Final Master Thesis

M.Sc.Automatic Control and Robotics

Using Unreal Engine as an engineering tool for traffic simulation and analysis

MEMORY

Author: Javier Manjón Prado

Director: Ernest Benedito Benet

Co-director: Arnau Doria Cerezo

Date: May, 2020

Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona

Abstract

This project studies the aggravating problem of road traffic and congestion, and some of its solutions: autonomous cars and traffic simulation software. Unreal Engine is presented as a new option for simulating a traffic environment.

An application is programmed that allows the user to create and edit roadtracks by clicking and dragging in the editor, along with autonomous vehicle models to populate them with.

The simulator is tested using different scenarios and obtaining relevant data. Finally, the obtained conclusions are mentioned, along with ideas and guidelines for future development.

ETSEIB

ETSEIB

Contents

1 Introduction **7**
1.1 Objectives . . . 7

2 Background **9**
2.1 Road traffic . . . 9
2.2 Autonomous vehicles . . . 10
2.3 Road traffic simulators . . . 11
2.3.1 SUMO . . . 12
2.3.2 TORCS and MADRaS . . . 12
2.3.3 MATLAB . . . 13
2.4 Unreal Engine as a traffic simulator . . . 13
2.4.1 Unreal Engine . . . 13
2.4.2 Vehicles and validity of physical models . . . 14
2.4.3 Unreal Engine time steps . . . 16
2.4.4 Mass-spring-damper example . . . 17

3 System architecture and models **21**
3.1 Road generator model . . . 22
3.2 Vehicle model . . . 25
3.2.1 Sedan model . . . 25
3.2.2 Vehicle parameters . . . 25
3.2.3 Flocking . . . 26
3.2.4 FlockSedan class . . . 27
3.2.5 FlockSedanLanes class . . . 30
3.2.6 HumanSedanLanes class . . . 32
3.3 Vehicle spawner and destroyer. Exporting vehicle data. . . . 32
3.4 Triggers . . . 33
3.5 Level Blueprint and HUD . . . 33

4 Simulation results and analysis **35**
4.1 Scenarios . . . 35
4.1.1 Road with dense traffic . . . 35
4.1.2 Road with dense traffic and speed limit change . . . 36
4.1.3 Closed loop road . . . 37

5 Environmental impact and budget **39**
5.1 Environmental impact . . . 39
5.2 Budget . . . 39

Conclusions **41**

Appendix A **47**

ETSEIB

List of Figures

1 Number of active cars per year. *Dirección General de Tráfico, DGT*. 9
2 Number of victims of car accidents per year. *Dirección General de Tráfico, DGT*. . . 10
3 Sumo editor window 12
4 Blueprint Visual Scripting. 14
5 Vehicle components. From Unreal Engine documentation 15
6 Representation of a vehicle as a rigid body (top) and as a collection of spring masses (bottom). 16
7 Editor window. 17
8 Viewport window. 18
9 Blueprint Visual Scripting for the mass-spring-damper model. 18
10 Mass-spring-damper model in world. 19
11 Blueprint basic inheritance. 21
12 Unreal Engine spline 22
13 Static meshes for road and guardrails. 22
14 Examples of road created with Road Generator 23
15 Splines generated for a two lane road. 24
16 Example of generated road. 25
17 Sedan vehicle model. 25
18 Speed matching flowchart. 28
19 Road following flowchart. 28
20 Collision avoidance flowchart. 28
21 Vectors generated by the flocking functions. The red arrow indicated the direction to a close neighbour, the black arrow indicated direction towards the lane center and the yellow arrow indicated the road direction. 29
22 Throttle and steering control flowchart. 29
23 Parameters needed to the vehicle control to function. 30
24 Collision avoidance taking lanes into account. This function is executed twice with different detection ranges, once for vehicles in the same lane, and once for adjacent vehicles. 31
25 Roadblock generated by a single initial vehicle collision. 31
26 Second type of vehicle. The color has been changed to red so that the vehicles are clearly identifiable. 32
27 Model of car spawner. 33
28 HUD display. 34
29 Test road. 35
30 Speed history for each vehicle when only one type of vehicle present. T = 0.5s. . 35
31 Speed history for each vehicle when the two types of cars are present. T = 0.5s . 36
32 Speed history for each vehicle when only one type of vehicle present and there is a speed limit variation. T = 0.5s 36
33 Speed history for each vehicle when the two types of cars are present and there is a speed limit variation. T = 0.5s 37
34 Road model. 37
35 Speed history of flock vehicles sampled every 2 seconds. 38
36 Speed history of human driven vehicles sampled every 2 seconds. 38
37 Save Text node 47

1 Introduction

1.1 Objectives

The main objective of this project is to create a free, open-source and user-friendly platform/ framework based on a game engine for simulating road traffic.

There are specific objectives that this platform must meet. These are:

- The simulations must be as **deterministic** as possible, and meet the required levels of **realism** in order to be useful for engineering applications.

- The platform must have ready-to-use models for vehicles and roads, and allow the user to edit them easily.

- The code must be scalable. New users must be able to edit and reprogram the software in order to meet their requirements.

The main aims for creating this simulation platform are the following:

- Encourage learners to apply their theoretical knowledge and test their ideas in a simulation environment.

- Serve as a starting point from where users can develop.

- The last objective is related to *gamification*. *Gamification* is the use of gaming techniques and elements in learning activities to increase motivation, learning rates, engagement, etc. This project aims to make traffic simulation and autonomous car programming engaging to the user.

2 Background

2.1 Road traffic

Road traffic refers to all the vehicles that are moving along a particular road or public highway. These vehicles can be buses, streetcars, cars, etc., and are often divided into heavy-duty and light-duty vehicles. In order to regulate and organize traffic, there are traffic laws and informal rules, such as the establishment of priorities, lanes, traffic lights and speed limits.

The number of active cars in Spain increases year after year, as shown in figure 1), due to factors such as the extensive and diffuse town planning of the last decade, which has increased the distance between the workplace and the homes of a great amount of citizens. Studies show that up to 37% of workplaces are on a different town to where the citizen lives [1]. As a result, 83% of Spain's population drives at least once a day, with a mean driving time of 73 minutes.

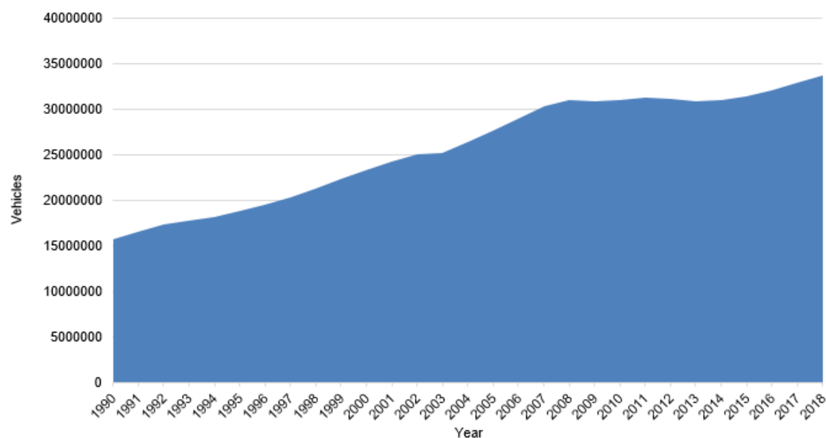

Figure 1: Number of active cars per year. *Dirección General de Tráfico, DGT*.

But this increasing number of vehicles and drive time come with negative side effects. Public health has been one of the areas affected. Passengers lose their free time either in traffic jams or trying to avoid them, which in turn causes stress and loss of attention span that results in car accidents. Up to this day, car accidents is the second most common cause of death in Spain(figure), even though there have been national-wide efforts to decrease the number of accidents. Air pollution is also a public health main issue, even though its effects may only appear after years of exposure.

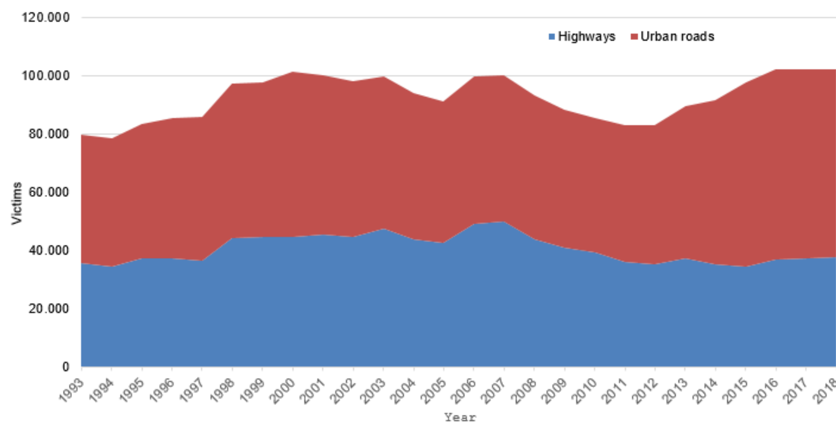

Figure 2: Number of victims of car accidents per year. *Dirección General de Tráfico, DGT*.

Less important, but still relevant, is the time lost. Studies show that passengers lose up to 119 hours per year in traffic jams in cities like Barcelona. This in turn causes an economic impact on the passenger, because of fuel consumption and even affects his quality of life, as he loses leisure and rest time. This lost time also affects national economy, the European Union estimates that the impact of traffic jams in the economy is around 1% of the GDP (Gross Domestic Product).

This traffic congestion problem is temporarily relieved in a manner of different ways, such as building new highways, public transport, low-emission zones (LEZ), etc.

2.2 Autonomous vehicles

A new factor that may improve traffic flow is the autonomous vehicle [2].

An autonomous car is a vehicle capable of operating without human assistance or even presence. According to the Society of Automotive Engineers (SAE), there are currently 6 levels of driving automation, ranging from full manual control (Level 0) to fully autonomous control (Level 5).

- **0. No automation:** The human driver performs all the driving tasks.
- **1. Driver assistance:** There is a single automated system in the vehicle (e.g. cruise control).
- **2. Partial automation:** The vehicle can steer and accelerate on its own. Human monitoring is required.
- **3. Conditional automation:** Includes environmental detection capabilities. The vehicle can perform most tasks on its own, but human control may still be needed.
- **4. High automation:** The vehicle performs all driving tasks under specific circumstances.
- **5. Full automation:** The vehicle performs all driving tasks under all conditions. No human interaction or monitoring is required.

An important difference that defines the level of automation of a vehicle is the ability to sense

the environment and react to it. Typical sensors are LIDAR, stereo vision, GPS and IMU. Sensor Fusion is used to integrate all the information from the sensors into useful and precise information for the vehicles automated system.

Nowadays, it is uncommon to find vehicles with no automation or driving assistance involved. Almost every car has a braking system that adjusts braking force in order to prevent the tires from sliding (ABS). Some even have Cruise Control or automatically steer in order to keep the vehicle between the road markings. Full automation of driving vehicles has widely-known benefits such as:

- Increased safety: Human drivers have a decreased attention span and a slower reaction time than automated vehicles. A wide percent of car accidents are related to human error.
- Greater efficiency: Accident avoidance may indirectly result in lesser traffic jams, with lower fuel consumption, lower pollution, etc. Efficient acceleration and steering manoeuvres also lead to fuel savings.
- Increased lane capacity: With quicker reaction times and environment knowledge, the safe space between vehicles can be reduced.

Despite the advantages, full automation of every car on the road is still not going to be present yet. Besides the engineering problems that need to be solved, one of the main concerns for introducing highly automated vehicles is the legal framework, that responds to both ethical and technical issues. The mean time in which a driver renews his car is 18 years ([1]). So even when the technology is present and legal, its integration on to the transportation system will be slow.

Automated and human driven vehicles will have to share highways that are adapted to human drivers. Only when automated vehicles make up a large percentage of the total number of vehicles, can the road infrastructure be simplified and adapted to new driving standards.

2.3 Road traffic simulators

The need for traffic simulators has risen in recent times due to the traffic problem mentioned above. Traffic simulators are used for research on traffic models in order to plan the development of traffic networks.

Traffic simulation can be used for predicting delays and congestion in highways, as well as air pollution, etc., these predictions in turn may serve as a tool to improve efficiency of the traffic network. Traffic simulators include road traffic, maritime, air and rail transportation simulators, although this section will focus on roadway simulators. There are several different available traffic simulators, each with its own capabilities and areas of application [3], but most require a paid license in order to use them. In this section, the focus will be the free-use road traffic simulators, such as SUMO and TORCS/MADRaS. Another free software that is used is the Next Generation Simulation (NGSIM), developed by the U.S. Department of Transportation. However, lately some research has reported errors in its database regarding vehicle position, velocity and acceleration [4].

2.3.1 SUMO

SUMO (Simulation of Urban Mobility) [5] is an open source microscopic and continuous road traffic simulator. It allows to simulate a given traffic demand, consisting on vehicles moving through a traffic network, at any moment. In this simulator, each vehicle is modelled individually, has it´s own route and moves through the network following it. Simulations on SUMO are deterministic by default, although there is the possibility of introducing randomness.

SUMO allows different vehicle types to be introduced in the simulation, and roads with multiple lanes and functionalities to change lane. SUMO uses XML files for the configuration of the simulation, where streets are defined along with traffic lights, junctions, etc.

The SUMO simulation software analyses traffic flow from a macroscopic point of view. The user will not have easy access to modify the individual vehicle behaviours.

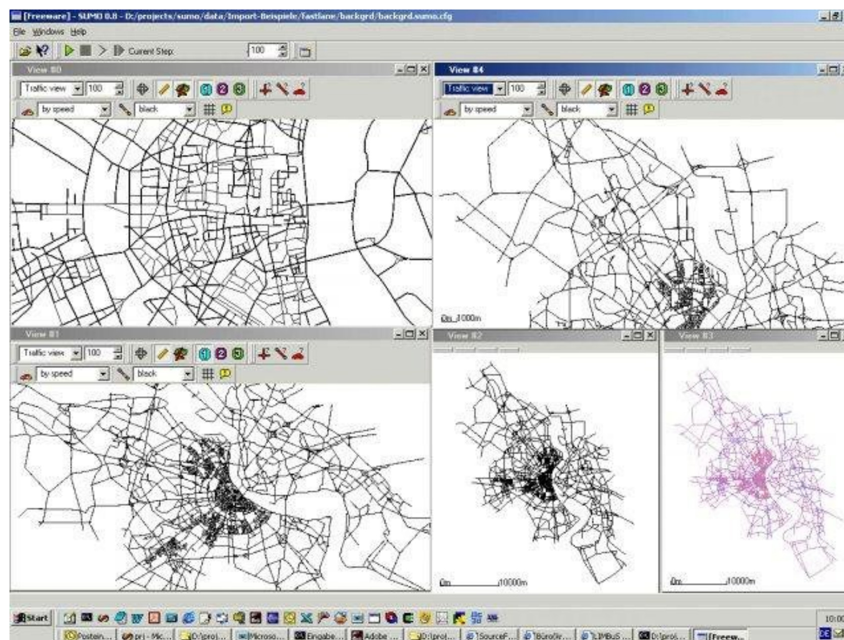

Figure 3: Sumo editor window

2.3.2 TORCS and MADRaS

TORCS [6] is an open source racing car simulator that is also used as a research platform due to the accessibility of its code, that makes it easily modifiable.

The fact that it is open source has led to the community adding many plugins to the software, such as vehicle types, etc. One of the biggest addons for TORCS is MADRaS [7]. MADRaS (Multi-Agent DRiving Simulator) implements a multi-agent version of TORCS, addressing one of the main issues many traffic simulators have. MADRaS allows the user to control independently each vehicle in the scenario. Both TORCS and MADRaS are used when a more microscopic, vehicle-centered is the focus.

The software is Python-based, so knowledge of this programming language is necessary in order to implement new vehicle behaviours. As a downside, the foundation of this software is a

racing car game, which limits the options the user has when creating a road network [8].

2.3.3 MATLAB

Despite requiring a paid license, MATLAB is mentioned here due to its predominance in the academic environment.

MATLAB has several packages related to road traffic simulation, such as *Autonomous Driving Simulation Framework for Traffic Studies* [9] a intersection-focused traffic simulator [10] and *Unreal Engine Driving Scenario Simulation* [11], which uses the Unreal Engine.

The main problem of these packages is that the simulation is very restricted to certain scenarios and types of roads and vehicles. The applications also have a very low limit in the quantity of vehicles present at the same moment in the world.

2.4 Unreal Engine as a traffic simulator

2.4.1 Unreal Engine

Unreal Engine, or UE4.24 in its current version, is an open-source, real-time 3D game engine created by *Epic Games* in 1998.

A game engine is a collection of development tools and environment that make the creation of a video-game easier. These tools include a graphic motor, a programming language, a physics motor that can simulate physical laws such as gravity, collisions, etc, audio, video, artificial intelligence, virtual reality and many more.

Since its creation, it has been used to develop many famous games, such as *Gears of War* and *Mass Effect*.

Although initially Unreal Engine was conceived as a game creation tool, as the computational power, multiple resources and accuracy in representing reality of the engine has grown, so have its applications out of the video-game industry. Recent examples are simulators for robotics [12], high precision surgery [13] [14], pharmaceutics, military applications [15], computational fluid dynamics for aerodynamic design [16], high fidelity museum applications, architecture, vehicle modelling [17] and other vehicle related applications [18] [19].

In order to use its components, Unreal offers users two programming languages: C++, in which the full engine program is built, and its own programming language, called *Blueprint Visual Scripting*, which is a Object Oriented visual scripting system that enables users to program by through wiring together function blocks and property references. A simple example where a loop is used to increment the value of a variable and once the loop ends shows it on screen is shown in figure 4.The sequence of events that activate can be seen in runtime as the program is executed, so it is simple to debug. Break points can be added to pause the Blueprint sequence.

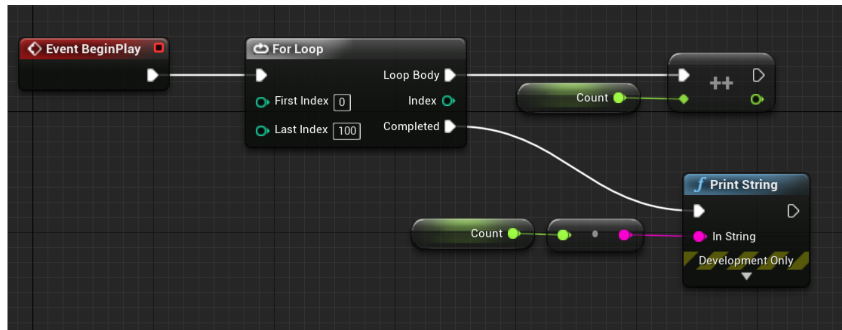

Figure 4: Blueprint Visual Scripting.

Moreover, UE4 is open-source, so every component is accessible, given C++ knowledge and posses ion of the Visual Studio development tool. The License Agreement states that the engine is free unless a gross revenue of 3000$ per quarter is obtained, which means that the software can be used freely for any research related application on the university domain, as there is no marketing of the product.

There are many more reasons as to why using Unreal Engine as an engineering simulator is an advantage. The usage of the graphical and physics motors and real-time simulation relieves the user from work on non-essential issues such as collision detection, physical laws, graphics, etc., and allows him to focus on defining in a higher level and programming the behaviours that the objects, such as vehicles, placed in the world have in relation to it. Using other valid software for simulation, such as Matlab, may mean that the user has to code everything from scratch.

Unreal Engine offers an extensive documentation [20], wiki [21] and free usable packages, assets and C++ classes. For example, Unreal Engine incorporates a Sedan package that includes the physical, collision and graphical model of the vehicle. It also provides a basic control that allows the user to drive the vehicle using the keyboard. There are also professional plugins such as such as *VehicleSim Dynamics* [22] plugin that allows to run industry standard CarSim and TruckSim math models in an Unreal environment, and Microsoft's *AirSim* [23] for autonomous vehicles.

As a downside, Unreal Engine may feel harder to learn and slower to prototype with than other software, due to its many interfaces, options, classes, etc. But the availability of already created and programmed assets (such as the one presented on this Master Thesis) and plugins provides with a quick way of implementing ideas with minimal training required.

As a side note, there is a current project focused in building an Unreal Engine plugin that can convert road data from a SUMO XML configuration files into roads in Unreal Engine [24]. This plugin will not import vehicles or vehicle data of any type, but it may help users with knowledge of the SUMO software to easily build a desired road network in Unreal Engine.

2.4.2 Vehicles and validity of physical models

Before using Unreal Engine as an engineering tool for simulation, it is important to verify the models validity.UE4 uses the *PhysX* physics engine developed by NVIDIA to drive its physical simulation calculations and perform all collision calculations.

NVIDIA is a multinational company known for being one of the most important producers of high-end graphics processing units (GPUs). Besides its production for the gaming industry, NVIDIA GPUs are used in scientific research in areas such as machine learning, virtual and augmented reality, computer vision, etc. The company also has several scientific research departments of its own.

Physics engines are specific software designed for simulating physical systems in other software. There are many commercial engines available, such as *Havok*, *Amazon Lumberyard* or *True Axis*.

NVIDIA PhysX engine uses rigid body dynamics simulation. With this method, the dynamics of several rigid body primitives, such as the sphere, box and cylinder, are preprogrammed. Then, complex objects, regardless of their graphical form, are modelled using combinations of these primitives and joints. This is called the physical model of an object.

PhysX has been successfully used in other applications requiring precision such as virtual surgery with force feedback [25] and other haptic related applications [26], soft robotics [27], etc.

In Unreal Engine, vehicles are created as a sum of different components, as shown in figure 5. Engineering related configuration is stored in the blueprints of both pairs of wheels and the physics asset of the vehicle.

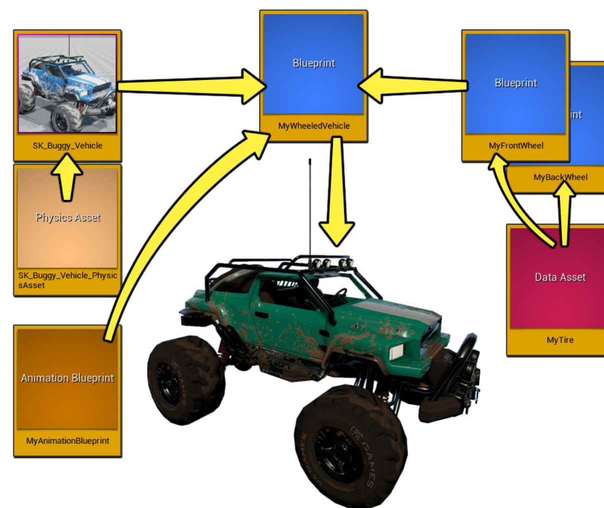

Figure 5: Vehicle components. From Unreal Engine documentation

The physic model of the vehicle that the PhysX engine implements is a combination of a collection of spring masses and a rigid body actor, as shown in figure 6 coupled with other components, as engine, clutch, gearbox, differential, etc. In order to translate one model into the other, the following equations are used:

$$
\begin{aligned}
M &= M_1 + M_2 \\
X_{cm} &= \frac{M_1 X_1 + M_2 X_2}{M_1 + M_2}
\end{aligned}
\tag{1}
$$

Where X_1, X_2 are the spring mass coordinates and M is the rigid body mass.

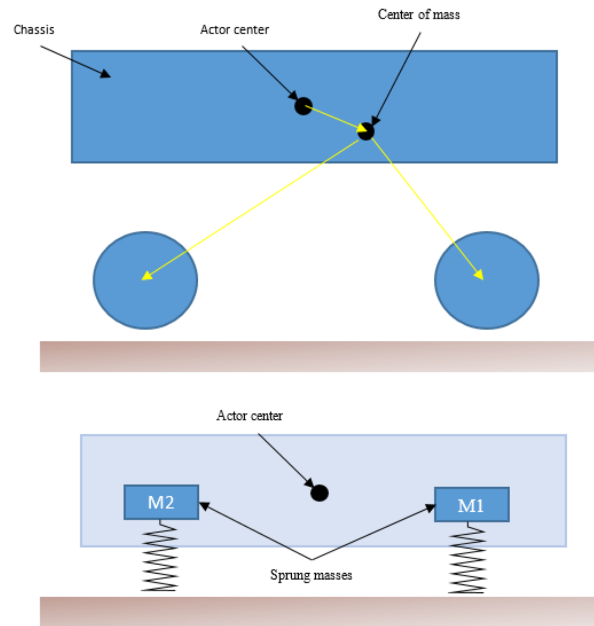

Figure 6: Representation of a vehicle as a rigid body (top) and as a collection of spring masses (bottom).

When using the vehicle in simulation, first the suspension and tire forces are computed using the spring mass representation. The suspension force is applied to both the rigid body and the tire. This load on the tire is then used to determine the tire forces generated in the contact plane, that depend on other parameters such as steer angle, friction, wheel rotation speed, etc. This tire forces are in turn applied to the rigid body of the vehicle.

Finally, the total sum of forces in the vehicle accelerates the vehicle following Newton´s Second Law

In order to apply rotation to the wheels, the vehicle model has a one-dimensional rigid body model of an engine that rotates in direct relationship with the accelerator pedal. This rotational speed and forces are transferred by means of a clutch into the gearing system, the differential and the wheels. More information from the model can be obtained at [28]. The wheel model used for simulations is the Pacejka tire model [29].

2.4.3 Unreal Engine time steps

The physics engine is embedded inside the Unreal Engine, so if the latter should also be as deterministic as possible in order to consider this software for engineering simulations.

Each object placed in the world will be subject to three main events:

- A **BeginPlay** event, which executes when the simulation starts and can be used to set initial states of the objects.

- A **Overlap** and **Hit** event, which executes when the object collides or overlaps with another

object in the world.

- A **Tick** event, which executes every *delta* seconds. Here the user defines the main functionalities and behaviour of the object. Delta is the sample time of the system.

There are more events, such as keyboard input events, and the user can implement any number of custom events he desires.

The main issue for a deterministic simulation is the **Tick** event. According to Unreal Engine's documentation, the tick time is semi-fixed. Since the engine is mainly created for games, it will always prioritize running at real-time for enhancing user experience. This means that if the computational load of the application is too high, the engine may decide to skip or change the sampling time to a higher value in order to maintain real-time. While the effect is barely noticeable in a computer game, it is enough to be accounted for in a simulation for an engineering application.

In order to fix the time steps, there is a section in the Unreal Editor preferences which allows the user to set parameters that define the simulation. Here, the command *benchmark* can be used to make the game run at fixed time steps. In this case, a fixed time-step of 8.3 milliseconds is chosen. This parameters ensures that the game will run at 120 frames per second, and if needed, will reduce this, but will never change the timestep.

2.4.4 Mass-spring-damper example

In order to demonstrate how to implement a model into Unreal Engine, an example project is created. A mass-spring-damper behaviour will be implemented in this project.

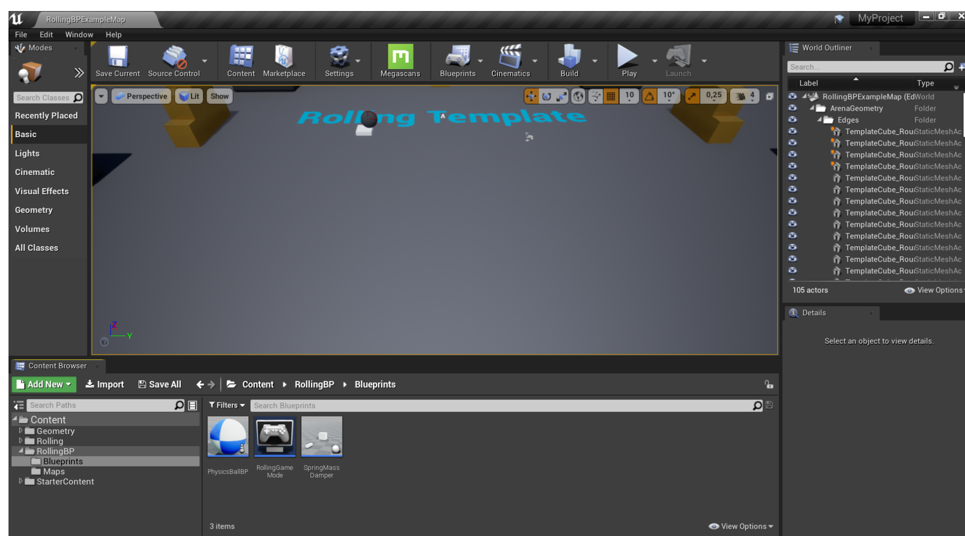

Figure 7: Editor window.

Figure 7 shows the editor window, this window will appear when the project is opened. In order to create a new blueprint, we click "New" in the *Content Browser* (bottom) and select the *Blueprint actor option*. A new window appears, where the blueprint can be created (figure 8). The first step is to select the objects that will conform the actor. In this case, two rectangular shapes are selected; one to act as a wall and another to act as the movable mass.

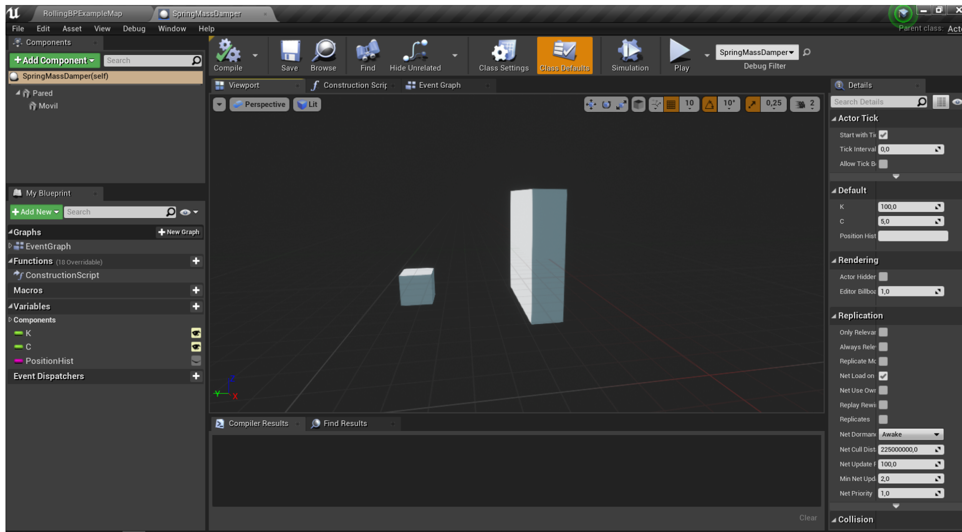

Figure 8: Viewport window.

After selecting and positioning the components, the models behaviour can be added. First, two new float variables, K and C are created, they represent the spring and damper constants. Then, as shown in figure 9, an "AddForce" function is added to the Tick event. This function required at least two inputs: The object upon which this force will be applied, and the direction of the force.

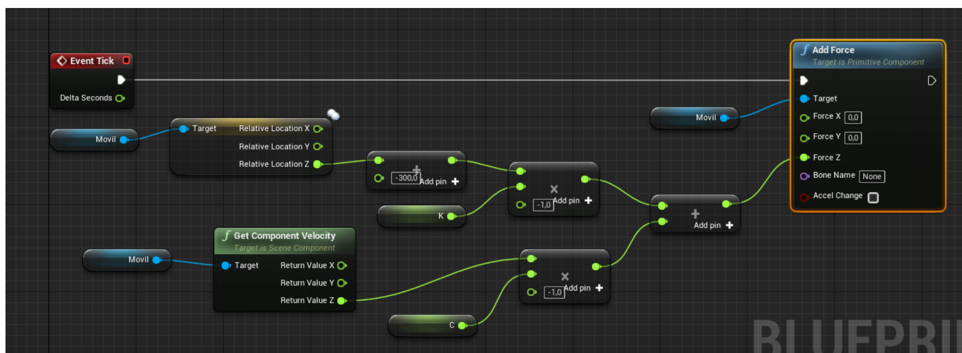

Figure 9: Blueprint Visual Scripting for the mass-spring-damper model.

The object where force is to be applied is the movable part, called "Movil". The direction of the force is computed by assuming a vertical axis restricted movement and applying the mass-spring-damper equation. In the component options of the movable object, a value of mass is selected and the gravity effect disabled.

$$F = -kx - c\dot{x} \tag{2}$$

The speed and position of the movable objects are extracted, multiplied by the coefficients and added to generate a force in a vertical direction.

Once this steps are done, the object can be compiled and saved. Placing the object into the world

and testing it´s behaviour is as simple as dragging the blueprint fro the content browser into the world and pressing the Play button.

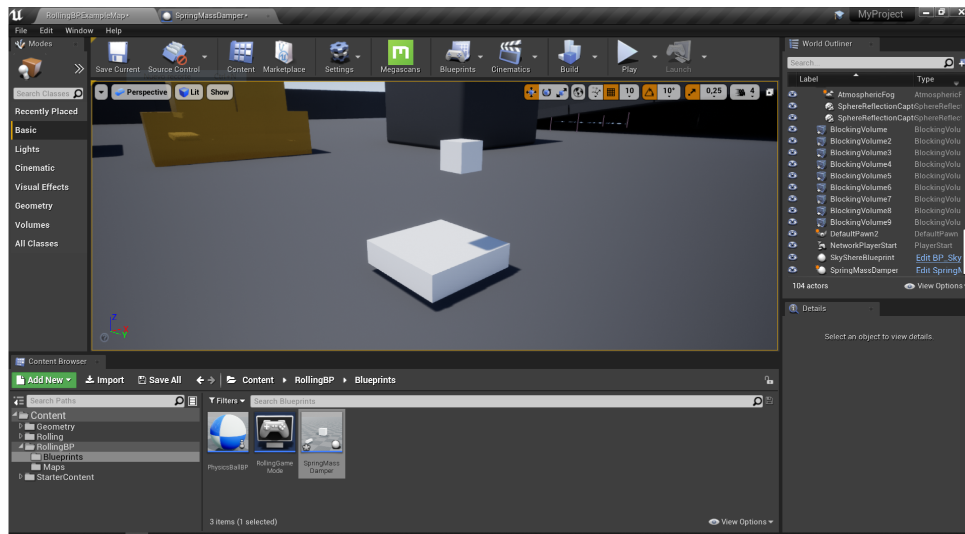

Figure 10: Mass-spring-damper model in world.

3 System architecture and models

The most important building block in Unreal Engine is the Blueprint Class (also called Blueprint), which is an object/class defined either with Blueprint Visual Scripting or C++. Blueprints function by using graphs of Nodes for different purposes, object construction, individual functions and general events that are specific to each instance of the object and implement behaviour and functionalities. Once a blueprint has been programmed, it can be simply dragged and dropped into a world level to create an instance. All instances of a blueprint will update when the blueprint is edited.

As an example, a car and its behaviours and properties (throttle, steering, colour..) can be constructed using a blueprint and assets such as the material of the tires. Then another blueprint with access to the cars properties and functions can be used to control it in order to drive.

As in the C++ programming language, there exists the possibility of child classes or blueprints. Unreal Engine's architecture is heavily class-based. One of the most common parent classes is the *Actor* class, which is basically an object that can be placed or spawned in the world. Actors in turn have child classes, as shown in figure 11. *Pawns* are Actors than implement functionalities in order to be possessed and receive input from a controller such as the *Player Controller* class, *Game Characters* are pawns that can walk, jump, etc.

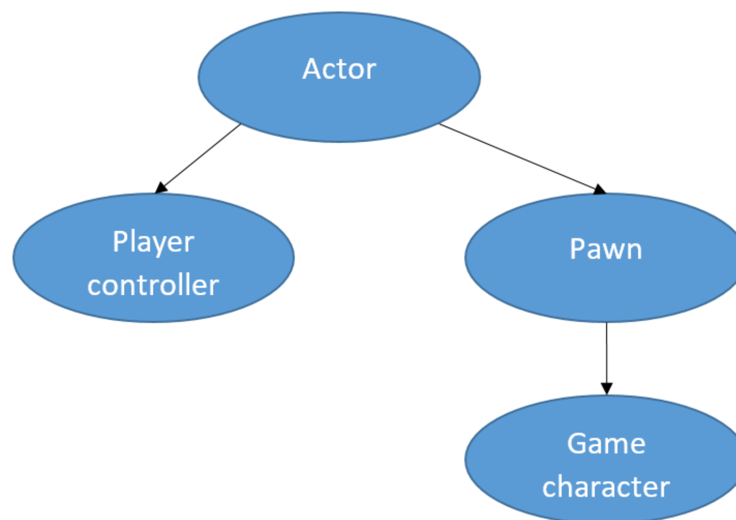

Figure 11: Blueprint basic inheritance.

All game-based functionalities of Actors and derived classes are associated and started by events, such as the *BeginPlay* event and the *Tick* event, that executes the associate code every *delta time* seconds, based on the refresh frequency of the monitor. This Tick event is what implements the sense of real time in the game.

3.1 Road generator model

In order to create a road generator, *splines* will be used as a basic component. Splines are a special math function that is defined piece-wise by a polynomial. Unreal Engine's editor allows the user to create spline components and place them in 3D scenarios. These splines will be used to define the desired trajectory of the road and act as its skeleton. New spline points can be added at any time to edit or add spine segments to the trajectory.

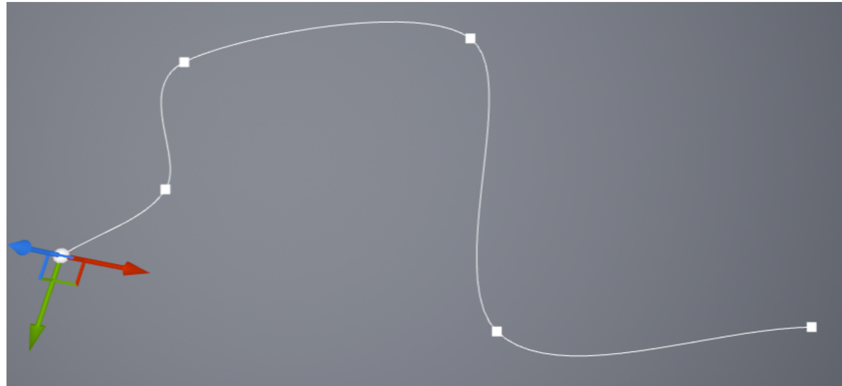

Figure 12: Unreal Engine spline

The editor provides the user with various assets, including static meshes. Static meshes are pieces of geometry that are formed by sets of polygons. They are the way that many game engines handle 3D models such as doors, cars, walls, etc. and their collisions objects. For the creation of the roads, static meshes for the road floor and both guardrails (figure 13) will be used. Although the guardrail meshes are not really necessary, they are very helpful to determine when a vehicle drives out of the road, as the collision will be detected by the engine and an event will be generated.

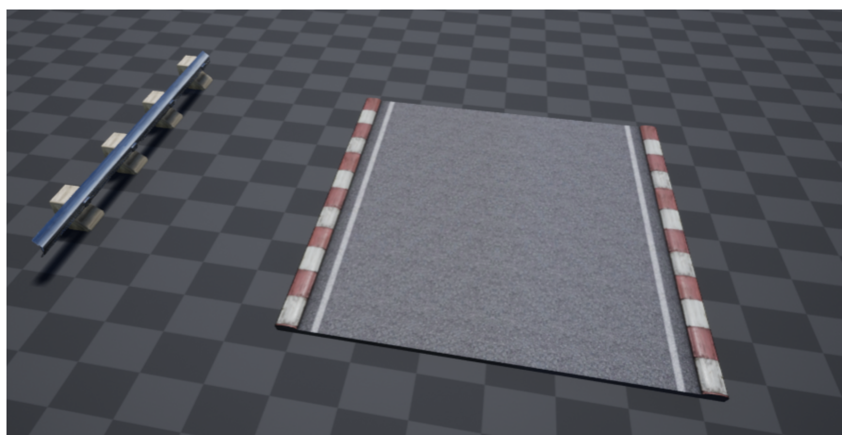

Figure 13: Static meshes for road and guardrails.

Once the road is created in the editor, it does not have to change during runtime. Because of this, the event graph that handles the road behaviour in runtime will remain empty. The programming of the road will be set on the *Construction Script*.

First of all, a structure called *RoadData* is created. This structure will handle the relevant data for each spline segment; whether there are guardrails, the width of the road, the bank of the road, etc. The first part of the Construction Script will obtain the number of spline points that the user has created, and add one of these data structures to an array for each. This array will be visible and editable, so that the user can define the configuration at the road at each point.

Once the spline is created, the objective is to build the road and guardrails static meshes so that they follow the spline. In order to do this, individual static meshes must be assigned to each segment between two spline points. If there are n spline points, there will be $n - 1$ segments. The function *BuildTrackElement* is coded with this objective. This function will use a loop to build the static mesh starting at one spline point and ending at the next. The parameters that are used to build the static mesh are the following:

- Start and end position.
- Start and end tangent.
- A reference to the static mesh.
- Start and end roll.
- Start and end width.

The static meshes parameters will be interpolated in case they vary between the start and end positions. The option to deactivate collisions is also implemented, as building and editing the road with enabled collisions can affect the performance of the editor.

The resulting blueprint will already be able to build roads with the length and shape that the user desires, by only clicking and dragging actions. The downside is that there is only one lane per road. In order to expand the functionalities, a new blueprint is created that implements the same functionalities plus the lane separation capacity.

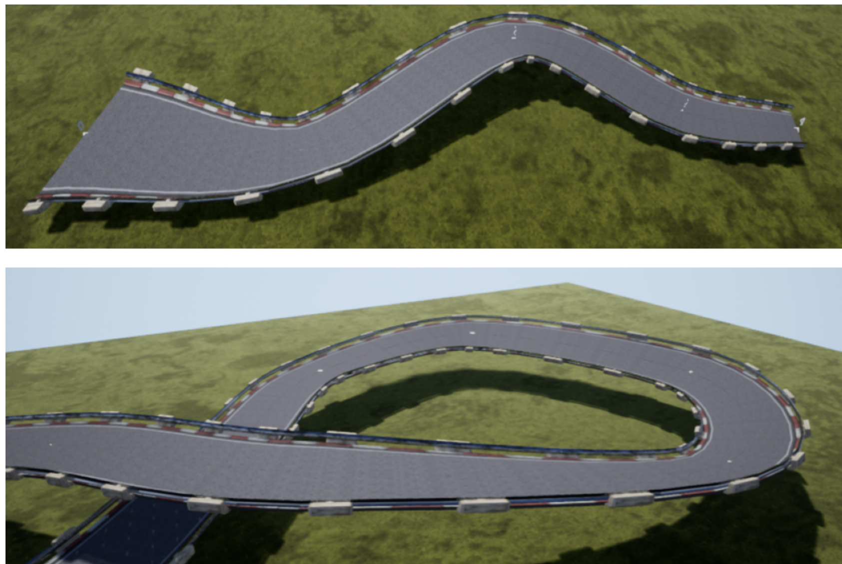

Figure 14: Examples of road created with Road Generator

In order to do so, a new variable is added: the number of lanes. If this number is bigger than one, the width of the road will be automatically changed to $3n$ meters, where n is the number of lanes. A for loop will be used to construct new splines, parallel to the original one, that divide longitudinally the road. This splines will be used to draw the lane markings and as paths for the vehicles to follow. An example is given in figure 15. This is done by means of a programmed function *AddParallelSplines*.

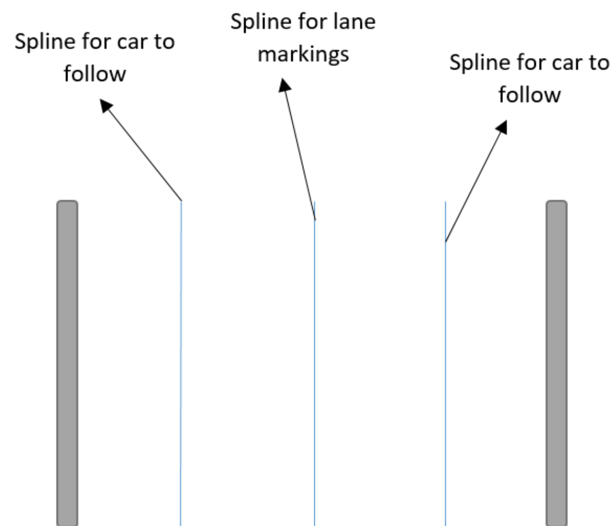

Figure 15: Splines generated for a two lane road.

Although lane markings are not meant to be used by the vehicles in simulation, being able to see them may help the user detect errors in the control of the vehicles, as it sets a point of reference from which the user may be able to locate the vehicle within the lane. *Decals* are an Unreal Engine material that can be projected onto meshes, such as the road mesh. This asset is a simple way to draw lane markings in the generate road, with the benefit of being rapidly interchanged and moved as necessary. Another option would be to define static meshes for each option (one lane, two lanes, three lanes...), but this would mean unnecessary work, plus it would be impossible to have all the options needed to satisfy the user's desires for road width, number of lanes, etc. Figure 16 shows the final outcome of the *RoadGenerator*.

Figure 16: Example of generated road.

3.2 Vehicle model

3.2.1 Sedan model

The vehicle models used in this simulation are derived from the *Sedan* vehicle blueprint offered by Unreal Engine. The Sedan vehicle class inherits from the *Wheeled Vehicle* class. In addition to its parent's functionalities, it includes handbrake, throttle and steering input, as well as gear changes. It allows the player to control everything by using keyboard inputs.

To define the model, the Sedan class uses a Skeletal Mesh (figure 17), a physics asset to handle collisions and other physical lays, an animation blueprint, a vehicle blueprint and tire blueprints.

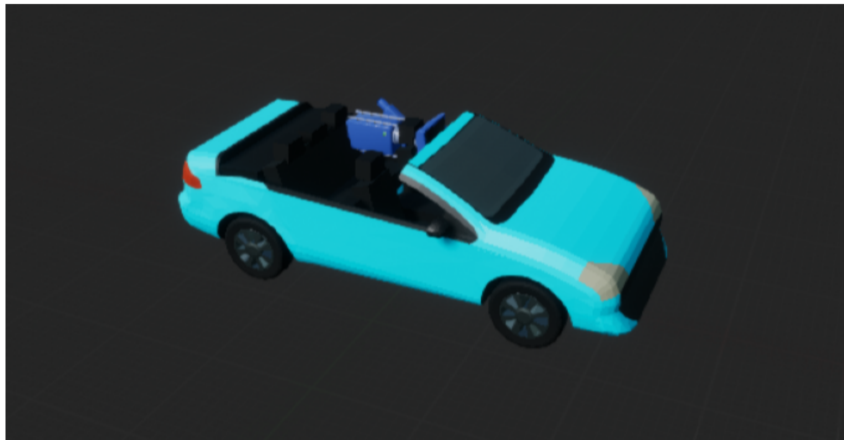

Figure 17: Sedan vehicle model.

3.2.2 Vehicle parameters

In this section, the most important parameters of the vehicle model will be briefly commented. These parameters can be changed at any moment by the user, therefor, as long as the user has detailed knowledge of a real car model, he can reproduce its behaviour on the engine.

Related to the vehicle:

- Mass of 1500 kg.

- Chassis width of 180 cm and height of 140cm.
- The torque curve outputs maximum power (500 NM) at 1900 rpm.
- The maximum RPM is 5800.
- Engine damping rate when at full throttle of 0.15.
- Damping rate when clutch is engaged of 2.
- Damping rate when clutch is disengaged (neutral gear) of 0.35.
- Steering curve that ensures that at top speed the maximum steering is 70%.
- Vehicle's chassis drag coefficient with air of 0.3.
- Gear changes happen automatically at predefined intervals.
- Gears 1, 2, 3, 4 and 5 multiply engine torque by 4, 2, 1.5, 1.1 and 1, respectively.

Regarding the wheels:

- Uses Ackerman model for steering.
- Wheel radius of 35cm and width of 10cm.
- Wheel mass of 20Kg.
- A damping rate of 0.25.
- Max brake torque of 1500NM
- Max handbrake torque of 3000NM, in case they are defined as "Affected by HandBrake".

The car model can accelerate up to 100 Km per hour in approximately 10 seconds. This value could be decreased if the intervals at which the gears change are edited. The maximum speed with these parameters is 173 Km/h.

In order to test the engine as a traffic simulator, the Sedan class is not valid, as it required the user to input the throttle and steering commands through the keyboard. Therefor, new classes that are capable of self driving are created.

3.2.3 Flocking

In order to make the vehicles autonomous, a flocking algorithm is implemented. Flocking algorithms simulate the behaviour of birds in real life. A study of the behaviours that define the movement of individual birds in flocks and their application into computer software was made by Craig Reynolds in [30] and [31].

Craig Reynolds defines a certain number of steering behaviours that can be combined for autonomous characters, the most important of which are listed below:

- **Seek**: This behaviour guides the character to a certain position in the world.
- **Flee** is the inverse behaviour. It will guide the actor away from a certain position.
- **Pursuit** and **evade** act the same way as seek and flee, but considering that the position of the point changes in time, so the point has a velocity vector.
- **Obstacle avoidance** takes place when the character is near to collision with other actor. It will act to steer the character away from the danger.
- **Path following** behavior steers the character in order to navigate along a predetermined path.
- **Flow field following** directs the motion of the character according to it's position in the world. The character steers in order to align itself with the local tangent of a flow field.

There are three extra behaviours that affect characters only when in a group, and that almost define completely by themselves the behaviour of a flock. These are:

- **Separation** gives a character the ability to maintain a certain distance between itself and others.
- **Cohesion** behaviour ensures the flock stability. It acts as inverse of the separation behaviour.
- The **alignment** behaviour aligns the character with other characters in a local neighbourhood, ensuring the global direction of the group is the same.

3.2.4 FlockSedan class

The *FlockSedan* class is a child of the Sedan class. It overrides the basic control functions of its parent class and implements the flocking algorithms discussed previously.

A vehicle in a road environment does not have need of all of the mentioned behaviours, as some of them are not in consonance with traffic regulations. For example, vehicles have not need to maintain a certain cohesion. On the contrary, the more separation between them the better. These are the behaviours that this blueprint implements:

- Path following.
- Flow field following.
- Collision avoidance and separation.

In order to implement these behaviours, a new set of variables and properties are included. The tick event and associated functions of it´s parent is also expanded to include a new sequence of functions. These functions, or behaviours, will output a velocity vector for each desired behaviour. For example, the Road Following behaviour will output a vector that leads the vehicle directly to the center of the road. The final velocity vector will be a weighted sum of these individual velocity vectors.

An instance of the FlockSedan class will require the user or another controller blueprint to reference the track or road that the vehicle must follow. There are also variables to store the speed limits of the road, as well as the detection range of the sensors on board. As the control of the throttle and steering of the car will involve PIDs, the user can also select the PID parameters of both throttle and steering.

The first of the new functions is related to matching the speed limits of the road. These speed limits are stored as a variable of the blueprint named *Speed*. In order to obtain the velocity vector for speed matching, the closest road spline point to the vehicle is obtained, then the direction of the spline on that point is obtained and multiplied by the speed (figure 18).

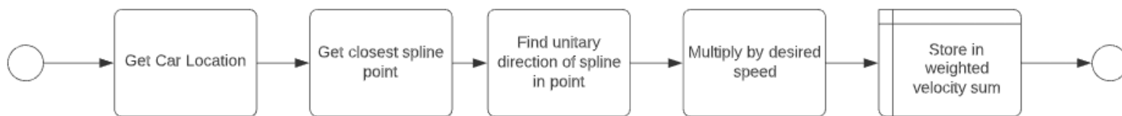

Figure 18: Speed matching flowchart.

The Road Following behaviour works in a similar way. The function searches for the vector that leads from the vehicle to the closest spline point, which is the center of the road. This vector is then weighted and added to the sum of vectors. The vector on itself is a measure of distance, so the importance of it in the sum of weighted vectors will increase along with the distance between the vehicle and the spline and center of the road (figure 19).

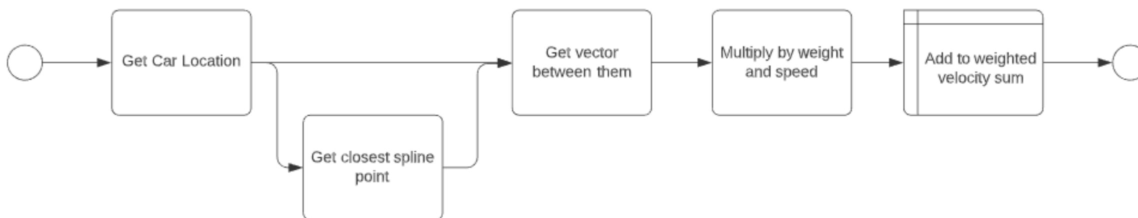

Figure 19: Road following flowchart.

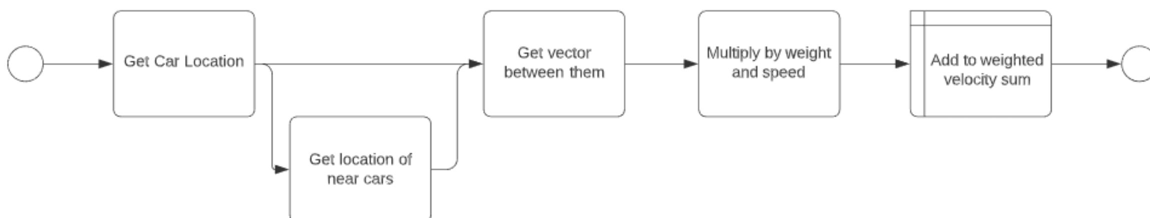

Figure 20: Collision avoidance flowchart.

Collision avoidance (figure 20) function will output a vector that leads the vehicle away from

cars that are within a user-defined range. The resulting vector of this function will be bigger the closer the car is to colliding with a neighbour.These vectors can be seen in figure 21.

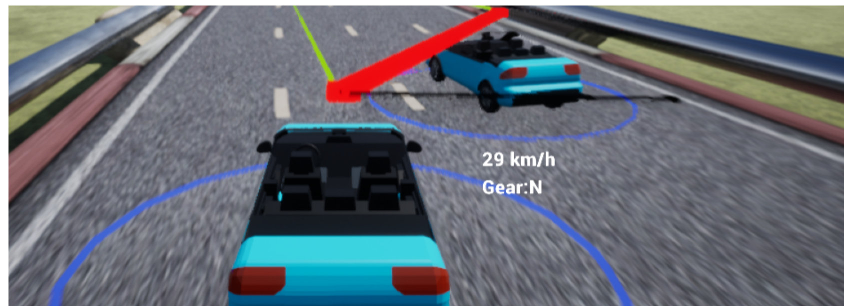

Figure 21: Vectors generated by the flocking functions. The red arrow indicated the direction to a close neighbour, the black arrow indicated direction towards the lane center and the yellow arrow indicated the road direction.

The final resulting vector is the used to obtain the the speed (magnitude) and the direction at which the vehicle must move. These references are fed to a PID block that will output the necessary control action for both throttle and steering (figure 22). Both output control actions are then clamped, as they maximum and minimum values should be 1 and -1 respectively. In order to control the vehicle correctly, additional steps must be performed when computing the steer control action, or else problems may arise when the vehicles heading is $-175°$ and the desired direction is $175°$. The vehicle, instead of turning slightly the remaining $10°$ may attempt to go from one direction to the other by passing through $0°$.

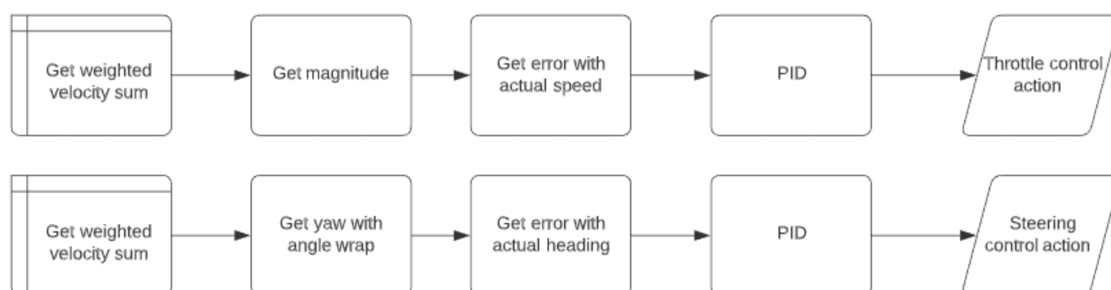

Figure 22: Throttle and steering control flowchart.

Although this vehicle control algorithm allows it to navigate through a track at the desired speed and direction, and avoiding collisions, it is not ideal. The flocking collision avoidance vector may cause the vehicle to invade other lanes when avoiding a collision, which in turn may cause other vehicles to crash.It also handles all near vehicles equally. Generally, in a real world environment, vehicles can get very close without danger as long as they are in different lanes. The distance at which a close vehicle is a sign of danger depends of the lane the other car occupies.

A vehicle (an instance) of this class can be created by the user by just dragging and dropping the asset into the world. In order for it to function, the following parameters must be set by

the user (figure 23). All these parameters can be edited at both construction and runtime. For example, the speed of the vehicle will change when the speed limits of the road change.

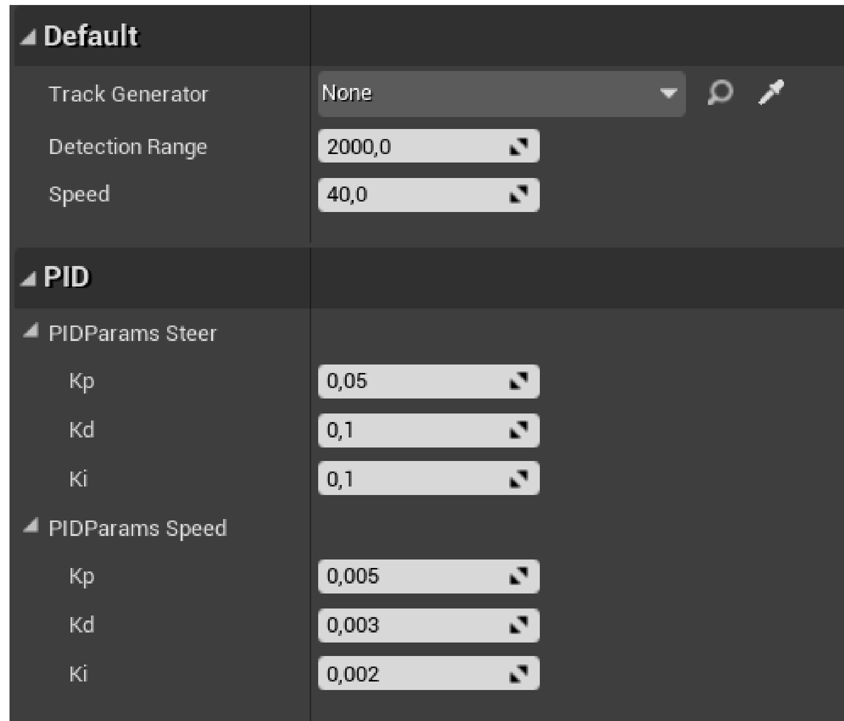

Figure 23: Parameters needed to the vehicle control to function.

- The track generator field is where the user defines which road the vehicle is following. When placing an object into the world, a name is automatically given to it. Selecting the name of the road in this field will allow the vehicle blueprint to obtain the road data needed for it to drive. Instead of selecting a name from a list, the colour picker symbol right gives the user the ability to directly click on the desired road in other to select it.
- The detection range acts as the radar of the vehicle. Vehicles within this range will be considered when building the vector to avoid collisions. The units are in cm.
- The speed is measured in km/h. It represents the speed at which the vehicle will try to stay. This speed can be surpassed in case of collision avoidance.
- PID parameters for steering and throttling can be edited here.

3.2.5 FlockSedanLanes class

The *FlockSedanLanes* is a child class from the FlockSedan which adds functionalities in order to improve driving and analysis.

The avoid collision function is overridden. Now, there are two avoiding behaviours, based on the situation of the other vehicle in the same lane or in a different one. The *avoid collision* vector is now projected into the spline direction (figure 24). That way, the vehicle may accelerate or brake in presence of danger, but always following the lane direction. It will not invade other lanes while trying to avoid a collision.

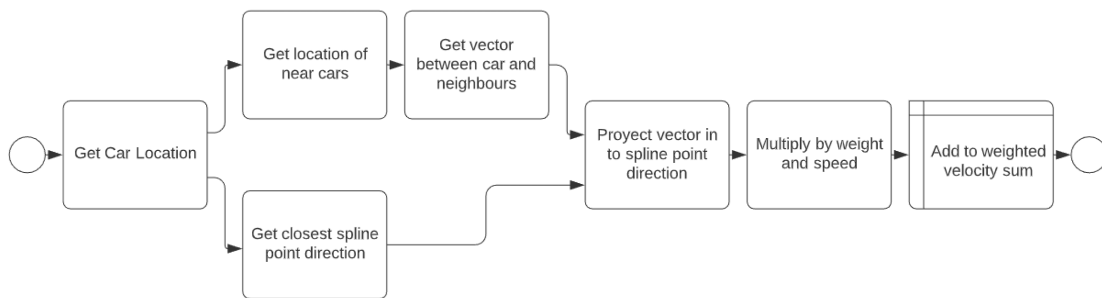

Figure 24: Collision avoidance taking lanes into account. This function is executed twice with different detection ranges, once for vehicles in the same lane, and once for adjacent vehicles.

This class also holds a variable that defines the road lane in which the vehicle stands. As mentioned in section 3.1, the road uses splines to define the lane center. These splines are stored in an array and the lane selection variable selects which row of the array (which spline) the vehicle adheres to.This variable can be changed in runtime to make the vehicle change lanes.

There are two more additions in this class. The first one is a function that will trigger on a 0.5 second loop. This function should be known to the user, as it is the function in charge of sending vehicle information to the game controller. Using this function, updates of the vehicle speed, throttle, etc. can be obtained in order to obtain information from the traffic system. This information can be then showed in a *Heads Up Display* or even printed to a text file in Windows in order to import it into analysis software.

The other addition is the *EventHit*. This event fires when the vehicle collides with something, and allows the user to define what behaviour should the vehicle have in that situation. The predefined behaviour involves checking if the object the vehicle has collided with is another vehicle or the road guardrails. In case the latter is true, the vehicle shall be removed from the world and a collision count global variable shall be updated. This function is important, as not removing the vehicles may cause collateral crashes and even a roadblock (figure 25). This crashes are not directly caused by the controller, so they are not in the scope of this study.

Figure 25: Roadblock generated by a single initial vehicle collision.

When instancing a vehicle of this class, in addition to the parameters required for it's parent, the user must define which lane the vehicle is going to drive on. The lane selection variable can also change at any moment in runtime.

3.2.6 HumanSedanLanes class

In order to have two different types of vehicles, another child of the FlockSedan class is created. This class emulates human behaviour when driving. In particular, human drivers tend to have a more aggressive behaviour when driving vehicles, and accelerate and brake harder. The reaction time is also slower and the attention span narrower.

This characteristics have been embedded in the class. In order to do so, the detection range the vehicle has been shortened, so that it reacts later to close neighbours. Also, the torque curve of this vehicle has been augmented for lower RPMs, so that it will accelerate more abruptly. While autonomous cars will try to drive exactly at the same speed the speed limit states, human drivers tend to drive at speeds a few km/h higher or lower. In order to simulate this behaviour, human driver-like vehicles are randomly given a speed that is $\pm 10\%$ of the speed limit.

Figure 26: Second type of vehicle. The color has been changed to red so that the vehicles are clearly identifiable.

3.3 Vehicle spawner and destroyer. Exporting vehicle data.

The *VehicleSpawner* and *VehicleDestroyer* classes are created so that the user can define one point at which vehicles will appear in the world and one point at which they will disappear.

If the vehicle spawner is set as active, it will create vehicles at its location every T seconds, where T is a random number in the range defined by the user. The spawner can create the two types of vehicles developed in sections 3.2.5 and 3.2.6. The probability with which it will span one or the other can be increased or decreased by the user at runtime, as well as the vehicle creation time interval. This object also sends data to a global variable referring to the number of vehicles that are being created per second.

When placing a vehicle spawner on the world, the following parameters must be set by the user:

- Spawn time min and max: Defines the interval at which vehicles are created.
- Track and lane: The vehicles created with this spawner will adhere to the track and lane specified here.
- Car speed: The initial and reference speed of the spawned cars.
- The probability of each car type to be spawned.

- Whether it is active (spawns cars) or not. This parameter can be edited by the user in case he wants cars to stop being created but does not want to remove the spawner.

Figure 27: Model of car spawner.

The *VehicleDestroyer* class has the same graphic model as the spawner, but its behaviour is different. When a vehicle overlaps with this objects model, it is removed from the world.

In order to extract vehicle data from Unreal Engine, an *AI Controller Class* is used. AI Controller Class is a special type of actor that can be used to control a Pawn class and implement pseudo-intelligent behaviours in it. In this application, this class is empty, as all the control is made on the vehicle blueprint. Instead, this class is used to compile data from the vehicle, such as speed, the control actions, etc., and store it.

3.4 Triggers

Triggers are a special type of actor in Unreal Engine. When a certain actor interacts with them, they cause an event to occur. In this framework, triggers are used points in a road where the speed limits changes.

A trigger can be dragged and dropped in the world the same way as any other object. In this case, it is dropped on the point of the road where we desire the speed limit to change.

The trigger will cause an event *OnActorBeginOverlap* to fire when any object enters it´s limits. The event function returns as output the reference of the actor that has overlapped it. The function will check if that actor is a vehicle by trying to cast it to one of the vehicle classes. In case it is, it will access it's parameters to change its reference speed.

These trigger objects are very simple to implement into the simulation and have many other uses besides changing the speed limits of the vehicles. They can be used to simulate areas of wet pavement by changing the friction factors of the wheels of each car, or the presence of traffic lights and crosswalks, etc.

3.5 Level Blueprint and HUD

Once all the needed assets have been created, there is a need for some type of blueprint that can control and monitor the simulation itself. This blueprint is called the *Level Blueprint*.

A level blueprint is a special type of blueprint that acts as a global event graph. There is only one per level. This blueprint can be used to fire off sequences of actions based on level events

or specific actor calls.

In this case, the level blueprint is needed to monitor the number of collisions, the mean speed of vehicles across the road and the number of vehicles entering the road. The level blueprint does this in conjunction with a *Game Instance*, which is the way that Unreal Engine handles global variables. It is also in charge of global keyboard events, such as changing level, or pressing the Esc key in order to abandon the simulation. Pressing the Esc key will also trigger an event which will gather all the stored vehicle information and export it to a text file. More on this function can be read at Appendix A.

The HUD allows the user to see the most important data from the world at runtime in screen. It is a graphic overlay. In this case, the HUD is created to show the user the available commands, such as increasing and decreasing the frequency of created vehicles, and the ratio of created vehicles. It also shows the user the mean speed of all vehicles in the road in order to identify traffic flow variations and the number of collisions. The developed HUD is shown in figure 28.

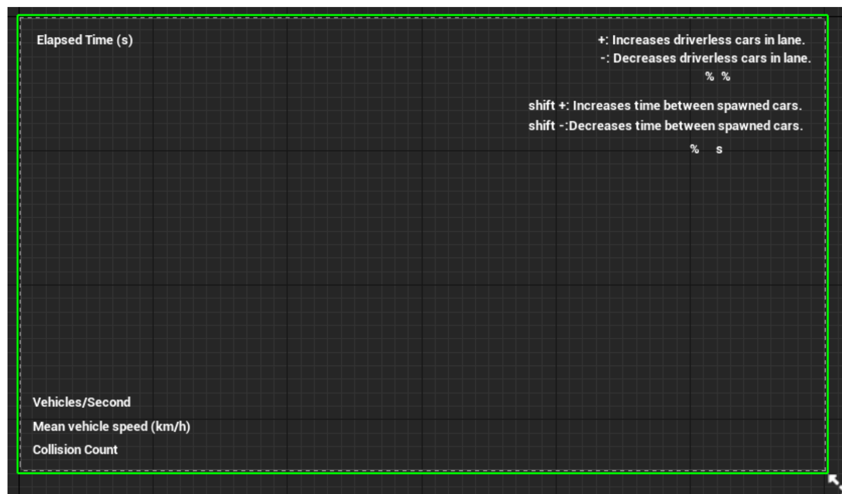

Figure 28: HUD display.

4 Simulation results and analysis

4.1 Scenarios

In order to test the simulation framework, several scenarios have been designed. Each scenario assigned to a level in Unreal Engine. A level is a collection of meshes, actors and other components in a world. It serves as a way to organize components and relieves pressure on the performance of the system, as only one level is simulated at a given time.

4.1.1 Road with dense traffic

The first level consists on a four lane road with dense traffic (four vehicles per second). Figure 29 shows the chosen road model. When simulating with only FlockSedanLanes vehicles, with a speed limit of 60km/h, there are no collisions. The evolution of the speed of each vehicle can be seen in figure 30, where each line represents a vehicle. Sample time T is 0.5 seconds. There is a slight downwards slope starting in the sixth sample, caused by the road curve and the adaptation to the rest of the environment. Nevertheless, the vehicles adapt quickly and the speed of 60 km/h is maintained perfectly throughout the rest of the road.

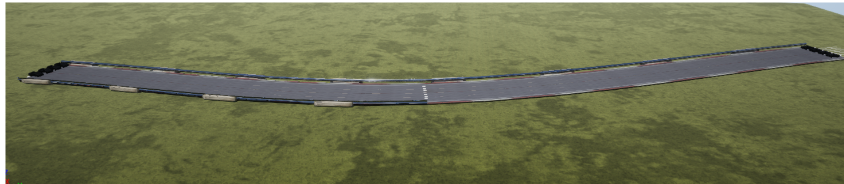

Figure 29: Test road.

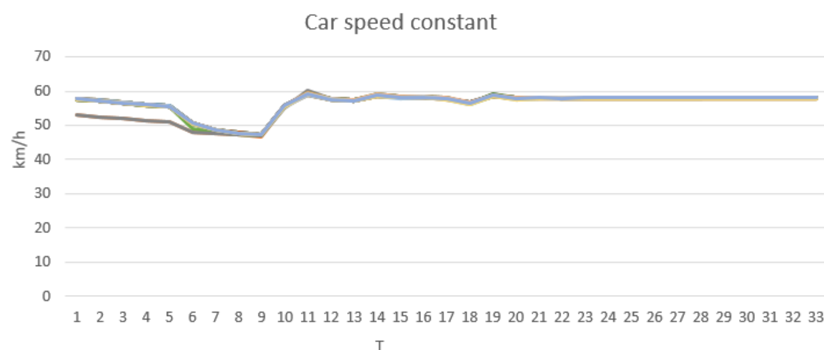

Figure 30: Speed history for each vehicle when only one type of vehicle present. T = 0.5s.

When introducing the second type of vehicle into the road, the human-like behaviour vehicle, we can observe in figure 31 that the speed of the vehicles is not as constant, there is more variation. The aggressive behaviour when both accelerating and braking leads to occasions where some vehicles need to reduce their velocity in order to avoid collisions. The mean speed of the traffic flow is then reduced.

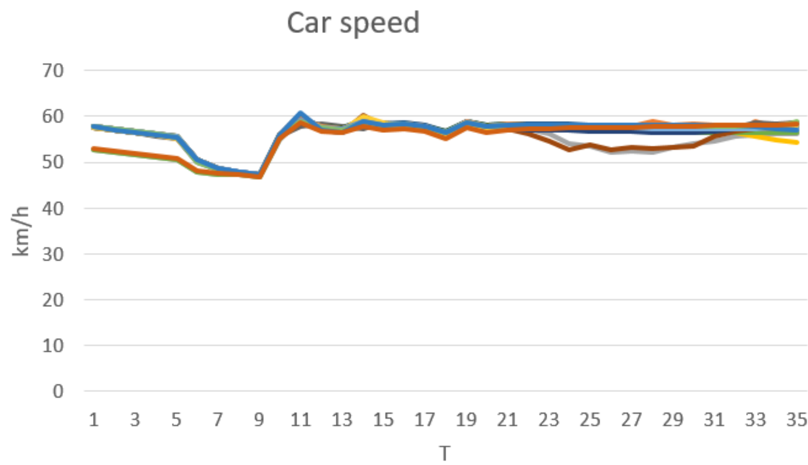

Figure 31: Speed history for each vehicle when the two types of cars are present. T = 0.5s

4.1.2 Road with dense traffic and speed limit change

In the second test, a speed limit variation is applied, allowing the vehicles to increase their velocity up to 100 km/h. The traffic density is maintained. Figure 32 shows the speed history of the vehicles when only one type of vehicle is present. It is mainly constant except at the point where the speed limit changes. Some of the vehicles are unable, due to collision danger, to speed up until a few seconds later in the simulation.

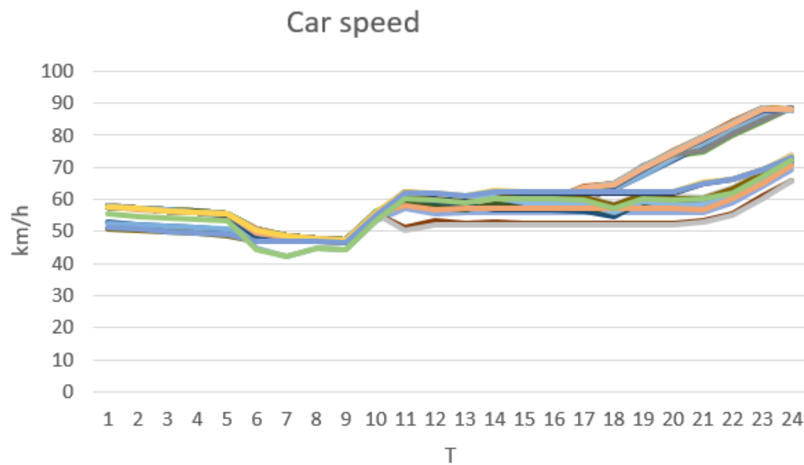

Figure 32: Speed history for each vehicle when only one type of vehicle present and there is a speed limit variation. T = 0.5s

When human driver-like vehicles are added in a ratio 1:1, the speed evolution is more chaotic. The acceleration and braking of the human-like vehicles is inconsistent, and leads to collisions and emergency braking. Despite the speed limit changing, not many vehicles are able to increase their velocity, some even have to slow down to velocities under 10 km/h.

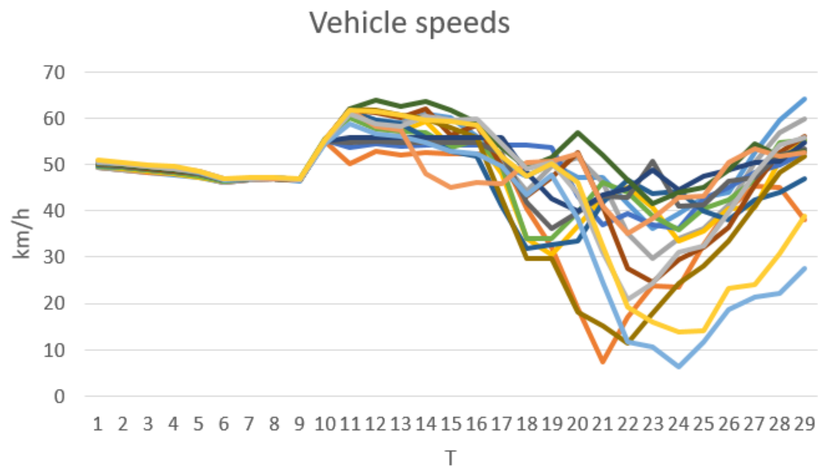

Figure 33: Speed history for each vehicle when the two types of cars are present and there is a speed limit variation. T = 0.5s

4.1.3 Closed loop road

The objective of this experiment is to simulate and reproduce the results obtained in [32]. The cited paper, written by the Mathematical Society of traffic flow in Japan, develops an experiment where 22 vehicles are positioned in a circular road of 230 meters. The vehicles are to drive at 30 km/h. The experiment shows that fluctuations appear and even stop the free flow of vehicles, forcing some cars to stop, in some time. There is a *"stop-and-go wave"* that propagates in the direction opposite of traffic flow.

In order to reproduce this test, a closed loop road is created in Unreal Engine, placing 50 vehicles of FlockSedanLanes class and setting the speed limit to 40 km/h.

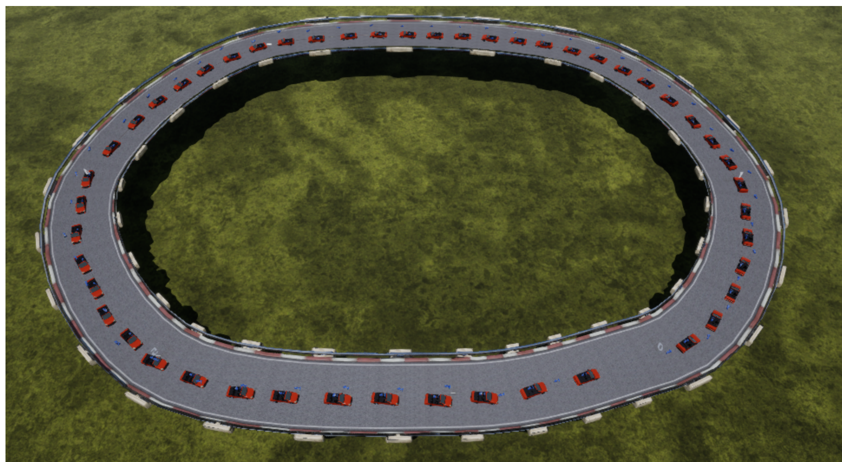

Figure 34: Road model.

Although reproducing the "stop-and-go" wave has not been possible, there are generated waves that move in the opposite direction to the traffic and force the vehicles to slow down at speeds around 5-10 km/h. These waves do not persist for long and tend to dissipate. Nevertheless,

there is a clear difference when using autonomous vehicles and human driven vehicles. Figures 35 and 36 show the speed for each type of vehicle. The mean speeds are 35.4 and 19.5 km/h. respectively, which is a great variation.

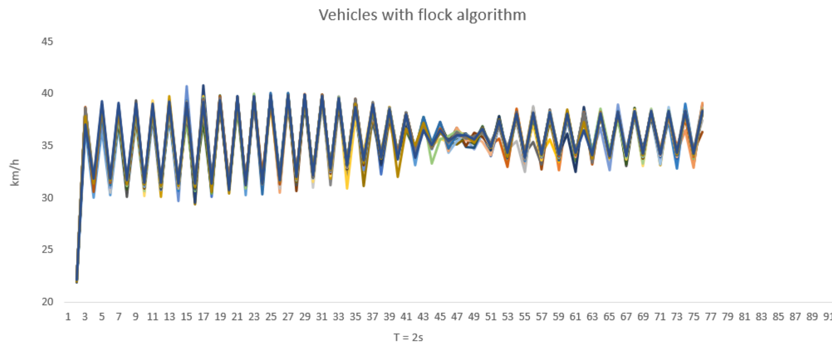

Figure 35: Speed history of flock vehicles sampled every 2 seconds.

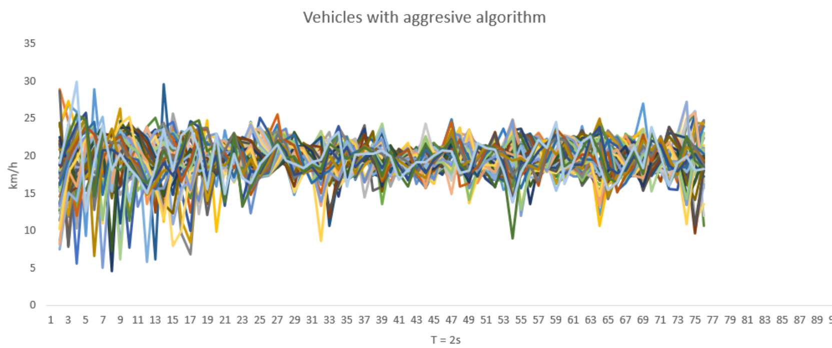

Figure 36: Speed history of human driven vehicles sampled every 2 seconds.

5 Environmental impact and budget

5.1 Environmental impact

As the realisation of this project has only required a computer and a software application (Unreal Engine), there is little negative environmental impact.

However, the use of traffic simulators such as the one developed and proposed during this project contributes greatly to reducing the amount of traffic in roads of every type, and therefore is a important factor in reducing contamination generated by vehicles, such as greenhouse gasses, noise, etc.

5.2 Budget

This project has been developed within the framework of a Master Thesis for which 12 ECTS of the whole Master degree are destined. As 1 ECTS is equivalent to 30 hours of work, we could approximate that the number of hours invested are 360h. Assigning these hours to a software engineer with a salary base of 30€/hour would result in the total cost shown below.

$$360h * 30\text{ €}/h = 10800\text{ €} \tag{3}$$

This would be the total cost of the project. The software, Unreal Engine, does not generate cost, as it is free for use unless a revenue of 3000$ per quarter is generated.

Conclusions

Having performed all the simulations mentioned above, we can conclude that the initial objectives of this Masters Thesis have been met.

The resulting Unreal Engine-based traffic simulation framework can be used to test new control algorithms for autonomous vehicles in a precise simulation of a traffic environment and the effect that they have on other neighboring vehicles. In this sense, the point of view of this application is in a middle-point between the traffic simulators mentioned in section 2.3: it is not as microscopic as the TORCS/MADRaS simulator, which is focused on individual vehicle behaviours and control, and it is not as macroscopic as SUMO, where the user can define city-wide road networks, but can´t vehicle properties in order to implement new control algorithms.

One of the advantages of this simulator is that it is embedded in the Unreal Engine editor. The Blueprint Visual Scripting language and extensive documentation allows user with little knowledge about programming to implement and test easily new prototype control algorithms for vehicles. The only thing a user needs is to compute a speed, a direction and place the car in the world.

As a final note, an Open-source project is never really finished. Future work could include implementing user-friendly functions that guide the vehicle from one road environment to other, for example, guiding a vehicle from a road to a roundabout.

References

[1] Instituto para la Diversificación y Ahorro de Energía Dirección General de Tráfico. La movilidad al trabajo: Un reto pendiente., 2019.

[2] Bernhard Friedrich. The effect of autonomous vehicles on traffic. In *Autonomous Driving*, pages 317–334. Springer, 2016.

[3] G Kotusevski and KA Hawick. A review of traffic simulation software. 2009.

[4] Benjamin Coifman and Lizhe Li. A critical evaluation of the next generation simulation (ngsim) vehicle trajectory dataset. *Transportation Research Part B: Methodological*, 105:362–377, 2017.

[5] Sumo project webpage. https://sumo.dlr.de/docs/index.html.

[6] Torcs project webpage. http://torcs.sourceforge.net/.

[7] Madras project webpage. https://github.com/madras-simulator/madras/wiki/.

[8] Guankun Su, Nan Li, Yildiray Yildiz, Anouck Girard, and Ilya Kolmanovsky. A traffic simulation model with interactive drivers and high-fidelity car dynamics. *IFAC-PapersOnLine*, 51(34):384–389, 2019.

[9] Autonomous driving simulation framework for traffic studies matlab package. https://github.com/nebnebgnahz/mathworks_autonomous_driving_project.

[10] Evan gravelle traffic simulator matlab package. https://github.com/evangravelle/traffic-simulator.

[11] Unreal engine driving scenario simulation matlab package. https://www.mathworks.com/help/driving/unreal-engine-driving-scenario-simulation.html.

[12] Stefano Carpin, Mike Lewis, Jijun Wang, Stephen Balakirsky, and Chris Scrapper. Usarsim: a robot simulator for research and education. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 1400–1405. IEEE, 2007.

[13] Hirofumi Seo and Takeo Igarashi. Enhancement techniques for human anatomy visualization. In *SIGGRAPH Asia 2018 Posters*, pages 1–2. 2018.

[14] Hirofumi Seo, Naoyuki Shono, Taichi Kin, and Takeo Igarashi. Real-time virtual brain aneurysm clipping surgery. In *Proceedings of the 24th ACM Symposium on Virtual Reality Software and Technology*, pages 1–2, 2018.

[15] Phongsak Prasithsangaree, Joseph M Manojlovich, Jinlin Chen, and Michael Lewis. Utsaf: a simulation bridge between onesaf and the unreal game engine. In *SMC'03 Conference Proceedings. 2003 IEEE International Conference on Systems, Man and Cybernetics. Conference Theme-System Security and Assurance (Cat. No. 03CH37483)*, volume 2, pages 1333–1338. IEEE, 2003.

[16] ADRIAN RG Harwood, Petra Wenisch, and ALISTAIR J Revell. A real-time modelling and simulation platform for virtual engineering design and analysis. In *Proceedings of 6th European Conference on Computational Mechanics (ECCM 6) and 7th European Conference on Computational Fluid Dynamics (ECFD 7), 11–15 June 2018, Glasgow, UK*. ECCOMAS, 2018.

[17] Jaroslav Matej. Virual reality and vehicle dynamics in unreal engine enviroment. *MM Science Journal*, 2016.

[18] Marc Tschentscher, Ben Pruß, and Daniela Horn. A simulated car-park environment for the evaluation of video-based on-site parking guidance systems. In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 1571–1576. IEEE, 2017.

[19] Jérôme Leudet, François Christophe, Tommi Mikkonen, and Tomi Männistö. Ailivesim: An extensible virtual environment for training autonomous vehicles. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 479–488. IEEE, 2019.

[20] Unreal engine documentation. https://docs.unrealengine.com/en-us/index.html.

[21] Unreal engine wiki. https://www.ue4community.wiki/unreal_engine_community_wiki.

[22] Vehiclesim dynamics asset. https://www.unrealengine.com/marketplace/en-us/product/carsim-vehicle-dynamics.

[23] Microsoft airsim asset. https://github.com/microsoft/airsim.

[24] Sumo2unreal project webpage. https://github.com/augmenteddesignlab/sumo2unreal.

[25] Anderson Maciel, Tansel Halic, Zhonghua Lu, Luciana P Nedel, and Suvranu De. Using the physx engine for physics-based virtual surgery with force feedback. *The International Journal of Medical Robotics and Computer Assisted Surgery*, 5(3):341–353, 2009.

[26] Leon Sze-Ho Chan and Kup-Sze Choi. Integrating physx and openhaptics: Efficient force feedback generation using physics engine and haptic devices. In *2009 Joint Conferences on Pervasive Computing (JCPC)*, pages 853–858. IEEE, 2009.

[27] Kyrre Glette and Mats Hovin. Evolution of artificial muscle-based robotic locomotion in physx. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1114–1119. IEEE, 2010.

[28] Michael Blundell and Damian Harty. *Multibody systems approach to vehicle dynamics*. Elsevier, 2004.

[29] Egbert Bakker, Hans B Pacejka, and Lars Lidner. A new tire model with an application in vehicle dynamics studies. *SAE transactions*, pages 101–113, 1989.

[30] Craig W Reynolds. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34, 1987.

[31] Craig W Reynolds. Steering behaviors for autonomous characters. In *Game developers*

conference, volume 1999, pages 763–782. Citeseer, 1999.

[32] Yuki Sugiyama, Minoru Fukui, Macoto Kikuchi, Katsuya Hasebe, Akihiro Nakayama, Katsuhiro Nishinari, Shin-ichi Tadaki, and Satoshi Yukawa. Traffic jams without bottlenecks—experimental evidence for the physical mechanism of the formation of a jam. *New journal of physics*, 10(3):033001, 2008.

Appendix A

SaveTxt function

Unreal Engine has not implemented a function capable of exporting in-game data during runtime into a text file. However, due to it´s code being open, it is possible to create such function in C++ using Visual Studio. Moreover, the function can be turned into a blueprint function, so that users can use it when programming with Blueprint Visual Scripting, and do not need to program in C++ or even have Visual Studio Installed.

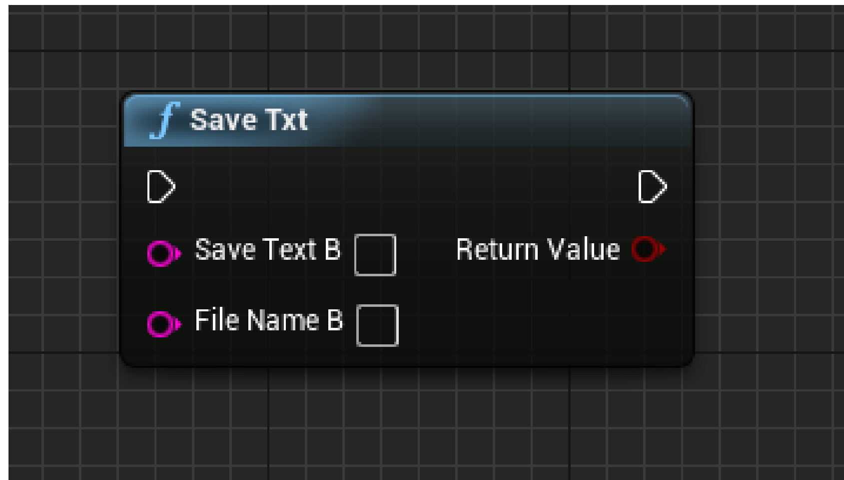

Figure 37: Save Text node

When the function is called, it will export the string stored in *SaveTextB* into a text file named *FileNameB* and store it in the project folder. In this case in particular, there will a generated text file for each vehicle that finished it´s route, called "Speeds.txt".

H file:

```
UCLASS()
class FLOCKING_API URWTxtFile : public UBlueprintFunctionLibrary
{
 GENERATED_BODY() public:
 UFUNCTION(BlueprintPure, Category="Custom",
     meta=(Keywords="LoadTxt"))

   static bool LoadTxt(FString FileNameA, FString& SaveTextA);

 UFUNCTION(BlueprintCallable, Category="Custom",
     meta=(Keywords="SaveTxt"))

   static bool SaveTxt(FString SaveTextB, FString FileNameB);
};
```

Cpp file:

```
#include "RWTxtFile.h"
bool URWTxtFile::LoadTxt(FString FileNameA, FString& SaveTextA)
{
 return FFileHelper::LoadFileToString(SaveTextA,
     *(FPaths::GameDir()+FileNameA));
}

bool URWTxtFile::SaveTxt(FString SaveTextB, FString FileNameB)
{
 return FFileHelper::SaveStringToFile(SaveTextB,
     *(FPaths::GameDir()+FileNameB));
}
```