INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

Cliente web para visualização de datasets no setor automóvel

DUARTE ALEXANDRE DOS SANTOS BARBOSA
outubro de 2019

POLITÉCNICO
DO PORTO

Web client for visualization of datasets in the automotive sector

Duarte Alexandre dos Santos Barbosa

Increased

Mestrado em Engenharia Eletrotécnica e de Computadores
Área de Especialização de Automação e Sistemas
Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto
2019

Este relatório satisfaz, parcialmente, os requisitos que constam da Ficha de Unidade Curricular de Tese/Dissertação, do 2º ano, do Mestrado em Engenharia Electrotécnica e de Computadores

Candidato: Duarte Alexandre dos Santos Barbosa, Nº 1140508,
1140508@isep.ipp.pt
Orientação científica: Miguel Leitão, JML@isep.ipp.pt
Co-orientação: João Leite da Silva, joaomanuel.leitedasilva@altran.com

Mestrado em Engenharia Eletrotécnica e de Computadores
Área de Especialização de Automação e Sistemas
Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto
2019

Agradecimentos

Queria agradecer a toda a minha família, em especial aos meus pais que sempre me suportaram e tornaram possível a oportunidade de me congratular Mestre. Este percurso académico foi longo o suficiente para criar grandes laços de amizade e, como tal, gostaria de agradecer a todos os amigos que surgiram durante todo este tempo que passei no ISEP e que me ajudaram a superar etapas, tanto académicas, como pessoais. Com eles, estes cinco anos pareceram cinco dias. Finalmente, queria agradecer à Altran pela oportunidade que me foi dada, a toda a equipa de R&D e Analytics que ajudaram na minha integração nesta empresa, em especial ao João Silva, co-orientador deste projecto. Por parte do ISEP queria agradecer a esta grande instituição que me acolheu aquando a saída do ensino secundária e na qual me sagrei engenheiro, com especial atenção ao professor Miguel Leitão pela sua orientação cientifica neste projecto.

Duarte Barbosa

Resumo

Este projecto tem como objectivo o desenvolvimento de uma aplicação *web* capaz de ilustrar dados provenientes de *Advanced Driving Assistance Systems* (ADAS) e de sistemas de *Autonomous Driving* (AD). Estes dados podem ter múltiplas fontes incluindo: vídeo, nuvens de pontos, localizações, mapas, velocidades, rótulos, caixas delimitadoras, todos quais devem ser visualizados simultaneamente e facilmente controláveis pelo interface da plataforma.

Tipicamente, as empresas teriam de desenvolver as suas próprias plataformas de visualização para dar suporte ao desenvolvimento e visualização de *logs* de dados. A premissa deste projecto é mudar este tipo de mentalidade, fornecendo uma plataforma de visualização genérica, que pode carregar *logs* de dados de diferentes fontes num formato facilmente configurável. O facto de esta aplicação ser baseada em *web* irá permitir que várias equipas espalhadas pelo mundo analisem os dados provenientes de sistemas autónomos. Para além disto, o sistema a desenvolver deve ser suportado por plataformas *open-source* e compatível com os produtos mais comuns.

Para alojar e configurar os dados, será usado o ecossistema Hadoop, uma vez que permite armazenar grandes volumes de dados ao longo de aglomerados de computadores, utilizando modelos de programação simples. Para a criação e instanciação dos serviços Hadoop que serão necessários para o projecto, foi utilizado o gestor Cloudera instalado numa máquina virtual o que permitiu, com um *setup* mínimo, simular o ecossistema Hadoop para todos os testes necessários.

De forma a servir a página *web*, foi utilizado node.js para escrever um *script* responsável por criar um servidor HTTP. O *script* de node é também utilizado para atender pedidos provenientes do cliente de visualização e servir os ficheiros de dados como resposta.

Palavras-chave: sistemas autónomos, *logs* de dados, nuvem de pontos, Robot Operating System, Hadoop, WebGL, Cloudera, servidor, node.js

Abstract

This project aims to develop a web platform that is capable of showing data from Autonomous Driving (AD) and Advanced Driving Assistance Systems (ADAS). This data can have multiple sources including video, point clouds, location, map, velocity, labels, bounding boxes, all of which must be visualized simultaneously and be easily controlled by the platform's interface.

Typically, companies would have to develop their unique visualization platform to support the development and visualization of data logs. The premise of this project is to change this kind of mindset, providing a generic visualization platform, that can load logged data from different sources in an easily configurable format. The fact that this application is web-based allows for various teams spread across the world to analyze data from these autonomous systems. Furthermore, the system to be developed must be supported by open-source platforms and compatible with the most common products.

The Hadoop ecosystem will be used, since it allows for large volumes of data to be stored across clusters of computers using simple programming models. The cloudera manager will be used to create and instantiate all the needed Hadoop services, installing it in a virtual machine which allows to simulate an Hadoop environment for all the tests made with minimal setup.

To serve the web page, we used node.js to write a script that would create an HTTP server. The node script is also used to attend requests from the visualization client and serve the needed data files as a response.

keywords: automotive, data logs, point cloud, Robot Operating System, Hadoop, WebGL, Cloudera, server, node.js

Content

List of Figures

List of Tables

Acronyms

AD Autonomous Driving

ADAS Advanced Driving Assistance Systems

API Application Programming Interface

CDH Cloudera Distribution with Apache Hadoop

CPU Central Processing Unit

DOM Document Object Model

FPS Frames Per Second

GLSL OpenGL shading language

GNSS Global Navigation Satellite System

GPS Global Positioning System

GPU Graphics Processing Unit

GUI Graphical User Interface

HDFS Hadoop Distributed File System

HTML Hypertext Markup Language

HTTP Hyper Text Transfer Protocol

IMU Inertial Measurement Unit

JSON JavaScript Object Notation

LiDAR Light Detection And Ranging

PCAM Pedestrian Crash Avoidance Mitigation

RAID Redundant Array of Independent Disks

RAM Random Access Memory

RDBMS Relational Database Management System

ROS Robot Operating System

TROCS Tartan Racing Operator Control Station

VM Virtual Machine

Chapter 1

Introduction

Many road accidents occur due to human error. The need to reduce such accidents led to the creation of an automated system to assist the driver called Advanced Driving Assist Systems (ADAS). ADAS is a driving support system that provides information about the surrounding environment, aiming to increase not only car safety but also road safety. These support systems provide a variety of features such as automated lighting, adaptive cruise control and collision avoidance, Pedestrian Crash Avoidance Mitigation (PCAM), incorporate satnav/traffic warnings, connection to smartphones, driver alert to other cars or dangers, lane departure warning system, automatic lane centering, or simply show obstacles in blind spots.

Autonomous driving is a topic that has increased in interest over the years. Companies are investing in technologies that can assist the driver, avoiding obstacles, alerting the presence of other cars or simply automatically turn on car lights. Although there seems to be a great investment in the features and tools referenced above, there also seems to be less focus on generic platforms for debugging purposes that can aid developers to create these kind of autonomous systems. Usually companies develop their own simulation software for debugging purposes. For instance, Tartan Racing has "Tartan Racing Operator Control Station (TROCS)", which is a graphical interface based on QT for developers to monitor telemetry from Boss (Tartan Racing's self driving car) while it is driving and replay data offline for algorithm analysis [1]. Another example is Waymo with their carcraft, a simulator that not only allows to predict the behavior of the car in a simulation environment, but also allows the analysis of the real data gathered from the car [2]. Uber visualization team also created a technology that can be used in this area called Deck.gl, a webgl based framework that has a layer approach, meaning that it can render different types of data, such as point clouds, maps and polygons on top of each other [3]. They further created a framework called streetscape.gl, which is a visualization toolkit for autonomous and robotics data encoded in the XVIZ protocol [4] that can be used to debug

data from autonomous driving vehicles. Nonetheless, all these technologies are frameworks for building visualization platforms and not a fully built platform. As we can see, there are not many generic visualizations tools for ADAS and AD visualization on the market, especially web based. This project aims to change this premise by implementing a Deck.gl based, generic web platform for ADAS and AD visualization.

1.1 Contextualization

This thesis was proposed by Altran Portugal, in the scope of Autonomous Driving (AD) projects in its research department. This 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. Other modules of the initiative include Perception, V2X communication, data labeling, infotainment and control. This particular project aims to develop a web platform that is capable of showing data from an ADAS. Synchronized and processed visualization of data produced by an ADAS is important for debugging and developing algorithms for these systems and, since web technologies are a current and growing trend in this area, it makes perfect sense to adapt to a web environment. Besides, web platforms allow for easy and fast analysis between teams, without the need for a complex setup to have a functional workstation.

1.2 Objectives

This thesis aims to develop a web platform that is capable of showing logged data from an ADAS. The data can have multiple formats including video, point clouds, location, map, velocity, labels, bounding boxes all of which originate from multiple sources such as LIDARSs, cameras, inertial navigation systems (GPS/IMU), etc. All this information must be visualized simultaneously and be easily controlled by the platform's interface. To do so, there were a planned set of goals:

- Develop a database using existing frameworks that is capable of storing logged data from an ADAS;
- Develop a server that can be used as an interface between client and database. It must also organize the data in parameters that the framework being used for the development of the web application could use;
- Develop a 3D visualization web application for the client side;

By the end of this project, there should be a 3D visualization web platform that should illustrate the following points:

- Present all the data provided by the ADAS;
- Handle real-time viewpoints and perspectives;
- Real-time selection of which components to visualize;
- Fully control the synchronized playback of all data (including backward functionality).

It is of our most interest to make this platform available not only in a desktop format but also as a mobile version. To do so we have to consider certain mobile restrictions, such as a lack of right and middle click, scroll, as well as the inability to drag and drop. With all of this in mind, one of the objectives of this project is also to develop a mobile version that has less interactivity, created only as a visualization tool and not for analysis.

To meet the compatibility and coherence needs of the company, the data supply should be based on Robot Operating System (ROS), through rosbags (logs) or in real-time, and the storage of data for visualization should be based on the Hadoop ecosystem.

1.3 Project work breakdown

In the beginning of the project, a set of initial tasks/features were planned according to what was considered as project needs. Those tasks are represented in Figure 1.1 as 2D Canvas, 3D canvas, and user interface, which are part of the web client application and the server and data lake which are part of the data server and storage.

1.3.1 Canvas 3D

Creation of a 3D canvas that is capable of illustrating the ADAS/AD system given point cloud, surrounding objects and camera images. In order to do so, this process will be organized in the following tasks:

T1 - Creation of a layer that is capable of illustrating the data from the point cloud.

T2 - Creation of a layer capable of illustrating the data from the origin vehicle.

T3 - Creation of a layer capable of illustrating all the surrounding objects data.

T4 - Creation of a layer capable of drawing road lines.

T5 - Creation of a layer capable of drawing the trajectory line of the respective vehicle.

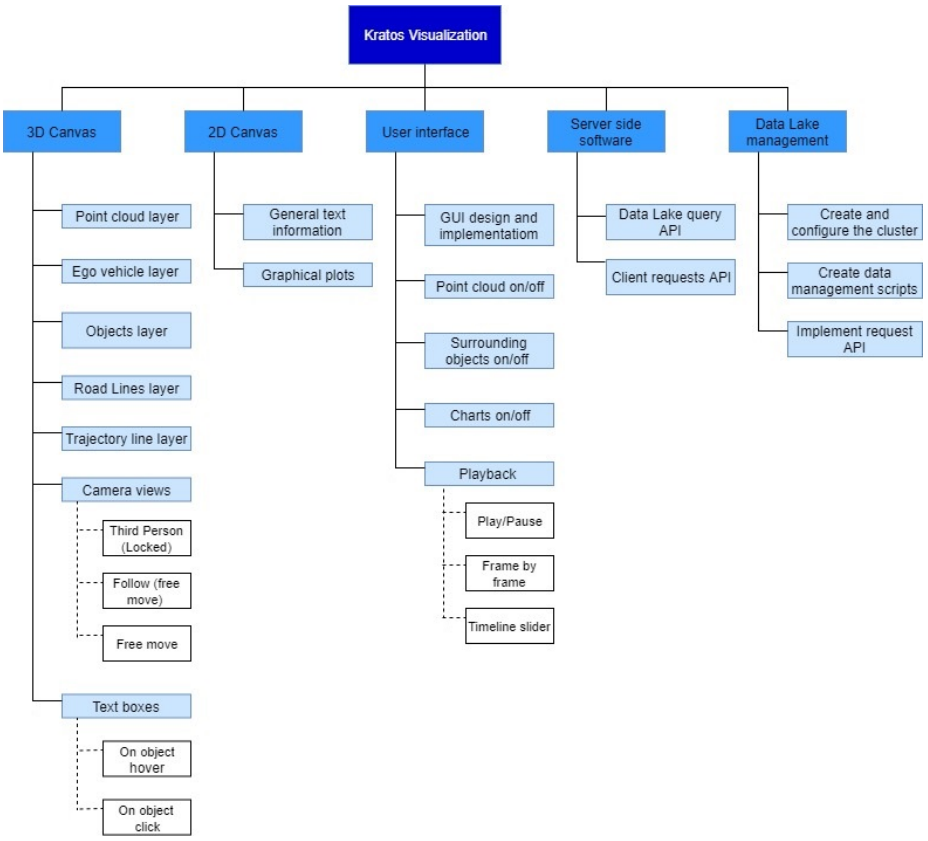


Figure 1.1: Work breakdown

T6 - Creation of different camera view modes, namely third-person view (with map movement locked), follow view (with map movement unlocked and camera following the ego vehicle) and free move view (with full control of map movement and camera).

T7 - Creation of information text boxes associated with the ego vehicle and the different objects. These text boxes can be shown on object hover or object click.

1.3.2 Canvas 2D

Creation of a 2D canvas that is capable of presenting general text information and graphic plots, with the various data given by the ADAS/AD system (accelerations, velocities, etc). This process will be organized in the following tasks:

T1 - Creation of a graphical plot interface for allowing the display of graphical information of data (for example, velocity or acceleration).

T2 - Creation of an interface for displaying general text information present on the data, as defined by the user.

1.3.3 User interface

Creation of a graphical user interface (GUI) capable of loading different sequences of frames and making the complete playback of those frames (step by step, play, pause), turning on and off the different charts and layers (point cloud, surrounding objects), changing the view mode and drag and drop some of the GUI elements. This process will be organized in the following tasks:

T1 - Design and implementation of the GUI with all the needed buttons, sliding windows, and tabs.

T2 - Adding the functionality of disabling or enabling the point cloud.

T3 - Adding the functionality of disabling or enabling all the objects in the map that have the same category associated (car, van, pedestrian, etc).

T4 - Adding the functionality of disabling or enabling the different charts, as well as some of their parameters (velocity, acceleration, etc).

T5 - Adding the functionality of complete playback, meaning that the user can play/pause the sequence, run the sequence step by step and change the current frame using a timeline slider.

1.3.4 Server side software

Creation of a server side software that will implement an interface for querying data from the storage and handle client requests. This process will be organized in the following tasks:

T1 - Design and implement a query API for querying the data lake according to the visualization client requests.

T2 - Design and implement an API for handling visualization client requests for information.

1.3.5 Data lake

Creation and configuration of a cluster, creation of data management scripts, implementation of the Request API. This process will be organized in the following tasks:

T1 - Create and configure a cluster capable of holding, in our platform defined format, the logged information of ADAS and AD datasets.

T2 - Create the scripts to handle the management of the data lake and all contained information.

T3 - Implement the query request handling API on the data lake instance.

1.4 Organization of the dissertation

This dissertation is composed of five chapters including the introduction. A brief project introduction, as well as the objectives and tasks that were planned to succeed in the implementation and development, were shown in the first chapter. Chapter two is a more technical analysis of the current state of the art. This means this chapter will focus more on the technologies and tools available to develop the project in hands, rather than studying similar solutions on the market. The third chapter exploits the project difficulties and the most important steps and decisions that were made, as well as the explanation behind them. The project results and analysis are illustrated in chapter four. A couple of metrics that were taken throughout the project development, as well as an introspective about these results will be presented in this chapter. Lastly, for the fifth and final chapter, not only some conclusions will be made, but also some future work planning.

Chapter 2

Analysis of available tools

To develop a data visualization app, there are a few requirements to keep in mind. It is needed a database to store all the information that is going to be shown, a server to work as a "bridge" between the database and the user interface, and lastly, the user interface itself. In this chapter we will present a set of tools that can aid in all these matters, such as the different frameworks available to develop a web platform capable of displaying 3D visualization Data, different types of storage platforms that can be used to produce a reliable database and lastly some different types of data sets that can be used within the objectives of this project, to supply the needed data.

2.1 3D visualization web frameworks

Thanks to APIs like WebGL, modern browsers are fully capable of rendering 2D and 3D interactive computer graphics without third-party plugins. It renders directly to the device's GPU so it can display complex images, animations, dynamic shading, and realistic physics, all inside a web page canvas element [5].

WebGL is a cross-platform, royalty-free API used to create 3D graphics in a Web browser, designed and maintained by the non-profit Khronos Group. "Based on OpenGL ES 2.0, WebGL uses the OpenGL Shading Language (GLSL) and offers the familiarity of the standard OpenGL API. Because it runs in the HTML5 Canvas element, WebGL has full integration with all Document Object Model (DOM) interfaces" [6].

WebGL has very extensive capabilities but, on the other hand, it has a slow-paced learning curve and so it is hard to understand for beginners. To counter this problem there are several frameworks built on top of WebGL. In this section, we will cover three different web-oriented 3D visualization frameworks, namely Three.js, Babylon.js and Deck.gl, all of which are based on the WebGL API.

2.1.1 Three.js

Three.js is a framework for coding general-purpose 3D online experiences. It is one of the most used libraries for WebGL implementations as an open-source project with over 900 contributors on GitHub. This library was created to take advantage of web-based renderers for creating GPU enhanced 3D graphics and animations. It has a flexible design, which makes it a great tool for general-purpose web animations like logos or modeling applications.

Using this framework in the HTML file is as simple as creating a script tag and linking it to the respective three.js JavaScript file as shown in the code below (Listing 2.1):

```
1    <script src="js/three.js"></script>
```

Listing 2.1: Include three.js source

To understand how this tool works, a simple example of how to make a 3D animation using a cube is shown. With Three.js, to create a container for displaying the animation, one can simply create an empty div element like the following code (Listing 2.2):

```
1    <div style="height: 100%; width: 100%" id="three"></div>
```

Listing 2.2: Create an empty div

To be able to display anything with three.js, we need three things-a scene, a camera, and a renderer-so that we can render the scene with a camera (Listing 2.3):

```
1       var div = document.getElementById('three');
2       var scene = new THREE.Scene();
3       var camera = new THREE.PerspectiveCamera( 75, div.offsetWidth /
      div.offsetHeight, 0.1, 1000 );
4       var renderer = new THREE.WebGLRenderer();
5       renderer.setSize( div.offsetWidth, div.offsetHeight );
6       div.appendChild( renderer.domElement );
```

Listing 2.3: Setup the animation environment

There are a few different cameras in three.js. In this example, the *PerspectiveCamera* was used as shown in line 3 of Listing 2.3. Its first attribute is the *field of view*. This attribute is the extent of the scene that is seen on the display at any given moment and this value is in degrees. The second one is the *aspect ratio*. To avoid the image being squished, it is advisable to use the width of the element divided by the height, which will maintain the aspect ratio. The next two attributes are the *near* and *far* clipping planes; these parameters prevent objects closer than *near*, or further than *far* to render [7].

Next is the renderer, in this case, it is used the *WebGLRenderer*, as shown in line 4 of the above code. three.js comes with a few others, often used as fallbacks for users with older browsers or for those who don't have WebGL support.

In addition to creating the renderer instance, it is also needed to set the size at which the app will be rendered. The coder can define the size of the app as the width and height of the browser window using the *setSize* method or, for performance-intensive apps, he can also give this method smaller sizes like *window.innerWidth/2* and *window.innerHeight/2*, which will make the app render at half size. It is possible to keep the size of the app, but rendering it a lower resolution. This is done by calling the *setSize* with false as *updateStyle*, which is the third argument. For instance, setSize(window.innerWidth/2, window.innerHeight/2, false) will render the app at half resolution, has long has the given <canvas> has 100% width and height.

The final step is to add the renderer element to the HTML document by appending-it to the div initially created, as shown in line 6 of the above code.

Creating a cube is done using the object *BoxGeometry*.This is an object that contains all the points (vertices) and fills (faces) of the cube. To give the cube a certain texture or color, three.js uses a variety of different materials. In this example it is used the *MeshBasicMaterial*, all materials take an object of properties that will be applied to them. In this case, it is only supplied a color attribute of 0x00ff00, which is green. This works the same way that colors work in CSS (hex colors). The third step is to add a *Mesh*. A mesh is an object that takes a geometry, and applies a material to it.

Finally, to add the created cube to the scene, we call *scene.add()*. By default, this will add the created element to the coordinates (0,0,0), which would cause both the camera and the cube to be inside each other. To avoid this, one can simply move the camera as shown in line 6 of the code below (Listing 2.4):

```
1       var geometry = new THREE.BoxGeometry( 1, 1, 1 );
2       var material = new THREE.MeshBasicMaterial( { color: 0x00ff00 }
      );
3       var cube = new THREE.Mesh( geometry, material );
4       scene.add( cube );
5
6       camera.position.z = 5;
```

Listing 2.4: Add a cube to the scene

To render the created cube we need what is called a render or animate loop, which will create a loop that causes the renderer to draw the scene every time the screen is refreshed (60 times per second on most screens) [7]. Lastly, to give the cube some sort of movement, we could use *cube.rotation* followed by the

respective axes, and increment its value to make a rotation animation (line 3 and 4 from Listing 2.5):

```
1       function animate() {
2         requestAnimationFrame( animate );
3         cube.rotation.x += 0.01;
4         cube.rotation.y += 0.01;
5         renderer.render( scene, camera );
6       }
```

Listing 2.5: Animate the cube

2.1.2 Babylon.js

Babylon.js is a real-time 3D engine using a JavaScript library for displaying 3D graphics in a web browser via HTML5. It was created by two Microsoft employees as an open-source project. This framework was originally made to build 3D games with HTML5, WebGL and Web Audio, but it evolved in a way that is now used for many different purposes outside gaming, including fields such as data visualization, making it very similar to three.js.

Just like three.js, to use this framework one can simply link to the respective JavaScript file, although Babylon.js has dependencies that require the open source Hand.js to be included as well. On the other hand, if linked to the correct repository, it is only needed to add the following script tag (Listing 2.6):

```
1     <script src="https://cdn.babylonjs.com/babylon.js"></script>
```

Listing 2.6: Add Babylon source code

To hold the 3D animation, Babylon.js requires an explicitly defined HTML5 canvas like the one in the code below (Listing 2.7):

```
1     <div style="height:100%; width: 100%" id="babylon"><canvas id="
      renderCanvas"></canvas></div>
```

Listing 2.7: Create an empty canvas

The next step is to load the Babylon 3D engine and attach it to the created canvas. This engine is responsible for preparing the scene and drawing to the respective canvas (Listing 2.8).

```
1         var canvas = document.getElementById('renderCanvas');
2         var engine = new BABYLON.Engine(canvas, true);
```

Listing 2.8: Load Babylon 3D engine

To be able to visualize any kind of 3D object we must create a scene, a camera to view the created scenes, and lighting. Just like in Three.js there are

many different types of cameras, in this case, is used the *FreeCamera*(line 2). All the different types of cameras have at least three parameters in common, a *name*, a camera *position* and the created *scene*. Babylon cameras also have a set of additional parameters, such as the field of view but in this example, only the very basics are going to be setup.

Unlike Three.js, Babylon requires adding lighting to the scene, otherwise any geometry would be completely black. In this example, the *HemisphericLight* (line 5) is used, which is an easy way to simulate an ambient environment light. A hemispheric light is defined by a direction, usually 'up' towards the sky. However, it is by setting the color properties that the full effect is achieved [8]. All these attributes can be seen in the following code (Listing 2.9):

```
1        var scene = new BABYLON.Scene(engine);
2        var camera = new BABYLON.FreeCamera('camera1', new BABYLON.
    Vector3(0, 5,-10), scene);
3        camera.setTarget(BABYLON.Vector3.Zero());
4        camera.attachControl(canvas, false);
5        var light = new BABYLON.HemisphericLight('light1', new
    BABYLON.Vector3(0,1,0), scene);
```

Listing 2.9: Create Babylon animation environment

After setting up all the cameras, scenes, lightning and renderer, it is time to add some geometry, in this case, a cube. To do that, we can simply use the method *CreateBox* and pass it a *name*, a set of *options*, and the respective *scene* (Listing 2.10 line 1). At the bare minimum, the size of the cube must be defined as one of the options. It can either be represented as a number making it the size of each cube side, or it can be set the height, width, and depth individually. As a material, only a simple green color is set (Listing 2.10 line 3), but more complex materials like textures could be used.

```
1        var cube = BABYLON.Mesh.CreateBox("box", 3.0, scene);
2        var myMaterial = new BABYLON.StandardMaterial("myMaterial",
    scene);
3        myMaterial.diffuseColor = new BABYLON.Color3(0, 1, 0);
4        cube.material = myMaterial;
5        cube.position.y = 1;
```

Listing 2.10: Define the cube and its materials

As a final step, a simple rotation is going to be applied, by incrementing the rotation parameter applied to the respective axes (lines 2 and 3 from Listing 2.11), inside the *runRenderLoop* method, which is going to be our animation loop.

```
1         engine.runRenderLoop(function(){
2             cube.rotation.x += 0.005;
3             cube.rotation.y += 0.01;
4             scene.render();
5             });
```

Listing 2.11: Start animation loop

2.1.3 Deck.gl

Deck.gl is a large-scale, WebGl-based data visualization library developed by Uber's visualization team. It has been developed in parallel with some modules:

- Luma.gl, a general WebGL library designed to be able to exchange and use information from both the raw WebGL API, as well as other WebGL libraries.
- React-map-gl, a React wrapper around Mapbox GL, that works very well with Deck.gl.
- Nebula.gl, a high-performance feature editing framework for Deck.gl.

Deck.gl was originally dependent on React, a great framework for making complex web applications, but overwhelming for typical data visualization needs. It now offers a native JavaScript scripting interface that can be used in any JavaScript environment or framework, making it easier for coders to leverage WebGL for interactive visualizations.

This library has a layered approach, which may help when using it with other libraries. The fact that this technology is layer-based helps mixing complex functionalities. For instance, it is possible to draw a map using a tool such as Mapbox GL JS defining it as a first layer an then add data from some sort of data set as a second layer.

Besides rendering images or objects, these layers can also filter, aggregate, interact, and represent abstract data. deck.gl is designed to allow you to take any data with which you can associate positions, and easily render that data on a map using a Deck.gl layer [9].

Another benefit of using Deck.gl is that it supports 64-bit precision floats, which solves one of WebGl's problems of only supporting 32-bit floats, making it impossible to perform tasks such as getting a centimeter precision when getting a location using a GPS. To be able to emulate 64-bit floats, this library adds the data into two 32-bit floats and joins them into a vector2. Also, all linear and non-linear math operations were modified to support 64-bit operations.

In conclusion, the main idea of using Deck.gl is to render a stack of visual overlays, usually (but not always) over maps. In sum, here are a few reasons to use deck.gl [3]:

- Deck.gl is built to handle large data sets
- Allows for various types of projections on top of a map
- It can handle various events, including mouse hover, click and move
- Has various types of different, well tested layers
- The Deck.gl layers, despite being very diverse, are easy to create and customize.

2.1.4 Summary

Three.js large community, numerous dedicated tutorials and learning books, make it a very friendly library for newcomers to 3D web visualization.

Babylon has a similar approach to Three.js. It was originally created for building 3D games but it has now a much wider range of applications. While Three.js has an unstable development cycle with lots of unmaintained libraries, tools, and extensions, Babylon.js has an organized and coherent development supported by Microsoft with the assurance that it remains open-source and free.

As it was shown throughout this section, these two libraries have very similar ways of rendering 3D animations. Babylon.js, being initially intended to create games, has some additional tools such as collision detection and antialiasing. Both Babylon.js and Three.js have a great variety of useful features.

Deck.gl is also a great choice for the ADAS visualization platform that's intended for this project. The framework's layered approach seems very useful when receiving data from several different vehicle sensors with different characteristics. For instance, it has a Point Cloud Layer that takes in points with 3d positions and renders them as spheres with a certain radius (perfect for LIDAR data). It can also render layers on top of each other, for example, it can render a polygon layer on top of a map layer, giving the idea that polygon is following a path on top of the map.

The following Table 2.1 presents some of the different features between Three.js, Babylon.js, and Deck.gl.

2.2 Data sets

In order to develop a web platform that is capable of showing data from an ADAS, we need to get a source of data. In this section two different data-sets

Table 2.1: Framework Comparison [10]

	Modeling	Animation	Integrated Audio	Integrated Networking	Integrated Physics	WebGL version	Licence
Three.js	No	Yes	Yes	No	No	Native (1.0)	MIT
Babylon.js	No	Yes	Yes	No	Yes	Native (1.0/2.0)	Apache License 2.0
Deck.gl	No	Yes	No	No	No	Native (1.0)	MIT

that can provide the needed data will be studied. They are the KITTI and the OXFORD ROBOTCAR data sets.

These data sets are usually composed of a set of files with information from all the data sources available in the car. In this project, we will focus on data sets that have at least GNSS information, one or more cameras and one or more LiDARs, to maintain a standard level of visual information quality for displaying.

2.2.1 KITTI

The KITTI data set was provided by Karlsruhe Institute of Technology and is composed by various sequences captured by driving around the mid-size city of Karlsruhe, in rural areas and on highways [11]. Up to 15 cars and 30 pedestrians are visible per image. This was accomplished by equipping a standard station wagon with two high-resolution color and grayscale video cameras. Accurate ground truth is provided by a Velodyne laser scanner and a GPS localization system [12]. The data was recorded using an eight-core i7 computer equipped with a RAID system, running Ubuntu Linux and a real-time database [13]. The following sensors were used:

- 1 x Inertial Navigation System (GPS/IMU): OXTS RT 3003 [14];
- 1 x Laserscanner: Velodyne HDL-64E [15];
- 2 x Grayscale cameras, 1.4 Megapixels: Point Grey Flea 2 (FL2-14S3M-C) [16];
- 2 x Color cameras, 1.4 Megapixels: Point Grey Flea 2 (FL2-14S3C-C) [17];
- 4 x Varifocal lenses, 4-8 mm: Edmund Optics NT59-917 [18].

The laser scanner spins at 10 frames per second, capturing approximately 100k points per frame. It has a 360-degree horizontal field of view and a 26.9-degree vertical field of view. The vertical resolution has a value of approximate 0.4 degrees, whereas the angular resolution (azimuth) has a value of 0.08 degrees.

The cameras are mounted approximately parallel to the ground plane. Even though the camera images are captured at a resolution of 1384 x 1032, in the

KITTI data set, the images are cropped to a size of 1382 x 512. 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) [13]. Figure 2.1 shows the location and orientation of each sensor on the vehicle.

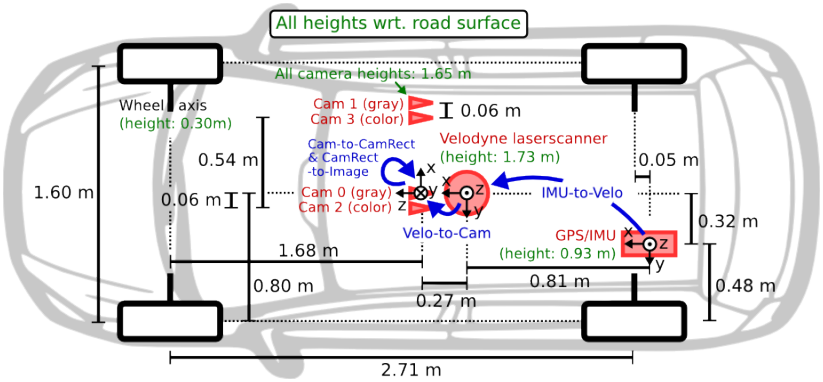


Figure 2.1: Kitti sensor positions on vehicle [13]

2.2.2 OXFORD ROBOTCAR

The Oxford RobotCar data-set was collected by Oxford University's, Oxford Robotics Institute and "contains over 100 repetitions of a consistent route through Oxford, UK, captured for over a year. The dataset captures many different combinations of weather, traffic, and pedestrians, along with longer-term changes such as construction and roadworks" [19]. RobotCar is equipped with the following sensors:

- Cameras:
  - 1 x Point Grey Bumblebee XB3 (BBX3-13S2C-38) trinocular stereo camera, 1280x960x3, 16Hz, 1/3" Sony ICX445 CCD, global shutter, 3.8mm lens, 66°HFoV, 12/24cm baseline [20];
  - 3 x Point Grey Grasshopper2 (GS2-FW-14S5C-C) monocular camera, 1024x1024, 11.1Hz, 2/3" Sony ICX285 CCD, global shutter, 2.67mm fisheye lens (Sunex DSL315B-650-F2.3), 180°HFoV [21].
- LIDAR:
  - 2 x SICK LMS-151 2D LIDAR, 270°FoV, 50Hz, 50m range, 0.5°resolution [22];
  - 1 x SICK LD-MRS 3D LIDAR, 85°HFoV, 3.2°VFoV, 4 planes, 12.5Hz, 50m range, 0.125°resolution [23].

- GPS/INS:
  - 1 x NovAtel SPAN-CPT ALIGN inertial and GPS navigation system, 6 axis, 50Hz, GPS/GLONASS, dual antenna [24].

The Figure 2.2 shows the location and orientation of each sensor on the vehicle.

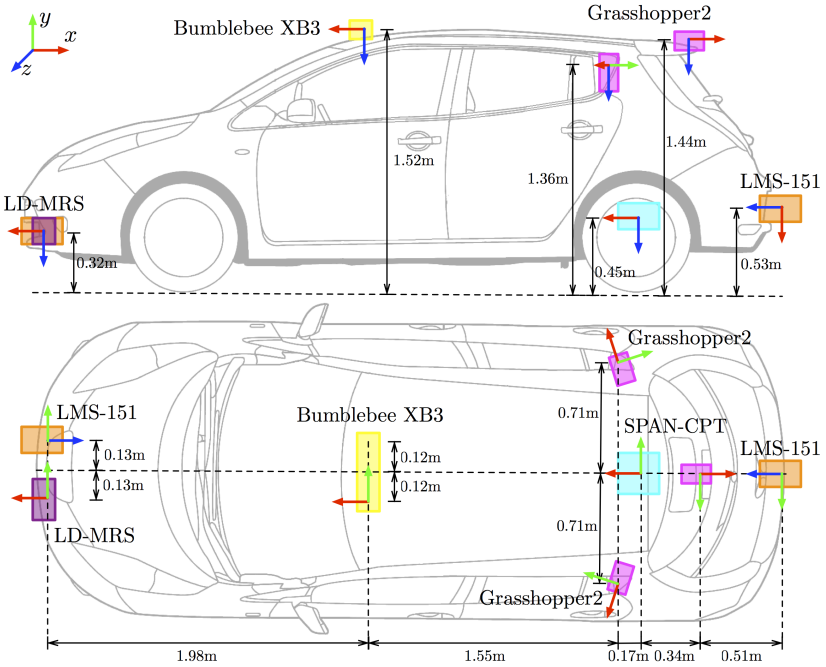


Figure 2.2: Robot car sensor positions on vehicle [25]

2.3 Hadoop Ecosystem

The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers. One of its biggest advantages is the high level of scalability that this ecosystem offers, meaning that it can be composed by a single server up to thousands of machines. On top of all this, the library itself is designed to detect and handle failures at the application layer, which culminates on a highly-available and secure service [26]. The Apache Hadoop project includes three different modules namely, Hadoop Distributed File System (HDFS), Hadoop YARN and Hadoop MapReduce.

2.3.1 HDFS

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. This system allows spreading the storage

of big data across a cluster of computers. HDFS also makes applications available to parallel processing [27].

HDFS has a master/slave architecture as shown in Figure 2.3. The *Name Node* is the commodity hardware that contains the GNU/Linux operating system and the namenode software. It acts as the master and is responsible for managing the file system namespace, regulating client's access to files and executing file system operations such as renaming, closing, and opening files and directories. The *Data Node* is a commodity hardware having the GNU/Linux operating system and datanode software; for every node in a cluster, there will be a datanode. Datanodes perform read-write operations on the file systems. As per client request, they also perform operations such as block creation, deletion, and replication according to the instructions of the namenode [28]. Any machine that supports Java can run the NameNode or the DataNode software.

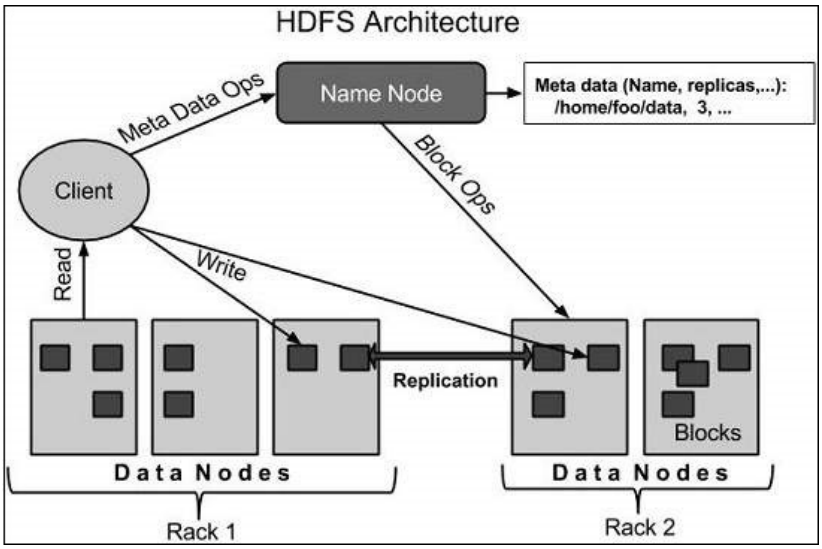


Figure 2.3: HDFS architecture [28]

HDFS is designed to reliably store very large files across machines in a large cluster, it does so by dividing each file in a sequence of blocks. These files are then replicated for fault tolerance, which means if some data is lost in one node it can still be recovered. Both block size and replication factor are configurable per file. For more information about the Hadoop distributed file system, please see paper [26]

2.3.2 YARN

Apache Hadoop YARN is a resource management and job scheduling technology. "The fundamental idea of YARN is to split up the functionalities of resource

management and job scheduling/monitoring into separate daemons. The idea is to have a global ResourceManager (RM) and per-application ApplicationMaster (AM)" [29].

Figure 2.4 illustrates the YARN architecture and its different components. There are two different types of managers in this technology, the *Resource Manager* and the *Node Manager*. The first one arbitrates resources among all the applications in the system, while the later is responsible for containers, monitoring their resource usage (CPU, memory, disk, network) and reporting the same to the ResourceManager/Scheduler. "The per-application ApplicationMaster is, in effect, a framework-specific library and is tasked with negotiating resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the tasks" [29].

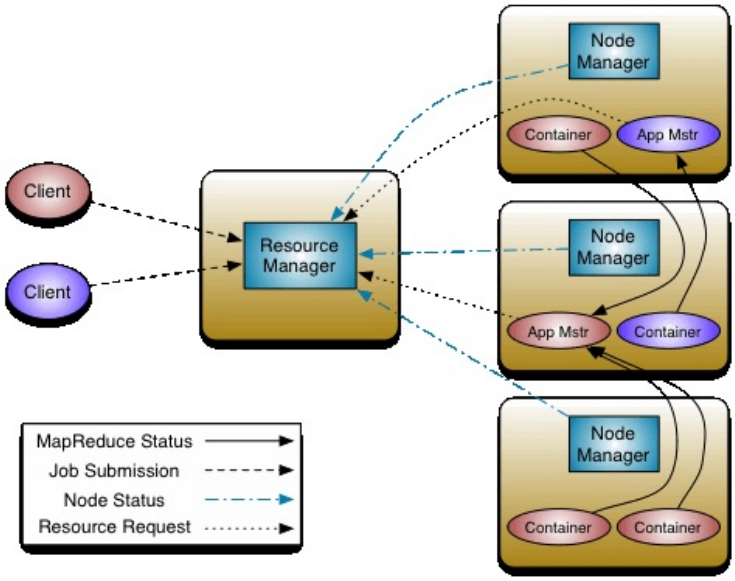


Figure 2.4: YARN Architecture [29]

YARN supports two interesting notions, *resource reservation* via the ReservationSystem and *Federation* via the YARN Federation feature. The first one allows users to specify a profile of resources over-time and temporal restrictions, and reserve resources to ensure the predictable execution of important jobs. The second one allows linking together multiple yarn (sub-)clusters and makes them appear as a single massive cluster. This has the advantage of using independent clusters together making high demanding jobs easier.

2.3.3 MapReduce

Hadoop MapReduce is a software framework that can be used to write applications that process vast amounts of data in-parallel on large clusters of computers. MapReduce has two different types of processing tasks, namely the *map* and the *reduce*. The map task is responsible for the primary processing of chunks that were split from the input data-set, then the framework sorts the outputs of the map task that are then inputted to the reduce tasks [30].

The following example illustrates how MapReduce could be used to count the number of times that the letters A and B are displayed in multiple files (Figure 2.5) and how the different MapReduce tasks would handle that kind of work. In this example, there are two machines in the *map* task whose job is to process two documents each and count how many times the letters A and B were shown. After this task is complete, the letters count is sorted and each machine is given one letter (it could be given to the same machine or other). Then, in the *reduce* task, one machine adds all the A letters and the other adds the B letters.

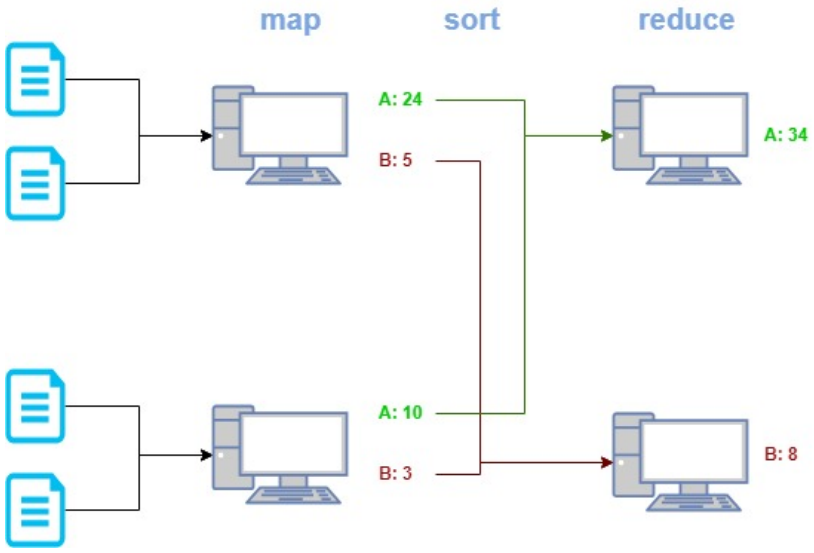


Figure 2.5: MapReduce example

Typically the MapReduce framework and the Hadoop Distributed File System are running on the same set of nodes. This configuration allows the framework to effectively schedule tasks on the nodes where data is already present, resulting in very high aggregate bandwidth across the cluster [31].

2.3.4 Spark

Apache Spark is a Big Data general-purpose computing engine and in-memory framework that aims to process large sets of data in parallel across clusters.

It lets the coder execute real-time and batch work in a scripting manner in a variety of languages with powerful fault tolerance [32], [33]. Spark also relies on a distributed storage system to function, such as *Hadoop Distributed File System (HDFS)* and it also needs a *cluster manager*, such as *Hadoop YARN*. Spark provides a platform that addresses many of MapReduce problems and adds more flexibility in its computation. It allows us to move away from having to break up the tasks into small jobs and also from having to wrangle with the complexity of building solutions on a distributed system development [32].

Spark is very versatile and was designed with the Hadoop ecosystem in mind, meaning that it can work alongside MapReduce or provide an alternative platform for PIG, HIVE and SEARCH to work on top of (Figure 2.6).

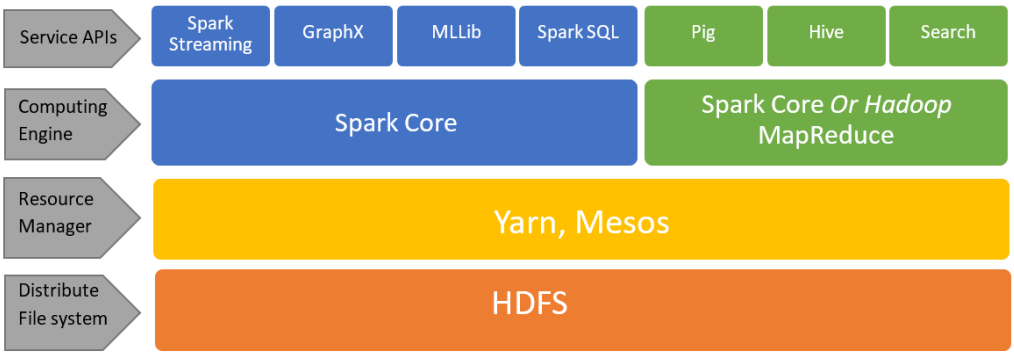


Figure 2.6: Spark context adapted from [32]

Spark has a set of useful APIs such as *Spark Streaming*, which allows for real-time results to be computed by enabling the implementation of ML Lib and Graphx on the live streams, *GraphX*, a very powerful library to handle graph-parallel computation, *ML Lib*, a Library to run machine learning algorithms on large data sets in a native distributed environment and *Spark SQL*, which allows the use of SQL queries to query non-relational distributed databases.

Spark uses a master-slave architecture as shown in Figure 2.7. The Driver acts as the master and is where the main method runs. It converts the program into tasks and then schedules the tasks to the workers. The workers act as the slaves and execute the delegated tasks from the driver. They are launched at the beginning of a Spark application and usually run for the whole life span of an application.

The Driver can communicate with the worker throughout different methods:

- *Broadcast Action*: The Driver sends the necessary data to each worker, this is for data sets under 1 GB of data.

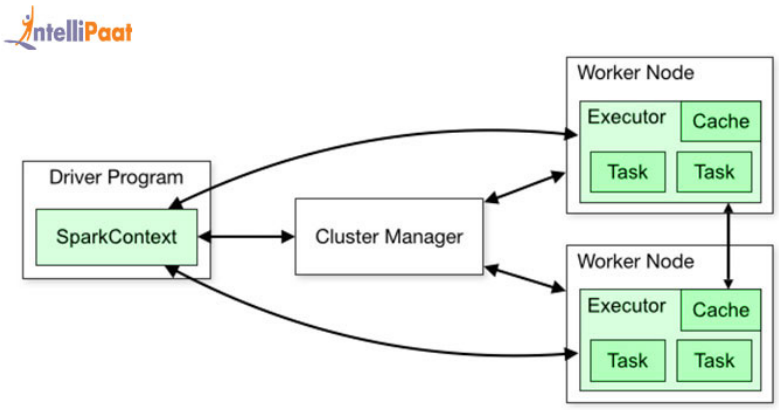


Figure 2.7: Spark architecture [34]

- *Take Action*: The Driver takes data from all Executors. This action can be very expensive and dangerous as the Driver might run out of memory and the network could become overwhelmed.
- *DAG Action*: The Driver transmits control flow logic to the worker. This is the least expensive action out of the three.

2.3.5 HBase

HBase is a type of "NoSQL" database, meaning that the database isn't a relational database management system (RDBMS) that supports SQL as its primary access language. It is column-oriented and runs on top of the Hadoop Distributed File System (HDFS). Each table in HBase contains rows and columns, much like a traditional database. Each table must have an element defined as a Primary Key and all access attempts to HBase tables must use this Primary Key. HBase allows for many attributes to be grouped forming what is known as *Column Family*. The elements of a column family are all stored together, this is different from a row-oriented relational database where all the columns of a given row are stored together. With HBase, one must predefine the table schema and specify the column families [35], [36].

Just like HDFS, HBase has a master-slave architecture. There is a *HMaster* that performs administrative functionalities like creating and deleting tables and is also responsible for monitoring all RegionServer instances in the cluster. The *RegionServers* acts as a slave and is responsible for serving and managing regions, which are the basic element of availability and distribution for tables, and are comprised of a Store per Column Family [37]. A RegionServer runs on a DataNode and hosts the data in the form of files known as *Hfiles*. HBase also

uses *ZooKeeper*, which is part of HDFS, as a distributed coordination service to maintain server state in the cluster. Zookeeper maintains which servers are alive and available, and provides server failure notification [38]. Figure 2.8 illustrates the Hbase architecture and all this different components.

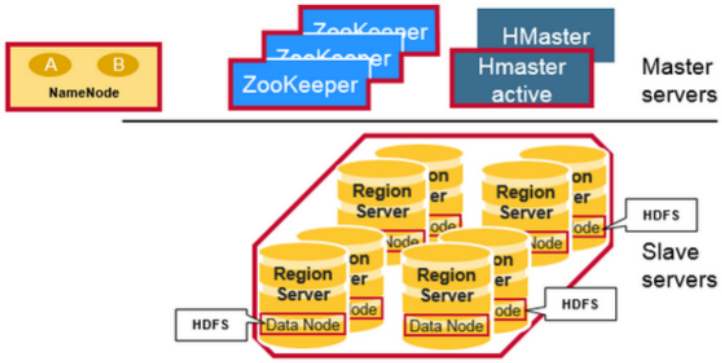


Figure 2.8: HBase architecture [38]

HBase clusters expand by adding RegionServers that are hosted on commodity class servers. It has many features which support both linear and modular scaling, such as [37]:

- Strongly consistent reads/writes: This makes it very suitable for tasks such as high-speed counter aggregation.
- Automatic sharding: HBase tables are distributed on the cluster via regions, and regions are automatically split and re-distributed as your data grows.
- Automatic RegionServer failover.
- Hadoop/HDFS Integration: HBase supports HDFS out of the box as its distributed file system.
- MapReduce: HBase supports massively parallelized processing via MapReduce for using HBase as both source and sink.
- Java Client API: HBase supports an easy to use Java API for programmatic access.
- Thrift/REST API: HBase also supports Thrift and REST for non-Java front-ends.
- Block Cache and Bloom Filters: HBase supports a Block Cache and Bloom Filters for high volume query optimization.

- Operational Management: HBase provides built-in web-pages for operational insight as well as JMX metrics.

Another interesting concept in HBase is the *meta table*. The *meta table* is used to find the RegionServers that are serving a particular row range of interest. With the help of these tables, the client can contact directly the Region Server without the need to make read or write requests going through the master. To avoid going through this process multiple times, this information is cached in client. If a region is reassigned, the client will requery the catalog tables to determine the location of the new region [37].

2.3.6 Hive

Apache Hive is a data warehouse software project built on top of Apache Hadoop. It provides an SQL dialect, called Hive Query Language (abbreviated HiveQL or just HQL), for querying data stored in a Hadoop cluster [39]. Mapping SQL operations to the low-level MapReduce Java API can be very challenging, that's when Hive comes to place. "Hive translates most queries to MapReduce jobs, thereby exploiting the scalability of Hadoop while presenting a familiar SQL abstraction" [40].

Hive is not a full database since its build on top of Hadoop; both Hadoop and HDFS impose limits on what it can do. The biggest limitation is that Hive does not provide a record-level update, insert, nor delete. Another problem to keep in mind is that queries that would finish in seconds for a traditional database take longer for Hive. That is because Hive queries have higher latency, due to the start-up overhead for MapReduce jobs [40]; therefore Hive is most suited for data warehouse applications where fast response times are not required and data is not changing rapidly.

Figure 2.9 shows the major components of Hive and its interactions with Hadoop; they are the *UI*, *Driver*, *Compiler*, *Metastore* and *Execution Engine*. Their purpose is the following [41]:

- *UI* : The user interface for users to submit queries and other operations to the system.
- *Driver* - The component which receives the queries. This component implements the notion of session handles and provides execute and fetch APIs modeled on JDBC/ODBC interfaces.
- *Compiler* : The component that parses the query, does semantic analysis on the different query blocks and query expressions and eventually generates an execution plan with the help of the table and partition metadata looked up from the metastore.

- *Metastore* : The component that stores all the structure information of the various tables and partitions in the warehouse including column and column type information, the serializers and deserializers necessary to read and write data and the corresponding HDFS files where the data is stored.
- *Execution Engine* : The component which executes the execution plan created by the compiler. The plan is a DAG of stages. The execution engine manages the dependencies between these different stages of the plan and executes these stages on the appropriate system components.

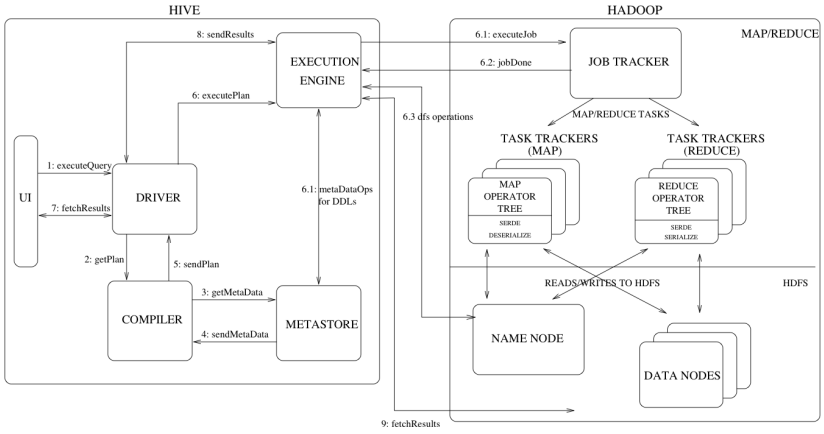


Figure 2.9: Hive architecture [41]

2.3.7 Summary

The Hadoop ecosystem has a vast variety of great tools that allows for the distributed processing of large data sets across clusters of computers using simple programming models, which is ideal to store large amounts of data. In this section, we studied various tools such as HDFS, YARN, MapReduce, Spark, HBase, and Hive but there are a lot more. Comparing all these technologies would be incorrect because they all have different functionalities. Keeping this in mind we could however compare them according to their usability.

In terms of tools for Big Data processing, MapReduce and Spark Core are both extremely important. The major advantage of Spark Core over MapReduce is that it is easy to scale data processing over multiple computing nodes. On the other hand, Apache Spark offers high-speed computing, agility, and relative ease of use. All these characteristics are perfect complements to MapReduce, in fact, it can be said that they have a symbiotic relationship [42]. In other words, these tools can work together and is probably the best way to do it. But since they can both be used as stand-alone tools, we will show some of the key differences between them.

In terms of speed, MapReduce strictly disk-based approach makes it slower than Spark that, not only can use a disk for processing, but also can use RAM. This brings up the next point, which is cost. Since Spark uses a large amount of RAM, it also makes it more expensive than MapReduce. Another interesting factor is that MapReduce can only be used for batch processing. However, besides batch processing, Spark can also be used for real-time data processing. All these considerations and more can be seen in Table 2.2.

Table 2.2: MapReduce vs Apache Spark [42]

	MapReduce	Apache Spark
Data Processing	Only for Batch Processing	Batch Processing as well as Real Time Data Processing
Processing Speed	Slower than Apache Spark because if I/O disk latency	100x faster in memory and 10x faster while running on disk
Category	Data Processing Engine	Data Analytics Engine
Costs	Less Costlier comparing Apache Spark	More Costlier because of a large amount of RAM
Scalability	Both are Scalable limited to 1000 Nodes in Single Cluster	Both are Scalable limited to 1000 Nodes in Single Cluster
Machine Learning	MapReduce is more compatible with Apache Mahout while integrating with Machine Learning	Apache Spark have inbuilt API's to Machine Learning
Compatibility	Majorly compatible with all the data sources and file formats	Apache Spark can integrate with all data sources and file formats supported by Hadoop cluster
Security	MapReduce framework is more secure compared to Apache Spark	Security Feature in Apache Spark is more evolving and getting matured
Scheduler	Dependent on external Scheduler	Apache Spark has own scheduler
Fault Tolerance	Uses replication for fault Tolerance	Apache Spark uses RDD and other data storage models for Fault Tolerance
Ease of Use	MapReduce is bit complex comparing Apache Spark because of JAVA APIs	Apache Spark is easier to use because of Rich APIs
Duplicate Elimination	MapReduce do not support this features	Apache Spark process every records exactly once hence eliminates duplication.
Language Support	Primary Language is Java but languages like C, C++, Ruby, Python, Perl, Groovy is also supported	Apache Spark Supports Java, Scala, Python and R
Latency	Very High Latency	Much faster comparing with MapReduce Framework
Complexity	Difficult to write and debugcodes	Easy to write and debug
Apache Community	Open Source Framework for processing data	Open Source Framework for processing data at a higher speed
Coding	More Lines of Code	Lesser lines of Code
Interactive Mode	Not Interactive	Interactive
Infrastructure	Commodity Hardware's	Mid to High-level Hardware's
SQL	Supports through Hive Query Language	Supports through Spark SQL

When it comes to HBase and Hive, we can say that they are two Hadoop based big data technologies that serve different purposes. "Hive and HBase are both data stores for storing unstructured data. HBase is a NoSQL database used for real-time data streaming whereas Hive is not ideally a database but a mapreduce based SQL engine that runs on top of hadoop" [43]. In other words Hive is a query engine that transforms database operations into mapreduce jobs, whereas HBase is a data storage particularly for unstructured data where operations are run in

real-time on the database. Apache Hive is mainly used for batch processing but HBase is mainly used for transactional processing wherein the response time of the query is not highly interactive.

Hive should be used for data warehousing requirements and when the programmers do not want to write complex MapReduce code, whereas HBase is an ideal big data solution if the application requires random read or random write operations or both. If the application requires to access some data in real-time then it can be stored in a NoSQL database [43].

Both Hive and HBase have their limitations, but they complement each other in various ways, in fact they are commonly used together on the same Hadoop cluster. It is possible to write HiveQL queries over HBase tables so that HBase can make the best use of Hive's grammar and parser, query execution engine, query planner, etc.

2.4 Encoding Tools

This project aims to work with data that can have high storage demand, such as the point clouds. Since the premise of the application is to use web technologies, it is extremely important to store all the data as efficiently as possible. For these reasons, we studied some compression tools to help saving space. A brief overview of two mesh and point cloud compression libraries, namely Draco and Corto, will be presented in this section.

2.4.1 Draco

Draco is a library for compressing and decompressing 3D geometric meshes and point clouds. This library's premise is to improve the storage and transmission of 3D graphics since it was designed for compression efficiency and speed. It supports various features such as compressing points, connectivity information, texture coordinates, color information, normals, and any other generic attributes associated with geometry [44]. For mesh compression, Draco uses the Edgebreaker algorithm which divides the mesh compression into various steps. "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" [45].

Draco can read .obj or .ply files as an input and output Draco-encoded files, or the other way around. One can encode point cloud data with draco_encoder by specifying the point_cloud parameter. If you specify the point_cloud parameter

with a mesh input file, draco_encoder will ignore the connectivity data and encode the positions from the mesh file.

Draco also possesses a Java Script decoder to make it possible to integrate the decoding process in a web application. To use the decoder, the first step is to create an instance of *DracoDecoderModule*, which will be used to create the *DecoderBuffer* and *Decoder* objects. The encoded data is supposed to be set in the *DecoderBuffer*, then one can identify the type of geometry (mesh or point cloud) using *GetEncodedGeometryType()*. Lastly, to return a mesh or a point cloud, simply call either *DecodeBufferToMesh()* or *DecodeBufferToPointCloud()*.

2.4.2 Corto

Just like Draco, Corto is also a library for compression and decompression of meshes and point clouds and is based on the compression algorithm developed for the Nexus project for the creation and visualization of multi-resolution models [46]. The encoding process visits one triangle of a mesh at a time and maintains a list of the edges of the processed region's boundary. This region's boundary keeps growing as new triangles are processed. The way this region grows is by adding the three edges of the first triangle to a list, then "iteratively one edge is being extracted from the list and the algorithm encodes the relation of the not-yet-visited triangle incident to the edge with respect to the boundary of the already encoded region" [45].

Corto can convert .ply and .obj into .crt, which is this library specific compressed format. To do that, simply run the command *corto* followed by a set of options and the path to the input file. The existing options are all available in Corto's website and are the following [46]:

- -o <output>: filename of the .crt compressed file. If not specified the extension of the input file will be replaced.
- -e <key=value>: add an exif property, or more than one.
- -p : treat the input as a point cloud.
- -v <bits>: vertex bits quantization. If not specified an euristic is used
- -n <bits>: normal bits quantization. Default 10.
- -c <bits>: color bits quantization. Default 6.
- -u <bits>: texture coordinate bits. Default 10.
- -q <step>: quantization step unit (float) instead of bits for vertex coordinates

- -N <prediction>: normal prediction can be ”delta” which uses the difference from previous normal (fastest), estimated, which uses the difference from compute normals (cheaper) or border, stores the difference only for boundary vertices(cheapest)
- -A : compute and add normals to the model.
- -P <file.ply>: decompress and save as .ply for debugging purposes.

Just like Draco, Corto also has a decoding library for JavaScript. In this case, it decodes a .crt as an ArrayBuffer and returns an object with attributes (positions, index, colors, etc). To have access to all these attributes, in the JavaScript file, simply load the compressed .crt using, for instance, *XMLHttpRequest()*, and then pass the response to the *CortoDecoder* constructor to create an instance of the decoder. Then by calling the method *decode*, one can have access to the attributes referenced above.

Chapter 3

Project development

This project, a web platform capable of showing data from an ADAS, will be based on the architecture represented by Figure 3.1. The user interacts with the application using any device capable of using a web browser, the app then acts as a client and sends a request to the server to get the data needed for visualization. The server is responsible for answering the request by making queries to the data lake and retrieving the correct data, which will then be passed to the client. What makes this architecture stand out from other client-server web architectures is the inclusion of ROS on the storage side. Our ROS nodes convert the datasets content into a format that can be interpreted by Deck.gl.

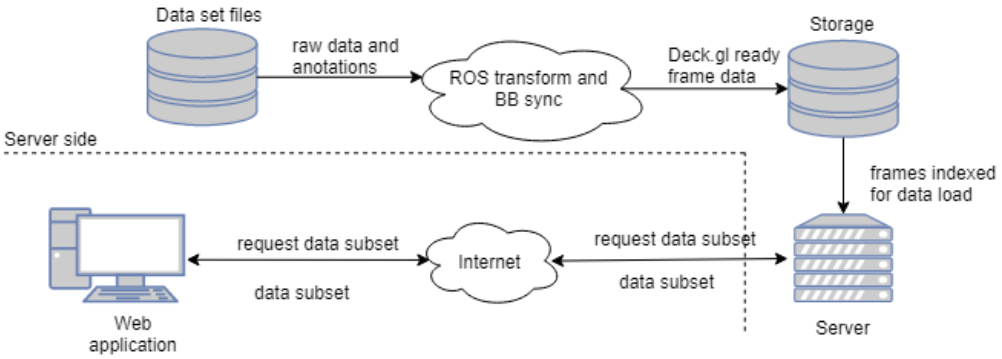


Figure 3.1: Project architecture

3.1 Visualization client

This component of the application is what represents everything related to rendering images and objects, plotting graphics and developing a user interface. The goal was to develop an application capable of rendering data in different layers, that could be manipulated separately. This was one of the reasons we

chose Deck.gl as our main development tool: not only it is built on top of WebGL which uses the GPU to easily render polygons and images while having a lower learning curve, but also has a layer approach, meaning that it can render different types of data, such as point clouds, maps, and obstacles on top of each other. It is worth mentioning that Deck.gl uses JSON format to represent its data structures so that the different types of layers can later be rendered. This means our data had to be converted into this format beforehand.

For plotting data, it was necessary to use another framework that can aid in this process. The tool chosen for this task is called Chart.js.

For designing the layout, we used used CSS/HTML to position elements and change their attributes to create a layout that is responsive to screen sizes and adjustments. Some additional tools, like bootstrap, were also used to enhance visual presentation.

3.1.1 Deck.gl

The first step when using Deck.gl is to instantiate a Deck.gl object. If the MapBox map is required, a key and an initial geographical location on the map can be added to the object, as shown in the code below [47].

```
new deck.DeckGL({
    mapboxApiAccessToken: '<your_token_here>',
    mapStyle: 'mapbox://styles/mapbox/light-v9',
    longitude: -122.45, //initial map longitude
    latitude: 37.8, //initial map latitude
    zoom: 12, //initial map zoom
});
```

The Deck.gl object is where all the layers, views, callbacks and event listeners can be added. To do so, after the object has been instantiated, we use a method called setProps().

Five different layers were created to maximize customization the point cloud layer, which uses a specific Deck.gl layer object whose sole purpose is to render point clouds, the ego layer for drawing the origin vehicle on the map, the bounding boxes layers for rendering all the other cars and obstacles on the road (these last two were both drawn using the polygon layer object), the trajectory layer for drawing the ego vehicle trajectory line using the path layer object and, lastly, the map layer, created when instantiating the Deck.gl object (the main object) and using mapbox as the base map.

3.1.2 Layer properties

The Layer class is the base class of all Deck.gl layers and it provides several base properties available in all layers. Some of the properties that were used in

this application and are worth mention are the following [48]:

id : The id is a string and must be unique among all layers at a given time. If more than one instance of a specific layer exists, they must possess different id strings for deck.gl to properly distinguish them.

data : deck.gl layers expect a variety of different types of data such as a JavaScript array of data objects, an object that implements the iterable protocol [49], any non-iterable object that contains a length field, a String with an URL pointing to the data source and a Promise.

visible : "Whether the layer is visible. Under most circumstances, using visible prop to control the visibility of layers is recommended over doing conditional. rendering" [50].

opacity : The opacity of the layer.

pickable : Whether the layer responds to mouse pointer picking events.

onHover : This callback will be called when the mouse enters/leaves an object of its deck.gl respective layer. It has two parameters, info and event, and requires pickable to be true for it to work.

onClick : This callback will be called when the mouse clicks over a Deck.gl layer object. It has two parameters, info and event, and requires pickable to be true for it to work.

coordinateSystem : Specifies how layer positions and offsets should be geographically interpreted. The default is to interpret positions as latitude and longitude.

coordinateOrigin : Required when the coordinateSystem is set to COORDINATE_SYSTEM.METER_OFFSETS and specifies a longitude and a latitude from which meter offsets are calculated.

modelMatrix : An optional 4x4 matrix that is multiplied into the affine projection matrices, it allows local coordinate system transformations to be applied to a layer.

3.1.3 Load data algorithm

After the web page has been loaded, to load the visualization data, the user must first click on the button "load data". This action starts an event that, among some initializations, executes the method that makes the data request to the server. If the server responds successfully, the data is sent back to the HTML page and is stored in the browser's memory. The page can then use that data to start the animation algorithm as shown in Figure 3.2.

To get the data from the server, i the JavaScript file responsible for loading the data to memory is used the fetch API, which provides a JavaScript interface

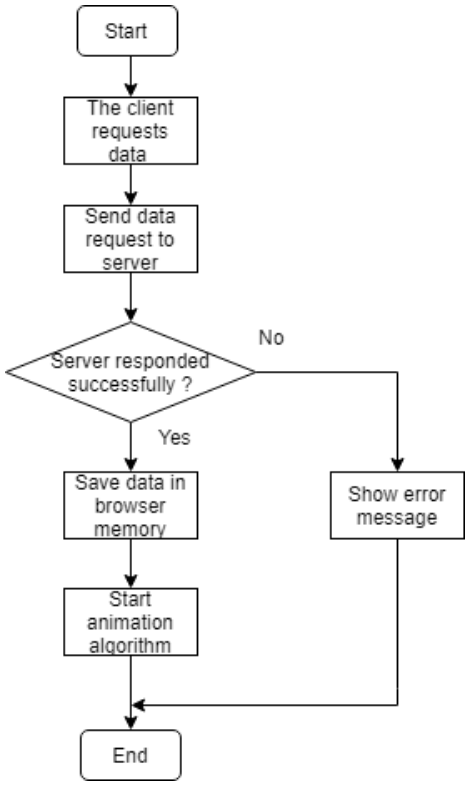


Figure 3.2: Data load flow chart

for accessing and manipulating parts of the HTTP pipeline, such as requests and responses [51].

3.1.4 Animation algorithm

The animation algorithm starts when the user clicks on the play button of the GUI after the data has started loading. The application goes through each frame that is stored into the browser memory and, in each iteration, updates all the Deck.gl layers, views and chart data.

The number of frames per second (FPS) of the animation will dictate the time between frames, for instance, if the animation is running at 10fps then a new frame will be animated every 100ms. To do so, a javaScript method called setTimeout() is used, which executes a block of code after a time interval defined by the programmer (in milliseconds).

Figure 3.3 illustrates the animation algorithm flow chart. This flow chart represents the computing logic that occurs in each frame iteration.

As stated above, the setTimeout() method will execute a set of code after a time interval in milliseconds. In each interval the program tests if the number

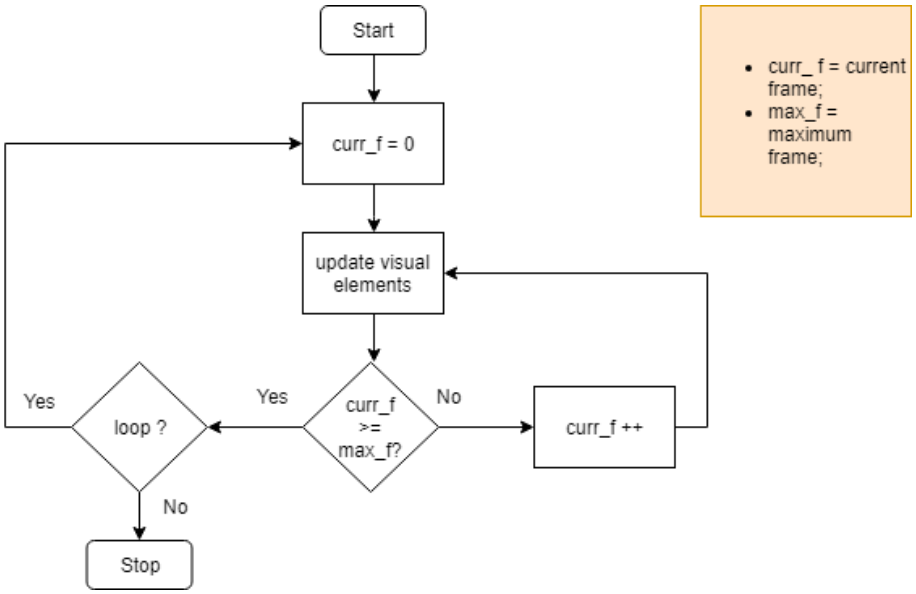


Figure 3.3: Animation flow without user inputs

of the current frame is greater than the max frame. If not the current frame is incremented, the progress bar width is updated depending on the percentage of frames that were already played (curr_f/max_f), the views and the layers are updated with the information of the new current frame and, lastly, the palatable information (for instance acceleration and velocity) that the user has chosen to plot in the GUI is added to the active charts. On the other hand, if the current frame is greater or equal than the max frame then the Boolean value of the loop flag is tested. If this flag has a false value, it means that the application is not set to run on loop and therefore the animation stops. If the loop flag is true, then the application is set to run on loop and the current frame is reset (the current frame value becomes 0) and all the chart data is deleted so it can be plotted again.

3.2 Data Structure

Deck.gl uses JSON format to represent its data structures. For this reason, we stored the data needed to draw the origin vehicle and the surrounding objects as JSON files. For the naming of these files, we concatenated the name of the category that the file represents with the respective animation frame. This means that the data for all the cars in frame 0 would be called ”car0”.

There are four distinct JSON file structures, each representing different elements. Two of these types of files have the information for drawing polygons and plotting vehicle-related data, while the other two are configuration files that represent metadata from the sequence to be loaded.

All the files that represent the origin vehicle information have their name starting with ego, followed by the number of the frame it represents. They contain information, on the geographical location of the vehicle (latitude and longitude) and several elements for plotting, such as accelerations and velocities. Bellow is the data structure of the origin vehicle JSON file (Listing 3.1):

```
{
    "lat":49.011213,
    "long":8.422885,
    "alt":112.834923,
    "roll":0.022447,
    "pitch":0.000010,
    "yaw":-1.221910,
    "vx":3.514768,
    "vy":0.037625,
    "vz":-0.038789,
    "ax":-0.305810,
    "ay":-0.196357,
    "az":9.994213,
    "wx":-0.017499,
    "wy":0.021393,
    "wz":0.145630
}
```

Listing 3.1: Origin vehicle file structure

The surrounding objects have different categories but each has the same data structure; the more noticeable difference between them is in the file names. Each file as the name of its category followed by the name of the frame it represents, this way, despite having the same structure, each file is easily identified. In terms of structure, each file is an array of objects that represent a different element of the same category in each frame (for instance one or more cars). These objects have a unique id, five points that represent the base of the polygon (the fifth point creates an edge that illustrates were the vehicle is pointing to) and a height, for extruding the base and create the polygon. All this information is visible in the snippet of Listing 3.2:

```
[
    {
        "id":1,
        "polygon":[
            [4.8597,-8.28416,-1.7511],
            [5.60203,-9.90772,-1.75904],
            [5.26865,-10.1701,-1.76148],
            [4.85211,-10.2506,-1.76303],
            [4.10978,-8.62702,-1.75509]
        ],
        "height":1.73906
    }
]
```

Listing 3.2: Category files data structure

There are two types of configuration files. The first one is an object containing all the sequences available where each element contains the sequence name as well as the path for the data files. This file is intended to be loaded as soon as the HTML page loads and has the structure shown in Listing 3.3:

```
 {
    "seq05":{
        "sequenceName":"Kitti sequence 5",
        "sequencePath": "seq05/",
        "sequenceTable":"inserttest14"
    },
    "seq14":{
        "sequenceName":"Kitti sequence 14",
        "sequencePath": "seq14/",
        "sequenceTable":""
    }
}
```

Listing 3.3: Directories configuration file structure

The second configuration file contains information about each sequence. It has the information on how many categories and point clouds there are in that sequence as well as their names. It also has other important information such as the camera feeds available, the number of frames in the sequence and the label of the data that can be plotted. Listing 3.4 illustrates the structure described above:

```
{
    "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 3.4: Sequence configuration file structure

3.3 Server and data lake

Since the beginning of the project, the idea was to create a remote server that could provide information stored in a data lake. There was a need to store large amounts of data largely due to the size of the point clouds, which can have millions of points. It was decided that the best solution would be to configure a Hadoop cluster to store all this data because it is a distributed environment.

The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers. One of its biggest advantages is the high level of scalability that this ecosystem offers, meaning that it can be composed by a single server up to thousands of machines. On top of all this, the library itself is designed to detect and handle failures at the application layer, which culminates on a highly-available and secure service [26].

Inside the Hadoop ecosystem there are a couple of services that are mandatory for the cluster to run, such as the Hadoop Distributed File System (HDFS), but there are also different types of software that run on top of Hadoop whose sole purpose is to manage the cluster and or the storage. For managing reading and writing of our large data-sets, we chose Apache Hive since it has a good amount of community support and provides an SQL dialect called Hive Query Language (abbreviated HiveQL or just HQL), making it more accessible. Although being able to write SQL queries to access our tables in the cluster is certainly much more intuitive, this kind of approach has some drawbacks. ”Hive translates most queries to MapReduce jobs, thereby exploiting the scalability of Hadoop” [40]. This creates a start-up overhead, making Hive queries have higher latency and therefore needing more time to execute a query when compared to traditional

databases. Some tests are still in progress but for now, the time that Hive takes to read data from a table, is satisfactory.

3.3.1 Cloudera

To test the Hadoop services, we needed to create a Hadoop environment to deploy our data. To do that, we used the Cloudera QuickStart virtual machines (VM), which provide everything necessary to try the Hadoop ecosystem. This VMs not only come pre-installed with most Hadoop services but also have Cloudera Manager which allows the user to easily manage all the clusters services, including instances, health, logs, etc.

In other words, Cloudera provides a scalable, flexible, integrated platform that makes it easy to manage rapidly increasing volumes and varieties of data. Cloudera provides the following products and tools [52]:

- Cloudera Distributed Hadoop (CDH) : Cloudera's 100% open-source platform distribution, including Apache Hadoop and built specifically to meet enterprise demands.
- Apache Impala : A SQL engine that provides fast, interactive SQL queries directly on Apache Hadoop data stored in HDFS, HBase, or the Amazon Simple Storage Service (S3).
- Cloudera Search : Cloudera Search provides easy, natural language access to data stored in or ingested into Hadoop, HBase, or cloud storage without requiring SQL or programming skills.
- Cloudera Manager : Cloudera Manager is an end-to-end application to deploy, manage, monitor, and diagnose issues with CDH deployments.
- Cloudera Navigator : Cloudera Navigator Data Management is an end-to-end complete solution for data governance, auditing, security and related data management tasks that are fully integrated with the Cloudera Distributed Hadoop platform.

From all the products described above, the most used was certainly Cloudera Manager since it allowed to monitor, start and stop all the Hadoop services as well as their instances. Figure 3.4 represents some of the features of this application.

Hue is another Cloudera application that was used and is extremely helpful. It is a query editor that can be used to create, write or read tables. This application has a useful user interface that allows not only to visually analyze each table and database created, but also to chose from various editors to interact with the

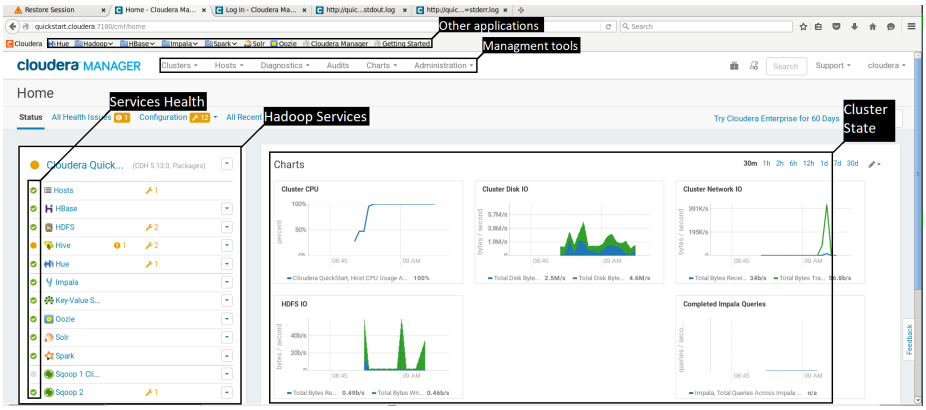


Figure 3.4: Cloudera manager annotated

installed Hadoop services. With Hue, we were able to create various tables to test Hive reading and writing.

Note that this Cloudera virtual machine was only used for test and academic purposes which means that in the future, were the project progress into production, it would be necessary to create our Hadoop environment to deploy data.

3.3.2 Client APIs

For creating the script that can query the Hive table inside the cluster and send that data to the visualization client, we choose Python. It is a programming language with immense online support and is widely used for creating servers. With all this in mind, the next step was to choose a Python client library that could connect to the cluster. The first library that was tested is called PyHive, however, this option was abandoned shortly after due to compatibility issues with some Python versions. The next idea was to use pyspark, which is a Python interpreter for spark. This method has several drawbacks: first, the Python script has to be inside the cluster since spark is an Hadoop service; second, since the script has to be inside the cluster it would have been necessary to create a communication channel, using sockets, for instance, to send the data over another client that would receive and parse this data outside the cluster; lastly, the time for reading from the table was not satisfactory since the first read request, from the client to the server, would often take from thirty to thirty six seconds.

With further research we found another Python library called pyHS2 that, despite having good reading times, it is no longer being maintained, which could lead to unsolvable issues in a later stage of the project.

Currently, impyla, a Python client for HiveServer2 implementations (e.g., Impala, Hive) for distributed query engines is being used. This is the best option

so far, it has Python interface to query the Hive tables (using HiveServer2), it can be used outside the cluster and has satisfactory reading times. Table 3.1 acts as a summary of what has been stated about these libraries. It is worth mentioning that the reading times were measured using a table filled with dummy content for test purposes only and not the actual data that we are trying to implement on the web application. Those tools referenced above were all tested using the same table.

Table 3.1: Hive reading times

Library	Average time (s)	Availability
PyHive	-	Has some compatibility issues with certain versions of Python
pyspark	31.672	Has slow reading times when making the first request, needs to be inside the cluster
pyHS2	1.2186	Good reading times, but it is deprecated
impyla	1.1165	Good reading times, actively supported

Despite impyla being a great option, there was a problem with using Hive. Even though Hive has a binary format and it is relatively simple to input binary data into Hive, querying this kind of data is not that straightforward. To insert binary data into a Hive table, one must load the intended file from the local file system into the Hadoop Distributed File System (HDFS). To do so we used the command line in Listing 3.5. To create a table for which we will store the data, we just need to create a column with the data type as binary and use the command "load data local inpath", as shown in Listing 3.6

```
1    hdfs dfs -put /src_binary_file /dst_binary_file
```

Listing 3.5: Insert data from the local file system into HDFS

```
1    CREATE TABLE table1(
2      rowid INT,
3      mydata BINARY);
4
5    load data local inpath '/path_to_binary_file' into
6    table table1;
```

Listing 3.6: Load binary file into Hive table

We were able to store a test image inside the Hive table by doing this procedure, however, if we tried to retrieve that same image using a select clause, the Hue editor would print various errors. The first measure taken to overcome these errors was to create various tables stored as different formats to understand if that was the cause of the error. The formats tested were Text File, Sequence File, RC File, AVRO File, ORC File and Parquet File. All these types gave the same errors except Text File that would print special characters when querying

in the Hue editor, which was expected since we were saving a table containing an image as a Text format.

Considering all these problems with Hive, we tried a different approach. Since we only needed the Hadoop cluster for the distributed storage, and there wasn't a need to process and rearrange the data inside the cluster, we considered storing our data files directly into HDFS as an option.

To retrieve the data stored inside HDFS we used webHDFS, which is based on the HTTP REST API [53]. Node.js has a webHDFS client library and, since our initial script for serving the web application and the local files was already written in node, this library seemed to be a great option. To create this file server that fetches data from HDFS, it was created an HTTP server using node. Then every time the client made a request, the node server would parse the file path from the request URL and use the webHDFS library to access that path. The HTTP server would then receive the data from HDFS and send that response to the web visualization client. The diagram in Figure 3.5 represents all this process.

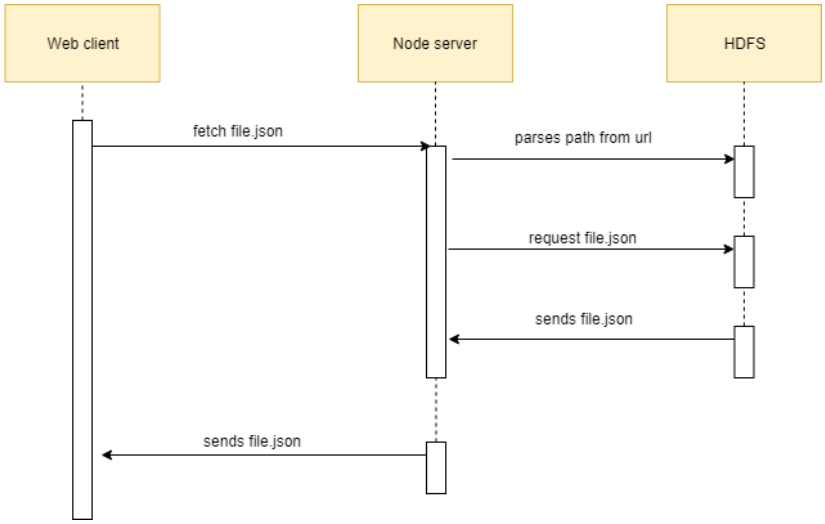


Figure 3.5: Getting HDFS files over webHDFS

3.4 Point cloud decompression

One of the biggest challenges of storing data into memory was the size of the point cloud. The first time the point cloud was loaded into the browser's memory the application stopped working due to memory overflow. The initial solution that we came up with was to reduce the density of the point cloud by eliminating the points of the ground, but even so the memory was almost at the limit.

It was obvious that we needed to compress the point cloud but, to progress with the web application development while compression methods were being studied, we created a fast encode method to reduce the point cloud size. This encoding consists in storing each point in a float instead of a list with x, y and z coordinates. In this format , x is the integer part of the float, y is defined by the first six digits of the decimal part and z is defined by the five last digits. Note that for y and z, the first digit represents the signal (0 is positive and 1 is negative). Figure 3.6 illustrates an example of this procedure. We chose these decimal values because the maximum decimal digits that JavaScript accepts without rounding the number is eleven. With this approach, we were able to save large amounts of space making the application running without issues. Note that this method was only meant to be used temporarily; one of its biggest drawbacks is that it lowers the point cloud resolution to centimeters. This happens because, if we used a millimeter resolution with the available decimal places, we would have a maximum number of 99999 for y and 9999 for z, that would correspond to a maximum value of 99.999 meters and 9.999 meters respectively. Since y is the transverse range of the laser and since the laser used for this data sets has a range of 120 meters, we could be excluding points of the point cloud. For this reason, we used a centimeter precision, that gives us 999.99 meter range in y. Even though a centimeter precision is not ideal, we considered that it is still acceptable for visualization purposes.

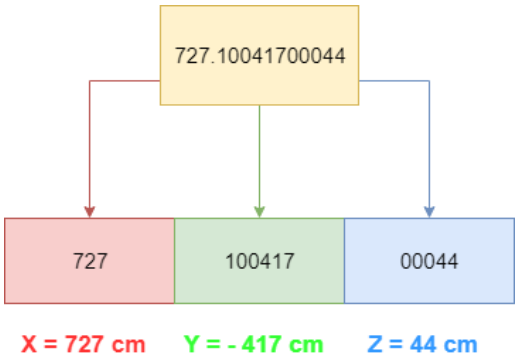


Figure 3.6: Float encode

Since the solution presented above was very implementation-specific and thought as a temporary one, there was a clear need for using a compression library. We decided to use Draco which ”is a library for compressing and decompressing 3D geometric meshes and point clouds. It is intended to improve the storage and transmission of 3D graphics” [44]. With this library we were able to compress the complete point cloud so that, when loaded, the browser’s memory wouldn’t overflow.

After being compressed, the point cloud needs to be decoded with the JavaScript

Draco decoder to use its data on the visualization client. This tool can decode meshes and point clouds previously encoded with the Draco encoder. There are two decoders available, one that uses WebAssembly and other that do not. The first step when decoding the point cloud is to detect whether or not the browser has WebAssembly support because, when it does, it generally has a superior decoding performance. Then the necessary files are loaded accordingly and the decoder module is created.

The first approach for decoding the point cloud consisted in pre-decoding it while frames were being loaded to memory. This process consists of using the decoder module to create a buffer array from the encoded data, get its geometry type (point cloud or mesh) and decode it accordingly. The output is then stored as a DracoFloat32Array in the device's memory while the browser's memory only stores a pointer to that array of binary data. This prevents the browser's memory from overflowing but by doing it, the point cloud is still consuming memory space on the device. This means if this were to be applied in mobile hardware with low specifications would most likely crash the application. Finally, the point cloud is decoded point by point back to browser's memory when the Deck.gl instance calls for each frame. The idea behind the pre-decode was to speed the decoding process when the frames were being animated. If the application had to fully decode the point cloud at each frame it would slow down the animation.

As expected, this solution worked very well for the desktop version however, when implemented in a low-end mobile device, the application wouldn't work well due to insufficient memory resources. This happened because, as stated above, when the data was in a pre-decoded state it would still consume memory, the only difference is that it wouldn't occupy the browser's memory. To overcome this problem we removed the pre-decode, meaning that all the data would be stored in browser's memory in an encoded format and only would be decoded in each frame animation when needed. This solved the memory problem but in return lowered the performance of the application. Since the decoding process is condensed in one step it also needs more time, which in return slows down the animation. This wasn't very noticeable on the desktop application, with only occurring a few slowdowns occuring occasionally but, on the mobile version, the drop in performance was evident. Figure 3.7 illustrates the differences between memory management with and without pre-decode.

We are making tests with web workers in the current state of the application to possibly improve the overall performance, both in the mobile and in the desktop versions. "A web worker is a JavaScript program running on a different thread, in parallel with main thread" [54]. The browser creates one thread per tab and each main thread can spawn an unlimited number of web workers until the user's system resources are fully consumed. As soon as a browser tab is closed, the main thread 'dies' and 'kills' any web workers it created. The JavaScript main

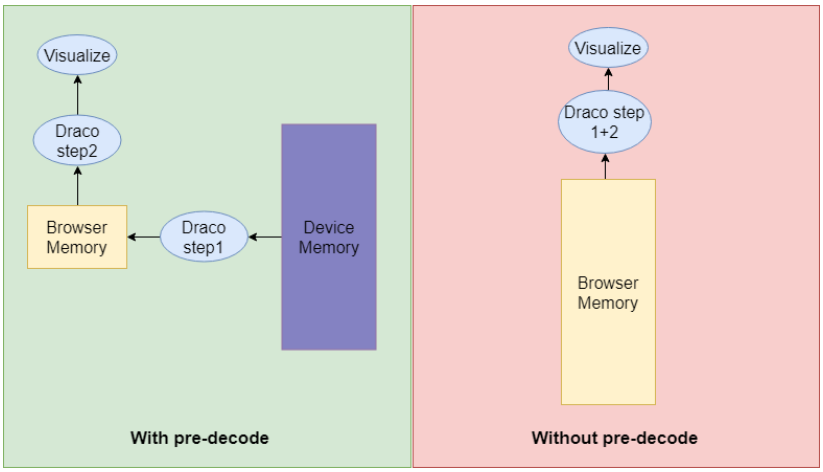


Figure 3.7: Memory management with and without pre-decode, note the difference in browser memory size which is the most scarce resource

thread communicates with its workers through message events and can ”kill” any of these workers that it created. A worker can also ”kill” itself if needed. Having this in mind, the idea was to spawn a worker on the animation and, in each frame, send the corresponding compressed point cloud, so that the worker could use Draco to decode it. Using this method, we can exploit the device multi-threading to improve animation performance. While the main thread is focused on rendering all the visual elements, the worker is in charge of decoding the point cloud. Figure 3.8 illustrates a side by side comparison between the animation with and without workers.

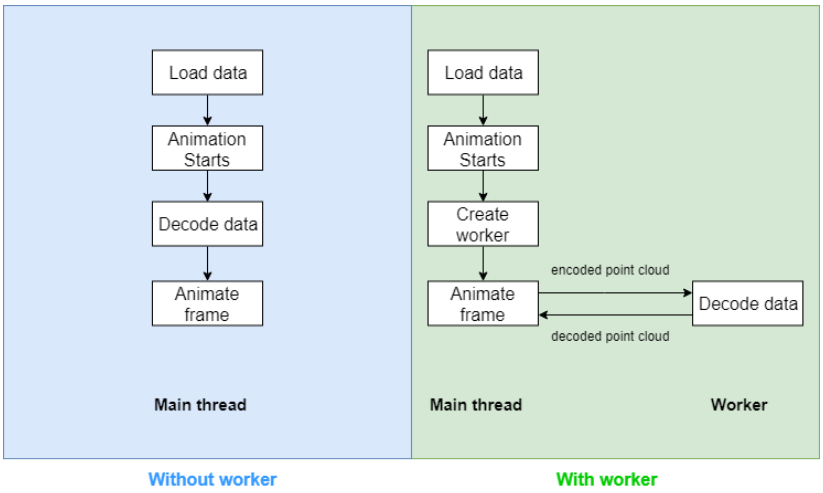


Figure 3.8: Animation with and without worker

Chapter 4

Results analysis

In this chapter, the working application with all its features will be shown. Each layout element will be explained, and all its functionalities will be analyzed sequentially to show how to use the application features in the correct order. Also, other important performance aspects such as the time necessary to animate a frame, the time to decompress the point cloud and the time the server takes to complete the client requests, will be analyzed .

4.1 Desktop web application

When the user loads the HTML page, a small window pops up with a step by step tutorial on how to use the application. The user can follow this tutorial until the end, or simply close the window by clicking on x symbol on the right corner of the header. The user can open this window any time by clicking on the question mark symbol on the top left corner of the side navigation bar. The tutorial described above can be seen in Figure 4.1.

The layout of the application is composed of an animation control bar, which controls the playback of all data and a navigation bar on the left corner that can be hidden and has three different tabs. The user is also able to create floating windows for plotting data or showing various camera feeds. The different categories are identified with bounding boxes with different colors and sizes. Figure 4.2 illustrates all the features referenced above as well as six different categories, namely the origin vehicle, the cyclist, the van, the car and the trajectory of the origin vehicle. We can also see all points of the point cloud. All of these categories are deck.gl layers, each vehicle is drawn using the polygon layer, although the origin vehicle has some slight parameter differences. The point cloud is drawn using the point cloud layer and the trajectory is drawn using the path layer. Note that the annotations indicating the category names in the figure were all edited and are not a feature of the application.

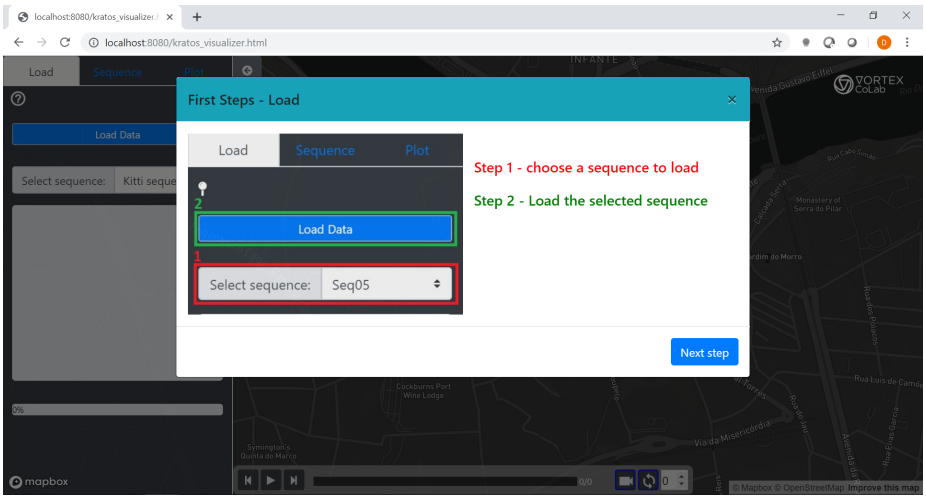


Figure 4.1: Application tutorial

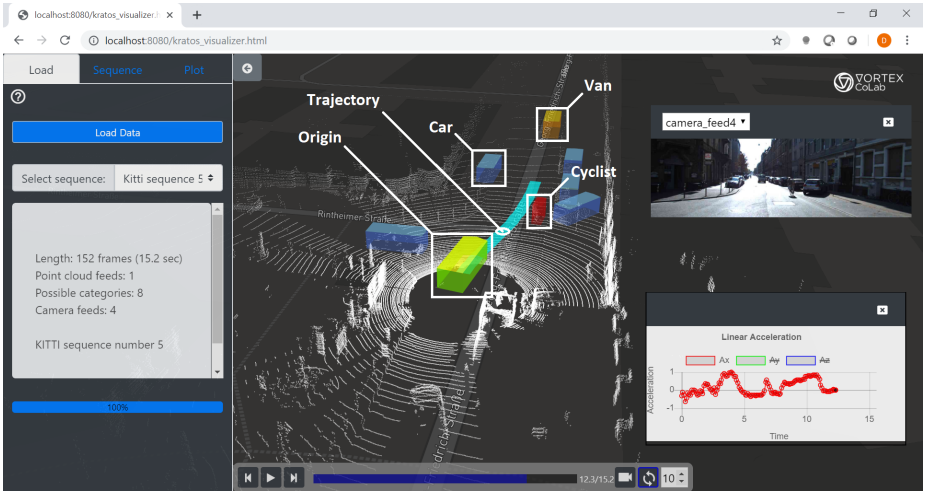


Figure 4.2: Screenshot of the application features with white text boxes to highlight the different categories

By default, the sidebar is visible and the load tab is selected. With the load tab (Figure 4.6.a), the user can choose one of the available pre-captured sequences, to load them in the browser's memory so that they can be animated. It has a button to load the selected data, a drop-down to select which sequence to load, a text field for showing some information about the chosen sequence, and a progress bar for indicating how much of the sequence has been loaded.

To manipulate visual information from the selected sequence, we created a separated tab called the sequence tab (Figure 4.6.b). Here the user can create new floating windows containing the camera feeds, and choose between all the

feeds using a drop down menu(Figure 4.3), or change layer properties such as visibility, color, and opacity using a tree-structured menu where the parent is a layer of the sequence and the children are visual options associated with that layer(Figure 4.4).

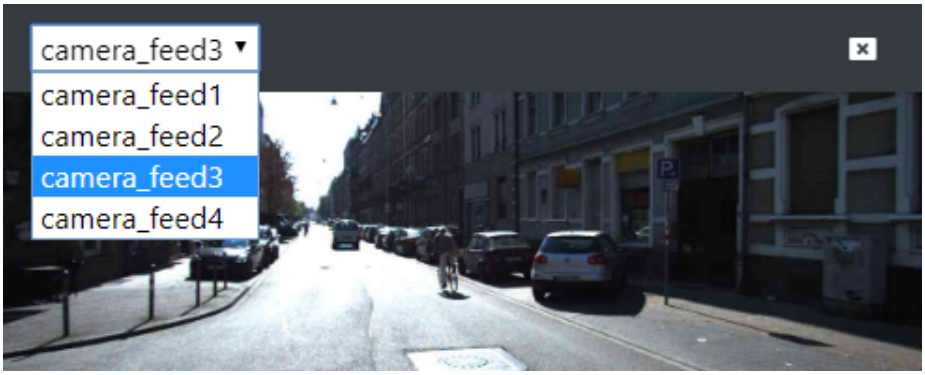


Figure 4.3: Camera feed

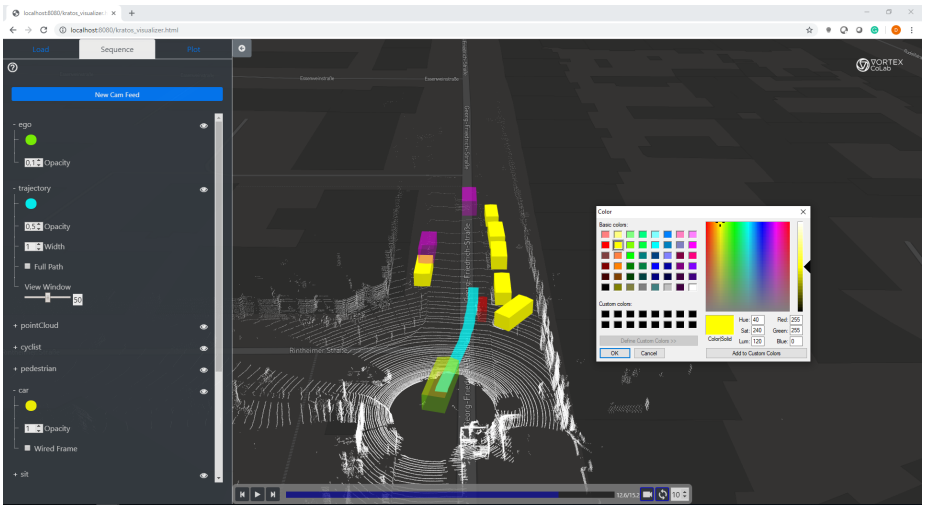


Figure 4.4: Layer customization

The third and last tab is the plot tab (Figure 4.6.c), as the name states this is where the user can create charts with data coming from the origin vehicle. Here, a group of pre-selected variables such as velocities and accelerations can be plotted. There is also a custom option that allows the user to select the variables that he would like to plot, as well as their colors and labels. When the user is done choosing the elements to plot, he can press the ”Create Plot” button to generate a floating re-sizeable window with the chosen information(Figure 4.5).

To fully control the playback of all data, we created a playback bar very

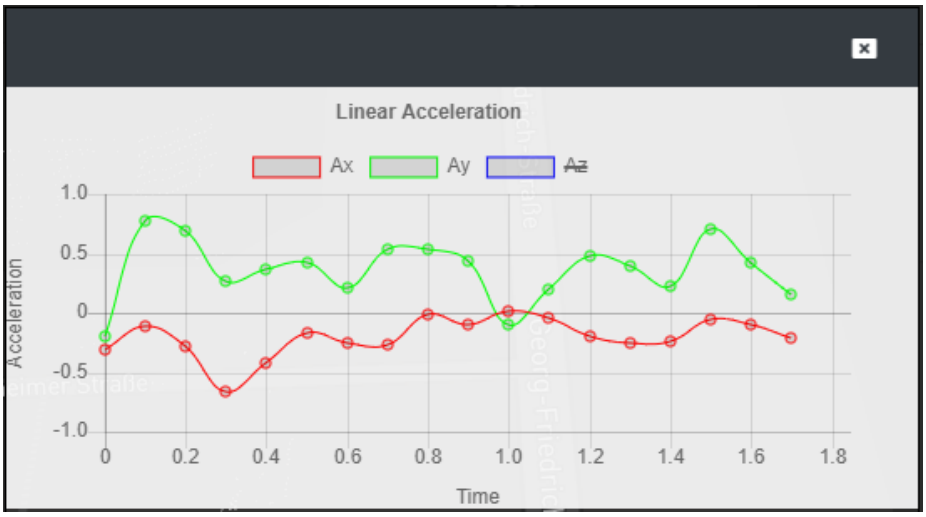
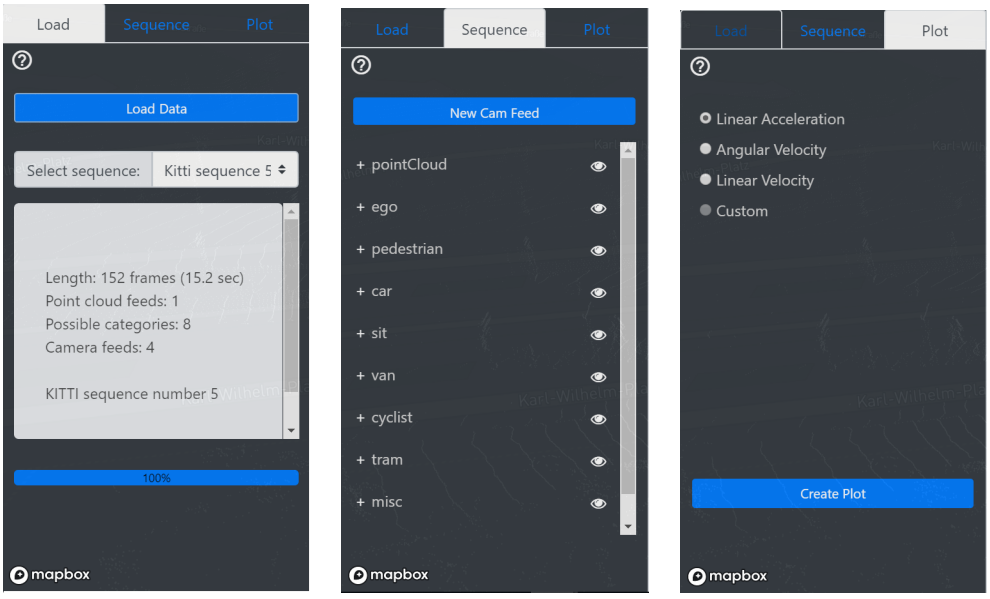


Figure 4.5: Plot window



a) **b)** **c)**

Figure 4.6: Side bar tabs: **a)** load tab; **b)** sequence tab; **c)** plot tab

similar to the ones available in media players. This bar is visible in Figure 4.7 and can start and stop the frame animation, run the animation step by step, jump back or forward using the timeline of the animation and choose from three different camera modes.

When it comes to the three-camera modes available, we developed them with different use cases in mind:

- The ”free move” allows for a more detailed analysis, that’s because in this mode the map doesn’t move with the animation, this means that regardless the current frame in the animation, the map stays static until the user moves it;
- The the ”follow locked” is thought for a temporal overview of the sequence, with a common fixed ”third person view” approach;
- the follow free move allows a temporal analysis from a specific point of view, useful for keeping focus on some specific perspective over the ego vehicle (origin vehicle);

New camera modes can be added to accommodate new specific needs along the project.

Figure 4.7: Playback bar

4.2 Mobile web application

For the mobile application, the idea was to reduce the number of features because of the limitations of mobile devices. The most evident is the lack of right and middle click, as well as the inability to drag and drop, which makes most of the functionalities of mapbox unusable. For this reason, we felt the need to create a completely different layout from the desktop version.

Since the idea of the mobile version is to be used only for visualization and not for analysis, most of the controllable elements of the layers were removed. In this version, the user can only toggle the visibility of the layer, all the other attributes, such as the color and opacity, are predefined. The user can no longer create a new window for displaying the camera feed nor the graphical plots, instead, there are three predefined plots on the sidebar (linear acceleration, linear and angular velocity) and the camera feeds now appear on the playback bar. For changing the different camera feeds it was added, on the playback bar, a new button that changes the feed every time its pressed. This button loops through all the different feeds and has a number that indicates which one is selected. New icons were also added on the bottom of the sidebar that represents the origin vehicle velocity, acceleration, and yaw. All these feature are visible in Figure 4.8.

In terms of camera perspectives, in this mode, there is only the ”follow locked” version. This means that the camera is always following the vehicle in an ”over

the shoulder" perspective and the map is always locked. This emphasizes even more the idea of using this version only as a visualization tool.

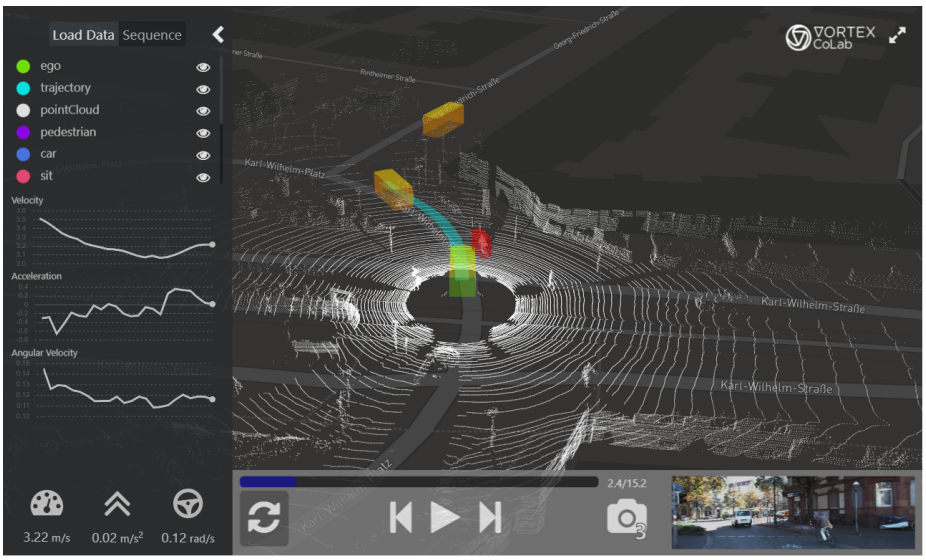


Figure 4.8: Web application mobile version

4.3 Metrics

In this kind of application, it is very important to keep track of the time that the different processes take to execute. For this reason, several metrics were collected, such as the time required to animate all the visual elements in each frame, the time to decode the point cloud and the time to load the data. To perform all these tests, we used a sequence from KITTI data-set as a benchmark, containing 153 frames, 4 camera feeds with a resolution of 1392 x 512 pixels and the object information of 9 Cars, 3 Vans, 2 Pedestrians and 1 Cyclist scattered throughout the sequence. All this data takes about 57.5 MB for the point clouds (compressed), 20.4 MB for the camera images and 0.156 MB for the JSON files (bounding boxes and origin vehicle) for a total of 78.06 MB, which means that since this sequence has 153 frames, these values translate to roughly 0.510 MB per frame (Figure 4.9).

4.3.1 Loading times

To enhance user experience it is important to keep track of the time needed to load data, this way it is possible to determine which values are acceptable and optimize the process. In Table 4.1 we can see fifteen iterations of data being loaded locally from the JSON files and the binary files (point clouds and images). The highest value in these measurements was about 8.1 seconds, while

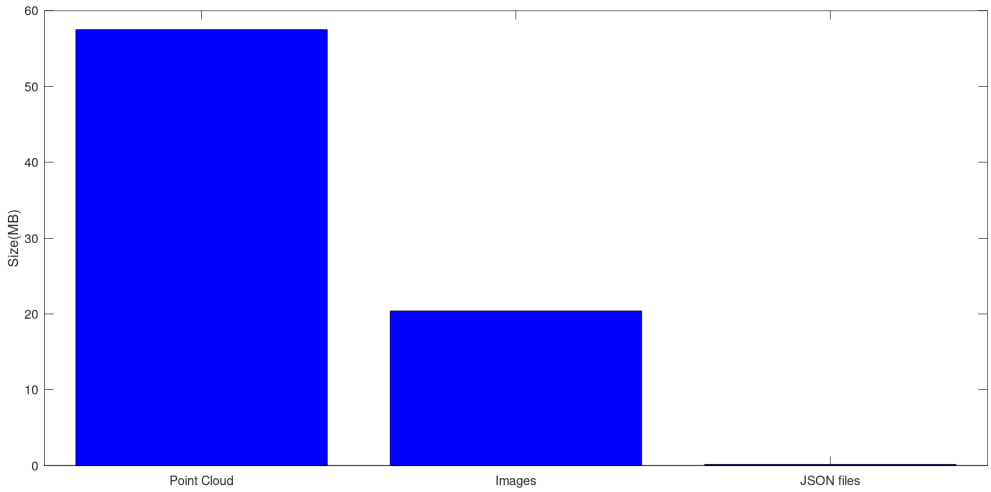


Figure 4.9: File sizes

the smallest was about 3.7 seconds. In this test, all the data was being loaded in from the local machine, so it is worth mentioning that these values will very likely grow when transitioning for a remote server. Also, these values were taken with the point cloud pre-decode method.

Table 4.1: Loading times for all the data in the sequence

load runs (s)	8.03	7.42	4.10	7.53	7.58	5.92	3.68	3.81	5.38	7.97	8.13	7.77	7.30	5.77	3.82
Average (s)	6.28														

After migrating from the pre-decode solution, to fully decode the point cloud when animating, we expected the loading time to lower, since we no longer had the initial overhead caused by the pre-decode. Table 4.2 illustrate the new loading values and they are in fact lower than the previous ones.

Table 4.2: Loading times without pre-decode

Load runs (s)	4.72	4.82	4.73	4.72	4.71	4.74	4.89	4.67	4.68	4.69	4.69	4.71	4.75	4.71	4.65
Average (s)	4.73														

After deploying all of the test sequence files into HDFS and implementing the communication of the web client with this file system, other loading tests were made. Table 4.3 illustrates fifteen load iteration with all the files coming from HDFS, installed in a Cloudera virtual machine. As we can see, it takes a considerable large amount of time to load all the data, compared with the previous test. This HDFS implementation is still in a very premature phase, which means that there is always room for some optimizations, but there could be a vast number of reasons for this kind of result. The first one has to do with the available resources. The Cloudera virtual machine is installed in a mid-

ranged computer that has limited resources and, on top of that, it has to share them with the operating system, and other programs and services. This puts the Hadoop service's constantly in a "bad health" state. If the data was deployed in a cluster specialized for this kind of services, there is a chance of increasing reading performance. Another aspect to keep in mind is that, when a large number of fetches is made in a row some browsers do not behave well. Managing how often fetching is done could improve performance both in reading files locally and from the virtual machine.

Table 4.3: Loading times with HDFS as storage

load runs (s)	24.41	24.33	24.53	23.75	22.91	22.06	20.84	23.12	22.98	21.18	23.64	21.34	20.39	22.84	24.13
Average (s)	22.83														

4.3.2 Animation times

The time to animate each frame and the time to decode the point cloud are related, since the sum of these two can not exceed the time between frames otherwise the animation wouldn't run smoothly. The number of frames per second (FPS) of the animation will dictate the time between frames, for instance, if the animation is running at 10fps, then each frame will be animated every 100 ms. Having this in mind, Figure 4.10 illustrates the time to update the visual elements in each frame. As we can see the average time to animate all the visual elements of a frame is roughly 35.6 milliseconds, which is less than fifty milliseconds, giving a decent margin to decode the point cloud since the frames in our animation are being animated every 100 ms.

4.3.3 Decoding and encoding times

The analyses of the encoding tools were done by another member of the project and in this section, it will be shown the results obtained by him. Table 4.4 illustrates the differences between Draco and Corto in various scenarios, such as encoding with different precisions (qp), with or without position and intensity (pos and int). The first row of the table represents the original point cloud for comparison. The second column illustrates the time needed for the encoding and the last three are related to the size of the point cloud. The third one illustrates the average size of the point cloud in bytes, whereas the forth represents it as a percentage. Lastly, the final column represents the compression Ratio.

Both Corto and Draco have very similar results, although Draco seems to have the upper head in the same circumstances, not only that but, at the time these tests were done, Draco seemed to be more actively supported.

In the initial attempt of using Draco, the process of decoding the point cloud was divided into two steps, a pre-decode that occurs when the data is being loaded

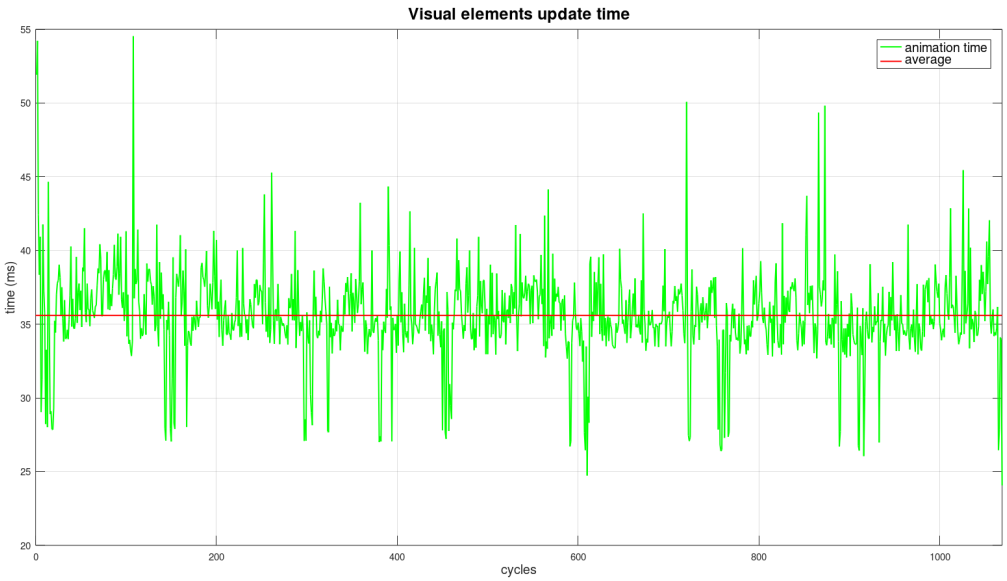


Figure 4.10: Time to update visual elements

Table 4.4: Encoding tests for Draco and Corto

	Time to encode (ms)	Size		
		Average (bytes)	Percentage	Compression Ratio
draco pos+int (original)	NA	1,937,907	100%	1.00
corto pos (qp = auto)	36	433,172	22.35%	4.47
draco pos+int (qp= 14, qg = 8)	52	289,293	14,93%	6.70
corto pos (qp = 14)	36	240,280	12.40%	8.07
draco pos (qp = 14)	41	186,326	9.61%	10.40
draco pos+int (qp =11 , qg = 8)	52	158,932	8.20%	12.19
corto pos (qp = 11)	30	84,706	4.37%	22.8

and the final decode that occurs when each frame is being animated. Figure 4.11 shows the time that both these processes take to execute, the average value being approximately 25.6 ms when data is being load and 14.6 ms in each frame. In this section, the most relevant is the one in the top of the figure, since it represents the time to decode the point cloud during the animation.

After the pre-decode method was deprecated, there were taken new values for the time to decode each frame. In this case, since there is no pre-decode, the only place where Draco decoder is being used is when each frame is being animated. As we can see by analyzing Figure 4.12 the time to decode the point cloud in each frame is approximately 40.8 ms which is very close to the sum of 40.2 ms that results from the two average times in Figure 4.11 40.2 ms.

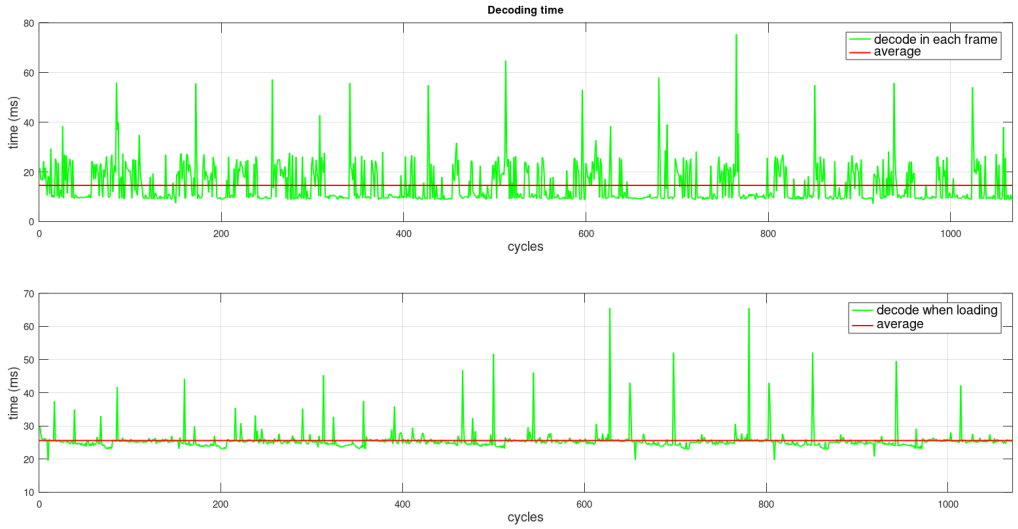


Figure 4.11: Decoding times: **top)** decode time in each frame; **bottom)** decode time when loading data

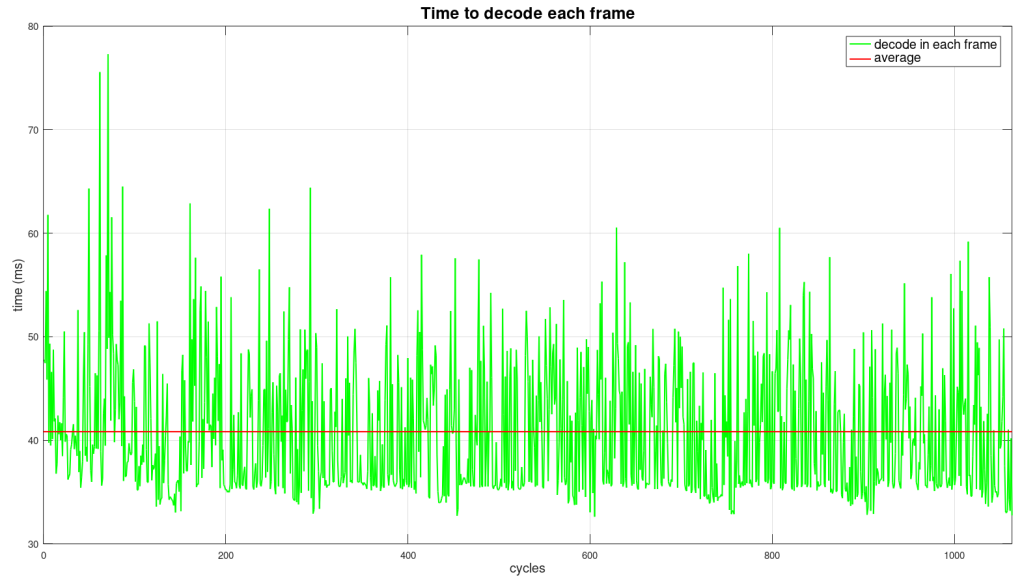


Figure 4.12: Time to decode each frame without pre-decode

4.4 Hive reading times

To better understand which of the python tools referenced in section 3.3 is the better option, there were taken several reads from a hive table filled with dummy values. This test table is structured according to what we consider to be a good format to store the layer's JSON data, in other words, it has three columns each

containing strings. The first one is the category key, this is what identifies what category in each frame the data is associated to (for instance car, van, cyclist), the second is the data and in this column we store the JSON data in a string format so that it can later be parsed into an object on the client-side and lastly the frame id which identifies the corresponding frame of all these elements. It is worth mention that we made use of a hive feature called partitioning, which uses one of the columns to store the data in separated partitions. This is helpful because if we queried the table for a specific element in a specific frame it would only go through the data in the corresponding partition when searching for the correct frame. This makes the reading process much faster in large tables because the queries don't have to look in the full table for a specific frame, but instead, it would search only in the partition that has that frame id value.

Having all the information above in mind, Figure 4.13 illustrates the difference in reading times between the python tools tested.

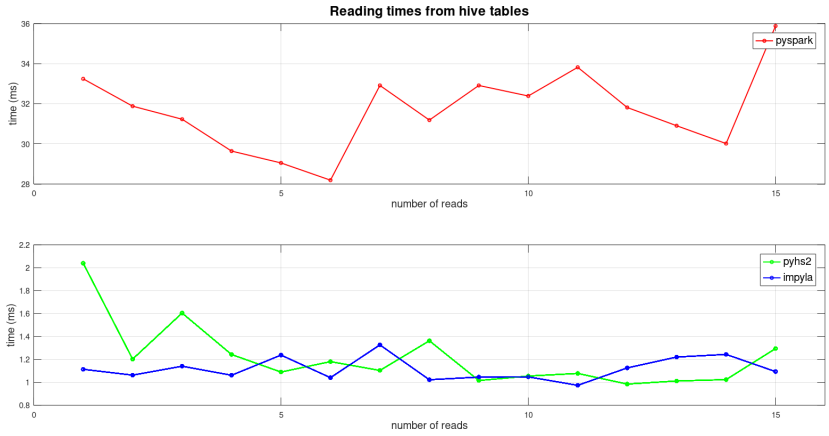


Figure 4.13: Reading times from hive tables

As referenced in section 3.3, the average reading time for pyspark is 31.672 seconds, for pyHS2 is 1.2186 and for impyla is 1.1165. By analysing the Figure, we can confirm that impyla is the fastest option, not only that, but is also actively supported, unlike pyHS2 that was deprecated. Note that our biggest concern is in reading time not in writing time, since the data is supposed to be written only once and not have multiple writings throughout the use of the application. Also note that as already stated above, these tables are filled with just test values and are not near the size of the real data, they are just used to compare the performance of these python library's and should not be mistaken for the time it takes to load all the JSON files. These test values take about 2.21 KB of space, which is only a fraction of the real data. Furthermore, these values were ploted into two separated graphics because, as we can see by analysing Figure 4.13,

pyspark has much higher reading times than the other two python libraries. If these values were ploted in the same graphic, than we wouldn't be able to visually analyse the differences between the three, because the difference in scale would be extremely high. This reinforces the idea that pyspark is not a great option for this specific implementation.

Chapter 5

Conclusions and further work

Companies have to develop their unique visualization platform to support the development and visualization of data logs from autonomous systems. The premise of this project is to change this kind of mindset, providing a generic visualization platform, that can load logged data from different sources in an easily configurable format. The fact that this application is web-based allows for various teams spread across the world to analyze data from these autonomous systems, without the need for a complex setup.

Deck.gl, the framework that is the base of our application, is so far capable of handling the project needs. Not only is it build on top of WebGL which uses the GPU to easily render polygons and images, while having a lower learning curve, but also has a layered approach, meaning that it can render different types of data, such as point clouds, maps, and obstacles on top of each other.

We consider that our goal is progressively being achieved, even though we have encountered various obstacles along the lines. One of them was the size of the point clouds, this kind of data needs a considerable amount of space and, since this application is web-based, memory is an important and limited resource. Having this in mind, it is extremely important to compress the point cloud, so all the data can be loaded into memory without any kind of loss in precision.

As stated throughout this dissertation, it is very important to keep track of the time some tasks take to execute. For instance, the time to animate each frame and the time to decode the point cloud are related, since the sum of these two can not exceed the time between frames otherwise the animation wouldn't run smoothly. We concluded that this objective was achieved when implementing the point cloud pre-decode method stated in section 3.4, however when using the full decode version, the animation would some times appear to slow down, because the time necessary to decode the point cloud and structure the data to provide it to the deck.gl instance would take longer than the time between frames. These changes were necessary, since as shown in section 3.4, the pre-decode version

would consume the device memory, making the mobile version almost unusable. We are currently running tests with web workers as a mean to decode the point cloud to optimize both the mobile and desktop version.

In terms of server software and data lake, we considered that Hive was a good choice, due to its resemblance to traditional databases and its SQL interpreter. Impyla, the python module that was chosen for writing the client-side script, was also satisfactory, it gave us the lowest reading time in our tests. Although Hive is, in fact, a good option for numeric and string information, we came to realize that this is not the case when it comes to binary data. We could not effectively query binary data from Hive so, as an alternative, we used HDFS to store all the files. To create both a reading and writing interface, it was used webHDFS and despite the implementation being successful, the results were far worse, making reading times taking almost two times longer than before. The HDFS implementation certainly needs some revisions due to the excess of time it takes, but it is a solid option to store the data.

In conclusion, we considered that the user interface is intuitive and has the features needed to promote a good data visualization, configuration, and analysis, even though the project still needs some optimizations, especially in loading data.

As future work, we intend to include visual object analysis by clicking, such as the distance between two objects. In terms of graphical plotting, there is going to be added the option to export the graphics that are already being drawn as well as being able to plot data from the surrounding objects (cars, cyclists, pedestrians, etc) if available. Also, at some point in the project, we considered changing the framework that is being used to draw the graphics, to a more lower level one so that we could have more freedom in the way we plot graphics and the data is presented.

The web workers seem to be a promising tool and, as such, it is of our most interest to continue to explore this feature. Lastly, some implementations are already being made to create a live mode, using web sockets. This last feature, at the time of writing this document, is in its early stages, due to the nature of the data that is being transmitted. We will most likely need to create our protocol at the application layer, this is because even though TCP ensures the order of the files sent, we can't guarantee that the files from the same frame will be sent sequentially through the TCP connection, some files are faster to process than other. This could create a case where the source sends all the JSON files from frame one and before sending the point cloud from the same frame it could already be sending the JSON files from frame two. This means that to effectively transfer the data via web socket, we would have to insert an identifier in each file (both the binary and the JSON files) so that in the client-side we could interpret which frame each file represents.

Lastly, it was highly intended, at some point in the development of the project, to publish at least a paper at a conference. We did manage to submit a paper in Springer's conference "ROBOT'2019" that got accepted and will be presented and published after the writing of this document.

References

[1] C. Urmson *et al.*, "Autonomous driving in urban environments: Boss and the urban challenge," *Journal of Field Robotics Special Issue on the 2007 DARPA Urban Challenge, Part I*, vol. 25, pp. 425–466, June 2008. [cited on p. 1]

[2] A. C. MADRIGAL, "Inside waymo's secret world for training self-driving cars," 2017. Available in `https://www.theatlantic.com/technology/archive/2017/08/inside-waymos-secret-testing-and-simulation-facilities/537648/`, accessed in June 2019. [cited on p. 1]

[3] UBER, *Introduction.* Available in `https://deck.gl/#/documentation/overview/introduction`, accessed in June 2019. [cited on p. 1, 13]

[4] UBER, *streetscape.gl.* `https://avs.auto/#/streetscape.gl/overview/introduction`, accessed in June 2019. [cited on p. 1]

[5] J. Hewitson, "Visualizing data through layers." Available in `Three.jsandBabylon.js:aComparisonofWebGLFrameworks`, accessed in January 2019. [cited on p. 7]

[6] Khronos, "What is webgl?." Available in `https://www.khronos.org/webgl/wiki/Getting_Started`, accessed in January 2019. [cited on p. 7]

[7] "Creating a scene." Available in `https://threejs.org/docs/index.html#manual/en/introduction/Creating-a-scene`, accessed in December 2018. [cited on p. 8, 9]

[8] "Lights." Available in `https://doc.babylonjs.com/babylon101/lights`, accessed in January 2019. [cited on p. 11]

[9] "Visualizing data through layers." Available in `http://deck.gl/#/documentation/developer-guide/using-layers`, accessed in January 2019. [cited on p. 12]

[10] "List of webgl frameworks." Available in `https://en.wikipedia.org/wiki/List_of_WebGL_frameworks`, accessed in January 2019. [cited on p. 14]

[11] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The kitti dataset," *International Journal of Robotics Research (IJRR)*, 2013. [cited on p. 14]

[12] "Welcome to the kitti vision benchmark suite!." Available in `http://www.cvlibs.net/datasets/kitti/index.php`, accessed in January 2019. [cited on p. 14]

[13] "Sensor setup." Available in `http://www.cvlibs.net/datasets/kitti/setup.php`, accessed in January 2019. [cited on p. 14, 15]

[14] OXTS, "Rt3000." Available in `https://velodynelidar.com/hdl-64e.html`, accessed in March 2019. [cited on p. 14]

[15] V. Lidar, "Hdl-64e." Available in `https://velodynelidar.com/hdl-64e.html`, accessed in March 2019. [cited on p. 14]

[16] FLIR, "Flea2 1.4 mp mono firewire 1394b (sony icx267)." Available in `https://www.ptgrey.com/flea2-14-mp-mono-firewire-1394b-sony-icx267-camera`, accessed in March 2019. [cited on p. 14]

[17] FLIR, "Flea2 1.4 mp color firewire 1394b (sony icx267)." Available in `https://www.ptgrey.com/flea2-14-mp-color-firewire-1394b-sony-icx267-camera`, accessed in March 2019. [cited on p. 14]

[18] E. Optics, "Varifocal imaging lenses." Available in `https://www.edmundoptics.com/f/varifocal-imaging-lenses/11853`, accessed in March 2019. [cited on p. 14]

[19] W. Maddern, G. Pascoe, C. Linegar, and P. Newman, "1 Year, 1000km: The Oxford RobotCar Dataset," *The International Journal of Robotics Research (IJRR)*, vol. 36, no. 1, pp. 3–15, 2017. [cited on p. 15]

[20] FLIR, "Bumblebee xb3 1.3 mp color firewire 1394b 3.8mm (sony icx445)." Available in `https://www.ptgrey.com/bumblebee-xb3-stereo-vision-13-mp-color-firewire-1394b-38mm-sony-icx445-camera`, accessed in March 2019. [cited on p. 15]

[21] FLIR, "Grasshopper2 1.4 mp color firewire 1394b (sony icx285)." Available in `https://www.ptgrey.com/grasshopper2-14-mp-color-firewire-1394b-sony-icx285-camera`, accessed in March 2019. [cited on p. 15]

[22] SICK, "2d lidar sensors lms1xx." Available in `https://www.sick.com/de/en/detection-and-ranging-solutions/2d-lidar-sensors/lms1xx/lms151-10100/p/p141840`, accessed in March 2019. [cited on p. 15]

[23] SICK, "3d lidar sensors ld-mrs." Available in `https://www.sick.com/ag/en/detection-and-ranging-solutions/3d-lidar-sensors/ld-mrs/ld-mrs400001/p/p112355?ff_data=JmZmX2lkPXAxMTIzNTUmZmZfbWFzdGVySWQ9cDExMjM1NSZmZl90aXRsZT1M/-RC1NUlMOMDAwMDEmZmZfcXVlcnk9JmZmX3Bvcz0xJmZmX29yaWdQb3M9MSZ/-mZl9wYWdlPTEmZmZfcGFnZVNpemU9OCZmZl9vcmlnUGFnZVNpemU9OCZmZl9/-zaW1pPTkzLjA=`, accessed in March 2019. [cited on p. 15]

[24] HEXAGON/NovAtel, "Span-cpt single enclosure gnss/ins receiver." Available in `https://www.novatel.com/products/span-gnss-inertial-systems/span-combined-systems/span-cpt/`, accessed in March 2019. [cited on p. 16]

[25] "Documentation." Available in `https://robotcar-dataset.robots.ox.ac.uk/documentation/`,accessed in January 2019. [cited on p. 16]

[26] T. A. S. Foundation, "Apache hadoop." Available in `https://hadoop.apache.org/`, accessed in January 2019. [cited on p. 16, 17, 36]

[27] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–10, May 2010. [cited on p. 17]

[28] "Hadoop - hdfs overview." Available in `https://www.tutorialspoint.com/hadoop/hadoop_hdfs_overview.htm`, accessed in January 2019. [cited on p. 17]

[29] T. A. S. Foundation, "Apache hadoop yarn." Available in `http://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html`, accessed in January 2019. [cited on p. 18]

[30] F. Li, B. C. Ooi, M. T. Özsu, and S. Wu, "Distributed data management using mapreduce," *ACM Computing Surveys (CSUR)*, vol. 46, January 2014. [cited on p. 19]

[31] T. A. S. Foundation, "Mapreduce tutorial." Available in `http://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html`, accessed in January 2019. [cited on p. 19]

[32] J. Schmidt, "A n00bs guide to apache spark." Available in `https://towardsdatascience.com/apache-spark-101-3f961c89b8c5`, accessed in January 2019. [cited on p. 20]

[33] P. Lathar, "Big data analysis: Apache spark perspective," *International Journal of Technical Innovation in Modern Engineering & Science (IJTIMES)*, vol. 4, May 2018. [cited on p. 20]

[34] "Apache spark architecture." Available in `https://intellipaat.com/tutorial/spark-tutorial/spark-architecture/`, accessed in January 2019. [cited on p. 21]

[35] IBM, "Apache hbase." Available in `https://www.ibm.com/analytics/hadoop/hbase`, accessed in January 2019. [cited on p. 21]

[36] A. Khetrapal and V. Ganesh, "Hbase and hypertable for large scale distributed storage systems a performance evaluation for open source bigtable implementations," 2008. [cited on p. 21]

[37] T. A. S. Foundation, "Apache hbase reference guide." Available in `http://hbase.apache.org/book.html#arch.overview`, accessed in January 2019. [cited on p. 21, 22, 23]

[38] "An in-depth look at the hbase architecture." Available in `https://mapr.com/blog/in-depth-look-hbase-architecture/`, accessed in January 2019. [cited on p. 22]

[39] A. Thusoo, J. Sen Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive - a warehousing solution over a map-reduce framework.," *PVLDB*, vol. 2, pp. 1626–1629, 08 2009. [cited on p. 23]

[40] J. Rutherglen, D. Wampler, and E. Capriolo, *Programming Hive 1st.* O'Reilly Media, Inc., 2012. [cited on p. 23, 36]

[41] L. Leverenz, "Design." Available in `https://cwiki.apache.org/confluence/display/Hive/Design`, accessed in January 2019. [cited on p. 23, 24]

[42] "Mapreduce vs apache spark- 20 useful comparisons to learn." Available in `https://www.dezyre.com/article/hive-vs-hbase-different-technologies-that-work-better-together/322`, accessed in January 2019. [cited on p. 24, 25]

[43] "Hive vs.hbase–different technologies that work better together." Available in `https://www.educba.com/mapreduce-vs-apache-spark/`, accessed in January 2019. [cited on p. 25, 26]

[44] Google, "Draco 3d data compression." Available in `https://github.com/google/draco`, accessed in July 2019. [cited on p. 26, 41]

[45] A. Doumanoglou, P. Drakoulis, N. Zioulis, D. Zarpalas, and P. 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*, vol. 9, pp. 190–203, March 2019. [cited on p. 26, 27]

[46] "Corto." Available in https://github.com/cnr-isti-vclab/corto, accessed in July 2019. [cited on p. 27]

[47] UBER, "Deckgl (scripting interface)." Available in https://deck.gl/#/documentation/deckgl-api-reference/scripting-interface/deckgl, accessed in March 2019. [cited on p. 30]

[48] UBER, "Layer class." Available in https://deck.gl/#/documentation/deckgl-api-reference/layers/layer, accessed in March 2019. [cited on p. 31]

[49] "Iteration protocols." Available in https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols, accessed in March 2019. [cited on p. 31]

[50] UBER, "Layer class." Available in https://deck.gl/#/documentation/deckgl-api-reference/layers/layer?section=visible-boolean-optional-, accessed in March 2019. [cited on p. 31]

[51] mozilla, "Using fetch." Available in https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch, accessed in June 2019. [cited on p. 32]

[52] "Overview of cloudera and the cloudera documentation set." Available in https://www.cloudera.com/documentation/enterprise/5-13-x/topics/introduction.html, accessed in July 2019. [cited on p. 37]

[53] T. A. S. Foundation, "Webhdfs rest api." Available in https://hadoop.apache.org/docs/r1.0.4/webhdfs.html, accessed in July 2019. [cited on p. 40]

[54] "Parallel programming in javascript using web workers." Available in https://itnext.io/achieving-parallelism-in-javascript-using-web-workers-8f921f2d26db, accessed in July 2019. [cited on p. 42]