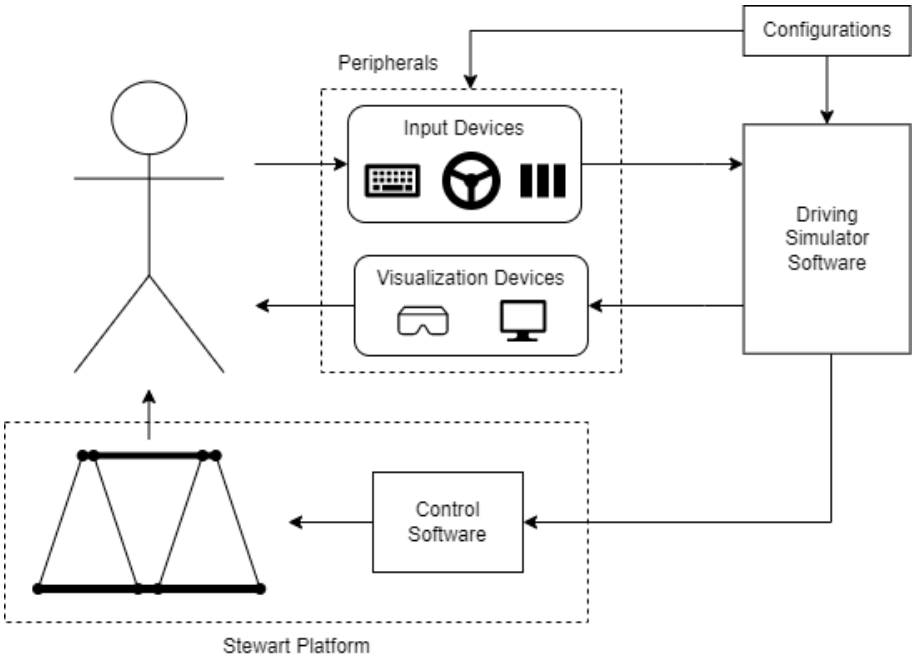


Automotive Virtual Reality Platform

FILIPE RAMOS MENDES
Licenciado

Trabalho de Projeto para obtenção do grau de Mestre em Engenharia Informática e de Computadores

Orientadores:
Doutor Pedro Miguel Florindo Miguens Matutino
Doutor Nelson Filipe Pereira dos Santos

Júri:
Presidente: Doutor Nuno Miguel Machado Cruz
Vogais:
Doutor Artur Jorge Ferreira
Doutor Pedro Miguel Florindo Miguens Matutino

Setembro de 2024

Automotive Virtual Reality Platform

FILIPE RAMOS MENDES
(Licenciado)

Trabalho de Projeto para obtenção do grau de Mestre em Engenharia Informática e de Computadores

Orientadores:
Doutor Pedro Miguel Florindo Miguens Matutino, ISEL
Doutor Nelson Filipe Pereira dos Santos, ISEL

Júri:
Presidente: Doutor Nuno Miguel Machado Cruz, ISEL
Vogais:
Doutor Pedro Miguel Florindo Miguens Matutino, ISEL
Doutor Nelson Filipe Pereira dos Santos, ISEL

Setembro de 2024

Acknowledgements

I would like to start by thanking ISEL, for providing me with the conditions to study and develop myself in the last years, enabling me to complete my Bachelor's degree so far. I would also like to thank ISEL and IPL for providing the required equipment and software to develop this project, and for developing the TrackDyman - IDICA2021 project, entitled "Estudo da dinâmica, segurança e eficiência de veículos automóveis".

I have to express my gratitude to my advisors, Prof. Pedro Miguens and Prof. Nelson Santos, without forgetting Prof. Sérgio André, who, although not an official advisor, was always available as one. Their availability, flexibility, comprehension and knowledge were essential to the successful completion of this project. Many students are not fortunate enough to encounter advisors like this throughout their academic path.

I would also like to thank my family and friends, for being supportive during the development of the project, especially during the toughest phases.

Statement of integrity

I declare that this **project work** is the result of my personal and independent research. Its content is original, and all sources listed in the bibliographic references were consulted and are duly mentioned in the text. I further declare that all scientific and technical references relevant to the development of the work are duly cited and included in the bibliographic references.

The author

Filipe Ramos Mendes

Lisbon, 23rd December 2024

Abstract

Nowadays virtual reality (VR) is becoming a leading topic in technology, as it provides a way for the user to immerse themselves in a virtual environment, that no other technology until now could provide. The user can perform actions and movements in the real world and watch them take place in the virtual world.

The proposed project intends to take advantage of this technology and design and implement a virtual reality driving simulator. The project is naturally composed of different components. The main one is the Stewart platform, which is connected to the driving simulator software that in turn is connected to the input devices, which include VR components such as the VR headset, and the driving components, such as a steering wheel and foot pedals.

A Stewart platform is an electromechanical system that in short is composed of a flat platform controlled by six linear actuators, which work together to provide rotation and translation movements on the three axes, resulting in six degrees of freedom (6 DoF). The driving simulator software is responsible for the vehicle's mechanics, how the vehicle brakes, accelerates, and steers. It receives the user's input, from the different input devices and transforms it into data that will be sent to the platform's software. It also provides a visual representation of the environment that will be shown to the user, for example, through the VR headset.

The project herein depicted implements a prototype of the overall system and performs several test scenarios, together with the Stewart platform.

Besides aiming to bring a realistic experience to the user using the simulator, this project was also developed with a high focus on the separation of functionalities, which means that each component described above could be replaced by a similar one, with the same purpose, without affecting the system.

Keywords

Stewart platform; Driving simulator; Unity; Virtual Reality

Resumo

Atualmente a realidade virtual (VR) tem sido um dos tópicos em destaque do mundo da tecnologia, oferecendo uma forma do utilizador se envolver num ambiente virtual de uma forma que até ao aparecimento da realidade virtual não tinha sido possível. É possível para o utilizador movimentar-se no mudo real e ver as suas ações replicarem-se num ambiente virtual.

Este projeto pretende tomar partido desta tecnologia e planear e implementar um simulador de condução que utilize realidade virtual. O projeto divide-se em diferentes componentes, sendo que o principal é uma plataforma Stewart, que está conectada ao *software* do simulador de condução, que por sua vez está conectado a dispositivos de entrada, que podem ser um teclado, um volante, pedais e uns óculos de realidade virtual.

Uma plataforma Stewart é um sistema eletromecânico composto por uma plataforma controlado por seis atuadores, que servem para recriar movimentos translacionais e rotacionais nos três eixos, oferecendo seis graus de liberdade. O *software* do simulador de condução é responsável pela simulação dos movimentos de um veículo, a forma como acelera, trava e vira. O *software* recebe o *input* do utilizador, dos diferentes dispositivos de entrada, e transforma-o em dados para enviar para o *software* de controlo da plataforma. O *software* do simulador de condução também proporciona uma representação visual do ambiente virtual, por exemplo através de óculos de realidade virtual.

Com a implementação deste projeto, vários cenários de teste podem ser realizados, em conjunto com a plataforma Stewart, e analisados.

Para além do objetivo de proporcionar uma experiência realista ao utilizador, este projeto também foi realizado com um foco na separação de funcionalidades, sendo que os componentes acima mencionados poderiam ser substituídos por componentes semelhantes, sendo que o projeto pouco seria afetado.

Palavras-chave

Plataforma *Stewart*; Simulador de condução; Unity; Realidade Virtual

Contents

List of Figures

List of Tables

Acronyms

DoF	Six degrees of freedom
VR	Virtual reality
3D	Three dimensional
DLL	Dynamic link library
TCP	Transmission Control Protocol
UI	User Interface
USB	Universal Serial Bus
PC	Personal Computer
UDP	User Datagram Protocol
JSON	JavaScript Object Notation
IP	Internet Protocol
ADU	Application Data Unit
PDU	Protocol Data Unit
MBAP	ModBus Application Protocol

Chapter 1

Introduction

Vehicle simulators have been developed for several decades, progressively increasing in sophistication and complexity in parallel with technological advancements over the last two centuries. Before the driving simulators, there were flight simulators, emerging in the military context, providing a way to train pilots with relatively low costs, avoiding the risks and costs of using a real aircraft [8]. One of the first flight simulators, seen in Figure 1.1 was built in France around 1910 and consisted of a barrel with wings attached to a universal joint, so that the instructors could push and pull the barrel to simulate the plane's movement [9].

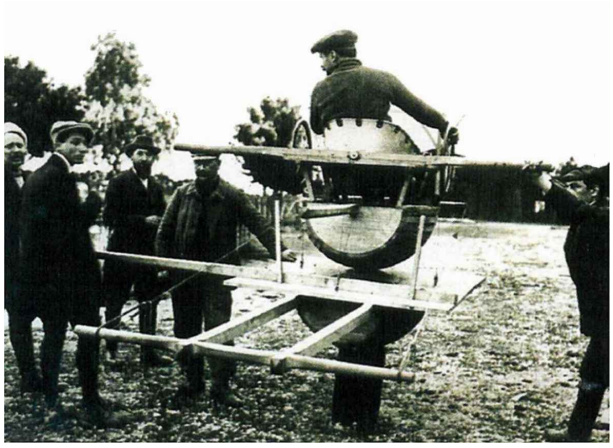


Figure 1.1 Training on the first flight simulator

Only a few decades later, in the early 1970s, the first vehicle simulator was developed. The automobile manufacturer Volkswagen developed a simulator, seen in Figure 1.2 with three degrees of freedom, able to simulate rotational movements in all of the three axes [10]. Similarly to the flight simulators, the driving simulators were ideal for reducing costs, in this case for studies and research, and to provide a safer and more controllable environment for studies of the vehicle's characteristics, the driver's behaviour and the environment [2]. Most of the simulators were developed by either vehicle companies or universities.

Some early simulators simply changed how the image was displayed to the user according to the steering wheel movement and the speed of the displayed image changed according to pedals' input [2]. These were not truly interactive simulators, as the user only controlled how the

Figure 1.2 Driving simulator developed by Volkswagen [1]

image, from pre-existing footage or from miniature representations of a road in a moving belt that moved towards the camera, was displayed, usually requiring the user to test a previously defined scenario. In these early models, the users only had visual feedback. The moving belt and the resulting image can be seen in Figures 1.3 and 1.4.

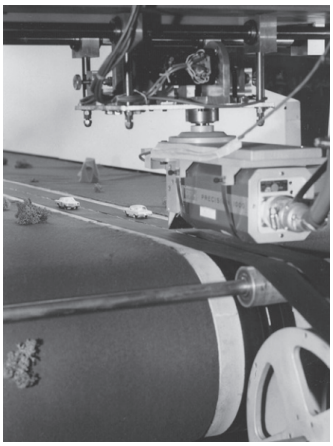


Figure 1.3 Miniature moving belt [2]

Figure 1.4 Driver's video [2]

With the advancement of technology and the emergence of Stewart platforms, moving platforms that support movement in six degrees of freedom, more advanced simulators could be developed. These advancements mainly included the addition of movement to the platforms. The driver and his seat, and in some models part or even the whole vehicle could be placed on top of the platform, which emulates the vehicle's movements and accelerations in the three axes, with translation and rotation movements. These simulators were much more interactive than earlier models and allowed the user to control the movement of the simulated vehicle. One of these simulators is represented in Figure 1.5.

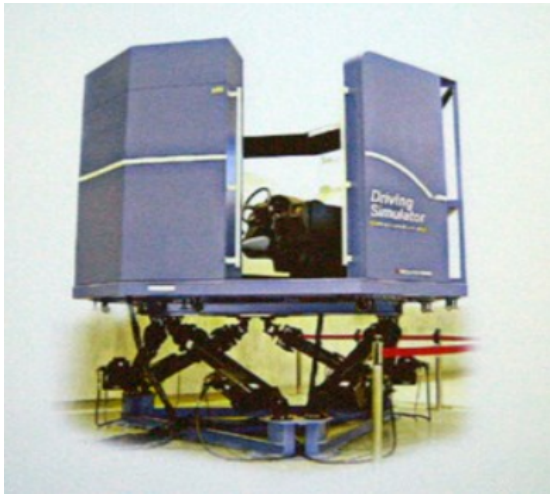


Figure 1.5 Driving simulator using a Stewart platform

The latest advancement in the field of driving simulators was the addition of other ways to recreate movement besides the capabilities of the Stewart platform. The NADS-1 [3] (National Advanced Driving Simulator), developed in 2001 by the Driving Safety Research Institute at the University of Iowa, uses a dome, on top of a Stewart platform which in turn can be moved by another platform. The dome contains the user, the full-sized vehicle, which is attached to the ground by four hydraulic actuators that are able to represent the road feel, and a 360º horizontal and 40º vertical visual display. The dome can also rotate 330º, overpowering the Stewart platform's rotational capabilities. The platform which supports the Stewart platform can also provide much bigger movements in the X and Y axes than the Stewart platform. The NADS-1 can be seen in Figure 1.6.

Figure 1.6 NADS-1 at the Driving Safety Research Institute [3]

Related to the appearance of virtual reality, in the 1950s, Morton Heilig, a cinematographer, created the Sensorama [11], a device which was a cabinet that included a 3D display, speakers, a vibrating chair among other features to immerse the user. Heilig would later invent the first example of a head-mounted display for non-interactive films. The military sector would also develop contributions for this field, introducing the ability to track the user's head movement.

Another important participant in the advancements of the virtual reality field was Ivan Sutherland. Sutherland contributed to the development of the head-mounted display, seen in Figure 1.7, which consists of a device suspended from the ceiling and, as the name indicates, rests on the user's head. The major difference from the inventions until now was that the device was connected to a computer that generated images instead of a camera [11] [12].

Figure 1.7 The head-mounted display being used

Until this point, most of the virtual reality devices were developed for research purposes. However, in the 1990s, the first consumer-focused device was launched [13]. The VFX1 Headgear was a helmet used together with a handheld controller and it was mainly used for video games. With new developments in the field, more advanced models were built and by more companies. In the 2010s, what would become one of the most important companies in the virtual reality world, at the time called Oculus VR, started developing the Oculus Rift, released in 2016. During its development, Facebook, Inc. acquired the company, providing more attention and resources to the development of virtual reality headsets. In 2021, Facebook, Inc. rebranded as Meta and since then developed some of the most advanced VR headsets, Meta Quest 2, Meta Quest Pro and Meta Quest 3 [14]. These models can function on their own, without the need for a computer, as opposed to earlier products from Oculus VR. Depending on the model, these headsets can provide hand tracking, eye tracking and even face tracking,

with computing power equivalent to a common laptop [15].

In 2023, Apple introduced their mixed reality headset, the Apple Vision Pro seen in Figures 1.8 and 1.9, entering the competition of the virtual reality headset market. With this headset, Apple intended to introduce the headset as a product to use in everyday life, weakening the association between virtual reality headsets and gaming and other more specific uses. The headset can fulfil most functions of a computer, having similar specifications and being able to display several virtual screens [16].

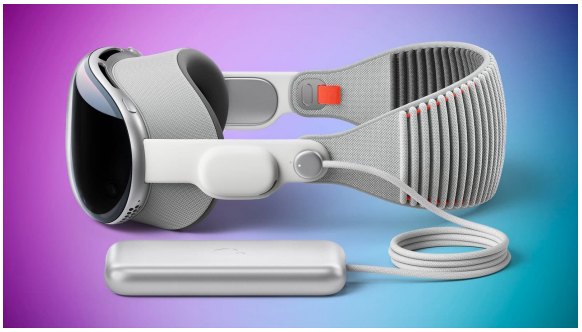


Figure 1.8 Apple Vision Pro

Figure 1.9 Apple Vision Pro in use

Virtual reality can easily be integrated with driving simulators that use Stewart platforms, as it works mainly as a way to display the image to the user. In most cases, there is already a virtual environment where the vehicle and the driver exist, so the user's head movement can be tracked and simulated in the virtual environment. This way, when the user looks around, the image shown changes accordingly.

The car manufacturer BMW recently presented an innovative way to combine driving and virtual reality. In 2022, BMW announced BMW M Mixed Reality, a driving experience in which the vehicle itself becomes the controller [17]. The driver drives a real car with a VR headset on, as seen in Figure 1.10, and while the car is being driven in a real environment, the image displayed on the headset is of a virtual environment, as seen in Figure 1.11. Although this is very close to the real driving experience, this is also a simulation, without the restrictions of using technologies such as the Stewart platform.

Figure 1.10 Driver using the VR headset

Figure 1.11 Example of an image displayed on the VR headset

1.1 Objectives

The intended outcome of this project is to conceptualize, develop and implement a modular driving simulator, with which a user can interact and almost feel like they are driving a real vehicle. To achieve this, a realistic simulation of the movement of a vehicle, and a realistic feeling to the user, should be provided by a physical platform and software. One specific objective of this project was to handle an already implemented piece of software responsible for the Stewart platform's movement that was intrinsically attached to a specific tool and to transform it in a way that would allow it to be more independent and easier to use.
While developing this project, the concept of modularity, which allows further updates of each module that composes the proposed system, is going to be taken into account, as it helps to achieve a better and clearer final project.

1.2 Document organization

This document will describe the different components used, their integration into this project, and how they help to achieve the objectives listed before. The document is organized into chapters.

In Chapter 2, the architecture of the project will be detailed, the different components of the project listed, and the main objective of each component identified.

In Chapter 3, the driving simulator software implementation will be described, with every component and interaction between components detailed. For each feature, the possible options will be listed, with the respective advantages and disadvantages, and the choices will be justified.

The Stewart platform is described in Chapter 4. After a brief introduction of what it is, both the physical aspect of it and the software that it uses to handle data will be detailed. What was done from the starting point of the system to the final product will be described.

Chapter 5 details step-by-step how to correctly set up every module of the project in order to run the application and test it.

Chapter 6 is used to list some of the problems encountered in the development phase of the project and how they were overcome. The chapter analyzes each situation before and after the implementation of the proposed solutions.

Chapter 7 analyzes every test scenario implemented in the project, comparing results from several phases of the project, from the driving simulator to the platform's movement.

Finally, Chapter 8, summarizes the main work herein described and lists what could be implemented in future iterations of the project and what could be improved.

Chapter 2

Automotive Virtual Reality Platform architecture

One of the major aspects developers take into account when developing a project is correctly separating it into different modules and separating the responsibilities of each of these modules. If done correctly, this separation improves the global quality of the project, makes the project easier to understand, and easier to add new features to it. It also provides an easier way to replace the existing modules with similar ones, if needed.

This project can easily be separated into three main components:

- *Peripherals* - the input devices are responsible for handling user input, who interacts with them in order to control the vehicle. The visualization devices are responsible for producing visual feedback to the user;
- *Driving simulator software* - responsible for representing the vehicle in the virtual environment, controlling its movement, and how it interacts with the surrounding environment. Sends data about the vehicle's movement to the next module;
- *Stewart platform* - composed by the control software and the physical platform. The software receives the data from the previous module and performs the calculations to convert it into a relative position for the platform to change into. The physical platform is where the user will be seated and where the user will sense the movement of the virtual vehicle.

These components and the respective data flow are depicted in Figure 2.1, which represents the first approach to the system architecture.

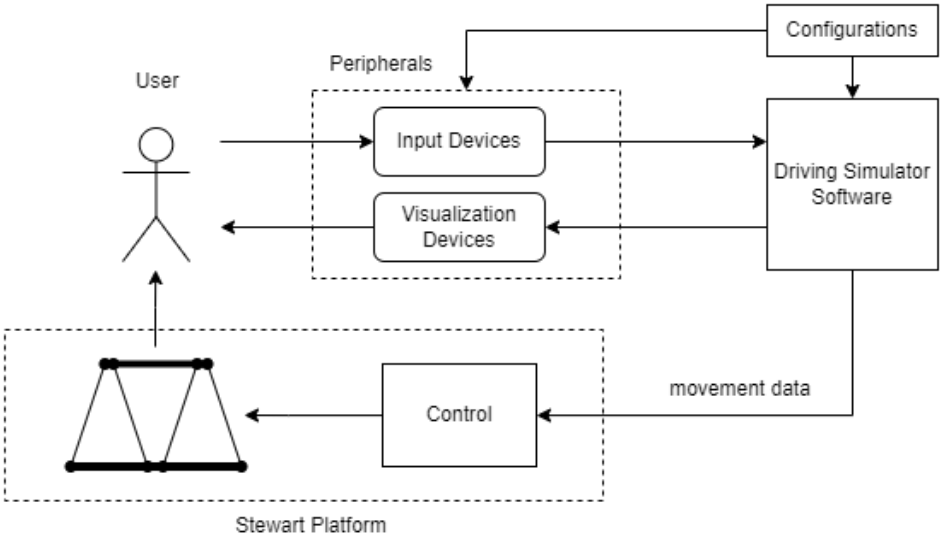


Figure 2.1 Automotive Virtual Reality Platform (AVRP) Architecture

Based on the general architecture, some of the components can be more detailed for this project, regarding some of the sub-components, in the case of the input devices, visualization devices, and the technology, for the driving simulator software.

For the development of the driving simulator software, there is a need for software that supports the control of a 3D object in a 3D environment, with support for physics mechanics. The used software has to allow the reception of user's input through different types of devices and has to be able to send data to another module. A game engine is capable of all of the requirements listed and does not have to be strictly related to the development of a video game.

At the starting phase of the project, there was already an existing implementation of the platform's control software which was improved upon and is responsible for the movement of the Stewart platform's legs. One of the objectives of this project is to decouple the model from the technology used to develop it. The diagram with the detailed components is in Figure 2.2.

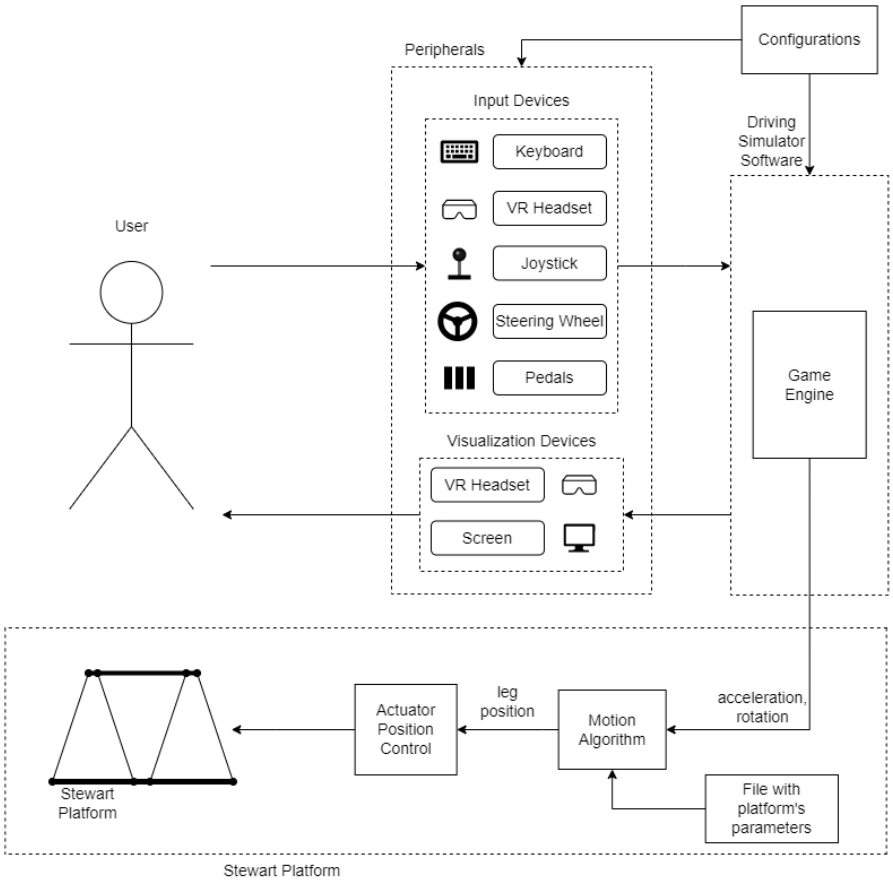


Figure 2.2 AVRP detailed architecture

Chapter 3

Driving Simulator Software

As previously mentioned, the most appropriate tool to implement the driving simulator for this project is based on a game engine. When choosing a game engine, there are a lot of options, but few are as well known as Unity [18] and Unreal Engine [19]. Both are free to use, offer support and version updates, and are well-documented. Both would be suitable for the development of this project, however, Unity is known to be more intuitive to learn and uses C# [20] for development, as opposed to C++ [21] in the case of Unreal Engine.

When using Unity, the user interacts with the Unity Editor, creating, placing and editing objects in a *Scene*, which represents the environment where the application takes place [22]. The Unity Editor also allows the user to change variables from these objects, to run the application directly from the Editor and even to change these variables during the execution of the application. However, in the final product, an executable file is produced, independent from this tool, and therefore, the user does not have that same freedom, and these variables can only be changed programmatically.

3.1 Introduction to Unity

Unity allows the representation of both 2D and 3D objects and, in both cases, this representation is performed through a *GameObject* [23], which is the base class for all entities existing in an environment. The *GameObjects* can contain *Components* [24], which can change the behaviour of their *GameObject*. Unity provides an extensive list of configurable *Components* that can be added. Custom scripts can also be added, similarly to other components, allowing the programmer to attach as many scripts as needed to the *GameObject*. Every *GameObject* has an essential *Component*, the *Transform* [25], which stores the position and the rotation of the object.

To an existing *GameObject*, a C# script can then be applied, in order to execute and have an effect on the environment. Besides being used to represent a 2D or 3D object, a *GameObject* can also be used without a physical representation, empty, and be used to handle the environment where the other objects exist and manage aspects such as what objects to create, when, and where.

The majority of the custom scripts created in Unity extend from the base class *MonoBehaviour* [26]. This class provides an extensive set of methods that can be overridden by the new class and that are related to the flow of events in the environment. These methods are often referred to as life cycle functions - methods with specific names that are called when specific events occur or at a specific phase of the program. The most relevant life cycle functions are: *i) Awake*; *ii) Start*; *iii) FixedUpdate*; *iv) Update*; *v) LateUpdate*; *vi) OnApplicationPause*; vii) *OnApplicationQuit*; and *viii) OnDestroy*. These life cycle functions follow a specific order and have a specific repetition frequency over the script execution. A summarised version of a script life cycle is depicted in Figure 3.1, which is based on Unity's documentation flowchart [4].

The *Awake* [27] method executes when a *GameObject* is activated and is used to initialize variables before the application starts [27]. The *Start* [28] method is called once per script and executes before any *Update* method is called.

The *Update* methods previously mentioned differ in the execution frequency and when they are called.

- *Update* [29] - executed each frame; the time interval between frames can vary.
- *FixedUpdate* [30] - executed every fixed timestep, has the frequency of the physics system. When using the default settings, the method is called approximately 50 times every second, executing approximately every 20ms;
- *LateUpdate* [31] - executed every frame after all the *Update* methods are executed.

The main difference between *Update* and *FixedUpdate* is the frequency at which they occur. *FixedUpdate* is executed at a fixed time interval, whereas *Update* is executed every frame. Unity tries to run the application at the fastest frame rate possible, however, the frame rate depends on the capacities of the machine where the application is executed. If the application has a lower frame rate than the number of fixed timesteps per second, several *FixedUpdates* can occur between two iterations of the *Update* method. This means that there are several physics updates for each frame [32].

It is recommended to use the *FixedUpdate* method when dealing with physics and when using a *Rigidbody* [30], as all physics calculations and updates occur immediately after *FixedUpdate*. Therefore, the code that handles the vehicle movement will be implemented inside this method.

The method *OnApplicationPause* is called if the application is Paused. The method *OnApplicationQuit* [33] is called after the application quits and can be used to execute cleanup code, such as closing connections. Finally, *OnDestroy* [34] is called when the scene or game ends.

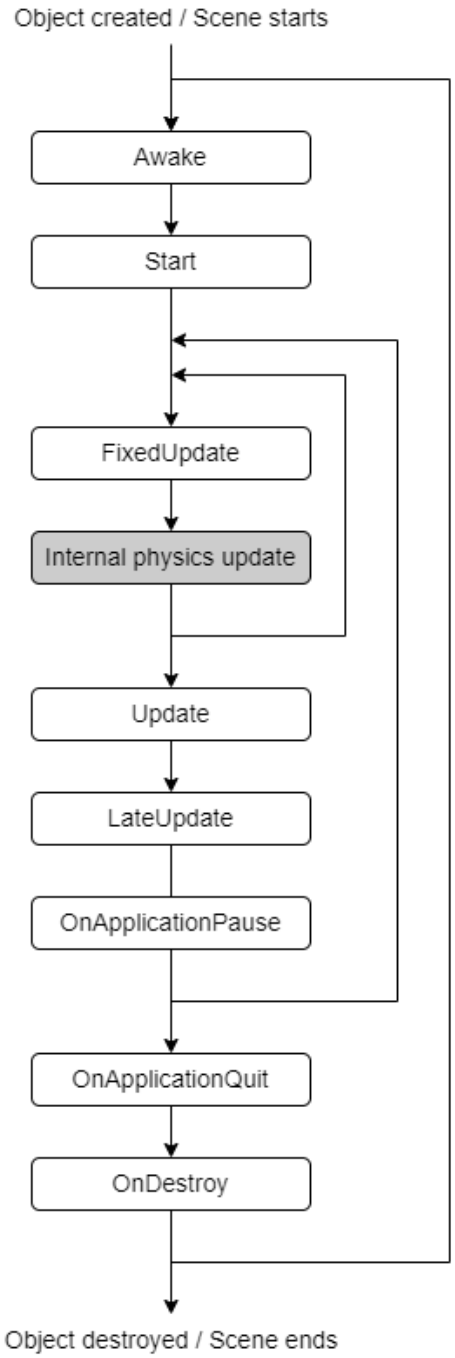


Figure 3.1 Script lifecycle flowchart [4]

3.2 Movement in Unity

To represent a moving 3D object in Unity, besides the script that handles movement, no other components are required. However, in order to represent more realistic movement and considering the interaction between the object and the environment it exists in, some component types can be added: *i) Rigidbody*; *ii) Box Collider*; and *iii) Wheel Colliders*.

A *Rigidbody* can be added to a *GameObject*, allowing Unity to control it through Unity's physics engine. This means that forces can now be applied to the *GameObject*'s *Rigidbody* and therefore control the 3D object itself. Unity will automatically apply the force of gravity to this object and pull it downwards.

In Unity, there is more than one way to implement movement on a 3D object. In the following paragraphs, three possible approaches will be described and the advantages and disadvantages will be compared, in order to choose the best method to use in this project. The main difference between these approaches is what is involved in producing the movement, it can be the object itself, the *Rigidbody* component or the *Wheel Collider* component.

The first approach uses the *Translate* method. This is the simplest way movement can be implemented in Unity. As the method is applied to the object itself and not its *Rigidbody*, the Unity physics engine isn't used and therefore it provides a much less realistic movement.

The second approach uses the methods *AddForce* and *AddRelativeForce*, which are available on the class *Rigidbody* and allow the appliance of forces on the object's Rigidbody. As they are applied to the *Rigidbody*, they require that the object has this component and therefore the Unity physics engine is used, bringing a more realistic movement to the object. There are four types of forces: *i) ForceMode.Force*; *ii) ForceMode.Acceleration*; *iii) ForceMode.Impulse*; and *iv) ForceMode.VelocityChange*. The approach of the application of forces directly could be used to implement the object's movement.

However, the last approach, which uses a specific type of collider implemented by Unity, provides more control over the breaking and has an easier syntax. Unity uses *Box Colliders* and other colliders to handle collisions - when *GameObjects* come into contact and what happens when they do. They are used to define the shape of the *GameObject*, without having a visible representation. Depending on the shape of the object, the name and shape of this component may be different. In the case of a sphere, the component is called *Sphere Collider*. More complex shapes can use the *Mesh Collider* component, which matches the geometry of the object [35].

A *Wheel Collider* is a type of collider specially designed for wheels, providing several features that help implement vehicle movement, such as wheel physics, suspension, and friction. As opposed to the previously mentioned components, which the vehicle only has one of, in the case of the *Wheel Colliders*, the vehicle has as many as it has wheels, even if with the same parameters.

The *Wheel Collider* component has several properties and some of the most important ones and the respective values being used are present in Table 3.1. Unity does not use a unit system, however, the units can be interpreted as units from the metric system.

Table 3.1 WheelCollider properties and respective values

Attribute	Value
Mass	20 kg
Radius	0,35 m
Wheel Damping Rate	0,25 m
Suspension Distance	0,6 m
Spring (Suspension Spring)	35000 N/m
Damper (Suspension Spring)	4500
Extremum Slip (Friction)	0,4
Extremum Value (Friction)	1
Asymptote Slip (Friction)	0,8
Asymptote Value (Friction)	0,5
Stiffness (Friction)	1

The class *WheelCollider* has the properties *brakeTorque*, *motorTorque*, and *steerAngle*, which provide a way to clearly accelerate, brake, and steer the object while providing a realistic implementation of movement. Instead of being changed on the object's *Rigidbody*, these properties are changed on each wheel's *WheelCollider*.

3.3 Unity project's GameObjects and classes

In a Unity project, most of the time a script is a component of a *GameObject*, which means they are associated with it and its other components. However, scripts should still be written and thought of as regular classes, being possible to have a Object-oriented programming perspective. The script has access to its *GameObject's* other components and can read their values, behaving differently according to the other components and benefiting from them. For example, the *VehicleController* class that will be described below would not make sense if not for the *Rigidbody* or the *WheelCollider* components of the *GameObject* that the script is attached to, as they are required for the vehicle's movement.

When developing a Unity project, as with any other software project, it is important to take into account the separation of concerns and to minimize the number of responsibilities and functions each class has. For this reason, it is more effective to define the main functionalities of the project in advance. From the user input to the point of sending data to the platform, the main functionalities to be performed by the project are the following:

1. Present a user interface to the user;
2. Manage the testing environment;
3. Handle user input;
4. Move the vehicle;
5. Analyze movement data from the vehicle;
6. Send data to the platform.

These functionalities were divided considering the architecture established in Chapter 2. When analyzing Figure 2.1, each one of these functionalities can be associated with a specific component, as depicted in Figure 3.2. It is important to note that, even though the numbers are represented near other components, all of the functionalities are implemented in the Driving Simulator Software.

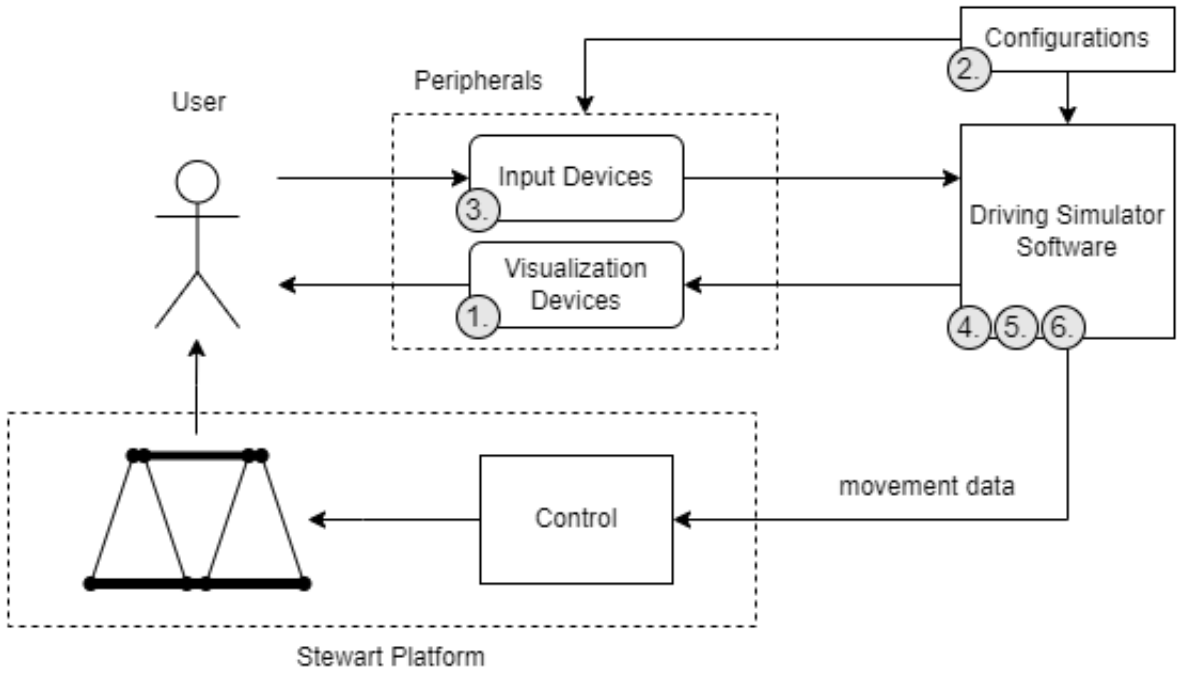


Figure 3.2 General architecture of the project with associated functionalities

With this first approach, some essential classes can immediately be identified. Since the project was developed with the intent of being able to replace components without needing to rewrite the code, the functionalities were separated into different classes. This allows for an easier re-utilization and replacement of the code and components. The utilization of VR in combination with physical objects also requires some type of calibration that is not similar to any of the previously mentioned functionalities and therefore requires another class.

The main classes that can be defined are the following: *i) UI; ii) Manager; iii) VehicleController; iv) Logger; v) PlatformInterface;* and *vi) SteeringWheelCalibrator.*

Most of the classes that will be discussed below are associated with a lifecycle and therefore will have associated *Start* and *Update* methods, which means most of the scripts will be constantly running and interacting with each other. These classes are all associated with a single *GameObject*, which most of the time will have a similar name, serving as a container for that class. For this reason, each GameObjects will be detailed below, with a focus on the classes attached to them. As *GameObjects* can have child and parent *GameObjects*, these can be structured in order to form a logical hierarchy and these hierarchies will also be detailed in the following sections.

In a Unity application, every *GameObject* exists in the context of a single *Scene*. Multiple *Scenes* can exist in a single Unity program, however, communicating between *Scenes* is a complex task and only makes sense in a larger project that has different contexts. As the same *GameObjects* are always used, only one *Scene* exists in this project. Even though a *Scene* is not a *GameObject*, a hierarchy can be established between it and the created *GameObjects*, where the *Scene* can be interpreted as the highest element in the hierarchy, as depicted in Figure 3.3. It is important to note that, in order to be more clear, only the relevant *GameObjects* are portrayed and the names displayed are indicative. In the case of the *Environment GameObject*, there may exist more than one *GameObject* with the same purpose, representing different environments.

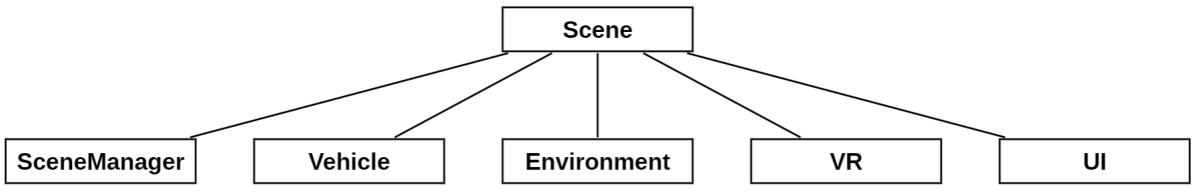


Figure 3.3 *Scene* and*GameObjects* hierarchy

Before detailing the created *GameObjects* and scripts, it is important to detail the Enumerations used throughout the project, as they are used by the developed scripts. These can be used to represent a choice from a set of mutually exclusive values [36]. Each Enumeration is in its own file, for better code organization and readability. Using enumerations also eases a possible expansion of the code. The created Enumerations are listed below and are represented in Figure 3.4 with their possible values listed:

- *DRIVING_MODE* - contains all the possible testing scenarios, including the free roam mode. Mainly used by the *VehicleController* class to know if the movement is performed by the user - free roam mode - or if it should be predetermined - other testing modes;
- *INPUT_DEVICE* - contains the supported input devices by the application, keyboard and joystick. Even though the steering wheel and pedals are not exactly a joystick, it is how Unity classifies this type of input;
- *MODE* - the two modes the application can run in, VR or running on a regular screen.

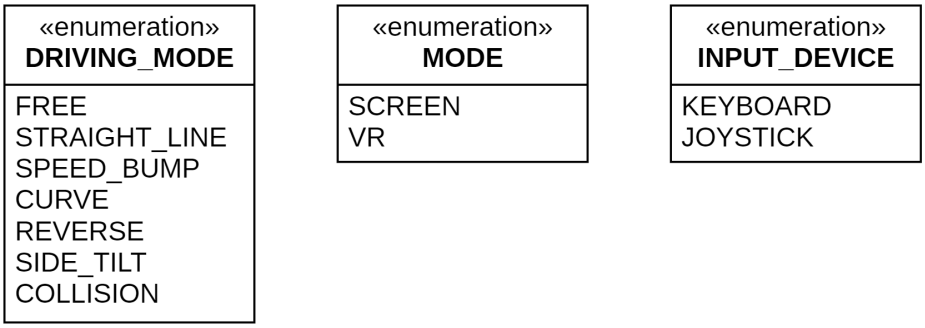


Figure 3.4 Enumerations with their possible values

When analyzing the diagrams depicted in Chapter 2, Figures 2.1 and 2.2, there is a direct representation of some components in the *Enumerations*, *MODE* is used to group all the possible Visualization Devices and *INPUT_DEVICE* is used for the Input Devices component.

3.3.1 VR *GameObjects*

Although they are not the most important *GameObjects* to detail, it is important to describe the *GameObjects* related to VR first as they will be referenced in the following sections.

While the utilization of VR in this project is not mandatory, it allows a better and more immersive experience for the user. The packages *XR Core Utilities* [37] and *XR Interaction Toolkit* [38] provide a set of predefined *GameObjects* and scripts that implement a large number of VR functionalities, such as the usage of VR controllers, using hands as controllers and

representing the user's head movement and consequently body movement in the virtual environment. However, the project only needs the head movement, the hand tracking and the hands to be represented in the virtual environment.

The package *XR Core Utilities* provides the *XR Origin (XR Rig) GameObject* [39], which implements the basics of a VR project. This object is defined as the centre of tracking space in an XR scene. It consists of several *GameObjects* hierarchically organized, with different components and scripts. By default, *XR Origin* is set to use the VR controllers, so it has a child *GameObject* for each of the controllers, left and right, with the respective scripts to make them work. As the project will not use them, these objects can be deleted and replaced by a *Hands GameObject* and its children *GameObjects*. All of the *GameObjects* and their relevant scripts and components will be detailed in this section, the hierarchy is represented in Figure 3.5.

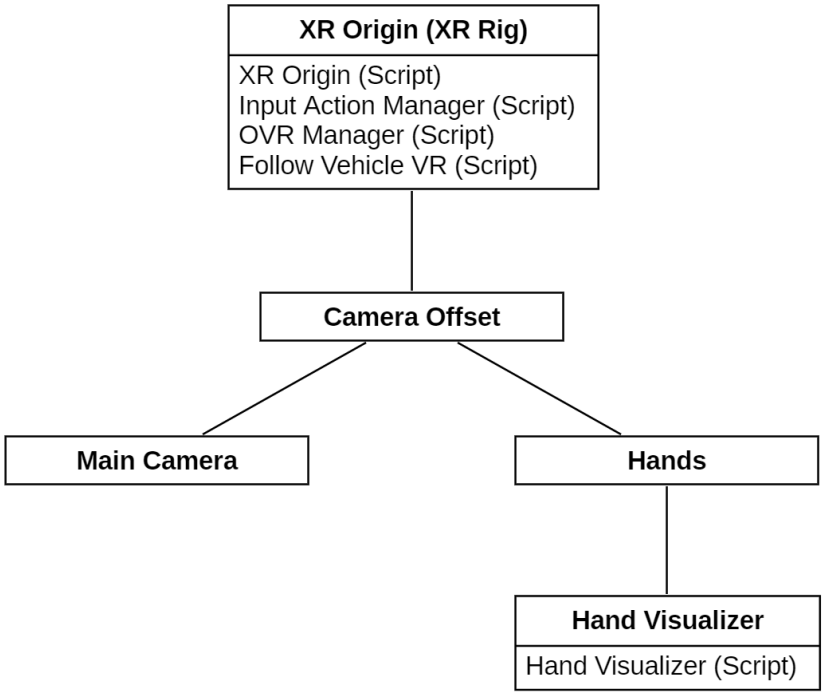


Figure 3.5 XR Origin GameObject structure

The *XR Origin GameObject* has several associated scripts, some of them were associated by default and others were added after adding the *GameObject* to the *Scene*.

The *XR Origin* script is used to configure how the user will see the virtual environment. The script is configured using the children *GameObjects* from the *XR Origin GameObject*, such as the *Camera*. One of the most important properties of this script is the *Tracking Origin Mode* property, which can be configured to different values, *NotSpecified*, *Device* or *Floor* [40]. In the case of the *Device* option, the input device, more specifically the VR headset, will be tracked relative to the first known location. With the *Floor* option, the headset will be tracked relative to a specified location on the floor, indicated by a *Vector3* object, corresponding to the virtual

coordinates. The *NotSpecified* option uses the default mode of the device. The chosen option in this project was the *NotSpecified* option.

The *Input Action Manager* script is used to configure how the different types of inputs are handled. Whether it is the user's hands or the VR controllers, this object associates actions and movements with different meanings to be used in the application code. In the case of this project, it is only used to help with the hand tracking.

The *OVR Manager* script is used specifically by Oculus Quest devices. This script can be configured to determine what should be tracked, such as the hands, eyes, body, etc., and how it is done. In all the settings the default values were used, except the *Hand Tracking Support* which was configured to *Hands Only*, to remove the possibility of using the VR controllers.

The *Follow Vehicle VR* is the only script that is not included in the package and its purpose is to make the *XR Origin GameObject* move according to the vehicle. Because the user is not moving when controlling the vehicle, if it were not for this script, the vehicle would move and the user's virtual position would remain in the original coordinates.

The *Camera Offset GameObject* [39] is used to offset the *Main Camera GameObject* from the *XR Origin GameObject*. The *Main Camera GameObject* is similar to a regular camera object, showing the visual output to the VR headset instead of the screen.

The *Hands GameObject* groups everything regarding the utilization of the hands in VR and was configured with the aid of the *XR Hands* package [41]. In the case of this project, where it is only needed to represent the hands and not use them as controllers, it is only necessary to have the *Hands Visualizer* script, which is attached to the *Hand Visualizer GameObject*. This script needs to be configured to use the hand's meshes, the 3D representation of each hand, which are also provided by the *XR Hands* package. The script also has some other variables that can be configured but are irrelevant to this project. After the configurations are completed, the hands are represented as depicted in Figure 3.6.

Figure 3.6 Example of how the user's hands can represented in the virtual environment

3.3.2 *Scene Manager GameObject*

The *Scene Manager GameObject* and its children *GameObjects* are responsible for handling how the application executes. Each child has a different responsibility and the Scene Manager is responsible for the coordination between them. The *GameObjects*' structure is represented in Figure 3.7, where every *GameObject* has its respective components, in this case, most *GameObjects* only have a single script with the same name, as they serve as a logical container for the script. This also allows for easier customization by the user in the Unity Editor.

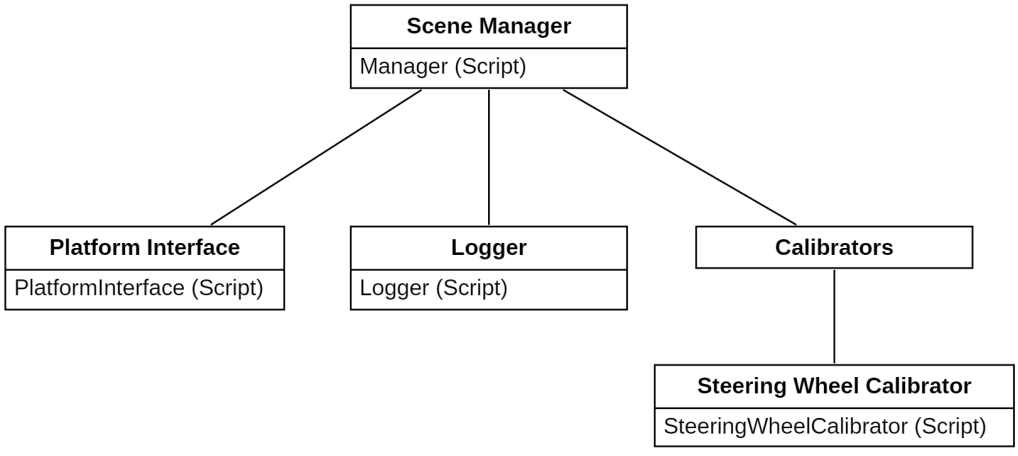


Figure 3.7 Scene Manager GameObject hierarchy

As previously mentioned, the *Scene Manager* is responsible for the coordination between its children *GameObjects* as well as others that are not present in the structure in Figure 3.7, such as the *VehicleController* and the environment. This is achieved by activating or deactivating *GameObjects* according to the user's settings. For example, if the user chooses to use the application in VR mode, the *GameObjects* for the regular cameras are deactivated and the *GameObject* that is responsible for everything VR-related is activated and vice-versa.

The *Manager* script starts by evaluating the *showUI* variable, which has to be defined in the Unity Editor before the execution. If true, the user is presented with the start screen of the application, where they can change some settings. If false, or after the user configuration, the *SetUpScene* method is called. This method is responsible for activating the correct cameras according to the mode, showing the correct driving environment and activating the *Calibrator* and the *Logger*, according to the settings. After this, the user calibrates the screen and, if intended, the SceneManager tries to establish a connection to the Stewart platform control software. If it fails, the application shows an error which requires the restart of the application. If it succeeds, the *SceneManager* is in the *ready* state and calls the *SetUpVehicle* method from the *VehicleController* class, sending the chosen driving mode, either free roam or a test environment, and the chosen input device. The flowchart depicted in Figure 3.8 roughly represents the logic followed by the *SceneManager* script.

The *Logger* class is directly associated with the vehicle and is responsible for reading its data and calculating other useful information. The vehicle's velocity can be accessed through the *Rigidbody* component, but some computations are required to obtain the acceleration. Each iteration of the *Logger*'s *FixedUpdate* updates the current and last velocity values and uses them to compute (3.1). *Time.fixedDeltaTime* is the time between physics updates, in other words, the time between each *FixedUpdate* iteration.

$$acceleration_{x,y,z} = \frac{currentVelocity_{x,y,z} - lastVelocity_{x,y,z}}{Time.fixedDeltaTime} \tag{3.1}$$

Every iteration of the *FixedUpdate* method calculates these values and, if the user wants to use the Stewart platform and is connected to it, the *Logger* sends the computed values to the platform control software.

When the Unity application execution is stopped, all the values that were calculated throughout the execution, are stored in text files so that they can be later analyzed and compared to the platform's data, for debug purposes. This is done inside the *OnApplicationQuit* method, which is provided by the *MonoBehaviour* class, and that runs when the execution is stopped.

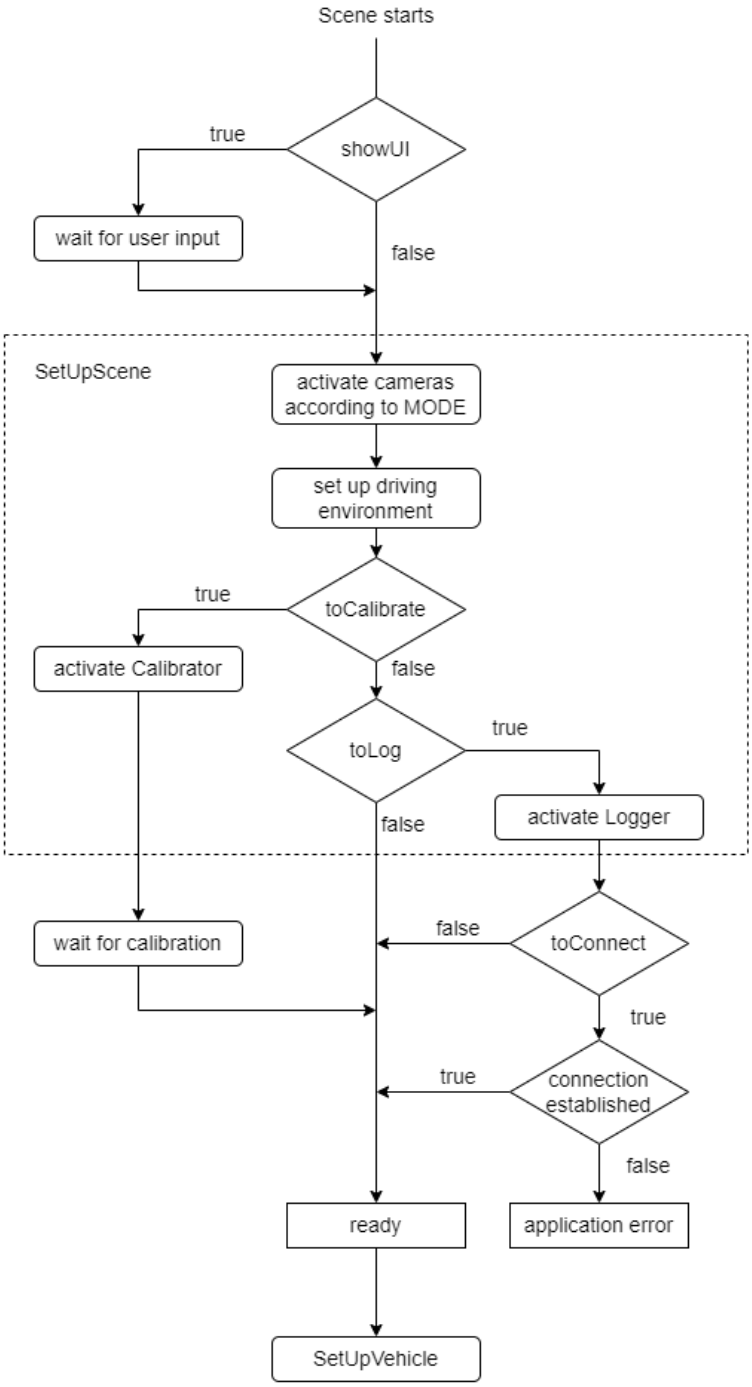


Figure 3.8 SceneManager script logic

The *PlatformInterface* class is responsible for the interaction with the external software that controls the platform and that is addressed in detail in Chapter 4. Even though this class inherits from *MonoBehaviour*, it does not make use of any of the *Start* or *Update* methods. Instead, it provides methods for another class to use, more specifically the *Logger* class. The *ConnectToPlatform* method is supposed to be used a single time to establish a TCP connection to the platform control software. On the other hand, the *SendData* method is supposed to be called constantly, every time the *FixedUpdate* method is executed in the *Logger* class. This method groups together all the data concerning the acceleration, velocity and rotation of the vehicle and sends it through the previously established connection.

The *Calibrators GameObject* is only relevant when using the VR headset and is meant to be the parent *GameObject* for all the possible calibration needs of the project. In the final stage of the project, only the steering wheel was meant to be calibrated, however, in future versions of the project, there may be a need to also calibrate other physical objects to represent them in the virtual space, for example, the foot pedals, gear stick or even the driver's seat. Each calibrator is allocated to a different *GameObject* for better organization, yet they could all be associated with the parent *GameObject*. However, as a calibrator could require more than a single script, to avoid overloading the *Calibrators GameObject*, this split was made.

The *SteeringWheelCalibrator* class is responsible for the calibration of the virtual representation of the physical steering wheel. To improve realism in the application, it is better for the user to see a representation of the steering wheel they are moving in real life. As the user or even the seat may not be in the same position relative to the steering wheel every time the application is executed, it is better to have some calibration performed, so the user can see their hands moving the virtual steering wheel according to the movement of the physical wheel.

One of the VR packages used in this project, *XR Hands*, provides a variety of methods and classes that allow the developers to use information from the captured image of the VR headset user's hands. The VR headset is capable of detecting the hands and fingers' movement and positioning, to interact with objects in the virtual environment. However, in this project, the need for VR is much simpler, and it is not needed to implement this type of interaction, it is just needed to detect the position of the user's fingertips. The virtual representation of the hands does not need to interact with the virtual steering wheel, if correctly calibrated when the user turns the physical steering wheel, the virtual hands will appear to touch and turn the virtual wheel.

The package *XR Hands* identifies each of the user's hands as an *XRHand* object [42] which is composed of a total of 26 joints, called *XRHandJoints* [43], being identified by the finger they belong and by its position in the said finger. Besides the fingers, these joints can also be the palm or the wrist [44]. All of the possible *XRHandJoints* are represented in Figure 3.9.

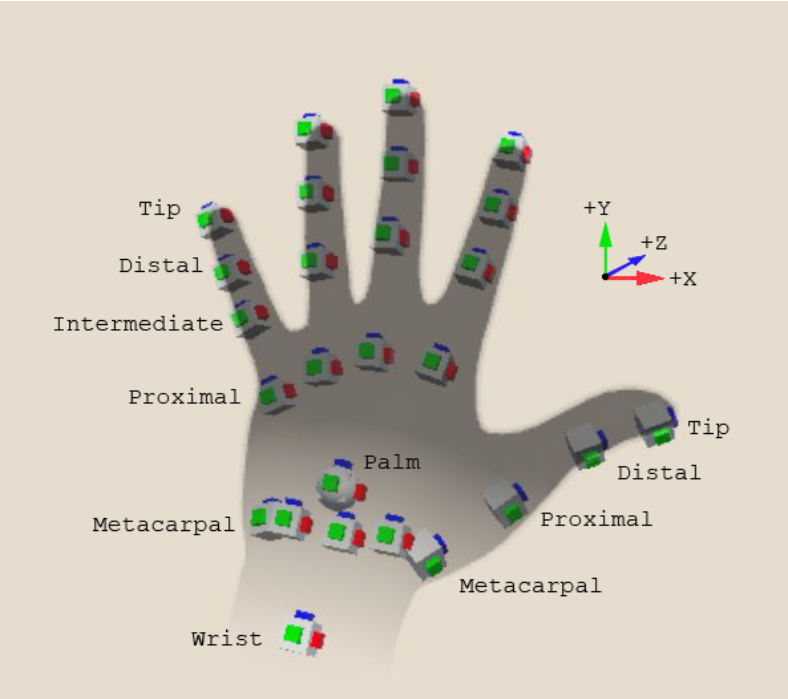


Figure 3.9 Every possible *XRHandJoint* in an *XRHand*

The calibration is performed by the user, who has to put their left index fingertip (represented as *A*) on the center of the physical steering wheel and the right index fingertip (represented as *B*) on the bottom of the steering wheel, aligned with the center, as depicted in Figure 3.10. A timer, set by default to 5 seconds, is shown to the user, who has this time to position their fingers in the correct positions on the physical steering wheel. When this timer ends, the positions of the fingertips are registered and the size and approximate position of the steering wheel relative to the user's eyes are calculated. The virtual representation of the steering wheel is then adjusted to be as close to the real steering wheel regarding the size and position, to make the user feel as if they are turning the virtual wheel. This is done by calculating the distance between both fingertips, which corresponds to the radius of the steering wheel. As each fingertip is represented by a *Vector3*, an object that holds the 3D coordinates of an object in a Unity *Scene*, this radius can be calculated with (3.2). The steering wheel is then positioned in the virtual environment in the correct position relative to the user's position in the Unity *Scene*, given by the *CameraOffset GameObject*, as in (3.3).

$$Steering\ wheel\ radius = |A - B| \tag{3.2}$$

$$Steering\ wheel\ position_{x,y,z} = A_{x,y,z} + cameraOffset_{x,y,z} \tag{3.3}$$

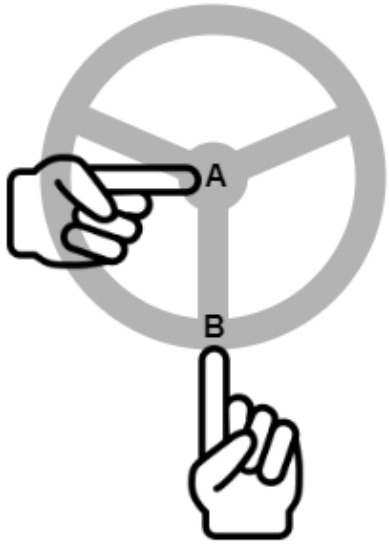


Figure 3.10 Diagram showing the user the calibration process of the steering wheel

Other methods of calibration can be implemented, even saving the user from performing the calibration themselves, for example having the VR headset recognize the steering wheel through the external cameras and adjusting the virtual position automatically. Even though this may achieve a more realistic representation, it is a much more complex alternative, and the similar result would not justify the different implementation, in the first approach of this project.

3.3.3 *Vehicle GameObject*

The *Vehicle GameObject* and its children are responsible for the vehicle's movement and representation. The *GameObjects*' structure is represented in Figure 3.11.

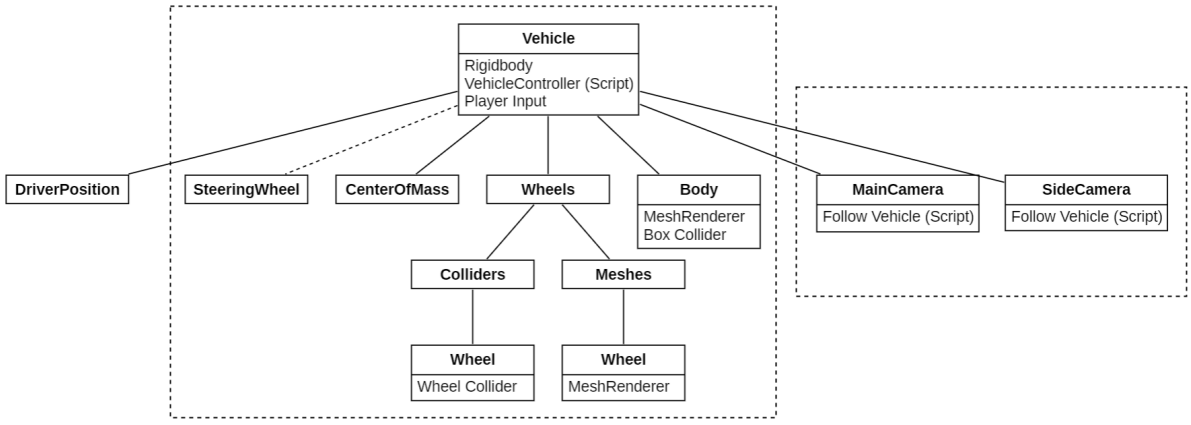


Figure 3.11 Vehicle GameObject structure

The *Vehicle GameObject* is composed by three essential components, the *Rigidbody*, the *PlayerInput* [45] and the *VehicleController* script. The *Rigidbody* component was already ad-

dressed in Section 3.2, the *PlayerInput* is a Unity component and is responsible for handling the user's input and making it possible to use in the script and the *VehicleController* script will be detailed later in this subsection, as it uses the *Vehicle's* child *GameObjects*, which are described below.

The *Body GameObject* is used to represent the vehicle itself in the virtual environment, through the *MeshRenderer* [46] component and to detect collisions with other objects in the environment, using the *Box Collider* component, which was described in Section 3.2.

The *Wheels GameObject* is subdivided into *Colliders* and *Meshes*, each one with a *GameObject* for each of the wheels of the vehicle. As the project is using a simple car, there are four *Colliders* and four *Meshes*. The *Colliders* are responsible for the movement of the vehicle and the *Meshes* are responsible for the representation of the wheels in the virtual environment and rotating them according to the steering.

The *SteeringWheel GameObject* isn't initially a child object, it is only associated after the calibration. If the user does not choose to calibrate, the object is not activated and is not associated with the *Vehicle GameObject*.

The *CenterOfMass* and *DriverPosition GameObjects* provide a way to represent a more realistic vehicle movement and to correctly position the driver's point of view if the VR headset is used.

The *MainCamera* and *SideCamera GameObjects* are *Camera GameObjects* [47], which provide the user with a view from a specific point. The *MainCamera* follows the vehicle from behind, while the *SideCamera* shows a view from the side. Both have the *FollowVehicle* script attached, with different parameters, which allow the *Camera* to keep updating its position.

The structure depicted in Figure 3.11 can be divided into the *GameObjects* that compose the vehicle itself, such as the wheels and the vehicle's body, the camera *GameObjects*, that are attached to the parent *GameObject* for a clearer organization and an auxiliary *GameObject*, the *DriverPosition* that can be used to keep the user's virtual position in place when using VR.

The *VehicleController* script is responsible for everything related to the vehicle's movement, using the different components and *GameObjects* discussed previously. The script provides a public method, *SetUpVehicle*, which is called by the *SceneManager*, giving the script information about the driving mode and the input device.

The flowchart in Figure 3.12 represents the logic followed by the *VehicleController* script. The *VehicleController* class inherits from the *MonoBehaviour* class, and as such uses the

Start and *FixedUpdate*. The *Start* method is responsible for the initialization of some variables, including some regarding the Input System and the *FixedUpdate*, which contains the logic behind the vehicle's movement and will run continuously in a loop until the program ends. The script starts by waiting to be in the ready state, which is achieved when the *SceneManager* finishes the necessary operations and calls the *SetUpVehicle* method. Then, the script can have two different workflows. Either the driving mode sent by the *SceneManager* is a test mode or it is in the free roam mode. In the case of a test mode, the *PerformTestScenario* method will be called, where the vehicle's movement is hard-coded and differs according to the specific test scenario to perform. In the case of the free roam, the user has full control of the vehicle's movement. The script will start by handling the user's input, it will apply an acceleration value based on this input, apply braking if that is the case and adjust the steering of the vehicle. Finally, it will update the wheels to represent the correct orientation and the method will end.

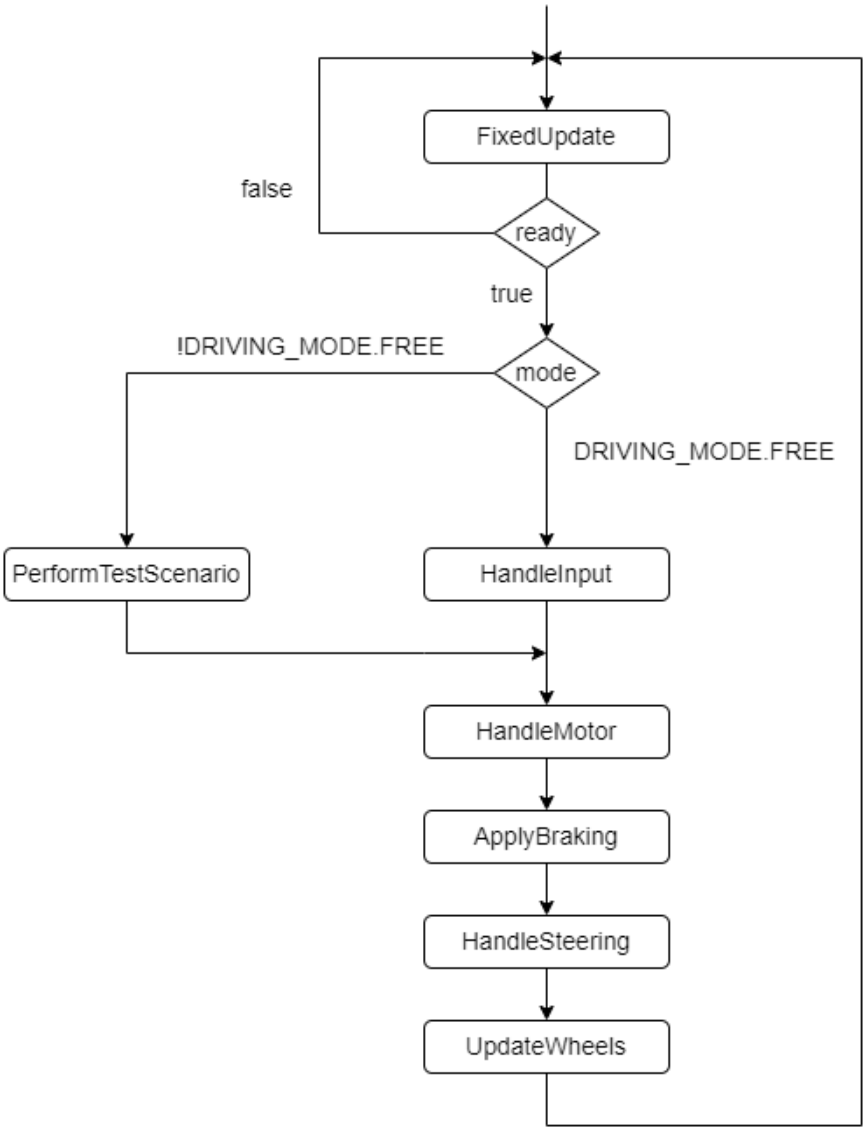


Figure 3.12 VehicleController logic loop

3.3.4 *UI GameObject*

In Unity, UI should be handled through the *Canvas GameObject* [48], which contains a *Canvas* component, a container for everything UI related, such as text boxes, buttons, and other elements the user can interact with. The UI can be designed to be presented to the user in a specific moment, it can also be overlaid on top of the user's field of view or it can even exist in a specific position inside the virtual environment, where the user may or may not see it depending on its position.

The *UI GameObject* serves to group all the different *GameObjects* relative to the user interface, which are represented in Figure 3.13. These are divided by functionality and appear at different stages of the application.

- *Start UI* - appears at the very start of the application and is used to register the user's settings;
- *Calibrator UI* - helps the user calibrate physical devices, showing a timer to calibrate and how to do it;
- *Error UI* - used to display error messages to the user;
- *Driver UI* - displays information helpful to the user, relative to the driving.

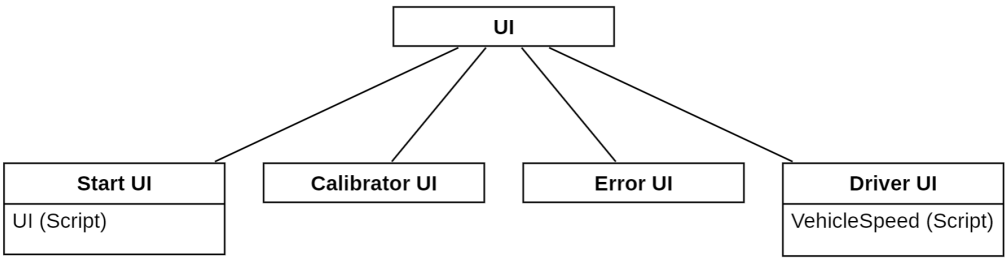


Figure 3.13 UI GameObject structure

The *Start UI GameObject* is responsible for the user interface present at the start of the execution of the application. Besides the aesthetic aspect and making the final product more user-friendly, the presence of a UI allows the user to configure how the application will work, for example, the main input device that will be used, the scenario to be tested etc. This is done through the *UI* script, which is responsible for processing the user's input on the menu. When the Start button is pressed, the script collects the options selected on the menu and sends the information to the *Manager* script, using the created *Enumerations*, so the *Manager* can correctly configure the application. Without the UI, these configurations would have to be performed in the Unity Editor. As the Unity Editor does not provide the performance and flexibility of the resulting executable from building the application, it is better to have the configuration aspect in the executable itself. This also allows for the use of a single executable file, instead

of multiple ones for each configuration or the need to build the application each time a change is made.

Figure 3.14 depicts the start screen of the application, which allows the user to configure the application.

Figure 3.14 Application start screen

The user can change six configurations in total:

- Input Device - The user chooses between the keyboard or steering wheel;
- Output Device - Choosing between using the VR headset or using a regular screen;
- Driving Mode - The user chooses between the free roam mode or one of the testing scenarios;
- Log Values - If selected, the *Logger* class is used and the values relative to the vehicle's movement are calculated and stored in files that can be later analyzed. This option also needs to be activated if the user wants to connect to the platform;
- Connect to Stewart platform - If selected, the application will try to connect to the Stewart platform control software and continuously send the vehicle data;
- Calibrate Steering Wheel - If selected, a second UI screen appears and instructs the user on how to calibrate the virtual representation of the physical wheel.

The *Calibrator UI* is presented to the user when the option to calibrate the wheel is selected. The calibration workflow is detailed in the *Calibrators* section, 3.3.2. The UI displays to the user the timer left to position the fingers and the diagram of how to correctly do so, as shown in Figure 3.15.

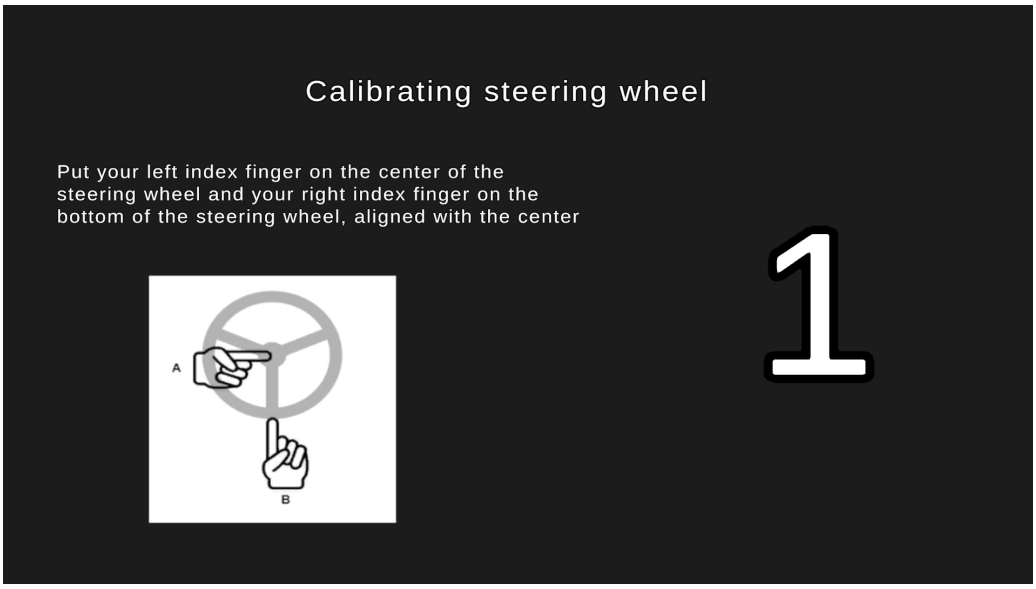


Figure 3.15 Calibrator screen

The *Driver UI* is used to show information regarding the driving to the user, working as a simple car dashboard and is shown in Figure 3.16. It has a single script associated, *VehicleSpeed*, that is associated with the vehicle and calculates and displays the speed.

Figure 3.16 Driver UI

The *Error UI* is used when there is an error with the application. Separated from the *Driver UI GameObject*, as it is intended for information and errors useful to the user regarding the Unity application and the other components, such as the Stewart platform control software. In other words, information for the user of the project as a whole, instead of the user as the driver. The *Error UI Canvas* only has a text box that can be modified by other classes, as exemplified in Figure 3.17.

Figure 3.17 Example error screen

3.3.5 Environment *GameObjects*

The environments where the vehicle can move around are represented by two *GameObjects* that group several other *GameObjects* together. The *Free Roam Environment GameObject* is used when the application is in free roam mode, where the vehicle can move around a setting similar to a real-life setting, in this case, a city with roads, curves, intersections etc. There are several children *GameObjects* which are not relevant to detail. The *Testing Environment* is used in the different testing scenarios, where there's a road and depending on the scenario, different *GameObjects* can be activated by the *SceneManager* and appear, such as speed bumps or a curb.

3.4 Unity project architecture

Figure 3.18 represents the Unity's project full architecture, where the interaction between *GameObjects* can be visualized. As can be seen on the diagram, the most important objects are the *SceneManager* and the *Vehicle*. The *SceneManager* needs to interact with most *GameObjects* as it is responsible for activating and deactivating them as needed, the *Vehicle* is the focus of the project, and it is more used by other objects than it uses them, unlike the SceneManager.

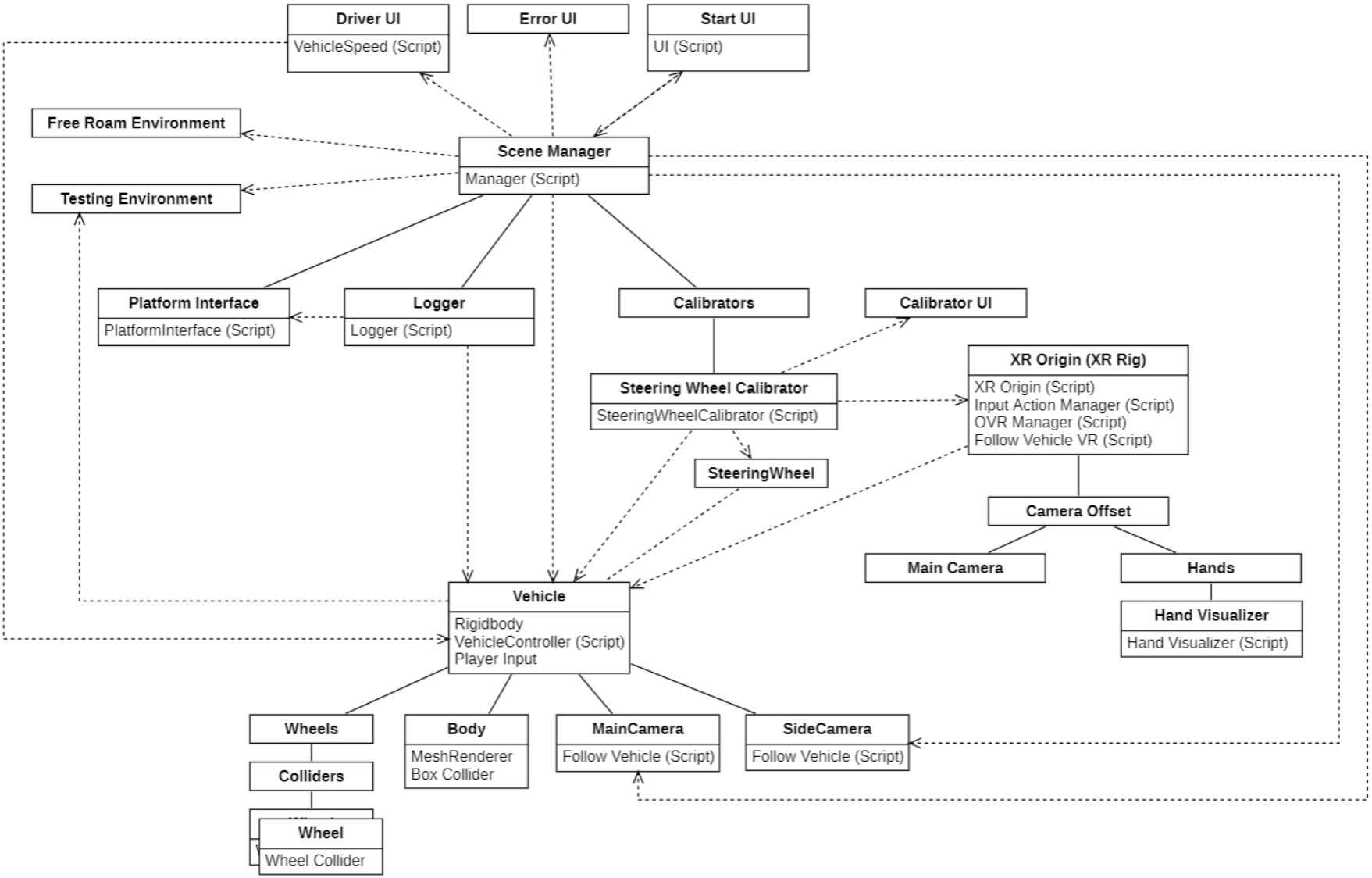


Figure 3.18 Interaction between GameObjects

3.5 Input devices

It is not intended for this project that the driving simulator is strictly linked with a specific input device. Different purposes and environments may require different types of input devices and as such, connecting different devices should be possible and should be done seamlessly.

3.5.1 Integration of input devices with Unity

Unity has always provided a way for the programmer to handle several types of input devices and schemes for a single program without having to handle each device individually and having to change the code every time a new device is supported.

Older versions of Unity used Unity's Input Manager [49] which is part of the core Unity platform, and it was once used in this project. However, it is not very intuitive to use when it comes to adding support for other input devices besides the keyboard. Unity later developed the Input System package, which has to be explicitly added to a Unity project throughout the Unity Package Manager, which provides a much more flexible and easy way to add new devices.

The Input System provides different possible workflows, with different complexities and purposes. The simplest is, consequently, the least flexible, where the programmer directly links the input device with the code, making it harder to add different devices. The chosen workflow requires the creation of an Actions Asset and the usage of the *PlayerInput* component, however, the code remains completely detached from the type of input device, making it easy for the programmer to add support for more devices and for the program to use the different devices with the same code.

The Input Action Asset is an asset that is used to organize the possible types of input, and what each button or key in a device represents, generalizing the actions the program should take. The Input Action Asset is structured into Action Maps, Actions and Bindings.

An Action Map represents a set of actions the user can perform in a specific scenario, for example, in the start screen of the application the press of a key can have a completely different meaning than the press of that same key during the driving of the vehicle. In the case of this project, a single Action Map with the name *Driving* was created.

Inside Action Maps, several Actions can be created, which represent an action the user of the application can take. For this project, the actions created were the possible actions a driver can perform: *i)* Steering; *ii)* Clutch; *iii)* Accelerator; *iv)* Brake; and *v)* Reverse.

For each Action, multiple Bindings can be associated and it is with these Bindings that the Input System provides the abstraction of the input device. A binding represents a way of performing an Action with an input device, for example, associating the *Space* key on the keyboard and the brake pedal with the *Brake* Action. The code using the user's input simply uses the name of the Action, in this case, *Brake*, instead of using the input directly, and, regardless

of the device, Unity will convey that the *Brake* Action is being performed.

Each Binding can then have a Control Scheme assigned. In the case of this project, there are two Control Schemes, Keyboard and Joystick, which are the steering wheel and foot pedals. Unity can then detect during the execution of the application which input device is being used, and then change the Control Scheme accordingly. In this project, as the input device will not change during the execution of the application and to minimize errors, the option to automatically switch Control Schemes was deactivated and the Control Scheme is chosen at the beginning.

Finally, Bindings can have Processors associated, which serve to manipulate the values coming from the input devices. Custom Processors can be created using scripts, but there are also plenty provided by Unity, such as *Clamp*, *Invert*, *Normalize* etc [50]. Besides these Processors, a custom *Shift* Processor was created, to make the accelerator pedal and the up arrow on the keyboard to have the same behaviour, as they have the same purpose. Without using this Processor, the code would have to be modified and check which input device was being used, which would result in less flexible software. This way, the input values are always within the same range of values, and when handled by the *VehicleController* script, the range is always the same, independently of the input device.

An image of the configuration screen of the Input Action Asset is in Figure 3.19, where all the Actions and Bindings can be seen, as well as an example of the usage of Processors.

The Input Action Asset is stored in a file with the *.inputactions* extension which stores data in the JSON format. An excerpt of this file is shown in Figure 3.20. The excerpt shows the configuration of the *Steering* Action with the use of the keyboard. This Action is then associated with a few Bindings, the *Arrows, Negative* and *Positive* Bindings. This configuration means that the left and right arrows, identified in the Binding by the *path* property, are used to handle the steering of the vehicle when the *Keyboard&Mouse* Control Scheme is selected. In the complete file, this is repeated for each Action in a Control Scheme.

The *PlayerInput* component is attached to the *Vehicle GameObject*, as the input will be handled within the *VehicleController* script that is associated with it. Its main purpose is to link the Input Action Asset to the *GameObject* that will use it. It can also define the control scheme and action map, however, the control scheme is defined by the script and there is only one action map.

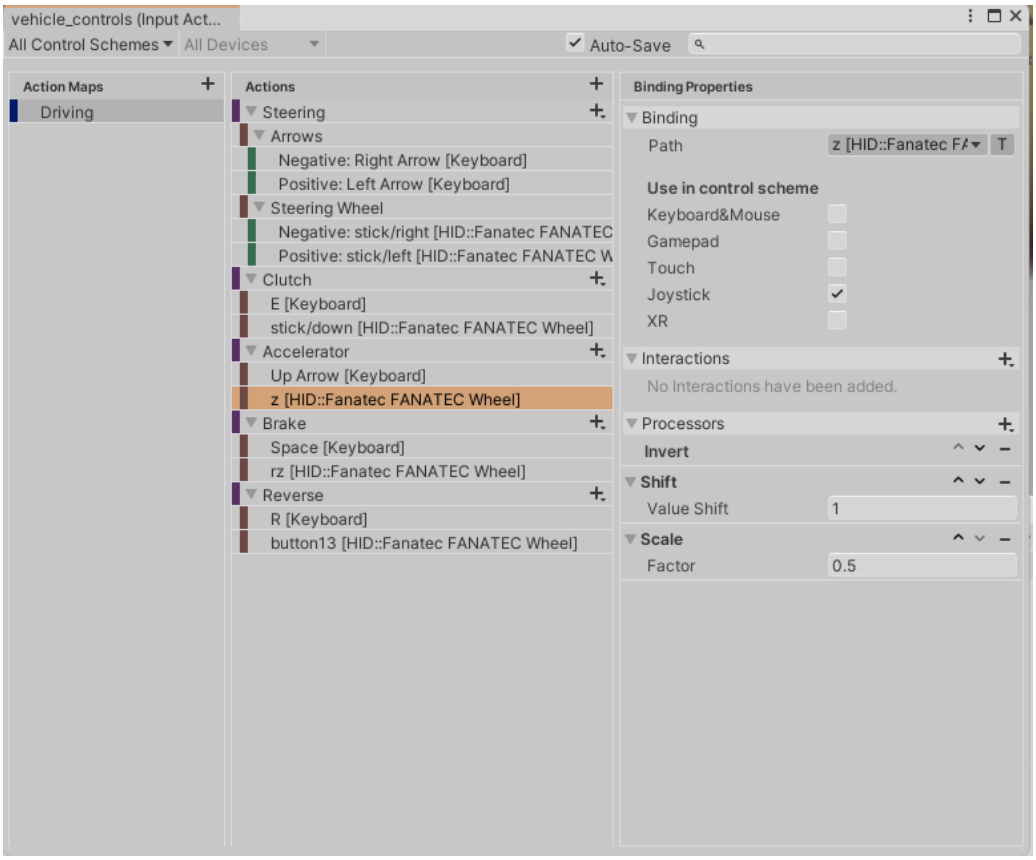


Figure 3.19 Input Action Asset configuration window

```
{
    "name": "vehicle_controls",
    "maps": [
        {
            "name": "Driving",
            "id": "dab797be-a155-4f21-9652-8b428af0f2d4",
            "actions": [
                {
                    "name": "Steering",
                    "type": "Value",
                    "id": "65bf94c5-92d4-4058-a6cd-a822a661b043",
                    "expectedControlType": "Axis",
                    "processors": "",
                    "interactions": "",
                    "initialStateCheck": true
                },
                (...)
            ],
```

```
"bindings": [
    {
        "name": "Arrows",
        "id": "004d306d-992a-4764-9aa7-478b7ca1c4bf",
        "path": "1DAxis",
        "interactions": "",
        "processors": "Invert",
        "groups": "",
        "action": "Steering",
        "isComposite": true,
        "isPartOfComposite": false
    },
    {
        "name": "Negative",
        "id": "7966659a-af53-4907-bad9-596f835a7e89",
        "path": "<Keyboard>/rightArrow",
        "interactions": "",
        "processors": "",
        "groups": "Keyboard&Mouse",
        "action": "Steering",
        "isComposite": false,
        "isPartOfComposite": true
    },
    {
        "name": "Positive",
        "id": "3be42952-8f86-4844-85d3-81bd41accc49",
        "path": "<Keyboard>/leftArrow",
        "interactions": "",
        "processors": "",
        "groups": "Keyboard&Mouse",
        "action": "Steering",
        "isComposite": false,
        "isPartOfComposite": true
    },
```

```
                    (...)
                ]
            },
            (...)
        ],
        (...)
}
```

Figure 3.20 *vehicle_controls.inputactions* file

3.5.2 Types of input devices

A few different input devices are used in this project, such as: Keyboard; VR Headset; Steering Wheel; Pedals; and Joystick. However, some of these devices may not fit exactly in the Input System's supported input devices list [51], for example, the steering wheel is identified as a Joystick and requires some configuration. From the list of devices used in the project, only the VR Headset isn't handled through the Input System.

The items on this list are not necessarily equivalent, which means that they may serve different purposes and work in different combinations. For example, the VR Headset may be used in combination with the Steering Wheel and the Pedals to provide the most realistic experience, but in a more testing-oriented environment, the keyboard may be more appropriate and easier to use.

The keyboard is used to function as the steering wheel and the pedals. Five keys can be used to control the vehicle:

- Left arrow - steer left;
- Right arrow - steer right;
- Up arrow - accelerate;
- Spacebar - brake;
- R key - put the vehicle in reverse.

The steering wheel is very straightforward and works as a real one. The steering wheel can perform one and a half turns to each side, corresponding to a total of three full turns, or 1080 degrees. The rotation is transformed into real values, which can have any value in the range of 1 to -1. When the steering wheel is fully to the left, it is represented as 1 and when it is fully to the right it is represented as -1.

Like the steering wheel, the pedals are also very straightforward, both the accelerator and the brake pedal provide input from 0 to 1, as real numbers, after the input values are adjusted to more user-friendly values using the Input System's Processors.

The gear stick is only used to put the vehicle in reverse.

3.6 System Deployment

When the project is in its development phase, it is common and more practical to test it through the Unity Editor. However, when deploying the product or for a better testing experience and performance, it is important to build the project and run the executable. This also allows a user to run the project without needing to have the Unity software installed.

Unity allows the option to build the program for several different platforms, but for this project, the only ones that could be applicable are Android and PC (Windows, Mac, Linux), which means the project could run on the dedicated computer or the VR headset. The VR headset used in this project was the Oculus Quest 2 [52], which does not need a computer to be used and runs Meta Horizon OS [53], an Android based operating system and therefore could be used to run a Unity executable.

The chosen option was the PC build, which results in an executable file - *.exe* - as it is the more logical approach, it is more independent from external devices and it is far more simple to implement without having to compromise any features. The Android build approach also has several cons that will be discussed in the corresponding section.

With the PC build, the dedicated computer is responsible for running the Unity project executable and is physically connected to all the input devices. Besides the keyboard, the steering wheel and the pedals are connected via USB and also the VR headset is connected via USB. The computer would also be responsible for executing the MATLAB library and for the communication with the Stewart Platform.

A diagram of the connections made in the project is shown in Figure 3.21.

With the Android build, the Oculus Quest 2 would be the one responsible for the computing. However, the Quest headset wouldn't be able to handle the platform controller side of the project, which means there would always be the need for a computer to handle it. The headset and the computer would have to communicate with each other via WiFi, which could add unnecessary delay and room for errors. These aspects alone are enough to discard this option, however, the Quest headset would also have to have the steering wheel connected to itself, which it doesn't support.

When choosing the platform in Unity's build settings, Windows, Mac and Linux are grouped together, however, the specific target platform needs to be specified, as different files are produced. The chosen target platform was Windows, as it is the operating system of the computer dedicated to the project.

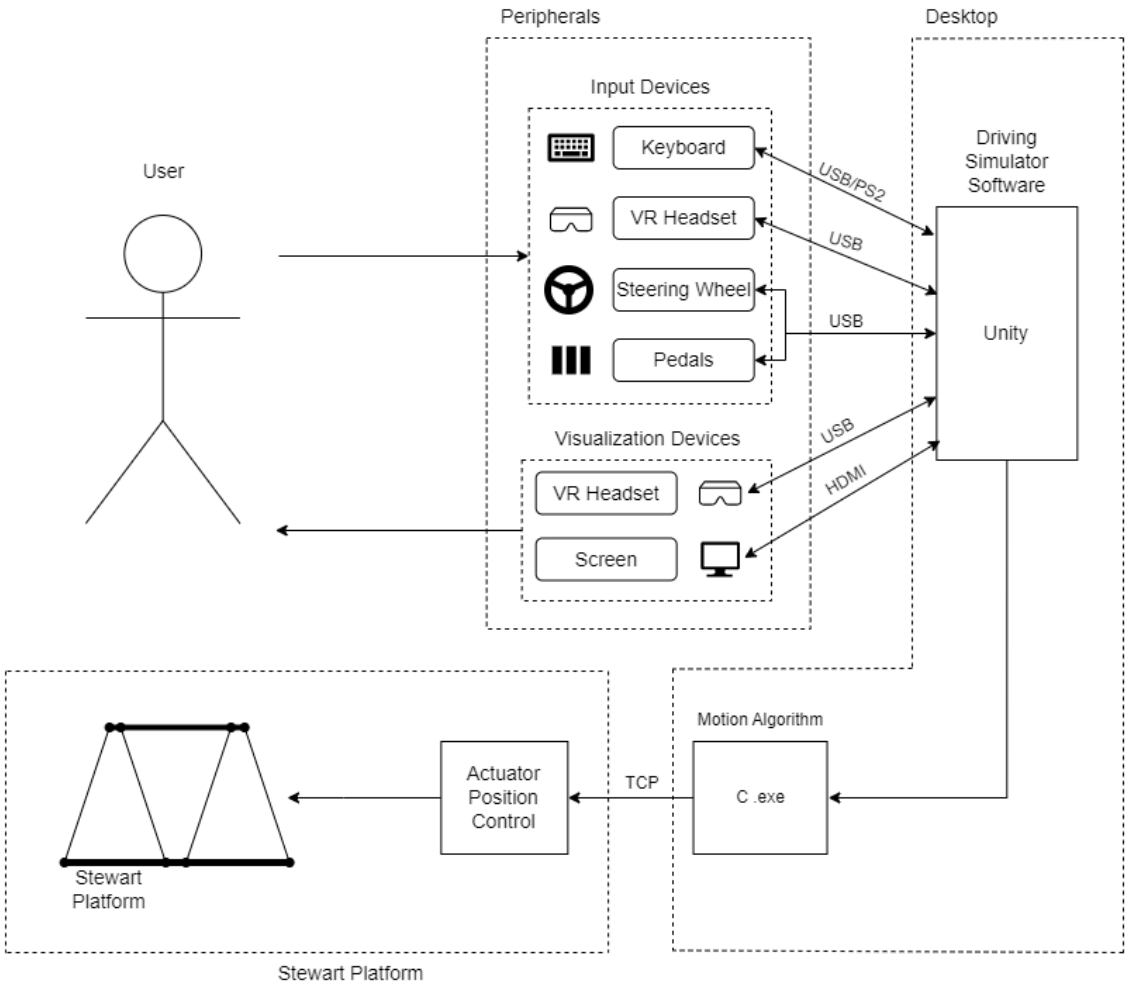


Figure 3.21 Components connections in a PC build

The build process in Windows creates the files and folders listed below, which are required for the application to run, with the exception of the last of the list [54]. The project name was set as *aceleracao*.

- aceleracao_Data - contains all the data needed to run your application;
- *UnityPlayer.dll* - contains all the native Unity engine code;
- *aceleracao.exe* - the program itself, which uses the Unity engine;
- *UnityCrashHandler64.exe* - the application that runs in the background and reports when the application crashes.

Chapter 4

Stewart platform

A Stewart platform is a type of parallel robot, which is a machine composed by a mobile platform, connected to a fixed base by a set of identical legs [55]. The first ideas and designs for a parallel robot date back to the 1930's, where James E. Gwinnett applied for a patent to build a rotational platform, seen in Figure 4.1 capable of tilting according to the movement represented on a movie [56]. This design is one of the first to present a machine with multiple degrees of freedom, however, the machine was never built. Around the same time, Willard L.V. Pollard designed a parallel robot which had three arms, rotary motors and supported five DoF to automate spray painting [57] [58], seen in Figure 4.2.

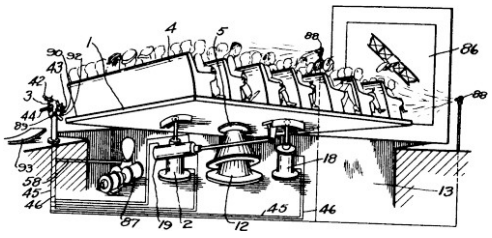


Figure 4.1 James E. Gwinnett's platform

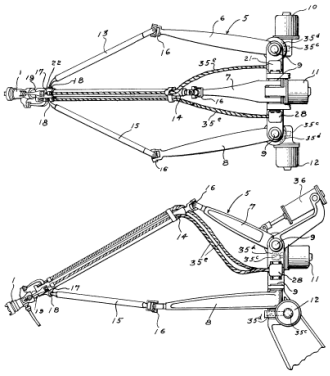
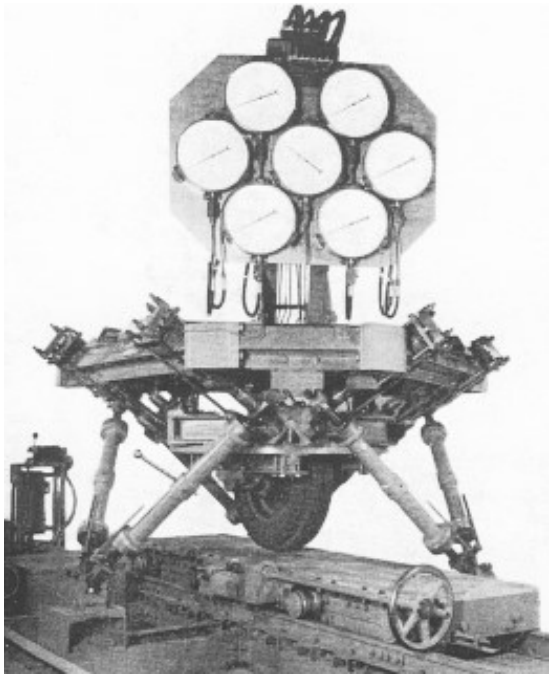


Figure 4.2 Willard L.V. Pollard's spray gun

In the late 1940's, a six leg platform was designed and built by Eric Gough, an engineer from Dunlop Rubber, a tire company. This platform, called "Universal Tyre-Testing Machine", was built with the objective of testing airplane tires and to study their properties, and it has many similarities with the Stewart platform. This model can be seen in Figures 4.3a and 4.3b.



(a) In 1954

(b) In 2000

Figure 4.3 Original Gough platform

Finally, in 1965, D. Stewart published the article "A Platform with Six Degrees of Freedom", where he presented a six DoF motion platform to use as a flight simulator and caused a great impact in the field of parallel kinematics [58]. In the article, Stewart proposed the platform as a solution to the lack of a test platform capable of simulating all degrees of motion, offering translation and rotation movements in all three axes, both individually or in combination [59]. The use cases described in the article were mainly for simulating vehicles, such as cars, ships, helicopters and airplanes but also for tools and machines. Not all of the design principles Stewart initially identified could be implemented, but the platform was designed with the goal of using only six motors operating on the same load. The original design aims and respective purposes are described in Figure 4.4.

Design aim	*Purpose*
(1) The use of not more than six motors.	To avoid redundancy and reduce cost
(2) Each motor reacting on the foundation.	To avoid interaction between motors.
(3) Each motor operates directly on the same load.	To achieve the maximum performance for a given power source.
(4) High pay load/structure weight ratio.	To achieve the maximum performance from power available.
(5) Each motor identified with one motion.	Simplicity of control.
(6) Low friction motions.	To reduce power losses and to obtain high response.

Figure 4.4 Original design aims and respective purposes

The original platform consisted of a triangular movable platform attached to three extensible legs. Parts of the design are depicted in Figures 4.5 and 4.6.

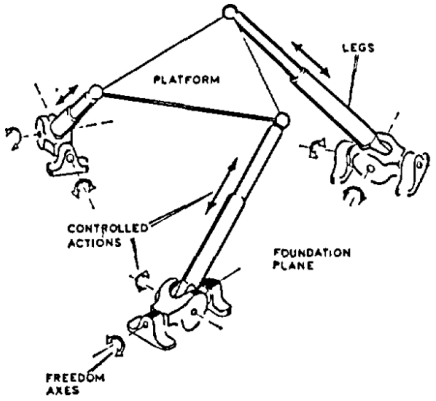


Figure 4.5 Design of the original Stewart platform

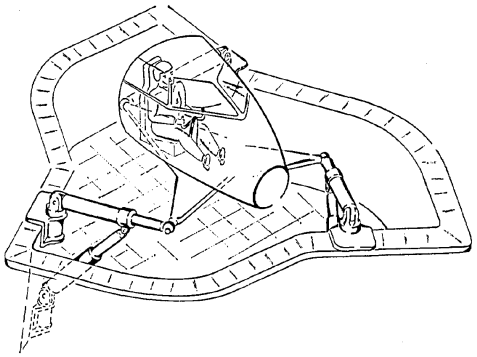
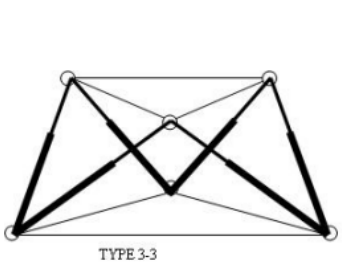


Figure 4.6 Original Stewart platform used as a flight simulator

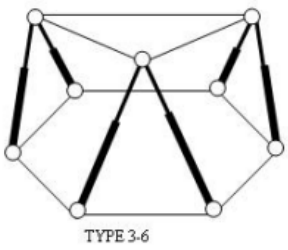
Since the publication of Stewart's article, the world of technology suffered huge advancements as well as the field of mechanical engineering, however, the modern Stewart platforms don't deviate much from the original design. What defines the Stewart platform is the fixed platform connected to a movable platform by six variable-length legs.

In the modern day, Stewart platforms have several utilizations in different areas, mainly for simulating purposes, such as flight or driving simulators, medicinal and research purposes and even in docking systems for space vehicles.

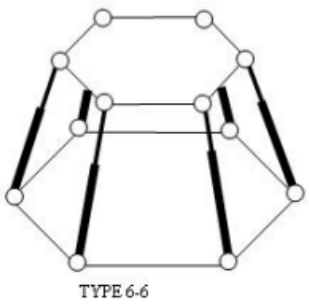
Stewart platforms can be classified into three categories, depending on the number of distinct attachment points in the base and the top plate [60]. The different types of Stewart platforms are illustrated in Figures 4.7a, 4.7b and 4.7c. A Stewart platform of type 3-3 has three attachment points in both plates, a platform of type 3-6 has three attachment points in the top plate and six in the base plate, whereas a platform of type 6-6 has six attachment points in both plates.



(a) Type 3-3



(b) Type 3-6



(c) Type 6-6

Figure 4.7 Types of Stewart platform

As previously mentioned, a Stewart platform allows what is called six degrees of freedom - 6 DoF. These six degrees are the six different types of movement an object can perform in a three-dimensional space:

- Translation movements
  - x axis: Surge (forward/backwards)
  - y axis: Sway (left/right)
  - z axis: Heave (up/down)
- Rotation movements
  - x axis: Roll
  - y axis: Pitch
  - z axis: Yaw

A visual representation of these types of movement is shown in Figure 4.8.

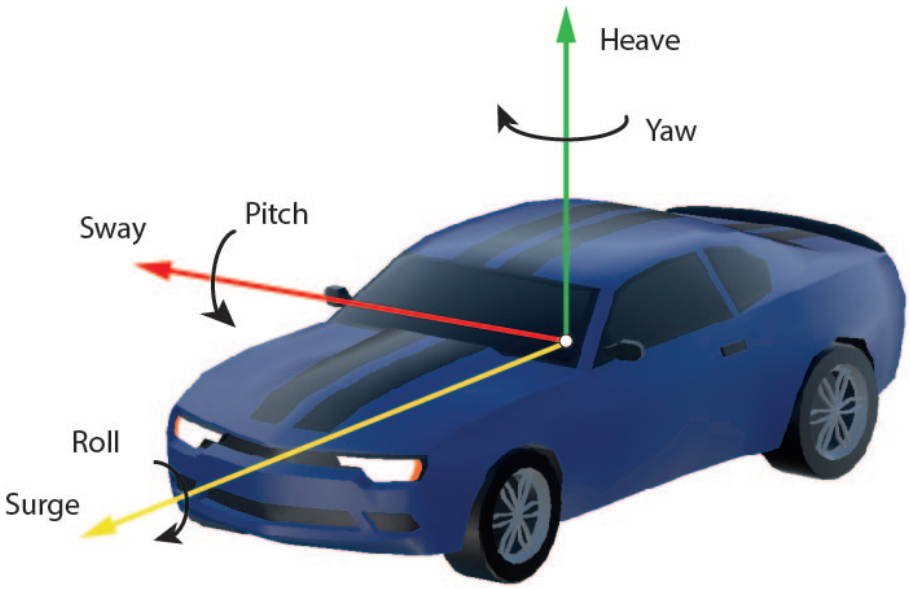


Figure 4.8 Representation of the six degrees of freedom

4.1 Model used

In this project, the Stewart platform interacts with the user of the driving simulator in a passive mode. Unlike input devices, the user does not provide commands directly but instead receives the physical sensations generated by the platform's movement. The user sits on the platform's seat and experiences the simulated motions and forces, which are reproduced in response to the behaviour of the vehicle model used in the simulator.

The physical structure of the Stewart platform used in this project is presented in Figure 4.9. The platform has a seat attached to the movable upper plate, where the user will sit, along with other devices such as the steering wheel and pedals. The platform used is of the 6-6 type, as it has 6 attachment points at the base and another 6 attachment points at the top. It consists of two irregular hexagonal structures, each with three equal shorter sides and three equal longer sides. The mathematical model associated with the platform's movements is developed based on the positions of these attachment points, i.e., the connections of the linear actuators with universal joints. This mathematical model is adjusted to control the upper plate where the chair is fixed. As shown in Figure 4.10, there is a considerable distance between the 6 attachment points at the top and the upper plate that is to be controlled. This adjustment has been incorporated into the final model of the Stewart platform (inverse kinematics), ensuring that it conforms to the physical dimensions of the structure.

Tables 4.1 and 4.2 present a summary of the characteristics of the Stewart platform that was used in this project, namely the weight and thickness of the fixed and movable plates and the physical limits of the movements of the six degrees of freedom.

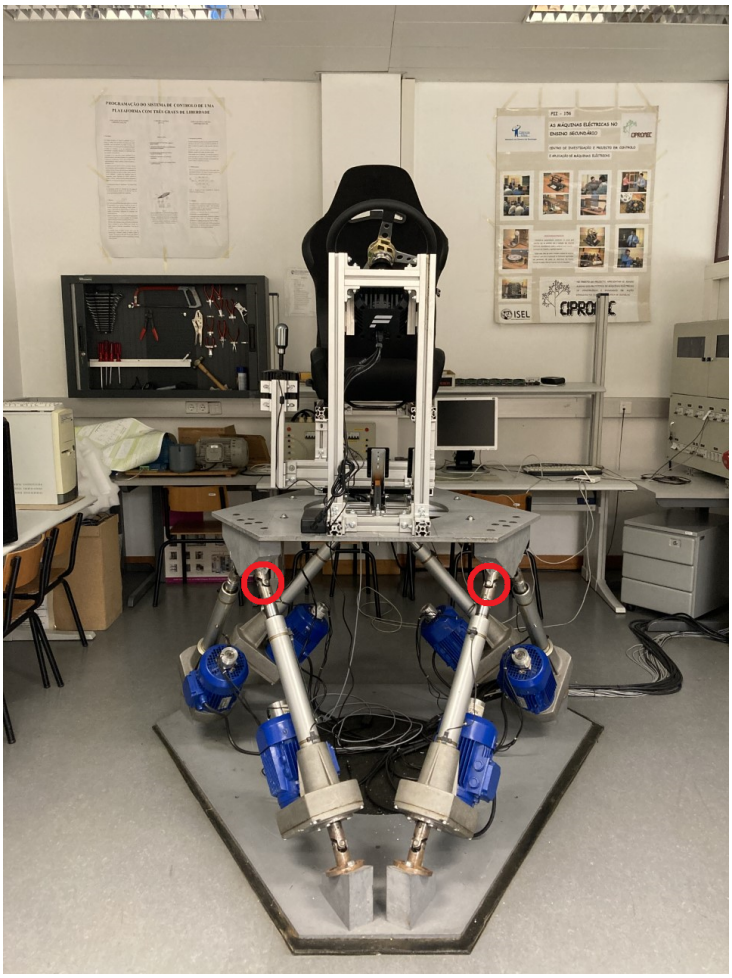


Figure 4.9 Stewart platform to be used in this project

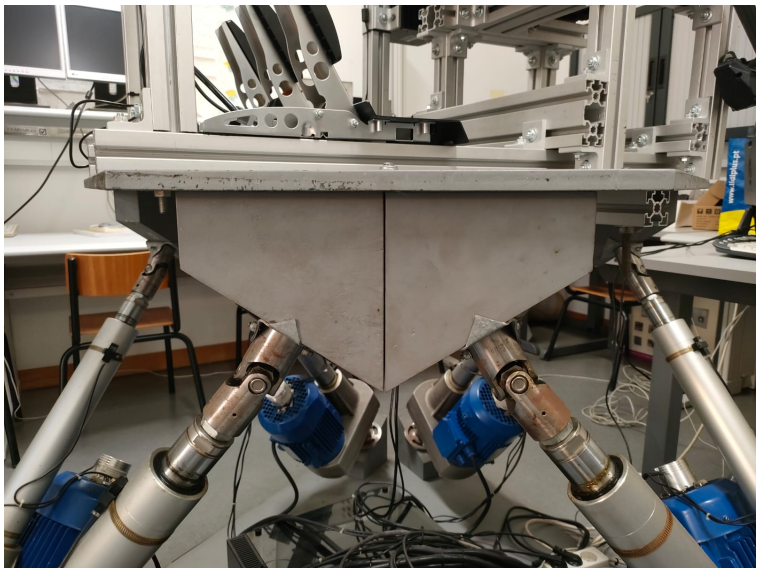


Figure 4.10 Sideways view of the platform's top plate

Table 4.1 The mass and thickness of both platforms [6]

	Mass (kg)	Thickness (m)
Movable platform	119.5	0.05
Fixed platform	164.3	0.05

Table 4.2 Movement metrics for the Stewart Platform [7]

Movement	Amplitude	Speed	Acceleration
Pitch	24 º	65 º/s	165 º/s^2
Roll	20 º	70 º/s	175 º/s^2
Yaw	28 º	100 º/s	100 º/s^2
Heave	0,30 m	0,50 m/s	1,30 m/s^2
Surge	0,26 m	0,80 m/s	2,00 m/s^2
Sway	0,22 m	0,70 m/s	1,80 m/s^2

4.2 Stewart platform control software

The path between the driving simulator software and the Stewart platform is not as straightforward as it may seem. The platform is not ready to handle the data as it is sent from Unity. The acceleration values are related to Unity's virtual environment, and cannot be directly interpreted in a real, physical environment as the platform, which exists in a limited space with limited movement capabilities, as opposed to Unity, which has a virtually infinite. For this reason, between the driving simulator software and the Stewart platform, a software module is required to implement a conversion process. This module, after handling the acceleration data,

sends it to the platform's internal software which converts it to position values for each of the actuator's legs.

At the initial phase of the project, this control module, responsible for converting the acceleration and rotation values of the virtual vehicle was already implemented in MATLAB Simulink, depicted in Figure 4.11.

MATLAB Simulink is a tool used to model and simulate systems. It provides an interface where the user can use different blocks with different functionalities, from different libraries, connect them together and build a system that can then be deployed or used to generate code [61].

The Simulink model has three main responsibilities:

- Receiving data from the driving simulator software, which works as a TCP client, as opposed to the Simulink project, which is running a TCP server to receive the data sent by the previous module;
- Converting acceleration and rotation into the Stewart platform's legs' position;
- Sending the calculated positions to the Stewart platform itself.

One of the objectives of this project is to instead of directly using the existing Simulink model, to generate C or C++ code and to use it as a library. This allows the usage of the model to be decoupled from the usage of MATLAB software. One other benefit from this is that executing the code is more efficient than running the model, as it is already compiled.

When converting the model into reusable code, there are a few approaches that can be considered. C code can easily be compiled into an executable file, however, in this case, an executable file would reduce the code's reusability, as it would associate the functionality of the model to the communication protocol used and it would cause a TCP server to always exist and be running when using the executable, limiting the capabilities of the model and making it more difficult to test. For this reason, it is better to decouple the functionality of the model from any communication protocols when creating a library, restricting its functionality to handling data and creating an external program to use the library. It is also intended that this code library receives parameters about the specific platform to use, in order to have a more generic solution.

When it comes to libraries in C, there are two types, static and dynamic libraries. Static libraries are linked when compiling a program, whereas dynamic libraries are linked at runtime. For this reason, in the case of static libraries, the library code is grouped together with the program's code in the final file, while in the case of dynamic libraries, the code is separated, guaranteeing a smaller final file and allowing the library to be used by different programs at the same time. Dynamic libraries, as they are only loaded at runtime, have the advantage of being able to be modified without requiring the program that uses them to be recompiled, which

happens in the case of static libraries. On the other hand, static libraries are faster to load and can be considered safer, as they are contained in the final executable file [62].

Some of the characteristics described may be irrelevant when considering the needs of the project, however, the option of using a dynamic library was chosen, mainly for the ability to modify it without having to recompile the external program. The library may be frequently subject to changes in order to test different variables or other details and the external program responsible for the communication with the platform will rarely be modified.

As this project is using a computer with the Windows operating system, the dynamic library is referred to as a Dynamic Link Library (*DLL*) [63].

4.2.1 Evolution of the control software

Even though one of the main objectives of this final project was to convert the existing model into a more flexible and easier to use library, the Simulink software provides several valuable features that a code library does not and that can be useful when working on later versions of the control software. For example, Simulink can plot the movement data in easier to understand and to explain graphs or it may be easier to understand the flow of the received data with a visual representation of the model. For these reasons, besides providing a way to easily generate a code library, the existing model was improved, as both versions may be useful in different scenarios.

The first minor change that was made in the model, before initiating the process of creating a workflow to generate the *DLL*, was to change how data was received. In the original version of the model, the data was received through a TCP/IP Receive block [64], which only allows the reception of data from a specific IP address. This was changed to a TCP Receive block [65], which is similar, but is not associated with an IP and allows for any client to connect to it, in combination with a TCP Server block [66], which creates and configures the server itself. The first option may work in scenarios where the TCP client always has the same IP address, but it would always be less flexible.

The MATLAB Simulink software provides several applications that can be used to analyze, test or complement the model. Some of these applications can be grouped in the Code Generation category, which can be used to convert MATLAB code or Simulink models into code in another language to use in applications other than MATLAB [67]. However, from the extensive list of Code Generation applications, only three are listed as capable of generating C or C++ code, which was a prerequisite of the project: MATLAB Coder, Embedded Coder and Simulink Coder. The Embedded Coder is listed as an extension of MATLAB Coder and Simulink Coder [68], so it was the chosen application to use to generate the code library.

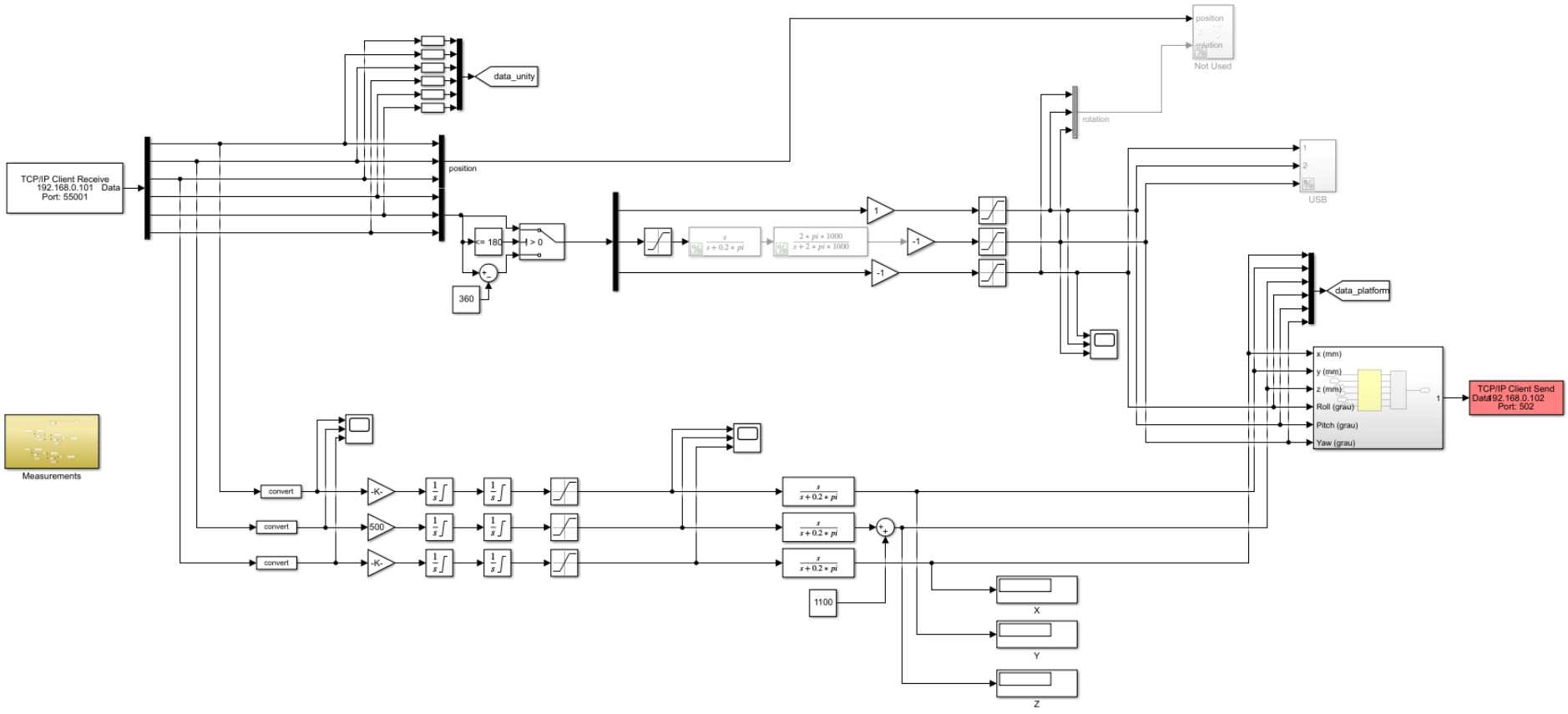


Figure 4.11 The first version of the MATLAB Simulink model

In order to achieve the desired final product, which is a flexible C *DLL*, several configurations need to be performed. These can be made in the Code Generation section of the Embedded Coder's settings and will be detailed below. The settings are saved and used every time the user generates the *DLL* either through MATLAB's interface or through the batch file.

The system target file is a file that controls the code generation stage of the build process [69]. This is one of the main configurations to be made and it decides what type of files are generated. MATLAB provides an extensive list, but only the option *ert_shrlib.tlc* provides the user with the ability to generate a shared library dependent of the operating system. In the case of Windows, used in this project it generates a Dynamic Link Library, a *.dll* file, but for UNIX based operating systems, it generates a shared object, a *.so* file [70].

Regarding the build process, it is important to select the option to "Package code and artifacts", as it allows for a bigger independence from the MATLAB software and for an easier relocation process of the generated artifacts [71]. It is recommended to use this functionality when the development environment doesn't have the MATLAB software [72]. It is also important to configure the C compiler to be used, in the case of this project, the MinGW-w64 [73] compiler was used and was the only one tested. The configuration window with these settings can be seen in Figure 4.12.

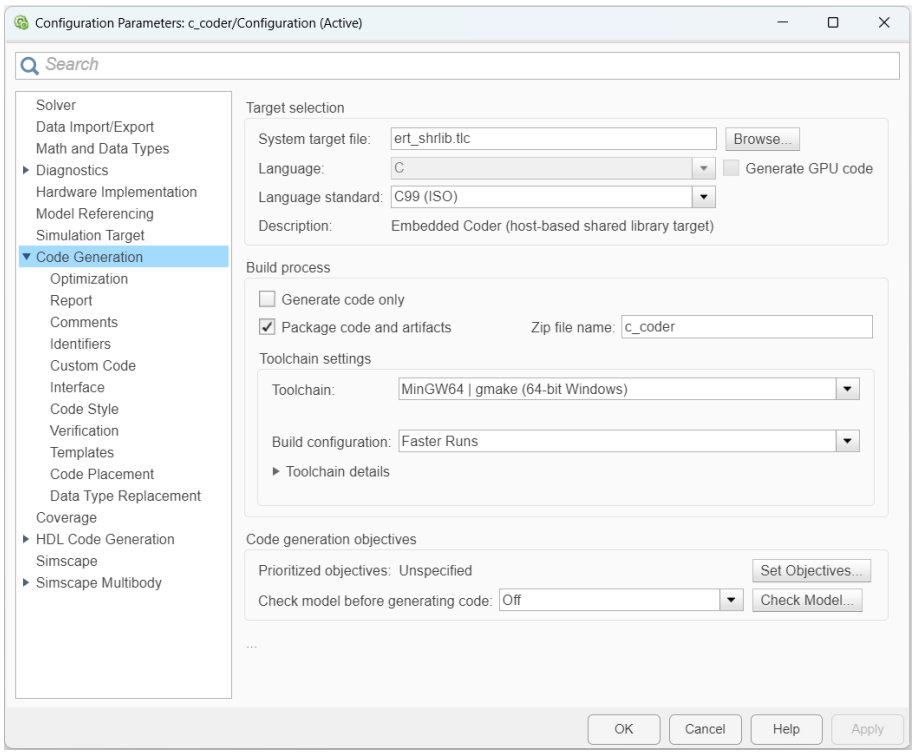


Figure 4.12 Code Generation settings window

The generated code contains two main functions of the model, *c_coder_initialize*, which initializes variables and *c_coder_step* which iterates the model one time. Custom code was used in the code generation in order to use these functions in external software and to easily debug the final application. The utilized code is in files *matlab_code.c* and *matlab_headers.h* and consists of wrapper functions for the generated initialize and step functions, that are ex-

ported by the *DLL*.

A few other settings needed to be changed in order to achieve the intended outcome. Some were related to the model's optimization and others to the code's reusability.

Some blocks used in the original Simulink model, such as blocks related to TCP communication, are not supported by code generation tools and have to be omitted from the model in order to generate the code.

Even though the TCP blocks had to be removed in order to generate the *DLL*, it is a better approach to decouple the data handling, which is the main functionality of the model, from any communication protocol. This way, the *DLL* has a single responsibility, transforming the virtual values from the simulator into real values usable by the platform. Another software can then communicate with the simulator, use the *DLL* to handle the data and then send it to the platform, benefiting from the library's reusability and flexibility. The communication protocol can then be modified in the external application without interfering with the model's logic.

As the generated code is in C, to facilitate the usage of the library, the external application that is responsible for the communication with the simulator and the platform was also written in C. This program essentially has the same three responsibilities as the original Simulink model, it opens a TCP server for the simulator to connect to, it processes the data, this time while making use of the *DLL*, and connects a TCP client to the software responsible for the platform's legs' movement. To improve the ability to test every component of the whole project individually, and to allow the testing of the library without needing to use the platform, the application provides a way to receive and process the simulator's data without sending them to the platform. This application is compiled into an executable file (*.exe*).

To more easily use and configure this application, the batch file *run_platform_interface.bat* was created, which uses the *platform_tcp_parameters.txt* text file, which contains the necessary configurations for the execution of the application. The file contains the following variables:

- *OPEN_PORT* - the port where the application's server will listen to clients, more specifically, the port where the simulator will connect to;
- *PLATFORM_IP* - the IP address where the platform is running;
- *PLATFORM_PORT* - the port where the platform expects clients to connect;
- *MOCK* - boolean variable, if true, the application does not attempt to connect to the platform and is in test mode, if false, the application tries to connect to the platform and sends the processed data.

The flow of the data when using the previously described components is described in Figure 4.13.

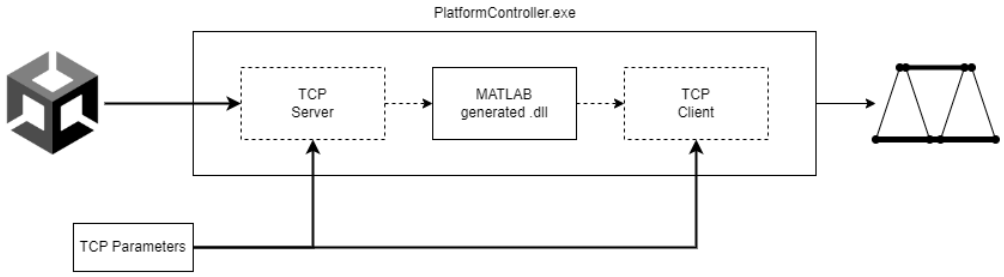


Figure 4.13 Utilization of the *DLL* in combination with the executable

To simplify the process of generating the library, the *generate_dll.bat* file was created. This file uses a MATLAB script to run the build process on the *c_coder.slx* file. When building the library, MATLAB runs the *init_simulink.m* script to initialize some variables used in the model. If the variables are modified, the *DLL* has to be generated again for them to take effect. The build process creates some additional files and folders, however, only the *.dll* and *.zip* files are relevant. The library, *c_coder_win64.dll* is the library to be used by other programs and *c_coder.zip* contains C header files that have to be included in C programs when using the library. The contents of the zip file are extracted and moved to the relevant folder in the C code's directory when executing the batch file. The files and scripts that are created and used by the batch file are depicted in Figure 4.14.

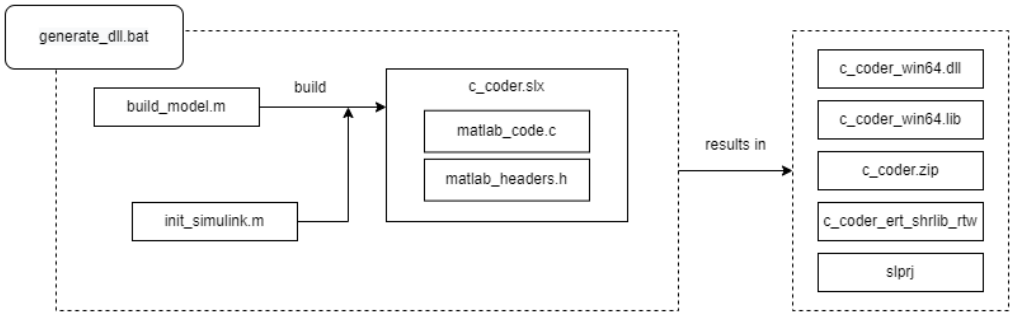
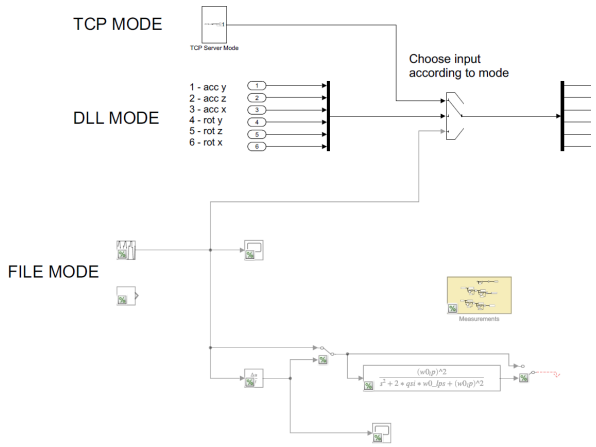


Figure 4.14 Files used and created by *generate_dll.bat*

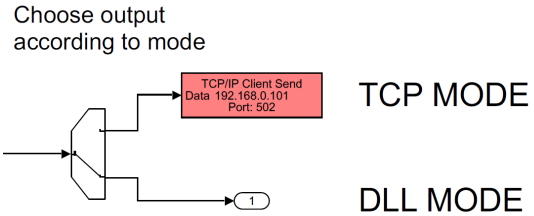
In the final stage of this project, the result is a modified Simulink model, a C *DLL* and a way to easily generate the *DLL* as needed, either because of changes in the model or changes in the variables, through the configurations associated with the model file and the batch file that streamlines the process. The modified Simulink model differs from the initial one in several aspects, but one of the most significant changes is the ability to use the model in three different ways: TCP mode, *DLL* mode and file mode.

As previously mentioned, using Simulink can still be useful, so the user should be able to use the model through Simulink. As the TCP blocks, which are essential for the Simulink usage, have to be omitted when generating the *DLL*, a Manual Variant Source [74] block and a Manual Variant Sink [75] block were added to the model. The Source block enables the user to choose which input will be used, while the others will be ignored by MATLAB when generating code and the Sink block does the same to the outputs. As there is one entry in the Source block for each input mode, the user will have to switch the input accordingly. The Sink block has two outputs, one for the *DLL* mode and the other for the TCP and file mode, which activates the TCP client block. The use of these blocks is represented in Figures 4.15a and 4.15b. The only restriction with this approach is that the user has to manually choose the mode and has to leave the DLL mode selected to generate the *DLL* in the future.

The final version of the Simulink model is in Figure 4.16.



(a) Input selection



(b) Output selection

Figure 4.15 Input and output modes of the Simulink model

For debugging purposes, a C# application was also developed, implementing a TCP server that emulates the Stewart platform, ready to receive the computed data from the control software in the same format as the Stewart platform. This application can be compiled into an executable and executed locally with the default port of the platform.

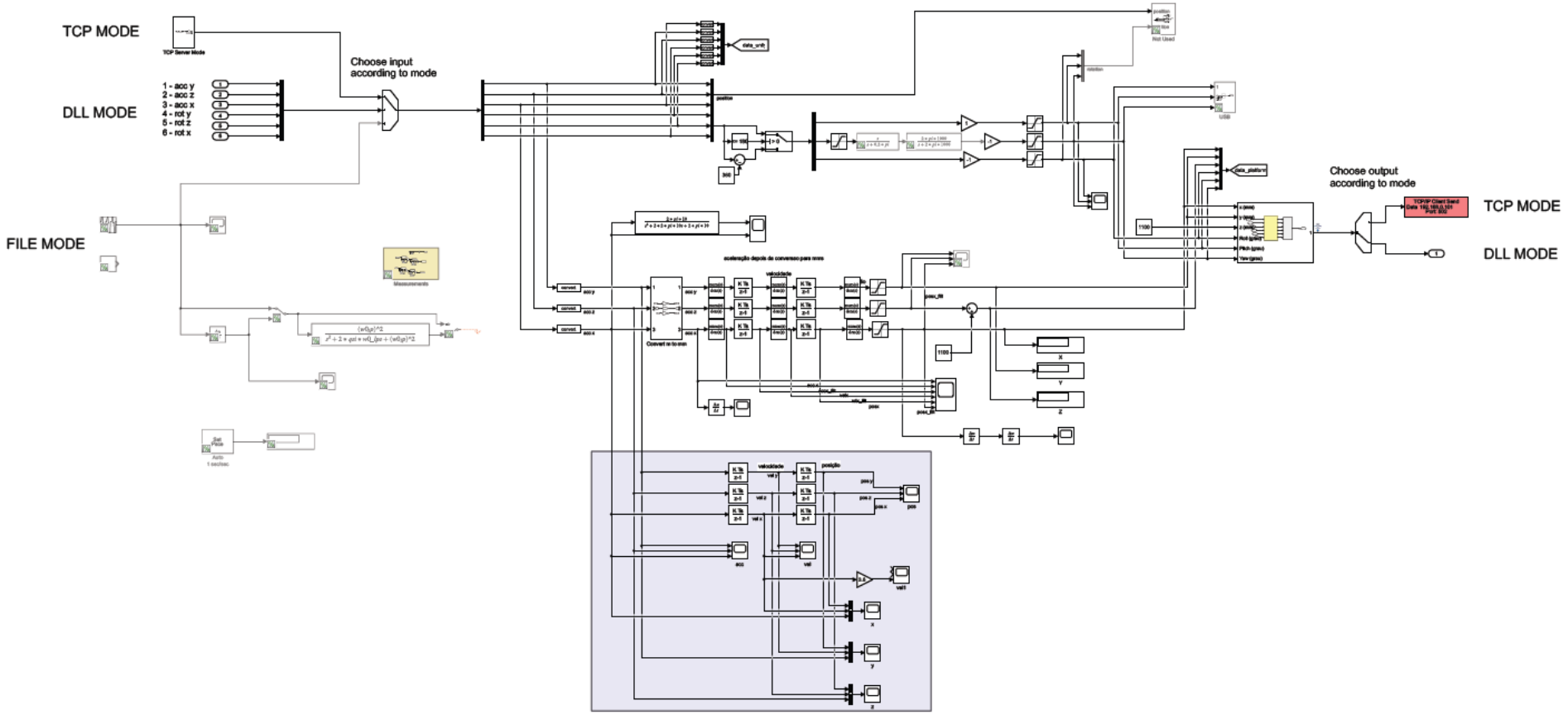


Figure 4.16 Final Simulink model

4.2.2 How the control software works

It is not in the scope of this project to change the way the control software works and it is not essential to fully understand it, however, it is important to have an understanding of its internal modules and more importantly how it handles data, both the inputs and the outputs.

The data that is sent to the control software represents virtual values that are meant to be interpreted in the context of the Unity game engine. The control software is responsible for handling the data and modifying it into values meaningful to the Stewart platform.

The first minor technicality is that Unity does not use the same cartesian coordinate system as the platform. In Unity, the positive x-axis points to the right, the positive y-axis points up, and the positive z-axis points forward [76] whereas in the platform coordinate system, the positive x-axis points forward, the positive y-axis points to the left, and the positive z-axis points up.

In total, six values are sent to the control software. Three values, one for each axis, regarding the acceleration of the vehicle, and three more, one for each axis, representing the rotation values of the vehicle. As depicted in Figure 4.17, which is a simplification of the control software's logic, the acceleration values are processed through a motion cueing algorithm and then are grouped together with the rotation values to be processed in the inverse kinematics module.

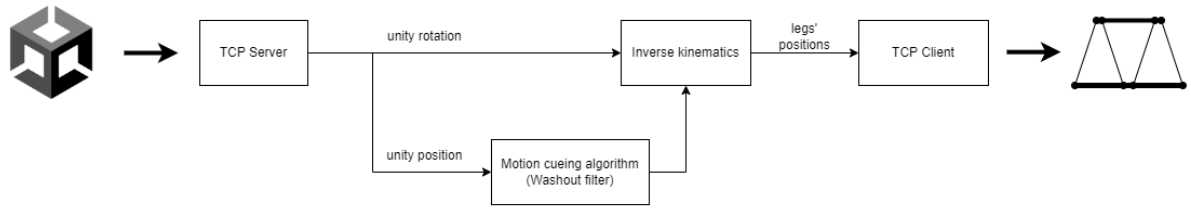


Figure 4.17 Simplified platform control software's logic

A Stewart platform cannot fully emulate real movement from a vehicle, as the platform can only move in limited space, attached to its base, whereas the vehicle can move freely in a virtually infinite physical space, without being attached to a fixed base. For this reason, the platform needs to adapt to this restriction, which is the functionality of a motion cueing algorithm, transforming the motion of a vehicle into motion in the simulator. There are a few algorithms to choose from, but the one implemented in the control software was a washout filter.

The washout filter's aim is to remove irrelevant signals and to bring the Stewart platform back to its neutral position. This movement is supposed to be performed under the threshold of human perception so that the user does not feel these movements and the platform can apply new accelerations [77]. Having the platform return to its original position allows greater accelerations to be applied to the platform, minimizing the physical limitations of the platform's movement. To better understand the washout filter's effects, an example will be analyzed below.

The example used consists of a simple scenario where the vehicle starts from a stopped position and starts accelerating to its maximum capacity, maintaining the accelerator input at its maximum for the remainder of the test.

Figure 4.18 displays the acceleration values sent from Unity to the Stewart platform control software and Figure 4.19 displays the values after they were processed by the washout filter used in the Stewart platform control software.

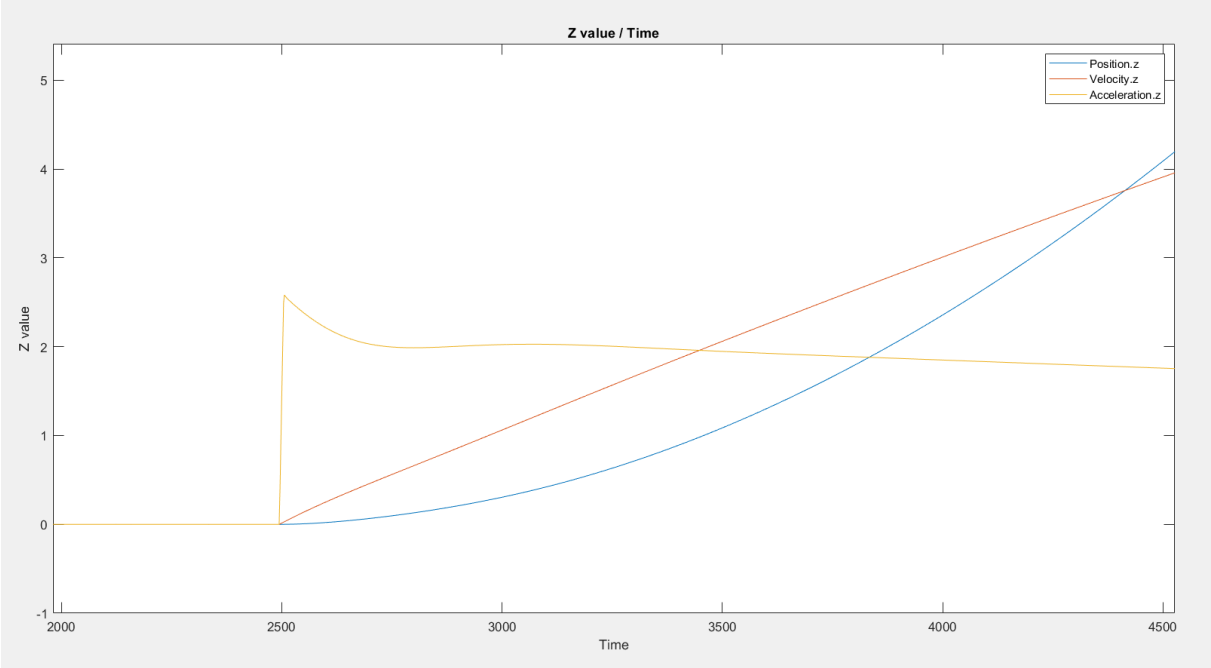
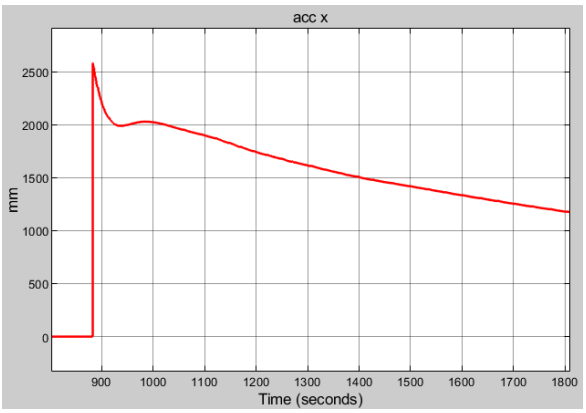
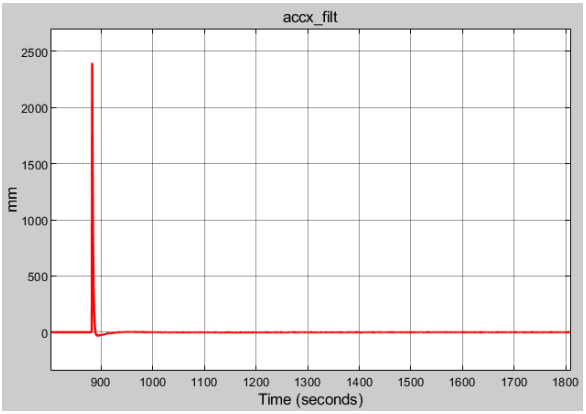


Figure 4.18 Acceleration values sent from Unity

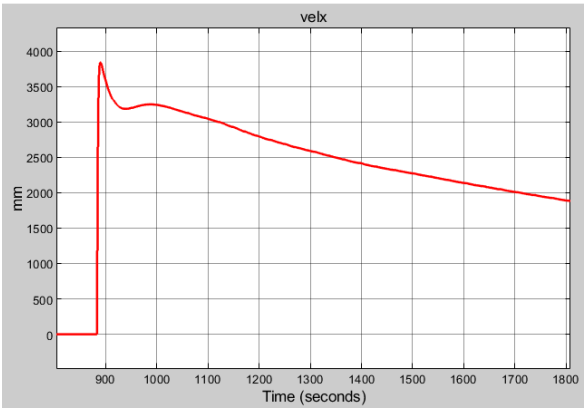
Figure 4.19 depicts six different graphs, which represent different steps of the Stewart platform control software. The graph in 4.19a corresponds to the acceleration values received from the Driving Simulator Software and the graph in 4.19b corresponds to these values processed through a high pass filter. The graph in 4.19c displays the velocity calculated from the acceleration values and the graph in 4.19d displays theses values processed through a high pass filter. The graph in 4.19e represents the position calculated from the previous velocity values and the graph in 4.19f represents the final position values used in the platform's movement. Of these graphs, the most important are the ones depicted in 4.19a and in 4.19f, as they consist of the input and output, respectively, of the subsystem of the Stewart platform control software which uses the washout filter.



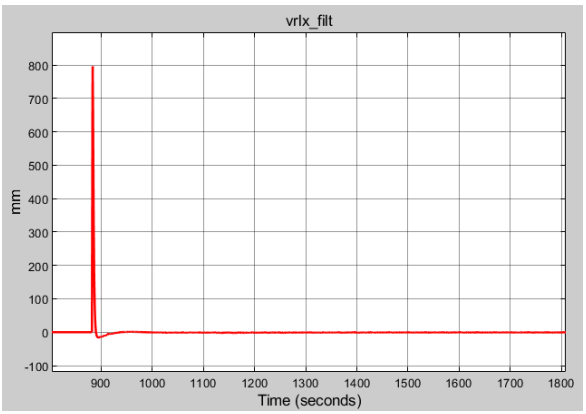
(a) Acceleration



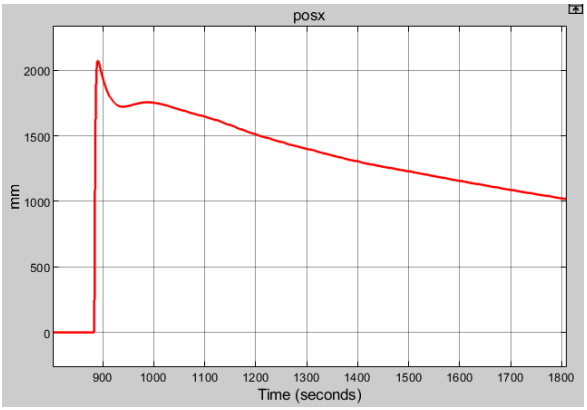
(b) Filtered acceleration



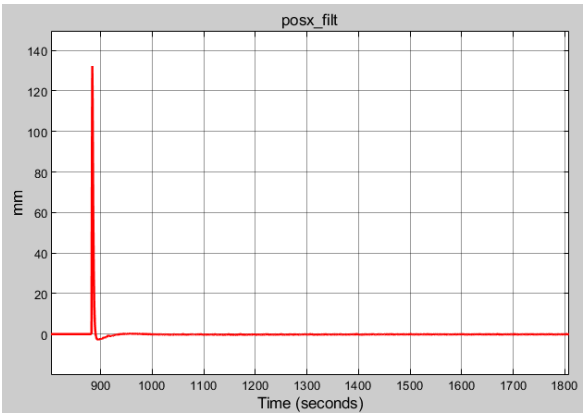
(c) Velocity



(d) Filtered velocity



(e) Position



(f) Filtered position

Figure 4.19 Processed values using the washout filter

Figure 4.18 also shows the position values of the vehicle in the virtual environment in Unity, which can be compared with *posx_filt*. The first displays the virtual position, while the latter shows the physical position of the platform. However, they do not match at all, due to the use of the washout filter. As the vehicle first accelerates, the platform moves forward, and when the acceleration is maintained, the platform moves back to its original position, which can be seen in *posx_filt*. This way, if the vehicle were to suffer an even bigger acceleration in the same direction, the platform would have enough space to perform another movement right away. Without the use of the washout filter, the platform would still be extended, in this case, forward around 120 mm. However, when the new higher acceleration value was to be applied, the newly calculated platform position would exceed the physical capacities of the platform, it wouldn't be able to move and therefore the user would not feel the acceleration.

Inverse kinematics is the process of calculating the platform's joint's angles from known coordinates [78], in this case, the coordinates of the vehicle filtered by the washout filter. This means that, from the vehicle's position, the inverse kinematics module calculates the position of each of the platform's legs in order to move the platform to the intended position.

The inverse kinematics module receives the output of the washout filter, the processed acceleration values, the vehicle's rotation values and some variables regarding the platform's dimensions, such as the base and top plate's side's length and the distance between the plates and the joints of the legs [7]. To increase the flexibility of this module, these variables could be defined and stored in an external file, that would be loaded when running the application. This way, a different-sized platform with similar behaviour could be more easily configured.

To send the final computed values to the platform's actuator position control software, the values need to be incorporated into that software's communication protocol. The actuator position control software uses ModBus TCP [79], a protocol typically used in communication between automation devices. The control software sends an application data unit (ADU), which is composed by the ModBus Application Protocol Header (MBAP Header) and a Protocol Data Unit (PDU) [7][79]. The format of the ADU is detailed in Table 4.3.

Table 4.3 ModBus TCP/IP ADU format

ModBus TCP/IP ADU					
MBAP Header				PDU	
Transaction ID	Protocol ID	Length	ID Slave	Function	Data
2 bytes	2 bytes	2 bytes	1 byte	1 byte	N bytes

The actuator position control software supports several different functions, however only one is being used in the control software. When the Function is set to the value 16, ModBus is configured to 16 bits values [79]. In Tables 4.4 and 4.5 the generic format of the messages sent to the actuator position control software is detailed.

Table 4.4 Generic format of the MBAP Header

MBAP Header						
Transaction ID		Protocol ID		Length		ID Slave
High	Low	High	Low	High	Low	-
0	2	0	0	0	19	1

Table 4.5 Generic format of the PDU

PDU																	
Function	Data																
16	7	208	0	6	12	H1	L1	H2	L2	H3	L3	H4	L4	H5	L5	H6	L6

In the Simulink model, the six values to be sent have a size of 2 bytes each, totalling 12 bytes. When encapsulated in the ADU, the data to be sent by the TCP server requires 25 bytes. The process of encapsulating this data in the Simulink model is depicted in Figure 4.20.

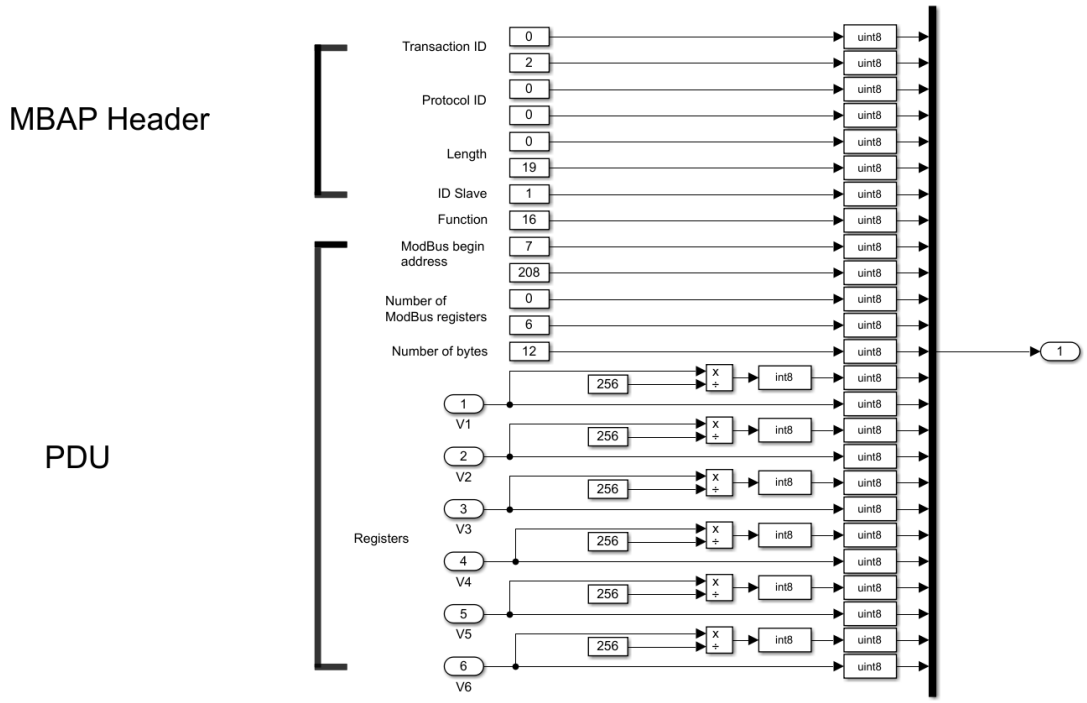


Figure 4.20 ModBus communication block in Simulink

In Figure 4.21, there is a real example of the data that is received by the actuator position control software, when only zeros are sent by the Driving Simulator Software. The first 13 values are always the same and the last 12 values correspond to the height of each one of the platform's legs, which depends on the data sent. In Figure 4.22, a value of 2 is sent, corresponding to an acceleration of 2 in the z axis, and the last 12 values change accordingly.

```
Bytes received: 24  | Input: 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000
Output: 0 2 0 0 0 19 1 16 7 208 0 6 12 5 95 5 95 5 95 5 95 5 95 5 95
```

Figure 4.21 ADU bytes' values when sending zeros

```
Bytes received: 24  | Input: 0.000000, 0.000000, 2.000000, 0.000000, 0.000000, 0.000000
Output: 0 2 0 0 0 19 1 16 7 208 0 6 12 4 4 4 4 6 234 5 134 5 134 6 234
```

Figure 4.22 ADU bytes' values when sending a single acceleration value

The washout filter was the chosen method to minimize the physical limitations of the platform, however, other methods can be implemented. In [80] and [81], the authors approach the utilization of backward pitches of the platform in combination with visual acceleration cues. This way, the platform can simulate larger trajectories, with sustained forward accelerations, using smaller movements. The objective is to slightly tilt the platform and the driver with a rotation below the vestibular semicircular canal threshold, to prevent the driver from detecting the rotation [81]. The semicircular canals are three organs located in the inner ear responsible for balance and stability, as they sense the rotation and orientation of the head [82]. However, both articles conclude that this tilt of the platform is not enough to guarantee a realistic perception of motion in all situations [81], referring to the need for visual cues. As the use of these cues may not always be possible in this project, this approach has not been considered herein.

Chapter 5

Project configuration

As this project involves several different software components and some hardware devices, the process to make the project run requires some steps. The required steps for each of the components to work will be described, herein. All of the software was maintained in a GitHub repository [83], which was continuously updated as the project progressed.

It is also important to list all the tools and software that were required throughout the development of the project, as well as their version. Some of the tools that have not been previously mentioned are accompanied by a brief description.

- Unity Hub v3.5.1 - Unity application that manages Unity projects and installations;
- Unity v2022.3.10f1;
- MATLAB R2021b;
- MinGW-w64 v12.2.0.03042023 - a tool that provides support for the GCC compiler on Windows systems;
- Oculus;
- Meta Quest Developer Hub;

The VR headset, the steering wheel, pedals and even the Stewart platform in some cases may not be needed for the user's usage and therefore do not always need to be connected for the correct functioning of the project.

5.1 Stewart platform

The Stewart platform needs to be running and connected to the network. To correctly turn on the platform, consult the platform's instructions [7].

The majority of the software and files relative to the Stewart platform Software are located in the MATLAB folder of the repository [84]. This includes *i)* MATLAB Simulink model; *ii)* folder with generated c code and associated files; *iii)* generated C *DLL*; and *iv)* folder with C application which uses the *DLL* to interact with the platform. The only files relative to this module of the project are the file that contains the necessary parameters to connect the control software to the platform and the batch files that handle created files, facilitate the use of these components and make the execution of the project easier, that are present in the main folder of the repository: *platform_tcp_parameters.txt*; *generate_dll.bat*; *run_platform_interface.bat*.

To run the Stewart platform control software the following steps should be followed:

1. The Stewart platform needs to be powered on and its TCP server available on the desktop's network;
2. The configuration file - *platform_tcp_parameters.txt* - should be changed if needed;
3. The batch file - *generate_dll.bat* - should be executed if any changes were performed to the model: (*>generate_dll.bat*).
4. The batch file responsible for the execution of the platform should then be executed: (*>run_platform_interface.bat*). The TCP server is now working and ready to receive data. This script also recompiles the code if any changes are made.

5.2 Steering wheel and foot pedals

The specific model used in this project is from Fanatec [85], however, most models should be similar in terms of assembly and connection. The pedals and gear stick connect themselves to the steering wheel controller, which is attached to the steering wheel. The controller is then connected to power and to the desktop via USB. Unity will automatically detect and use the input device.

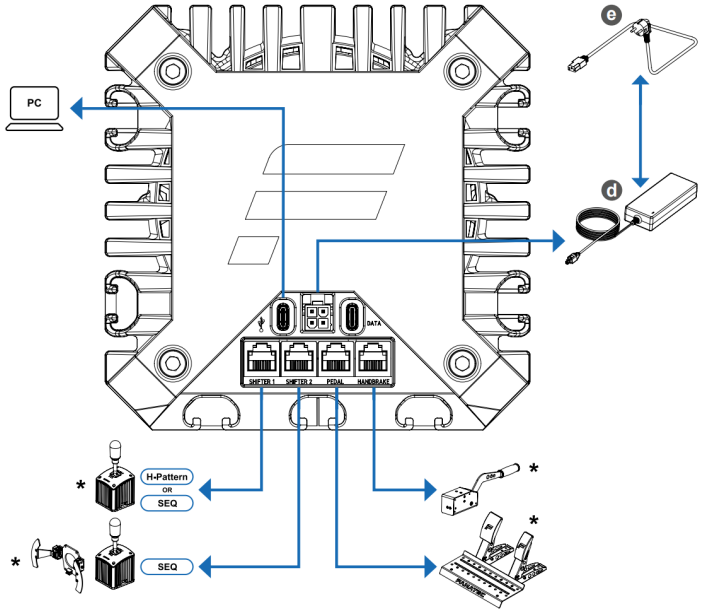


Figure 5.1 Fanatec CSL DD cable connections [5]

5.3 VR headset

As previously mentioned, the specific headset used in the project is the Oculus Quest 2, which requires the Oculus desktop app.

The headset needs to be connected to the desktop via USB and turned on. After turning it on, some configurations may be needed according to the headset's prompts, such as defining a safe area to use the headset.

If correctly connected, when the Unity application starts executing, the user should see the virtual environment and their hands represented.

5.4 Driving Simulator Software

The source code for the driving simulator software is available on the Unity folder of the repository [86]. This includes all the scripts developed for the project, assets used and external packages installed. As some files are not supposed to be uploaded to the repository, when the repository is cloned and Unity is opened, it may take some time to fully open the project. With the project fully opened, the executable file should then be built and executed. The configurations needed for the correct build process are saved with the project, so no further configuration is required.

As it can be very time-consuming, to simplify the process of execution of the project, the executable file is also available on the GitHub repository. This file can executed using the batch file (>*run_simulator.bat*)

If everything was performed correctly, the Unity project should start and show the UI. The user may choose the preferred configurations and begin testing the simulator.

Chapter 6

Experimental Results

Developing a realistic driving simulator is an extremely complex task, involving the simulation of numerous factors, from the behaviour of each individual wheel to the various effects that different types of terrain can have. For this reason, it wouldn't be possible to implement a more realistic experience herein.

Unity provides a service, the Unity Asset Store [87][88], which allows programmers to develop custom assets, either free or paid, that are moderated by Unity, and can range from simple 3D objects to collections of scripts. In some cases, these assets can be very complex, implementing functionalities such as the player's behaviour and vehicle's movement. One of the most popular assets in this field is *Edy's Vehicle Physics* [89], which is available in both a free version and a paid version. This asset offers a realistic simulation of a vehicle's movement, even implementing a gear system, which was not considered in the implementation herein developed. This package could be analyzed in a future version of the project. However, it doesn't offer as much control of the code as a custom implementation and it may be too complex in a phase where it is more important to test more basic features and the interaction with the Stewart platform.

6.1 Analysis of Unity's vehicle tutorial

Before analysing the results produced from the test scenarios, it is also important to analyze some problems that emerged throughout the development of the project and how they were solved or minimized. Some of these problems are linked to Unity's implementation of some of the components, such as the *WheelColliders* themselves and as such were hard to suppress.

Unity provides a step-by-step tutorial on creating an implementation of a vehicle controller [90], which served as the basis of the code developed in this project. The tutorial also details how to configure the essential components used, such as the four *WheelColliders* and the *Rigidbody*. Some of the relevant variables and the corresponding values are shown in Table

6.1, the remaining relevant variables that were not mentioned in the tutorial were left with their respective default values. The *Fixed Timestep* property from the Project Settings was set to its default value, 0.02 - 20 milliseconds. It is also important to highlight that the input from the user is processed in a different way from the implemented solution. The Unity tutorial uses the older Input Manager, whereas the implemented solution uses the newer Input System Package. This may cause some different results, however, this detail will not be discussed in this section.

Table 6.1 Relevant variables and corresponding values used in the Unity tutorial

Component	**Attribute**	**Value**
Rigidbody	Mass	1500
WheelCollider	Suspension Distance	0.45
	Radius	0.44
Car Control script	*motorTorque*	2000
	brakeTorque	2000

Even though Unity provides a complete tutorial, even when following all the steps some problems persist, as will be described in this section. The graphs detailing the acceleration values were obtained by creating a separate *Scene* in the Unity project where the tutorial was followed and by using a modified version of the *Logger* script used in the project. The results shown in Figure 6.3 were obtained when running the Unity tutorial Scene with the test scenario, referred to from now on as Test 1, where the vehicle accelerates forward and comes to a full stop only due to the vehicle's drag. A diagram depicting the vehicle's trajectory during Test 1 is in Figure 6.1. The input values are represented in Figure 6.2, to better understand the test scenario.

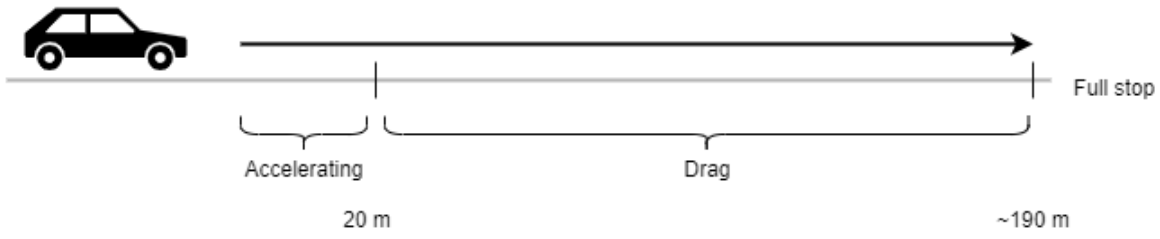


Figure 6.1 Vehicle's trajectory during Test 1

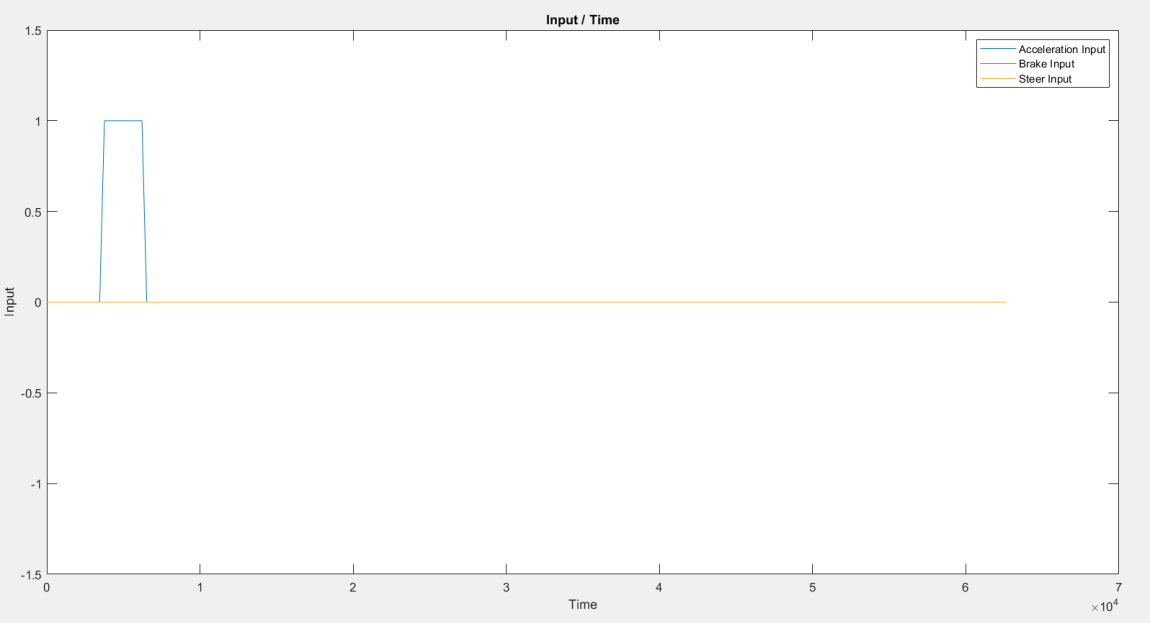


Figure 6.2 Input values from Test 1

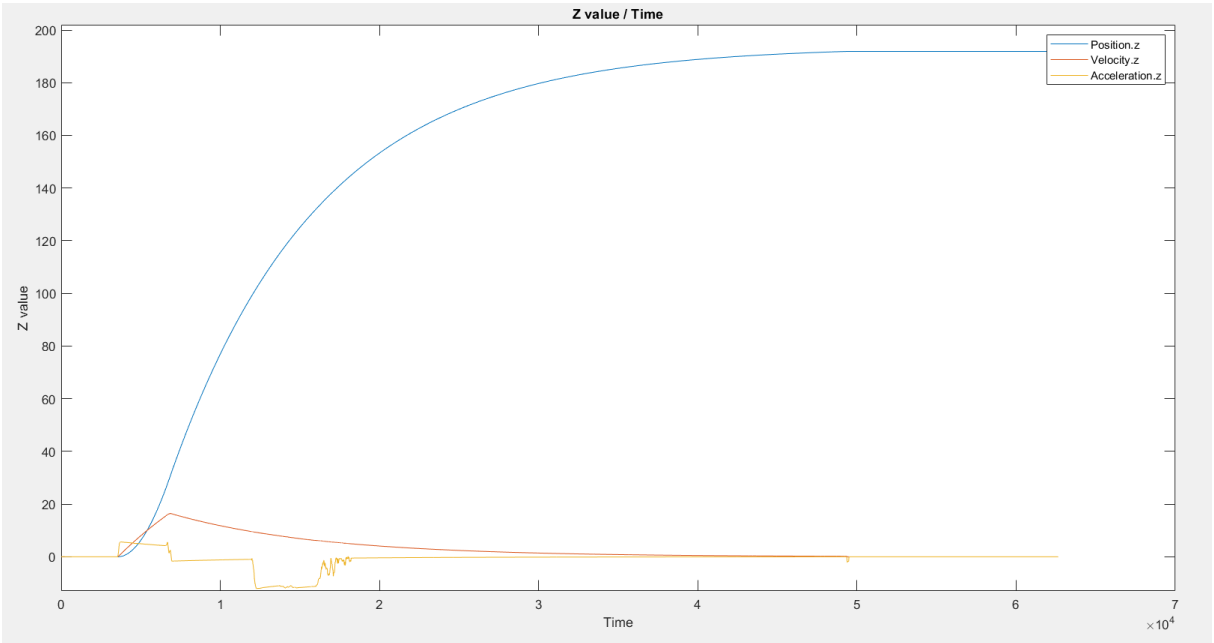


Figure 6.3 Position, velocity and acceleration values from Test 1

After analysing Figure 6.3, which represents the vehicle's acceleration on the Z axis (forward), some problems are easily identifiable. Firstly, in the decelerating phase of the test, which can be seen more clearly in Figure 6.4, there are some unexpected oscillations in the velocity values, which in turn cause some significant fluctuations in the acceleration values. Secondly, when the vehicle is coming to a full stop, having near zero acceleration, there is a negative spike in acceleration, shown in Figure 6.5, which does not make sense in a real-life scenario.

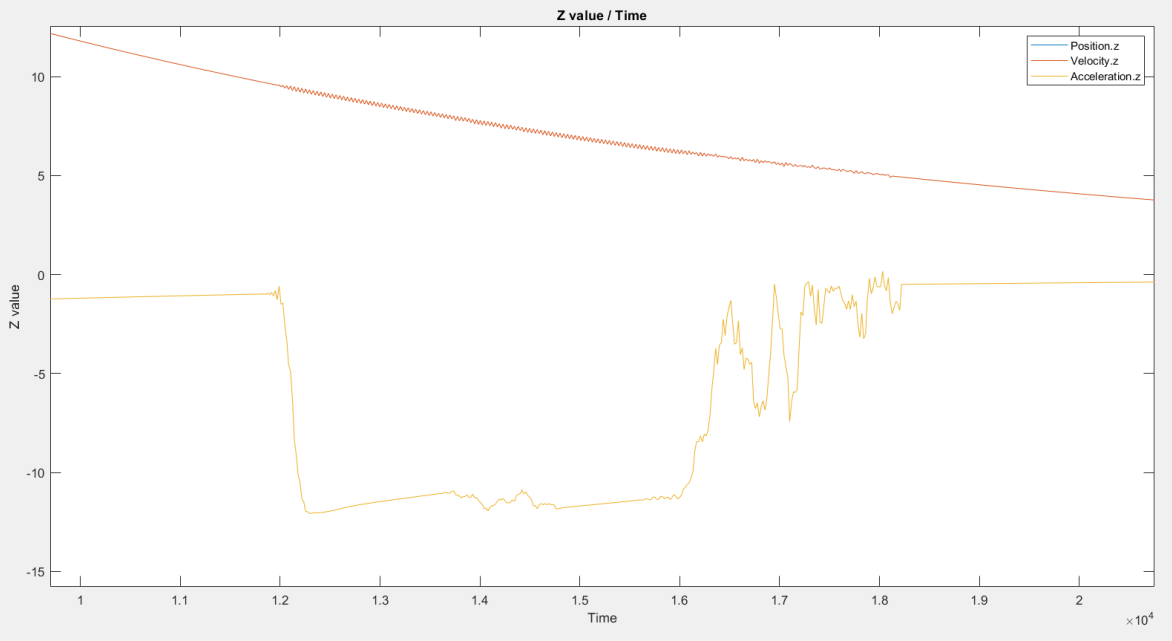


Figure 6.4 Velocity and acceleration values from Test 1 - 10 to 20 seconds

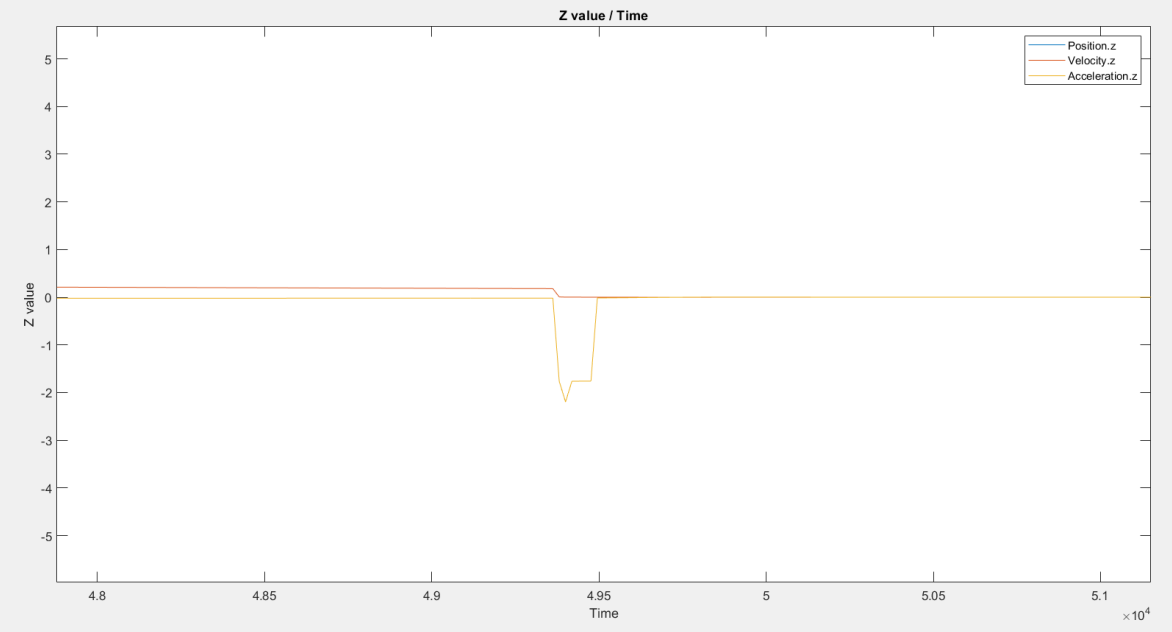


Figure 6.5 Velocity and acceleration values from Test 1 - 48 to 51 seconds

The analysis of the Unity tutorial and its problems serves to show that some of the issues encountered in the implemented solution stem from Unity itself and not from the proposed solution. These problems are something that needs to be tackled and not something that can be easily avoidable. In the following sections, these problems and their respective solutions are going to be detailed in the context of the main test scenario of the implemented solution, as the effects of the proposed solutions are more clear.

Another problem that was detected during the development of this project was related to the interaction between objects and their colliders. For example, when testing a scenario involving speed bumps, it was detected that the vehicle's wheels might not always be represented on top of the speed bump even if they should be. Figures 6.6 and 6.7 illustrate this problem. The vehicle is moving forward and when going past a speed bump, represented in Figure 6.6 by the orange outline and in Figure 6.7 by the white object, instead of the vehicle and the wheel moving up accordingly, the wheel first goes through the speed bump and only later suffers its effect. This causes the acceleration values to change abruptly, which in turn can cause abrupt movements in the Stewart platform, as the wheel suddenly changes from overlapping the speed bump to going to its top in the difference of a frame. The smaller the time between frames, the bigger the spikes in acceleration, as it is calculated by dividing the difference in velocity by the time interval between frames.

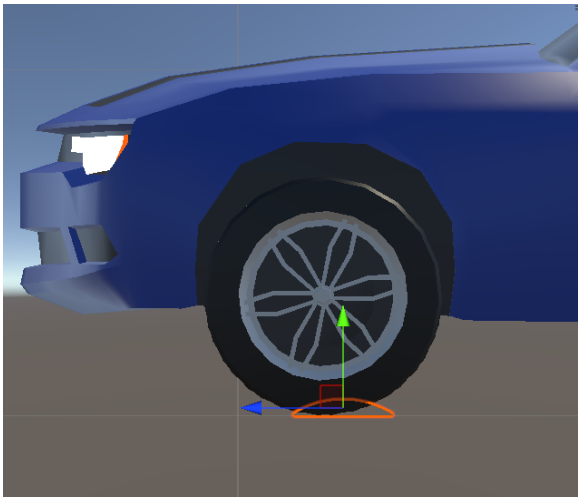


Figure 6.6 Sideways view of the wheel and the speed bump overlapping

Figure 6.7 The wheel and the speed bump overlapping

6.2 Timestep variation

The first problem to be tackled was the oscillations in the data. It is important to note that Unity is a tool mostly used for game development and not as much to try to emulate realistic behaviours, such as vehicle movement. For this reason, in some applications, even though the visual representation of a moving object appears to be correct and without problems when analyzing its data, such as position or velocity values, properties that are in constant change, there might be some minor fluctuations. In most cases, this does not pose a problem, however, in this project, a Stewart platform is processing these values in real-time, which means every acceleration value is going to affect the platform's position and is going to be felt by the user on top of the platform. Every oscillation that isn't coherent with the vehicle's movement will alter the user experience, so it must be minimized.

The oscillations are likely caused by the use of *GameObjects* such as the *Rigidbody*, which is related to Unity's physics system and which influences Unity's handling of the precise values of the object's position in the 3D space. The *FixedUpdate* method is directly linked to Unity's physics system and is executed every 20 ms by default. This value is called *Fixed Timestep* and dictates how frequently the physics calculations are performed. It can be configured in the *Time* settings from the *Project Settings*. As this value decreases, the physics calculations are performed more frequently and consequently, the position and velocity values are better handled. It is also important to note that even though *Fixed Timestep* is set to a specific value, Unity will not always be able to run *FixedUpdate* at that particular frequency. The frequency may be a few milliseconds more or less than the specified value, however, whatever the achieved value is, it will be constant throughout the execution of the application.

Some tests were performed to determine an appropriate value for this attribute where a balance exists between performance and the intensity of the oscillations. A test scenario similar to the one tested with the Unity tutorial was used with the implemented solution. Figures 6.8, 6.9, 6.10, 6.11, 6.12 and 6.13 show the results of these tests with the *Fixed Timestep* set to 20 ms, 15 ms, 10 ms, 5 ms, 2 ms and 1 ms, respectively. These tests use predefined values for the user to input, in order to be independent of possible variations.

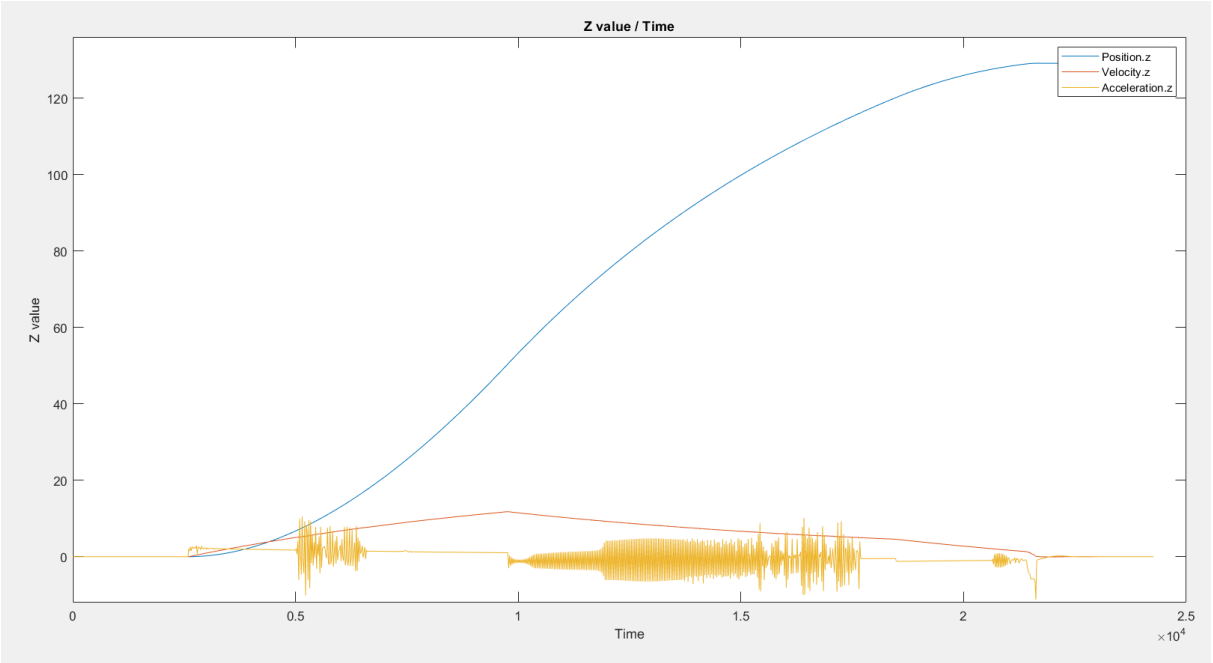


Figure 6.8 *Fixed Timestep* set to 20 ms

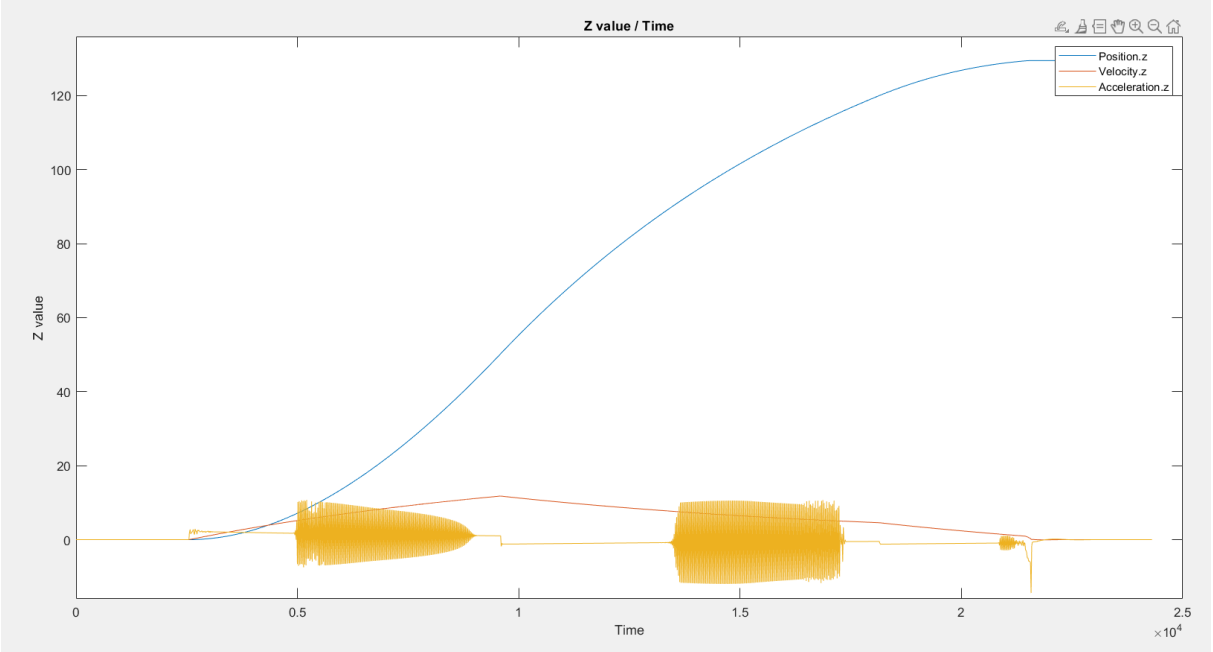


Figure 6.9 *Fixed Timestep* set to 15 ms

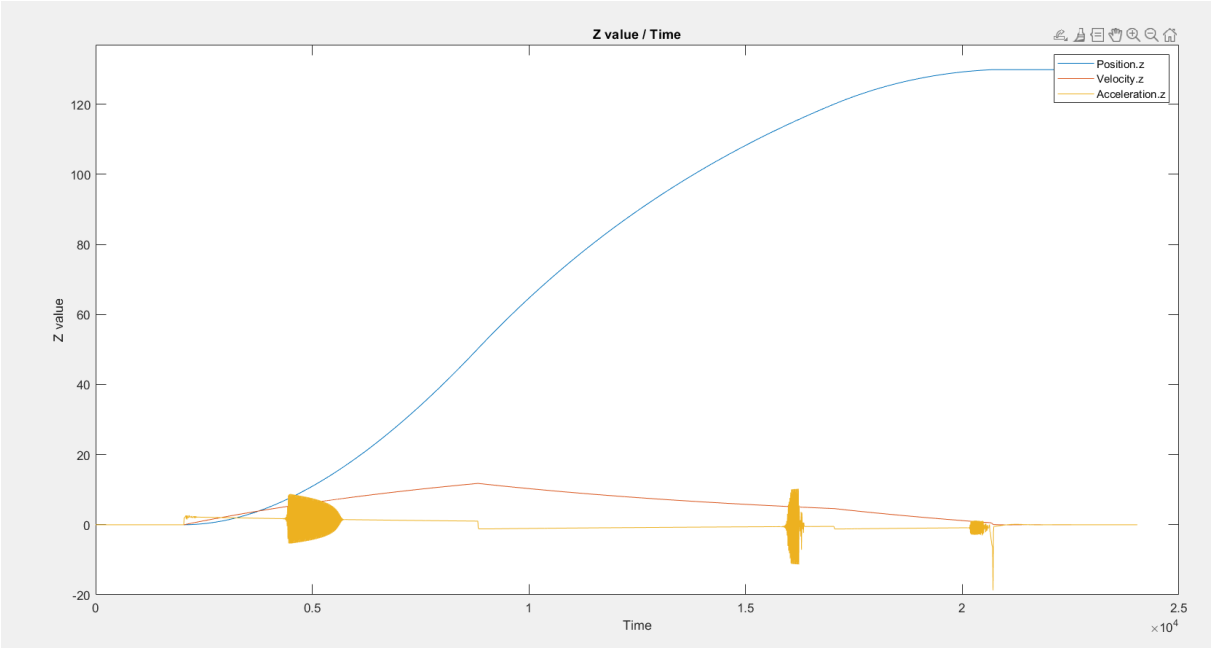


Figure 6.10 *Fixed Timestep* set to 10 ms

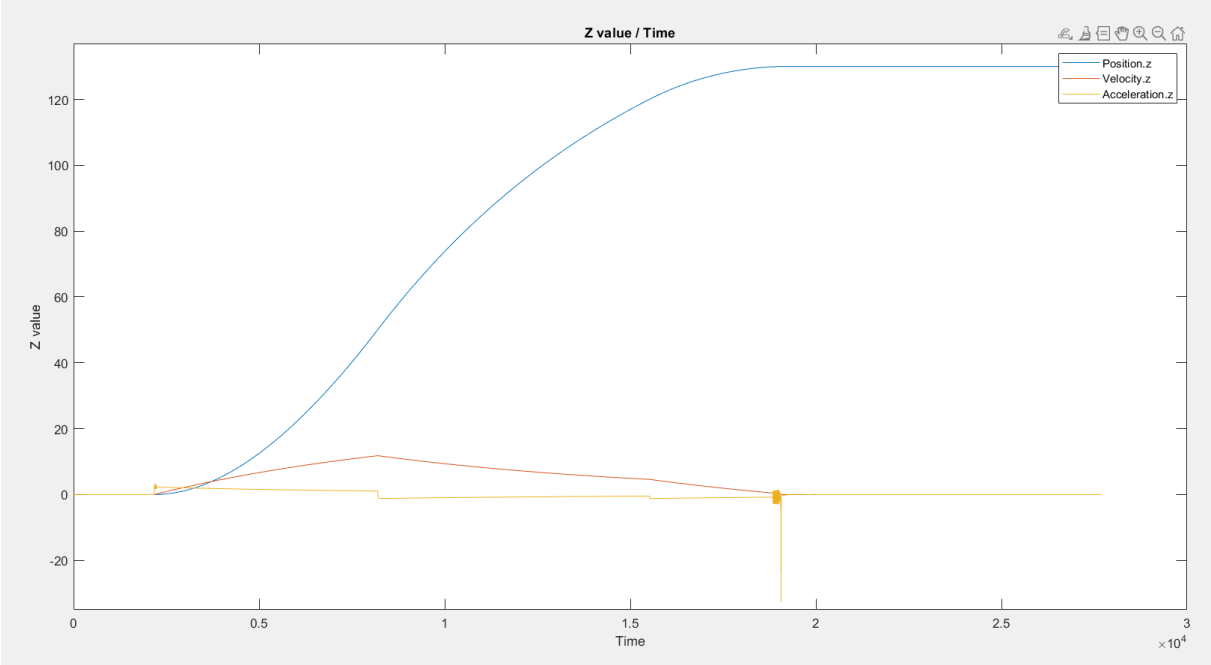


Figure 6.11 *Fixed Timestep* set to 5 ms

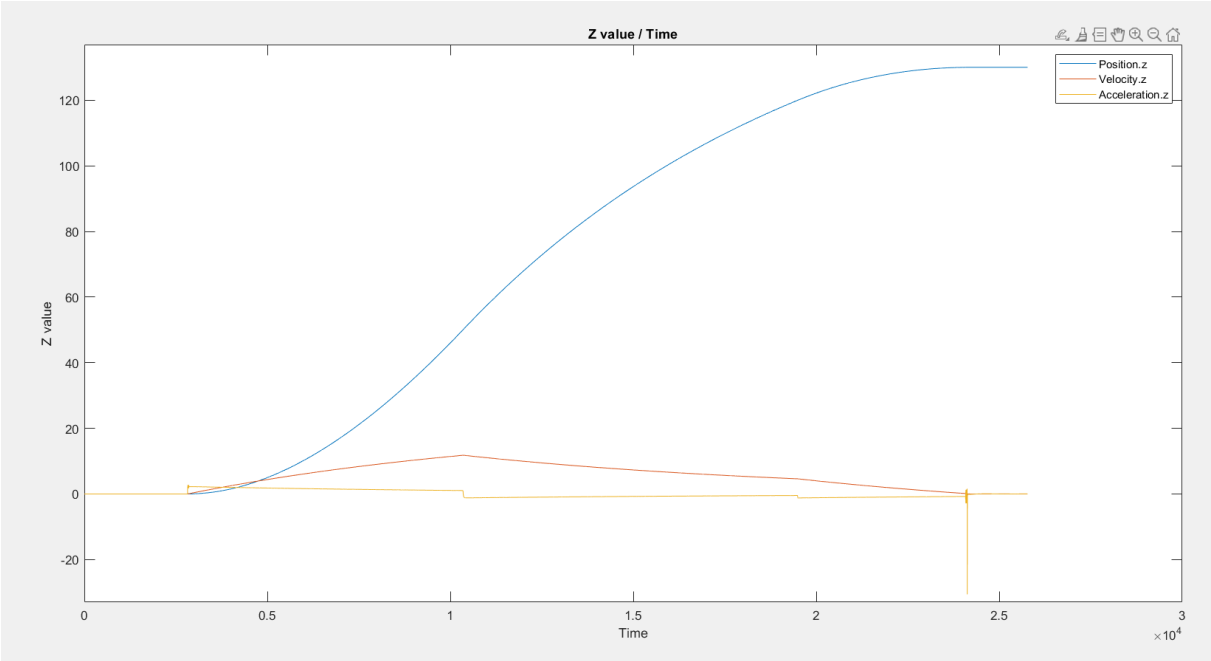


Figure 6.12 *Fixed Timestep* set to 2 ms

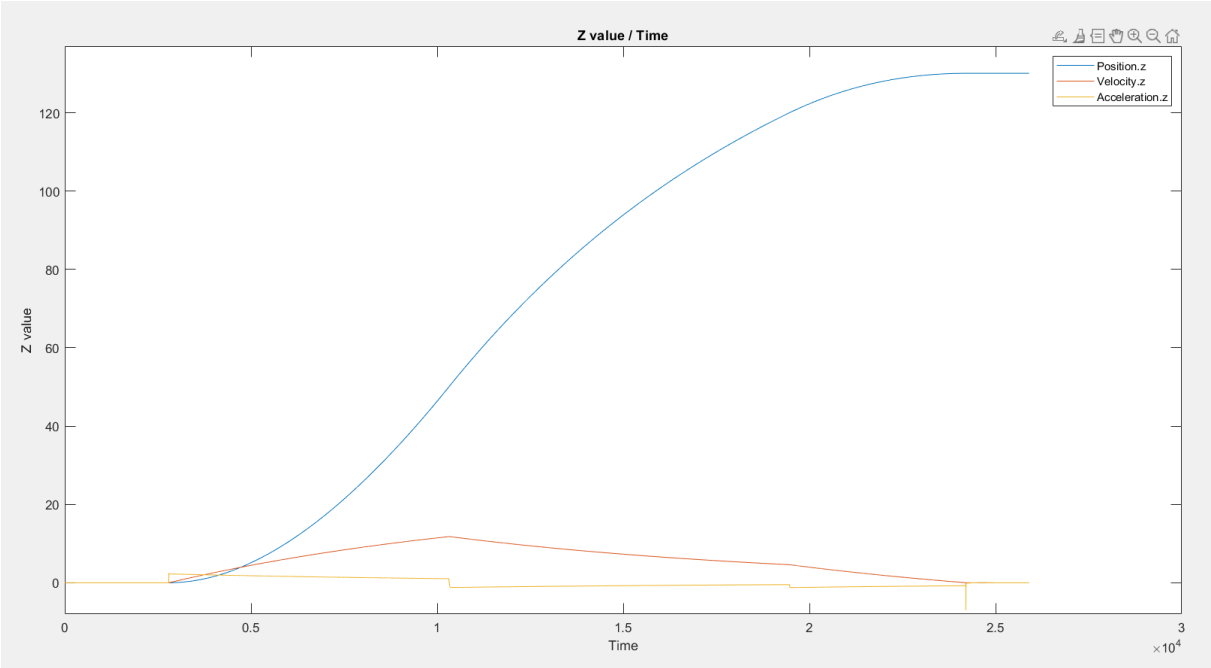


Figure 6.13 *Fixed Timestep* set to 1 ms

After analyzing the results from the tests, it can be concluded that reducing the *Fixed Timestep* attribute drastically lowers the intensity of the oscillations present in the vehicle's acceleration values, which in turn reduces the unwanted movements of the platform. The chosen value for the *Fixed Timestep* attribute was 1 ms as it greatly reduces the noise in the acceleration values without harming the program's performance. Using anything lower than 1 ms will not bring any significant improvement and will require higher computation requirements.

6.3 Averaging acceleration values

To avoid spikes in acceleration and deceleration and to minimize any remaining oscillations in the acceleration values, some filtering is performed on the acceleration values in real-time.

A moving average system was implemented, where instead of directly sending the acceleration values to the platform's control software, the most recent values are averaged with some of the previous ones. These calculations are performed in the *Logger* script, which is configured to use two variables that define how they are performed. The *maxWindowSize* variable defines how many values the window, which is the set of values to be averaged, can use. The *maxIncreaseAcceleration* variable sets the maximum value of the acceleration, if the current acceleration is bigger than this variable, the value is completely discarded and not used in the moving average calculations. The same logic is used for the negative acceleration values, comparing *maxIncreaseAcceleration* with the absolute value of the acceleration. These values can easily be changed through the Unity Editor, however, *maxWindowSize* is set to 5 and *maxIncreaseAcceleration* is set to 7 by default. The *Logger* script also uses the *averageAcceleration* variable to determine if the acceleration values are to be averaged or not, which can also be changed in the Unity Editor.

Figures 6.14 and 6.15 show the effect of the moving average. The former shows the acceleration values directly calculated from the vehicle and the latter shows the acceleration values averaged in a window with a maximum of five values. The values are obtained from the same test scenario, where the vehicle accelerates and comes to a full stop without directly breaking. As can be seen, the negative spike around the 2.3 x 10^4 ms mark on the Time axis, completely disappears when using the proposed moving average.

In this Chapter, the problems encountered during the development of the Driving Simulator Software were analyzed and the proposed solutions were detailed. Chapter 7 details the interaction between the Driving Simulator Software with the Stewart platform, analyzing the results produced by different test scenarios.

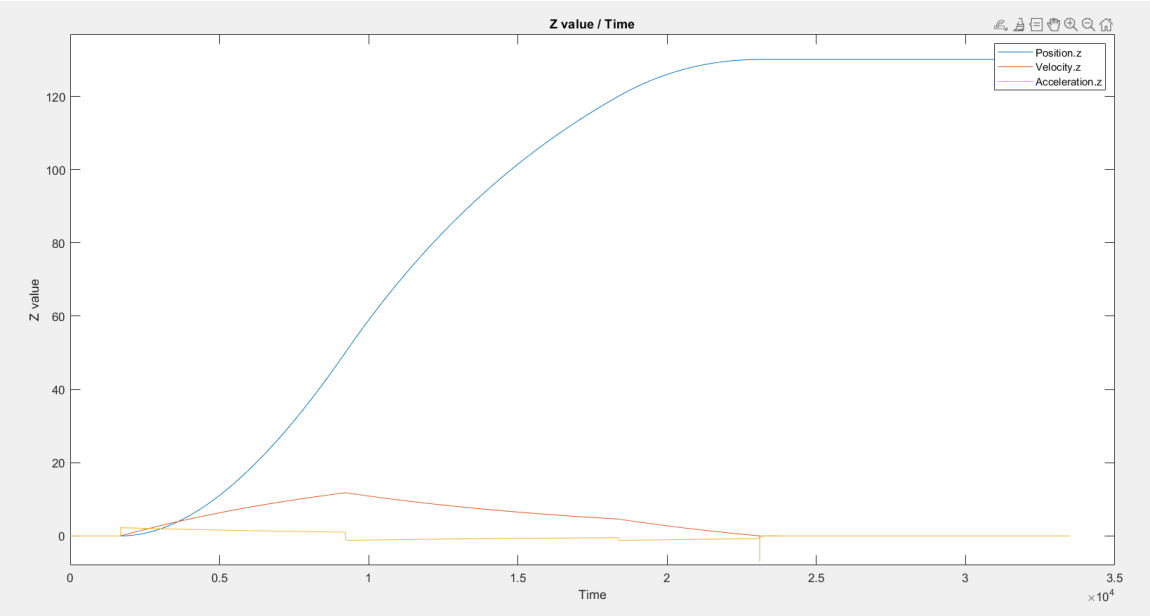


Figure 6.14 Not averaging acceleration values

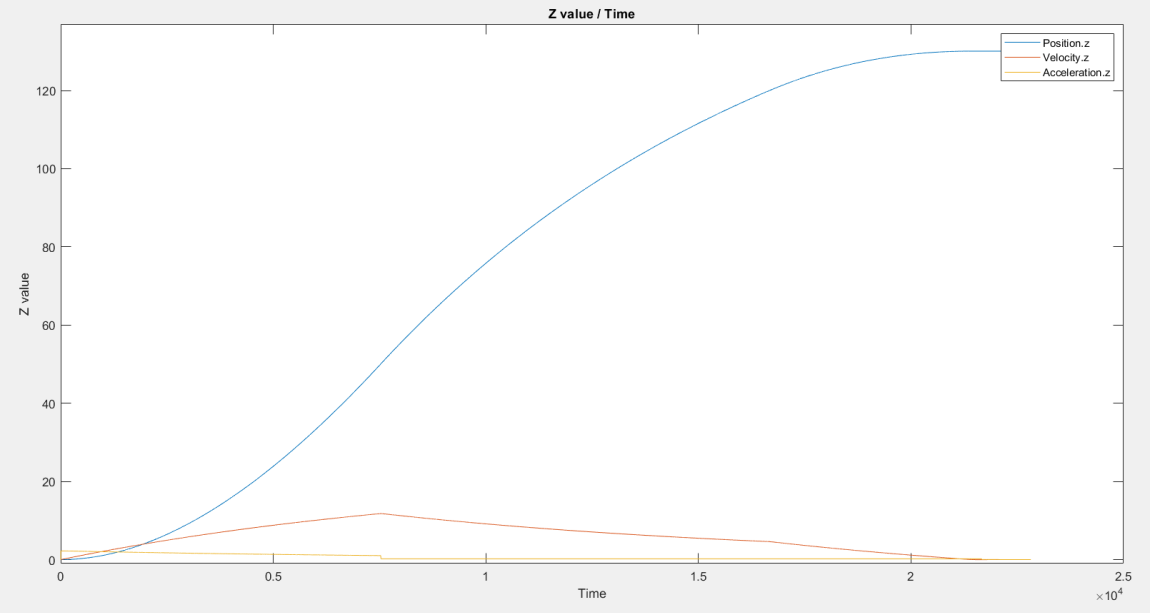


Figure 6.15 Averaging acceleration values

Chapter 7

Project results

In order to analyze the results of the interaction between the driving simulator software and the Stewart Platform, it is important to have some control scenarios, that are independent of user input and only change according to predefined parameters. With this in mind, several test scenarios were arranged, which cover regular day-to-day components of a driving session, while also aiming to test every axis of the platform. The position, velocity, and acceleration data can be analyzed in different components of the project, such as the driving simulator, the platform's control software, and finally, the platform itself, through the platform's accelerometer. The data can then be displayed on graphs to visualize the modifications done to the data. In this section, each test scenario will be described and the data produced shown and analyzed. The data sent represents the acceleration and rotation values in an ideal environment.

Before analyzing the data, it is important to establish the conditions of the test environment. The driving simulator and the control software are both running on the same computer, which has the specifications detailed in Table 7.1. The test scenarios, with the exception of the free roam mode, are tested without any additional weight on top of the platform.

Table 7.1 Dedicated desktop specifications

Specification	Details
Operating System	Windows 11
RAM	16 GB
Processor	Intel Core i5-10400F
Graphics Card	AMD Radeon RX 6600

In order to more easily evaluate the relationship between the Driving Simulator Software and the movement of the Stewart platform, an additional program was developed. The program, developed in C#, is able to send specific values through a TCP connection, during specific time intervals, allowing a more controlled environment with less varying values. This way, the data can more easily be compared to the platform's accelerometer's data, without variations introduced by the Driving Simulator Software, such as friction, the vehicle's suspension and other variables.

7.1 Testing scenarios

In this section, the different scenarios created to test the developed application, from the Driving Simulator Software to the Stewart platform control software and the physical behaviour of the Stewart platform will be analyzed. For each scenario, there will be a representation of the vehicle's movement implemented in the Unity software, a graph displaying the acceleration values sent from the Driving Simulator Software, and also a graph displaying the processed values in the Stewart platform control software, which represent the values sent to the platform, in order to understand the importance of some of the components and solutions implemented throughout the project, such as the washout filter and the use of an average of the received acceleration values.

To create a complete analysis of the project, several test scenarios were created, each scenario with a specific axis of the platform in mind. Before analyzing each scenario it is important to recall that Unity uses a specific coordinate system, the positive x-axis points to the right, the positive y-axis points up, and the positive z-axis points forward. The main scenarios created were *Forward*, to test the z-axis, *Curve*, to test the x-axis and *Speed Bump* to test the y-axis. The results from the following tests were obtained with the *Fixed Timestep* set to 1 ms. It is also important to note that MATLAB Simulink has the setting of *Fixed-step size* set to 0.02. This setting defines the size of the fixed step the solver takes during simulation [91]. This means that MATLAB Simulink does not run a real-time simulation. Consequently, in this Chapter, the graphs taken from this tool, related to the washout filter, have the *Time* axis in a different scale from real-time.

7.1.1 Forward

In the *Forward* test scenario, the vehicle takes the path depicted in Figure 7.1. It starts with the vehicle in a stopped position, the vehicle then starts accelerating for 12,5 m. After these 12,5 m, the vehicle stops accelerating, but it does not brake either. When the vehicle moves 45 m from the starting point, it starts braking until it comes to a full stop. The vehicle completely stops 55 m from the starting point. Figure 7.2 shows these changes in the accelerator and brake input.

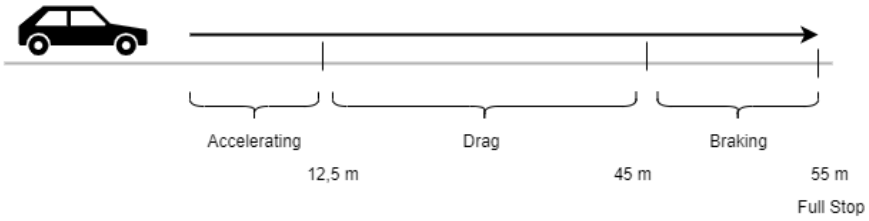


Figure 7.1 *Forward* Test - vehicle path

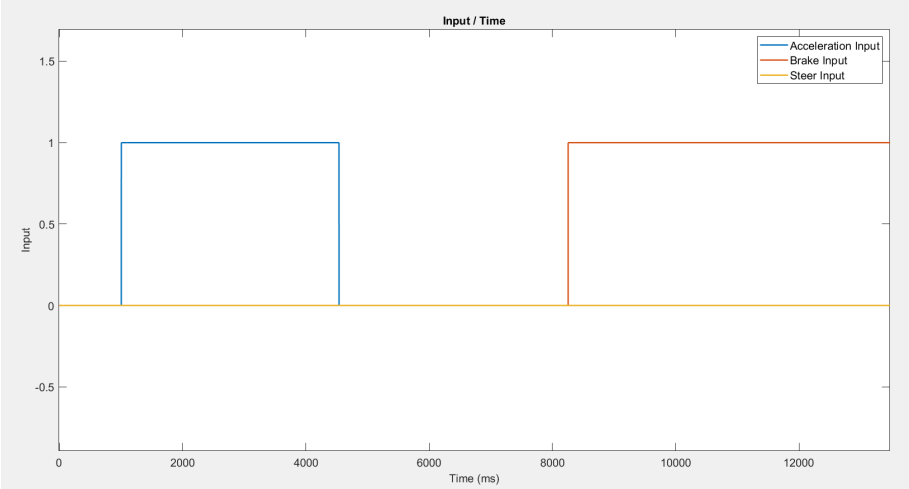
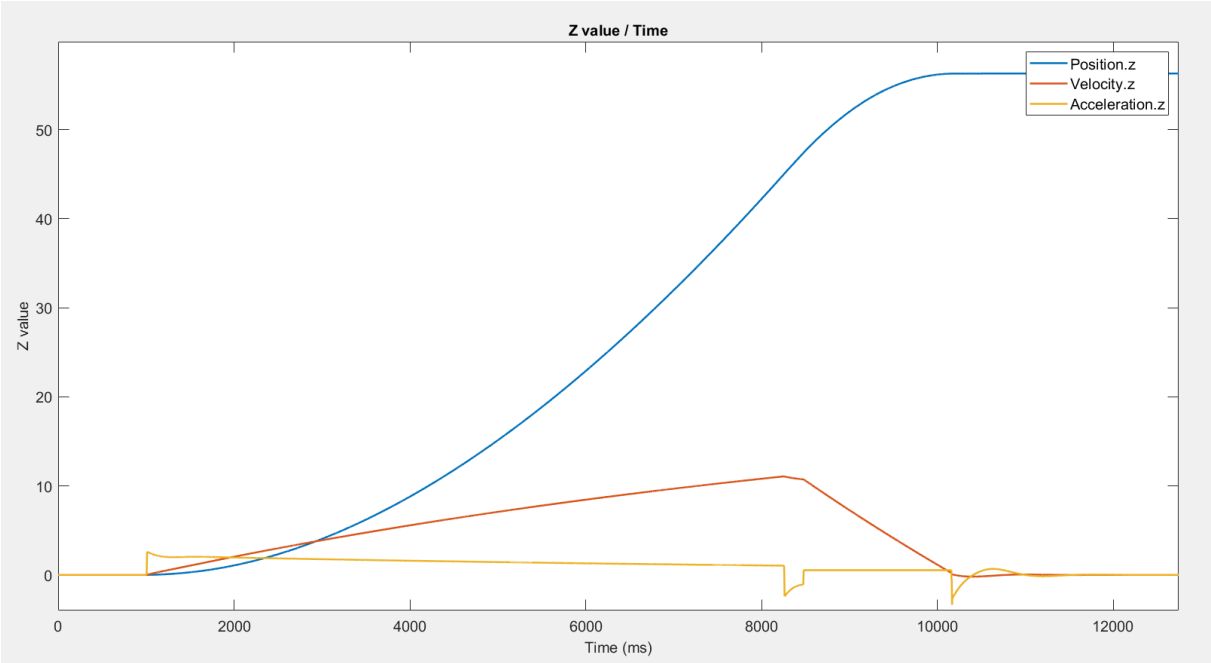
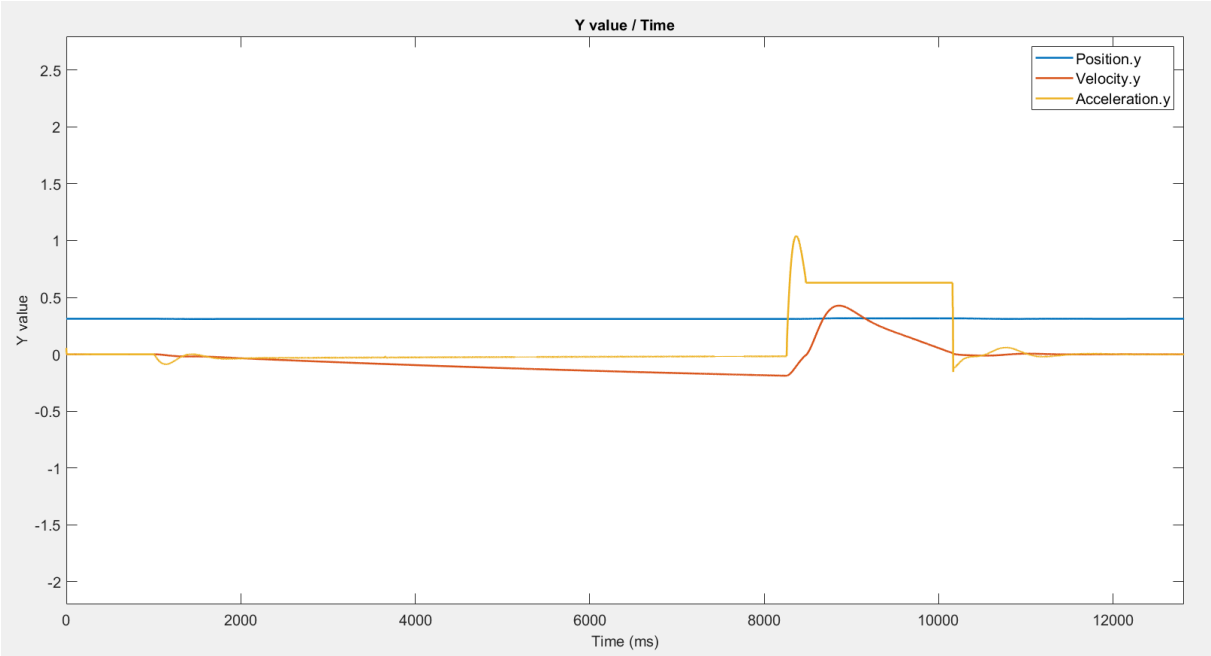


Figure 7.2 *Forward* test - input values

Figures 7.3a and 7.3b show the position, velocity and acceleration values produced by this test in the z and y axes (Unity), respectively. The values in the y axis (Unity) are minimal, but show that the initial acceleration of the vehicle produces a slight movement in the y axis, due to the vehicle's suspension. The x axis (Unity) is not shown for this test, as it does not have any relevant values. Note that the graphs don't have the same amplitude scale.



(a) z axis (Unity)

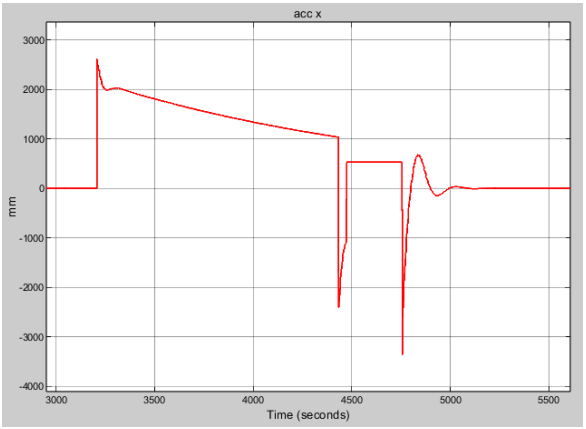


(b) y axis (Unity)

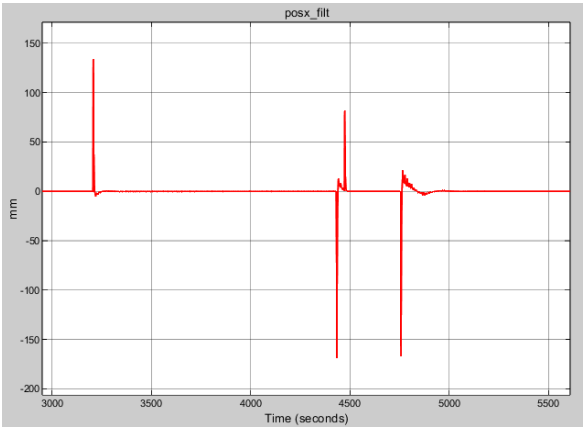
Figure 7.3 *Forward* test - position, velocity and acceleration

From the graphs, it can be observed that there is a clear and expected rise in the acceleration at the start of the test. After the acceleration, there is a drop in the acceleration of the vehicle, due to the vehicle's suspension. When the vehicle starts braking, there is another drop in acceleration, and then the vehicle comes to a full stop.

Figure 7.4 depicts the graphs produced by the platform control software using MATLAB Simulink. These graphs correspond to the values depicted in Figure 7.3a, on the z axis (Unity), the forward axis, however, the platform processes these values as the x axis (Platform). Figure 7.4a depicts the received acceleration values. Figure 7.4b represents the position values sent to the platform. Four different spikes can be identified. The first represents a forward movement, of about 12.5 cm, corresponding to the initial acceleration. The second and third correspond to the stop in the acceleration, where the vehicle repositions itself and starts slowing down due to friction. The last spike corresponds to the vehicle coming to a full stop after braking, where the platform moves about 16 cm backwards.



(a) Acceleration values produced by the vehicle



(b) Calculated position values for the platform

Figure 7.4 Processed values using the washout filter

7.1.2 Speed Bump

In the *Speed Bump* test scenario, there are two speed bumps along the vehicle's path. The first is 35 m after the starting point and the second is at the 80 m mark. The vehicle's approach to both speed bumps is similar, it starts by accelerating and stops accelerating a few meters before going over the speed bump. In the case of the first speed bump, the vehicle begins to accelerate again after going over it. In the case of the second speed bump, the vehicle does not accelerate again and the stopping of the vehicle is not covered by this scenario, to better analyze different types of movements. The path taken by the vehicle is pictured in Figure 7.5. Figure 7.6 shows these changes in the accelerator and brake input.

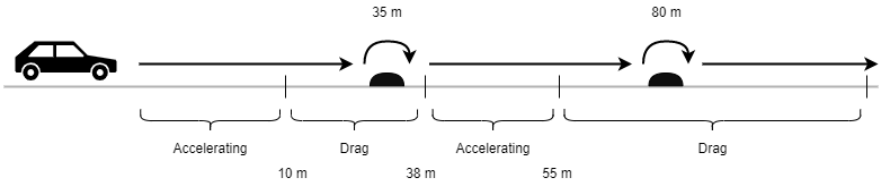


Figure 7.5 *Speed Bump* test - vehicle path

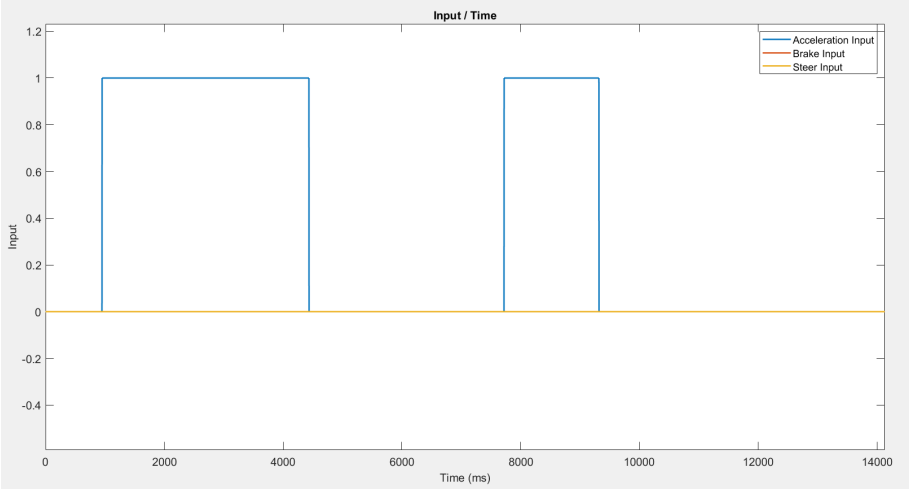
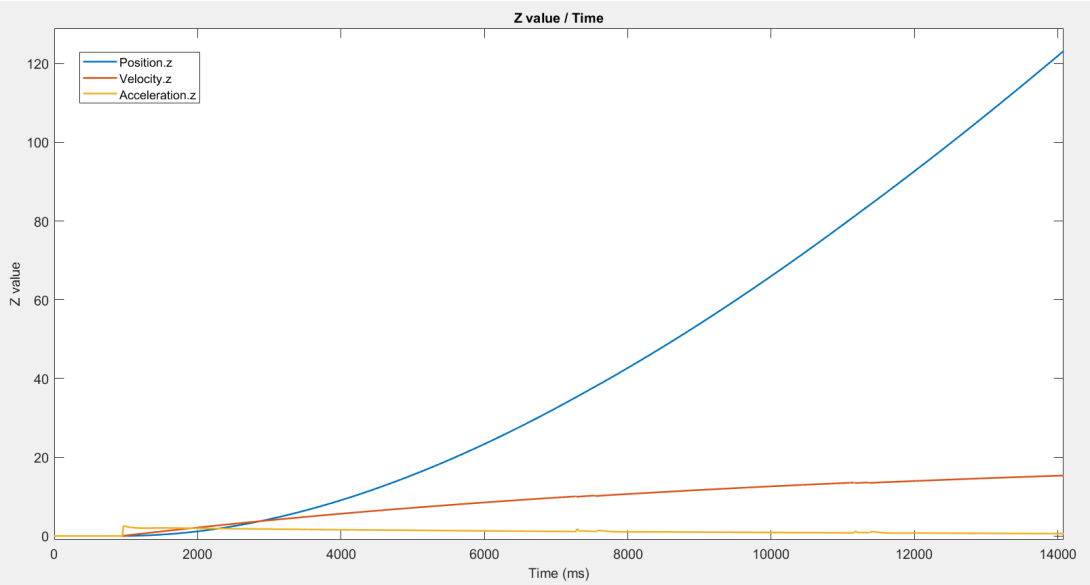
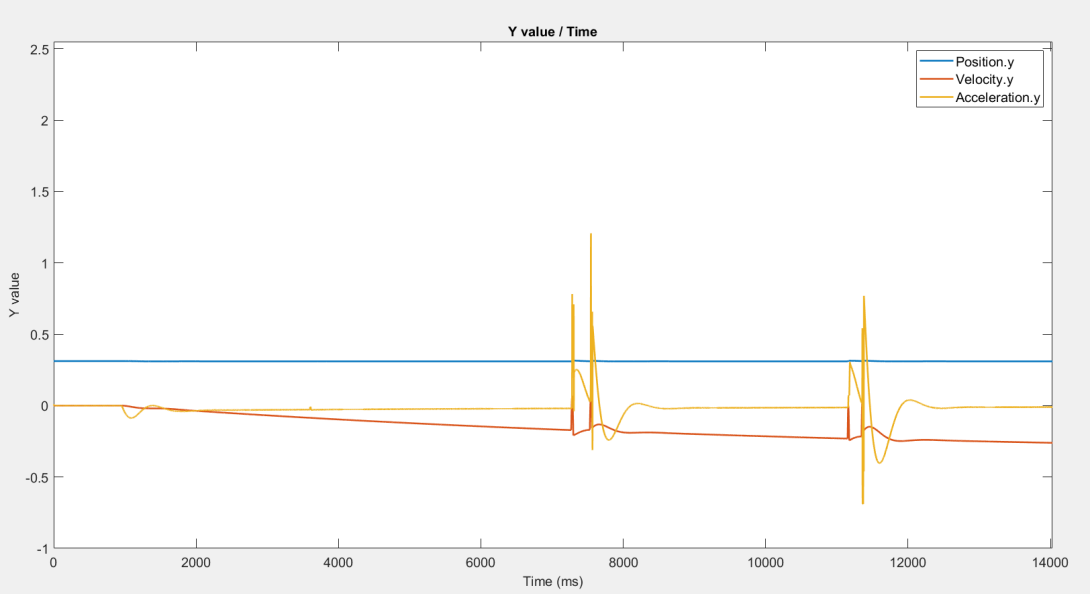


Figure 7.6 *Speed Bump* test - input values

Figures 7.7a and 7.7b show the position, velocity and acceleration values produced by this test in the z and y axes (Unity), respectively. The x axis (Unity) is not shown for this test, as it does not have any relevant values.



(a) z axis (Unity)



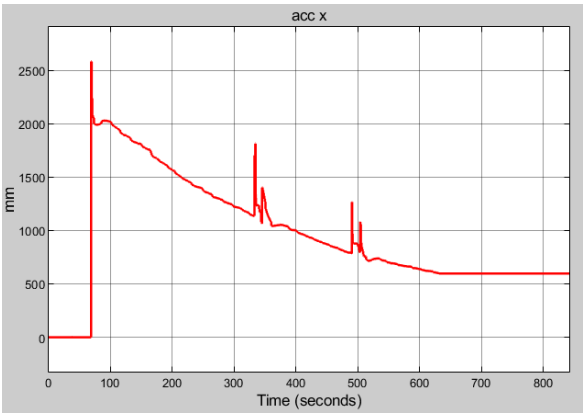
(b) y axis (Unity)

Figure 7.7 *Speed Bump* test - position, velocity and acceleration

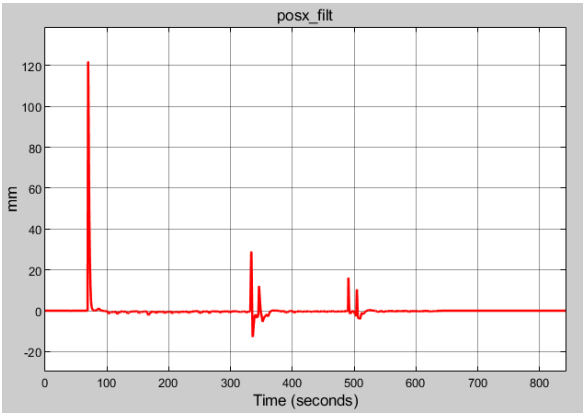
Figure 7.7a, shows a similar scenario to the *Forward* scenario in the z axis (Unity), the major difference can be seen in Figure 7.7b, the y axis (Unity), where there are a few acceleration variations. These correspond to the vehicle going over the two speed bumps. Two groups of variations can be identified, one before the 8000 ms mark and the other before the 12000 ms mark, one for each speed bump. In each group, there can also be identified two different spikes, the first corresponding to the front wheels and the second to the rear wheels. In the second speed bump, the spikes are closer together as the vehicle goes over it at a higher velocity.

Figures 7.8 and 7.9 depict the graphs produced by the platform control software using MATLAB Simulink. In these graphs the axis previously referred to as z axis is now the x axis and the y axis is the z axis, due to the difference in the coordinate system between Unity and MATLAB Simulink. In Figure 7.8a there is an initial spike in acceleration, due to the vehicle starting to move and then two smaller groups of spikes in acceleration due to the speed bumps. These values are then transformed into a big initial forward movement in the platform, of about 12 cm and two groups of smaller movements, of only about 2 cm, for the different wheels going over the speed bumps. The movements caused by the speed bumps are more noticeable in Figure 7.9b, which shows up and down movements and represents the calculated position movements for the platform using the acceleration values shown in Figure 7.9a.

The spikes in Figures 7.8 and 7.9 happen at the same time, which means that the platform performs movements in different axes at the same time.

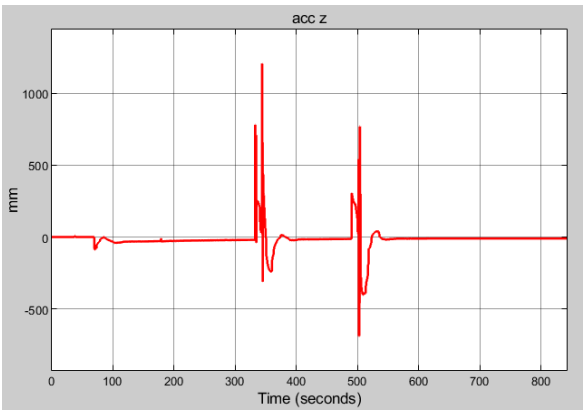


(a) Acceleration values produced by the vehicle in the x axis (Platform)

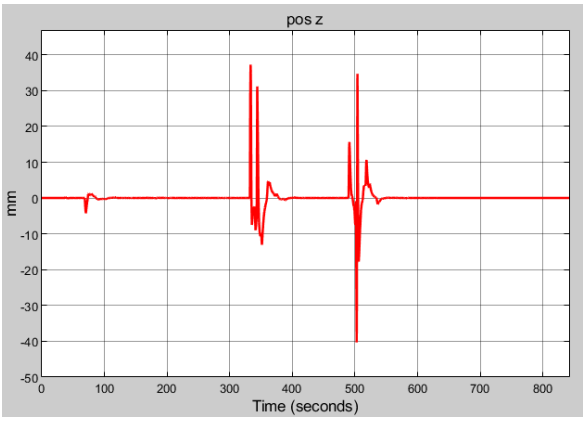


(b) Calculated position values for the platform in the x axis

Figure 7.8 Processed values in the x axis (Platform) using the washout filter



(a) Acceleration values produced by the vehicle in the z axis (Platform)



(b) Calculated position values for the platform in the z axis

Figure 7.9 Processed values in the z axis (Platform) using the washout filter

7.1.3 Curve

In the *Curve* test scenario, the vehicle starts by accelerating for a few meters. After the initial acceleration, the vehicle stops accelerating, but it does not steer yet. After 20 m of not accelerating, the vehicle starts turning to the right. When the vehicle reaches an angle of 65º, it stops turning and continues to slowly decelerate. The braking was not included in this test scenario, to better analyze the movements of the platform. The path taken by the vehicle is pictured in Figure 7.10. Figure 7.11 shows these changes in the accelerator and steer input.

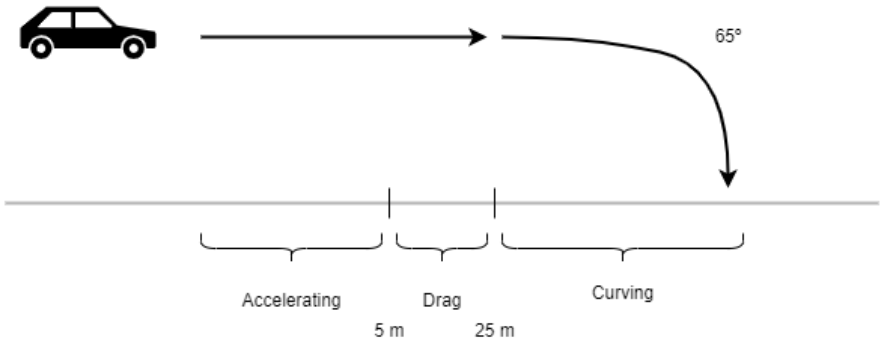


Figure 7.10 *Curve* test - vehicle path

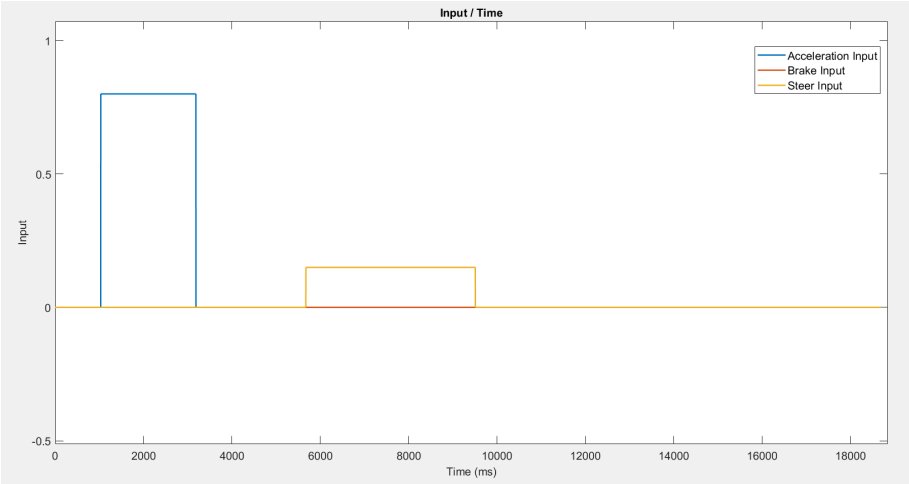
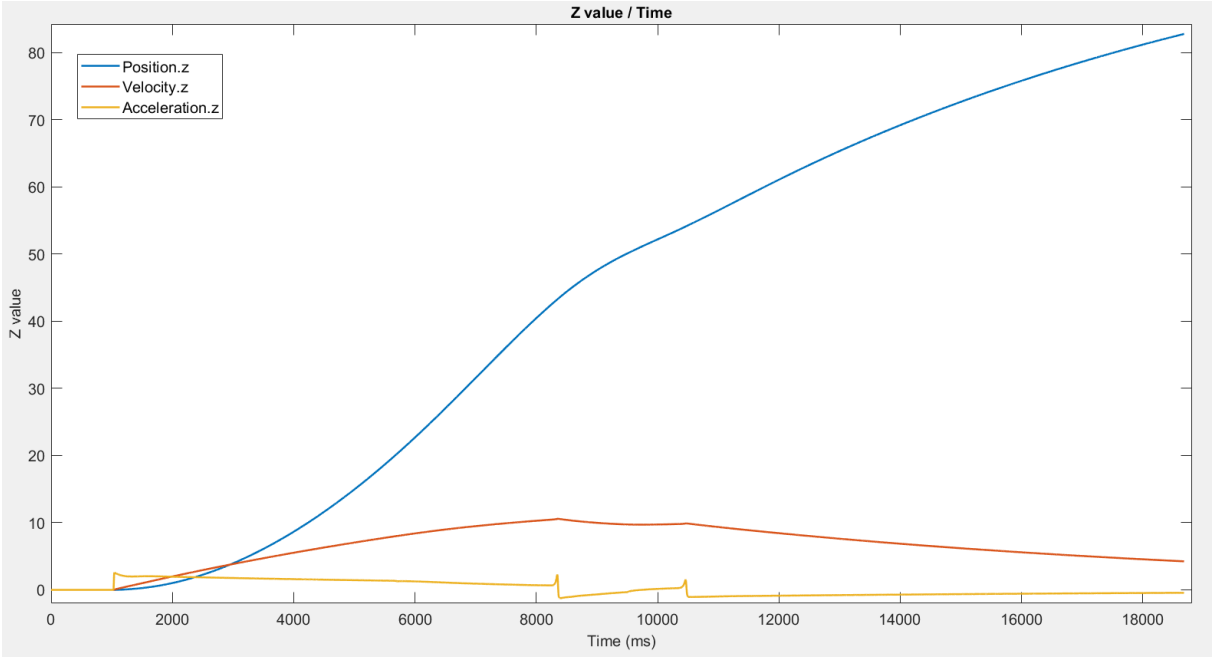
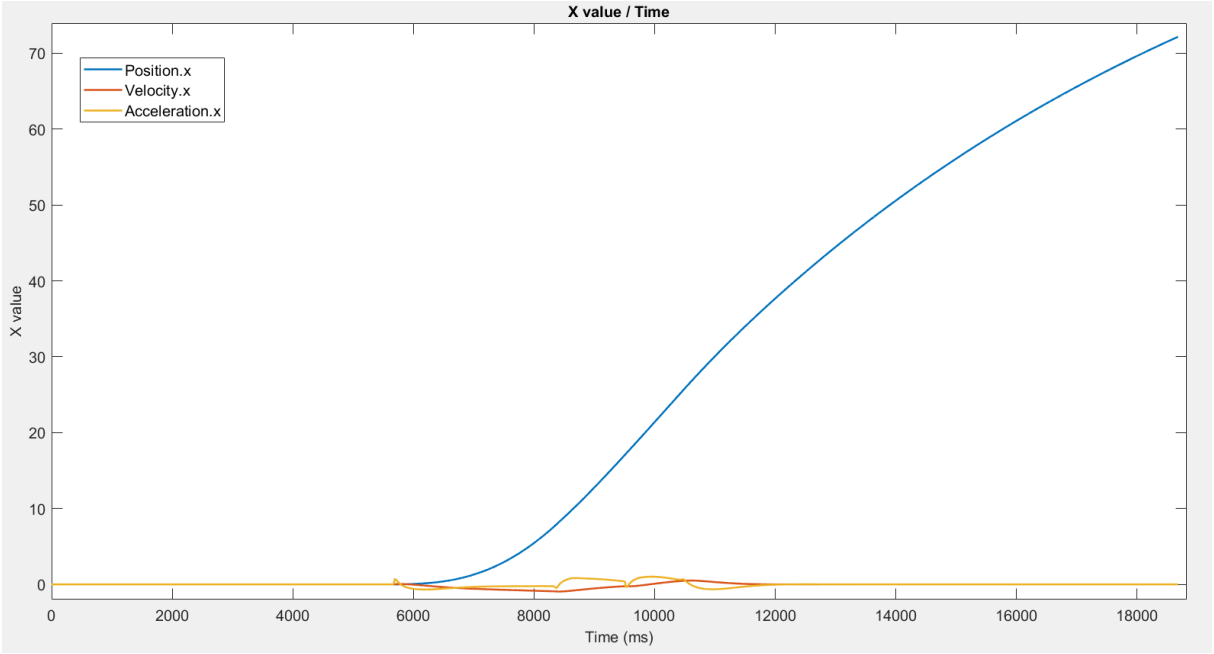


Figure 7.11 *Curve* test - input values

Figures 7.12a and 7.12b show the position, velocity and acceleration values produced by this test in the z and x axes (Unity), respectively. The y axis (Unity) is not shown for this test, as it does not have any relevant values.



(a) z axis (Unity)



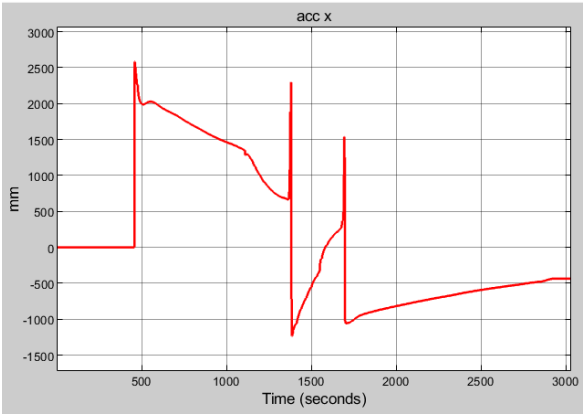
(b) x axis (Unity)

Figure 7.12 *Curve* test - position, velocity and acceleration

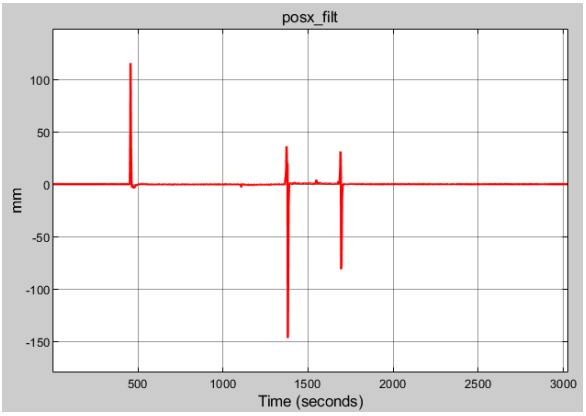
The *Curve* test scenario is more complex to analyze than the previous scenarios, however, the values shown in Figure 7.12a, the z axis (Unity), are similar to the previous scenarios with some decelerations and in Figure 7.12b there are some variations in the acceleration values due to the car turning and its suspension adapting.

Figures 7.13 and 7.14 depict the graphs produced by the platform control software using MATLAB Simulink. Recall that in these graphs the axis previously referred to as z axis is now the x axis and the x axis is the y axis, due to the difference in the coordinate system between Unity and MATLAB Simulink.

Figures 7.13a and 7.14a show the received acceleration values in the x and y axes (Platform), respectively. In Figure 7.13b, the calculated position values for the x axis (Platform) are depicted, showing an initial spike due to the initial acceleration and two other spikes corresponding to the beginning of the curve and the end of the curve, where the car starts to reposition itself due to the vehicle's suspension. In Figure 7.14b, the calculated position values for the platform are depicted, with several varying movements during the duration of the curve.

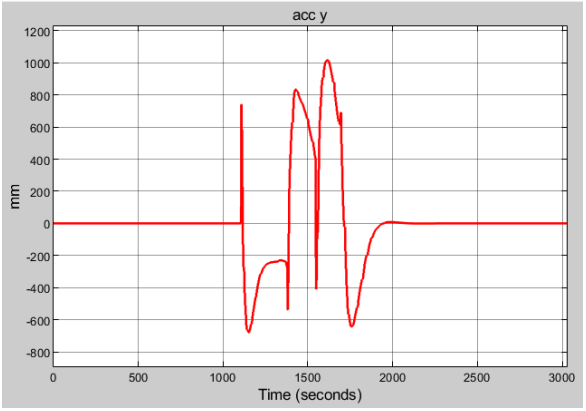


(a) Acceleration values produced by the vehicle in the x axis (Platform)

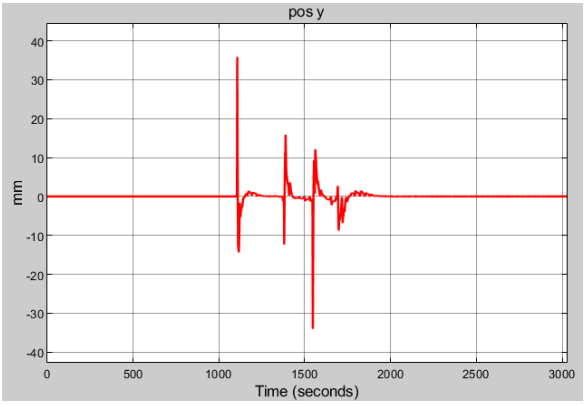


(b) Calculated position values for the platform in the x axis

Figure 7.13 Processed values in the x axis (Platform) using the washout filter



(a) Acceleration values produced by the vehicle in the y axis (Platform)



(b) Calculated position values for the platform in the y axis

Figure 7.14 Processed values in the y axis (Platform) using the washout filter

The test scenarios detailed in this Chapter were conducted using the Stewart platform, where the platform accurately performed the movements calculated by the Stewart platform control software, portraying the movement simulated by the Driving Simulator Software.

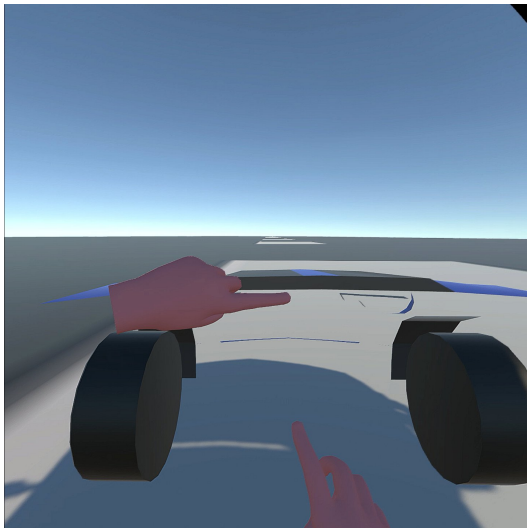
7.2 Utilization of VR

As previously mentioned, although the utilization of virtual reality is not essential to this project, it offers a greater sense of immersion. Besides being able to look around the virtual environment, from inside the virtual vehicle, one other aspect that adds to the experience is the visual representation of the steering wheel. If the calibration process, detailed in Chapter 3, is performed correctly, the user is able to position their hands in a virtual steering wheel that appears to have the same dimensions as the real one. When the user moves the physical steering wheel, the virtual steering wheel moves accordingly, providing a better experience.

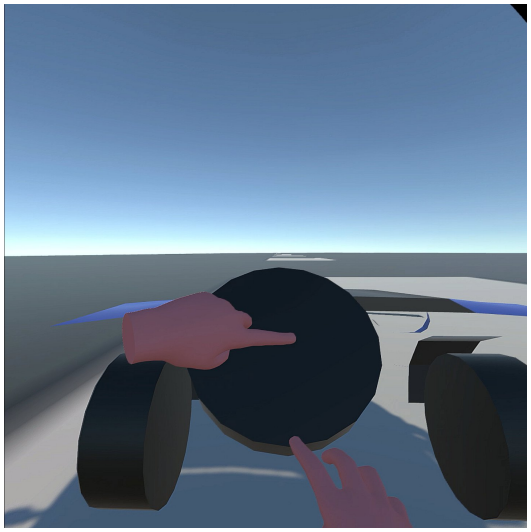
The calibration process is the responsibility of the user, who is able to set the virtual steering wheel to the dimensions they desire at the time of the application's execution. The application is completely independent of the physical steering wheel's dimensions, configured only with the wheel's degrees of steering.

Two steps of the calibration process are depicted below. Figure 7.15a shows the user following the calibration process, putting the left index finger in the middle of the steering wheel and the right index finger in the bottom of the steering wheel. Figure 7.15b shows the steering wheel in the correct place, after the calibration process. The user is holding the physical steering wheel and the virtual hands can be seen holding the virtual steering wheel at the right places.

The application executes in the same manner independently of how the calibration was performed, as it is only a visual aspect of the project.



(a) User calibrating the steering wheel



(b) Calibrated steering wheel

Figure 7.15 Steering wheel calibration steps

Chapter 8

Conclusions and Future Work

This final Chapter has two Sections. In Section 8.1, a final analysis of the project is made, detailing what was achieved and how the objectives of the project were fulfilled. In Section 8.2, some aspects of the project that can be improved are detailed.

8.1 Conclusions

The finished product of this project consists of a system that allows a user to experience the effects of the Stewart platform in different test scenarios, choosing between experiencing them with the use of VR or not.

In Chapter 2, an early version of the project's architecture was introduced. With the project implemented and the different modules detailed, a more comprehensive version of the architecture can be analyzed. In Figure 8.1, the complete architecture of the project is shown. Comparing it to the version from Chapter 2, the technologies are now specified, in the case of the driving simulator and the motion algorithm, with the latter also having its structure more detailed, highlighting the usage of the *DLL* and the executable that uses it that were developed in the project.

Considering the main objective proposed, the project was developed with the concepts of modularity and loose coupling in mind. This way, in further iterations of the project, if it is to be extended or modified, the different modules can be replaced by equivalent ones or can even be modified internally without causing impactful changes in the project. It also allows for each module to be tested separately, without needing the remaining ones to work.

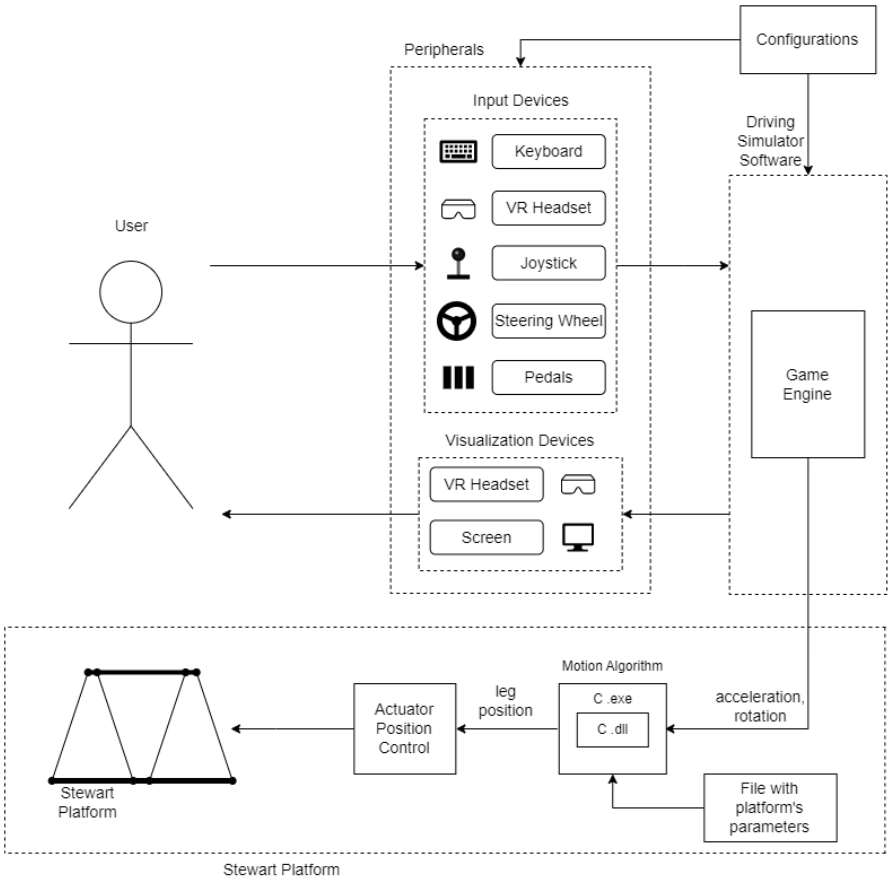


Figure 8.1 The complete architecture of the project

In terms of input devices, as mentioned in Section 3.5, the Unity project currently offers support for two input devices, a keyboard and a steering wheel, which can be used interchangeably, depending only on the user's choice. If there is a need to add support for more devices, only the Input Action Asset needs to be changed, without needing to change any code.

Inside the Unity project, as each class is built around a single functionality and they are attached to a *GameObject*, they can be removed or replaced by a similar class without causing much effect on the project as a whole. However, regarding the driving simulator module, the aspect that may be more relevant to modify is the vehicle. In the final stage of the project, the vehicle model used in testing was a common four-wheel vehicle, but it may be useful to test with other vehicles, such as buses, motorcycles, trucks etc. Only the *Vehicle GameObject* and some of its child *GameObjects* need to be changed. Changing the model of the vehicle and the *Wheels GameObjects* that are accessed by the *VehicleController* class is enough to change the vehicle to be tested without having to implement other components, as the class is independent of the number of wheels the vehicle has. The *Rigidbody* component of the *Vehicle GameObject* has properties regarding the vehicle's mass and can be parameterized.

When the application sends the acceleration and rotation data, through the *Platform Interface GameObject*, it isn't required that the data is sent to the platform's control software. The data can be sent to a TCP server prepared to receive data in the same format, allowing the driving software module to be tested independently.

In the platform's control software, the feature described above also applies. The software, as described in Section 4, consists of a TCP server as an entry point and a TCP client that sends the processed data. It does not require that the data be sent from the driving simulator and then sent to the platform. The data can be sent from another TCP client that sends data in the same format, from a text file for example, and then sent to another TCP server for testing purposes, which allows the project to be tested without the need of a physical platform.

Within the platform's control software, the way it works can also be easily changed. Several parameters of its internal functions are stored in a configuration file, which is used when generating the library. This file can be expanded to store more software parameters. The model can also be modified internally without affecting the rest of the project.

The developed project could also support the use of different Stewart platforms, but it would be more difficult, as it would have to be a platform that supported the same communication protocol. The platform's dimensions would also have to be changed in the control software, but this last detail could be simplified if the configuration file mentioned above stored these values.

8.2 Future Work

The future work can be related to the Driving Simulator Software, the Stewart platform control software or the project as a whole. Besides what can be improved, more tests for each of these components can also be developed to have a better comprehension of the capabilities of the project.

Implementing the movement of a vehicle in code in a realistic way can be very difficult, as several factors affect it. Therefore, the values used in Unity to implement it may not be as accurate as they can be, and some different values could provide more realism.

In the driving simulator software, the user experience could be improved in some points, such as a better UI, more automatic tasks, such as an automatic calibration of the steering wheel or being able to use hands to interact with the UI, when using VR, instead of resorting to the desktop's mouse. Even though these improvements could provide a better user experience, the time needed to do them wouldn't be justified when looking at the project's scope.

As previously mentioned, modularity and flexibility were a big focus when developing the project, and every component was designed with these concepts in mind. However, in some components, these concepts were easier to implement. In the case of the platform's control software, it was harder, as it was a pre-existing piece of software. This component could be modified to use variables from a file that is loaded when executing it, to facilitate the testing and the usage of the model.

As the complete system involves several different components that communicate between them through different communication protocols, this may introduce some latency to the system. As the system requires different devices and even different machines to work, this latency can be difficult to measure, however, an external device could be used to measure the time between the user's input to the platform's response. As long as this time is below the average human response time, it is not a problem.

Bibliography

[1] Volkswagen driving simulator. URL: https://www.nextperception.eu/2022/03/01/human-in-the-loop-in-driving-simulation-a-nextperception-evolution. (accessed: 19.12.2024).

[2] Donald Fisher, Matthew Rizzo, Jeff Caird, and John Lee. *Handbook of driving simulation for engineering, medicine, and psychology*. CRC Press, 01 2011.

[3] Nads-1 Simulator. URL: https://dsri.uiowa.edu/nads-1. (accessed: 19.12.2024).

[4] Script lifecycle flowchart. URL: https://docs.unity3d.com/Manual/ExecutionOrder.html. (accessed: 19.12.2024).

[5] Fanatec CSL DD Manual. URL: https://fanatec.com/media/pdf/75/8e/6f/P1904A-CSL-DD-Manual-EN_03.pdf, . (accessed: 19.12.2024).

[6] R. Pereira, J.C. Quadrado, and Fernando A. Silva. Electromechanical gough-stewart platform direct dynamic study. In *Proceedings of the 8th Portuguese Conference on Automatic Control (CONTROLO 2008)*, Vila Real, Portugal, July 2008. University of Trás-os-Montes and Alto Douro.

[7] Nelson Filipe Pereira dos Santos. Modelização de um sistema de simulação de uma aeronave aplicado a uma plataforma eletromecânica de gough-stewart. 2010.

[8] Petr Bouchner. Interactive driving simulators – history , design and their utilization in area of hmi research. URL https://api.semanticscholar.org/CorpusID:140112037.

[9] P.A. Hancock, D.A. Vincenzi, J.A. Wise, and M. Mouloua. *Human Factors in Simulation and Training*. CRC Press, 2008. ISBN 9781420072846. URL https://books.google.pt/books?id=cgT56UW6aPUC.

[10] N. Beni. Realization of a soft-real-time automotive simulator with human interaction. Master's thesis, Politecnico di Milano, 2013.

[11] History of Virtual Reality. URL: https://www.vrs.org.uk/virtual-reality/history.html, . (accessed: 19.12.2024).

[12] Catchup with Ivan Sutherland - Inventor Of The First AR Headset. URL: https://www.forbes.com/sites/johnwerner/2024/02/23/

catchup-with-ivan-sutherlandinventor-of-the-first-ar-headset/. (accessed: 19.12.2024).

[13] Virtual Reality. URL: https://en.wikipedia.org/wiki/Virtual_reality, . (accessed: 19.12.2024).

[14] Reality Labs. URL: https://en.wikipedia.org/wiki/Reality_Labs. (accessed: 19.12.2024).

[15] Compare your Meta Quest. URL: https://www.meta.com/quest/compare/, . (accessed: 19.12.2024).

[16] Apple Vision Pro. URL: https://www.apple.com/apple-vision-pro/specs/. (accessed: 19.12.2024).

[17] Bmw M Mixed Reality. URL: https://www.bmw-m.com/en/topics/magazine-article-pool/m-mixed-reality.html. (accessed: 19.12.2024).

[18] Unity. URL: https://unity.com/, . (accessed: 19.12.2024).

[19] Unreal Engine. URL: https://www.unrealengine.com/en-US/unreal-engine-5. (accessed: 19.12.2024).

[20] C# language documentation. URL: https://learn.microsoft.com/en-us/dotnet/csharp/. (accessed: 19.12.2024).

[21] Standard C++. URL: https://isocpp.org/. (accessed: 19.12.2024).

[22] Unity Scene. URL: https://docs.unity3d.com/Manual/CreatingScenes.html, . (accessed: 19.12.2024).

[23] GameObject documentation. URL: https://docs.unity3d.com/ScriptReference/GameObject.html. (accessed: 19.12.2024).

[24] Introduction to components. URL: https://docs.unity3d.com/Manual/Components.html. (accessed: 19.12.2024).

[25] Transform documentation. URL: https://docs.unity3d.com/ScriptReference/Transform.html, . (accessed: 19.12.2024).

[26] MonoBehaviour documentation. URL: https://docs.unity3d.com/ScriptReference/MonoBehaviour.html. (accessed: 19.12.2024).

[27] Awake documentation. URL: https://docs.unity3d.com/ScriptReference/MonoBehaviour.Awake.html. (accessed: 19.12.2024).

[28] Start documentation. URL: https://docs.unity3d.com/ScriptReference/MonoBehaviour.Start.html, . (accessed: 19.12.2024).

[29] Update documentation. URL: https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html. (accessed: 19.12.2024).

[30] FixedUpdate documentation. URL: https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html. (accessed: 19.12.2024).

[31] LateUpdate documentation. URL: https://docs.unity3d.com/ScriptReference/MonoBehaviour.LateUpdate.html. (accessed: 19.12.2024).

[32] Time and frame rate management. URL: https://docs.unity3d.com/Manual/TimeFrameManagement.html. (accessed: 19.12.2024).

[33] OnApplicationQuit documentation. URL: https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnApplicationQuit.html, . (accessed: 19.12.2024).

[34] OnDestroy documentation. URL: https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnDestroy.html, . (accessed: 19.12.2024).

[35] Mesh collider component reference. URL: https://docs.unity3d.com/Manual/class-MeshCollider.html, . (accessed: 19.12.2024).

[36] Enumeration documentation. URL: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/enum. (accessed: 19.12.2024).

[37] Xr Core Utilities package. URL: https://docs.unity3d.com/Packages/com.unity.xr.core-utils@2.2/manual/index.html, . (accessed: 19.12.2024).

[38] Xr Interaction Toolkit package. URL: https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.5/manual/index.html, . (accessed: 19.12.2024).

[39] Xr Origin component. URL: https://docs.unity3d.com/Packages/com.unity.xr.core-utils@2.2/manual/xr-origin-reference.html, . (accessed: 19.12.2024).

[40] Enum XROrigin.TrackingOriginMode. URL: https://docs.unity3d.com/Packages/com.unity.xr.core-utils@2.2/api/Unity.XR.CoreUtils.XROrigin.TrackingOriginMode.html, . (accessed: 19.12.2024).

[41] Xr Hands package. URL: https://docs.unity3d.com/Packages/com.unity.xr.hands@1.3/manual/index.html, . (accessed: 19.12.2024).

[42] Struct XRHand. URL: https://docs.unity3d.com/Packages/com.unity.xr.hands@1.3/api/UnityEngine.XR.Hands.XRHand.html, . (accessed: 19.12.2024).

[43] Struct XRHandJoint. URL: https://docs.unity3d.com/Packages/com.unity.xr.hands@1.3/api/UnityEngine.XR.Hands.XRHandJoint.html, . (accessed: 19.12.2024).

[44] Enum XRHandJointID. URL: https://docs.unity3d.com/Packages/com.unity.xr.hands@1.3/api/UnityEngine.XR.Hands.XRHandJointID.html, . (accessed: 19.12.2024).

[45] The PlayerInput component. URL: https://docs.unity3d.com/Packages/com.unity.inputsystem@1.5/manual/PlayerInput.html. (accessed: 19.12.2024).

[46] Mesh Renderer component. URL: https://docs.unity3d.com/Manual/class-MeshRenderer.html, . (accessed: 19.12.2024).

[47] Camera GameObject. URL: https://docs.unity3d.com/ScriptReference/Camera.html. (accessed: 19.12.2024).

[48] Canvas GameObject. URL: https://docs.unity3d.com/Packages/com.unity.ugui@2.0/manual/UICanvas.html. (accessed: 19.12.2024).

[49] Unity's Input Manager. URL: https://docs.unity3d.com/Manual/class-InputManager.html, . (accessed: 19.12.2024).

[50] Unity's Input Manager. URL: https://docs.unity3d.com/Packages/com.unity.inputsystem@1.7/manual/Processors.html#predefined-processors. (accessed: 19.12.2024).

[51] Unity's input system supported input devices. URL: https://docs.unity3d.com/Packages/com.unity.inputsystem@1.7/manual/SupportedDevices.html, . (accessed: 19.12.2024).

[52] Oculus Quest 2. URL: https://www.meta.com/quest/products/quest-2/. (accessed: 19.12.2024).

[53] Meta Horizon OS. URL: https://www.meta.com/blog/quest/meta-horizon-os-open-hardware-ecosystem-asus-republic-gamers-lenovo-xbox/, . (accessed: 19.12.2024).

[54] Windows Build Settings. URL: https://docs.unity3d.com/Manual/WindowsStandaloneBinaries.html. (accessed: 19.12.2024).

[55] Parallel Robot. URL: https://www.sciencedirect.com/topics/computer-science/parallel-robot. (accessed: 19.12.2024).

[56] Amusement Device US patent. URL: https://patents.google.com/patent/US1789680A/en. (accessed: 19.12.2024).

[57] Position-controlling apparatus patent. URL: https://patents.google.com/patent/US2286571A/en. (accessed: 19.12.2024).

[58] Ilian Bonev. The True Origins of Parallel Robots. URL: https://www.parallemic.org/Reviews/Review007.html, 2003. (accessed: 19.12.2024).

[59] D. Stewart. A platform with six degrees of freedom. *Aircraft Engineering and Aerospace Technology*, 38:30–35, 1965. URL https://api.semanticscholar.org/CorpusID:108542148.

[60] A.V. Sonar, K.D. Burdick, R.R. Begin, E.M. Resch, E.M. Thompson, E. Thacher, J. Searleman, G. Fulk, and J.J. Carroll. Development of a virtual reality-based power wheel chair simulator. In *IEEE International Conference Mechatronics and Automation, 2005*, volume 1, pages 222–229 Vol. 1, 2005. doi: 10.1109/ICMA.2005.1626551.

[61] Simulink. URL: https://www.mathworks.com/products/simulink.html. (accessed: 19.12.2024).

[62] What are the differences between static and dynamic (shared) library linking? URL: https://cs-fundamentals.com/tech-interview/c/difference-between-static-and-dynamic-linking, . (accessed: 19.12.2024).

[63] What is a DLL. URL: https://learn.microsoft.com/en-us/troubleshoot/windows-client/setup-upgrade-and-drivers/dynamic-link-library. (accessed: 19.12.2024).

[64] TCP/IP Receive. URL: https://www.mathworks.com/help/instrument/tcpipreceive.html, . (accessed: 19.12.2024).

[65] TCP Receive. URL: https://www.mathworks.com/help/releases/R2021b/slrealtime/io_ref/tcpreceive.html, . (accessed: 19.12.2024).

[66] TCP Server. URL: https://www.mathworks.com/help/releases/R2021b/slrealtime/io_ref/tcpserver.html, . (accessed: 19.12.2024).

[67] Code Generation Applications. URL: https://www.mathworks.com/help/overview/code-generation.html?s_tid=hc_product_group_bc. (accessed: 19.12.2024).

[68] Embedded Coder. URL: https://www.mathworks.com/help/ecoder/index.html?s_tid=hc_product_card. (accessed: 19.12.2024).

[69] System target file. URL: https://www.mathworks.com/help/rtw/ref/systemtargetfile.html. (accessed: 19.12.2024).

[70] Package Generated Code as Shared Libraries. URL: https://www.mathworks.com/help/ecoder/ug/creating-and-using-host-based-shared-libraries.html, . (accessed: 19.12.2024).

[71] Package code and artifacts. URL: https://www.mathworks.com/help/rtw/ref/packagecodeandartifacts.html, . (accessed: 19.12.2024).

[72] Relocate Code Generated from a Simulink Model to Another Development Environment. URL: https://www.mathworks.com/help/dsp/ug/relocate-code-generated-from-a-simulink-model.html, . (accessed: 19.12.2024).

[73] Mingw-w64. URL: https://www.mingw-w64.org/. (accessed: 19.12.2024).

[74] Manual Variant Source. URL: https://www.mathworks.com/help/simulink/slref/manualvariantsource.html, . (accessed: 19.12.2024).

[75] Manual Variant Sink. URL: https://www.mathworks.com/help/simulink/slref/manualvariantsink.html, . (accessed: 19.12.2024).

[76] Rotation and orientation in Unity. URL: https://docs.unity3d.com/Manual/QuaternionAndEulerRotationsInUnity.html. (accessed: 19.12.2024).

[77] Sung-Hua Chen and Li-Chen Fu. An optimal washout filter design for a motion platform with senseless and angular scaling maneuvers. pages 4295 – 4300, 08 2010. doi: 10.1109/ACC.2010.5530820.

[78] Marko B. Popovic and Matthew P. Bowers. 2 - kinematics and dynamics. In Marko B. Popovic, editor, *Biomechatronics*, pages 11–43. Academic Press, 2019. ISBN 978-0-12-812939-5. doi: https://doi.org/10.1016/B978-0-12-812939-5.00002-1. URL https://www.sciencedirect.com/science/article/pii/B9780128129395000021.

[79] Modbus Application Protocol. URL: https://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf. (accessed: 19.12.2024).

[80] Daniel Berger, Joerg schulte pelkum, and Heinrich Bülthoff. Simulating believable forward accelerations on a stewart motion platform. *TAP*, 7, 01 2010. doi: 10.1145/1658349.1658354.

[81] Anca Stratulat, Vincent Roussarie, Jean-Louis Vercher, and Christophe Bourdin. Improving the Realism in Motion-Based Driving Simulators by Adapting Tilt-Translation Technique to Human Perception. In Hirose, M, Lok, B, Majumder, A, Schmalstieg, and D, editors, *2011 IEEE VIRTUAL REALITY CONFERENCE (VR)*, Proceedings of the IEEE Virtual Reality Annual International Symposium, pages 47–50. IEEE, 2011. URL https://hal.science/hal-01436024. IEEE Virtual Reality Conference (VR), Singapore, SINGAPORE, MAR 19-23, 2011.

[82] Semicircular canal. URL: https://www.britannica.com/science/semicircular-canal. (accessed: 19.12.2024).

[83] Github repository. URL: https://github.com/Filipe-Mendes/tfm-simulador, .

[84] Matlab folder in Github repository. URL: https://github.com/Filipe-Mendes/tfm-simulador/tree/main/MATLAB, .

[85] Fanatec. URL: https://fanatec.com/eu-en/, . (accessed: 19.12.2024).

[86] Unity folder in Github repository. URL: https://github.com/Filipe-Mendes/tfm-simulador/tree/main/Unity, .

[87] Unity Asset Store. URL: https://assetstore.unity.com/, . (accessed: 19.12.2024).

[88] Introduction to Asset Store. URL: https://unity.com/pt/pages/introduction-to-asset-store, . (accessed: 19.12.2024).

[89] Edy's Vehicle Physics. URL: https://assetstore.unity.com/packages/tools/physics/edy-s-vehicle-physics-403. (accessed: 19.12.2024).

[90] Create a car with Wheel colliders. URL: https://docs.unity3d.com/Manual/WheelColliderTutorial.html, . (accessed: 19.12.2024).

[91] Fixed-step size (fundamental sample time). URL: https://mathworks.com/help/simulink/gui/fixedstepsizefundamentalsampletime.html, . (accessed: 19.12.2024).