effects when creating VR applications, they must enforce the following essential practices [50]:

- Shrink the field of view;
- Allow people to control the movement;
- Add fixation points for users to be able to focus on and maintain their eye stability;
- Create scenes that let users rest a little bit after a while;
- Reduce virtual rotations and angular velocity.

2.3 Necessary Tools and Libraries

To know the tools and libraries that are needed to use in the application, a decision about which framework to use had to be made. Since A-Frame was designed to encompass a wide variety of tools and includes the usability of three.js, this was chosen as the main framework for Kratos VR.

In this section, the tools and applications needed to fully develop Kratos VR are explored. The variety of web applications necessary to create each of these tools will be explored, and its A-Frame versions will be meticulously discussed.

2.3.1 Real World Map

To create the application, a map of a real-world location is necessary, however, implementations of maps in VR are scarce, especially in A-Frame. This map needs to accept any coordinates on Earth and provide the respective location, with roads and buildings included, around it.

One of the earlier discussions when concerning this issue was using Mapbox, the platform used in Kratos Visualization, but there was not any A-Frame or three.js Mapbox acceptable components. The only one created with Mapbox was made for maps in large-scale only [51]. The only way to use Mapbox in VR was with its 3D Unity integration [52]. Even so, an A-Frame solution was necessary since using Unity could cost a lot of time with other needed features.

Some A-Frame components were discussed, however, most of them suffered from the main issue of not being able to zoom as far as needed, or were not updated with the current A-Frame version. A notable example is an A-Frame component made with Tangram, an OpenGL map integration from Mapzen whose main feature is displaying open-source map tiles [53].

Even so, a solution was found with the VR Map component, from professor Robert Kaiser [54]. This integration used OpenStreetMap data via Mapnik, a collection of tiles representing the world in twenty different zoom levels [55]. The last zoom level contains millions of tiles, each with 50m². Besides showing the roads, this component was also intended to demonstrate buildings and trees in the form of polygons. All it needs to load the location are the longitude and latitude coordinates. Figure 2.11 illustrates the location of the installations of Altran in Gaia, using this component.

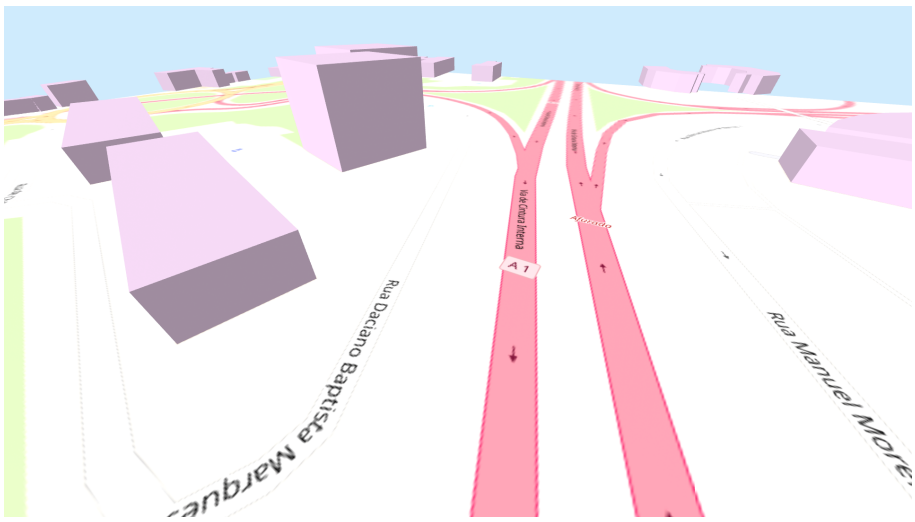

Figure 2.11: VR Map

The main issue regarding this application is the fact that the entire map is loaded at the start instead of when moved. This can impact the memory of the application in case the sequences are too long.

2.3.2 User Interface

There are various VR frameworks and libraries available and the user interface for an A-Frame application can be divided into two groups: inside or outside the A-Frame scene. If the UI is outside the scene, it can work as any HTML interface available, functioning as a 2D interface. However, there was a need to toggle between entering and leaving the scene when wanting to use it. And because of this reason, Kratos VR should maintain its menu inside the scene.

After this, when discussing design ideas at the start of the development of the application, all the GUI gadgets and features that had to be implemented were arranged. This included buttons, checkboxes, drop-downs, and sliders, as well as the possibility to customize these gadgets. The capability of supporting VR input was crucial.

After searching for different libraries or components, some incredible and creative ideas were found. However, one common aspect that each of these libraries had was the fact that they were not updated since 2018 or earlier. This means that besides creating the menu, the component had to be updated if implemented in the project. Regarding the different GUI gadgets, all of the studied libraries had support for buttons and checkboxes.

Table 2.5 provides an answer to the questions that were wondered when analyzing each GUI library if they were updated and customizable and had support for other features and GUI gadgets.

Component / Library	Updated	Customizable	Drop-down	Slider	VR Input
A-Frame Collection	No	Yes	No	Yes	Yes
A-Frame GUI	No	Yes	No	Yes	No
A-Frame Material	No	Yes	No	Yes	Yes
A-Frame State & Proxy Event	Yes	Yes	Yes	No	Yes
Dat.GUI VR	No	No	Yes	Yes	Yes

Table 2.5: A-Frame menu libraries capabilities

After some careful consideration, the best options chosen were Dat.GUI VR and A-Frame state components in conjunction with the A-Frame proxy event component. Dat.GUI VR was first created for three.js but had a similar version for A-Frame [56]. Although it has not been updated in three years, it has the necessary GUI gadgets for the application and great documentation available [57].

The other discussed option was using the A-Frame state component to create different menu states, and use the proxy-event component to manage the buttons. With this, it is possible to create checkboxes by changing the color of the button text, and drop-downs with arrows that behave like buttons. Figure 2.12 illustrates a great example created by Supermedium developer and A-Frame co-creator Kevin Ngo [58].

Figure 2.12: Interface example by Supermedium

2.3.3 Graphs in VR

The usage of graphs has become one of the most proficient and suitable ways for visualizing data. In virtual reality, this is not different. Numerous VR applications display data through the use of 3D graphs, like scatter plots or histograms [59]. Some studies have already been made on the usability of VR graphs instead of 2D graphs, and although not inherently intuitive, more information can be gathered with a VR aspect [60]. Figure 2.13 shows a 3D Scatter Plot, one of the most common graphs displayed in VR.

For Kratos VR, on the other hand, there is a need to display graphs for the velocity and acceleration of the main vehicle during the animation of the sequence. The ego file contains information about the linear and angular velocity, as well as linear acceleration (This file is better specified in section 2.1.2.2). This information should be made available to the user, while the animation of the sequence is running. With this in mind, the objective was creating a 2D data plot chart inside the VR scene.

One A-Frame component that drew close attention called A-Framedc, defined different types of charts in 3D. One of its key features is that it can represent a 2D bar chart in 3D [61]. However, some factors can reduce performance and deteriorate the graph both visually and functionally. HMDs have very limited screen space and combining the fact that this component was not updated since 2017, and that the graphs do not show the complete labels and can not be modified, the already established 2D graph applications were preferable to look into.

Chart.js provides a simple and flexible way to design 2D bar charts. Being in constant development, this open-source project contains great documentation and one of its main functionalities is easily converting the graph to images in base64. Besides this, the graphs can be fully customized and it is possible to add and remove data to the graph when needed. The biggest disadvantage of using this is loading images in every frame, which can significantly reduce the performance of the

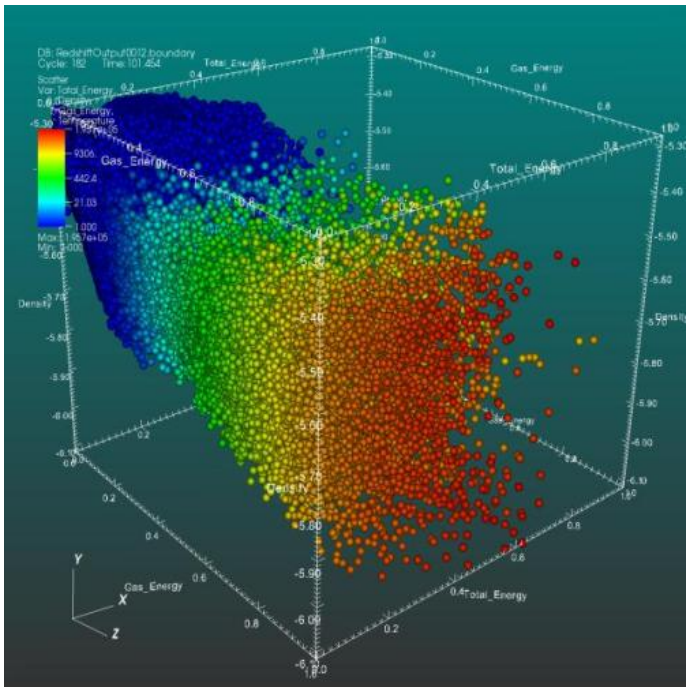

Figure 2.13: 3D Scatter Plot

application [62].

2.3.4 Draco Encoding Library

When dealing with point clouds, the application needs to be prepared to allow data that can contain a huge amount of storage demand. As such, encoding tools like Draco (created by Google) can help in reducing unnecessary space [63]. With this in mind, Kratos VR requires tools like Draco, and assuring A-Frame can provide them is paramount.

Google created this tool to compress and decompress point clouds or other meshes, intending to improve the storage of 3D graphics. Draco provides ten alternative compression levels by using the Edgebreaker algorithm [64] [65]. ”At each stage, the input mesh is divided into disjointed regions that may share a vertex but no edges. The edges bounding each region constitute a polygonal curve which is called a “loop”. The edges of the loop are called “gates” with one gate being active at each step. At every step, there is a triangle incident to the active gate that is not yet visited” [66].

Some of the main advantages of Draco include:

- Compression of common attributes like classification values.
- Support for further lossy compression by using between 1 and 31 quantization bits.
- Use of a KD-tree based encoder, Draco can rearrange points for optimal compression.
- Use of parallel decoding and GPU dequantization to get better performance [67].

For A-Frame, there were hardly any Draco compression tools available, however, there was an updated project that combined Draco with A-Frame entities. This was done by creating a JS function that registered a Draco model entity into the A-Frame scene. By providing the URL where the Draco file is stored, it is later decompressed and attached to the entity.

Curious features are the possibility to change the color of the point clouds, as well as its material. However it still only works with older A-Frame versions and needs to be updated [68]. Figure 2.14 demonstrates the differences between different point cloud meshes of a bunny using this component. The left point cloud has no material, while the mesh on the right has a material with an orange color.

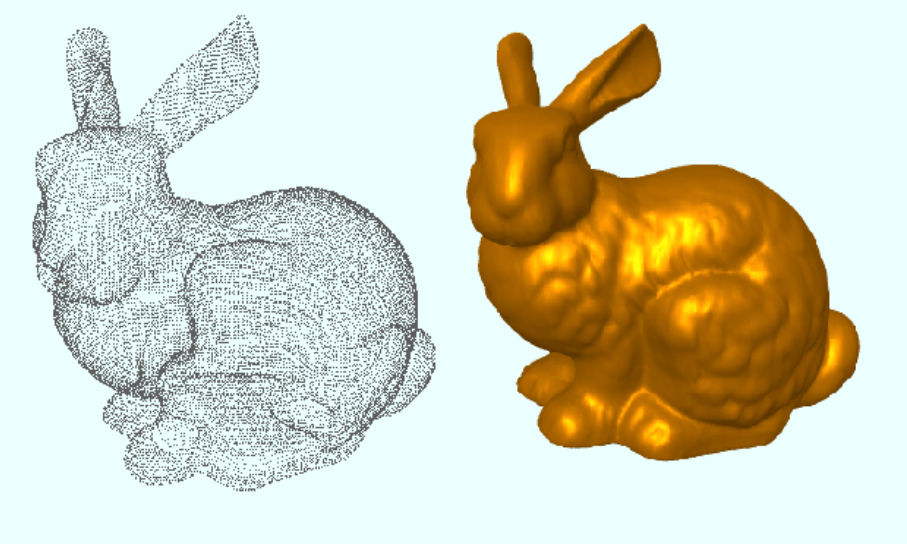

Figure 2.14: Draco different examples in A-Frame

There are already projects that have made significant progress concerning point clouds. PCL (Point Cloud Library), presents an advanced and considerable approach to the subject of 3D perception, providing a basic tutorial on how to implement PCLs in different software applications. PCL is fully integrated with ROS and has been already used in a variety of projects in the robotics community [69].

2.4 Related Work

Besides the already mentioned WebGL application, created in partnership with ISEP [9], there are significant studies in this area that deserve an acknowledgment. Most of these projects have inspired Kratos VR with their advancements, to cultivate different practices when addressing some concepts and perfect the application.

The study [70] provides a detailed comparison of the performances between WebGL and WebVR in VR environments. It showed that WebGL and WebVR have similar performances when executing on a PC and that WebVR performs way better when executing on mobile devices. When

executing on a computer, the following factors were considered: CPU, GPU, RAM consumption, and Frames per second (FPS).

More noteworthy to mention concerning WebVR, are the studies in [25] and [71] which introduce to WebVR and its frameworks and provides an evaluation of the VR usage in web browsers. It also provides an introduction on how to set up your web VR application.

When comparing Kratos VR, some recent developments already test autonomous driving in virtual reality. The study in [72] introduces a simulation environment for both virtual and augmented reality, that integrates an autonomous vehicle inside a virtual environment. However, it is mostly designed to test how the vehicle will handle real obstacles and traffic. An image of the application created by rFpro for this purpose is shown in Figure 2.15.

Figure 2.15: rFpro VR Workstation

The paper in [73] provides an analysis of the studies made in virtual reality and autonomous driving safety. This paper concludes that there is a lack of research on this field when VR is applied in a machine-centered approach.

Chapter 3

Visualization of the Data

This chapter provides a detailed analysis of the visual component of Kratos VR, with a meticulous discussion of the proposed and chosen solutions, failed attempts, and its complete development. The architecture for this project can be seen in Figure 3.1. Since the Kratos Visualization's infrastructure was reused, some functions of the application were reused in Kratos VR. This was the case for the way in which the data is loaded into the application and a small part of the main code for the functionalities aspect. Despite both the applications using Draco, this method had some slight differences in VR, since the specified component for A-Frame was not updated. This is better detailed in section 3.5. Both applications were developed and tested using Node.js, an open-source JavaScript runtime environment that executes JavaScript code outside a web browser. The main Node.js file is the same used both applications. The other elements in the architecture were created and designed specifically for Kratos VR.

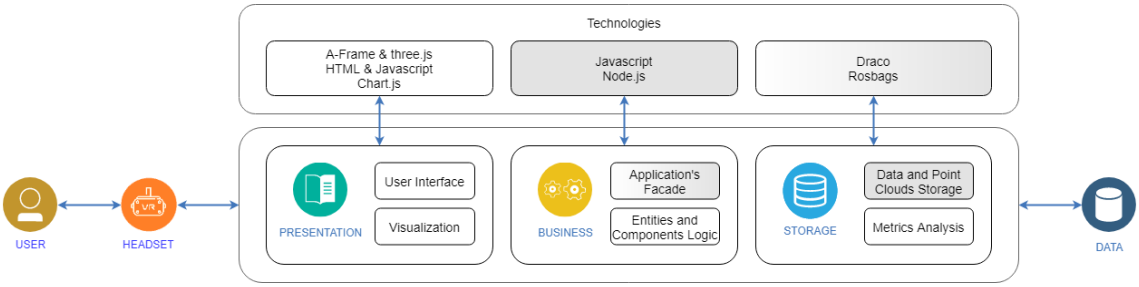

Figure 3.1: The architecture of Kratos VR

The end result of the project will consist of a web application capable of being accessed through any web browser. Despite being made for VR devices, it is possible to access the application through any equipment that supports the necessary requirements. When the user loads the data through the application, the server will retrieve it from the server and then pass it to the client.

3.1 Component Selection

Regarding the technologies used during the development of the application, various significant approaches were considered. The two main and more stable options were creating a web browser platform with WebXR or developing an application with the game engine Unity. While choosing WebXR gave the advantage of starting with an already built-in infrastructure, the latter had been a reliable platform for creating VR experiences a lot longer. WebXR is still in its early stages, and as such, the decision was impacted by the necessary components that would have to be used in the project. Table 3.1 displays the components that already had acceptable implementations in each platform.

Component	WebGL	WebXR	Unity
Draco	Yes	No	No
Real World Map	Yes	Yes	Yes
Easily customizable GUI	Yes	No	Yes
Trajectories	Yes	No	Yes
Text Boxes	Yes	No	No
Graph Plots	Yes	Yes	Yes

Table 3.1: Necessary components

Although Unity had more components implemented and a more understandable and stable interface, the key factor for this decision was the use of Draco and the rebuilding of the infrastructure. Unity did not provide an acceptable Draco conversion system and because of that, adapting the Draco component from Kratos Visualization to WebXR was easier. Besides this, a lot more time would be required to develop the infrastructure since using WebXR would provide a head start by reusing some files and functions from the Kratos Visualization application. As for the framework, the three.js alternative, A-Frame, was chosen and with that, the development process began.

3.2 Scene Structure

Despite most of the files in the application being in JavaScript format, the main HTML file had to encompass all the necessary elements inside the *a-scene* tag. Dividing the HTML file into other small ones would be more coherent and would respect the coding practices, however, there was no other way to keep the entities in the same scene tag if that happened. This included assets (images, fonts, and UI elements templates), all the different menu states, the camera, the elements, the point clouds, the data plots, the sky color, and the map. Considering the enormous amount of code in this file, Listing 3.1 displays a more comprehensible simplified version.

```
1  <a-scene>
2      <!-- Assets -->
3      <a-assets>
4          <!-- Images -->
```

```
5           <img id="pageIconImg">
6           <img id="logoImg">
7
8           <!-- Base templates -->
9           <a-mixin id="fontSnippet"></a-mixin>
10          <a-mixin id="fontRoboto"></a-mixin>
11
12          <!-- Menu and button templates -->
13          <a-mixin id="buttonBackground"></a-mixin>
14          <a-mixin id="buttonHoverEffect"></a-mixin>
15          <a-mixin id="buttonText"></a-mixin>
16          <a-mixin id="button"></a-mixin>
17      </a-assets>
18
19      <!-- Menu container -->
20      <a-entity id="menu" position="0 1.6 -4" rotation="20 -45 0">
21          <a-entity id="menuBackground"></a-entity>
22          <a-entity id="menuBack"></a-entity>
23
24          <!-- Main menu -->
25          <a-entity id="mainMenu" bind__visible="menu === 'main'">
26              <a-entity id="appName"></a-entity>
27              <a-entity id="title"></a-entity>
28              <a-entity class="buttons"></a-entity>
29          </a-entity>
30      </a-entity>
31
32      <a-sky color="#ECECEC"></a-sky>
33
34      <!-- Camera -->
35      <a-entity camera-listener>
36
37          <!-- VR Controls -->
38          <a-entity id="leftHand" laser-controls="hand: left"></a-entity>
39          <a-entity id="rightHand" laser-controls="hand: right""></a-entity>
40
41          <a-camera>
42              <!-- Elements inside the view of the camera -->
43              <!--<a-cursor></a-cursor>-->
44              <!-- Screen Test -->
45              <a-text id="frameNumber"></a-text>
46              <a-text id="loadingSequence""></a-text>
47              <a-text id="figureText"></a-text>
48              <a-text id="coordText"></a-text>
49
50              <!-- Data Plots and Cam Feeds -->
51              <div id="dataPlotCanvas">
52                  <canvas id="dataPlot"></canvas>
53              </div>
```

```
54              <a-image id="camFeedImg"></a-image>
55              <a-image id="dataPlotImg"></a-image>
56          </a-camera>
57
58      </a-entity>
59
60      <!-- Elements -->
61      <a-entity id="elements"> </a-entity>
62      <a-entity id="trajectory"> </a-entity>
63
64      <!-- Point clouds -->
65      <a-entity id="pointClouds" rotation="-90 0 0"></a-entity>
66
67      <!-- Map -->
68      <a-entity id="ground"></a-entity>
69      <a-entity id="map">
70          <a-entity id="tiles"></a-entity>
71          <a-entity id="items"></a-entity>
72      </a-entity>
73
74  </a-scene>
```

Listing 3.1: Main HTML file

3.3 Real World Map Addition

Regarding the map used in the application, there was a need to include the real-world street names of any geographic location. Using the VR Map component described in Section 2.3.1, the map of a certain sequence was loaded, by using the latitude and longitude coordinates inside the ego file in the data folder.

The buildings and the trees were discarded since it would require a huge amount of memory and space. Originally when loading the application, 49 (7x7) tiles were loaded, which amounts to 350m². Since some of the sequences were larger, the number of tiles was increased to 121 (11x11). This means that a map of 550m² is loaded at the start. An example of a single tile is seen in Figure 3.2.

Listing 3.2 shows the loading function of the tiles, where the variable *tilesFromCenter* represents the number of sections around the main tile. The function *addTile* includes a tile in the A-Frame scene, as an entity with tag <*a-plane*>. These tiles contain the image of the map section retrieved from one server of the Mapnik open-source tool.

```
1  function loadGroundTiles() {
2    for (let x = 0; x < (tilesFromCenter + 1); x++) {
3      for (let y = 0; y < (tilesFromCenter + 1); y++) {
4        addTile(x, y);
5        if (x > 0)
6          addTile(-x, y);
```

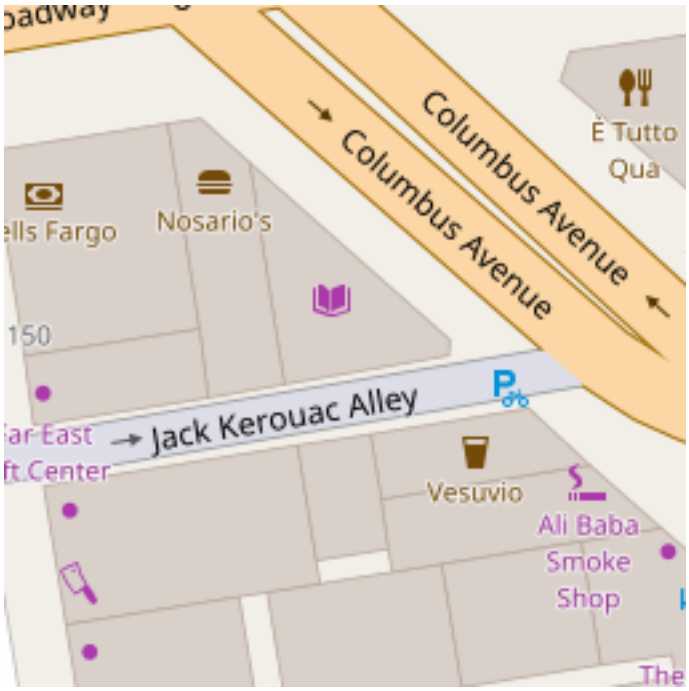

Figure 3.2: A 50m² tile of Jack Kerouac Alley next to Columbus Avenue in San Francisco

```
7         if (y > 0)
8           addTile(x, -y);
9         if (x > 0 && y > 0)
10          addTile(-x, -y);
11      }
12    }
13  }
```

Listing 3.2: A-Frame component registration example

3.4 Elements Implementation

One of the most important aspects of Kratos VR is the careful and detailed way that the elements are displayed. These were divided into three groups: the main vehicle or ego, the categories of vehicles specified in the configuration file of the sequence (Listing 2.2), and the trajectory. Each of these elements, except the trajectory, contains the point structure of the polygon in their own file (Listing 2.3). It is also important that some info would be displayed on the screen when the cursor of the controller intersects with either the ego or the other vehicles.

On top of this, when the sequence is loaded, all the elements are visible and without their wireframe model. Their color and opacity have default values, however, all of these features can be customized in the personalization menu, best detailed in section 4.2.4.

3.4.1 Addition of Polygons

The addition of polygons was accomplished with the class *Object3D* from three.js, mentioned in section 2.2.2.3. For each element that appears in the sequence, an entity had to be created and attached to a unique 3D object. For this to happen, a component needs to be registered.

Each element contains an id provided by its JSON file. This id was also reused for the component registration since each component present in the A-Frame scene needs a different name. Other three.js features were used for the mesh and geometry of the figure. The functionalities of three.js definitely facilitated this development. The main function for the creation of these polygons can be seen in Listing 3.3.

```
1  /**
2   * @brief Creates the HTML element and draws the figure
3   * @param element - The JSON information of the element (id, polygon, height...
        )
4   * @param name - The category of the figure as a string (Ex: car, ego, van...)
5   * @param isEgo - If the object is the ego
6   */
7   function createFigure(element, name, isEgo) {
8      // Main HTML element
9      let elContainer = document.getElementById("elements");
10     // Creates a new HTML element as an A-Frame entity
11     let figure = document.createElement('a-entity');
12     // Color of the Element
13     let color = objAttributes[name][0][1];
14     // Adds info on the screen when the cursor intersects with it
15     createTextBoxes(figure, name, color, element.polygon);
16     // Creates the component and adds an unique id
17     let figureID = name + element.id;
18     // Attaches the component to the entity
19     figure.setAttribute(figureID, '');
20     // Adds the figure to the elements container
21     elContainer.appendChild(figure);
22     // Creates variables for the mesh of the figure
23     let shape, extrudedMesh, geometry;
24
25     // Registration the A-Frame component
26     AFRAME.registerComponent(figureID, {
27         init: function () { // Function called once when the figure is loaded
28             // Calculates the position of the element
29             egoPosition = calculatePosition(element, isEgo);
30             // Sets the previous calculated position
31             this.el.setAttribute("position", egoPosition);
32
33             // Creates a vector with the points from the JSON file of the
                   element
34             let points = [];
35             for (let i = 0; i < points.length; i++)
```

```
36                  points.push(new THREE.Vector3(element.polygon[i][0],
                        element.polygon[i][1], element.polygon[i][2]));
37              }
38              shape = new THREE.Shape(points); // Adds the points to the shape
39          },
40          update() { // Function called whenever the element is updated
41              let isVisible = objAttributes[name][3]; // If the figure is visible
42              if (isVisible) {
43                  // Removes a mesh in case there was any
44                  this.el.object3D.remove(extrudedMesh);
45                  // Creates the new mesh and the geometry of the figure
46                  createFigureMesh(element, this.el, name, color, shape, isEgo,
                        extrudedMesh, geometry);
47              } else { // If the figure was not visible, only removes the mesh
48                  this.el.object3D.remove(extrudedMesh);
49      }}});
50  }
```

Listing 3.3: Registration of the 3D objects

In figure 3.3 one can see the different visibility types. The polygon of the orange van on the top left is visible and without its wireframe model. The polygon of the pedestrian on the far right has its wireframe model activated, while the polygon of the cyclist, in the middle, is not visible. The user can change this in the interface of the menu, and the client can easily add more types of visibility if needed. This is better discussed in section 4.2.4.

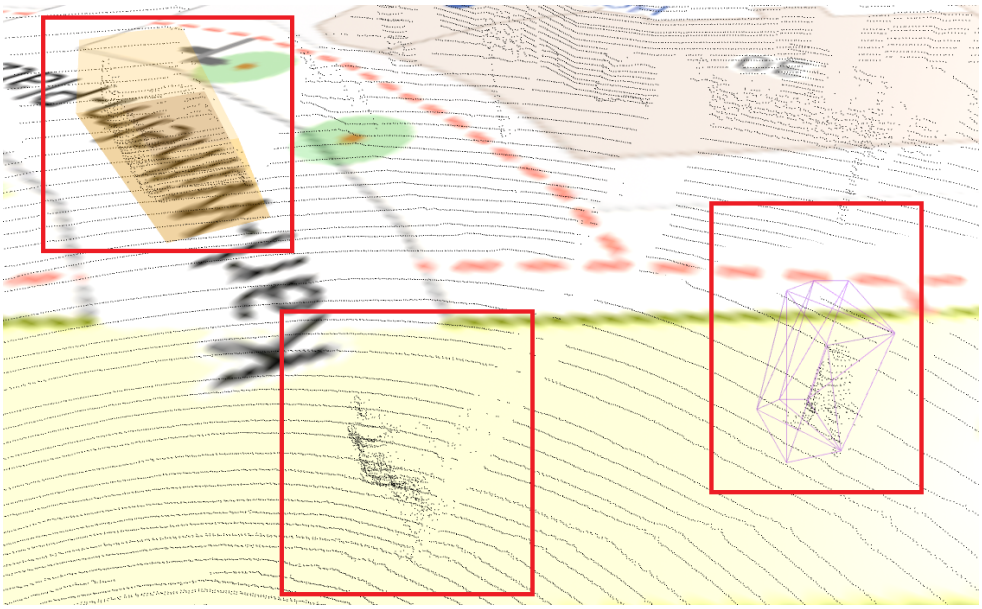

Figure 3.3: Types of polygons visibility. Top left: an orange polygon with default opacity for a van; center: polygon for a cyclist is invisible; right: purple polygon for a pedestrian has a wireframe

3.4.2 Map Positioning

When implementing the position of the elements, several issues that delayed the development of this part were encountered. The main cause for this was the fact that in Kratos Visualization, the ego (main vehicle) and the elements are built on top of layers. Deck.gl is well prepared for layer management. The center is given in a certain latitude and longitude with the map, abstracting the details of the translations of the elements. As such, as the animation continued, the layers were being updated with the ego always in that specified center. These layers included all the elements, the point clouds, and the map. Since the map chosen for Kratos VR was based on tiles acquired from an external server, loading the map every time when a frame changed was going to create memory exhaustion problems.

To solve this problem, the elements and the point clouds had to be moved to their respective positions. Since the ego only contains information about its geographical coordinates, a conversion to the Cartesian system was necessary. As such, a study about this area had to be made. With this in mind, a first iteration of the solution was using the following formulas, where ϕ represents the latitude, ψ the longitude and r the radius of the Earth [74].

$$x = r * \cos(\phi) * \cos(\psi) \tag{3.1}$$

$$y = r * \cos(\phi) * \sin(\psi) \tag{3.2}$$

$$z = r * \sin(\phi) \tag{3.3}$$

However these gave incorrect results, and because this formula is assuming the Earth is a sphere, while the World Geodetic System (WGS) assumes it as an ellipsoid, in their WGS-84 revision [75]. With this, another proposed solution was using the Haversine formula, in equation 3.4.

$$d = 2r\sin^{-1}\sqrt{\sin^2(\frac{\phi_2 - \phi_1}{2}) + \cos(\phi_1)\cos(\phi_2)\sin^{-1}(\frac{\psi_2 - \psi_1}{2})} \tag{3.4}$$

This formula is very important in navigation, used to calculate the distance (d) between two points on the Earth's surface specified in longitude and latitude. When converting to Cartesian coordinates, it uses the Spherical Trigonometry formula instead of a Law of Cosines based approach which was initially intended for 2D trigonometry, therefore having a nice balance of accuracy over complexity [76].

To comprehend the formula used for the position of the ego, the concept of bearing needs to be understood. Bearing is the horizontal angle between the direction of an object and another object, or between it and that of true north. Figure 3.4 illustrates the bearing between Cape Town, South Africa, and Melbourne, Australia, which is the shortest route [77].

At last, a JavaScript function was devised, with the help of turf.js, a powerful library for advanced GIS (Geographic Information Systems) analysis. This library helped determine the bearing

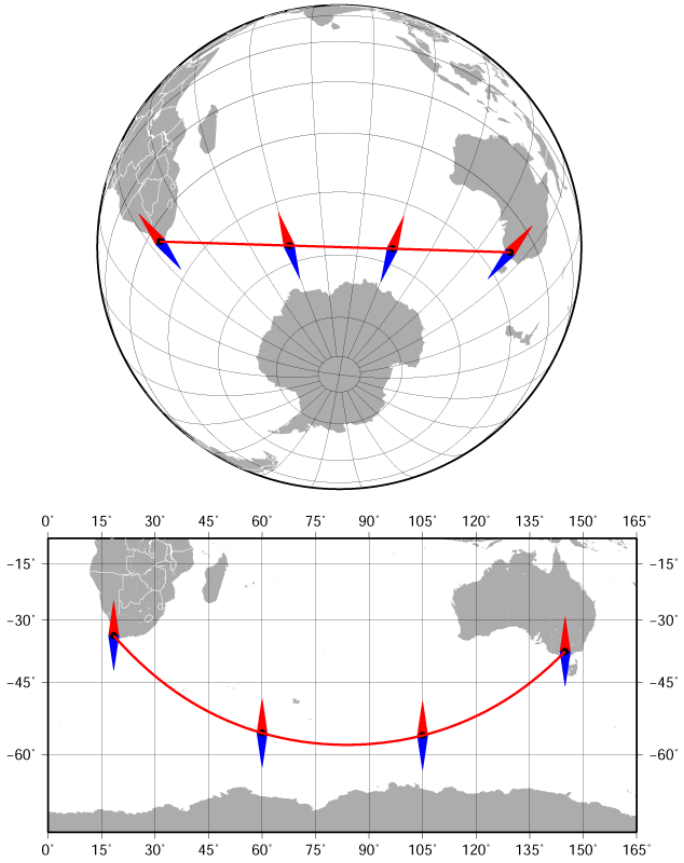

Figure 3.4: Bearing between Cape Town and Melbourne: top: Earth in 3D; bottom: Earth in 2D

for the latitude and longitude coordinates of the ego in the next frame. Since every other element besides the ego is positioned in a way that the ego is point (0,0,0), only the next position of the ego needs to be calculated, and then the element needs to be translated. The bearing is the angle measured in degrees from the north line (0 degrees).

Therefore, the function in listing 3.4 was defined, where the bearing and the distance between the geographic coordinates in the current frame and the coordinates in the next frame were calculated. The common trigonometric functions can then be applied and the Cartesian coordinates of the next position of the ego are acquired.

For small size sequences, the ego position still maintains its precision, however with large size sequences, there were some notable deviations [78].

```
1 function getCoordsOfNextPoint( point1Frame, point2Frame ) {
2     let bearing = turf.bearing(point1Frame, point2Frame);
3     // Distance between the two points (In km)
4     let distance = turf.distance(point1Frame, point2Frame, {units: 'kilometers'
          });
5     return { x: distance * 1000 * Math.cos(bearing),
6              y: distance * 1000 * Math.sin(bearing) }
7 }
```

Listing 3.4: Function to calculate next ego position

3.4.3 Trajectory of the Ego

Besides the elements of the scene, there was a need to give the option to visualize the path of the ego to the user. One discussed option on how to implement this in A-Frame was the use of spline interpolation. This method a smooth line within the range of a discrete set of known data points. While line interpolation only connects each dot, a spline uses low-degree polynomials such that they fit smoothly together [79]. An example of this can be seen in a data plot in Figure 3.5.

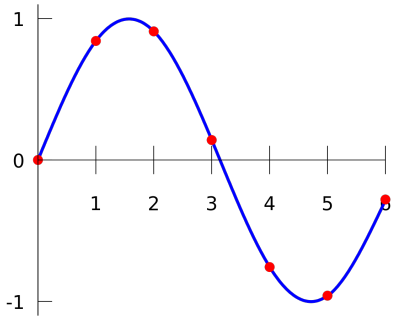

Figure 3.5: Spline interpolation

When searching on how to implement splines in A-Frame, there were not any available components to do this. One first approach was uniting each of the possible coordinates of the ego and afterward, creating a line interpolation. However, after doing this, progressing to a spline interpolation would not make much difference since the distance between each point was very small. If implementing it, this would cause some performance issues since it had to calculate the path for each point.

Nevertheless, another approach by using a dashed line (composed of rectangles) was devised. This allowed the creation of a rectangle of the line once in every ten coordinate points of the ego. This was done by using midpoints between each of the polygon points of the ego. Using this, a rectangle of the line was created every 10 steps. Figure 3.6 illustrates this method, from an above perspective. Both the yellow and orange rectangles represent a single rectangle of the line, created by using the position of the vehicle.

The current position of the vehicle is represented in green and its position after 10 frames in red. The dots represent the coordinates of the polygon and the midpoints. To increase the frame rate of the animation, the trajectory is created while the sequence is loading. Figure 3.7 displays the final result in the application.

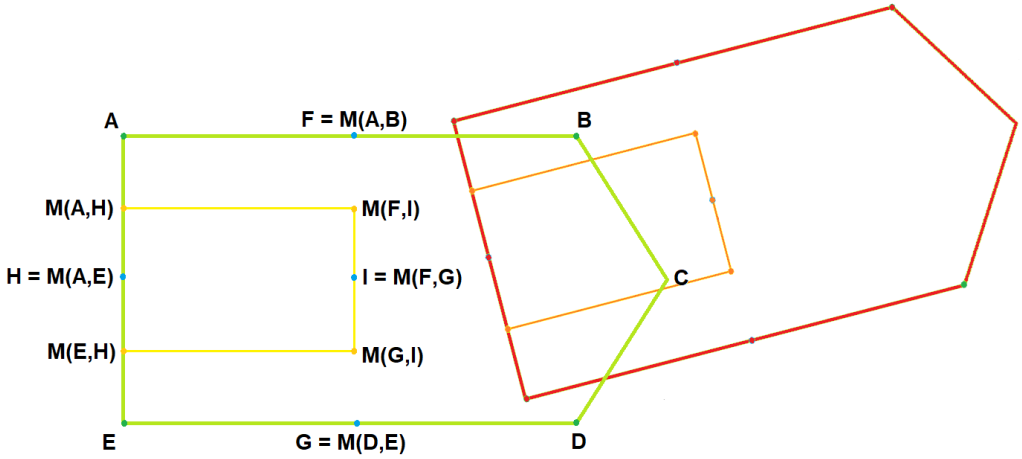

Figure 3.6: Ego (in green) and trajectory line (in yellow) in frame x and frame x+10 (Ego in red and trajectory line in orange), seen from an above perspective

3.4.4 Raycaster Intersection

When creating a data visualization application, the user must be able to collect information while visualizing the scenario. One idea, also established in Kratos Visualization, was providing information to the user whenever pointing at an element. This needed to happen when he points the controller to this element, intersecting with the 3D polygon of the object. To understand how to do this, there was a need to use Raycasting.

Raycasting is a set of techniques used in many domains of computer graphics that focus on the intersection of "rays" casted from specific points into the scene. Used in immersive environments for selection of targets at the distance, it is widely used in 3D environments. In VR it can take a form of a ray cast from a hand-held controller to select targets. This means that when the user points, with a mouse or a controller, to a certain object, this object will be selected and interacted with [80].

The component Raycaster from A-Frame, already provides a solution to this issue, by creating a listener whenever the element intersects with the cursor. This raycaster was added to select the menu buttons and intersect with the elements [81]. Figure 3.8 illustrates the raycaster used in Kratos VR with the controller on the bottom of the screenshot.

Whenever the raycaster aligns with the 3D object of the element, the listener in Listing 3.5 handles the info that appears on the screen. Normally, most of the HMDs have small screens, which means that showing the elements, menu, data plots, cam feeds and this information at the same time, could produce size issues. This is why, at the moment, the only information being displayed are the name and current position of the element.

```
1  // Adds info and color when the cursor touches the figure
```

Figure 3.7: Ego (represented in green) and its trajectory (in blue) in Kratos VR

```
element.addEventListener('raycaster-intersected', function (evt) {
    element = evt.target;
    if(element.id !== previousTextBoxTarget) {
        figureText.setAttribute('value', 'Figure: ' + name);
        coordText.setAttribute('value', elementPos);
    }
});
```

Listing 3.5: Raycaster Intersection

3.5 Point Clouds Visualization

During the planning of the development of this application, when developing the point clouds, multiple issues already were predicted to occur. The only known A-Frame component for Draco (detailed in section 2.3.4) had not been updated for more than three years. Since there were no other feasible alternatives, that component had to be improved and revised.

As in Kratos Visualization, all the point clouds are stored into a variable when loading a certain sequence. This guarantees that the application does not suffer from latency and time delays when running the animation. Because of issues that arise when changing the A-frame version some compatibility problems started to appear with three.js and Draco.

The middle of the mesh of the point cloud is empty since it represents the ego vehicle. This is why the point clouds always stay in the same position as the ego. An illustration of this can be seen in Figure 3.9. Although this should happen, after loading all the elements and point clouds, and running the animation, the point clouds were not in their supposed position. This position deviation problem was one of the main and ongoing issues during the development process.

Most of these problems were solved after updating the point cloud component. However, the *DracoLoader* class, created by Google, still had some issues. The problem with *DracoLoader*

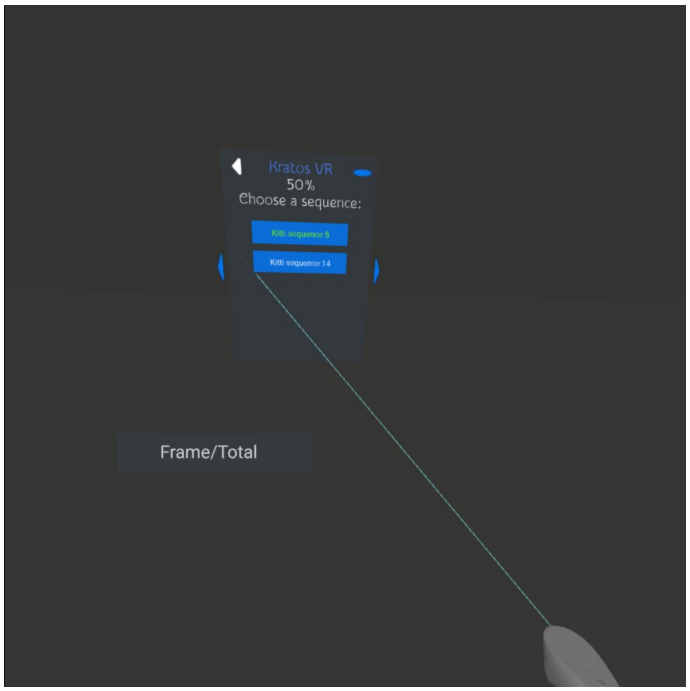

Figure 3.8: Raycaster used in Kratos VR

was that after acquiring the file location of the point cloud it created the entire process of decompressing and storing it all over again. Since this was already done for all the point clouds when the sequence was loaded, there was no need to do it again.

As such, a new function called *loadDraco* had to be created and placed inside the *DracoLoader* class. This function is displayed in Listing 3.6. The main objective of this function is skipping all the unnecessary steps and only collecting the information of the point cloud when needed. The process for the point clouds development is as follows:

1. When the user selects the sequence, the point clouds are loaded into memory and stored in compressed format;
2. Each of the point clouds is, at the same time, stored in compressed format by Draco;
3. When they are supposed to appear in the sequence, they are decoded again by Draco and displayed on the screen;
4. After the frame changes, the point cloud is encoded again.

The process is repeated for each of the following point clouds with steps three and four. This ensures that the application does not suffer from memory exhausting problems. After this was solved, the animation began to run smoothly.

```
1  loadDraco: function(callback) {
2          DRCLoader.getDecoderModule()
3              .then( function ( module ) {
```

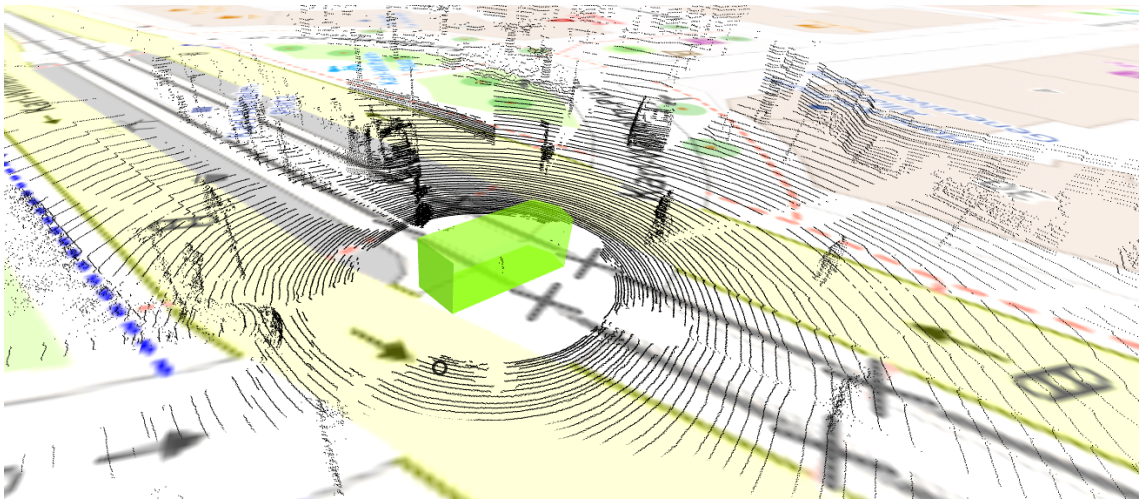

Figure 3.9: Example of a point cloud with the ego

```
4                 let dracoDecoder = module.decoder;
5                 // Here is how to use Draco Javascript decoder and get the
                      geometry
6                 let buffer = new dracoDecoder.DecoderBuffer();
7                 buffer.Init(new Int8Array(p_cloud1[current_frame]), p_cloud1[
                      current_frame].byteLength);
8                 let decoder = new dracoDecoder.Decoder();
9                 // Determine what type is this file: mesh or point cloud.
10                let geometryType = decoder.GetEncodedGeometryType(buffer);
11                callback(this.convertDracoGeometryTo3JS(dracoDecoder, decoder,
12                    geometryType, buffer, {}));
13            });
14    },
```

Listing 3.6: Function to load the Point clouds

Chapter 4

Interaction with the Application

While the previous chapter focused on the visual components of the application, this section deals with the interaction part of the developed application. One of the key factors of the success of an application is the fact that the user can interact with it smoothly and without issues. This is why the methods used for the interface development will be described and, additionally, the procedures for integrating any VR controller with the application will be detailed.

4.1 Interface Approaches

As briefly stated in section 2.3.2, there was some uncertainty on which method to use when developing the interface. This chapter will look into the two main approaches that were developed, and provide a detailed analysis for each of them.

4.1.1 Using Dat.GUI VR

One first approach was using Dat.GUI VR and follow its easily understandable API documentation. Since this library can allow the use of buttons, checkboxes, sliders, and dropdowns, the application would make the most of it. The menu that appeared when first loading the website using this library, can be seen in Figure 4.1.

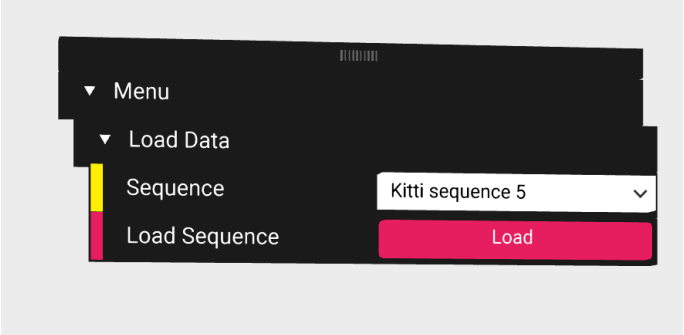

Figure 4.1: Main menu with Dat.GUI VR

However, some issues started to emerge during the development because it was not updated since 2017. A-Frame had a completely new version since then, and some of the main features of Dat.GUI VR were outdated. After updating the version, other problems had to be fixed in the library and some obsolete functions needed to be updated. However, there were still some problems that could not be solved. To supplement this, the Oculus controllers were not working as intended and some design issues, like the menu being too large, were starting to increase.

Since a sequence can have many categories of objects, and each of them can contain different personalization options, the interface could occupy a huge amount of space that the VR headset cannot handle. An example of this can be seen in Figure 4.2. Considering that figuring out a way to fix this was taking unnecessary time and assuming that further A-Frame versions would damage the library, even more, one solution was creating a different and more efficient interface.

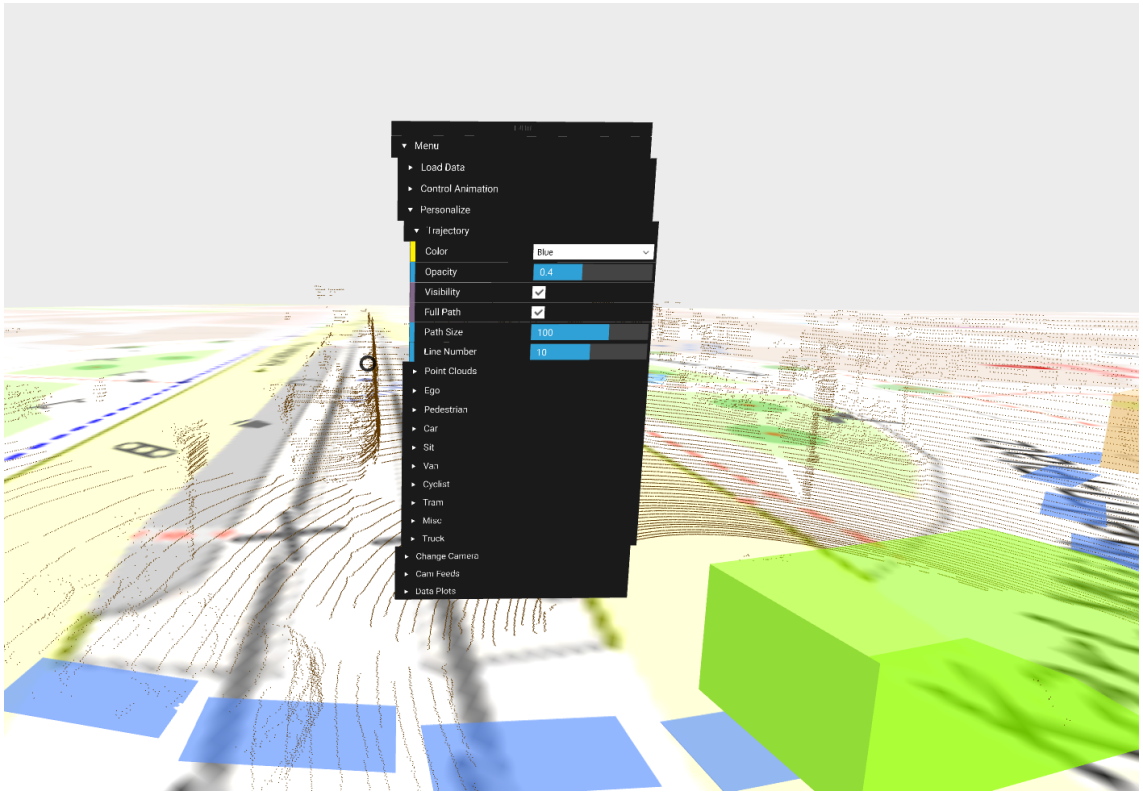

Figure 4.2: Dat.GUI VR problem: An example of how the menu can be incredible large

4.1.2 Combining A-Frame Components

The next decision was choosing a more stable method for the interface. With many components created specifically for A-Frame, one solution with superframe collection, from Supermedium was found [82]. This collection contained different individual components, each with a different purpose.

The combined use of the proxy-event and state components allowed the creation of a menu with different multi-purposed buttons. The state component defines different states for each of the menus of the application. The menus can be changed when clicking their respective buttons. By doing this, it is ensured that only one menu is in the scene at all times, and there cannot be any interference with other states. The *proxy-event* component captures the event when a button is pressed. By attaching a certain function to a button the state of the menu can be changed when that button is clicked.

One other important component that was implemented in the application is the *look-at* component. This component allowed the interface of the menu to always face the position of the camera. Since the user can change its perspective, it assures that the menu is always seen from the front. With this, the main groundwork for the interface is complete.

Listing 4.1 demonstrates an example for the HTML element of a single button. This button is only visible and raycastable (meaning it can be clicked) when the menu where this button is attached is the one appearing on the scene. When the button is pressed, the component proxy-event calls the targeted function.

```
1 <a-entity
2     text="value: Button; color: #FFFFFF" proxy-event="as: function"
3     bind-toggle__raycastable="menu === 'main'"
4     bind__visible="menu === 'main'">
5 </a-entity>
```

Listing 4.1: HTML Button Element

The use of arrows allowed the creation of a more efficient system than using dropdowns. These arrows can function as buttons and when pressed, the page of the current menu can be changed. Because of this, if the application contains multiple sequences, it can have multiple pages and determine how many sequences appear in each of them. These pages can be easily changed by alternating between the arrows.

Blue arrows were also added at each side of the menu, with the functionality of translating the position of the menu. This helps the user putting the menu in an acceptable position when visualizing the elements. There is also an option to minimize and maximize the menu so that the screen does not get crowded. With the possibility of also customizing the buttons and text appearance, size, and color, the user can select his preferred design options. Figure 4.3 demonstrates the first page of the category menu.

4.2 Menu Structure and Development

When loading the application the user can choose the sequence to visualize from the set of available sequences. After this, the menu will have the following buttons:

- Choose Sequence - Choose another sequence to visualize;
- Control Animation - Control the animation of the sequence;

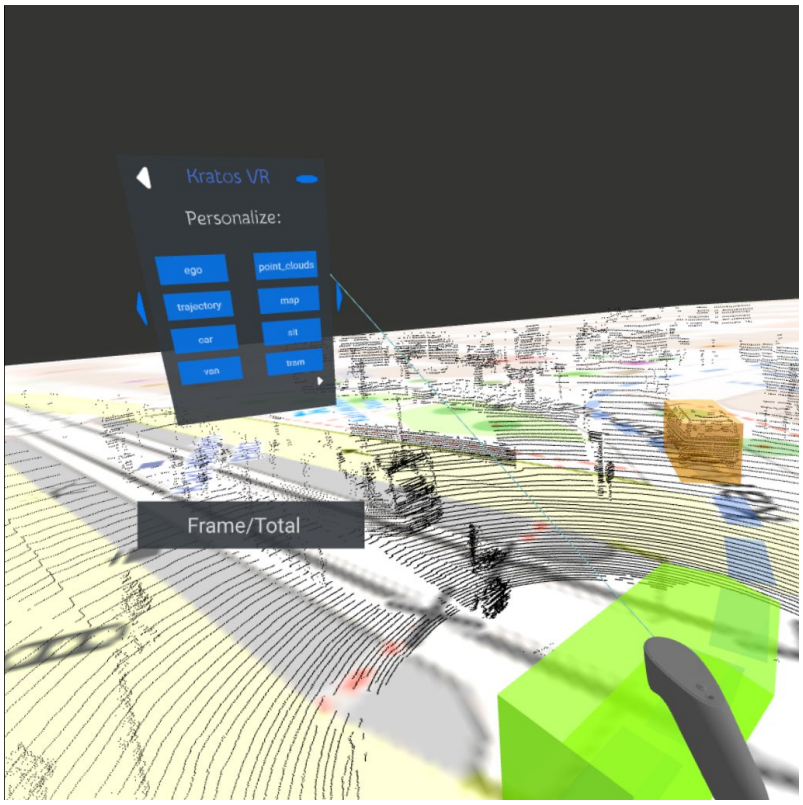

Figure 4.3: Category menu

- Personalize Elements - Change the color, opacity, visibility and other attributes of an element;
- Change Perspective - Change the perspective of the camera;
- Camera Feeds - View the available camera feeds of the sequence;
- Data Plots - Visualize and interact with the data plots of the sequence.

This section will explore the process of developing each of these interface items and examine the resulting approaches. For each of these items, the best procedure to integrate it inside the VR scene and how the user should interact with it will be discussed.

4.2.1 Animation Management

When first developing this menu state, there was a restriction that should be met: The user must be able to easily change to any frame of his choosing. With this in mind, the first obvious approach was using a slider bar. Unfortunately, there is not a good solution for this, and adapting any of the mentioned libraries in section 2.3.2 would take time and occupy unnecessary space.

As such, one solution was using the available components and create three types of arrow buttons. In Figure 4.4, one can see the resulting interface for this menu section. In a certain frame,

it is possible to increase or decrease this number by 1, 10, or even 50 frames. This ensures that the user only has to click a small number of buttons to travel to the frame he wants.

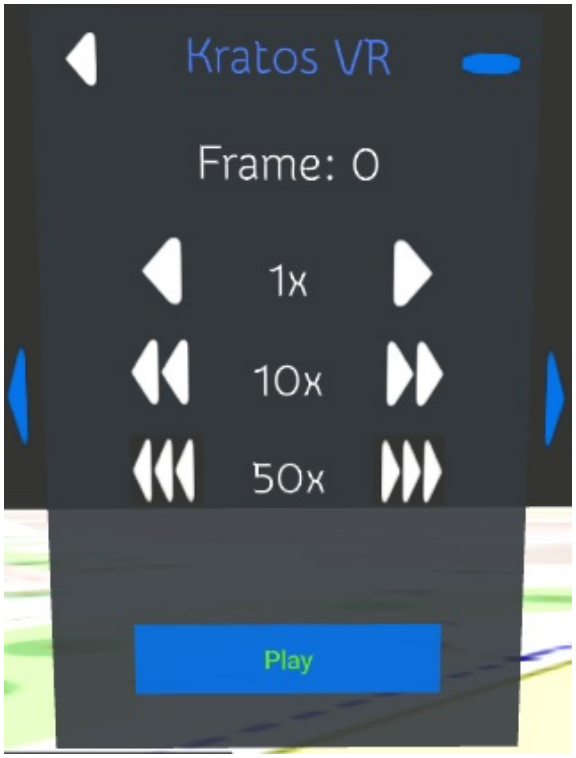

Figure 4.4: Control animation menu

Besides these features, this menu also contains the pause and play button. This allows the user to pause or play the animation during its visualization. On the other hand, it is also possible to pause or play the animation by using the VR controller. This is later discussed in section 4.3.

4.2.2 Perspective of the Camera

Another important aspect of the application was deciding how the user would visualize the information around him. While in Kratos Visualization selecting different perspectives was very limited, the opposite should happen inside a VR scenario. By allowing the user to be wherever he wants, he can collect the complete information he desires.

However, doing this inside a VR scene was very complicated, since as the camera moved, the other UI elements should follow the camera as well. This included the menu, the camera feeds, and the data plots. Since the camera feeds and data plots were only images, the most efficient solution was placing those images inside the element of the camera, and decrease their x position. On the other hand, the menu can be interacted with and should be in the viewpoint of the user at all times. As such, a component to make the menu follow the camera was devised, displayed in Listing 4.2.

```
1 AFRAME.registerComponent('camera-listener', {
```

```
    init: function() {
        cameraEl = document.getElementById("camera");
        camPos = cameraEl.getAttribute('position');
        camRot = cameraEl.getAttribute('rotation');
    },
    tick: function () {
        let newCamPos = cameraEl.getAttribute('position');
        if(!arePosEqual(camPos, newCamPos)) {
            camPos = newCamPos;
            menu.position.copy( camPos );
            // Translate the menu to their fixed position
            menu.translateY( 1 );
            menu.translateZ( -2 );
        }
    }
});
```

Listing 4.2: Function to make the menu follow the camera

The position of the camera changes frequently, while the animation is running. This is because the camera usually follows the position of the ego since the information appears around it. This component checks if the position of the camera has changed, with the tick function running at every second. In case this is true, the menu is also translated to his default position.

Concerning the interface of the menu, the user has various possibilities. He can change any parameter of the position of the camera (x, y, z), by increasing or decreasing its value. Besides this, there are already default values he can select, like positioning the camera inside the ego, or on top of it. This is illustrated in Figure 4.5.

4.2.3 Images Addition to the Scene

In the main menu, there are two buttons that display additional information: camera feeds and data plots. These allow the application to add images to the screen. The camera feeds show synchronized images taken from the cameras of the main vehicle. The user has the option to choose which camera feed he wants to see and alternate between them. These feeds are provided by the data present in the loaded sequence, which will correspond to any type of image that the data providers place there.

Besides this, the user can visualize data plots of some statistics about the sequence. Ranging from angular velocity to linear acceleration, it is possible to change the data plots parameters and alternate between them. Created with Chart.js, Listing 4.3 displays how the data plots are created.

```
function createDataPlot() {
    // Gets the data plot chosen by the user
    let chosenPlot = getPlotData(chosenDataPlot);
    // When loaded, the image is created
    let options = {
        legend: {labels: {fontColor: "white"}},
        scales: { yAxes: [{ticks: {fontColor: "white"}}],
```

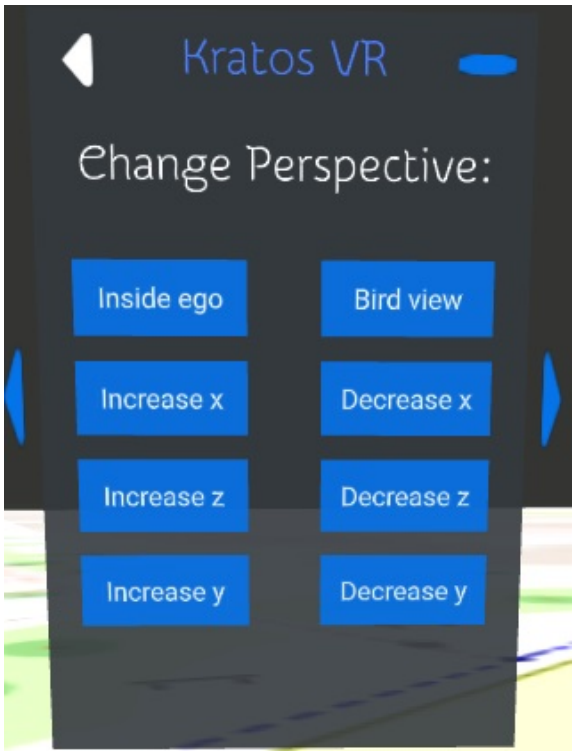

Figure 4.5: Menu of the Perspective of the Camera

```
                    xAxes: [{ticks: {fontColor: "white"}}] },
        bezierCurve : false,
        animation: {onComplete: done}
    };
    // Creates and adds the chart to the canvas
    let ctx = document.getElementById("dataPlot").getContext("2d");
    myLine = new Chart(ctx,{
        data: chosenPlot,
        responsive: true,
        maintainAspectRatio: false,
        type: "line",
        options: options
    });
}
```

Listing 4.3: Function to create data plots

Since the data plots were displayed in the form of images, a way to minimize the performance cost was necessary. These plots are always updated at each frame and can lose or gain new data. Because of this, a function that converted the data plot into images, using base64 encoding (an algorithm to represent binary data in an ASCII string format) was created.

Figure 4.6 displays the way these images are represented. They are placed on top of the screen, and the user can look at the sequence while collecting information about the camera feed images

and the data plots.

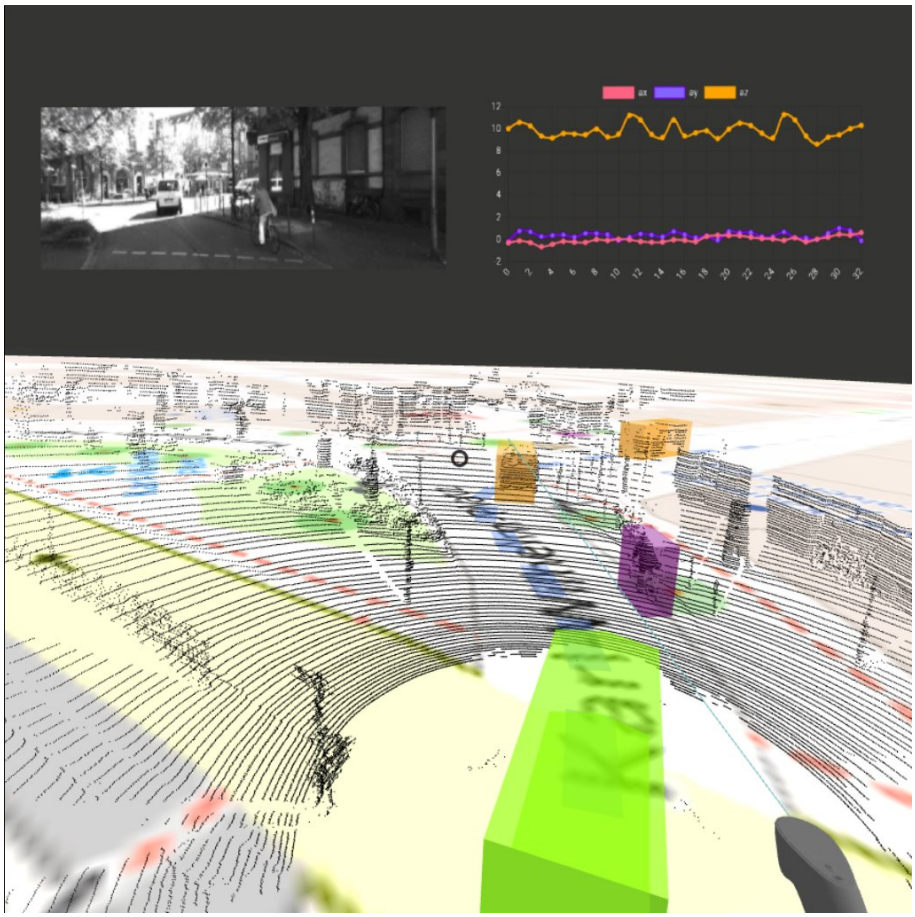

Figure 4.6: Camera feeds and data plots in the scene. Top left: camera feed image of the current sequence; top right: linear velocity of the main vehicle

4.2.4 Personalization of the Elements

When developing a visualization application, it is important that the user likes what he sees on the screen. Each user should be able to personalize the color, opacity, and filling of each element in the scenario so that the resulting color scheme fits their taste or visualization goals.

Personalizing elements is also important because of numerous reasons, like alternating between light and dark mode color scheme. Dark mode improves visual acuity, reduces visual fatigue, and improves usability when using HMDs [83]. In essence, Kratos VR needs to satisfy all tastes so anyone can enjoy interacting with the application.

In this menu, the user can change any visibility of an element, including the map, the point clouds, the ego, the trajectory, and any category of vehicles of the sequence. Regarding the last three, it is also possible to change the opacity and color of these elements, as well as adding or removing a wireframe model. While this is made through the use of buttons, a color wheel was integrated for the color replacement.

To do this, the A-Frame Colorwheel component was used [84]. However, like most A-Frame components, it was not updated and it suffered some changes to get adjusted to the application. This color wheel allows the user to choose any color by selecting it in the wheel. It is also possible to modify the selected brightness of the color, in a slider bar next to it. Figure 4.7 illustrates the color palette besides the menu when personalizing a van.

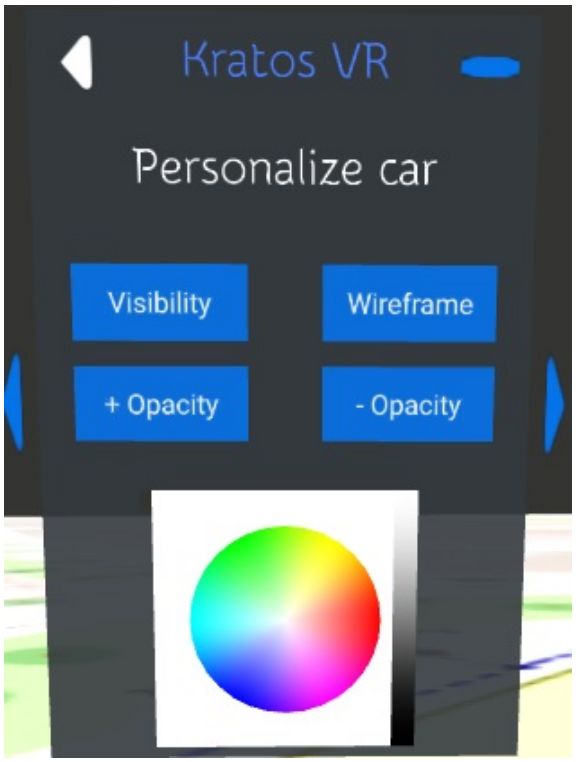

Figure 4.7: Color palette in personalization menu

Besides these features, the personalization menu for the trajectory contains supplementary buttons. These allow us to show the full or smaller path of the trajectory. The user can also change the line type of the trajectory, between dashed, scarce, or solid. With this easily built interface, the client can add more personalization options if he wishes to do so.

4.3 VR Input

The final key component to complete the interaction section is the VR controllers implementation. Not only it is necessary that the application allows the use of any VR controller in the market, but it also supports the usability of most of their possible controls. There are various types of controls that any VR controller can contain, ranging from a simple button to a joystick.

To allow this, an event listener needs to be created for each possible control. Each listener is composed of the input and one of its possible interaction options (starting or stopping to touch or press the control). However, for simplicity reasons, only one option for each control was accepted.

In table 4.1 the different types of controls are shown, as well as their chosen role and supported VR controllers. The controls that most VR devices support are the trigger and the touchpad. As such, both contain the most important roles: controlling the interface of the menu and pausing the animation, respectively. The rest of the roles were divided among the other controls, however, none of these needed and all of them can be changed in the menu.

Control	Role	Supported Controllers
Button (A)	Increase frame by 10	Oculus Touch
Button (B)	Increase frame by 1	Oculus Touch
Button (X)	Decrease frame by 10	Oculus Touch
Button (Y)	Decrease frame by 1	Oculus Touch
Grip (Right Hand)	Change the visibility of the data plots	Oculus Touch / Vive / Windows Motion
Grip (Left Hand)	Change the visibility of the camera feeds	Oculus Touch
Thumbstick	Play & Pause the animation	Oculus Touch / Windows Motion
Touchpad	Play & Pause the animation	Daydream / GearVR / Oculus Go / Vive / Windows Motion
Trigger	Click on the menu buttons	GearVR / Oculus Go / Oculus Touch / Vive / Windows Motion

Table 4.1: Types of VR input

Chapter 5

Results and Analysis

This chapter provides a complete analysis of some performance tests that were taken during the development of Kratos VR. These tests take into account the duration of the loading of the data into the application and the structure and visualization of both Kratos applications (Kratos Visualization and Kratos VR). The data analyzed consists of the elements, the point clouds, the trajectory, the data plots, and the camera feed images.

An experiment was created where the application was tested and reviewed. This was made with a small group of participants, some with VR experience. The participants also took an evaluation test for some motion sickness symptoms they may have when interacting with the application.

5.1 Performance

In data visualization applications, the loading times of all the different data must be analyzed. For this reason, several metrics were taken during the development of Kratos VR. To do this, a sequence from the Kitti Vision Benchmark Suite data set with 153 frames was used, whose information is represented in Table 5.1. The data plots are not included in this table, since they are created during the animation of the sequence. The HMD in which the application was tested is the Oculus Go, with 3GB of memory.

Data	Size (MB)	Other Information
Elements	0.156	1 Ego, 9 Cars, 3 Vans, 2 Pedestrians and 1 Cyclist
Camera Feeds	20.4	Resolution of 1392 x 512 pixels
Point Clouds	57.5	Compressed
Total	78.06	0.510 MB per frame

Table 5.1: Information about the tested sequence

This section will evaluate the loading time of the sequence and take into account the time it takes to load each data (elements, point clouds, camera feeds, and data plots) in each frame of the animation.

5.1.1 Sequence Loading Times

When loading a certain sequence, the average loading time has to be well known, to later be able to optimize the process. To do this, ten iterations were executed, as illustrated in Table 5.2.

Load Time (s)	34.96	36.46	37.55	33.63	35.83	37.54	35.37	34.89	36.71	36.16
Average (s)	35.91									

Table 5.2: Sequence loading times (in seconds)

The values presented above are expected since the headset being used does not have the processing power of a normal computer. Besides this, during the loading of the sequence, the following processes are executed:

- Reading of the configuration file of the chosen sequence;
- Loading of the entire data from the JSON files;
- Loading of all the images from the camera feeds;
- Loading of all the binary files from the point clouds;
- Pre-decoding of each point cloud file;
- Creation of the data plots;
- Creation of the trajectory of the ego.

Table 5.3 displays the average loading times (in ms) for the trajectory when loading the sequence, using the same ten iterations as before. These values are acceptable since the process to create the trajectory utilizes all the JSON files of the ego to get its position. With this, the midpoints are calculated and each line of the trajectory is created. This process is best detailed in section 3.4.3.

Load Time (ms)	543.2	588	435.7	536.3	445.6	504.8	477.9	549.9	456	556.2
Average (ms)	509.4									

Table 5.3: Trajectory loading times (in milliseconds)

5.1.2 Animation Times

Regarding the animation of the sequence, it is also important to know the time it takes to animate the frames. Figure 5.1 illustrates the average time (in ms) for ten iterations, for each data of the animation to be loaded in each frame of the sequence, excluding the point clouds. Since the point clouds demand more memory, they were analyzed differently in the next figure. Inside the Oculus Go scene, the animation runs smoothly with an average of nine frames per second. There

are some notable deviations in the beginning since it takes some time to create all the elements needed for the sequence. In the following frames, the elements are only updated.

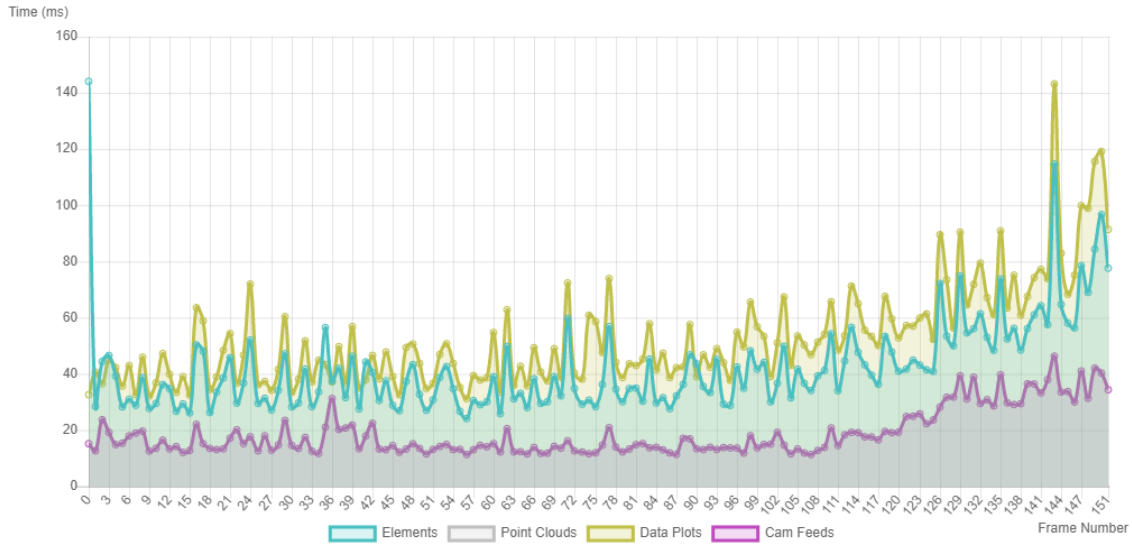

Figure 5.1: Average loading time (in ms) of ten iterations, excluding the point clouds, for each data in the sequence

The data plots also took some time to load in each frame, since chart.js has to update each parameter of each of the three data plots available (linear velocity, linear acceleration, angular velocity). The loading of the camera feed images is the process that takes less time. This is because the images are already loaded into memory, and the image source of the HTML element is the only parameter being updated.

It is also evident that the processing times increase in the last frames, by looking at the graph. This happens because some data only increases and cannot be deleted. An example of this is in the data plots. The plots need more data since, in each frame, they are updated with the information of the ego. Another example is the collection of the metrics, that were only required to create this graph. Data about each frame was stored into memory, to later be exported at the end of the sequence and create the graphs presented here.

When including the point clouds, the animation suffers from *lagging* since the point clouds take a long time to be decoded and loaded into the sequence, as displayed in Figure 5.2. This figure illustrates the clear comparison between the loading time of the point clouds and the other data. Better ways to optimize the loading times of the point clouds inside a standalone VR headset are still being investigated.

5.2 VR Headset Comparison

Unfortunately, the Oculus Quest only arrived three days before the deadline of the dissertation. As such, a small experiment was conducted to compare both Oculus devices. In future sessions

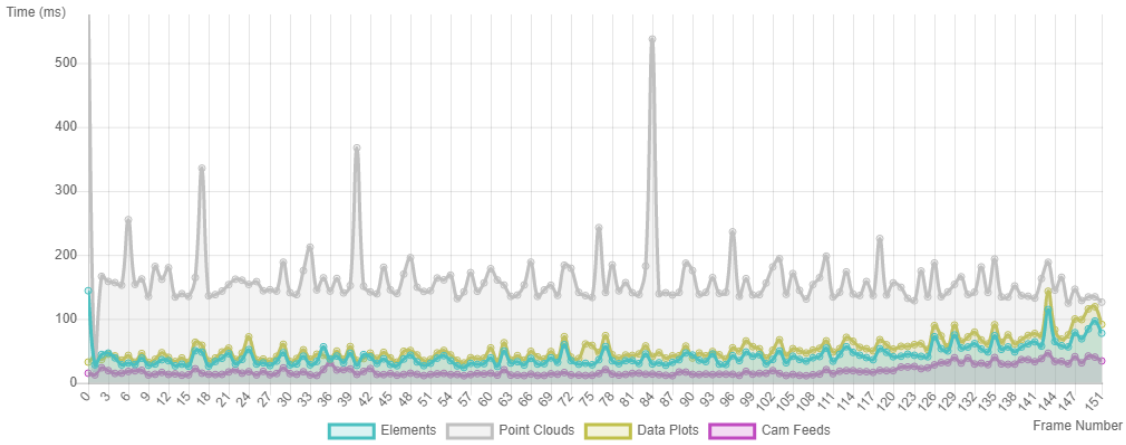

Figure 5.2: Average loading time (in ms) of ten iterations for all the data in the sequence

involving participants, a more extensive comparison between these HMDs will be done.

Using Oculus Quest, the values of the times are very different. Figure 5.3 illustrates the average time of five iterations for each data to load in each frame, without the point clouds. The values of the loading times of each data are decreased by almost half, and this is due to the Oculus Quest having 4G of memory.

Even so, it can be optimized even more since in this experiment, both the data plot and camera feed images are appearing on the screen at the same time. Better ways to display images and reduce the loading times are currently being investigated.

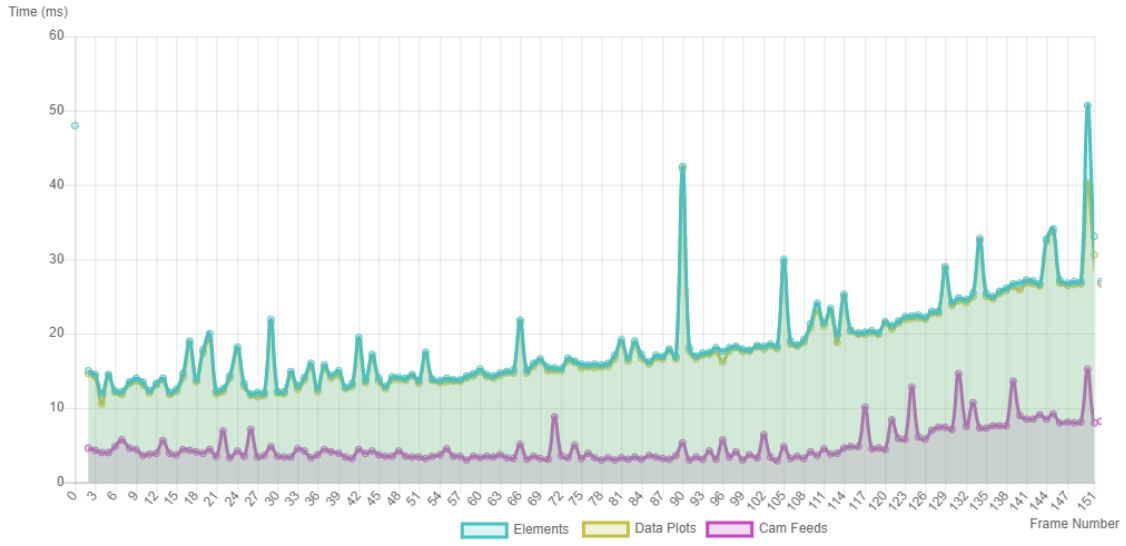

Figure 5.3: Average loading time (in ms) of five iterations, excluding the point clouds, for each data in the sequence, using the Oculus Quest

5.3 User Testing

An experiment of Kratos VR was made with a small group of participants. The participants were asked to test both Kratos Visualization and Kratos VR, and afterward, fill a questionnaire to report their analysis and experience in using the application. The questions and answers to this questionnaire are presented in Appendix A.

In the following subsections, an evaluation of the technology acceptance model (TAM), an information systems theory that models how users come to accept and use a technology, and a comparison between both Kratos applications is made. The participants were also asked if they had any motion sickness while wearing the headset.

The group consisted of six participants, all with limited knowledge about autonomous driving and virtual reality. The demographics of the participants are shown in Table 5.4.

Age	Male	Female
Lower than 18	0	1
Between 18 and 48	1	2
More than 48	1	1

Table 5.4: Demographic of the participants

Unfortunately, this experiment was affected by the pandemic situations happening during the middle of 2020, and as such, it was not possible to have more participants. Nonetheless, after the conditions are better, the experiment will be redone with more people, including participants that know both AD and VR.

5.3.1 Technology Acceptance

An evaluation of Kratos VR was made following the TAM [85], adapted to VR hardware [86]. Figure 5.4 illustrates the structural hypothesized model used that serves as the basis for the experiment.

The questionnaire contains the questions that the participants were asked to answer, adapted to the Kratos VR application. Table 5.5 illustrates the results of the questionnaire, where the participants were asked for each question, to answer between 1 to 5. If the participant answered 1, he strongly disagrees with the hypothesis of the question, while if he answered 5 he strongly agrees. The factor loading represents the percentage of the average of these values.

These first results show that the participants agree that VR is effective and productive in the AD area. However, there are some adjustments to be made in the foreseeable future to make the application more clear and understandable. Although it shows that the participants have some apprehension in purchasing VR devices, the answers to some questions prove that virtual reality is fascinating and impressive. The reason for the apprehension in buying these devices may be due to the current prices on the market.

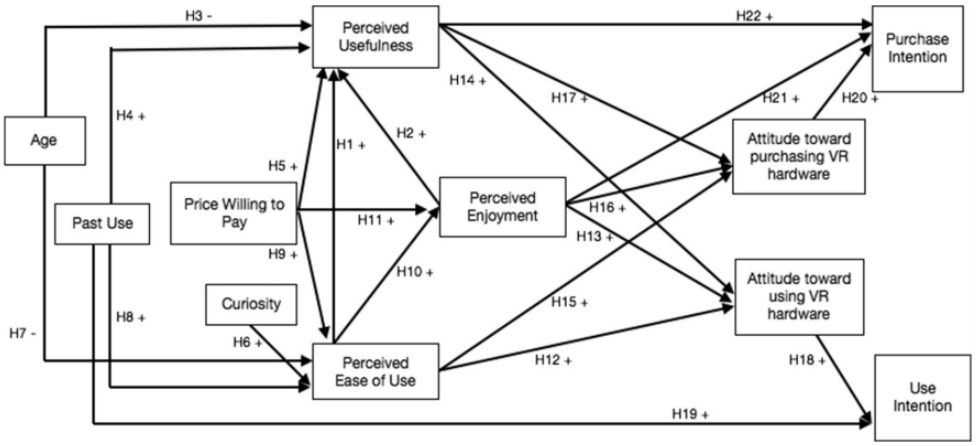

Figure 5.4: TAM structural model, created by Kerry Manis and Danny Choi [86]

It is worth to mention that this experiment will still be made with more participants, to collect more data as the application progresses. Since only six participants conducted this experiment, the results are not justified to reach a definitive conclusion.

5.3.2 Kratos Comparison

The experiment with Kratos Visualization and Kratos VR was made using the following steps:

1. A small tutorial was provided to the participant on how to use both applications;
2. The participant was then asked to use the application in either Kratos VR or Kratos Visualization;
3. Small tasks were asked to be done, like changing the perspective of the camera, customizing a vehicle and controlling the animations;
4. After completing these steps, the participant used the other application that he did not choose in Step 2, and repeated step 3;
5. Finally, the participant was asked to fill the questionnaire.

The results for this questionnaire are displayed in Table 5.6.

The results show that while Kratos VR has more potential and better visuals, it still suffers from being displayed on a small screen. Better ways to present the necessary information about the sequence are currently being researched, with some plans for further work regarding this, described in chapter 6.

5.3.3 Sickness Evaluation

While the participants tested the application, they were asked to recall if any symptoms had occurred. For this, a motion sickness questionnaire created by Robert S. Kennedy et al was used

Construct	Measurement instrument	Factor Loading (0 - strongly disagree; 1 - strongly agree)
Perceived usefulness	I believe using VR helps autonomous driving productivity.	0.75
	I believe using VR increases autonomous driving effectiveness.	0.75
	Using VR improves the quality of autonomous driving applications.	0.75
	Using VR improves the usefulness of autonomous driving applications.	0.833
Perceived ease of use	I believe using VR applications would be easy for me.	0.708
	I believe Kratos VR was clear and understandable.	0.708
	I find Kratos VR flexible to interact with.	0.875
	It would be easy for me to become skillful at using VR applications.	0.333
Perceived enjoyment	I believe using Kratos VR was enjoyable.	0.833
Intention to use	There is a high likelihood that I will use VR visualization applications within the foreseeable future.	0.583
	I intend to use VR visualization applications within the foreseeable future.	0.25
	I will use VR visualization applications within the foreseeable future.	0.125
Intention to purchase	There is a high likelihood that I will purchase VR devices within the foreseeable future.	0.417
	I intend to purchase VR devices within the foreseeable future.	0.167
	I will purchase VR devices within the foreseeable future.	0.083
Curiosity	VR is interesting to me.	0.958

Table 5.5: Form using TAM

[87]. Illustrated in Table 5.7, this questionnaire allows the participants to describe their symptoms if they had any.

Since only six participants experimented in the questionnaire, for now, the results did not provide the desired feedback. Of all the six participants, one male, older than 48 answered that he had slight sweating, and one female, older than 48, answered that she had a slight dizziness. The experiments took no longer than 15 minutes while using the headset. Even so, during the further development of the application, measures to diminish the probability of having these symptoms will be made.

Questions	Kratos Visualization	Kratos VR
Application with better usability	33.3%	66.7%
Application that shows more information	83.3%	16.7%
Application that shows more potential	0%	100%
Application with better visuals	0%	100%

Table 5.6: Results of Kratos Comparison

		None	Slight	Moderate	Severe
1	General discomfort	6	0	0	0
2	Fatigue	6	0	0	0
3	Headache	6	0	0	0
4	Eye strain	6	0	0	0
5	Difficulty focusing	6	0	0	0
6	Salivation increasing	6	0	0	0
7	Sweating	5	1	0	0
8	Nausea	6	0	0	0
9	Difficulty concentrating	6	0	0	0
10	*Fullness of the head*	6	0	0	0
11	Blurred vision	6	0	0	0
12	Dizziness with eyes open	5	1	0	0
13	Dizziness wit eyes closed	6	0	0	0
14	Vertigo	6	0	0	0
15	Stomach awareness	6	0	0	0
16	Burping	6	0	0	0

Table 5.7: Motion sickness form based on the Simulation Sickness Questionnaire by Robert S. Kennedy et al [87].

Chapter 6

Conclusions and Future Work

Countless companies need data visualization applications to test their experiments involving autonomous driving data. It is believed that Kratos VR achieved its goal of contributing to the research made in this area. This dissertation provided a generic visualization web platform that can be accessed through any VR device connected to the internet. Because of this, any developer around the world can use it to analyze AD data.

The objectives of this dissertation were mostly accomplished, although the application faced some issues along with its development. Since most VR devices have very limited memory usage and the point clouds require a reasonable amount of space, better ways to optimize the process of the decoding and loading of the point clouds are being investigated.

Despite the results of this dissertation, the integration of autonomous driving and virtual reality is guaranteed to have the potential to progress. VR is growing immensely and with the constant amount of new techniques and enhancements being developed in this area, there will always be room for improvement on this project.

6.1 Results of Kratos VR

The usage of A-Frame proved to be completely adequate in handling the needs of the project. Although it still requires a lot of work to improve its features and main libraries, it is assured that the community will handle this new framework with care and commitment. WebXR is in its first stages and new features and libraries are being developed constantly.

The animation without the point clouds runs smoothly and hardly any issues arise when functioning. Still, a better way to improve the loading of the data plots and elements during the animation is being researched.

It is believed the interface of the application is user friendly and follows the main guidelines for VR applications, despite very few participants experimenting with the application. The results show that the menu is easy to interact with and simple to learn. Being one of the first AD data visualization applications of its kind, there are many ways in which this application can grow.

Both the Oculus Go and Oculus Quest have different functionalities, and since their controllers and their input works on the application, it is believed that this application can run in any VR device, without any issues, and as long it is connected to the internet.

6.2 Future Work

Kratos VR was intended to serve as an iteration to Kratos Visualization but the project will continue to be further developed and improved. Some ideas are being developed and will soon be implemented in the application, like the addition of sounds when the user presses a certain button or sees the information about a vehicle. There are also new types of AD data, like radar returns and ultrasonic echos, that should also be available to use inside the application. The application will also soon use data sets from other sources, instead of just the KITTI Vision Benchmark Suite.

The map can use any tiles available in a designated server, as such, one of our objectives is to give this choice to the user. In the menu, the user should be able to selected and customize the appearance of the maps. Besides this, the size of the map will also be adapted to each sequence, since at the start, the size of the map is always the same for every sequence.

An option to improve the control of the animation is the addition of a slider, by replacing the buttons to decrease and increase the frames. One other idea is to use the thumbstick in the Oculus Quest controller, and the touchpad in the Oculus Go controller to change the frames.

One other interesting aspect that will be implemented in the application is the customization of the controllers. For now, the controller is represented by its respective model inside the scene. One idea is to make the images of the camera feeds or the data plots appear on top of a certain area of one controller. For example, the user can see an image of the camera feed on top of his wrist. With this, the user can alternate between images of camera feeds or data plots, and the animation becomes smoother since only one image appears on the screen. This guarantees better visibility of the image for the user since he is the one controlling the position of the image through the position of his hand with the controller.

An integration with ROS will also be investigated, with the option of facilitating the usage of the AD data. With the Oculus Quest arriving very late, there was almost no chance of implementing much of the functionalities that use six DoF or two controllers. This includes the following:

- Allow the user to walk around the scene;
- Allow the user to change the background to the scenario of his location (since the Oculus Quest has four wide-angle cameras located on each corner of the headset);
- Utilize the thumbstick to move around or change frames;
- Add different purposes for the raycasting line of each controller.

Another idea is the use of mobile phones. By allowing the application to be viewed on a small screen, the user can utilize VR equipment like Google Cardboard or Google Daydream. More

experiments with a larger number of participants will also be created, with the option to compare all the different types of data visualization. With this in mind, it is guaranteed that this application will be further worked and developed on.

The pandemic situation slightly affected some of the development of this dissertation, however, this project will continue and more tests and experiments will be made shortly. There are also plans to create a publication about the work developed for this dissertation.

Bibliography

[1] Rahul Kala. Advanced driver assistance systems. In *On-Road Intelligent Vehicles*, volume 4, pages 59 – 82. Butterworth-Heinemann, Dec 2016.

[2] Inga Harris. Embedded software for automotive applications. In *Software Engineering for Embedded Systems*, volume 22, pages 767 – 816. Newnes, Oxford, Dec 2013.

[3] Uber. deck.gl - introduction. *Large-scale WebGL-powered Data Visualization*, Dec 2015. Available at ***https://deck.gl/***.

[4] Duarte Barbosa, Miguel Leitão, and João Silva. Web client for visualization of adas/ad annotated data-sets. In Manuel F. Silva, José Luís Lima, Luís Paulo Reis, Alberto Sanfeliu, and Danilo Tardioli, editors, *Robot 2019: Fourth Iberian Robotics Conference*, volume 1092 of *Advances in Intelligent Systems and Computing*, pages 191–202. Springer, Nov 2019.

[5] William R. Sherman and Alan B. Craig. Introduction to virtual reality. In *Understanding Virtual Reality*, volume 1 of *The Morgan Kaufmann Series in Computer Graphics*, pages 5 – 37. Morgan Kaufmann, San Francisco, Jan 2002.

[6] Altran Portugal. Engenharia automóvel e desenvolvimento de veículos - altran. Available at ***https://www.altran.com/pt/pt-pt/quem-somos/***.

[7] Joe Bardi. What is virtual reality? vr definition and examples. *Marxent Labs*, Jul 2019. Available at ***https://www.marxentlabs.com/what-is-virtual-reality/***.

[8] Vortex Colab. Engenharia automóvel e desenvolvimento de veículos - altran. Available at ***http://www.vortex-colab.com/organisation/***.

[9] Duarte Alexandre dos Santos Barbosa. Web client for visualization of datasets in the automotive sector. Master's thesis, Instituto Superior de Engenharia do Porto, Aug 2019.

[10] Eric Gundersen. About mapbox. *Mapbox*, 2010. Available at ***https://www.mapbox.com/about/company/***.

[11] David Hershberger, David Gossow, and Josh Faust. About ros. *ros.org*, 2007. Available at ***https://www.ros.org/about-ros/***.

[12] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. *ICRA Workshop on Open Source Software*, 3:1–5, Jan 2009.

[13] David Hershberger, David Gossow, and Josh Faust. Rviz. *ros.org*, 2018. Available at ***http://wiki.ros.org/rviz***.

[14] J. O'Neill, S. Ourselin, T. Vercauteren, L. Da Cruz*, and C. Bergeles*. VRViz: Native VR Visualization of ROS Topics. *Conference on New Technologies for Computer and Robot Assisted Surgery*, 2019.

[15] Russell Toris. Roslibjs. *Ros.org*, 2017. Available at ***http://wiki.ros.org/roslibjs***.

[16] Will Maddern, Geoffrey Pascoe, Chris Linegar, and Paul Newman. 1 year, 1000 km: The oxford robotcar dataset. *The International Journal of Robotics Research*, 36, Nov 2016.

[17] Andreas Geiger, P Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: the kitti dataset. *The International Journal of Robotics Research*, 32:1231–1237, Sep 2013.

[18] Gary Bishop and Henry Fuchs. Research directions in virtual environments: Report of an nsf invitational workshop, march 23-24, 1992, university of north carolina at chapel hill. *SIGGRAPH Comput. Graph.*, 26(3):153–177, Aug 1992.

[19] Michael A. Gigante. 1 - virtual reality: Definitions, history and applications. In R.A. Earnshaw, M.A. Gigante, and H. Jones, editors, *Virtual Reality Systems*, pages 3 – 14. Academic Press, Boston, 1993.

[20] George Robertson, Mary Czerwinski, and Maarten Dantzich. Immersion in desktop virtual reality. In *Proceedings of the 10th annual ACM symposium on User interface software and technology*, pages 11–19, Jan 1997.

[21] Jonathan Steuer. Defining virtual reality: Dimensions determining telepresence. *Journal of Communication*, 42(4):73–93, Dec 1992.

[22] Grigore Burdea and Philippe Coiffet. Virtual reality technology. *Presence*, 12:663–664, Dec 2003.

[23] Tomasz Mazuryk and Michael Gervautz. Virtual reality - history, applications, technology and future. Technical report, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria, Feb 1996.

[24] Richard Fussenegger. *WebGL™ – 3D Canvas Programming: History, Development and Future*. PhD thesis, FH Joanneum, Apr 2012.

[25] Srushtika Neelakantam and Tanay Pant. Introduction to vr and webvr. In *Learning Web-based Virtual Reality*, pages 1–4. Apress, Mar 2017.

[26] Blair Maclntyre and Trevor F. Smith. Thoughts on the future of webxr and the immersive web. In *2018 IEEE International Symposium on Mixed and Augmented Reality Adjunct (ISMAR-Adjunct)*, pages 338–342, Oct 2018.

[27] Arturo Paracuellos and Blair Maclntyre. Progressive webxr. *Mixed Reality Blog*, pages 1–4, Apr 2018.

[28] Manish Goregaokar. Converting a webgl application to webvr. *Mozilla Hacks – the Web developer blog*, Sep 2018. Available at ***https://hacks.mozilla.org/2018/09/converting-a-webgl-application-to-webvr/***.

[29] Diego Marcos, Kevin Ngo, and Don McCurdy. Introduction – a-frame. *A-frame*, Dec 2015. Available at ***https://aframe.io/docs/1.0.0/introduction/***.

[30] Andy Gill. Aframe: A domain specific language for virtual reality: Extended abstract. In *Proceedings of the 2nd International Workshop on Real World Domain Specific Languages*, RWDSL17. Association for Computing Machinery, Feb 2017.

[31] Diego Marcos, Kevin Ngo, and Don McCurdy. Html & primitives – a-frame. *A-Frame*, Dec 2015. Available at ***https://aframe.io/docs/1.0.0/introduction/html-and-primitives.html***.

[32] Solange Gomes Santos and Jorge C. S. Cardoso. Web-based virtual reality with a-frame. In *2019 14th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–2, Jun 2019.

[33] Diego Marcos, Kevin Ngo, and Don McCurdy. Entity-component-system – a-frame. *A-Frame*, Dec 2015. Available at ***https://aframe.io/docs/1.0.0/introduction/entity-component-system.html***.

[34] Brian Danchilla. *Three.js Framework*, pages 173–203. Apress, Jan 2012.

[35] Diego Marcos, Kevin Ngo, and Don McCurdy. Developing with three.js – a-frame. *A-Frame*, Dec 2015. Available at ***https://aframe.io/docs/1.0.0/introduction/developing-with-threejs.html***.

[36] Francisco Moreno, Esmitt Ramirez, Francisco Sans, and Rhadamés Carmona. An open source framework to manage kinect on the web. In *2015 Latin American Computing Conference (CLEI)*, Oct 2015.

[37] Matthew Jackson. Vr with aframe vs three.js vs babylon.js. *Stellar Build*, Apr 2016. Available at ***http://www.stellarbuild.com/blog/article/vr-with-aframe-vs-threejs-vs-babylonjs***.

[38] Wikipedia contributors. List of webgl frameworks. *Wikipedia*, Jan 2020. Available at ***https://en.wikipedia.org/wiki/List_of_WebGL_frameworks***.

[39] Jason Jerald, Peter Giokaris, Danny Woodall, Arno Hartholt, Anish Chandak, and Sebastien Kuntz. Developing virtual reality applications with unity. In *2014 IEEE Virtual Reality (VR)*, pages 1–3, Mar 2014.

[40] Y. Kuang and X. Bai. The research of virtual reality scene modeling based on unity 3d. In *2018 13th International Conference on Computer Science Education (ICCSE)*, pages 1–3, Aug 2018.

[41] Ahmed Hussein, Fernando Garcia, and Cristina Olaverri Monreal. Ros and unity based framework for intelligent vehicles control and simulation. In *2018 IEEE International Conference on Vehicular Electronics and Safety (ICVES)*, pages 1–5. IEEE, Jul 2018.

[42] Paul E. Dickson, Jeremy E. Block, Gina N. Echevarria, and Kristina C. Keenan. An experience-based comparison of unity and unreal for a stand-alone 3d game development course. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '17, page 70–75, New York, NY, USA, Jun 2017. Association for Computing Machinery.

[43] Eleftheria Christopoulou and Stelios Xinogalos. Overview and comparative analysis of game engines for desktop and mobile devices. *International Journal of Serious Games*, 4:21–36, Dec 2017.

[44] Josh Petty. What is unity 3d & what is it used for? *Concept Art Empire*, Mar 2019. Available at ***https://conceptartempire.com/what-is-unity/***.

[45] Dejan Gajsek. Unity vs unreal engine for xr development: Which one is better? *CircuitStream*, Mar 2020.

[46] Breno Carvalho, Marcelo Soares, André Neves, Gabriel Soares, and Anthony Lins. *Virtual Reality devices applied to digital games: a literature review*, pages 125–147. CRC Press, Jul 2016.

[47] Christoph Anthes, Rubén García Hernandez, Markus Wiedemann, and Dieter Kranzlmüller. State of the art of virtual reality technologies. In *2016 IEEE Aerospace Conference*, pages 1–19, Mar 2016.

[48] Carsten Lecon. Motion sickness in vr learning environments. In *14th International Conference on Computer Science and Education*, COM2018-2514, page 1–16, May 2018.

[49] Yan Yan, Ke Chen, Yu Xie, Song Yiming, and Yonghong Liu. *The Effects of Weight on Comfort of Virtual Reality Devices*, pages 239–248. Springer International Publishing, Jan 2019.

[50] Lisa Rebenitsch. Managing cybersickness in virtual reality. *XRDS*, 22(1):46–51, Nov 2015.

[51] Matthias Treitler and Jess Tellford. A-frame map. *Github*, Oct 2016. Available at ***https://github.com/jesstelford/aframe-map***.

[52] Dany Laksono and Trias Aditya. Utilizing a game engine for interactive 3d topographic data visualization. *ISPRS International Journal of Geo-Information*, 8:361, Aug 2019.

[53] Matthias Treitler. A-frame tangram component. *Github*, Aug 2017. Available at ***https://github.com/mattrei/aframe-tangram-component***.

[54] Robert Kaiser. Vr map - a-frame demo using openstreetmap data. *Kairo*, Jul 2018. Available at ***https://home.kairo.at/blog/2018-07/vr_map_a_frame_demo_using_openstreetmap***.

[55] Artem Pavlenko. Mapnik. *OpenStreetMap*, Apr 2018. Available at ***https://mapnik.org/***.

[56] Google Data Arts Team. Dat.gui vr. *Github*, Apr 2017. Available at ***https://github.com/dataarts/dat.guiVR***.

[57] Andrés Cuervo. A-frame dat.gui component. *Github*, Jul 2017. Available at ***https://github.com/cwervo/aframe-datgui-component***.

[58] Kevin Ngo. A-frame - building user interfaces. *Glitch*, May 2019. Available at ***https://glitch.com/ aframe-building-ui***.

[59] Henrik Nagel, Erik Granum, and Peter Musaeus. Methods for visual mining of data in virtual reality. *In Proceedings of the International Workshop on Visual Data Mining, in conjunction with ECML/PKDD2001, 2nd European Conference on Machine Learning and 5th European Conference on Principles and Practice of Knowledge Discovery in Databases*, pages 9–12, 10 2001.

[60] Paul Sullivan. *Graph-Based Data Visualization in Virtual Reality: A Comparison of User Experiences*. PhD thesis, California Polytechnic State University, Jun 2016.

[61] Francisco Hidalgo. *Virtual Reality Data Dashboard*. PhD thesis, Escuela Técnica Superior de Ingeniería de Telecomunicación, Universidad Rey Juan Carlos, Jun 2017.

[62] Nick Downie. Chart.js. *Chartjs*, Mar 2013. Available at ***https://www.chartjs.org/***.

[63] Frank Galligan. Draco 3d data compression. *Draco 3D Graphics Compression*, Oct 2017. Available at ***https://google.github.io/draco/***.

[64] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. In *IEEE Transactions on Visualization and Computer Graphics*, volume 5, pages 47–61, Jan 1999.

[65] Tianxin Huang and Yong Liu. 3d point cloud geometry compression on deep learning. In *Proceedings of the 27th ACM International Conference on Multimedia*, MM '19, page 890–898, New York, NY, USA, Oct 2019. Association for Computing Machinery.

[66] Alexandros Doumanoglou, Petros Drakoulis, Nikolaos Zioulis, Dimitrios Zarpalas, and Petros Daras. Benchmarking open-source static 3d mesh codecs for immersive media interactive live streaming. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9:190–203, Mar 2019.

[67] Sean Lilley. Compressing massive point clouds with 3d tiles and draco. *Cesium*, Feb 2019. Available at ***https://cesium.com/blog/2019/02/26/draco-point-clouds/***.

[68] Pocket7878. Draco a-frame example. *GitHub*, Jan 2017.

[69] Radu Rusu and Steve Cousins. 3d is here: Point cloud library (pcl). *IEEE International Conference on Robotics and Automation 2011 (ICRA 2011)*, pages 1–4, May 2011.

[70] Renato Toasa, Paúl Baldeón Egas, Miguel Saltos, Mateo Perreño, and Washington Quevedo. *Performance Evaluation of WebGL and WebVR Apps in VR Environments*, pages 564–575. Springer International Publishing, Cham, Oct 2019.

[71] Srushtika Neelakantam and Tanay Pant. *Bringing VR to the Web and WebVR Frameworks*, pages 5–9. Apress, Mar 2017.

[72] Matyas Szalai, Balázs Varga, Tamas Tettamanti, and Viktor Tihanyi. Mixed reality test environment for autonomous cars using unity 3d and sumo. In *2020 IEEE 18th World Symposium on Applied Machine Intelligence and Informatics (SAMI)*, pages 73–78, Jan 2020.

[73] Alexandre Nascimento, Anna Carolina Muller Queiroz, Lucio Vismari, Jeremy Bailenson, Paulo Cugnasca, Joao Junior, and Jorge Almeida. The role of virtual reality in autonomous vehicles' safety. In *2019 IEEE International Conference on Artificial Intelligence and Virtual Reality (AIVR)*, pages 50–507, Dec 2019.

[74] Stelios Kouzeleas. Conversion of gps data to cartesian coordinates via an application development adapted to a cad modelling system. *2nd International Conference on Experiments/Process/System Modelling/Simulation & Optimization (2nd IC-EpsMsO), Athens, 4-7 July, 2007*, pages 1–5, Jan 2007.

[75] National Imagery and Mapping Agency. United states department of defense, world geodetic system 1984: Its definition and relationships with local geodetic systems. *Tech. Rep., TR8350.2*, Jan 2000.

[76] George Gerdan and Rod Deakin. Transforming cartesian coordinates x, y, z to geographical coordinates. *Australian Surveyor*, 44:1–12, Jun 1999.

[77] Mangesh Nichat. Landmark based shortest path detection by using dijkestra algorithm and haversine formula. *International Journal of Engineering Research and Applications(IJERA)*, 3:162–165, May 2013.

[78] Chris Veness. Calculate distance, bearing and more between latitude/longitude points. *Movable Type Scripts*, Aug 2016. Available at ***https://www.movable-type.co.uk/scripts/latlong.html***.

[79] Michael Unser and Ateram Aldroubi. Polynomial spline signal processing algorithms. In *[Proceedings] ICASSP-92: 1992 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 177–180, Sep 1992.

[80] Krzysztof Pietroszek. Raycasting in virtual reality. In *Encyclopedia of Computer Graphics and Games*. Springer, Mar 2018.

[81] Diego Marcos, Kevin Ngo, and Don McCurdy. Raycaster. *A-Frame*, Dec 2015. Available at ***https://aframe.io/docs/1.0.0/components/raycaster.html***.

[82] Supermedium. Superframe. *Github*, May 2016. Available at ***https://github.com/supermedium/superframe***.

[83] Kangsoo Kim, Austin Erickson, Alexis Lambert, Gerd Bruder, and Greg Welch. Effects of dark mode on visual fatigue and acuity in optical see-through head-mounted displays. In *Symposium on Spatial User Interaction*, SUI '19, pages 1–9, New York, NY, USA, Oct 2019. Association for Computing Machinery.

[84] Mo Kargas. A-frame colorwheel. *Github*, Sep 2017. Available at ***https://github.com/mokargas/aframe-colorwheel-component***.

[85] Jen-Her Wu and Shu-Ching Wang. What drives mobile commerce?: An empirical evaluation of the revised technology acceptance model. *Information & Management*, 42(5):719 – 729, Jul 2005.

[86] Kerry Manis and Danny Choi. The virtual reality hardware acceptance model (vr-ham): Extending and individuating the technology acceptance model (tam) for virtual reality hardware. *Journal of Business Research*, 100:503 – 513, Dec 2018.

[87] Robert S. Kennedy, Norman E. Lane, Kevin S. Berbaum, and Michael G. Lilienthal. Simulator sickness questionnaire: An enhanced method for quantifying simulator sickness. *The International Journal of Aviation Psychology*, 3(3):203–220, 1993.

[88] Quinate Ihemedu-Steinke, Prashanth Halady, Gerrit Meixner, and Michael Weber. *VR Evaluation of Motion Sickness Solution in Automated Driving*, pages 112–125. Springer International Publishing, 06 2018.

Appendix A

Form of the conducted experiment

A.1 About the participant

A.1.1 Questions

1. What is your gender?
   (a) Male
   (b) Female
2. What is your age group?
   (a) Under 18
   (b) 18-32
   (c) 32-48
   (d) Over 48
3. Autonomous Driving Knowledge
   (a) I don't have knowledge about this subject
   (b) I know the subject but my work doesn't relate with it
   (c) My work is related with this subject
4. VR Experience
   (a) First time using VR
   (b) I rarely use VR
   (c) I use VR frequently
   (d) I develop VR applications

A.1.2 Answers

Participant	Gender	Age Group	AD Knowledge	VR Experience
1	Female	18-32	No	Yes
2	Female	18-32	Knows the subject	No
3	Male	18-32	Knows the subject	Yes
4	Female	Over 48	No	No
5	Male	Over 48	Knows the subject	No
6	Female	Under 18	No	No

A.2 After using Kratos VR

A.2.1 Questions

1. I understood everything that the application presented.
   (a) Answer 1 to 5 (1 - strongly disagree; 5 - strongly agree)
2. I learned more about Autonomous Driving after using Kratos VR.
   (a) Answer 1 to 5 (1 - strongly disagree; 5 - strongly agree)
3. I had some difficulty when interacting with the Menu.
   (a) Answer 1 to 5 (1 - strongly disagree; 5 - strongly agree)
4. I felt comfortable using the VR headset.
   (a) Answer 1 to 5 (1 - strongly disagree; 5 - strongly agree)
5. Did you have any motion sickness?
   (a) Filling grid, illustrated in Table 5.7

A.2.2 Answers

Participant	Q1	Q2	Q3	Q4	Motion Sickness
1	5	4	1	5	None
2	5	5	2	4	None
3	5	5	1	5	None
4	4	4	4	3	Slight Dizziness (Eyes opened)
5	5	4	3	3	Slight Sweating
6	3	5	1	5	None

A.3 Comparison between Kratos Visualization and Kratos VR

A.3.1 Questions

1. Which is the easier application to use?
   (a) Kratos Visualization
   (b) Kratos VR
2. What application shows more information?
   (a) Kratos Visualization
   (b) Kratos VR
3. What application do you believe has more potential?
   (a) Kratos Visualization
   (b) Kratos VR
4. What application did you enjoy more to use?
   (a) Kratos Visualization
   (b) Kratos VR

A.3.2 Answers

Participant	Easier to use	Shows more information	Has more potential	Enjoyed more
1	Kratos VR	Kratos VR	Kratos VR	Kratos VR
2	Kratos VR	Kratos Visualization	Kratos VR	Kratos VR
3	Kratos VR	Kratos Visualization	Kratos VR	Kratos VR
4	Kratos Visualization	Kratos Visualization	Kratos VR	Kratos VR
5	Kratos VR	Kratos Visualization	Kratos VR	Kratos VR
6	Kratos VR	Kratos VR	Kratos VR	Kratos VR

A.4 Technology Acceptance Model

A.4.1 Questions

1. I believe using VR helps autonomous driving productivity.
   (a) Answer 1 to 5 (1 - strongly disagree; 5 - strongly agree)
2. I believe using VR increases autonomous driving effectiveness.

   (a) Answer 1 to 5 (1 - strongly disagree; 5 - strongly agree)

3. Using VR improves the quality of autonomous driving applications.

   (a) Answer 1 to 5 (1 - strongly disagree; 5 - strongly agree)

4. Using VR improves the usefulness of autonomous driving applications.

   (a) Answer 1 to 5 (1 - strongly disagree; 5 - strongly agree)

5. I believe using VR applications would be easy for me.

   (a) Answer 1 to 5 (1 - strongly disagree; 5 - strongly agree)

6. I believe Kratos VR was clear and understandable.

   (a) Answer 1 to 5 (1 - strongly disagree; 5 - strongly agree)

7. I would find VR applications flexible to interact with.

   (a) Answer 1 to 5 (1 - strongly disagree; 5 - strongly agree)

8. It would be easy for me to become skillful at using VR applications.

   (a) Answer 1 to 5 (1 - strongly disagree; 5 - strongly agree)

9. I believe using Kratos VR was enjoyable.

   (a) Answer 1 to 5 (1 - strongly disagree; 5 - strongly agree)

10. There is a high likelihood that I will use VR visualization applications within the foreseeable future

    (a) Answer 1 to 5 (1 - strongly disagree; 5 - strongly agree)

11. I intend to use VR visualization applications within the foreseeable future.

    (a) Answer 1 to 5 (1 - strongly disagree; 5 - strongly agree)

12. I will use VR visualization applications within the foreseeable future.

    (a) Answer 1 to 5 (1 - strongly disagree; 5 - strongly agree)

13. There is a high likelihood that I will purchase VR devices within the foreseeable future

    (a) Answer 1 to 5 (1 - strongly disagree; 5 - strongly agree)

14. I intend to purchase VR devices within the foreseeable future.

    (a) Answer 1 to 5 (1 - strongly disagree; 5 - strongly agree)

15. I will purchase VR devices within the foreseeable future.

(a) Answer 1 to 5 (1 - strongly disagree; 5 - strongly agree)

16. VR is interesting to me.

    (a) Answer 1 to 5 (1 - strongly disagree; 5 - strongly agree)

A.4.2 Answers

Participant	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
1	3	3	4	4	4	1	5	3
2	5	4	5	5	4	4	4	2
3	4	5	4	5	4	5	5	3
4	4	5	3	3	2	2	4	1
5	4	3	4	5	4	3	4	1
6	4	4	4	4	5	1	5	2

Participant	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16
1	4	4	1	1	3	2	2	5
2	4	4	1	1	2	1	1	5
3	5	5	5	4	4	3	2	5
4	4	3	1	1	2	1	1	2
5	4	3	1	1	2	1	1	5
6	3	4	4	1	2	2	1	5