Testing vehicle safety in simulated driving scenarios

Final Project Report

Author: Vakaris Paulavičius

Supervisor: Professor Mohammad Reza Mousavi

Student ID: 20062023

April 5, 2023

Abstract

The vision of a civilisation where driverless cars seamlessly navigate the lively streets and high-speed underground tunnels, transporting passengers from point A to point B, is remarkably portrayed in many futuristic movies and science fiction novels. However, this futuristic dream is not as far-off as it may seem. In fact, many vehicles operating the streets today already incorporate some level of automation, such as cruise control and automatic parking. While it is clear that we are making progress towards fully autonomous vehicles, the question remains: when will we finally witness entirely driverless cars on the roads? Although autonomous vehicles already hold the potential to surpass human drivers in terms of safety, there is currently no practical way to test this claim. Or is there? This paper proposes an innovative approach - evaluating high-level vehicle safety through simulations on virtual roads in an automated and time-efficient manner. The proposed software consists of three main components: an automated scenario generation tool that, with the help of a sophisticated machine learning algorithm, can generate diverse driving scenarios, a vehicle safety monitoring system designed for use in the simulator CARLA and data analysis tools allowing for efficient and visualised analysis of obtained data through diagrams and location marking on 2D map representations. In addition to introducing the automated safety testing software bundle, this article also dives deeper into the realm of AVs by discussing the technology, safety vulnerabilities, how AVs are classified and the comparison of human drivers to computer-controlled cars.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Vakaris Paulavičius

April 5, 2023

Abbreviations

Here is a list of abbreviations that are used throughout the report. It is assumed that not every person reading this article is fluent in computer scientists' terminology and thus, having a guide might come in handy.

AI artificial intelligence

AV autonomous/automated vehicle

ML machine learning

XAI explainable artificial intelligence

ISO International Organization for Standardization

NHTSA National Highway Traffic Safety Administration

Contents

Chapter 1

Introduction

This research paper aims to present a methodology for evaluating how well modern AV implementations can participate in realistic traffic scenarios and get from one location to another safely.

The fundamental obstacle that science and major car manufacturers need to overcome to set up the widespread recognition of autonomous vehicles is establishing trust in the AVs and assuring that they can be a safer and more convenient means of transportation. This is is well presented in the available literature considering autonomous vehicles [1].

However, according to the 2022 survey carried out by Policygenius in the U.S. [2], 76% of respondents said that they would feel less safe driving in cars with self-driving features, while a similar proportion (73%) said they would feel less safe knowing others are driving on the road in autonomous vehicles. This shows that although people have been restless for decades trying to establish general acceptance of technology that will massively alter how we see and use transportation today, they have not succeeded yet.

One thing that could change this perception is autonomous vehicles appearing on the streets and demonstrating that statistically, they can be better vehicle operators than human drivers. However, the law that we live by today does not allow autonomous vehicles to navigate the streets freely because there is a lack of standards and regulations for it, and it is not defined who has to be responsible if a machine makes a mistake. The recently emerged branch of AI, the XAI, promises to take this a step forward, providing the machine with the ability to explain its actions. This is particularly relevant in situations where an AV gets involved in an accident, and it is crucial to understand the circumstances that led to this and what were the vehicle operators doing to prevent this or at least minimise the impact.

An alternative that is available today is demonstrating the AI potential in a virtual, simulated world where there is no law preventing autonomous vehicles from driving on the streets and no living being able to get hurt. This article focuses on creating a framework that would allow testing the AV implementations in such a virtual setting and analysing the observations, thus determining whether the AI agent is acting safely and, if not, where the issues lie.

We start by exploring the concept of autonomous vehicles, how safe they could be and what vulnerabilities they might have if they are finally put on the streets. In Chapter 3, we continue our journey by analysing the road statistics in Europe and the U.S. from 2021 until 2022 and discussing how the scenario generation algorithm could be designed based on the observations. Subsections of the chapter mentioned above also analyse human behaviour in driving situations and discuss the relevance of statistics to this project. In Chapter 4, we discuss the chosen simulator CARLA, its advantages and shortcomings, and explain its role in the research while also covering details of how different AV implementations could be used with the simulator. In Chapter 5, we finally arrive at the scenario generation tool, one of the three main framework components. This part covers the tool's development process and inner workings while also emphasising the importance of road statistics and traffic data for its development. The following Chapter 6, dives into detail about how safety can be measured in the simulation environment, what tools are needed for that and describes in detail the safety measuring system this article is proposing that can determine how safe the vehicles are performing in the simulated scenarios. This section also provides an overview of what safety metrics are evaluated and proposes a formula to assess the performance. In Chapter 7, we introduce the data analysis tool, the last of the three main software components. We talk in detail about how it can be used to both analyse the performance scores of the participated agents and group interesting clusters of information obtained from the sensors and simulation recordings. After introducing the main software components, in Chapter 8 we will swiftly look at how the system functions as a whole and how all the modules work together. Chapter 9 focuses on evaluating how well the system functions by looking at the research involving human participants and a CARLA agent driving the generated scenarios. The chapter looks at the safety monitoring system in action and applies the data analysis tools to assess participant performance. It also compares how well the CARLA autonomous agent is executing the given scenarios compared to human participants.

The article ends in Chapter 10, having presented a working framework that generates diverse driving scenarios based on real-life data, allows human drivers and AI agents to drive in the

scenarios and measures their safety performance. This last chapter overviews what has been achieved and what could be improved. It discusses the relevance of this work and why the chosen subject is so impactful. In addition, it talks about the ethical questions concerning the project and the field of autonomous driving as a whole. The last section of the article provides a short self-reflection of the author about the challenges faced throughout the project's development.

Chapter 2

What is an autonomous vehicle

Before diving into the technical details that surround this project, it is crucial to make it clear what AV stands for and how the ideas of its development matured over the course of human history (Section 2.1). This chapter aims to give an overview of what makes an AV autonomous (Section 2.2), talks about the standardised way of classifying the autonomy levels in Section 2.3 and discusses the AV safety vulnerabilities in Section 2.4.

2.1 History of self-driving cars

The idea of a self-driving car surprisingly reaches back as far as the 1500s when Leonardo Da Vinci designed his self-propelled cart [3] that could move without being pushed or pulled. Some regard this as the grandfather of today's automated vehicles. Although it might not have looked like something revolutionary, it shows that people living more than 400 years before the emergence of the first computer were already looking for ways to reduce human interaction in such basic everyday activities as pushing a cart.

In 1925, American engineer Francis Houdini tested the first driverless car that was operated using a remote-control setup that guided the car using radio signals [4]. Despite the fact that the car riding Manhattan streets demonstrated what an autonomous vehicle could look like, it did bring little success to the constructor. The project was shut down when the operator lost control of the vehicle twice and eventually crashed it into another vehicle.

Although the two occasions mentioned above sound like attempts to produce an autonomous vehicle, both Da Vinci's and Houdini's creations needed human input in their movement, whether winding the clog to start it or using the remote control device to decide its move-

ments.

However, things changed forever when in 1958, almost twenty years after the initial concept was created, General Motors launched their prototype of a car that could ride the road on itself following a wire embedded into the asphalt. The concept was simple. The car contained sensors that could sense the current flowing through the wire, thus making it go according to the location of the wire and where it leads. It showed that a truly autonomous car has to be able to decide what to do by itself, relying on the data captured from the surrounding world.

This idea was reflected in the upcoming tries to create a self-driving vehicle. One includes a Japanese car built in 1977 that could slowly drive on the streets following the white street markers, and it could do so using cameras that were able to identify lane markings. In 1995, researchers at Carnegie Mellon University developed a car called NavLab5 [5] that could steer itself on roads and highways using computers and video sensors. It was taken on an almost 3000 miles tour through the U.S., where it manoeuvred itself 98% of the time, with human supervisors only needing to brake and throttle.

As highlighted by the organisation TWI Global - "Today, a fully autonomous vehicle, is considered one that can operate itself and perform necessary functions without any human intervention through the ability to sense its surroundings." [6]

One question that might arise is how exactly the vehicle would be able to know all its surroundings, determine what actions are legal and allowed, and navigate from one location to another without violating rules and causing harm to people or damaging properties. It might be about time to discuss the inside of an autonomous vehicle.

2.2 What makes an AV

At the core of an AV is a car with all the necessary parts for riding on the road. What distinguishes it from the cars that people are used to seeing is that it is equipped with multiple sensors providing all the necessary information about the world surrounding the vehicle. That data is then processed by the software responsible for all the logic and calculations that determine what commands need to be given to the semiconductors that make real-time driving actions.

Without sensors, the idea of an autonomous car would be impossible. They allow the car to see what is on the road, analyse what other road participants are doing, predict their movements and collect information needed to make future decisions regarding steering, braking or acceleration. Three essential sensors are needed for an autonomous vehicle[7], the first being

LiDAR.

LiDAR stands for "Light Detection and Ranging". It is by far the most crucial sensor in the realm of autonomous vehicles because it allows for object detection and range estimation. It emits laser beams that bounce off objects and return to the sensor allowing it to create a detailed 3D map of the environment around it, letting it know where every object is and how far it is from the vehicle. It might come as a surprise, but many modern phones are also equipped with LiDAR sensors allowing for features such as taking selfies with a blurred background, playing augmented reality games or simply measuring distance using the phone's camera. LiDAR sensors come in handy when determining the exact position, size, and shape of an object is needed. They are typically mounted on the vehicle's roof, providing a high-resolution 360-degree coverage of the vehicle's surroundings. They work well in dark conditions but have a limited range, typically a few hundred meters and lack accuracy in foggy and snowy conditions [8].

That is where radars come into help. Using radio waves, they can detect objects surrounding the car and determine how far they are and how quickly or in what direction they are moving. For this reason, many car manufacturers use radars for parking sensors. One advantage that radars have compared to LiDAR sensors is their range which can be up to several kilometres in some cases. In addition, radars are less affected by weather conditions because, unlike LiDAR sensors, they do not rely on lasers but radio waves. Because of the advantages and disadvantages of both LiDAR sensors and radars, they are often used in combination. Where one lacks precision, the other one contributes. Where one is affected by the rain and cannot function properly, the other provides reliable enough data.

Cameras are the third type of commonly used sensor. They capture high-resolution images and videos of the world surrounding the vehicle. Images can then be used to read road signs, determine the colour of the traffic light, and detect pedestrians and other road objects. Although cameras are much cheaper than LiDAR sensors, they have disadvantages in detecting stuff in low light or high contrast situations. Cameras also have a limited field of view and are less effective in detecting objects that are far away or not entirely in the camera's frame.

AVs also utilise GPS sensors that use satellite signals to determine a vehicle's location and track its movements, and IMU sensors that measure the vehicle's acceleration and orientation.

After the data is collected from the sensors, it is passed to the vehicle's onboard computers, which generate a model of the environment. The model is used to decide how to navigate and what obstacles to avoid.

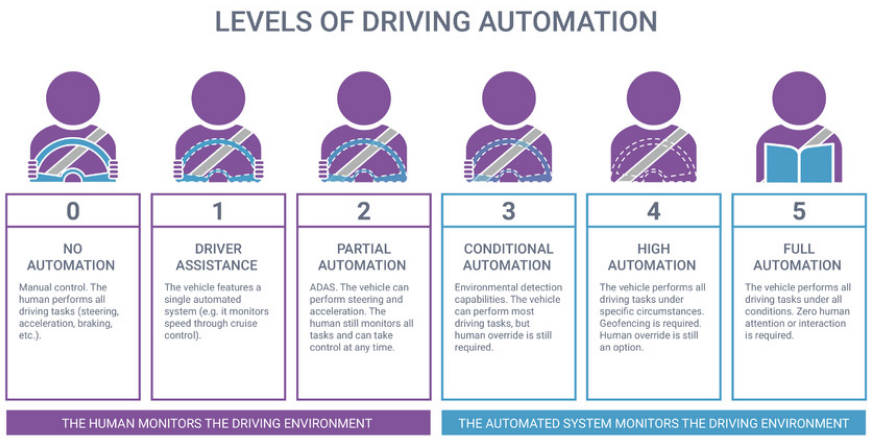

Figure 2.1: Levels of automated driving

In order to change the movement of the vehicle, computers use algorithms that analyse the data and determine the appropriate actions. For example, if sensors detect a stopped car in front of the vehicle, the algorithms may determine that the vehicle's speed needs to be reduced in order to avoid a collision. The computers then send corresponding signals to semiconductor-based actuators to engage the necessary actions, such as reducing speed or changing course.

We will not dive into more detail about how all the parts jointly work together and execute tasks because that would require writing a whole book. However, to better understand the article, it is important to have heard about the main aspects surrounding AVs because the same rules also apply to the simulated world, which is trying to be as reminiscent of the real world as possible. More about that will be covered in Chapter 4 when we talk about the virtual driving simulators and CARLA, in particular.

2.3 The six levels of autonomy

Moving on, it is relevant to mention that the classification of vehicles in terms of their autonomy is non-binary, meaning that there are not only fully autonomous vehicles and those entirely operated by humans, but various other levels of autonomy exist in between. This chapter will explain how autonomy levels are categorized and what level of autonomy is present in vehicles nowadays.

In 2014, The Society of Automotive Engineers (SAE) has proposed a standardization for categorizing the level of autonomy in vehicles, ranging from Level 0 (no automation) to level 5 (full automation) [9]. As shown in Figure 2.1, these levels are determined by the extent to which the human and the automated system are involved in monitoring the environment and

controlling the vehicle.

Currently, the majority of cars on the market fall into the lower levels of autonomy, with the driver still responsible for most aspects of driving. Some common automation features that new cars may offer to the owners include lane assisting, cruise control, automatic emergency braking, blind spot monitoring, automatic parking, autosteering and automatic lane changing.

Vehicles falling into the upper three categories are not yet available on the market, and it might be another several years until they become widely adopted. The main reasons for that are technological and regulatory challenges and public scepticism about AV safety.

Luckily, this article aims to address these questions by proposing a system for testing highly automated vehicles in a simulated environment, thus assessing whether the software agents controlling the vehicles are working as intended. More about that will be discussed in Chapter 6, when the proposed safety performance evaluation system is introduced.

2.4 Safety of autonomous vehicles

With regard to safety, there is a general belief that autonomous vehicles have the potential to improve transportation safety by addressing one of the leading causes of accidents and fatalities on the roads: human error. Drivers can be distracted by activities such as texting or talking on the phone or may become impaired or fatigued, leading to potentially dangerous situations. AVs, on the other hand, are equipped with sensors that constantly monitor their surroundings and detect potential hazards, allowing them to respond more quickly than a human driver could. This can be particularly important in emergencies, where reacting as quickly as possible might be vital.

Moreover, AVs can potentially eliminate road congestion and substantially improve traffic flow. Suppose there were no human-operated cars on the roads, and every participant would be operated by an AI with the ability to communicate among themselves and road infrastructure using radio waves. In that case, vehicles could better understand other vehicles' intentions and how the surrounding world will change. For instance, which driver might be turning soon, when the traffic light will change to green or where the road hazards are that should be avoided. If such communication were possible, travelling time would be reduced because all the vehicles would drive more efficiently.

In addition to improving overall road safety, AVs could help increase the mobility of people who cannot drive a car, for example, people with disabilities or older people.

However, it is essential to note that AVs also present potential risks, one being hardware

malfunction and the other being cyber-attacks. It is inevitable that mechanical components sometimes come with factory defects or develop a fault with time. They could contribute to inaccurate information that may lead to wrong and unsafe decision-making. For this reason, companies developing AVs must test all the critical components comprehensively to ensure their correctness. In addition, the future owners of the AVs must be provided with adequate systems that constantly check the state of the sensors and provide alerts that will announce the need for either technical service or recalibration if they are not in the optimal state.

Cyber-attacks could come into play anywhere where an AV communicates with online systems. It is inevitable that AVs, like other modern systems, will rely on external systems and services in one way or another. It could be by downloading software updates (which talking about the constantly evolving AV nature would be frequent), getting the latest traffic information for route planning, getting the weather forecast for traffic predictions or simply sending data to the manufacturer for analysis. All the information being sent can possibly be tampered with by malicious third parties and result in incorrect data, system faults and reduced safety.

Although it is crucial for AV manufacturers and regulators to prioritize safety in the design and operation of these vehicles, this research paper focuses solely on vehicle performance rather than correctness of their components and assumes that the systems are cryptographically safe and cannot be exploited by anyone. As mentioned in the book written by RAND Corporation in 2018 called "Measuring automated vehicle safety: Forging a framework": Comparing these machines at vehicle (or higher) levels fits people's intuition about safety [10].

Chapter 3

Human drivers and the road

This chapter discusses how people behave on the roads and why collecting real-life traffic data is crucial for AI model training. Section 3.1 analyses traffic incident data collected in the EU and U.S. in 2020, while the following Section 3.2 links data to human behaviour and analyses why some types of accidents happen more frequently than others. In Section 3.3, we finish the chapter by discussing how the traffic data and human psychology analysis can help us understand how the simulated scenarios should be designed and what situations they should reenact. Let us begin by looking at the statistics.

3.1 Road incident statistics

For the purpose of this research, it is important to understand the situations that lead to human driver failures on the road. These failures can occur due to a variety of factors, including distractions, participant impairment, road and weather conditions, place.

According to the European Commission's published data for road safety in the EU in 2020 [11], 52% of fatal road incidents happened in rural areas, while 40% occurred in urban areas and only 8% on motorways. The analysis also indicates that 70% of all fatalities in urban areas were made up of vulnerable road users such as cyclists, moped and motorcycle drivers and pedestrians, that account for the vast 37% of victims in city areas.

In the traffic safety facts annual report tables provided by The National Highway Traffic Safety Administration of U.S. Department of Transportation [12], it is clear that the majority of fatal road crashes in the U.S. in 2020 happened from 3PM until 11:59PM on working days and from 6PM and 3AM on weekends. The NHTSA's data set also indicates that the majority

(83%) of fatal crashes happened during normal weather conditions while rainy weather caused only 7% of fatal incidents. Out of the 83% of crashes that happened during normal weather conditions, 46% of accidents happened during daylight while 28% happened at night and 21% happened at night but in lit areas.

This shows an interesting fact that although there is a general perception that driving in snowy or foggy conditions is more prone to becoming a victim of an accident, the statistics show that it is entirely contrary. Most crashes happen when the conditions are best for all the human drivers (well-lighted areas in the middle of the day when everything is well observable).

It is also worth mentioning that 30% of all road fatalities occurred in alcohol-impaired road accidents. Before linking the data covered in this section to the goal of this project, it is important to look at why we have such a large number of incidents on the road.

3.2 Human nature and behaviour

A standard individual essentially has four main sensors when it comes to driving (two eyes and two ears) that provide information about the surrounding world and a brain which is responsible for making decisions.

Although a people have everything necessary to operate the vehicles on roads and following the established rules at all times they could do that successfully and safely, there are many factors that could make the roads unsafe regardless.

One reason is that emotions are a key aspect of human life and can heavily impact how a person makes decisions. When a person is feeling strong emotions such as anger, frustration, or stress, it can affect their ability to concentrate and make good decisions while driving. For example, a person who is feeling angry may be more likely to take risks or engage in aggressive driving behaviors, such as tailgating, speeding, or cutting off other drivers. This can increase the risk of collisions and accidents on the road.

Similarly, a person who is feeling stressed or distracted may have difficulty paying attention to the surroundings or react to situations. This is due to the humans' limited cognitive energy reserve. Neurological science has demonstrated that the human brain is incapable of focusing on two things at once as stated in the article "The Impossibility of Focusing on Two Things at Once" written by M. Hernandez [13]. This means that even listening to music or having a passenger telling a cunning story about his recent promotion reduces the driver's alertness and clouds the perception of their surroundings.

There are other factors that can have significant impact on how a person is able to operate

the vehicle one being fatigue or impairment, such as being influenced by alcohol or drugs.

Artificial intelligence has the potential to improve driving safety by addressing some of these factors. On one hand, AVs are essentially exceptionally sophisticated software systems with dozens and possibly hundreds of sensors providing reasonably accurate and reliable information about the surroundings and multiple cores for computation able to make trillions of calculations every second. Moreover, systems controlling the vehicle cannot be distracted by either delicious food advertisements on the street or a song on the radio. Even in the case of the passenger being able to communicate with the vehicle vocally, it could sacrifice some of the computational power of one of its multiple cores for that fraction of a second needed to calculate the most appropriate answer. The AV systems cannot become fatigued or get influenced by substances or emotions. If programmed well their behavior and decision making cannot be altered by external individuals or systems.

However, it is important to note that AI systems are not immune to hardware malfunctioning, software errors or cyber attacks by very sophisticated attackers. AV safety was discussed in Section 2.4.

3.3 Statistics relevance to the research

Now that we discussed the trends in road accident statistics and looked a bit into the differences between human and AI vehicle operators, it is time to understand why all this information is crucial for developing adequate systems able to generate drivable scenarios and analyse vehicle safety in situations that are the most prone to accidents.

Comprehensive road statistics analysis can help ensure that the simulated scenarios accurately reflect the real-world conditions that AVs will encounter. Scenario generation has to take into account when the most accidents occurred and what factors, such as weather, lighting, road surfaces, or places, influenced them the most. Extracting accurate data from the statistics allows testing the AVs in realistically challenging situations and shows that the research is as representative of the real world as possible.

One way road data could be used is to train AI models to predict what situations could be challenging to drive and prone to accidents. We will discuss more about scenario generation when we introduce a way of automatically generating driving scenarios using a machine learning model in Chapter 5. The proposed ML algorithm is able to predict the parameters of new scenarios based on the scenario examples it has seen previously and then put the numbers to life with the help of the CARLA modules.

From the data, analysed in Section 3.1, it is clear that most simulated scenarios should occur in rural and urban areas reflecting situations that impact road safety the most. However, highway roads also account for a substantial proportion of road incidents; thus, some scenarios should also happen on highways.

Regarding the weather statistics from Section 3.1, it is clear that to mirror the conditions prone to vehicle collisions; scenarios need to be designed in various weather conditions, with most of them based on normal weather circumstances.

The datasheet also indicates that many road incidents involved pedestrians and two-wheel vehicle drivers, e.g., cyclists and motorcycle drivers.

To sum up, it is clear that in order to develop a reliable system that can generate driving scenarios that accurately reflect real-world conditions, an enormous amount of diverse traffic data is needed that could be restructured so it can be used to train machine learning algorithms. More details about this process and the chosen way to implement the framework component responsible for scenario generation are discussed in Chapter 5.

Chapter 4

The simulated world

In this chapter, we will be focusing on the simulator used in this project - CARLA. The reasons why this simulation framework was chosen, along with its advantages and role in the research, are explained in detail in Section 4.1. Additionally, Section 4.2 covers the limitations of the simulator and explores the potential inaccuracies that may arise during the research due to action taking place in the simulated environment.

4.1 CARLA simulator

As mentioned in the introduction, this work aims to design a system for evaluating vehicle safety performance in realistic simulated driving situations and later evaluate the system's accuracy by testing it with the participants. For this, a sophisticated simulator was needed that could provide a reasonably accurate representation of the human world, from road infrastructure and people to physics and weather impacting them. In addition, it had to have a way of manipulating the world that is being simulated, launching a first-person driving mode, and connecting AV implementations to the simulation. Compared to other competing frameworks, CARLA (Car Learning to Act) [14] has proven to be an ideal choice for this project.

One of the key benefits of CARLA is its scalable client-server architecture, which allows for efficient manipulation and customization of the simulated environment. The server runs the simulation, renders objects and sensors, and maintains physics. On the other hand, the client can manipulate the simulated world by changing weather conditions, spawning actors and controlling them, and getting information about the simulated world and what is happening there at any given time step. It does so using the CARLA Python API, which provides a

comprehensive set of use cases and commands well-documented by CARLA developers. Another benefit of the client-server architecture is that it is relatively easy to connect different AV agent implementations to the simulator and have a defined behaviour of the server and all its components working separately on their own.

The third benefit is the Agent implementations provided by the CARLA development team. They allow self-driving agents to be put on the map and instructed on where to drive. Although far more sophisticated AI implementations can be bridged with CARLA, like Autoware.ai, which is briefly mentioned in section Section 9.1, the CARLA BasicAgent is used to compete against human drivers in the software testing in Chapter 9. The main reason for that is that this project aims to introduce a way of automatically testing the safety of AI implementations operating vehicles rather than connecting different implementations with the simulator. The proposed ideas work with any implementations that can run in the CARLA environment. The safest and most time-efficient way to ensure the framework performs its tasks was to use built-in and easy-to-customise AV implementations

4.2 The limitations of the simulator

Although CARLA is a powerful tool for testing how AI agents and human drivers operate the vehicles, it is important to understand that it has limitations compared to the real world.

One of them is that it cannot account for the real world's unpredictability. Many factors could impact how an AV reacts, and it is challenging to hard-code them all when designing the scenarios that could occur on the road, similar to how it is difficult to tell the AV how to react in all those situations. There are infinite possible scenarios; in some, the autonomous agent could react well; in others, not so much. This research focuses on designing scenarios with realistic traffic conditions and unpredictable events that require quick reaction and decision-making. The scenarios will test the adaptability and robustness of AVs in unexpected circumstances while also monitoring the overall vehicle's safety.

Another simulator's limitation is the fixed number of road conditions programmed into it. Although CARLA offers a wide variety of customisable stuff, it is still restrained by parameters set by the developers. To exemplify this, snowy winter conditions with iced roads cannot be simulated because there is no way to facilitate this. In addition, there is no efficient way to check if the vehicle has stopped at a stop sign, which is the opposite when talking about checking if the vehicle ran a red light.

Moreover, it is important to note that CARLA is only as accurate and good as the algorithms

that power it. If the internal calculations are inaccurate, that could affect the simulations and lead to incorrect conclusions about the performance and safety of the drivers.

Chapter 5

Designing the driving scenarios

This chapter explores various approaches for generating scenarios, with a specific focus on the method selected by this article. In Section 5.1, we start by giving an overview of some of the possible ways that scenarios could be generated, while Section 5.2 and Section 5.3 examine the chosen scenario and path generation methods and discuss their benefits and drawbacks in terms of accurately predicting new scenarios. Finally, in Section 5.4, we conclude the chapter by discussing potential improvements for the current method of scenario generation.

5.1 How the driving scenarios could be designed

Let us begin by discussing the need to design driving scenarios for autonomous vehicles. According to Ruxin Li and Runping Zhain's article "Estimation and Analysis of Minimum Traveling Distance in Self-driving Vehicle to Prove Their Safety on Road Test" [15], self-driving vehicles must drive at least 7.2 billion kilometres to prove their accuracy and confidence to an acceptable level. However, this distance is enormous, equivalent to circling the Earth approximately 180,000 times. If we imagine a car driving on the road 24/7 at an average speed of 60km/h, it would take that car 13,699 years to complete this distance. Therefore, simply putting a vehicle on the road and testing it to the fullest extent would be inadequate, given the time required. This is where the simulated environment comes into play.

A well-constructed framework that can accurately test how the software system controlling the autonomous vehicle behaves can be a huge time-saver allowing fully autonomous cars to see the daylight more quickly. However, as we will soon see, this task is very challenging, and some traditional ways of solving it might not work.

One of the traditional ways a scenario generation could be imagined is by designing the scenarios manually, putting objects into specific locations, and trying to recreate complex driving situations in the simulated world. However, time is the enemy, and doing everything by hand would not be much better than letting the autonomous vehicle drive for decades or even more. In addition, it is impossible to think of all the possible edge cases the vehicle might encounter because life is stochastic and non-fully observable. An infinite amount of possible things can happen at any given time, and hard-coding them in the software would bring poor yield. What is needed is a way of creating complex and realistic driving situations automatically, and assessing how good the generated scenarios are.

This is a perfect moment to start talking about AI in a bit more depth and discuss how it can help us in the case of scenario generation. Specifically, the rest of this section focuses on machine learning, a concept that has long been laid on the shelf and was made possible in practice only in the last few decades because of the immensely increased computational power resources and advanced hardware capabilities. Although a young member of the technology field, machine learning has already demonstrated its significance in various areas, such as medicine for predicting illnesses, economics and finance for predicting market changes, etc.

One of the fascinating aspects of machine learning applications is that they are not ordinary programs designed to perform specific tasks. One could argue that they are still meant to perform particular tasks, mainly predict values, but how they do it is markedly different. These algorithms are designed to act based on the data they have been trained on and make predictions about new data using their knowledge and gathered domain information.

To illustrate how the machine learning algorithms work and understand why we are talking about them, let's look at a specific example use case in the finance industry - fraud detection. Banks and credit card companies use machine learning algorithms to analyse customer data and transactions to identify suspicious activity. They train the algorithms on various instances of fraud, allowing algorithms to investigate patterns and detect new fraudulent behaviour, protecting customers from financial losses due to fraud and helping financial institutions avoid losses from fraudulent transactions. Companies such as Mastercard [16], PayPal [17] and others use sophisticated machine learning algorithms for fraud detection in their payment systems.

Now returning from the finance field back to autonomous driving, machine learning algorithms can also be used to predict new scenarios based on the examples they have been trained on. One thing worth mentioning before we dive into the possible ways of teaching ML algorithms to generate scenarios is that they require enormous amounts of training data to perform

well and avoid underfitting. Underfitting occurs when the ML model is too simple and fails to capture the complexity of the data it is trying to model. Another problem that may arise is overfitting. This is when the algorithm is too specific and cannot generalize the information well, meaning it has learned to classify the training data but hasn't learned the underlying patterns and thus fails to work with new inputs. That said, one cannot think that giving the algorithm a few descriptions of driving situations from the real world would make the algorithm perform well. In reality, an algorithm would have to be trained on millions and even billions of precisely described scenarios to achieve its goal - creating new scenarios.

What is meant by precisely described scenarios is not that an algorithm needs essays describing the actors and actions that took place in the real world but well-structured data acceptable by the chosen style of the algorithm. Usually, the data is translated into meaningful numerical values that the ML algorithm can synthesize. Depending on the choice of the algorithm, the training data is processed differently, usually altering the internal domain representation of the ML model. This section will not dive deeply into the machine learning world. Still, it will give an overview of what concepts could be combined with machine learning methodologies to assist in scenario generation.

One instance of a machine learning algorithm that could be used for scenario generation is multi-output regression. The way it works is it is trained on instances of scenario descriptions containing the significant metrics in numerical form. Each of the instances is labelled by one or more labels. After a considerable time of training, the algorithm is able to predict new metrics given the labels as input. This approach is implemented as part of the solution for the automatic scenario generation that this article presents and is discussed in more detail in Section 5.2.

Another way of training the model could be by using video recordings or fragments of them with precisely labelled data and objects and described dangers. A deep neural network could use this data to build an inner model of what dangerous situations look like, how the participants behave, etc. With the help of other machine learning and data analysis models, this model could be used to design new unseen situations or at least translate them into ones usable by virtual simulators. This approach is better explained in the article "Automated generation of virtual driving scenarios from test data" by Robin van der Made [18].

5.2 The selected approach for scenario generation

Now that we understand the possible approaches that could be used in scenario generation, let us dive into detail about what path this article chose and how the algorithm proposed by this project is functioning. Having already introduced the simulator used in this project in Chapter 4, it is time to investigate how CARLA's functionalities could be used to our advantage when generating the scenarios.

As a sophisticated project, CARLA provides a wide range of configurations allowing users to customise the simulated world they see fit. One of the significant components for that is the TrafficManager. Its main objective could be described in one sentence - controlling all the actors in the simulation and executing their actions according to the specified behaviour.

Before we begin analysing the scenario generation method described in this project, it is worth mentioning that for this research, there was no access to any large database whose data could be used for machine learning. Thus, the algorithm being presented is simplified and uses pseudo data that might not represent real-world conditions. In addition, the algorithm is prone to underfitting because the data used to train it was generated using a primitive approach (creating a few unique training instances and then creating copies of them with all the values increased or decreased by some percentage).

After a lengthy analysis of the simulation world and the TrafficManager with all its functionalities, it was decided to describe scenarios using attributes presented in tables in Appendix A; Table A.1, containing all the attributes that describe the weather that can be customised by the client using the CARLA simulator and Table A.2, containing all the necessary attributes describing the traffic conditions that can be created and customised using CARLA client and the TrafficManager. A scenario could be described in full by combining the attributes of both tables, assigning a value to each of them, and putting them into a JSON object, the chosen standard for scenario specification in this project. A scenario example can be seen in Figure A.1 in Appendix A.

Before moving on, it is worth noting that the algorithm training data and all the other files specifying the necessary modifications, such as what maximum value each attribute can possess, are stored in the scenario_generation_data/learning_data in the project files. The generated scenarios are stored in the directory scenario_generation_data/generated_scenarios. The machine learning algorithm can be found in the software/generating_scenarios directory. Finally, how to run the scenario generation is explained in the Section B.1 in Appendix B, where the user guide is presented.

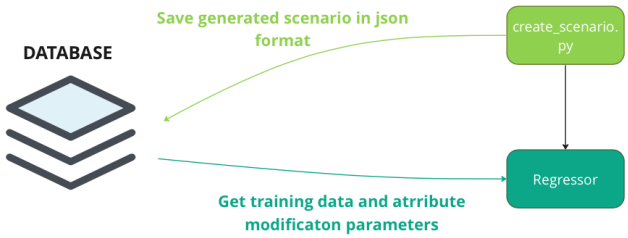

Figure 5.1: Scenario generation overview

Now that we described what the scenarios look like, gave references to where the necessary files can be found, it is time to look at how the algorithm works. The algorithm implementation can be found in the **regressor.py** file. At its core, the Regressor uses machine learning algorithms provided in the *Scikit-learn* library [19]. It is done so because *Scikit-learn* is a popular open-source library delivering a wide variety of well-implemented and accurate machine-learning tools used by many companies and developers. This ensures that, at the very core, our algorithm is behaving correctly. The algorithm also uses other library methods to split the training data into training and testing sets, work with .csv files and data frames.

Once called, the algorithm first trains itself on the training data provided in the scenarios.csv file that can be found in learning_data directory. After the training phase is complete, the algorithm tries to predict what attributes should a new scenario possess given the input labels: difficulty, wetness and sun altitude angle. The algorithm then checks if the predicted values are within their valid ranges (the ranges for each attribute are described in minimums.csv and maximums.csv files in learning_data directory), rounds them using values from rounding.csv and saves the newly generated scenario in a JSON file to the directory generated_scenarios. The high-level scenario generation infrastructure is shown in Figure 5.1. The purpose of the generated scenario file is to describe the traffic and weather conditions. It includes information used by other components that populate the world, customise the weather and set up the traffic. Both PathBuilder and Scenario classes use the generated scenario, and the ways they use it are described in Section 5.3.

One thing that was not mentioned yet is why the chosen labels are difficulty, wetness and sun altitude angle. The first one, difficulty, describes how complex the scenario is overall. For instance, the harsher the weather conditions and the more disobedient drivers there are, the more challenging the scenario is. The second label, wetness, is used to differentiate between dry

and clear weather and rainy or stormy weather. The smaller the wetness value, the smaller the attribute values related to precipitation, fog, rain deposits, etc. The third and last label, the sun altitude angle, is used to differentiate between the different times of the day. To sum up, these three labels allow the algorithm to generate the scenario of desirable difficulty, wetness level and also the preferred time of day.

5.3 Building paths and populating the simulator

The previous section mentioned that the scenarios generated by the scenario generation algorithm are used by the PathBuilder and Scenario classes. Let us begin by discussing the path generation method this article proposes.

To test how the vehicle is performing in the scenarios, it is not enough to specify the weather parameters or how many other participants will be involved. We need to give the vehicle an objective that it would try to achieve. In this case, the aim of the car is to drive some route and reach the destination safely.

Luckily, as this project aims to create a framework for automatic vehicle safety testing, manual path generation for every run is not a valid solution. That is why the PathBuilder class was developed (stored in software/carla_scripts/scenarios/path_builder.py). This class is responsible for creating a path given the scenario JSON file.

The algorithm works as follows; first, it looks at all possible maps in the data/maps/opendrive_format directory and loops through all the maps looking for one that satisfies the following criteria from the scenario file:

$$\begin{cases} number\_of\_junctions \leq total\_number\_of\_unique\_junctions\_in\_map \\ distance\_in\_metres \leq total\_map\_road\_length \end{cases} \tag{5.1}$$

If the map that is being analysed satisfies the criteria, it is chosen, and the path-building algorithm proceeds to try and create a path on that map; if the map does not meet the criteria or a path cannot be created in it, the algorithm takes another map and proceeds with the same actions until a path is generated or the algorithm is aborted meaning it is impossible to create a path in any of the maps given the scenario requirements.

It is also worth mentioning how the path itself is created when the correct map is found. If a valid map is found, the algorithm proceeds as follows: first, it selects a road element randomly, then enters a while loop and keeps looking for road successors, adding them to the

list of checkpoints until the scenario criteria are satisfied. The successor elements can be of two types: a road element or a junction. If the currently analysed road element points to another road element, the while loop proceeds with the successor road element in the next iteration. However, if the current road element points to a junction, then a random successor of the junction is selected for the next iteration. Note that the successor of the junction is always a road element. If at any point the algorithm encounters a loop (two road elements pointing to each other) or a dead-end, it starts the process all over again. The algorithm currently has a limited amount of tries that it can have on a single map $x = 10,000$. If in x tries the creation of the path is unsuccessful, the map file is either corrupt or a path there cannot be generated. This is made so that the algorithm never enters eternal while loops. The high level overview of the algorithm can be seen in the pseudocode fragment below:

Algorithm 1 High-abstraction-level path building algorithm

```
procedure PathBuilding(scenario)
    maps ← read OpenDrive maps from the database
    for map in maps do
        if map is valid then
            r ← randomly pick a road element from the map
            road_elements ← []
            total_length ← 0
            number_of_junctions ← 0
            while number_of_junctions < required_number_of_junctions
                        and total_length < required_total_length
                        and not road_elements contains dead-ends do
                total_length ← total_length + r.length
                road_elements.insert(r)
                successor ← r.successor
                if successor is junction then
                    number_of_junctions ← number_of_junctions + 1
                    r ← random successor exit that is not current r
                if successor is road element then
                    r ← successor
            path ← build path from road_elements
            if path was found then
                return path                          ▷ Else, try another map
    return None          ▷ If no path was found over all the maps, return None
```

If the maps in the opendrive_format directory are well formatted, the final generated path could look like the one presented in Figure 5.2 where the blue colour marks the beginning and the red one the end. The colourful lines, gradually moving from blue to red, indicate the lanes that a vehicle can drive on. To draw the path on the map, the draw_lanes_terminal.py script has to be run with the path's name as the argument -p. The script is located in software/-carla_scripts/helper_scripts directory.

Figure 5.2: Well-formatted path

However, the algorithm not always behaves in our best interest. In Section 4.2, when discussing the limitations of the simulator, we mentioned that CARLA is only as accurate and good working as the algorithms that power it. One of its issues revealed itself when designing this path generation algorithm. To begin with, CARLA, for its map specifications, uses OpenDRIVE format, which is a standard way of specifying map layouts and road networks[20]. The algorithm proposed by this article also uses OpenDRIVE files to analyse the road networks and thus generate paths.

As mentioned before, when a map is determined, the path generation algorithm randomly takes the first road element. If that initial pick is indeed a valid road segment, the algorithm can generate a path like the one pictured in the Figure 5.2.

Unfortunately, the bigger part of OpenDRIVE maps provided by CARLA are poorly structured in some places (in version 0.9.13), meaning that the files specify that a road element is present when it is not. An example of a generated incorrect path can be seen in figure Figure 5.3, where the points selected by the algorithm are outside the map where there is no road. After analysing the OpenDRIVE maps provided by CARLA, it was discovered that the specifications are indeed misleading. In addition, some maps possess other issues, like one road element pointing to itself or loops where two distinct road segments point to each other. The problems are illustrated in Figure C.1 and Figure C.2 in Appendix C.

An example of a generated path can be found in Figure A.2 in Appendix A. The points from the list **path_checkpoints** are used by the GlobalRoutePlanner to build a path that a vehicle has to drive. The **town** attribute indicates the map, and the **start_location** tells where

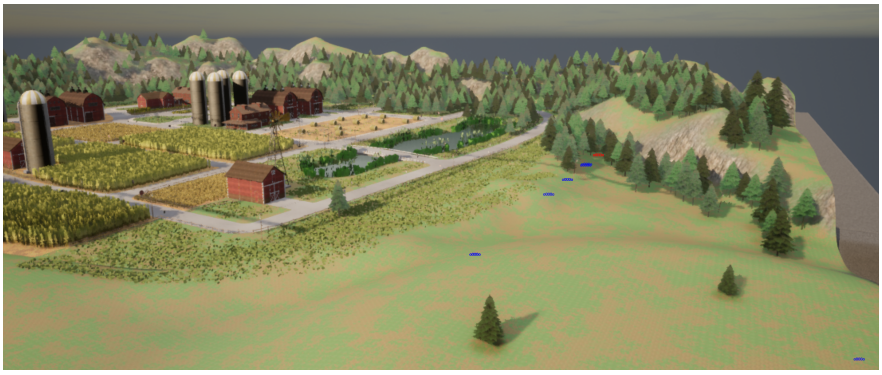

Figure 5.3: Poorly-formatted path

to spawn the car.

Now that we have talked about the PathBuilder and the issues it was facing, it is time to look at how the Scenario class uses both the generated path and the generated scenario to fulfill the simulation requirements. The Scenario class can be seen as a module responsible for spawning all the non-hero actors and specifying their behaviour according to the generated scenario attributes.

It first tries to spawn the necessary amounts of pedestrians, vehicles and two-wheel vehicles. If there are not enough spawn points for all the cars, the maximum number of them is spawned. After a successful spawn, the module customises the behaviour of all actors according to the values provided in the scenario file. For instance, if the $proportion\_of\_light\_ignoring\_vehicles = 0.2$ and the $light\_ignoring\_percent = 50$, then the algorithm randomly selects a fifth of the vehicles and sets their behaviour as such: when a red traffic light is approached, there is a 50% chance that the car will ignore the traffic light. It does so, for all the behaviour-changing scenario attributes. Once the actors are spawned and their behaviour is set, the Scenario module assigns all the actors to TrafficManager to take care of their movements in the simulation. How the simulation takes place and when the TrafficManager is called will be discussed in more detail in Chapter 8, when we talk about what the simulation session looks like and how all the components work together.

5.4 Possible future improvements

Having introduced the scenario generation methodology, it is now time to finish this chapter by discussing the possible improvements that could be implemented in the future. First, to ensure that invalid paths are never generated, well-formatted OpenDRIVE map specifications

should be used. From all the current map specifications, after some testing, only the Town01 showed the potential of being well-formed. Using other maps sometimes produced wrong paths. Another part that should be improved is the path-building algorithm itself. Right now, it creates the paths by randomly trying to create one. It would be better if it did this in some structured way, looking for interesting patterns that could be utilised. However, the OpenDRIVE files currently contain only information about the road elements and junctions, providing very little extra information. In other words, more sophisticated road network descriptions should be used, providing information about roundabouts, highways, u-turns, road signs and other things. Finally, the scenario generation algorithm could be improved in several ways. Firstly, well-described and realistic scenarios could be used for training. Currently, the algorithm uses naively generated training data, as discussed in Section 5.2. Furthermore, the machine-learning algorithm could be improved or even changed to use deep neural networks and predict scenarios from more sophisticated training data sources like video material.

Chapter 6

Assessing vehicle safety

In this chapter, the article proposes a way of assessing vehicle safety on the virtual roads. We begin by giving an overview of the suggested method and explaining why it was chosen in Section 6.1. In Section 6.2, we continue our discussion by explaining what safety metrics are monitored in the simulations and how the collected data is processed. The chapter is finished in Section 6.3 after providing a mathematical way of evaluating the vehicle's performance.

6.1 Monitoring and recording data

After implementing the scenario generation and path-building modules, the next step was to design a sub-system to monitor vehicle actions and record the behaviour, allowing for the data analysis later. As mentioned in Section 2.4, when we talked about the safety of autonomous vehicles, this article deals with evaluating the safety at the vehicle level, meaning how the car acts on the road. For the purpose of this research, it was decided to track the vehicle's actions during the simulation process and then save obtained information to XML files. The XML (Extensible Markup Language) file standard was chosen for its convenient structuring of large files, allowing for efficient data parsing in the tree structures.

The central class of the safety monitoring system is the **Manager** class, which is responsible for checking the vehicle's state at each time step and informing the other monitoring classes if data is needed to be recorded. The manager class and the monitors are built based on the observer design pattern when one object broadcasts updates to multiple other objects. Having given the overview of the system, let's continue by discussing each component participating in the safety assessment.

Let us begin by talking about the monitors of the system, the first one being the **RouteMonitor**. This class tracks the vehicle's progress along its route and ensures it stays within the allowed lanes. The route needed to drive is obtained from the GlobalRoutePlanner (provided by CARLA developers), which returns a list of waypoints given a list of coordinates. The coordinates come from path files discussed in Section 5.2. Using these waypoints, the route that is monitored by the RouteMonitor is created by recursively finding all the lanes that the vehicle could use. At each time step, the RouteMonitor checks if the car has not left the permitted lanes and tracks the number of route points that have been covered. This metric is later used to calculate route completion. At the end of the simulation, the RouteMonitor saves the route completion, whether the finish was reached, and the coordinates of all the route points and whether they have been reached to the data/recordings/participant_[A]/scenario[1]/route_data.xml file.

Another monitoring class implemented was the **CollisionMonitor**, which records data about collisions that the vehicle got involved in. Whenever a crash occurs and is detected by the CollisionSensor present in the Manager class, the CollisionMonitor records information about it, including the collision type (collision with a human, another vehicle, road objects, or buildings), the penalty points given to the driver, the time step, and the coordinates where it happened in the recording/recordings/participant_[A]/scenario[1]/collision_data.xml file.

In addition to the route and collision monitors, a **SpeedingMonitor** was developed to record data about instances when the vehicle was speeding. At each time step, the Manager class checks if the driver is exceeding the speed limit, and the SpeedingMonitor records data about it, including the allowed speed at that location, the vehicle's speed, the penalty points received for speeding, the time step, and the coordinates where it happened. This data is saved to the data/recordings/participant_[A]/scenario[1]/speeding_data.xml file.

A **LaneMonitor** was also implemented to record instances when the vehicle crossed solid and double solid road markings. In addition, instances of crossing broken lines without showing correct turn indicators do not go unnoticed. The LaneMonitor saves the lane type, the penalty points given, the time step, and the coordinates to the data/recordings/participant_[A]/scenario [1]/lane_marking_violation_data.xml file.

A **TrafficMonitor** was created to track stop sign, stop marking and traffic light violations. The Manager class observes the vehicle at each time step and records traffic violations in the data/recordings/participant_[A]/scenario[1]/road_traffic_violation_data.xml file. The saved data includes the violation type, the penalty points given, the time step, and the coordinates where it happened.

Finally, a **VehicleLightMonitor** was implemented to register data about the state of the vehicle's lights and any improper use. For example, failing to use turn indicators or not having headlights in the dark would result in light violations recorded in the data/recordings/participant_[A]/scenario[1]/vehicle_light_misuse_data.xml file. The saved data included the violation type, the penalty points given, the time step when it occurred, and the location where it happened.

In addition to the monitor classes, a **Recorder** class is responsible for creating a log file following the CARLA standards that holds all the data from the simulation. The file can later be used to replay the simulation using the **Replayer** class via the replay.sh script that can be found in software/carla_scripts directory.

When the simulation ends, the manager instructs all its monitors to write the data stored in their buffers to the corresponding files. For example, participant_A's simulation results of scenario one would be stored at data/recordings/participant_A/scenario1/ in different .xml files, e.g. collision_data.xml, speeding_data.xml. The manager also gives a command to the Recorder to stop recording and save the .log file of the simulation that can be later used to replay the simulation. This gives the ability to analyse the simulations multiple times and thus catch all the minor details that can be important or monitor some data manually if needed.

To better understand how the data is stored or how the monitoring works, the reader is encouraged to look at the implemented classes in the software/carla_scripts/recording directory in the source code. Additionally, the overview of the observer design pattern can be seen in Figure 8.1 in Chapter 8. To see what the recorded data files look like, please refer to Figure A.3 in Appendix A showing an example of collision_data.xml file or explore the data/recordings directory.

6.2 Metrics to record

In this section, we look at what quantitative metrics from the recordings we can use to evaluate a vehicle's performance in a given scenario. After long consideration, two groups of metrics were developed: time and accuracy-related metrics allowing to give positive points to the user and safety-related metrics, which give penalty points to the user for violating some safety requirements. The two groups are discussed in the following two subsections.

6.2.1 Time and accuracy-related metrics

The first one considers how neatly and optimally the vehicle is being operated and includes these metrics:

Timeliness – Each scenario has a specific optimal time to complete without violating road rules or breaking speed limits. The timeliness is calculated by dividing the optimum time value by the time the driver took to complete the route. The quicker the driver is, the larger the timeliness value will be. The value proportionally increases or decreases the driver's score depending on his swiftness.

Proportion of route completed – The route is a line of dots on the map, the first being the starting position of the vehicle and the last being the finish position of the path that needs to be driven. Each dot has coordinates associated with it and is in the middle of the lane. This metric shows what percentage of waypoints were covered. For the waypoint to be considered covered, the vehicle's centre point must pass it within a certain distance. This ensures that if a car is driving not within the boundaries of the lane that it can drive on (one wheel is on the line, for instance), the waypoint will not be covered, and thus the final score will be lower. It is worth noting that if the road has multiple parallel lanes that can be driven by the vehicle, the algorithm checks if the vehicle is within the boundaries of any of those lanes. If it is not within one of those lanes but is on some of the lane markings between two lanes and is showing a turn indicator (the vehicle is changing lanes), the waypoints within a certain distance from the vehicle's centre position will be covered. This is an exception to the rule that the vehicle must stay within the lane's bounds. However, because the action takes place in a simulated world and it is difficult to feel the vehicle and be convinced that it stays within the bounds of the lane, the gamma discount factor will be applied to reduce this metric's weight.

6.2.2 Safety related metrics

The second group is evaluating how safely the vehicle is being driven and considers these metrics:

Speeding instances – This metric indicates how many penalty points were assigned to the user due to exceeding speed limits. This is important because the vehicle's speed directly impacts the safety of the people sitting inside and the safety of other drivers, passengers, pedestrians, and animals. In an ideal situation, both AV and the human-drivers need to obey speed limits because speeding delays the reaction time, narrows the vision angle and increases the braking distance resulting in diminished safety.

Collision instances – This metric shows how many negative points the vehicle earned throughout the simulation because of collisions with any objects (pedestrians, road objects, other vehicles, buildings). Moreover, collisions while speeding will result in even more negative points added to the final score because hitting something at high speed, will typically result in more damage done and any violation done while speeding will almost always result in the driver being guilty and responsible for causing the incident.

Traffic light violations – This metric shows how many negative points the vehicle earned throughout the simulation because of running a red traffic light. In the case of a vehicle violating these rules while speeding, even more negative points will be added to the score because the risk of causing a severe road incident is increased, resulting in diminished overall road safety.

Lane marking violations – This metric shows how many negative points the vehicle earned throughout the simulation for crossing road lines other than broken ones (- - - - -). Lane markings have to be obeyed as they are there to prevent people from overtaking or turning where it is unsafe to do. Again, lane violations while speeding will give more negative points than not speeding.

Vehicle light violations – Driving at night without vehicle headlights on or in a fog without headlights and fog lights on will result in negative points given to the vehicle. In addition, changing lanes without respective turn indicators blinking will result in a worse outcome from the route metric. Showing turn indicators and having headlights on in the dark is extremely important for the vehicle's passengers and other traffic participants. The fact that all the road actors are visible and their intentions are clear directly impacts how safe the road is.

Another important metric for evaluating safety is the number of penalty points the participant receives for not stopping at stop signs and stop lane markings. This metric is of great importance as stop signs and markings are typically placed in dangerous areas. However, the current version of CARLA does not provide an efficient way to check for these violations thus this is left as a place for further improvements. More about it will be talked about in Chapter 10 when we talk about the possible future improvements of this project.

6.3 Evaluating the performance

Now that monitoring of the vehicle's actions was discussed and the metrics being captured were presented, it is time to give an overview of how the performance can be evaluated. Let us begin by introducing a table with penalty scores for violations of different safety requirements. This table can be found in Appendix A in Table A.3. It consists of three columns: the first one

highlights the category of the violation, the second one gives the number of penalty points a driver is assigned if he violates the requirement, and the third column shows the values that are assigned to the driver's score for doing the violation while speeding.

Note that the values in the table are just a proposition of what could be used, and they can and should be changed to fit the project's needs accordingly. The weights shown in the table are used to assess the negative part of the driver's performance, namely how unsafe it is. All the values introduced in this section are used to evaluate the performance of participants competing in scenarios in Chapter 9.

In order to evaluate the driver's performance, a mathematical formula was developed that consists of three main elements: the positive score, the penalty score and the discount factor γ. The discount factor is a hyper parameter and is there to deal with the inaccuracies caused by the simulated environment. It should be adjusted depending on the simulated environment and the objective of the research. The exact purpose of the γ will be discussed later when we will discuss how the penalty score is calculated because it is heavily impacted by γ.

Let us first talk about the positive evaluation element. It is made of the road completion c multiplied by the scenario difficulty d multiplied by the timeliness of the driver. The meaning of the c is what proportion of the route the vehicle completed successfully. The difficulty d is retrieved from the scenario file, indicating how challenging the scenario was. The timeliness is the optimal scenario completion time divided by the time it took for the driver to complete it. In simpler terms, the quicker the more complex scenario is completed, with the driver sticking to the path without any deviation, the higher the positive score they will receive.

The penalty score is the sum of all the penalty points from all monitors multiplied by the discount factor γ. It was mentioned before that γ is a hyper-parameter, meaning that the γ value can change the penalty score's effect on the final score. It is used to deal with the inaccuracies of the simulation and monitor malfunction. To exemplify this, consider a case where $\gamma = 1$. In that case, we are essentially saying that all the penalty scores received by the vehicle are correct and reasonable, giving the final score a maximum negative amount. However, the world is imperfect, especially the simulated world; thus, it is rational to assume that not all the sensors are 100% correctly obtaining all the data and not all the physics are impeccably implemented. As we will see in Chapter 9, there are many instances when simulator physics breaks down, and weird things start to happen. For example, upon colliding with another car, a car gets tossed into the air and lands a few streets further. Continuing about the monitors, the traffic monitor might capture that the vehicle did not stop at a stop sign, although it did, just

a few centimetres before it. Naturally, a parameter is needed to account for the unaccountable. By using the discount parameter, we are reducing the total penalty score while at the same time also reducing the miscalculation influence on the final score. Depending on the simulator and how well the physics and monitoring work, the γ value should be adjusted. Summing all up, the final formula makes an equation shown below:

$$score = c \times \frac{t_o}{t} \times d - \gamma \times \sum_{x=1}^{n_m} f(m_x) \tag{6.1}$$

where: γ = discount factor and $0 < \gamma \leq 1$

m = a set of lists of penalty values

$f(m_x)$ = sum of all the penalty points in list m_x and $m_x \in [0; +\infty)$

c = proportion of route completed and $0 < c \leq 1$

t_o = optimal amount of time (in seconds) to complete the scenario and $t_o \in (0; +\infty)$

t = amount of time to complete the scenario and $t_o \in (0; +\infty)$

d = difficulty of the scenario and $0 \leq d \leq 1000$

n_m = number of lists of penalty values in set m

However, it was not mentioned yet, how the function $f(m_x)$ is calculated. If we assume that $f(m_x) = f(x)$, since m_x is a list of penalty scores of metric x, we get the formula shown below:

$$f(x) = \sum_{i=1}^{n_x} x_i \tag{6.2}$$

where: $x_i = i_{th}$ score from list x and $x_i \in (0; +\infty)$

n_x = number of scores in list x

Now that the final formula has been introduced, it is time to return to one bit that was mentioned but not analysed yet, the optimal time value t_o.

The purpose of the t_o is to proportionally increase the final positive score if the driver is driving faster than the average and proportionally decrease it if the driver is driving more slowly than the average driver. If we look back at the score calculation formula, we can see that the optimal time value t_o is divided by the time the driver took to complete the route. The positive score is mainly determined by the scenario difficulty d, while the c is there to either retain the maximum score or decrease it if the driver did not complete the whole route. On the other

hand, the t_o divided by t increases the score if the driver is quick and decreases it if he is slow. Let us take a look at an example. Suppose the scenario's difficulty is 500 and the optimal time value to complete that scenario t_o is 200 seconds. If the driver finished the route in 100 seconds, meaning he is two times quicker than the calculated average, he gets $c \times 2 \times 500 = 1000c$ positive points for the scenario. However, if he were to complete the scenario more slowly than the optimal, he would get $score < d$.

Given the simulated environment, this article proposes a mathematical formula for t_o evaluation. It consists of three main parts. The first is the distance s divided by the average maximum allowed speed v_{avg}. This part tells us the ideal time the vehicle would take to finish the distance if it was driving in a straight line without any stops and other traffic. However, the roads are full of other drivers changing lanes, slowing down, making turns, talking on the phone and not seeing the green light, etc. For this reason, the first part of the formula is multiplied by $1 + \alpha$, where α denotes the intensity of traffic $\alpha = 1$ being the road is almost a gridlock and $\alpha = 0$ being the road is completely empty, and there are no other vehicles on it except the driver himself. Of course, intensity value should consider other factors such as road works, diversions or road surface quality. However, for the purpose of this project and the simulator's limitations, we are only considering the traffic density. The third part of the formula is a sum of all stops the vehicle has to make on the road. This includes pedestrian crossings, traffic lights, stop signs/markings and others. The complete formula for t_o can be found below:

$$t_o = \frac{s}{v_{avg}} \times (1 + \alpha) + \sum_{i=1}^{n} t_{avg\_ni} \tag{6.3}$$

where: s = length of the route in m

v_{avg} = average speed limit over the route in m/s

α = the intensity value and $0 \leq \alpha \leq 1$

n = number of of stops a vehicle has to make

t_{avg_ni} = average time in seconds the vehicle has to be stopped for at stop ni

However, a sharp-sighted reader might notice that an accurate calculation should consider the historical data, thus allowing for more confidence about the calculated value. Unfortunately, there is currently no source that could provide historical data for the simulated environment. Even if there were one, it would not necessarily be accurate one since the parameters of the simulator can be easily customised, affecting how the data is captured. Moreover, the t_o does not include variables determining how long the vehicle would take to reduce the speed when a

stop is encountered and how much time and distance it would cover to reach the speed limit again after the obstacle is passed. The formula implicitly assumes that this metric is handled by the t_{avg_ni} indicating the average time the vehicle takes to pass a probable stop. Additionally, it is worth mentioning that the last bit of the t_o formula, namely the sum of all the stops, also includes the inevitable stops generated by the scenario generation algorithm. As mentioned in Chapter 5 and Chapter 10, in the future, the scenario generation algorithm could be improved to generate individual situations and not only general scenarios. If this was the case, and the algorithm had generated a situation that involves a collision, the average time the driver would take to drive past that collision should be included in the sum of all stops. This is because every detail, affecting the speed of the vehicle is significant and should be taken into account.

In summary, although the research in this project uses the introduced formula for safety score calculation, it serves only as a proposition of what could be used for evaluation. In fact, the formula used to evaluate the software in Chapter 9 is simplified because of the simulator's limitations and poor structuring of the OpenDRIVE maps used. It considers only junctions as inevitable stops that the vehicle has to make and gives them an average waiting time of 12 seconds. This is because of the missing information in the OpenDRIVE map files the CARLA development team provided. An example of this can be seen from Figure C.3 until Figure C.6, in Appendix C that exemplify how road elements in the same map are specified differently and that on average, less than 30% of the road elements have a speed limit specified. In addition, some maps contain information about where the traffic lights are, and others do not include that information. Much more research and proving needs to happen before we can establish the correctness and quality of the safety evaluation function. A significant amount of effort was paid in search of an existing function in the literature that would have already survived the test of time and could be used here. However, no function found in the literature could suit the needs of this project. For this reason, the article aims to be inventive but risk-taking simultaneously, leaving the reader to judge the formula's correctness.

Chapter 7

Obtained data analysis

This chapter will discuss what can be done with the data obtained from the simulations. In Section 7.1, we will introduce the data analysis tool able to generalise data of many participants and extract locations where violations occurred. The Section 7.2 will present two ways the analysed data can be depicted visually, mainly via diagrams and location marking on the map. The chapter is finished in Section 7.3 after introducing a script calculating the safety performance score of a driver in a particular scenario building on the concepts discussed in Section 6.3.

7.1 The data analysis tool

Let us begin by discussing how the data obtained from the simulations could be processed. This article proposes an easily customisable way to perform data analysis that suits the user's needs. An overview of the analysis process can be seen in Figure 7.1.

Here the user specifies the parameters in the analysis_parameters.json file and runs the run_analysis.sh script, which takes the necessary data from the database, performs the analysis and saves the analysed data back to the database in a new folder whose name is specified by the user. An example of what the analysis_parameters.json file looks like can be seen in Figure A.4 in Appendix A. In the file, the user specifies which participants, scenarios and safety metrics must be analysed. The algorithm then performs the analysis and stores the data that can be divided into three categories: the overall performance of participants in the scenarios (stored in participants.xml file), the average scenario scores of the participants (stored in scenarios.xml file) and files with locations of where the violations occurred (stored in separate files in points

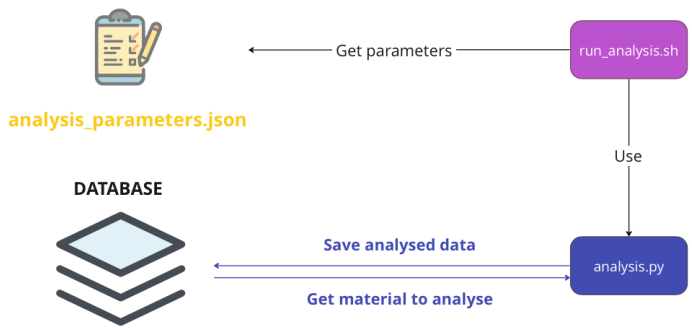

Figure 7.1: An overview of data analysis

directory). To deepen the understanding of the analysed data, the reader is highly encouraged to have a look at an example of data analysis in data/analysed_data/data/example_analysis directory.

7.2 Visualising the data

Now that we have introduced a way of analysing the data and described the basic principles, it is time to see how analysed data can be put in a visual format for further analysis. The overview of the visualisation methods can be seen in Figure 7.2.

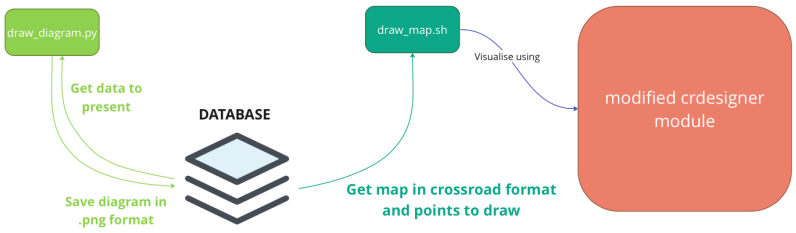

Figure 7.2: Data visualisation process

The last section mentioned that the data analysis tool creates three types of processed data: participant analysis, scenario analysis and metrics analysis. The first two categories can be used to draw diagrams indicating the general tendencies. An example of the diagram can be seen in Figure 7.3. The diagram displays the amount of penalty points each participant received for collisions in scenario 1. The information can help compare participants, including

the AI implementations, in different scenarios. Similar diagrams can be generated to compare the scenario scores concerning different metrics. For instance, to compare which scenarios were more prone to traffic rule violations or breaches of other safety metrics.

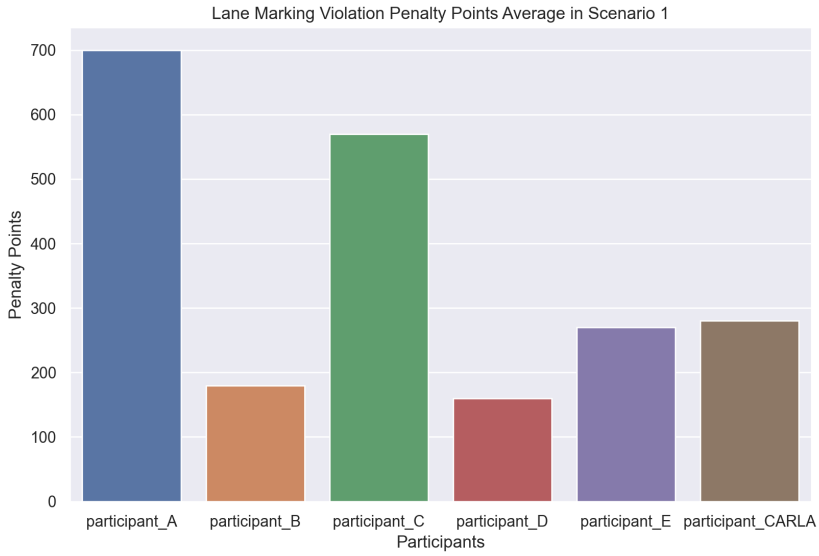

Figure 7.3: An example of a diagram

The third category of analysed data is locations where the violations occurred. Depending on the scenarios, participants and metrics specified in the analysis_parameters.json file, the files in the points directory will differ. Taking a look at an example should clarify the purpose of this type of analysis. Suppose the parameters file specifies that the analysis should be performed on all the participants (listing their names) in scenario 1 and should consider the collisions only. The analysis tool would create a file points/scenario1_collision_data.xml containing all the locations where the collision occurred during the simulation runs of that particular scenario. If we then use the **draw_map.sh** script and specify that file to be pictured, the view similar to the one in Figure 7.4 will appear on the screen.

In this figure, the blue Xs indicate the places where the analysed participants made collisions. Information like this can help identify the places where the vehicle could not react on time or did not react appropriately, and the action resulted in a collision. With the help of the **replayer.sh** script, the simulations can be replayed to analyse the cause of the collision further and thus improve the AI model or simply observe the tendency. It is worth mentioning that the **draw_map.sh** script can take two lists as parameters, depicting the points in red and blue Xs. This can be helpful in comparing the different participants or categories of participants.

As shown in the Figure 7.2, the map drawing process is facilitated by the CommonRoad

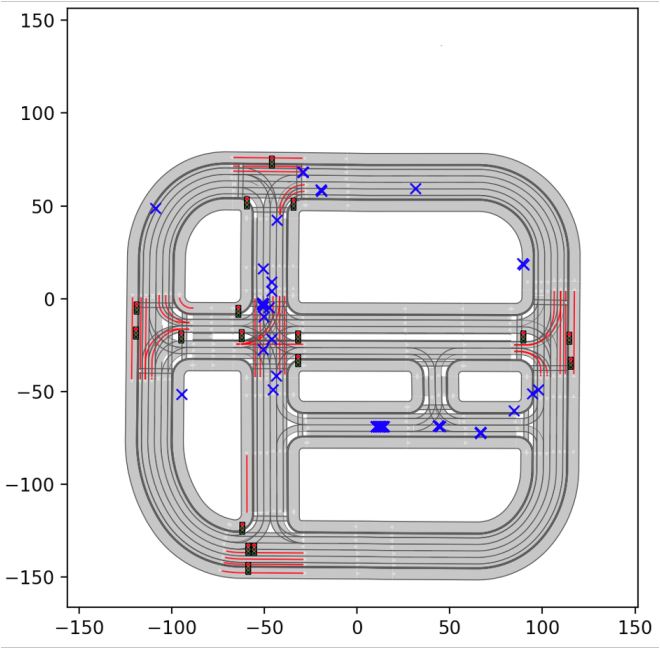

Figure 7.4: An example of a map with points marked

Scenario Designer module [21]. Note that this article uses a modified version of the CommonRoad Scenario Designer. This is due to the fact that the most recent version of it (version 0.6.0) did not have an option to provide a map and a list of points to be represented on the screen when launching the module. It only allows that after the tool was launched, and even then, it only allows for a single point to be marked on the screen. A few modifications were made to the module to launch the visualisation by just running a single command in the terminal. In addition, a bug was fixed not allowing to pass some arguments to the application. The owners of the project were informed about the error found. Because the CommonRoad Scenario Designer uses a different format for drawing the map, the OpenDRIVE files had to be converted to the CommonRoad map specification format. For this, a converter was created that can be found in the data/maps directory. The modified crdesigner module is located in the software/python_libraries directory.

7.3 Calculating the safety score

In this section, we will briefly mention how the safety evaluation formula introduced in Section 6.3 can be used to calculate the safety performance scores of participants in scenario simulations. For that, the **score_calculator.py** script is used, which is located in the software/data_analysis directory. The script takes two arguments: -p participant's name and -s

scenario that the participant drove. Both arguments have to make a path in the data/recordings directory, meaning there must be data about the participant driving that particular scenario. An example of what is given to the user once the score-calculating script is run can be seen in Figure 7.5.

The script gathers data from the database, processes it, applies the formula and prints the score to the terminal, also indicating what is considered a perfect score. More about how to perform the data analysis, visualisation and score calculation is explained in Appendix B where it is demonstrated how to use the project step by step. We will also come back to data analysis and score calculation in Chapter 9 when we analyse how human drivers and a CARLA agent performed in the scenarios.

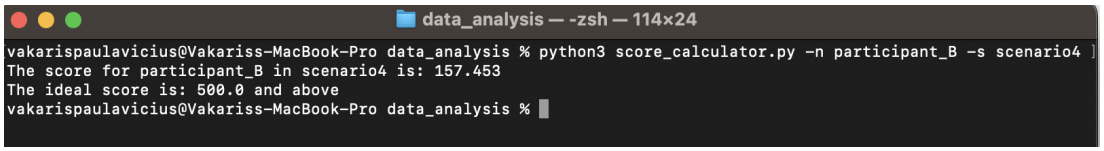

Figure 7.5: A score calculation example

Chapter 8

How the software works

This chapter aims to sum up everything discussed over the last several chapters and give an overview of how the software bundle presented in this article functions as a whole. Note that this chapter does not include instructions on how to run the software. For that, please refer to Appendix B and its sections explaining how to do that step by step.

The process of running simulations and storing the obtained data is managed by the run.py script which delegates the tasks to other scripts until the simulation session is concluded. It is called by the **run.sh** script, which the user must use to start the simulation session. It takes the participant's name as an argument and an optional keyword,"ai", indicating that an AI implementation will drive the scenarios. The run.py script creates a directory for each participant in the data/recordings directory and, within it, creates a directory for each scenario in which the participant will participate. This approach allows for easy access and organisation of the recorded data during analysis.

The scenarios that the vehicle has to drive are listed in the carla_scripts/scenario_list.json file. An example of the scenario list file can be seen in Figure A.5 in Appendix A. Here all the scenarios that the participant is supposed to drive are shown. If the scenario or path is not specified (they are null), the algorithm will automatically generate them during run-time, resulting in a new and unseen route.

For each scenario, a corresponding **scenario_manager.py** script is launched that takes care of the scenario simulation process. It first generates a path and scenario if they are not specified, then changes the map and weather, populates the map, and defines the behaviour of actors on the road. In the beginning, all the simulation actors are immobilised to ensure that each simulation begins with the same conditions for each participant.

Once everything is set up, the hero vehicle is spawned. Depending on if the "ai" keyword was specified when the run.sh script was launched, the scenario manager either launches a manually controlled vehicle or the self-driving one. For that, either driver.py (keyboard or steering wheel) or self_driver.py scripts are called.

If a human driver controls the car, then once the Python application for driving is launched, the driver, in order to start the simulation, needs to press the BACKSPACE key. This causes the Scenario to unfreeze all actors in the simulation, allowing them to move around the world according to their designated behaviour. The simulation also launches the Manager module responsible for all performance calculations and violation marking.

The simulation can end in two ways: reaching the finish line or pressing the BACKSPACE key to abort the simulation. The latter option was added to allow immediate simulation termination if something goes wrong and make it easy to rerun the simulation later (thanks to the flexible structure). Once the simulation finishes, all data and the simulation recording are saved to disk, and the subsequent simulation is launched. The process is repeated until all the scenarios have run from the scenario_list.json file. Note that manual driving can be done either using a keyboard or a steering wheel. The type used can be changed by uncommenting line 158 or line 157 in software/carla_scripts/simulation_manager.py script.

If the AV implementation is driving in the scenarios, everything is happening automatically, and no buttons need to be pressed. Only the run.sh script needs to be run with the participant's name and the keyword "ai".

The simulation process is also illustrated in Figure 8.1, providing a visual overview of how the components act together.

It is crucial to mention that not only the CARLA agent can drive in the simulations but any AV implementation that can be bridged with the CARLA simulator. After the connection to the simulator is complete, all that is needed is to give the waypoints generated by the GlobalRoutePlanner to the agent to follow, name the vehicle "hero" within the simulated environment and start the Manager module when the simulation starts and terminate it once the finish is reached. Everything else works independently, talking to the server directly and receiving the necessary information.

Once the simulations are complete, the user can run the analysis and then work with the analysed data. More about how to use all the software elements can be found in Appendix B.

Several additional Python scripts were written to understand the CARLA world better and help design the scenarios. These include print_coordinates.py, which prints the precise

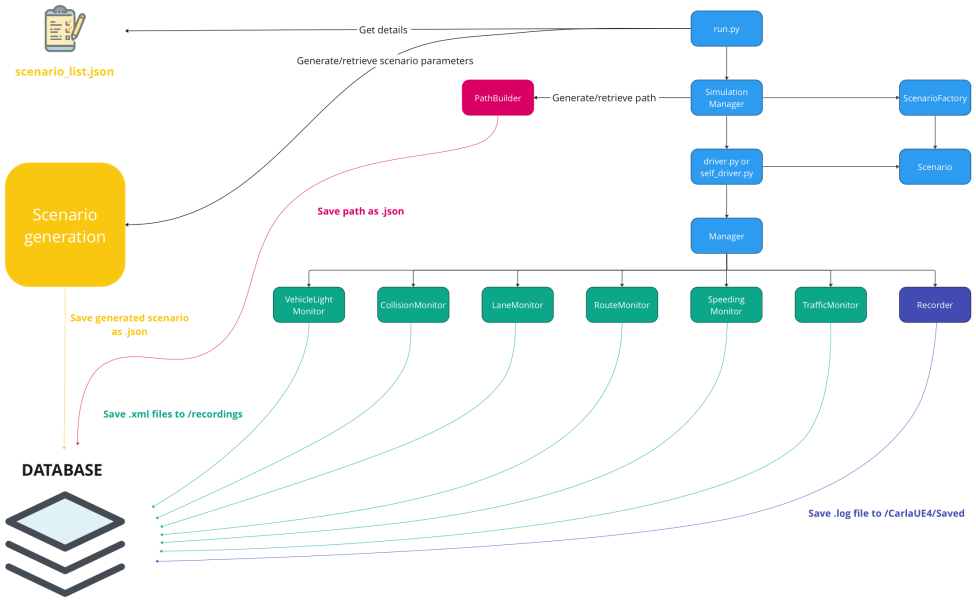

Figure 8.1: An overview of the simulation process

coordinates of the spectator, who can fly anywhere in the world, and draw_lanes_terminal.py, which generates a route when given start and finish coordinates and marks all legal lanes on the map. Other useful scripts can be found in the sub-directories of the software/carla_script directory.

One necessary script to mention is the replay.sh, which enables the replay of any simulation driven by a specific participant on the screen. For example, by executing the command "/replay.sh participant_A 1" in the terminal, the simulation of the first scenario driven by participant_A will be replayed.

Chapter 9

Software testing and evaluation

After introducing the software components proposed by this article for testing vehicles in a simulated environment, we will now look at how the tools can be used in practice. Specifically, we will discuss how the software was tested in a lab with five human participants and an AI agent. We start in Section 9.1 by discussing how the participants were chosen and what had to be done to facilitate their participation in the simulations. Next, in Section 9.2, we will look at how the simulations took place and talk about the things observed during the process. Finally, in Section 9.3, we conclude the chapter by discussing the results of the simulations, comparing the safety performance scores of the human drivers and an AI agent and evaluating how well the proposed system operates.

9.1 Selecting the participants

For this project, it was decided that the best way to test the developed software is by inviting external participants to drive in the generated scenarios, observing how monitors collected data, and running data analysis evaluating their performance.

As this project aims to create a framework for evaluating autonomous vehicle safety, some AI implementations had to be involved in the process. Initially, the leading candidate for participation was the Autoware.AI framework allowing a vehicle to be controlled by the Autoware software in the simulated environment. The reason it was such a good candidate and remains one is that Autoware.AI is a sophisticated software bundle, implementing a level four autonomous agent (discussed in Section 2.3), possessing the ability to participate in traffic. It is also relatively easy to connect it with the CARLA simulator by connecting Autoware.AI and

CARLA using ROS and Autoware bridges provided by the respective developers.

However, due to the complexity of both software systems, the high learning curve and issues not allowing the Autoware.AI agent to follow a set of waypoints, it was decided to use a more straightforward implementation. As the purpose of this project is not to connect different AI agent implementations with the simulator and run them but rather to create a software system for their safety analysis, the choice of the AI participant in the research holds very little significance. For this reason, CARLA BasicAgent was chosen to participate in the software testing.

While still in the initial steps of the project development, it was already established that for the AVs to be considered safe and finally witness the daylight, they have to fit people's safety requirements. It was then that the idea of comparing human drivers and AI agents in the simulated environment made its way onto a piece of paper. It is intriguing how both AI-controlled vehicles and humans would behave in the same situations. For this reason, from the beginning, it was clear that the project needed to be developed to facilitate user testing in the lab.

The first step was to decide how many participants should take part in the research and what prerequisites are needed to allow this process. It was decided that having five people driving in the scenarios would give a good overview of general trends and make for exciting research. As with all projects involving human research, an ethics committee approval was required. For this, an application was submitted, and a positive response came back, allowing the research to occur. The ethical review reference number is LRU-22/23-34770.

Moving on, it is evident that for a participant to reflect the behaviour of a typical driver in the simulated environment, he needs to have had some experience driving on the roads. For this reason, the only requirement for a person to participate in the research was to have a valid driver's license. The licenses were not inspected, but it was assumed that the participants willingly participating in the research were truthful. Five participants were found, and all agreed to take part. Their consent forms are stored safely together with the project files.

While the AI agent only requires the software to drive in the scenarios, the human participants require controllers to operate cars. In order to be as reflective of the real-world conditions as possible, the research had to provide the participants with the tools they are accustomed to when it comes to driving a car: the steering wheel and pedals. For this reason, it was decided to run a controlled experiment in the university lab containing all the necessary hardware.

In addition to the physical tools, software scripts allowing the vehicles to be controlled

by both people and AI were needed. For this, manual driving scripts were borrowed from the example scripts provided by the CARLA development team that can be found in the Carla/PythonAPI/examples directory. The scripts were then modified to suit the purpose of the project. The scrips are named driver_keyboard.py and driver_steeringwheel.py for the human participants and the self_driver.py for the CARLA agent. The scripts can be found in the software/carla_scripts directory.

9.2 Human drivers vs. AI

The research took place in a Bush House computer lab (30 Aldwych, London WC2B 4BG, United Kingdom). The lab contained the steering wheel and pedal-equipped powerful computers allowing the simulations to run at the highest quality graphics. This ensured that the impact of the uncontrolled variables (concerning how people perceive the environment) was minimised. The technical specifications of the computer used can be seen in Figure A.6.

All the participants were tested in five scenarios generated by the scenario generation and path-building algorithms. As discussed in Chapter 3, when we examined the road incident statistics, the generated scenarios must be diverse to reflect various road conditions. For this reason, the five scenarios cover both day and night conditions, rainy and sunny weather, urban and rural areas and highways. Some maps had their start and finish coordinates altered slightly to make the vehicle start or finish in places like a sideline or a parking lot space. This is due to the shortcomings of the path-building algorithm discussed in Chapter 5. More about it will be discussed in the following chapter, where we conclude the project by talking about the issues experienced and possible future improvements.

For the experience to be identical for all the participants, the same vehicle was used throughout all the scenarios' simulations. In addition, randomisation seeds were used with each scenario, guaranteeing identical vehicle behaviour and their arrangement on the map. The scenarios that were driven can be found in the software/carla_scripts/scenario_list.json file. The fields "name" and "details" could be changed to have values pointing to *null*, thus resulting in entirely new and random scenarios being generated. However, as this research aims to get an accurate overview of how people drive, all the scenarios had to be identical for all the participants. That is why randomisation seeds and predefined scenarios were used.

However, ensuring an equal experience for all participants proved to be a challenging endeavour, fraught with various obstacles. To begin with, technical glitches posed a significant problem, such as the steering wheel powering down during simulations or the accelerator pedal

giving impulses as if it was being pushed despite having no input. Issues like these resulted in specific scenarios being rerun. Additionally, unforeseen misbehaviour from certain vehicles within the simulation also had a significant effect, mainly on the scenarios themselves. This includes vehicles flying into the air after a collision and landing somewhere else, causing gridlock on the road and thus impacting the flow of traffic.

As reflected by the participants, although the steering wheel and pedals were there, what was lacking was the feeling of actually sitting in a car and being able to turn the head and observe the environment. This was not well conveyed by the static monitor giving the view of the world only at a specific angle at the time. In addition, the participants said they were missing the noises from the environment as the monitors and the CARLA simulator did not provide any sounds. Some also remarked that the steering wheel was too sensitive, and they did not feel the car well. Having said that, it is safe to say that the fact that the participants were driving the car in a virtual environment had an effect, presumably negative, on their performance.

What was also noted from the simulation runs was that the participants found driving at night more difficult than during bright conditions. Overall, the participants said that they enjoyed testing their driving skills in the simulator and gave positive verbal feedback about the process of the research.

All five participants successfully completed all five predefined scenarios. The data collected from their scenario runs can be found in the directory data/recordings from participant_A until participant_E.

Regarding the CARLA agent participating in the simulations, there were no technical issues with the equipment. However, there were some problems with how the vehicle executed the given path. At some points during a run, the vehicle would make a 360 degrees turn and then continue driving the path. Although the exact cause remains undetermined, it most likely was provoked by the inaccuracies caused by the GlobalRoutePlanner, which created the path. The run of the scenarios by the CARLA agent can be found in the participant_CARLA directory.

9.3 The outcome

In this section, we will evaluate the performance of the safety monitoring tools considering the data they obtained from the scenario executions. In addition, we will utilise the data analysis tools introduced in Chapter 7 to help us assess vehicle safety and provide some insights into where the most issues occurred.

Let us begin by analysing whether the monitoring tools had done their duties. As mentioned previously, the flow of each simulation session is captured in log files found in the subdirectories of the data/recordings directory. The log files are associated with each participant and record how the simulations were fulfilled, whether the monitors successfully recorded data to their intended files, etc. After examining each session_logs.log file, no errors or misbehaviours were spotted, meaning that simulation sessions were completed as planned. After looking at each scenario directory, it was confirmed that all the files were present. Having established that no errors occurred when creating the files and no issues were spotted in the simulation session records, it was time to look at the contents of the recording files to see whether they contained the right data in the correct format.

This process was performed manually because no intelligent software system was developed for checking the correctness of the files.

After observing the files, it was confirmed that the data was recorded following the XML file structure and formatting conventions described in the code. The collision files contained the instances of collisions, where they occurred and at what time; the other files contained their respective violation data. The general accuracy of data was checked by replaying some of the simulations and ensuring that the recorded violations occurred. The majority of the violation data was captured as planned.

However, there were instances that brought inaccurate data, thus affecting the safety score of the participants. While replaying the simulations, it was observed that whenever another car bumped into the participant's car, a collision was recorded, giving the participant penalty points for situations where he was the victim. In addition, there were some instances where a collision with one vehicle was recorded multiple times, giving more penalty points than intended. This can be seen in the collision data of the participant_D in scenario 2.

The first issue with the collision monitor was caused by the simplified implementation of the CollisionSensor, which only considers the instances when another object entered the vehicle's bounding box. It was not distinguishing between the driver running into another object and an object running into the driver. The second issue of the monitor recording the same data multiple times was confirmed to be temporary wrongdoing by the CollisionSensor, as no other instances like this were present. Overall, it is safe to say that the monitors were performing their tasks, although sometimes not doing them right.

Having examined the work of the safety monitors and established that they mostly recorded the right violations, it is now time to use the data analysis tools to analyse the observations

even more and evaluate the performance of each of the participants. For that, the data analysis tool was employed. After specifying all the scenarios, all the metrics and all the participants in the analysis_parameters.json file, the scenario analyser was run and completed the analysis successfully. The analysis can be found in the data/analysed_data/data directory under the name example_analysis. After observing the participants.xml and scenarios.xml file content and manually checking that the values determined by the analyser were correctly calculated, it was confirmed that the analysis tool behaved as expected, producing the correct scores. For example, the sum of all the collision penalty points gathered by the participant_A over all five scenarios is 5750. The analyser correctly tells us that the collision point average of this participant is 1150, given that he drove in five scenarios. The same calculations were manually performed on other metrics to ensure the correctness of the data analysis tool.

Similarly, the point extraction (from the points subdirectory in the analysis folder) was checked to be correct by looking at the coordinates in each file and ensuring they are present in the recording files. This check was performed on a simpler (one participant) analysis because checking for each point in the example_analysis/points would take a very long time. Nevertheless, if the algorithm behaves correctly with few participants, it is safe to assume that it will also behave correctly with many participants.

The visualisation tools were tested using various metrics from the analysed data. Although the diagrams comparing the scores were being drawn correctly, the map drawing tool marking the locations on the map was sometimes accurately reflecting the coordinates and sometimes marking the points where there was no road. The correctly represented points can be seen in Figure A.7, and the incorrectly marked points can be seen in Figure A.8 in Appendix A. Due to the complexity of the CommonRoad Scenario Designer and lack of documentation, it was challenging to determine the root cause of why the points are being represented as they are. For this reason, this issue remains unsolved.

It is now time to move on to the safety evaluation bit. For this, the score_calculator.py script was used, whose purpose was described in Chapter 7. The correctness of the algorithm was verified by running the algorithm to evaluate the performance and then manually recalculating the score and comparing the values. This method proved that the algorithm behaves as expected and accurately applies the formula introduced in Chapter 6.

However, the algorithm is not using the formula to its fullest because of the lack of information retrievable from the simulation environment and the map description files. For instance, there is no straightforward way of retrieving the allowed speed limit in a specific road element

using the CARLA simulator. Instead, they can be retrieved from the road elements specified in the OpenDRIVE map descriptions. However, as explained in Chapter 7, most road elements in the map descriptions do not provide any information about the allowed speed limits. Because of this, a lot of assumptions had to be made. For example, if the speed limit on the road element is missing, the maximum allowed speed is assumed to be 50km/h. This can affect the final score dramatically if the permitted speed is 30km/h, but the algorithm assumes that it is 50km/h, thus calculating the lower optimum time value to complete the route resulting in the driver either speeding to make it to the finish line on time or losing the points for driving too slow. Similar issues occurred with counting the mandatory stops on the map, mainly because there is no way of retrieving the number of stop signs or pedestrian crossings on the road. For this reason, an assumption was made that the vehicle would have to wait an average of 12 seconds at each junction. This value is arbitrary and does not serve as an accurate measure. However, it is impossible to accurately determine such metrics without more data.

Having said all that, the score calculator was used to calculate the scores of all the participants in each scenario. The formula used γ value 0.7, meaning that the collected data about the penalty points is 70% accurate. The highest score achieved was 435.661 by the participant_C in scenario4. The scenario's perfect score, calculated using the formula, was 500 points and above. The ideal score is achieved by completing the route in full, in the exact time given and without doing any violations. If the driver completes the course even quicker than the calculated optimum time, he gets more points. The lowest score achieved was -18979.65 by participant_D in scenario2. The score was so negative because of the issues with the collision monitor mentioned before, where one collision was recorded multiple times resulting in many penalty points given to the driver.

Overall, the scores of all the participants were mostly highly negative because of the large amounts of penalty points given for each penalty. This can be addressed by adjusting the penalty points assigned for each violation. Also, it was observed that the participants also performed lots of violations, such as accidentally crossing solid lane markings and not showing turn indicators when turning or headlights in the dark, adding to significant negative scores. Out of all six participants from the research, the CARLA agent ended up being the fourth in terms of all the penalty points obtained, which is 10237.

Chapter 10

Conclusion

This chapter concludes the discussion of AVs and the introduction of an alternative way of measuring vehicle safety in a simulated environment. In Section 10.1, an overview of the project is given, explaining what has been developed. The following section discusses the research relevance of the project and talks about the attitude with which the project was implemented. In Section 10.3, a list of possible future improvements is provided for people willing to take this project further. In Section 10.4, ethical concerns surrounding the project and the autonomous driving field are considered. The chapter is concluded in Section 10.5, after the author's self-reflection about the challenges faced over the development of this project.

10.1 What has been presented in this article

In the early chapters of this article, it was established that for autonomous vehicles to appear on the streets, their safety needs to be recognised by the general public. However, we soon discovered that traditional ways of testing autonomous vehicles, namely by exploiting them on the roads, would be inefficient given the time needed and the non-deterministic nature of the world. A new and innovative approach was needed - an approach that would look at the problem through another perspective - the simulated one. The article focused on developing a software system allowing for automated vehicle safety assessment in the virtual setting.

Built around the simulator CARLA, the introduced system consists of three main components. The first is an automatic scenario generation algorithm that, with the help of a sophisticated machine learning algorithm and a path-building script, can create diverse traffic scenarios. Generated scenarios allowed for a vehicle to be tested on the roads and its actions

recorded by the safety monitoring system, which is the second component. The system is responsible for observing the vehicle's actions and transmitting the data to its monitors that record the violations in the database. The data analysis tool then handles the recorded data by generalising it and preparing it for visualisations using diagrams and 2D map representations, allowing for further analysis of behaviour patterns. In addition, an equation for evaluating the vehicle's performance in a given scenario was developed that allows for a simulation run to be allocated a safety performance score. Moreover, besides all the technical details involving the development of the system, the article also discussed in detail how the ideas of autonomous vehicles matured throughout human history, compared the AI behaviour with that of the human drivers, tested the proposed software system with participants and evaluated how well the system works in practice.

10.2 Future improvements

As with any bigger project developed in a limited amount of time and with limited resources, there is always a place for improvement. This section aims to give some guidelines about what could be improved and in what direction.

Let us start with the scenario generation algorithm. As mentioned in the Section 5.4, the algorithm could be upgraded in several ways. Firstly, a set of real-world data could be used to have a more accurately trained algorithm whose generated scenarios would be more reflective of the real-world conditions. In addition, a more sophisticated algorithm could be used able to create even more diverse and unique situations, allowing for the automatic generation of other vehicle behaviour. For instance, generating situations where an actor in the simulation acts according to some patterns when the participant vehicle is close (suddenly changing lanes, for instance). However, the current implementation is only able to create general scenarios where the behaviour is set to all the vehicles with the hope that some interesting and unusual traffic conditions will make their way to the surface. What would also be good to have is a system able to analyse how challenging and diverse the scenarios actually are.

The path generation algorithm also has places for improvement, starting with how the paths are built. At the moment, the algorithm creates paths by trial and error, randomly choosing the starting element and then proceeding from there. It would be much better if it did that in a structured way, reasoning about why the path has to look like that and relating it to the generated scenario. This would allow for even more exciting paths and behaviour patterns to be built.

In terms of the safety monitoring system and the data analysis tools, they currently do not provide a way to use the observations for model improvement. They are there to record the violations and then look at the general behaviour patterns without linking the observations with some mistakes in the model. This, in addition to the improvements to the collision sensor (mentioned in Section 9.3, should be addressed in future updates.

Moreover, the current software does not provide automated tests to check the components' validity and correctness. As with autonomous vehicles, the systems measuring their safety must also be flawless. For that, a comprehensive test suite should be developed, ensuring that all components work as intended and that updates to one do not break anything in another.

10.3 Research relevance and project's integrity

This article aims to contribute to the field of autonomous driving by presenting a method of testing driverless cars in realistic traffic situations. As highlighted earlier in the paper, demonstrating the safety and reliability of autonomous vehicles to the general public is vital in bringing all the hard work in the field to reality. This statement is supported by other literature as well [22]. Every little step in this direction is significant, and by raising people's curiosity and awareness about the latest developments, we are getting closer to achieving this goal. By introducing this project, we hope to grab the attention of young people and inspire their engagement in this field, emphasizing that it can be both enjoyable and fruitful.

In addition to contributing to the field of autonomous driving, the project aimed to leverage existing open-source projects. By doing so, the project aimed to both benefit from existing systems and introduce innovative ways of utilizing these tools while also showing appreciation for other software developers and their creations. This article expresses gratitude towards the projects mentioned, namely the CARLA simulator and the CommonRoad Scenario Designer.

All parties involved in the project's development were aware of the BCS Code of Conduct[23] and always sought to follow the rules. That is why the project was developed with utmost care and honesty, first learning and understanding the problem and only then trying to accomplish the goals. Moreover, the research involving human participants was carried out in a professional and respectful manner resulting in mutual professional and personal growth.

10.4 Ethical concerns

The field of autonomous driving has always been concerned with ethical questions, as the introduction of such technology would have a significant impact on the lives of many people. Thus, the intentions of engineers need to be carefully considered, and the possible implications rationally weighted. There is no room for doubt regarding the well-being of living beings. Therefore, it is essential to thoroughly test the underlying technology and hardware to ensure they meet the necessary standards. As the use of artificial intelligence in the field is inevitable, proper precautions must be taken. The algorithms cannot be biased, resulting in harm to some groups of people because of the lack of training data or biases in the training data. The systems must be developed with the intention of producing rational and unbiased decisions resulting in the best possible outcome for all parties.

Although this project did not extensively explore the ethical implications of autonomous driving, it did take steps to prevent biased outcomes in the proposed scenario generation method. The article aimed to create diverse scenarios for vehicle safety testing to get comprehensive analysis as discussed in Chapter 2 and Chapter 5. Additionally, it was assumed that CARLA's set of models for vehicles and pedestrians was diverse; thus, the algorithm randomly selected models without any prejudice. While these steps may not have addressed all potential biases, they demonstrate a commitment to ethical considerations in the field. This project is in favour of ethical and unbiased project development.

10.5 Author's self-reflection

The purpose of this section is for me, as the author of this paper, to reflect on my journey and the work produced.

Doing this project allowed me to better understand the complexities surrounding autonomous driving and provided me with hands-on experience contributing to the field. The journey was challenging and full of unexpected changes and issues, constantly pushing me towards upgrading the project and looking for new ways of achieving my goals.

Having minimal programming experience in Python and no experience in machine learning, coming up with an automated way of generating the scenarios proved to be particularly challenging. The journey began with studying the machine learning methods on my own, then moved to implementations of various classification and regression algorithms and finally to applying the knowledge in solving the scenario generation in this project. Although this project

presents a simple and not the best way of generating the scenarios, attempts to have a more sophisticated algorithm guided by deep neural networks were made. However, the method was not pursued because of the lack of suitable training data and a deep understanding of the CARLA simulator and the advanced machine learning methodologies. In addition, attempts were made to access some global data sources (like the SafetyPool repository[24]) containing scenarios that could be used as training examples. Unfortunately, several tries to get access did not bring any results.

Despite having many technical issues with my computer, which was barely running the simulations at the lowest quality graphics, I managed to successfully conclude my project and develop a prototype software system for vehicle safety evaluation in a virtual environment. I wish the university had provided us with either more lab time or the right equipment to engage with the project fully. Nevertheless, I enjoyed doing this project, and the challenges faced did not discourage but motivated me to move forward and engage in problem-solving.

I believe that the prototype introduced in this article has the potential to become a tool that would be helpful to the industry. However, much more work must be done for it to be effective and trustworthy, and it should be pursued in a team of developers rather than by a single person. Looking back, I think I have taken a bite too big to chew and decided to create an extensive and comprehensive system. I should have chosen one specific aspect, for instance, scenario generation, and focused on doing that only. The end result would have been a well-constructed artefact instead of a broad but less well-implemented system. The possible ways in which the system could be continued were expressed in the previous section.

The journey also taught me about time planning and work organisation. It also showed me that developing software in groups is usually easier than working entirely independently.

In the end, I feel that I have learned many new skills, such as implementing my own machine-learning algorithms, writing shell scripts, performing multi-threading in Python, working with JSON and XML files, analysing data, automating processes and writing academic texts with LaTeX. I also realised that I need to improve my software engineering skills and deepen my knowledge of artificial intelligence to be a better expert in the future and make the field of IT even more influential.

Bibliography

[1] Jong Kyu Choi and Yong Gu Ji. Investigating the importance of trust on adopting an autonomous vehicle. *International Journal of Human-Computer Interaction*, 31(10):692–702, 2015.

[2] Rachael Brennan, Logan Sachon. Self-driving cars make 76% of Americans feel less safe on the road, Sep. 14, 2022 [Online].

[3] J Fuller. Did davinci really sketch a primitive version of the car? *HowStuffWorks. com*, page 1, 2008.

[4] C Engelking. The 'driverless' car era began more than 90 years ago. *Retrieved February*, 20:2019, 2017.

[5] T. Jochem, D. Pomerleau, B. Kumar, and J. Armstrong. Pans: a portable navigation platform. In *Proceedings of the Intelligent Vehicles '95. Symposium*, pages 107–112, 1995.

[6] Twi global, 2022. Accessed on March 15, 2023.

[7] Jelena Kocić, Nenad Jovičić, and Vujo Drndarević. Sensors and sensor fusion in autonomous vehicles. In *2018 26th Telecommunications Forum (TELFOR)*, pages 420–425, 2018.

[8] Jorge Vargas, Suleiman Alsweiss, Onur Toker, Rahul Razdan, and Joshua Santos. An overview of autonomous vehicles sensors and their vulnerability to weather conditions. *Sensors*, 21(16):5397, 2021.

[9] Sae levels of driving automation refined for clarity and international audience, 2021. Accessed on April 4, 2023.

[10] Laura Fraade-Blanar, Marjory S Blumenthal, James M Anderson, and Nidhi Kalra. *Measuring automated vehicle safety: Forging a framework*. 2018.

[11] European commission Directorate-General for Mobility and Transport. *Road safety in the EU: fatalities in 2021 remain well below pre-pandemic level*, 3 2022.

[12] National Highway Traffic Safety Administration. *Traffic Safety Facts Annual Report Tables*, 6 2022.

[13] Morela Hernandez. The impossibility of focusing on two things at once. *MIT Sloan*, 2018.

[14] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator. In *Conference on robot learning*, pages 1–16. PMLR, 2017.

[15] Ruxin Li and Runping Zhai. Estimation and analysis of minimum traveling distance in self-driving vehicle to prove their safety on road test. In *Journal of Physics: Conference Series*, volume 1168, page 032101. IOP Publishing, 2019.

[16] Mastercard and network international launch new artificial intelligence fraud-prevention solution, 2023. Accessed on April 4, 2023.

[17] The power of data: How paypal leverages machine learning to tackle fraud, 2021. Accessed on April 4, 2023.

[18] Robin van der Made, Martijn Tideman, Ulrich Lages, Roman Katz, and Martin Spencer. Automated generation of virtual driving scenarios from test drive data. In *24th International Technical Conference on the Enhanced Safety of Vehicles (ESV) National Highway Traffic Safety Administration*, number 15-0268, 2015.

[19] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[20] Marius Dupuis, Martin Strobl, and Hans Grezlikowski. Opendrive 2010 and beyond–status and future of the de facto standard for the description of road networks. In *Proc. of the Driving Simulation Conference Europe*, pages 231–242, 2010.

[21] Sebastian Maierhofer, Moritz Klischat, and Matthias Althoff. Commonroad scenario designer: An open-source toolbox for map conversion and scenario creation for autonomous vehicles. In *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*, pages 3176–3182. IEEE, 2021.

[22] Tingru Zhang, Da Tao, Xingda Qu, Xiaoyan Zhang, Rui Lin, and Wei Zhang. The roles of initial trust and perceived risk in public's acceptance of automated vehicles. *Transportation research part C: emerging technologies*, 98:207–220, 2019.

[23] Bcs code of conduct. Accessed on April 4, 2023.

[24] Safety pool™ scenario database, 2022. Accessed on April 4, 2023.

[25] Lucas Caton. code2pdf - a simple tool for convert your source code to a pdf file, 2011. Accessed on April 4, 2023.

Appendix A

Extra Information

This appendix contains tables, various images of files, and maps to better illustrate the topics discussed throughout the report.

Attribute	Min Value	Max Value	Rounding Value
cloudiness	0	100	3
precipitation	0	100	3
precipitation deposits	0	100	3
wind intensity	0	100	3
sun azimuth angle	0	360	3
sun altitude angle	-90	90	3
fog density	0	100	3
fog distance	0	$+\infty$	3
wetness	0	100	3
fog falloff	0	$+\infty$	3
scattering intensity	0	$+\infty$	3
mie scattering scale	0	$+\infty$	3
rayleigh scattering scale	0	$+\infty$	3
dust storm	0	100	3

Table A.1: Weather related scenario attributes

Attribute	Min Value	Max Value	Rounding Value
number of pedestrians	0	$+\infty$	0
number of vehicles	0	$+\infty$	0
number of two wheel vehicles	0	$+\infty$	0
proportion of speeding vehicles	0	1	3
proportion of vehicles without lights	0	1	3
proportion of light ignoring vehicles	0	1	3
light ignoring percent	0	100	3
proportion of sign ignoring vehicles	0	1	3
sign ignoring percent	0	100	3
proportion of vehicle ignoring vehicles	0	1	3
vehicle ignoring percent	0	100	3
proportion of walker ignoring vehicles	0	1	3
walker ignoring percent	0	100	3
proportion of keeping right vehicles	0	1	3
keeping right percent	0	100	3
proportion of lane changing vehicles	0	1	3
lane change percent	0	100	3
proportion of misbehaving pedestrians	0	1	3
proportion of running pedestrians	0	1	3
proportion of road crossing pedestrians	0	1	3
number of junctions	0	$+\infty$	0
distance in metres	0	$+\infty$	3
area in square metres	0	$+\infty$	3
difficulty	0	1000	3

Table A.2: Traffic related scenario attributes

Category	Not speeding	Speeding
No beams and no fog lights	50	50
No beams	30	30
No fog lights	10	10
Hit pedestrian	600	1200
Hit vehicle	250	500
Hit two-wheel vehicle	400	800
Hit road object	150	300
Red light violation	50	100
Stop sign/marking violation	40	80
Solid lane marking	20	60
Double solid lane marking	40	100
Broken lane marking no turn indicator	10	30
Light speeding	1	1
Heavy speeding	3s	3

Table A.3: Penalty score table

```
{
    "number_of_pedestrians": 17,
    "number_of_vehicles": 47,
    "number_of_two_wheel_vehicles": 8,
    "proportion_of_speeding_vehicles": 0.22,
    "proportion_of_vehicles_without_lights": 0.24,
    "proportion_of_light_ignoring_vehicles": 0.2,
    "light_ignoring_percent": 26.569,
    "proportion_of_sign_ignoring_vehicles": 0.187,
    "sign_ignoring_percent": 28.303,
    "proportion_of_vehicle_ignoring_vehicles": 0.216,
    "vehicle_ignoring_percent": 15.7,
    "proportion_of_walker_ignoring_vehicles": 0.253,
    "walker_ignoring_percent": 16.869,
    "proportion_of_keeping_right_vehicles": 0.187,
    "keeping_right_percent": 31.601,
    "proportion_of_lane_changing_vehicles": 0.288,
    "lane_change_percent": 24.191,
    "proportion_of_misbehaving_pedestrians": 0.251,
    "proportion_of_running_pedestrians": 0.284,
    "proportion_of_road_crossing_pedestrians": 0.328,
    "number_of_junctions": 8,
    "distance_in_metres": 802.702,
    "area_in_square_metres": 3375170.561,
    "cloudiness": 20.472,
    "precipitation": 2.689,
    "precipitation_deposits": 0.0,
    "wind_intensity": 24.422,
    "sun_azimuth_angle": 313.767,
    "fog_density": 5.306,
    "fog_distance": 0.0,
    "fog_falloff": 0.101,
    "scattering_intensity": 1.047,
    "mie_scattering_scale": 0.03,
    "rayleigh_scattering_scale": 0.038,
    "dust_storm": 0.0,
    "difficulty": 300.0,
    "wetness": 0.1,
    "sun_altitude_angle": 67.0
}
```


Figure A.1: Example of a generated scenario

```
{
  "path_checkpoints": [
    {
      "x": 112.867,
      "y": -14.310,
      "z": 0.400
    },
    {
      "x": -52.180,
      "y": -6.970,
      "z": 0.400
    },
    {
      "x": 40.692,
      "y": 51.025,
      "z": 0.400
    },
    {
      "x": 43.639,
      "y": 130.790,
      "z": 0.400
    },
    {
      "x": -103.212,
      "y": -14.857,
      "z": 0.400
    }
  ],
  "start_location": {
      "x": 112.867,
      "y": -14.310,
      "z": 0.400,
      "yaw": -90
  },
  "town": "Town10HD_Opt"
}
```


Figure A.2: Example of a generated path

```xml
<?xml version="1.0" ?>
<CollisionData>
  <PenaltyPointsTotal>700</PenaltyPointsTotal>
  <NumberOfCollisionInstances>3</NumberOfCollisionInstances>
  <CollisionInstances>
    <Instance>
      <PenaltyPoints>250</PenaltyPoints>
      <Description>Hit vehicle</Description>
      <CollisionIntensity>12786.265512536726</CollisionIntensity>
      <Time>37.281s</Time>
      <Location>
        <X>-50.800228</X>
        <Y>16.336308</Y>
        <Z>-0.000915</Z>
      </Location>
    </Instance>
    <Instance>
      <PenaltyPoints>300</PenaltyPoints>
      <Description>Hit road object speeding</Description>
      <CollisionIntensity>3978.5656853911505</CollisionIntensity>
      <Time>53.346s</Time>
      <Location>
        <X>-18.755812</X>
        <Y>23.543898</Y>
        <Z>0.027082</Z>
      </Location>
    </Instance>
    <Instance>
      <PenaltyPoints>150</PenaltyPoints>
      <Description>Hit road object</Description>
      <CollisionIntensity>20890.64664324663</CollisionIntensity>
      <Time>67.504s</Time>
      <Location>
        <X>31.434811</X>
        <Y>59.229317</Y>
        <Z>0.154669</Z>
      </Location>
    </Instance>
  </CollisionInstances>
</CollisionData>

```

Figure A.3: An example of collision_data.xml file structure

```json
{
    "participants": [
        "participant_A",
        "participant_B",
        "participant_C",
        "participant_D",
        "participant_E",
        "participant_CARLA"
    ],
    "scenarios": [
        "scenario1"
    ],
    "metrics": [
        "collision_data",
        "lane_marking_violation_data",
        "vehicle_light_misuse_data",
        "route_data",
        "speeding_data",
        "road_traffic_violation_data"
    ]
}
```

Figure A.4: An example of analysis_parameters.json file

```json
{
    "scenarios": [
        {
            "name": "scenario1",
            "details": "path1",
            "vehicle": "vehicle.mercedes.coupe_2020",
            "randomization_seed": 74111222
        },
        {
            "name": "scenario2",
            "details": "path2",
            "vehicle": "vehicle.mercedes.coupe_2020",
            "randomization_seed": 23115235
        },
        {
            "name": "scenario3",
            "details": "path3",
            "vehicle": "vehicle.mercedes.coupe_2020",
            "randomization_seed": 57894561
        },
        {
            "name": "scenario4",
            "details": "path4",
            "vehicle": "vehicle.mercedes.coupe_2020",
            "randomization_seed": 34586451
        },
        {
            "name": "scenario5",
            "details": "path5",
            "vehicle": "vehicle.mercedes.coupe_2020",
            "randomization_seed": 75861238
        }
    ]
}
```


Figure A.5: An example of the scenario list

OS: Ubuntu 20.04.4 LTS
CPU: 11th Gen Intel Core i7-11700KF @ 3.60GHz × 16
RAM: 64GB
GPU: NVIDIA GeForce RTX 3090

Figure A.6: Technical specifications of the lab computer

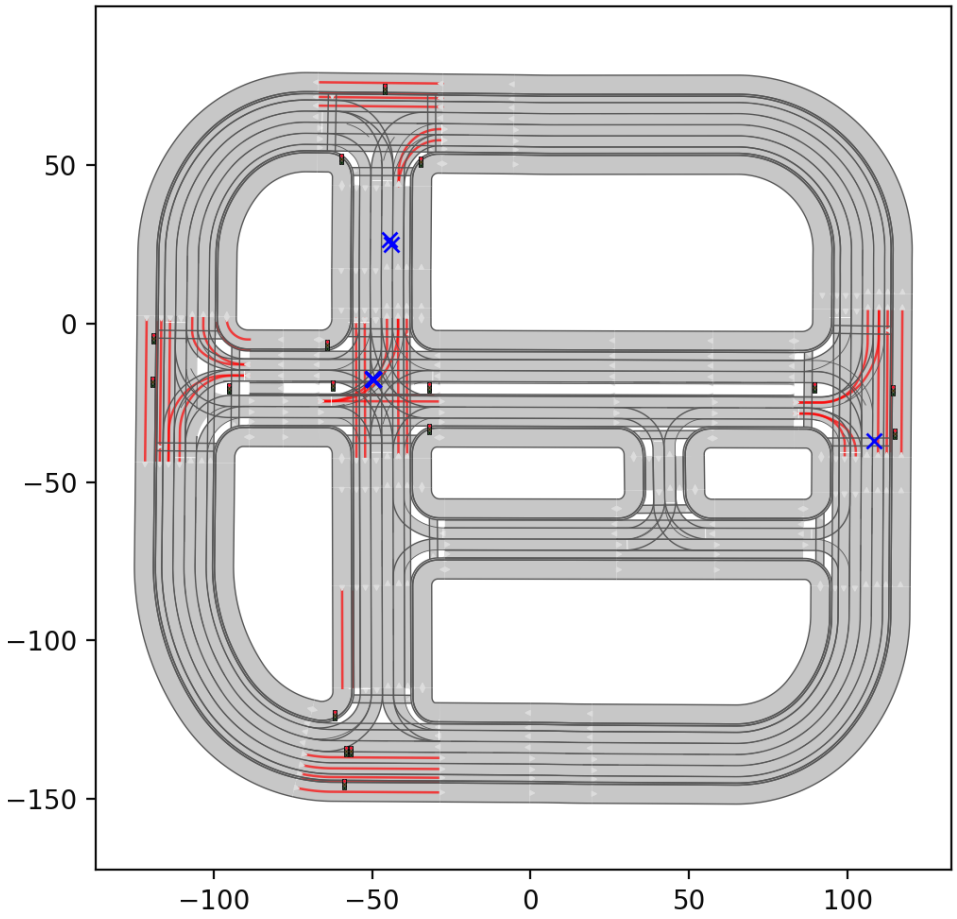

Figure A.7: Correctly marked collision locations

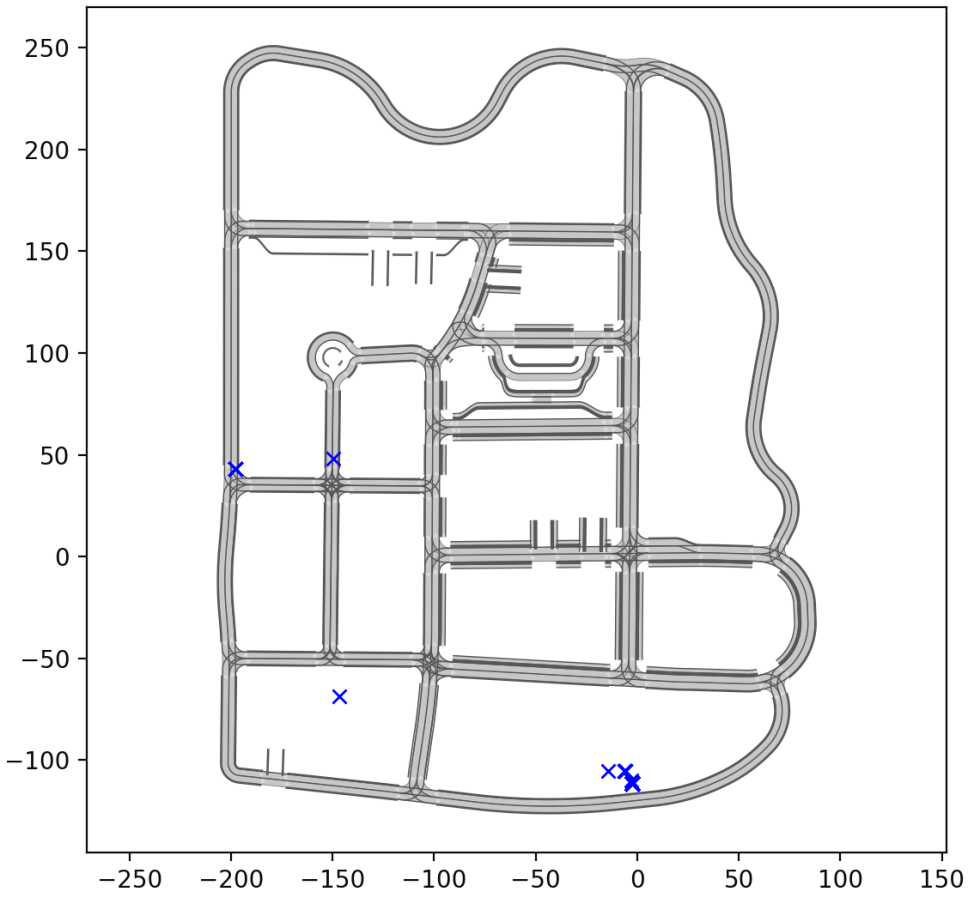

Figure A.8: Incorrectly marked collision locations

Appendix B

User Guide

The purpose of this chapter is to provide a user guide on how to use the software proposed in this article. Each section provides instructions for different subcomponents of the framework. Note that the required python modules must be installed before running any of the software. They can be found in the software/python_libraries/requirements.txt file. These dependencies can be installed by creating a virtual environment at the root of this project and running the command:

$$pip3\ install\ -r\ requirements.txt$$

Please note that most directories also contain README.md files with instructions and descriptions for using the software.

B.1 How to generate scenarios

This section gives a brief overview of how to generate the scenarios. The scenario generation script can be found in the software/generating_scenarios directory and is named **create_scenario.py**. It takes the following optional arguments:

- -d the difficulty of the scenario ($0 \leq d \leq 1000$)
- -s the sun altitude ($-90 \leq s \leq 90$), where 90 means that the sun is at its peak and -90 means that it is an absolute night
- -w the weather wetness level in the scenario ($0 \leq w \leq 100$)
- -f the name of the file to save the scenario to (without the .json ending)

- -r flag meaning that the algorithm should be trained using a randomisation seed (the same every time)

An example of the command in the terminal can be seen in the image below:

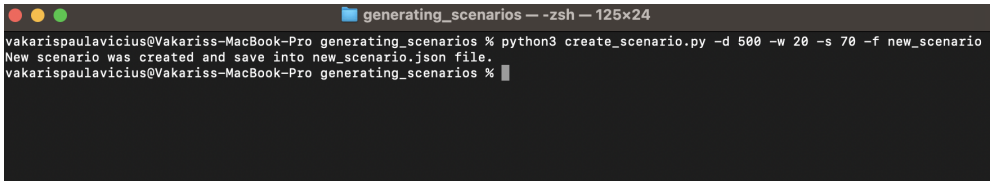

Figure B.1: How to generate a scenario

The following command would generate a new scenario in a new file **new_scenario.json** and save the file to the data/scenario_generation_data/generated_scenarios directory. The file can be later used to create a specific path, or it could be used as a predefined scenario for the simulations.

B.2 How to build paths

This section briefly overviews how a script that builds paths can be run. The script itself is located in the software/carla_scripts directory. Its name is **path_generator.py**. It takes the following mandatory arguments:

- -f the name of the file to save the built path to (without the .json ending)
- -s the name of the scenario to be referenced when designing the path (without the .json ending)

An example of the command in the terminal can be seen in the image below:

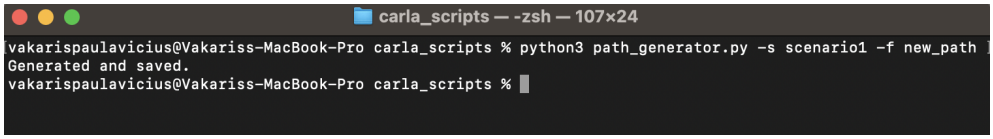

Figure B.2: How to build a path

The following command would build a new path in the file **new_path.json** and save the file to the data/paths directory. The file can be later used as a predefined path for scenarios or inspected on the map using helper scripts, for instance, draw_lanes_terminal.py.

B.3 How to run the simulations

This section presents a simple way of running the scenarios. At first, the user has to decide how many scenarios he wants the software to run and whether the scenarios should be predefined or created at run-time. If the user decides to use predefined scenarios, he must describe that in the scenario_list.json file in the software/carla_scripts directory. An example of the scenario list can be seen in Figure A.5. If the user wants the path and the scenario to be generated at run-time, the attributes "name" and "details" of each scenario element in the scenario list file should be set to *null*. The scenario simulation process requires the CARLA simulator to be running. Once the CARLA simulator is running and accessible, the user needs to navigate to the software/carla_scripts directory in the second terminal and run the following command to run the simulations with manual control:

$$/run.sh\ [name]$$

In order to change whether the steering wheel is to be used or the keyboard, the user needs to comment and uncomment lines 157 and 158 in scenario_manager.py, respectively.

The element [name] needs to be replaced by the name given to the participant. All the recorded data will be stored in the data/recordings/[name] directory, and the details of the simulations will be visible in the data/recordings/[name]/session_logs.log file. In order to launch the Carla agent, the following command has to be run:

$$/run.sh\ [name]\ ai$$

The keyword "ai" (at the end of the line) differentiates manually controlled and self-driving cars. **Important: if at any point the program crashes and the cause is unknown, the parts "> /dev/null 2 > &1" need to be commented out on lines 26 and 33 in the run.sh script in addition to line 160 in the scenario_manager.py script. This allows for message output to the terminal.** All the errors should be handled by the try-except blocks, and the appropriate messages should be written to the session_logs.log files. The steps mentioned before work as a plan b if something goes wrong.

B.4 How to perform analysis

Two types of analysis can be performed. Let us start with the analysis method meant to assist in the score calculation process called scenario analysis. The script can be found in the software/data_analysis directory and is called scenario_analyser.py. It aims to analyse both the

scenario and the path and extract information such as the average allowed speed on the path, the number of pedestrian crossings and other. Note that in order to run this script, the CARLA simulator must be running. This is needed because the script needs to use the built-in functions of CARLA and access the map attributes. The script takes two mandatory arguments:

- -p the name of the path (without the .json ending)
- -s the name of the scenario (without the .json ending)

The following command is used to perform the scenario analysis:

$$python3\ scenario\_analyser.py\ -s\ scenario1\ -p\ path1$$

As a result, it produces a file named the same as the scenario given as the argument. It saves this file into the data/simulation_details directory. This file is later used by the score calculator discussed in Section B.6

The second analysis tool is responsible for analysing the recordings of the scenario runs. It synthesises and processes information that is later used by the visualisation tools. The data analysis tool can be found in the software/data_analysis directory and is named **run_analysis.sh**. It takes a name as an argument and processes the recordings according to the specifications in the analysis_parameters.json file. How the parameters file looks can be seen in Figure A.4. It creates a new directory in data/analysed_data/data with the name provided as the argument and stores the analysed data there.

The following command can be used to run an example analysis:

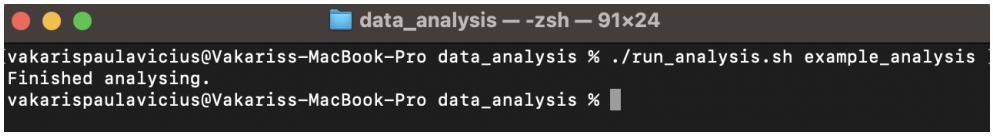

Figure B.3: How to perform data analysis

B.5 How to draw diagrams and maps

This section discusses how to visualise the analysed data. The visualisations can be done in two ways: drawing the diagrams or visualising interesting points on the map. Let us look at the diagram drawing first. The diagram drawing script can be found in the software/-data_analysis/visualisation/diagram_drawing directory. In order to make matters easier, an example file is provided in the directory, and it is called draw_diagram_example.sh. The comments in the script show which places should be edited to draw different diagrams. More about

the customisations of it was discussed in Section 7.2 when we talked about how the visualisation of data is performed. The following command would produce an image from Figure 7.3.

./draw_diagram_example.sh

For the map visualisation, the crdesgner module is required. If all the modules from the requirements.txt file were installed successfully, as mentioned at the beginning of this chapter, then it should already be present in the system. However, this project modified the module to better fit the project's needs. In order to perform the map visualisation, the modified version from software/python_libraries/crdesigner needs to be used. Simply replace the package in the system with the one provided in this project's repository.

As with the diagram drawing, an example script is provided in the software/data_analysis/visualisation/map_drawing directory. Its name is draw_map_example.sh. In order to draw a different map or a different list of points, the script's code should be edited accordingly. The following command should produce a view similar to the one in Figure 7.4.

./draw_map_example.sh

Note that if maps other than the ones provided by CARLA are used in the simulations, their OpenDRIVE description files must be added to the data/maps/opendrive_format directory. For the CommonRoad Scenario Designer to depict them, they must be converted from OpenDRIVE to CommonRoad format. It is done by the xml_converter.py script present in the data/maps directory.

B.6 How to calculate the score

This section concludes the user guide by examining how the participant's performance in a specific scenario can be evaluated. It is done by the score_calculator.py script present in the software/data_analysis directory. The script takes two mandatory arguments:

- -n the name of the participant
- -s the name of the scenario (without the .json ending)

It then looks at the files in data/recordings/[participant's name]/[scenario], extracts the necessary information and calculates the score printing it to the terminal. The following is an example of how this is done:

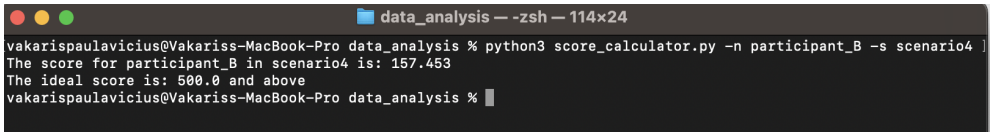

Figure B.4: How to evaluate the performance

In addition to calculating the score and printing it to the terminal, the script also provides the ideal score for that particular scenario. The scores are calculated based on the formula introduced in Section 6.3.

Appendix C

Issues

This appendix contains various images of files and terminal output to better illustrate the issues encountered. The following figures are referenced in the report.

```
<road name="Road 31" length="9.4456216548435208e+0" id="31" junction="-1">
    <link>
        <predecessor elementType="junction" elementId="1162"/>
        <successor elementType="road" elementId="41" contactPoint="end"/>
    </link>
```

Figure C.1: Road 31 pointing to road 41

```
<road name="Road 41" length="1.0000000000000000e+0" id="41" junction="-1">
    <link>
        <predecessor elementType="junction" elementId="798"/>
        <successor elementType="road" elementId="31" contactPoint="end"/>
    </link>
```

Figure C.2: Road 41 pointing back to road 31 creating a loop

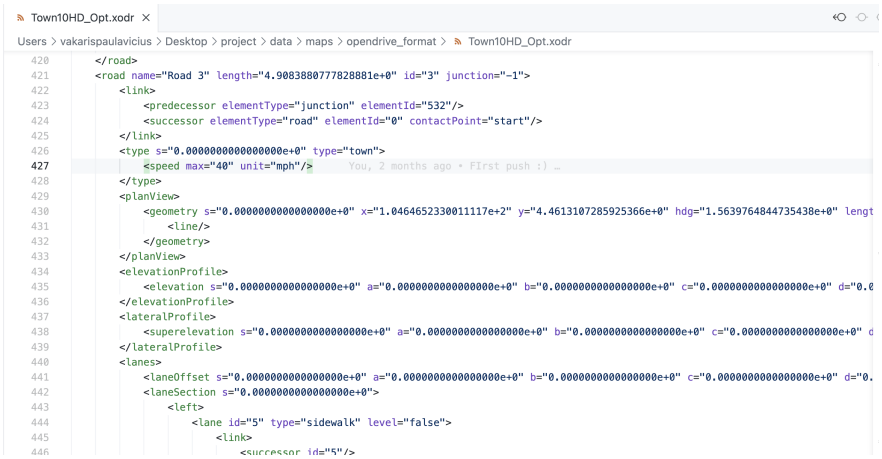

Figure C.3: The road element specifies the speed limit (Town10HD_Opt)

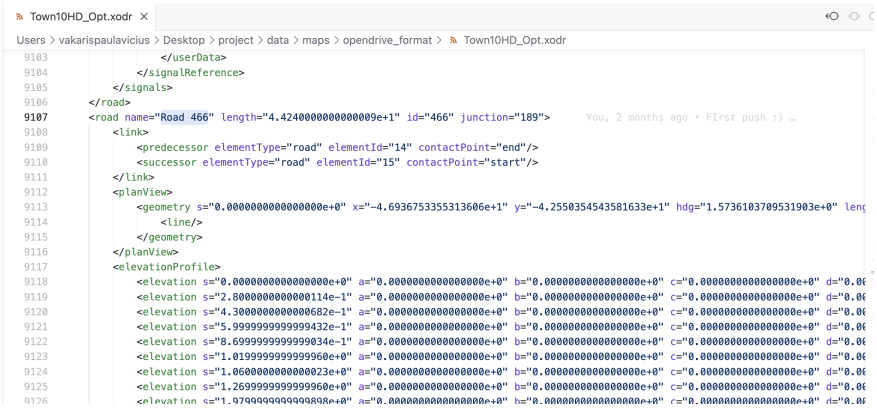

Figure C.4: The road element does not specify the speed limit (Town10HD_Opt)

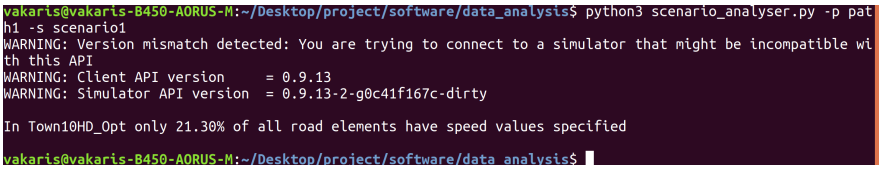

Figure C.5: The proportion of road elements indicating speed limit (Town10HD_Opt)

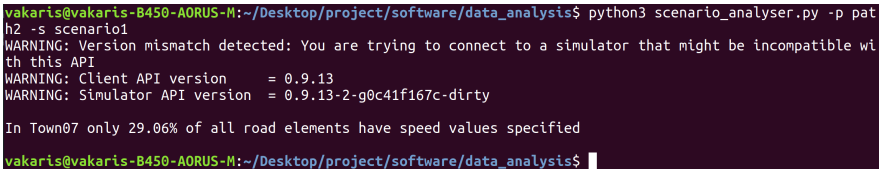

Figure C.6: The proportion of road elements indicating speed limit (Town07)

Appendix D

Source Code

The source code PDF was generated using an open-source tool called code2pdf developed by Lucas Caton [25]. The data files and the CommonRoad Scenario Designer module were omitted. However, they can be found in the source code zip. The whole project is also stored on GitHub. It can be accessed using the following URL: `https://github.com/pvakaris/av_safety_evaluation_project`. The repository is left private to avoid plagiarism and other misuses. The code used in the video demonstration was written before April 6, 2023.

File: ./software/data_analysis/analysis.py

```python
import argparse
from classes.parsing import Parser
import json

parameters_file = "analysis_parameters.json"

argparser = argparse.ArgumentParser(description=__doc__)
argparser.add_argument(
    '--save_location', '-s',
    required=True,
    help='Where to save the analysis')
args = argparser.parse_args()

def main():
    try:
        parameters = read_parameters()
        parser = Parser(parameters, args.save_location)
        try:
            parser.parse()
        except Exception as e:
            print("An exception occured parsing the recording files: ", e)
        try:
            parser.save()
        except Exception as e:
            print("An exception occured saving the analysed data: ", e)
    finally:
        print("Finished analysing.")

def read_parameters():
    try:
        with open("analysis_parameters.json", "r") as file:
            parameters = json.load(file)
        return parameters
    except Exception as e:
        print("Something wrong happened trying to get data from {}".format(parameters_file), e)

if __name__ == '__main__':
    main()
```

File: ./software/data_analysis/classes/parsing.py

```python
from classes.data_analysers import CollisionDataAnalyser, VehicleLightDataAnalyser, RouteDataAnalyser, LaneMarkingDataAnalyser, SpeedingDataAnalyser, RoadTrafficD
import xml.etree.ElementTree as ET
from xml.dom.minidom import parseString

# Creates a DataAnalyser gicen the analysis type.
class DataAnalyserFactory:
    @staticmethod
    def create_analyser(analysis_type, scenarios, participants):
        if analysis_type == 'collision_data':
            return CollisionDataAnalyser(scenarios, participants)
        elif analysis_type == 'lane_marking_violation_data':
            return LaneMarkingDataAnalyser(scenarios, participants)
        elif analysis_type == 'vehicle_light_misuse_data':
            return VehicleLightDataAnalyser(scenarios, participants)
        elif analysis_type == 'route_data':
            return RouteDataAnalyser(scenarios, participants)
        elif analysis_type == 'speeding_data':
            return SpeedingDataAnalyser(scenarios, participants)
        elif analysis_type == 'road_traffic_violation_data':
            return RoadTrafficDataAnalyser(scenarios, participants)
        else:
            raise ValueError('Invalid analyser type')

# Used to read and process recorded data according to the specification provided
# in analysis_parameters.json
class Parser():

    def __init__(self, parameters, save_location):
        self.participants = parameters["participants"]
        self.scenarios = parameters["scenarios"]
        self.metrics = parameters["metrics"]
        self.save_location = '../../data/analysed_data/data/{}'.format(save_location)

        try:
            self.analysers = []
            for metric in self.metrics:
                analyser = DataAnalyserFactory.create_analyser(metric, self.scenarios, self.participants)
                self.analysers.append(analyser)
        except Exception as e:
            print("Could not initiate all the analysers correctly", e)

    # Make all the DataAnalysers do their analyses
    def parse(self):
        for analyser in self.analysers:
            analyser.run()

    # Save the analysed data
    def save(self):
        try:
            for analyser in self.analysers:
                analyser.save_points(self.save_location + "/points")
            self.save_participants()
            self.save_scenarios()
        except Exception as e:
            print("Something wrong happened trying to save the analysed data:", e)

    # Write relevant data to the participants.xml file
    def save_participants(self):
        try:
            root = ET.Element("Participants")
            for participant in self.participants:
                participant_element = ET.SubElement(root, "Participant")
                ET.SubElement(participant_element, "Name").text = str(participant)
                analysis = ET.SubElement(participant_element, "Analysis")
                # For each participant write what each of the analysers calculated
                total_amount_of_points = 0
                for analyser in self.analysers:
                    ET.SubElement(analysis, analyser.save_file_xml_element_name_average).text = str('%.2f' % (analyser.participant_score[participant] / len(self.s
                    if analyser.category != "route_data":
                        total_amount_of_points += analyser.participant_score[participant]

                ET.SubElement(participant_element, "TotalNumberOfPenaltyPoints").text = str('%.2f' % total_amount_of_points)
            xml_str = ET.tostring(root, 'utf-8', method='xml')
            xml_formatted = parseString(xml_str).toprettyxml(indent="  ")
            with open("{}/participants.xml".format(self.save_location, participant), "w+") as file_writer:
                file_writer.write(xml_formatted)
        except IOError as e:
            print("An error occured trying to write analytics to participants.xml file:", e)
        except Exception as e:
            print("Something wrong happened trying to save participants analytics:", e)

    # Write relevant data to the scenarios.xml file
    def save_scenarios(self):
        try:
            root = ET.Element("Scenarios")
            for scenario in self.scenarios:
                scenario_element = ET.SubElement(root, "Scenario")
                ET.SubElement(scenario_element, "Name").text = str(scenario)
                analysis = ET.SubElement(scenario_element, "Analysis")
                # For each scenario write what each of the analysers calculated
                for analyser in self.analysers:
                    ET.SubElement(analysis, analyser.save_file_xml_element_name_average).text = str('%.2f' % (analyser.scenario_score[scenario] / len(self.partic
            # Write to file
            xml_str = ET.tostring(root, 'utf-8', method='xml')
            xml_formatted = parseString(xml_str).toprettyxml(indent="  ")
            with open("{}/scenarios.xml".format(self.save_location, scenario), "w+") as file_writer:
                file_writer.write(xml_formatted)
        except IOError as e:
            print("An error occured trying to write analytics to scenarios.xml file:", e)
        except Exception as e:
            print("Something wrong happened trying to save scenario analytics:", e)


```

File: ./software/data_analysis/classes/data_analysers.py

```python
import os
import xml.etree.ElementTree as ET
from xml.dom.minidom import parseString

# DataAnalysers are used to parse data from files according to their type and then save it to analysed data files
class DataAnalyser:
    def __init__(self, scenarios, participants):
        self.path_to_recordings = '../../data/recordings/'
        self.scenarios = scenarios
        self.participants = participants

        # A list of pairs: (participant, score_on_all_scenarios)
        self.participant_score = {participant: 0.0 for participant in participants}

        # A list of pairs: (scenario, score_of_all_participants)
        self.scenario_score = {scenario: 0.0 for scenario in scenarios}

        # A list of pairs: (scenario, interesting points of all participants)
        self.scenario_points = {scenario: [] for scenario in scenarios}

    # This method goes through the recording files and processes files that are relevant
```

```
    def analyse_files(self, xml_category):
        try:
            # First it looks for a directory for each participant according to the participants specified
            # in the analysis.parameters.json
            for participant in self.participants:
                participant_directories = os.listdir(self.path_to_recordings)
                if participant in participant_directories:
                    item_path = os.path.join(self.path_to_recordings, participant)
                    if os.path.isdir(item_path):
                        # Then it looks for a directory for each scenario specified in the analysis.parameters.json
                        for scenario in self.scenarios:
                            scenario_directories = os.listdir(item_path)
                            if scenario in scenario_directories:
                                path = os.path.join(item_path, scenario, "{}.xml".format(self.category))
                                # And then it looks for a file according to the analysers category
                                # If one is found, it is processed
                                if os.path.isfile(path):
                                    try:
                                        self.process_file(path, scenario, participant, xml_category)
                                    except Exception as e:
                                        print("An exception occured trying to process file {}:".format(path), e)
        except Exception as e:
            print("An exception occured trying to gather data from recording files:", e)

    # Used to process a single file
    def process_file(self, path_to_file, scenario_name, participant_name, xml_category):
        tree = ET.parse(path_to_file)
        root = tree.getroot()
        if self.category == "route_data":
            penalty_points_total = root.find("ProportionOfRouteCompleted").text
            location_data = []
            for instance in root.findall("{}/Instance".format(xml_category)):
                if instance.find("WasReached").text == "0":
                    x = instance.find("Location").find("X").text
                    y = instance.find("Location").find("Y").text
                    location_data.append((x, y))
        else:
            penalty_points_total = root.find("PenaltyPointsTotal").text
            location_data = []
            for instance in root.findall("{}/Instance/Location".format(xml_category)):
                x = instance.find("X").text
                y = instance.find("Y").text
                location_data.append((x, y))

        self.scenario_points[scenario_name].extend(location_data)
        self.scenario_score[scenario_name] += float(penalty_points_total)
        self.participant_score[participant_name] += float(penalty_points_total)

    # Used to write all gathered point to the points directory.
    def save_points(self, path):
        try:
            for scenario, points_list in self.scenario_points.items():
                root = ET.Element("Points")
                for x, y in points_list:
                    point = ET.SubElement(root, "Point")
                    ET.SubElement(point, "X").text = str(x)
                    ET.SubElement(point, "Y").text = str(y)

                xml_str = ET.tostring(root, 'utf-8', method='xml')
                xml_formatted = parseString(xml_str).toprettyxml(indent="  ")

                with open("{}/{}_{}.xml".format(path, scenario, self.category), "w+") as file_writer:
                    file_writer.write(xml_formatted)
        except IOError as e:
            print("An error occured trying to save points of {}:".format(self.category), e)
        except Exception as e:
            print("Something wrong happened trying to save participants analytics:", e)

    # Run the data analyser
    def run(self):
        self.process_data()

    # Abstract method
    def process_data(self):
        raise NotImplemented("The method being called is abstract")


# Different data analysers that look for different metrics
class CollisionDataAnalyser(DataAnalyser):
    def __init__(self, scenarios, participants):
        self.category = "collision_data"
        self.save_file_xml_element_name_average = "CollisionPenaltyPointsAverage"
        super().__init__(scenarios, participants)

    def process_data(self):
        self.analyse_files("CollisionInstances")


class VehicleLightDataAnalyser(DataAnalyser):
    def __init__(self, scenarios, participants):
        self.category = "vehicle_light_misuse_data"
        self.save_file_xml_element_name_average = "VehicleLightMisusePenaltyPointsAverage"
        super().__init__(scenarios, participants)

    def process_data(self):
        self.analyse_files("VehicleLightsMisuseInstances")

class RouteDataAnalyser(DataAnalyser):
    def __init__(self, scenarios, participants):
        self.category = "route_data"
        self.save_file_xml_element_name_average = "ProportionOfRouteCompletedAverage"
        super().__init__(scenarios, participants)

    def process_data(self):
        self.analyse_files("Waypoints")


class LaneMarkingDataAnalyser(DataAnalyser):
    def __init__(self, scenarios, participants):
        self.category = "lane_marking_violation_data"
        self.save_file_xml_element_name_average = "LaneMarkingViolationPenaltyPointsAverage"
        super().__init__(scenarios, participants)

    def process_data(self):
        self.analyse_files("LaneMarkingViolationInstances")


class SpeedingDataAnalyser(DataAnalyser):
    def __init__(self, scenarios, participants):
        self.category = "speeding_data"
        self.save_file_xml_element_name_average = "SpeedingPenaltyPointsAverage"
        super().__init__(scenarios, participants)

    def process_data(self):
        self.analyse_files("SpeedingInstances")


class RoadTrafficDataAnalyser(DataAnalyser):
    def __init__(self, scenarios, participants):
```

```
        self.category = "road_traffic_violation_data"
        self.save_file_xml_element_name_average = "RoadTrafficViolationPenaltyPointsAverage"
        super().__init__(scenarios, participants)

    def process_data(self):
        self.analyse_files("RoadTrafficViolationInstances")
```

File: ./software/data_analysis/score_calculator.py

```
import argparse
import json
import xml.etree.ElementTree as ET

# The gamma value
GAMMA = 0.7

metrics = [
        "collision_data",
        "lane_marking_violation_data",
        "vehicle_light_misuse_data",
        "route_data",
        "speeding_data",
        "road_traffic_violation_data"
    ]

argparser = argparse.ArgumentParser(description=__doc__)
argparser.add_argument(
    '--scenario', '-s',
    help='The simulation to consider')
argparser.add_argument(
    '--name', '-n',
    help='The name of the participant')
args = argparser.parse_args()

# Used to print the final score of the participant to the screen
def main():
    try:
        details = read_details()
        try:
            score, ideal_score = calculate_the_score(details)
            print("The score for {} in {} is: ".format(args.name, args.scenario) + str(score))
            print("The ideal score is: {} and above".format(ideal_score))
        except Exception as e:
            print("An exception occured calculating the score:", e)
    except Exception as e:
        print("Something went wrong...", e)

# Function that calculates the final score according to the formula specified in the final report Chapter 7
def calculate_the_score(details):
    score = 0
    d = details["difficulty"]
    alpha = details["intensity"]
    v_avg = details["average_speed"]
    s = details["distance"]
    c, t, penalty_points = do_route_analysis(args.scenario, args.name)
    time_for_additional_stops = sum([entry["time"] for entry in details["stops"]])

    # The final formula in action
    t_o = (s / v_avg) * (1 + alpha) + time_for_additional_stops
    score = (c * (t_o / t) * d) - GAMMA * penalty_points
    return float('%.3f' % score), d

# Sum up all the penalty points
def do_route_analysis(scenario, name):
    penalty_points = 0
    try:
        for metric in metrics:
            tree = ET.parse("../../data/recordings/{}/{}/{}.xml".format(name, scenario, metric))
            root = tree.getroot()
            if metric == "route_data":
                c = float(root.find("ProportionOfRouteCompleted").text)
                t = float(root.find("SimulationTime").text[:-1])
            else:
                penalty_points += float(root.find("PenaltyPointsTotal").text)
    except Exception as e:
        print("An exception occured trying to gather data from recording files:", e)

    return c, t, penalty_points

# Read the details of the simulation
def read_details():
    try:
        with open("../../data/simulation_details/{}.json".format(args.scenario), "r") as file:
            details = json.load(file)
        return details
    except Exception as e:
        print("Something wrong happened trying to get data from {}.json".format(args.scenario), e)

if __name__ == '__main__':
    if args.name and args.scenario:
        main()
    else:
        "Please make sure that you pass the required arguments -s (scenario name) and -n (participant's name)"
```

File: ./software/data_analysis/README.md

````
# Data Analysis

## Introduction
This directory is dedicated to processing the recorded simulation data and presenting it in a meaningful way. In addition, it contains tools to evaluate the perfor
## Content
The directory contains the following:
- analysis_parameters.json: A JSON file that holds information about the participants, scenarios, and metrics to be analyzed.
- analysis.py: A Python script that performs the data analysis based on the parameters defined in the analysis_parameters.json file.
- run_analysis.sh: A shell script to run the analysis using the analysis.py script.
- visualisation: Subdirectory containing scripts that for graphical representation.
- classes: Additional classes to help the analysis of recorded data files.
- score_calculator.py: a script to calculate the score of the participant in the given scenario
- scenario_analyser.py: a script to analyse the scenarios generating files that help to evaluate the performance of a participant

## analysis_parameters.json
Imagine that analysis_parameters.json file has the following structure:

```
{
    "participants": [
        "ParticipantA",
        "ParticipantB"
    ],
    "scenarios": [
        "scenario1"
    ],
    "metrics": [
        "collision_data",
        "road_traffic_violation_data"
    ]
}
```
This file states that we want to analyze the collisions and road traffic violation data from scenario1 driven by Participants A and B.

**The list of all possible metrics:**

```
    "collision_data",
    "lane_marking_violation_data",
    "vehicle_light_misuse_data",
    "route_data",
    "speeding_data",
    "road_traffic_violation_data"
```

## Running the Analysis
To run the analysis, simply execute the following command in the terminal:

```
bash run_analysis.sh name_of_analysis
```

Analysed data will be saved in the ./analysed_data/data/name_of_analysis directory.

Replace "name_of_analysis" with a desired name for the analysis instance.

## Results
The analysis tool will process the recording according to the parameters specified in the analysis_parameters.json file and produce the following files:
- participants.xml: Contains information about each of the specified participants and their average scores.
- scenarios.xml: Contains information about each of the specified scenarios and the average scores of participants.
- n .xml files in the points subdirectory: Each file contains a list of x and y coordinates, representing interesting patterns in the data.

For instance, the file scenario1_collision_data.xml will contain all the collision points for Participants A and B driving scenario1.

## Visual Representation
For visual representation of the data, please refer to the subdirectory [visualisation](./visualisation/).

## Scenario analyser
Before evaluating the score of a participant, we need to run the scenario analyser that will gather data from the simulated world and will create a file that will

The `scenario_analyser.py` script takes two arguments: -p indicating the path file and -s indicating the scenario file. It creates a file in the [simulation_detail

## Score evaluation
The score evaluation is performed using the `score_calculator.py` script. The script takes arguments -n indicating the name of the participant and -s argument indi
````

File: ./software/data_analysis/run_analysis.sh

```
#!/bin/bash

name=$1


if [ -z "$name" ]; then
     echo "If you want to run data analysis, the name for the analysis has to be specified."
     echo "Example use: ./run_analysis.sh name"
     exit 1
fi

cd ../../data/analysed_data/data/

if [ -d "$name" ]; then
  echo "Directory $name already exists. Please choose another name. Aborting."
  exit 1
fi

mkdir $name
mkdir ./$name/points

cd ../../../software/data_analysis
python analysis.py -s $name
```

File: ./software/data_analysis/visualisation/README.md

```
## Information about drawing

This directory contains subdirectories for different kind of visual analysis methods.

Please refer to the subdirectories for more details.
```

File: ./software/data_analysis/visualisation/diagram_drawing/draw_diagram.py

```python
import xml.etree.ElementTree as ET
import argparse
import seaborn as sns
import matplotlib.pyplot as plt
import sys

def draw_bar_chart(args):
    # Parse XML file
    try:
        tree = ET.parse(args.xml_file)
    except ET.ParseError as e:
        print(f"Error parsing XML file: {e}")
        sys.exit(1)
    root = tree.getroot()
    names = []
    scores = []
    for participant in root:
        name = participant.find("Name").text
        try:
            score = float(participant.find("Analysis").find(args.category).text)
        except ValueError as e:
            print(f"Error parsing score value: {e}")
            sys.exit(1)
        names.append(name)
        scores.append(score)

    # Plot bar chart
    sns.set_palette("Blues", color_codes=True)
    sns.set_style("darkgrid")
    sns.set(font_scale=1.3)

    sns.set_context("notebook")
    sns.barplot(x=names, y=scores)
    plt.xlabel(args.xlabel)
    plt.ylabel("Penalty Points")
    plt.title(args.title)
    if args.save:
        try:
            plt.savefig(args.save)
        except Exception as e:
            print(f"Error saving the diagram: {e}")
            sys.exit(1)
    plt.show()

def main():
    # Parse command line arguments
    parser = argparse.ArgumentParser()
    parser.add_argument("--xml_file", type=str, help="Path to the XML file")
    parser.add_argument("--category", type=str, help="Category of the data")
    parser.add_argument("--save", type=str, help="Path to save the diagram")
    parser.add_argument("--title", type=str, help="Title of the diagram")
    parser.add_argument("--xlabel", type=str, help="xlabel text")
    args = parser.parse_args()

    # Draw bar chart
    draw_bar_chart(args)

if __name__ == "__main__":
    main()
```

File: ./software/data_analysis/visualisation/diagram_drawing/README.md

```markdown
## Information about diagram drawing

To draw the diagram look at the [draw_diagram_example.sh](./draw_diagram_example.sh).

**The list of all possible categories:**

```
      CollisionPenaltyPointsAverage
      LaneMarkingViolationPenaltyPointsAverage
      VehicleLightMisusePenaltyPointsAverage
      ProportionOfRouteCompletedAverage
      SpeedingPenaltyPointsAverage
      RoadTrafficViolationPenaltyPointsAverage
```
- save_filename - specifies the name of the file that the diagram should be saved to. This file is an image.
- title - specifies what shoudl be written at the top of the diagram
- xlabel - x axis label
- **data_filename** - points to the analysed data directory and then to either participants.xml or scenarios.xml. Please note, that at this moment, the diagrams c
```

File: ./software/data_analysis/visualisation/diagram_drawing/draw_diagram_example.sh

```bash
#!/bin/bash

####### ONLY THIS BIT IS MEANT TO BE EDITED #######
category="LaneMarkingViolationPenaltyPointsAverage"
title="Lane Marking Violation Penalty Points Average in Scenario 1"
xlabel="Participants"
save_filename="example_participants"
data_filename="example_analysis/participants.xml"
####### ####### ####### ####### ####### ####### ###

data_path="../../../../data/analysed_data/data/$data_filename"
save_path="../../../../data/analysed_data/diagrams/$save_filename"

python draw_diagram.py --xml_file "$data_path" --category "$category" --save "$save_path" --title "$title" --xlabel "$xlabel"
```

File: ./software/data_analysis/visualisation/map_drawing/draw_map.sh

```
#!/bin/bash

# map specified which commonroad format file to use to depict the map layout
map=$1

# first_file is path to an .xml file containing the locations to mark on the map
first_file=$2

# second_file contains locations the same way the first_file does
second_file=$3

# For this, neccessary python libraties need to be installed
# More information on this can be found in the python_libraries directory

# Check if any of the arguments were provided
if [[ -z "$map" || -z "$first_file" ]]; then
    # If either map or first_file is missing, exit the script and echo a message
    echo "Error: missing arguments. Please specify map and first_file with points."
    exit 1
fi

# Check if map or first_file is missing
if [[ ! -f "../../../../data/maps/commonroad_format/$map" || ! -f "../../../../data/analysed_data/data/$first_file" ]]; then
    # If either map or first_file is missing, exit the script and echo a message
    echo "Error: map file or first_file does not exist. Please check the paths provided."
    exit 1
fi

# Launch crdesigner with appropriate arguments and disable any output to the terminal
echo "Drawing the map and marking the points."
if [ -n "$second_file" ]; then
    crdesigner -p1 "../../../../data/analysed_data/data/$first_file" -i "../../../../data/maps/commonroad_format/$map" -p2 "../../../../data/analysed_data/data/$se
else
    crdesigner -p1 "../../../../data/analysed_data/data/$first_file" -i "../../../../data/maps/commonroad_format/$map" > /dev/null 2>&1
fi
echo "Finished drawing the map and points."

# Remove files directory and its contents
rm -rf "files"
```

File: ./software/data_analysis/visualisation/map_drawing/README.md

```
## Data visualisation on the map

This directory contains scripts that visualise the specific points on the map using the modified crdesigner module which can be found [here](../../../python_libra

Please note that I am not the author of the crdesigner module. It is a slightly modified version allowing for additional files to be specified when launching it re

To use the map drawing tool, please refer to the `draw_map_example.sh` script.

This script takes a CommonRoad type XML map description and a list of points from the analysed data. An example of how files containing points look like can be fou

The `draw_map_example.sh` script also allows to provide a second file containing points.

Points from the first file are marked in blue color and points form the second file are marked in red color. This can be useful to analyse where human drivers and
```

File: ./software/data_analysis/visualisation/map_drawing/draw_map_example.sh

```
#!/bin/bash

first_file="example_analysis/points/scenario2_collision_data.xml"
#second_file="example_analysis/points/scenario1_collision_data.xml"
map="Town07_commonRoad.xml"

# Second file can also be passed into the draw_map script to draw two depict two different datasets on the map
bash draw_map.sh $map $first_file #$second_file
```

File: ./software/data_analysis/scenario_analyser.py

```
import glob
import os
import sys
import time
import random
import logging
import argparse
import xml.etree.ElementTree as ET
import numpy
import json
sys.path.append("..")
from carla_scripts.config import SAMPLING_RESOLUTION
from agents.navigation.global_route_planner import GlobalRoutePlanner

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla


argparser = argparse.ArgumentParser(description=__doc__)
argparser.add_argument(
    '--scenario', '-s',
    help='The scenario to consider')
argparser.add_argument(
    '--path', '-p',
    help='The path to consider')
args = argparser.parse_args()


# From location makes a route
def get_route(map, locations):
    waypoints = []
    grp = GlobalRoutePlanner(map, SAMPLING_RESOLUTION)

    for i in range(len(locations)-1):
        waypoints.extend(grp.trace_route(locations[i], locations[i+1]))
    return waypoints

# Read the path file
def read_path_file():
    file_path = "../../data/paths/{}.json".format(args.path)
    try:
        with open(file_path) as file:
            data = json.load(file)
            locations = []
            for loc in data["path_checkpoints"]:
```

```
                locations.append(carla.Location(x=float(loc["x"]), y=float(loc["y"]), z=float(loc["z"])))
            return locations, data["town"]
    except Exception as e:
        print("Could not read the file {}".format(args.path))

# Retrieve road elements from the OpenDRIVE file
def get_road_elements(map):
        path_to_file = "../../data/maps/opendrive_format/{}.xodr".format(map)
        tree = ET.parse(path_to_file)
        root = tree.getroot()

        # Get all road elements from the XML
        roads = []
        for road in root.findall("./road"):
            roads.append(road)
        return roads

# Get total distance of all waypoints
def get_total_distance(waypoints):
    total_distance = 0
    for i in range(len(waypoints)-1):
        distance = waypoints[i][0].transform.location.distance(waypoints[i+1][0].transform.location)
        total_distance += distance
    return total_distance

# Calculate the intensity and retrieve the difficulty
def analyse_scenario(total_number_of_spawn_points):
    file_path = "../../data/scenario_generation_data/generated_scenarios/{}.json".format(args.scenario)
    try:
        with open(file_path) as file:
            data = json.load(file)
            total_vehicle_amount = int(data["number_of_vehicles"]) + int(data["number_of_two_wheel_vehicles"])
            # Assume that at most 80% of the spawn points are viable since if we were to put cars into all of them, it would result in the biggest gridlock in the
            possible_spawn_point_amount = int(0.8 * total_number_of_spawn_points)
            intensity = total_vehicle_amount / possible_spawn_point_amount
            if intensity > 1: intensity = 1

            return int(data["difficulty"]), intensity
    except Exception as e:
        print("Could not read the file {}".format(file_path))

# Calculate the average speed of all waypoints and also return how many road elements do not have the speed specified at all
def get_average_speed(waypoints, roads):
    road_speed, how_many_elements_have_no_max_speed_specified = get_road_speed_dictionary(roads)
    average_speed = 0
    speed_limits = []

    for waypoint in waypoints:
        id = waypoint[0].road_id
        if waypoint[0].road_id is not None and id in road_speed:
            speed_limits.append(road_speed[waypoint[0].road_id])

    if len(speed_limits) > 0:
        average_speed = sum(speed_limits) / len(speed_limits)

    return average_speed, how_many_elements_have_no_max_speed_specified

# Create a dictionary of road_id ---> max allowed speed
def get_road_speed_dictionary(roads):
    speed_dict = {}
    i = 0
    for road in roads:
        id = int(road.get('id'))
        type = road.find('type')
        speed_elem = None

        if type is not None:
            speed_elem = type.find('speed')

        if speed_elem is None:
            # The standard speed --> 50km/h converted to m/s
            i += 1
            speed_dict[id] = float(50 / 3.6)
        else:
            unit = speed_elem.get('unit')
            value = float(speed_elem.get('max'))
            if unit == 'mph':
                speed_dict[id] = value * 1.60934 / 3.6
            else:
                speed_dict[id] = value / 3.6

    return speed_dict, i

# For now count how many junctions there are and for each assign a time value
def get_stops(waypoints):
    stops = []
    junctions = 0
    last_waypoint = "road"
    for waypoint in waypoints:
        if waypoint[0].is_junction:
            if last_waypoint == "road":
                junctions += 1
                last_waypoint = "junction"
        else:
            if last_waypoint == "junction":
                last_waypoint = "road"

    for i in range (junctions):
        stops.append({
            'name': "Junction",
            'time': 12
        })
    return stops


# Save scenario to file
def write_to_file(obj):
    with open("../../data/simulation_details/{}.json".format(args.scenario), "w+") as f:
        json.dump(obj, f, indent = 4)


# Used to analyse the path and the scenario to retireve important data needed to evaluate the performance of the participants
def main():
    try:
        client = carla.Client("localhost", 2000)
        client.set_timeout(10)
        locations, map_name = read_path_file()

        # Change the map to get the map element
        world = client.load_world(map_name)
        map = world.get_map()

        waypoints = get_route(map, locations)
        roads = get_road_elements(map_name)

        total_distance = get_total_distance(waypoints)
        difficulty, intensity = analyse_scenario(len(map.get_spawn_points()))
        average_speed, how_many_elements_have_no_max_speed_specified = get_average_speed(waypoints, roads)

        stops = get_stops(waypoints)
```

```
183
184         final_obj = {
185             "path": args.path,
186             "scenario": args.scenario,
187             "difficulty": float('%.3f' % difficulty),
188             "intensity": float('%.3f' % intensity),
189             "distance": float('%.3f' % total_distance),
190             "average_speed": float('%.3f' % average_speed),
191             "stops": stops
192         }
193
194         write_to_file(final_obj)
195
196         # Uncomment to see how (not) wll the OpenDRIVE maps in carla are specified
197         # print("")
198         # print("In {} only {}% of all road elements have speed values specified".format(map_name, ('%.2f' % (100 - how_many_elements_have_no_max_speed_specified
199         # print("")
200         print("Successfully analysed and save data to data/scenario_details/{}.json file".format(args.scenario))
201     except Exception as e:
202         print("Something has gone wrong.", e)
203
204 if __name__ == '__main__':
205     main()
```

File: ./software/carla_scripts/replay.sh

```
1 #!/bin/bash
2
3 file="$1_scenario$2_recording.log"
4 python3 replayer.py -f $file
```

File: ./software/carla_scripts/path_generator.py

```
1  import argparse
2  import json
3  import logging
4  from scenarios.path_builder import PathBuilder
5
6  parser = argparse.ArgumentParser(description="Generate path parameters for a given scenario")
7  parser.add_argument('-s', '--scenario', default='example', help='Name of scenario')
8  parser.add_argument('-f', '--filename', default='example', help='Filename')
9  args = parser.parse_args()
10
11 logger = logging.getLogger(__name__)
12 logging.basicConfig(level=logging.INFO, filename="../../data/paths/logs/logs.log", filemode="w", format="%(asctime)s - %(name)s - %(levelname)s - %(message)s")
13
14 # Get a file indicating scenario parameters
15 def load_scenario():
16     logger.info("Loading scenario {} data".format(args.scenario))
17     file_path = "../../data/scenario_generation_data/generated_scenarios/{}.json".format(args.scenario)
18     try:
19         with open(file_path) as file:
20             scenario_data = json.load(file)
21             logger.info("Successfully read scenario {}.json data".format(args.scenario))
22             return scenario_data
23     except FileNotFoundError as e:
24         logger.exception("Scenario {} file could not be found.".format(args.scenario), e)
25     except json.JSONDecodeError as e:
26         logger.exception("The file {}.json could not be decoded as JSON".format(args.scenario), e)
27
28 # The main script
29 def main(agrs):
30     logger.info("Begin path generation")
31     try:
32         scenario_data = load_scenario()
33         path = PathBuilder.generate_details(scenario_data, args.filename)
34         if path: print("Generated and saved.")
35     except Exception as e:
36         logger.fatal("Fatal error. Shutting down the generation.")
37
38 if __name__ == '__main__':
39     main(args)
```

File: ./software/carla_scripts/run.py

```
1  import argparse
2  import json
3  import os
4  import shutil
5  import logging
6  import random
7  from simulation_manager import SimulationManager
8  import subprocess
9
10 parser = argparse.ArgumentParser(description="Run all the scenarios for the participant")
11 parser.add_argument('-n', '--name', default='example', help='Name of the participant')
12 parser.add_argument(
13     '--ai',
14     action='store_true',
15     help='Is the ai driving?')
16 args = parser.parse_args()
17
18 logger = logging.getLogger(__name__)
19 logging.basicConfig(level=logging.INFO, filename="../../data/recordings/{}/session_logs.log".format(args.name), filemode="a", format="%(asctime)s - %(name)s - %(
20
21 scenario_list_file = "scenario_list.json"
22
23 # Read the scenario json file and extract all names of the scenarios to run
24 def get_scenarios():
25     try:
26         with open(scenario_list_file) as file:
27             scenarios = json.load(file)
28             logger.info("Successfully extracted scenarios from file {}".format(scenario_list_file))
29             return scenarios
30     except FileNotFoundError as e:
31         logger.error("Scenario list file could not be found.", exc_info=True)
32         raise e
33     except json.JSONDecodeError as e:
34         logger.error("The file {} could not be decoded as JSON".format(scenario_list_file), exc_info=True)
35         raise e
36
37 # Create a directory in the participant's results directory to store results about the simulation of a particular scenario
38 def create_directory_for_scenario(scenario_name):
39     path = "../../data/recordings/{}/{}".format(args.name, scenario_name)
40     if os.path.exists(path):
41         shutil.rmtree(path)
42     try:
43         os.makedirs(path)
44     except OSError:
45         logger.error("Could not create a directory for scenario {} for participant {}".format(scenario_name, args.name))
```

```
        raise Exception
    logger.info("Successfully created a directory for scenario {}".format(scenario_name))

def try_to_get_name_for_new_scenario():
    path = "../../data/scenario_generation_data/generated_scenarios/"
    i = 0
    # Try to find a name for a new scenario file for a 1000000 times
    while i < 1000000:
        name = "scenario{}".format(i)
        path = os.path.join(path, name)
        if not os.path.exists("{}.json".format(path)):
            return name
        i += 1
    raise Exception("Could not find an unoccupied name for the new scenario. Most likely names scenario[1-1000000] are taken")

def check_scenario(scenario):
    if scenario["name"] is not None:
        return scenario["name"]
    else:
        logger.info("The scenario name was null. Predicting a new scenario.")
        # Generate a new scenario and give it a name and return it
        try:
            name = try_to_get_name_for_new_scenario()
            difficulty = random.randint(1, 1000)
            logger.info("Scenario generator will now try to create a new scenario named {} of difficulty {}".format(name, difficulty))
            # Launch the first person driving mode
        except Exception as e:
            logger.error("The scenario file was not specified and also a new one could not be created", exc_info=True)
            raise e

        try:
            logger.info("Launching create_scenario.py script.")
            # python3: can't open file '../software/generating_scenarios/create_scenario.py'
            process = subprocess.run(["python3", "../../software/generating_scenarios/create_scenario.py", "-d{}".format(difficulty), "-f{}".format(name)], stdou
            # Uncomment to print to the terminal
            # print(process.stdout.decode(), process.stderr.decode())
            logger.info("Successfully generated a new scenario {}".format(name))
            return name
        except Exception as e:
            logger.error("Could not open the driver.py script", exc_info=True)
            raise e

current_simulation_manager = None

# Launch a simulation manager for the scenario
def launch_scenario(scenario):
    name = check_scenario(scenario)
    create_directory_for_scenario(name)
    global current_simulation_manager
    current_simulation_manager = SimulationManager(name, scenario["details"], args.name, scenario["vehicle"], scenario["randomization_seed"], args.ai)
    current_simulation_manager.begin_simulation()


# The main script
def main(agrs):
    logger.info("THE SIMULATIONS BEGIN NOW")
    failed_scenarios = []
    try:
        data = get_scenarios()
        for scenario in data["scenarios"]:
            try:
                logger.info("Starting to work on scenario {}".format(scenario["name"]))
                launch_scenario(scenario)
            except Exception as e:
                failed_scenarios.append(scenario["name"])
                logger.error("Failed to simulate the scenario {}".format(scenario), exc_info=True)

        # The end of all the simulations
        logger.info("Successfully completed {} out of {} simulations".format((len(data["scenarios"])-len(failed_scenarios)), len(data["scenarios"])))
        if len(failed_scenarios) > 0:
            logger.warning("The failed scenarios are:")
            for f_s in failed_scenarios:
                logger.warning(f_s)
    except KeyboardInterrupt:
        logger.warning("The simulation was quit manually.")
    except Exception as e:
        logger.error("Something went horribly wrong.", exc_info=True)
    finally:
        global current_simulation_manager
        if current_simulation_manager is not None:
            current_simulation_manager.close_scenario()
        logger.info("Shutting down the simulation.")

if __name__ == '__main__':
    main(args)
```

File: ./software/carla_scripts/replayer.py

```python
#!/usr/bin/env python

# Copyright (c) 2019 Computer Vision Center (CVC) at the Universitat Autonoma de
# Barcelona (UAB).
#
# This work is licensed under the terms of the MIT license.
# For a copy, see <https://opensource.org/licenses/MIT>.

import glob
import os
import sys

import carla

import argparse

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

# Used to replay the specified recording
# Provide the neccessary args
def main():

    argparser = argparse.ArgumentParser(
        description=__doc__)
    argparser.add_argument(
        '--host',
        metavar='H',
        default='127.0.0.1',
        help='IP of the host server (default: 127.0.0.1)')
    argparser.add_argument(
        '-p', '--port',
        metavar='P',
        default=2000,
        type=int,
        help='TCP port to listen to (default: 2000)')
    argparser.add_argument(
        '-s', '--start',
        metavar='S',
        default=0.0,
        type=float,
        help='starting time (default: 0.0)')
    argparser.add_argument(
        '-d', '--duration',
        metavar='D',
        default=0.0,
        type=float,
        help='duration (default: 0.0)')
    argparser.add_argument(
        '-f', '--recorder-filename',
        metavar='F',
        default="test.log",
        help='recorder filename (test1.log)')
    argparser.add_argument(
        '-c', '--camera',
        metavar='C',
        default=0,
        type=int,
        help='camera follows an actor (ex: 82)')
    argparser.add_argument(
        '-x', '--time-factor',
        metavar='X',
        default=1.0,
        type=float,
        help='time factor (default 1.0)')
    argparser.add_argument(
        '-i', '--ignore-hero',
        action='store_true',
        help='ignore hero vehicles')
    argparser.add_argument(
        '--spawn-sensors',
        action='store_true',
        help='spawn sensors in the replayed world')
    args = argparser.parse_args()

    try:
        print("Beginning to replay the recording at {}".format(args.recorder_filename))
        client = carla.Client(args.host, args.port)
        client.set_timeout(120.0)

        # Set the time factor for the replayer
        client.set_replayer_time_factor(args.time_factor)

        # Set to ignore the hero vehicles or not
        client.set_replayer_ignore_hero(args.ignore_hero)

        # Replay the session
        client.replay_file(args.recorder_filename, args.start, args.duration, args.camera, args.spawn_sensors)

    except KeyboardInterrupt:
        pass
    except Exception as e:
        print("Something wrong happened trying to replay the recording:", e)
    finally:
        print("Stopped replaying.")


if __name__ == '__main__':
    main()
```

File: ./software/carla_scripts/config.py

```
# General variables

DISTANCE_FROM_WAYPOINT = 0.7
FINISH_DISTANCE_FROM_WAYPOINT = 3.0
DISTANCE_FROM_WAYPOINT_CHANGING_LANE = 1.0
SAMPLING_RESOLUTION = 0.5
SAMPLING_RESOLUTION_AI = 5
FPS = 60
RANDOMIZATION_SEED = 187349

# Penalty related variables

NO_BEAMS_NO_FOG_LIGHTS = 50
NO_BEAMS_NO_FOG_LIGHTS_TEXT = "Dark and foggy - no lights at all"

NO_BEAMS = 30
NO_BEAMS_TEXT = "Dark - no low beams"

NO_FOG_LIGHTS = 10
NO_FOG_LIGHTS_TEXT = "Dark and foggy - no fog lights"

HIT_PEDESTRIAN_TEXT = "Hit pedestrian"
HIT_PEDESTRIAN_PENALTY = 600
HIT_PEDESTRIAN_PENALTY_SPEEDING = 1200

HIT_VEHICLE_TEXT = "Hit vehicle"
HIT_VEHICLE_PENALTY = 250
HIT_VEHICLE_PENALTY_SPEEDING = 500

HIT_BICYCLE_TEXT = "Hit bicycle"
HIT_BICYCLE_PENALTY = 400
HIT_BICYCLE_PENALTY_SPEEDING = 800

HIT_ROAD_OBJECT_TEXT = "Hit road object"
HIT_ROAD_OBJECT_PENALTY = 150
HIT_ROAD_OBJECT_PENALTY_SPEEDING = 300

RED_LIGHT_TEXT = "Red light"
RED_LIGHT_PENALTY = 50
RED_LIGHT_PENALTY_SPEEDING = 100

SOLID_TEXT = "Solid lane marking"
SOLID_PENALTY = 20
SOLID_PENALTY_SPEEDING = 60

DOUBLE_SOLID_TEXT = "Double solid lane marking"
DOUBLE_SOLID_PENALTY = 40
DOUBLE_SOLID_PENALTY_SPEEDING = 100

BROKEN_NO_TURN_INDICATOR_TEXT = "Broken line wrong or no turn indicator"
BROKEN_NO_TURN_INDICATOR_PENALTY = 10
BROKEN_NO_TURN_INDICATOR_PENALTY_SPEEDING = 30

LIGHT_SPEEDING_PENALTY = 1
LIGHT_SPEEDING_TEXT = "Speeding under 20km/h"

HEAVY_SPEEDING_PENALTY = 3
HEAVY_SPEEDING_TEXT = "Speeding over 20km/h"

# If speeding exceeds this amount, it is considered heavy speeding (50 allowed ---> 71+ is considered heavy speeding)
HEAVY_SPEEDING_THRESHOLD = 20
```

File: ./software/carla_scripts/classes/route.py

```
from config import FINISH_DISTANCE_FROM_WAYPOINT, DISTANCE_FROM_WAYPOINT, DISTANCE_FROM_WAYPOINT_CHANGING_LANE
from classes.helpers import distanceBetweenTwoLocations

class Route():

    def __init__(self, waypoints):
        self.routepoints = []
        for waypoint in waypoints:
            self.routepoints.append((waypoint, 0))
        # Last routepoint is out finish point
        self.finish_lane_waypoints = self.get_finish_lane_waypoints(waypoints[len(waypoints) - 1][0])


    def get_finish_lane_waypoints(self, waypoint):
        return [*self.getAllLeftWaypointsRecursively(waypoint, [], 0), *self.getAllRightWaypointsRecursively(waypoint, [], 0), waypoint]

    # Return percent of route completed
    def get_route_completion(self):
        length = len(self.routepoints)
        completed = sum(x[1] for x in self.routepoints)
        return '%.2f' % (completed / length)

    # Get shortest distance comparing each waypoint in waypoints
    # with the provided location
    def get_shortest_distance(self, location):
        shortest_distance = None
        for routepoint in self.routepoints:
            waypoint_location = routepoint[0][0].transform.location
            distance = distanceBetweenTwoLocations(location, waypoint_location)
            if shortest_distance is None or distance < shortest_distance:
                shortest_distance = distance
        return shortest_distance

    # Return true if the last location on route was reached
    def is_finished(self, player_location):
        for waypoint in self.finish_lane_waypoints:
            distance = distanceBetweenTwoLocations(player_location, waypoint.transform.location)
            if distance <= FINISH_DISTANCE_FROM_WAYPOINT:
                return True
        return False

    # Marks the routepoints (x, 1) if the routepoint's distance is <= DISTANCE_FROM_WAYPOINT
    def recalculate(self, location):
        for i in range(len(self.routepoints)):
            routepoint = self.routepoints[i]
            waypoint_location = routepoint[0][0].transform.location
            distance = distanceBetweenTwoLocations(location, waypoint_location)
            if distance <= DISTANCE_FROM_WAYPOINT:
                waypoint = routepoint[0]
                self.routepoints.pop(i)
                self.routepoints.append((waypoint, 1))

        # print("Completed: " + str(self.get_completage()))

    ###          ADVANCED STUFF

    # Marks the routepoints (x, 1) if the routepoint's distance is <= DISTANCE_FROM_WAYPOINT
    def advanced_recalculate(self, location, is_changing_lane):
        for i in range(len(self.routepoints)):
            routepoint = self.routepoints[i]
            waypoint = routepoint[0]
            left_waypoints = self.getLeftValidWaypointsRecursively(waypoint[0], [], 0)
```

```
            right_waypoints = self.getRightValidWaypointsRecursively(waypoint[0], [], 0)
            waypoints = [*left_waypoints, *right_waypoints, waypoint[0]]
            for w in waypoints:
                distance = distanceBetweenTwoLocations(location, w.transform.location)
                if is_changing_lane:
                    if distance <= DISTANCE_FROM_WAYPOINT:
                        self.routepoints.pop(i)
                        self.routepoints.append((waypoint, 1))
                else:
                    if distance <= DISTANCE_FROM_WAYPOINT_CHANGING_LANE:
                        self.routepoints.pop(i)
                        self.routepoints.append((waypoint, 1))
        # print("Completed: " + str(self.get_completage()))

    def get_all_waypoints(self):
        waypoints = []
        for routepoint in self.routepoints:
            waypoint = routepoint[0][0]
            waypoints.append(waypoint)
            for w in self.getLeftValidWaypointsRecursively(waypoint, [], 0):
                waypoints.append(w)
            for w in self.getRightValidWaypointsRecursively(waypoint, [], 0):
                waypoints.append(w)
        return waypoints

    # Get all lanes on the current road (to the left of the waypoint) that are valid drive for the vehicle driving this route.
    def getLeftValidWaypointsRecursively(self, waypoint, list, recursive_calls):
        if waypoint == None or recursive_calls > 8:
            return list
        lane_change = waypoint.lane_change
        if str(lane_change) == "Left" or str(lane_change) == "Both":
            left_lane_waypoint = waypoint.get_left_lane()
            if left_lane_waypoint == None:
                return list
            list.append(left_lane_waypoint)
            recursive_calls += 1
            return self.getLeftValidWaypointsRecursively(left_lane_waypoint, list, recursive_calls)
        else:
            return list

    # Get all road lanes on the map to the left of the current lane. Primarily used when marking the finish line.
    def getAllLeftWaypointsRecursively(self, waypoint, list, recursive_calls):
        left_lane_waypoint = waypoint.get_left_lane()
        if left_lane_waypoint is not None:
            list.append(left_lane_waypoint)
            recursive_calls += 1
            return self.getLeftValidWaypointsRecursively(left_lane_waypoint, list, recursive_calls)
        else:
            return list

    # Get all lanes on the current road (to the right of the waypoint) that are valid drive for the vehicle driving this route.
    def getRightValidWaypointsRecursively(self, waypoint, list, recursive_calls):
        if waypoint == None or recursive_calls > 8:
            return list
        lane_change = waypoint.lane_change
        if str(lane_change) == "Right" or str(lane_change) == "Both":
            right_lane_waypoint = waypoint.get_right_lane()
            if right_lane_waypoint == None:
                return list
            list.append(right_lane_waypoint)
            recursive_calls += 1
            return self.getRightValidWaypointsRecursively(right_lane_waypoint, list, recursive_calls)
        else:
            return list

    # Get all road lanes on the map to the right of the current lane. Primarily used when marking the finish line.
    def getAllRightWaypointsRecursively(self, waypoint, list, recursive_calls):
        right_lane_waypoint = waypoint.get_right_lane()
        if right_lane_waypoint is not None:
            list.append(right_lane_waypoint)
            recursive_calls += 1
            return self.getRightValidWaypointsRecursively(right_lane_waypoint, list, recursive_calls)
        else:
            return list
```

File: ./software/carla_scripts/classes/helpers.py

```
import numpy

# Get distance between start and finish locations
# start and finish are both of type carla.Location
def distanceBetweenTwoLocations(start, finish):
    first = numpy.array((start.x, start.y, start.z))
    second = numpy.array((finish.x, finish.y, finish.z))
    return numpy.linalg.norm(first-second)

# The following list is used to match the vehicle light state and figure out what lights are on
# The first element of the tuple is the "code" that the simualtion returns about the state of car's lights
# The second element is a tuple of five True/False values representing the states of the following
# (LeftBlinker, RightBlinker, LowBeam, HighBeam, FogLights)
pairs = [
        ("NONE",(False, False, False, False, False)),
        # Left
        ("LeftBlinker",(True, False, False, False, False)),
        ("40",(True, False, False, False, False)),
        ("96",(True, False, False, False, False)),
        ("104",(True, False, False, False, False)),
        # Left Position
        ("33",(True, False, False, False, False)),
        ("41",(True, False, False, False, False)),
        ("97",(True, False, False, False, False)),
        ("105",(True, False, False, False, False)),
        # Left Low
        ("99",(True, False, True, False, False)),
        ("107",(True, False, True, False, False)),
        ("35",(True, False, True, False, False)),
        ("43",(True, False, True, False, False)),
        # Left Fog
        ("227",(True, False, True, False, True)),
        ("235",(True, False, True, False, True)),
        ("163",(True, False, True, False, True)),
        ("171",(True, False, True, False, True)),
        # Right
        ("RightBlinker",(False, True, False, False, False)),
        ("24",(False, True, False, False, False)),
        ("80",(False, True, False, False, False)),
        ("88",(False, True, False, False, False)),
        # Right Position
        ("17",(False, True, False, False, False)),
        ("25",(False, True, False, False, False)),
        ("81",(False, True, False, False, False)),
        ("89",(False, True, False, False, False)),
        # Right Low
        ("19",(False, True, True, False, False)),
        ("27",(False, True, True, False, False)),
        ("83",(False, True, True, False, False)),
        ("91",(False, True, True, False, False)),
        # Right Fog
        ("211",(False, True, True, False, True)),
        ("219",(False, True, True, False, True)),
        ("147",(False, True, True, False, True)),
        ("155",(False, True, True, False, True)),
        # Low
        ("3",(False, False, True, False, False)),
        ("11",(False, False, True, False, False)),
        ("67",(False, False, True, False, False)),
        ("75",(False, False, True, False, False)),
        # Fog
        ("131",(False, False, True, False, True)),
        ("139",(False, False, True, False, True)),
        ("195",(False, False, True, False, True)),
        ("203",(False, False, True, False, True)),
        # High
        ("HighBeam",(False, False, False, True, False)),
        ("12",(False, False, False, True, False)),
        ("68",(False, False, False, True, False)),
        ("76",(False, False, False, True, False)),
        # Left High
        ("36",(True, False, False, True, False)),
        ("44",(True, False, False, True, False)),
        ("100",(True, False, False, True, False)),
        ("108",(True, False, False, True, False)),
        # Right High
        ("20",(False, True, False, True, False)),
        ("28",(False, True, False, True, False)),
        ("84",(False, True, False, True, False)),
        ("92",(False, True, False, True, False)),
        # Low High
        ("7",(False, False, True, True, False)),
        ("71",(False, False, True, True, False)),
        ("79",(False, False, True, True, False)),
        ("15",(False, False, True, True, False)),
        # Fog High
        ("135",(False, False, False, True, True)),
        ("143",(False, False, False, True, True)),
        ("199",(False, False, False, True, True)),
        ("207",(False, False, False, True, True)),
        # Left High Low
        ("39",(True, False, True, True, False)),
        ("47",(True, False, True, True, False)),
        ("103",(True, False, True, True, False)),
        ("111",(True, False, True, True, False)),
        # Left High Fog
        ("167",(True, False, True, True, True)),
        ("175",(True, False, True, True, True)),
        ("231",(True, False, True, True, True)),
        ("239",(True, False, True, True, True)),
        # Right High Low
        ("23",(False, True, True, True, False)),
        ("31",(False, True, True, True, False)),
        ("87",(False, True, True, True, False)),
        ("95",(False, True, True, True, False)),
        # Right High Fog
        ("151",(False, True, True, True, True)),
        ("159",(False, True, True, True, True)),
        ("215",(False, True, True, True, True)),
        ("223",(False, True, True, True, True))
    ]

# The default value to return in case the match among the pairs was not found (NO LIGHTS ARE ON)
default_match = (False, False, False, False, False)
```

File: ./software/carla_scripts/recording/lane_monitor.py

```python
import xml.etree.ElementTree as ET
from xml.dom.minidom import parseString
from config import SOLID_PENALTY_SPEEDING, SOLID_TEXT, SOLID_PENALTY, DOUBLE_SOLID_PENALTY_SPEEDING, DOUBLE_SOLID_TEXT,DOUBLE_SOLID_PENALTY, BROKEN_NO_TURN_INDICA

import logging
logger = logging.getLogger(__name__)
class LaneMonitor():

    def __init__(self, save_file):
        self.save_file = save_file
        self.buffer = []
        self.sum = 0

    def add_to_buffer(self, lane_type, timestep, player, is_speeding, turn_indicator_text):
        location = player.get_location()
        steering_angle = player.get_control().steer
        points_message = self.points_given_and_message(lane_type, is_speeding, steering_angle, turn_indicator_text)
        # Add only violations and lleave-out 0 penalty-point entries
        if(points_message and points_message[0] != 0):
            self.sum += points_message[0]
            self.buffer.append({
                'points': points_message[0],
                'message': points_message[1],
                'time': str('%.3f' % (timestep / 1000)) + "s",
                'location': {
                    'x': '%.6f' % location.x,
                    'y': '%.6f' % location.y,
                    'z': '%.6f' % location.z
                }
            })

    def write_to_file(self):
        length = len(self.buffer)
        root = ET.Element("LaneMarkingViolationData")
        ET.SubElement(root, "PenaltyPointsTotal").text = str(self.sum)
        ET.SubElement(root, "NumberOfLaneMarkingViolationInstances").text = str(length)

        entries = ET.SubElement(root, "LaneMarkingViolationInstances")
        for entry in self.buffer:
            entry_xml = ET.SubElement(entries, "Instance")
            ET.SubElement(entry_xml, "PenaltyPoints").text = str(entry['points'])
            ET.SubElement(entry_xml, "Description").text = entry['message']
            ET.SubElement(entry_xml, "Time").text = str(entry['time'])
            location = ET.SubElement(entry_xml, "Location")
            ET.SubElement(location, "X").text = str(entry['location']['x'])
            ET.SubElement(location, "Y").text = str(entry['location']['y'])
            ET.SubElement(location, "Z").text = str(entry['location']['z'])

        xml_str = ET.tostring(root, 'utf-8', method='xml')
        xml_formatted = parseString(xml_str).toprettyxml(indent="  ")

        try:
            path = "{}lane_marking_violation_data.xml".format(self.save_file)
            with open(path, 'w+') as file_writer:
                file_writer.write(xml_formatted)
            logger.info("Lane monitor successfully saved recordings to the file {}".format(path))
        except Exception as e:
            logger.error("Lane monitor did not save data to the file {}".format(path), e)

    def points_given_and_message(self, lane_type, speeding, steering_angle, turn_indicator_text):
        if lane_type == "Solid":
            if speeding:
                return(SOLID_PENALTY_SPEEDING, SOLID_TEXT + " speeding")
            else:
                return(SOLID_PENALTY, SOLID_TEXT)
        elif lane_type == "SolidSolid":
            if speeding:
                return(DOUBLE_SOLID_PENALTY_SPEEDING, DOUBLE_SOLID_TEXT + " speeding")
            else:
                return(DOUBLE_SOLID_PENALTY, DOUBLE_SOLID_TEXT)
        elif lane_type == "Broken":
            if turn_indicator_text == "Left" and steering_angle <= 0:
                # Turning left legally
                pass
            elif turn_indicator_text == "Right" and steering_angle >= 0:
                # Turning right legally
                pass
            else:
                if speeding:
                    return(BROKEN_NO_TURN_INDICATOR_PENALTY_SPEEDING, BROKEN_NO_TURN_INDICATOR_TEXT + " speeding")
                else:
                    return(BROKEN_NO_TURN_INDICATOR_PENALTY, BROKEN_NO_TURN_INDICATOR_TEXT)
        else:
            return(0, 'None')
```

File: ./software/carla_scripts/recording/speeding_monitor.py

```python
import xml.etree.ElementTree as ET
from xml.dom.minidom import parseString
from config import LIGHT_SPEEDING_PENALTY, HEAVY_SPEEDING_PENALTY, LIGHT_SPEEDING_TEXT, HEAVY_SPEEDING_TEXT, HEAVY_SPEEDING_THRESHOLD

import logging
logger = logging.getLogger(__name__)
class SpeedingMonitor():

    def __init__(self, save_file):
        self.save_file = save_file
        self.buffer = []
        self.sum = 0

    def add_to_buffer(self, allowed_speed, player_speed, timestep, location):
        points_message = self.points_given_and_message(allowed_speed, player_speed)
        self.sum += points_message[0]
        self.buffer.append({
            'points': points_message[0],
            'message': points_message[1],
            'allowed_speed': allowed_speed,
            'player_speed': player_speed,
            'time': str('%.3f' % (timestep / 1000)) + "s",
            'location': {
                'x': '%.6f' % location.x,
                'y': '%.6f' % location.y,
                'z': '%.6f' % location.z
            }
        })

    def write_to_file(self):
        length = len(self.buffer)
        root = ET.Element("SpeedingData")
        ET.SubElement(root, "PenaltyPointsTotal").text = str(self.sum)
        ET.SubElement(root, "NumberOfSpeedingInstances").text = str(length)

        entries = ET.SubElement(root, "SpeedingInstances")
        for entry in self.buffer:
            entry_xml = ET.SubElement(entries, "Instance")
            ET.SubElement(entry_xml, "PenaltyPoints").text = str(entry['points'])
            ET.SubElement(entry_xml, "Description").text = entry['message']
            ET.SubElement(entry_xml, "AllowedSpeed").text = str(entry['allowed_speed'])
            ET.SubElement(entry_xml, "PlayerSpeed").text = str(entry['player_speed'])
            ET.SubElement(entry_xml, "Time").text = str(entry['time'])
            location = ET.SubElement(entry_xml, "Location")
            ET.SubElement(location, "X").text = str(entry['location']['x'])
            ET.SubElement(location, "Y").text = str(entry['location']['y'])
            ET.SubElement(location, "Z").text = str(entry['location']['z'])

        xml_str = ET.tostring(root, 'utf-8', method='xml')
        xml_formatted = parseString(xml_str).toprettyxml(indent="  ")

        try:
            path = "{}speeding_data.xml".format(self.save_file)
            with open(path, 'w+') as file_writer:
                file_writer.write(xml_formatted)
            logger.info("Speeding monitor successfully saved recordings to the file {}".format(path))
        except Exception as e:
            logger.error("Speeding monitor did not save data to the file {}".format(path), e)

    def points_given_and_message(self, allowed_speed, player_speed):
        absolute_value = player_speed - allowed_speed
        if absolute_value <= HEAVY_SPEEDING_THRESHOLD:
            return (LIGHT_SPEEDING_PENALTY, LIGHT_SPEEDING_TEXT)
        else:
            return (HEAVY_SPEEDING_PENALTY, HEAVY_SPEEDING_TEXT)
```

File: ./software/carla_scripts/recording/recorder.py

```python
import glob
import os
import sys

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla

import argparse
import random
import time
import logging

import logging
logger = logging.getLogger(__name__)

class Recorder():

    def __init__(self, participant_name, scenario_name):
        self.client = carla.Client('127.0.0.1', 2000)
        self.participant_name = participant_name
        self.scenario_name = scenario_name

    def start_recording(self):
        try:
            path = '{}_{}_recording.log'.format(self.participant_name, self.scenario_name)
            self.client.start_recorder(path, True)
            logger.info("Successfully started recording the simultion to {}".format(path))
        except Exception as e:
            logger.error("An exception occurred trying to start recording", e)

    def stop_recording(self):
        try:
            self.client.stop_recorder()
            logger.info("Stopped the recorder. The recording log file can be found in CarlaUE4/Saved/")
        except Exception as e:
            logger.error("Could not stop the recorder", e)
```

File: ./software/carla_scripts/recording/traffic_monitor.py

```
import xml.etree.ElementTree as ET
from xml.dom.minidom import parseString
from config import RED_LIGHT_PENALTY, RED_LIGHT_PENALTY_SPEEDING, RED_LIGHT_TEXT

import logging
logger = logging.getLogger(__name__)
class TrafficMonitor():

    def __init__(self, save_file):
        self.save_file = save_file
        self.buffer = []
        self.sum = 0

    def add_to_buffer(self, speeding, timestep, location, type):
        points_message = self.points_given_and_message(type, speeding)
        self.sum += points_message[0]
        self.buffer.append({
            'points': points_message[0],
            'message': points_message[1],
            'time': str('%.3f' % (timestep / 1000)) + "s",
            'location': {
                'x': '%.6f' % location.x,
                'y': '%.6f' % location.y,
                'z': '%.6f' % location.z
            }
        })

    def write_to_file(self):
        length = len(self.buffer)
        root = ET.Element("RoadTrafficViolationData")
        ET.SubElement(root, "PenaltyPointsTotal").text = str(self.sum)
        ET.SubElement(root, "NumberOfRoadTrafficViolationInstances").text = str(length)

        entries = ET.SubElement(root, "RoadTrafficViolationInstances")
        for entry in self.buffer:
            entry_xml = ET.SubElement(entries, "Instance")
            ET.SubElement(entry_xml, "PenaltyPoints").text = str(entry['points'])
            ET.SubElement(entry_xml, "Description").text = entry['message']
            ET.SubElement(entry_xml, "Time").text = str(entry['time'])
            location = ET.SubElement(entry_xml, "Location")
            ET.SubElement(location, "X").text = str(entry['location']['x'])
            ET.SubElement(location, "Y").text = str(entry['location']['y'])
            ET.SubElement(location, "Z").text = str(entry['location']['z'])

        xml_str = ET.tostring(root, 'utf-8', method='xml')
        xml_formatted = parseString(xml_str).toprettyxml(indent="  ")

        try:
            path = "{}road_traffic_violation_data.xml".format(self.save_file)
            with open(path, 'w+') as file_writer:
                file_writer.write(xml_formatted)
            logger.info("Traffic monitor successfully saved recordings to the file {}".format(path))
        except Exception as e:
            logger.error("Traffic monitor did not save data to the file {}".format(path), e)

    def points_given_and_message(self, type, is_speeding):
        if type == "redlight":
            if is_speeding:
                return(RED_LIGHT_PENALTY_SPEEDING, RED_LIGHT_TEXT + " speeding")
            else:
                return(RED_LIGHT_PENALTY, RED_LIGHT_TEXT)
        else:
            return(0, "Unknown")
```

File: ./software/carla_scripts/recording/vehicle_light_monitor.py

```python
import xml.etree.ElementTree as ET
from xml.dom.minidom import parseString
from classes.helpers import pairs, default_match
from config import FPS, NO_BEAMS_NO_FOG_LIGHTS, NO_BEAMS, NO_FOG_LIGHTS, NO_BEAMS_NO_FOG_LIGHTS_TEXT, NO_BEAMS_TEXT, NO_FOG_LIGHTS_TEXT

import logging
logger = logging.getLogger(__name__)
class VehicleLightMonitor():

    def __init__(self, save_file):
        self.save_file = save_file
        self.buffer = []
        self.sum = 0
        self.just_wrote = True

        self.left_turn_indicator = False
        self.right_turn_indicator = False
        self.fog_lights = False
        self.low_beam = False
        self.high_beam = False

    def set_lights(self, code, weather, time, location):
        lights = self.match_lights(code)
        self.left_turn_indicator = lights[0]
        self.right_turn_indicator = lights[1]
        self.low_beam = lights[2]
        self.high_beam = lights[3]
        self.fog_lights = lights[4]
        # Write every 10s
        value = (time/1000) % 10
        if not self.just_wrote and (value > 0 and value <= 0.2):
            data = self.check_lights(weather)
            if(data):
                self.add_to_buffer(data[0], data[1], data[2], data[3], time, location)
            self.just_wrote = True
        elif value > 0.2:
            self.just_wrote = False
        else:
            pass

    def check_lights(self, weather):
        sun_altitude_angle = weather.sun_altitude_angle
        fog_density = weather.fog_density
        if(sun_altitude_angle < 30):
            if(fog_density > 50):
                if(not self.fog_lights):
                    if(not self.low_beam):
                        return (NO_BEAMS_NO_FOG_LIGHTS, NO_BEAMS_NO_FOG_LIGHTS_TEXT, sun_altitude_angle, fog_density)
                    else:
                        return (NO_FOG_LIGHTS, NO_FOG_LIGHTS_TEXT, sun_altitude_angle, fog_density)
                else:
                    if(not self.low_beam):
                        return (NO_BEAMS, NO_BEAMS_TEXT, sun_altitude_angle, fog_density)
            else:
                if(not self.low_beam):
                    return (NO_BEAMS, NO_BEAMS_TEXT, sun_altitude_angle, fog_density)
        else:
            if(fog_density > 50):
                if(not self.fog_lights):
                    if(not self.low_beam):
                        return (NO_BEAMS_NO_FOG_LIGHTS, "Foggy - no lights at all", sun_altitude_angle, fog_density)
                    else:
                        return (NO_FOG_LIGHTS, "Foggy - no fog lights", sun_altitude_angle, fog_density)
                else:
                    if(not self.low_beam):
                        return (NO_BEAMS, "Foggy - no low beams", sun_altitude_angle, fog_density)
            else:
                return None

    def turning_left(self):
        return self.left_turn_indicator

    def turning_right(self):
        return self.right_turn_indicator

    def add_to_buffer(self, points_given, type, sun_altitude_angle, fog_density, timestep, location):
        self.sum += points_given
        self.buffer.append({
            'points': points_given,
            'message': type,
            'sun_altitude_angle': sun_altitude_angle,
            'fog_density': fog_density,
            'time': str('%.3f' % (timestep / 1000)) + "s",
            'location': {
                'x': '%.6f' % location.x,
                'y': '%.6f' % location.y,
                'z': '%.6f' % location.z
            }
        })

    def write_to_file(self):
        length = len(self.buffer)
        root = ET.Element("VehicleLightMisuseData")
        ET.SubElement(root, "PenaltyPointsTotal").text = str(self.sum)
        ET.SubElement(root, "NumberOfVehicleLightMisuseInstances").text = str(length)

        entries = ET.SubElement(root, "VehicleLightMisuseInstances")
        for entry in self.buffer:
            entry_xml = ET.SubElement(entries, "Instance")
            ET.SubElement(entry_xml, "PenaltyPoints").text = str(entry['points'])
            ET.SubElement(entry_xml, "Description").text = entry['message']
            ET.SubElement(entry_xml, "SunAltitudeAngle").text = str(entry['sun_altitude_angle'])
            ET.SubElement(entry_xml, "FogDensity").text = str(entry['fog_density'])
            ET.SubElement(entry_xml, "Time").text = str(entry['time'])
            location = ET.SubElement(entry_xml, "Location")
            ET.SubElement(location, "X").text = str(entry['location']['x'])
            ET.SubElement(location, "Y").text = str(entry['location']['y'])
            ET.SubElement(location, "Z").text = str(entry['location']['z'])

        xml_str = ET.tostring(root, 'utf-8', method='xml')
        xml_formatted = parseString(xml_str).toprettyxml(indent="  ")

        try:
            path = "{}vehicle_light_misuse_data.xml".format(self.save_file)
            with open(path, 'w+') as file_writer:
                file_writer.write(xml_formatted)
            logger.info("Vehicle light misuse monitor successfully saved recordings to the file {}".format(path))
        except Exception as e:
            logger.error("Vehicle light misuse monitor did not save data to the file {}".format(path), e)


    def match_lights(self, code):
        for pair in pairs:
            if pair[0] == str(code):
                return pair[1]
        return default_match

```

File: ./software/carla_scripts/recording/route_monitor.py

```python
import carla
import xml.etree.ElementTree as ET
from xml.dom.minidom import parseString
from classes.route import Route
from agents.navigation.global_route_planner import GlobalRoutePlanner
from config import SAMPLING_RESOLUTION
import json

import logging
logger = logging.getLogger(__name__)

class RouteMonitor():
    def __init__(self, path_locations, map, save_file):
        self.route = self.get_route(path_locations, map)
        self.save_file = save_file

    # Takes a location as an argument and checks if it belongs to the route
    def is_on_path(self, location):
        pass

    def update(self, location, is_changing_lane):
        self.route.advanced_recalculate(location, is_changing_lane)

    def finish_reached(self, player_location):
        if self.route.is_finished(player_location):
            return True

    def write_to_file(self, finish_reached, simulation_time):
        root = ET.Element("RouteData")
        ET.SubElement(root, "SimulationTime").text = str('%.3f' % (simulation_time / 1000))+"s"
        ET.SubElement(root, "ProportionOfRouteCompleted").text = str(self.route.get_route_completion())
        ET.SubElement(root, "FinishReached").text = str(finish_reached)
        ET.SubElement(root, "NumberOfWaypoints").text = str(len(self.route.routepoints))
        waypoints = ET.SubElement(root, "Waypoints")
        for routepoint in self.route.routepoints:
            location = routepoint[0][0].transform.location
            entry_xml = ET.SubElement(waypoints, "Instance")
            ET.SubElement(entry_xml, "WasReached").text = str(routepoint[1])
            location_elem = ET.SubElement(entry_xml, "Location")
            ET.SubElement(location_elem, "X").text = str('%.6f' % location.x)
            ET.SubElement(location_elem, "Y").text = str('%.6f' % location.y)
            ET.SubElement(location_elem, "Z").text = str('%.6f' % location.z)

        xml_str = ET.tostring(root, 'utf-8', method='xml')
        xml_formatted = parseString(xml_str).toprettyxml(indent="  ")
        try:
            path = "{}route_data.xml".format(self.save_file)
            with open(path, 'w+') as file_writer:
                file_writer.write(xml_formatted)
            logger.info("Route monitor successfully saved recordings to the file {}".format(path))
        except Exception as e:
            logger.error("Route monitor did not save data to the file {}".format(path), e)


    def get_route(self, path_locations, map):
        waypoints = []
        grp = GlobalRoutePlanner(map, SAMPLING_RESOLUTION)
        locations = self.transform_to_carla_locations(path_locations)
        for i in range(len(locations)-1):
            waypoints.extend(grp.trace_route(locations[i], locations[i+1]))
        return Route(waypoints)

    def transform_to_carla_locations(self, path_locations):
        locations = []
        for loc in path_locations:
            locations.append(carla.Location(x=float(loc["x"]), y=float(loc["y"]), z=float(loc["z"])))
        return locations

    # WORKS WHERE THERE ARE ONLY TWO LOCATIONS PROVIDED
    #
    #
    # def get_route(self, pathName, map):
    #     grp = GlobalRoutePlanner(map, SAMPLING_RESOLUTION)
    #     locations = self.read_path_file(pathName)
    #     # Define a route
    #     waypoints = grp.trace_route(locations[0], locations[1])
    #     return Route(waypoints)

    # def read_path_file(self, pathName):
    #     start = None
    #     finish = None
    #     with open("./paths/{}.txt".format(pathName), "r") as file:
    #         i = 0
    #         for line in file.readlines():
    #             values = [float(j) for j in line.split(',') if j.strip()]
    #             if i == 0:
    #                 start = carla.Location(x=values[0], y=values[1], z=values[2])
    #             else:
    #                 finish = carla.Location(x=values[0], y=values[1], z=values[2])
    #             i+=1
    #     return (start, finish)
```

File: ./software/carla_scripts/recording/collision_monitor.py

```python
import xml.etree.ElementTree as ET
from xml.dom.minidom import parseString
from config import HIT_PEDESTRIAN_PENALTY_SPEEDING, HIT_PEDESTRIAN_TEXT, HIT_PEDESTRIAN_PENALTY, HIT_VEHICLE_PENALTY_SPEEDING, HIT_VEHICLE_TEXT, HIT_VEHICLE_PENAL
from config import HIT_BICYCLE_TEXT, HIT_BICYCLE_PENALTY, HIT_ROAD_OBJECT_PENALTY_SPEEDING, HIT_ROAD_OBJECT_TEXT, HIT_ROAD_OBJECT_PENALTY

import logging
logger = logging.getLogger(__name__)
class CollisionMonitor():

    def __init__(self, save_file):
        self.save_file = save_file
        self.buffer = []
        self.sum = 0

    def add_to_buffer(self, intensity, other_actor, location, timestep, is_speeding):
        points_message = self.points_given_and_message(other_actor.type_id, intensity, is_speeding)
        self.sum += points_message[0]
        self.buffer.append({
            'points': points_message[0],
            'message': points_message[1],
            'intensity': intensity,
            'time': str('%.3f' % (timestep / 1000)) + "s",
            'location': {
                'x': '%.6f' % location.x,
                'y': '%.6f' % location.y,
                'z': '%.6f' % location.z
            }
        })

    def write_to_file(self):
        length = len(self.buffer)
        root = ET.Element("CollisionData")
        ET.SubElement(root, "PenaltyPointsTotal").text = str(self.sum)
        ET.SubElement(root, "NumberOfCollisionInstances").text = str(length)

        speeding_entries = ET.SubElement(root, "CollisionInstances")
        for entry in self.buffer:
            entry_xml = ET.SubElement(speeding_entries, "Instance")
            ET.SubElement(entry_xml, "PenaltyPoints").text = str(entry['points'])
            ET.SubElement(entry_xml, "Description").text = str(entry['message'])
            ET.SubElement(entry_xml, "CollisionIntensity").text = str(entry['intensity'])
            ET.SubElement(entry_xml, "Time").text = str(entry['time'])
            location = ET.SubElement(entry_xml, "Location")
            ET.SubElement(location, "X").text = str(entry['location']['x'])
            ET.SubElement(location, "Y").text = str(entry['location']['y'])
            ET.SubElement(location, "Z").text = str(entry['location']['z'])

        xml_str = ET.tostring(root, 'utf-8', method='xml')
        xml_formatted = parseString(xml_str).toprettyxml(indent="  ")
        try:
            path = "{}collision_data.xml".format(self.save_file)
            with open(path, 'w+') as file_writer:
                file_writer.write(xml_formatted)
            logger.info("Collision monitor successfully saved recordings to the file {}".format(path))
        except Exception as e:
            logger.error("Collision monitor did not save data to the file {}".format(path), e)

    def points_given_and_message(self, victim_type, intensity, speeding):
        # Intensity to be used
        if victim_type.startswith("walker"):
            if speeding:
                return(HIT_PEDESTRIAN_PENALTY_SPEEDING, HIT_PEDESTRIAN_TEXT + " speeding")
            else:
                return(HIT_PEDESTRIAN_PENALTY, HIT_PEDESTRIAN_TEXT)
        elif victim_type.startswith("vehicle"):
            if speeding:
                return(HIT_VEHICLE_PENALTY_SPEEDING, HIT_VEHICLE_TEXT + " speeding")
            else:
                return(HIT_VEHICLE_PENALTY, HIT_VEHICLE_TEXT)
        elif victim_type.startswith("bicycle"):
            if speeding:
                return(HIT_BICYCLE_PENALTY_SPEEDING, HIT_BICYCLE_TEXT + " speeding")
            else:
                return(HIT_BICYCLE_PENALTY, HIT_BICYCLE_TEXT)
        else:
            if speeding:
                return(HIT_ROAD_OBJECT_PENALTY_SPEEDING, HIT_ROAD_OBJECT_TEXT + " speeding")
            else:
                return(HIT_ROAD_OBJECT_PENALTY, HIT_ROAD_OBJECT_TEXT)

    ### UNUSED

    def get_actor_display_name(self, actor, truncate=250):
        name = ' '.join(actor.type_id.replace('_', '.').title().split('.')[1:])
        return (name[:truncate - 1] + u'\u2026') if len(name) > truncate else name
```

File: ./software/carla_scripts/recording/manager.py

```python
import carla
import weakref
import math
import time
import sys
from recording.route_monitor import RouteMonitor
from recording.lane_monitor import LaneMonitor
from recording.collision_monitor import CollisionMonitor
from recording.traffic_monitor import TrafficMonitor
from recording.speeding_monitor import SpeedingMonitor
from recording.vehicle_light_monitor import VehicleLightMonitor
from recording.recorder import Recorder

import logging
logger = logging.getLogger(__name__)
class Manager():

    def __init__(self, path_locations, world, player, participant_name, scenario_name):
        self.path_locations = path_locations
        self.world = world
        self.player = player
        self.participant_name = participant_name
        self.scenario_name = scenario_name
        self.map = self.world.get_map()
        self.time = 0

        self.speeding = False
        self.junction_visited_in_last_timestep = False

        self.collision_sensor = CollisionSensor(self.player, self.world, self)
        self.lane_invasion_sensor = LaneInvasionSensor(self.player, self.world, self)
        save_file = '../../7data/recordings/{}/{}/'.format(participant_name, scenario_name)
        self.route_monitor = RouteMonitor(self.path_locations, self.map, save_file)
        self.lane_monitor = LaneMonitor(save_file)
        self.collision_monitor = CollisionMonitor(save_file)
        self.speeding_monitor = SpeedingMonitor(save_file)
        self.traffic_monitor = TrafficMonitor(save_file)
        self.vehicle_light_monitor = VehicleLightMonitor(save_file)
```

```
39
40          self.recorder = Recorder(participant_name, scenario_name)
41          self.recorder.start_recording()
42
43          self.time_of_last_collision = None
44          self.actor_of_last_collision = None
45          logger.info("Manager was successfully initialised")
46
47      # Called at each timestep. Checks for violations and records to respective monitors
48      def record(self, simulation_time):
49          # logger.info("Manager recording at {}".format(str('%.3f' % (simulation_time / 1000))+"s"))
50          try:
51              player_location = self.player.get_location()
52              self.time = simulation_time
53              # Check if turn indicators are on
54              weather = self.world.get_weather()
55              self.set_lights(weather, self.time, player_location)
56              # Check if turn indicators are on
57              is_changing_lane = False
58              if self.vehicle_light_monitor.turning_left() or self.vehicle_light_monitor.turning_right():
59                  is_changing_lane = True
60
61              self.route_monitor.update(player_location, is_changing_lane)
62              speeding = self.check_speeding()
63              violating_traffic = self.check_traffic_violations()
64              if speeding:
65                  self.speeding_monitor.add_to_buffer(speeding[1], speeding[0], self.time, player_location)
66              if violating_traffic:
67                  self.traffic_monitor.add_to_buffer(self.speeding, self.time, player_location, "redlight")
68              # logger.info("Observations were added to the buffers successfully.")
69              if self.route_monitor.finish_reached(player_location):
70                  logger.info("Finish was reached. Terminating the simulation.")
71                  self.shut_down(True)
72                  return True
73          except Exception as e:
74              logger.error("Something went wrong adding observations to monitor buffers.", e)
75
76      # Called when finish is reached or simulation is quit manually
77      def shut_down(self, finish_reached):
78          logger.info("Shutting the manager down. Saving data from monitors to files.")
79          try:
80              self.route_monitor.write_to_file(finish_reached, self.time)
81              self.lane_monitor.write_to_file()
82              self.collision_monitor.write_to_file()
83              self.traffic_monitor.write_to_file()
84              self.speeding_monitor.write_to_file()
85              self.vehicle_light_monitor.write_to_file()
86              self.recorder.stop_recording()
87          except Exception as e:
88              logger.error("Something went wrong saving data from monitors to files.", e)
89          try:
90              logger.info("Destroying sensors.")
91              self.delete_sensors()
92          except Exception as e:
93              logger.error("Something went wrong destroying sensors owned by the Manager.", e)
94      # After the simulation is over, delete sensors
95      def delete_sensors(self):
96          self.collision_sensor.sensor.stop()
97          self.collision_sensor.sensor.destroy()
98          self.lane_invasion_sensor.sensor.stop()
99          self.lane_invasion_sensor.sensor.destroy()
100
101     # Record the collision to the collision monitor buffer
102     def record_collision(self, intensity, other_actor, player_location):
103         # If there was a collision witht the same actor in the past 2sec, disregards this (collision sensor sometimes counts a collision multiple times)
104         if(self.actor_of_last_collision and self.actor_of_last_collision == other_actor.id and (self.time - self.time_of_last_collision)/1000 < 2):
105             pass
106         else:
107             self.time_of_last_collision = self.time
108             self.actor_of_last_collision = other_actor.id
109             self.collision_monitor.add_to_buffer(intensity, other_actor, player_location, self.time, self.speeding)
110
111     # Record the lane marking violations to the lane monitor buffer
112     def record_lane_invasion(self, text):
113         if self.vehicle_light_monitor.turning_left():
114             light_indicator_text = "Left"
115         elif self.vehicle_light_monitor.turning_right():
116             light_indicator_text = "Right"
117         else:
118             light_indicator_text = "None"
119         self.lane_monitor.add_to_buffer(text, self.time, self.player, self.speeding, light_indicator_text)
120
121     # Check if the vehicle speeding
122     def check_speeding(self):
123         current_speed = self.player.get_velocity().x
124         # Convert from m/ to km/h and round to 2 places after comma
125         current_speed = float('%.2f' % (abs(current_speed)*3.6))
126         speed_limit = self.player.get_speed_limit()
127         if current_speed and speed_limit and current_speed > speed_limit:
128             self.speeding = True
129             return (current_speed, speed_limit)
130         else:
131             self.speeding = False
132             return None
133
134     # Check if any traffic violations occurred (for now checks only for red lights)
135     def check_traffic_violations(self):
136         waypoint = self.map.get_waypoint(self.player.get_location())
137         if waypoint.is_junction:
138             if not self.junction_visited_in_last_timestep and str(self.player.get_traffic_light_state()) == "Red":
139                 self.junction_visited_in_last_timestep = True
140                 return True
141             else:
142                 return False
143         else:
144             self.junction_visited_in_last_timestep = False
145             return False
146
147     # Update vehicle light monitor state (update which ligths are on)
148     def set_lights(self, weather, time, player_location):
149         light_state = self.player.get_light_state()
150         self.vehicle_light_monitor.set_lights(light_state, weather, time, player_location)
151
152
153 # The foundations of this sensor were borrowed from CARLA developers
154 class LaneInvasionSensor(object):
155     def __init__(self, parent_actor, world, manager):
156         self.sensor = None
157         self.manager = manager
158         bp = world.get_blueprint_library().find('sensor.other.lane_invasion')
159         self.sensor = world.spawn_actor(bp, carla.Transform(), attach_to=parent_actor)
160         # We need to pass the lambda a weak reference to self to avoid circular
161         # reference.
162         weak_self = weakref.ref(self)
163         self.sensor.listen(lambda event: LaneInvasionSensor._on_invasion(weak_self, event))
164
165     @staticmethod
166     def _on_invasion(weak_self, event):
167         self = weak_self()
168         if not self:
```

```
            return
        lane_types = set(x.type for x in event.crossed_lane_markings)
        text = ['%r' % str(x).split()[-1] for x in lane_types]
        self.manager.record_lane_invasion(text[0].replace("'", ""))


# The foundations of this sensor were borrowed from CARLA developers
class CollisionSensor(object):
    def __init__(self, parent_actor, world, manager):
        self.sensor = None
        self.player = parent_actor
        self.manager = manager
        bp = world.get_blueprint_library().find('sensor.other.collision')
        self.sensor = world.spawn_actor(bp, carla.Transform(), attach_to=self.player)
        # We need to pass the lambda a weak reference to self to avoid circular
        # reference.
        weak_self = weakref.ref(self)
        self.sensor.listen(lambda event: CollisionSensor._on_collision(weak_self, event))

    @staticmethod
    def _on_collision(weak_self, event):
        self = weak_self()
        if not self:
            return
        impulse = event.normal_impulse
        intensity = math.sqrt(impulse.x**2 + impulse.y**2 + impulse.z**2)
        self.manager.record_collision(intensity, event.other_actor, self.player.get_location())

```

File: ./software/carla_scripts/run.sh

```
#!/bin/bash

name=$1
ai=$2

if [ -z "$name" ]; then
    echo "If you want to run simulations, you need to provide the name of the participant"
    echo "Example use: ./run.sh participant1"
    echo "The name is needed to associate each participant with their simualtion results"
    exit 1
fi

# First create a directory for the participant
cd ../../data/recordings/
mkdir -p $name

# Create a log file for the session inside
touch $name/session_logs.log
chmod 666 $name/session_logs.log

cd ../../software/carla_scripts

if [ -n "$ai" ]; then
    if [ "$ai" = "ai" ]; then
        echo "Starting the simulations for AI agent. For the logs, check the $name/session_logs.log file in the recordings directory."
        python3 run.py -n "$name" --ai > /dev/null 2>&1
    else
        echo "If you want the AI implementation to run the scenarios write ai as the second argument of this script"
        exit 1
    fi
else
    echo "Starting the simulations in manual control. For the logs, check the $name/session_logs.log file in the recordings directory."
    python3 run.py -n "$name" > /dev/null 2>&1
fi
echo "Finished."
```

File: ./software/carla_scripts/README.md

```
## About the simulations

To run the simulations, your machine must meet the software requirements stated in the [README.md](../../README.md) file in the software directory.

Launch the UE4 engine running CARLA server following the guide in CARLA's [documentation](https://carla.readthedocs.io/en/latest/).

```
bash run.sh [participant's name e.g. wilson]

This will run n scenarios sequentially, as stated in the [scenario_list.json](./scenario_list.json) file.

To let the CARLA Agent execute the scenarios, please run the command stated above but add 'ai' at the end.

Example:

bash run.sh [participant's name e.g. carla_agent] ai

Recordings

All simulations are recorded and monitored. The recorded data will be stored in:
`data/recordings/[participant's name]/`

The recording of the simulation itself will be stored in:

  `CarlaUE4/Saved/`

```
bash replay.sh [participant's name] [scenario name]


This will replay the simulation of scenario driven by the participant.
```

File: ./software/carla_scripts/self_driver.py

```
import os
import glob
import sys
import numpy as np

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass
import carla

import math
import random
```

```python
import time
import base64
import pygame
import pickle
from agents.navigation.global_route_planner import GlobalRoutePlanner
from config import SAMPLING_RESOLUTION_AI
from carla import ColorConverter as cc
from agents.navigation.behavior_agent import BehaviorAgent

import argparse
import collections
import datetime
import logging
import math
import random
import re
import weakref
from recording.manager import Manager
from config import FPS
from pygame import K_TAB
from pygame import K_ESCAPE
import argparse
import logging
from config import FPS


# ==============================================================================
# -- Global functions ----------------------------------------------------------
# ==============================================================================


def find_weather_presets():
    rgx = re.compile('.+?(?:(?<=[a-z])(?=[A-Z])|(?<=[A-Z])(?=[A-Z][a-z])|$)')
    name = lambda x: ' '.join(m.group(0) for m in rgx.finditer(x))
    presets = [x for x in dir(carla.WeatherParameters) if re.match('[A-Z].+', x)]
    return [(getattr(carla.WeatherParameters, x), name(x)) for x in presets]


def get_actor_display_name(actor, truncate=250):
    name = ' '.join(actor.type_id.replace('_', '.').title().split('.')[1:])
    return (name[:truncate - 1] + u'\u2026') if len(name) > truncate else name

def get_actor_blueprints(world, filter, generation):
    bps = world.get_blueprint_library().filter(filter)
    if generation.lower() == "all":
        return bps

    # If the filter returns only one bp, we assume that this one needed
    # and therefore, we ignore the generation
    if len(bps) == 1:
        return bps

    try:
        int_generation = int(generation)
        # Check if generation is in available generations
        if int_generation in [1, 2]:
            bps = [x for x in bps if int(x.get_attribute('generation')) == int_generation]
            return bps
        else:
            print("   Warning! Actor Generation is not valid. No actor will be spawned.")
            return []
    except:
        print("   Warning! Actor Generation is not valid. No actor will be spawned.")
        return []


# ==========================

class World(object):
    def __init__(self, carla_world, hud, args, scenario, client, path_details):
        self.world = carla_world
        self.sync = args.sync
        self.actor_role_name = args.rolename
        self.path_details = path_details
        self.scenario = scenario
        self.client = client
        self.scenario_name = args.scenario
        self.participant_name = args.name
        try:
            self.map = self.world.get_map()
        except RuntimeError as error:
            print('RuntimeError: {}'.format(error))
            print('  The server could not send the OpenDRIVE (.xodr) file:')
            print('  Make sure it exists, has the same name of your town, and is correct.')
            sys.exit(1)
        self.spawn_point = None
        self.init_spawn_point(path_details["start_location"])
        self.hud = hud
        self.player = None
        self.collision_sensor = None
        self.lane_invasion_sensor = None
        self.gnss_sensor = None
        self.imu_sensor = None
        self.radar_sensor = None
        self.camera_manager = None
        self._weather_presets = find_weather_presets()
        self._weather_index = 0
        self._actor_filter = args.filter
        self._actor_generation = args.generation
        self._gamma = args.gamma
        self.restart(args)
        self.recording_enabled = 0

        # All about the simulation
        self.simulation_clock = None
        self.simulation_start_tick = 0
        self.simulation_time = 0
        self.manager = None

        self.recording_start = 0
        self.constant_velocity_enabled = False
        self.show_vehicle_telemetry = False
        self.doors_are_open = False
        self.current_map_layer = 0
        self.map_layer_names = [
            carla.MapLayer.NONE,
            carla.MapLayer.Buildings,
            carla.MapLayer.Decals,
            carla.MapLayer.Foliage,
            carla.MapLayer.Ground,
            carla.MapLayer.ParkedVehicles,
            carla.MapLayer.Particles,
            carla.MapLayer.Props,
            carla.MapLayer.StreetLights,
            carla.MapLayer.Walls,
            carla.MapLayer.All
        ]

    def init_spawn_point(self, start_location):
        try:
```

```
            self.spawn_point = carla.Transform(carla.Location(x=start_location["x"], y=start_location["y"], z=start_location["z"]), carla.Rotation(pitch=0.0, yaw
        except Exception as e:
            logger.error("Could not initiate spawn location for the driver", exc_info=True)
            raise e
    def restart(self, args):
        self.player_max_speed = 1.589
        self.player_max_speed_fast = 3.713
        # Keep same camera config if the camera manager exists.
        cam_index = self.camera_manager.index if self.camera_manager is not None else 0
        cam_pos_index = self.camera_manager.transform_index if self.camera_manager is not None else 0
        # Get a random blueprint.
        blueprint = random.choice(get_actor_blueprints(self.world, self._actor_filter, self._actor_generation))
        blueprint.set_attribute('role_name', self.actor_role_name)
        if blueprint.has_attribute('color'):
            blueprint.set_attribute('color', '{},{},{}'.format(args.red, args.green, args.blue))
        if blueprint.has_attribute('driver_id'):
            driver_id = random.choice(blueprint.get_attribute('driver_id').recommended_values)
            blueprint.set_attribute('driver_id', driver_id)
        if blueprint.has_attribute('is_invincible'):
            blueprint.set_attribute('is_invincible', 'true')
        # set the max speed
        if blueprint.has_attribute('speed'):
            self.player_max_speed = float(blueprint.get_attribute('speed').recommended_values[1])
            self.player_max_speed_fast = float(blueprint.get_attribute('speed').recommended_values[2])

        # Spawn the player.
        if self.player is not None:
            spawn_point = self.player.get_transform()
            spawn_point.location.z += 2.0
            spawn_point.rotation.roll = 0.0
            spawn_point.rotation.pitch = 0.0
            self.destroy()
            self.player = self.world.try_spawn_actor(blueprint, spawn_point)
            self.show_vehicle_telemetry = False
            self.modify_vehicle_physics(self.player)
        while self.player is None:
            if not self.map.get_spawn_points():
                print('There are no spawn points available in your map/town.')
                print('Please add some Vehicle Spawn Point to your UE4 scene.')
                sys.exit(1)
            if self.spawn_point is None:
                spawn_points = self.map.get_spawn_points()
                self.spawn_point = random.choice(spawn_points) if spawn_points else carla.Transform()
            self.player = self.world.try_spawn_actor(blueprint, self.spawn_point)
            self.show_vehicle_telemetry = False
            self.modify_vehicle_physics(self.player)
        # Set up the sensors.
        self.collision_sensor = CollisionSensor(self.player, self.hud)
        self.lane_invasion_sensor = LaneInvasionSensor(self.player, self.hud)
        self.gnss_sensor = GnssSensor(self.player)
        self.imu_sensor = IMUSensor(self.player)
        self.camera_manager = CameraManager(self.player, self.hud, self._gamma)
        self.camera_manager.transform_index = cam_pos_index
        self.camera_manager.set_sensor(cam_index, notify=False)
        actor_type = get_actor_display_name(self.player)
        self.hud.notification(actor_type)

        if self.sync:
            self.world.tick()
        else:
            self.world.wait_for_tick()

    def toggle_radar(self):
        if self.radar_sensor is None:
            self.radar_sensor = RadarSensor(self.player)
        elif self.radar_sensor.sensor is not None:
            self.radar_sensor.sensor.destroy()
            self.radar_sensor = None

    def modify_vehicle_physics(self, actor):
        #If actor is not a vehicle, we cannot use the physics control
        try:
            physics_control = actor.get_physics_control()
            physics_control.use_sweep_wheel_collision = True
            actor.apply_physics_control(physics_control)
        except Exception:
            pass

    def startScenario(self):
        if self.simulation_clock is None:
            logger.info("Starting the scenario")
            self.scenario.start()
            self.hud.notification('The simulation has started')
            self.manager = Manager(self.path_details["path_checkpoints"], self.world, self.player, self.participant_name, self.scenario_name)
            self.simulation_clock = pygame.time.Clock()
            self.simulation_start_tick = pygame.time.get_ticks()

    def finishScenario(self):
        logger.warning("Finish was NOT REACHED. Terminating the simulation.")
        self.manager.shut_down(False)
        self.scenario.finish(self.client)

    def tick(self):
        if self.simulation_clock is not None:
            self.simulation_clock.tick(60)
            self.simulation_time = self.simulation_clock.get_time()
            if self.manager.record(self.simulation_time):
                return True
            self.hud.tick(self, self.simulation_time/1000)
        else:
            self.hud.tick(self, 0)

    def render(self, display):
        self.camera_manager.render(display)
        self.hud.render(display)

    def destroy_sensors(self):
        self.camera_manager.sensor.destroy()
        self.camera_manager.sensor = None
        self.camera_manager.index = None

    def destroy(self):
        if self.radar_sensor is not None:
            self.toggle_radar()
        sensors = [
            self.camera_manager.sensor,
            self.collision_sensor.sensor,
            self.lane_invasion_sensor.sensor,
            self.gnss_sensor.sensor,
            self.imu_sensor.sensor]
        for sensor in sensors:
            if sensor is not None:
                sensor.stop()
                sensor.destroy()
        if self.player is not None:
            self.player.destroy()

# ==============================================================================
# -- HUD -----------------------------------------------------------------------
# ==============================================================================
```

```
277
278
279 class HUD(object):
280     def __init__(self, width, height):
281         self.dim = (width, height)
282         font = pygame.font.Font(pygame.font.get_default_font(), 20)
283         font_name = 'courier' if os.name == 'nt' else 'mono'
284         fonts = [x for x in pygame.font.get_fonts() if font_name in x]
285         default_font = 'ubuntumono'
286         mono = default_font if default_font in fonts else fonts[0]
287         mono = pygame.font.match_font(mono)
288         self._font_mono = pygame.font.Font(mono, 12 if os.name == 'nt' else 14)
289         self._notifications = FadingText(font, (width, 40), (0, height - 40))
290         self.server_fps = 0
291         self.frame = 0
292         self.simulation_time = 0
293         self._show_info = True
294         self._info_text = []
295         self._server_clock = pygame.time.Clock()
296
297     def on_world_tick(self, timestamp):
298         self._server_clock.tick()
299         self.server_fps = self._server_clock.get_fps()
300         self.frame = timestamp.frame
301         self.simulation_time = timestamp.elapsed_seconds
302
303     def tick(self, world, time):
304         self._notifications.tick(world, time)
305         if not self._show_info:
306             return
307         t = world.player.get_transform()
308         v = world.player.get_velocity()
309         c = world.player.get_control()
310         compass = world.imu_sensor.compass
311         heading = 'N' if compass > 270.5 or compass < 89.5 else ''
312         heading += 'S' if 90.5 < compass < 269.5 else ''
313         heading += 'E' if 0.5 < compass < 179.5 else ''
314         heading += 'W' if 180.5 < compass < 359.5 else ''
315         colhist = world.collision_sensor.get_collision_history()
316         collision = [colhist[x + self.frame - 200] for x in range(0, 200)]
317         max_col = max(1.0, max(collision))
318         collision = [x / max_col for x in collision]
319         vehicles = world.world.get_actors().filter('vehicle.*')
320         self._info_text = [
321             'Vehicle: % 20s' % get_actor_display_name(world.player, truncate=20),
322             'Route time: % 12s' % datetime.timedelta(seconds=int(time)),
323             '',
324             'Speed:   % 15.0f km/h' % (3.6 * math.sqrt(v.x**2 + v.y**2 + v.z**2)),
325             '']
326         if isinstance(c, carla.VehicleControl):
327             self._info_text += [
328                 ('Throttle:', c.throttle, 0.0, 1.0),
329                 ('Steer:', c.steer, -1.0, 1.0),
330                 ('Brake:', c.brake, 0.0, 1.0),
331                 ('Reverse:', c.reverse),
332                 ('Hand brake:', c.hand_brake),
333                 ('Manual:', c.manual_gear_shift),
334                 'Gear:        %s' % {-1: 'R', 0: 'N'}.get(c.gear, c.gear)]
335         self._info_text += [
336             '',
337             'Collision:',
338             collision,
339             '']
340
341     def toggle_info(self):
342         self._show_info = not self._show_info
343
344     def notification(self, text, seconds=2.0):
345         self._notifications.set_text(text, seconds=seconds)
346
347     def error(self, text):
348         self._notifications.set_text('Error: %s' % text, (255, 0, 0))
349
350     def render(self, display):
351         if self._show_info:
352             info_surface = pygame.Surface((220, self.dim[1]))
353             info_surface.set_alpha(100)
354             display.blit(info_surface, (0, 0))
355             v_offset = 4
356             bar_h_offset = 100
357             bar_width = 106
358             for item in self._info_text:
359                 if v_offset + 18 > self.dim[1]:
360                     break
361                 if isinstance(item, list):
362                     if len(item) > 1:
363                         points = [(x + 8, v_offset + 8 + (1.0 - y) * 30) for x, y in enumerate(item)]
364                         pygame.draw.lines(display, (255, 136, 0), False, points, 2)
365                     item = None
366                     v_offset += 18
367                 elif isinstance(item, tuple):
368                     if isinstance(item[1], bool):
369                         rect = pygame.Rect((bar_h_offset, v_offset + 8), (6, 6))
370                         pygame.draw.rect(display, (255, 255, 255), rect, 0 if item[1] else 1)
371                     else:
372                         rect_border = pygame.Rect((bar_h_offset, v_offset + 8), (bar_width, 6))
373                         pygame.draw.rect(display, (255, 255, 255), rect_border, 1)
374                         f = (item[1] - item[2]) / (item[3] - item[2])
375                         if item[2] < 0.0:
376                             rect = pygame.Rect((bar_h_offset + f * (bar_width - 6), v_offset + 8), (6, 6))
377                         else:
378                             rect = pygame.Rect((bar_h_offset, v_offset + 8), (f * bar_width, 6))
379                         pygame.draw.rect(display, (255, 255, 255), rect)
380                     item = item[0]
381                 if item:  # At this point has to be a str.
382                     surface = self._font_mono.render(item, True, (255, 255, 255))
383                     display.blit(surface, (8, v_offset))
384                 v_offset += 18
385         self._notifications.render(display)
386
387
388 # ==============================================================================
389 # -- FadingText ----------------------------------------------------------------
390 # ==============================================================================
391
392
393 class FadingText(object):
394     def __init__(self, font, dim, pos):
395         self.font = font
396         self.dim = dim
397         self.pos = pos
398         self.seconds_left = 0
399         self.surface = pygame.Surface(self.dim)
400
401     def set_text(self, text, color=(255, 255, 255), seconds=2.0):
402         text_texture = self.font.render(text, True, color)
403         self.surface = pygame.Surface(self.dim)
404         self.seconds_left = seconds
405         self.surface.fill((0, 0, 0, 0))
406         self.surface.blit(text_texture, (10, 11))
```

```
407
408     def tick(self, _, time):
409         delta_seconds = 1e-3 * time
410         self.seconds_left = max(0.0, self.seconds_left - delta_seconds)
411         self.surface.set_alpha(500.0 * self.seconds_left)
412
413     def render(self, display):
414         display.blit(self.surface, self.pos)
415
416
417 # ==============================================================================
418 # -- CollisionSensor -----------------------------------------------------------
419 # ==============================================================================
420
421
422 class CollisionSensor(object):
423     def __init__(self, parent_actor, hud):
424         self.sensor = None
425         self.history = []
426         self._parent = parent_actor
427         self.hud = hud
428         world = self._parent.get_world()
429         bp = world.get_blueprint_library().find('sensor.other.collision')
430         self.sensor = world.spawn_actor(bp, carla.Transform(), attach_to=self._parent)
431         # We need to pass the lambda a weak reference to self to avoid circular
432         # reference.
433         weak_self = weakref.ref(self)
434         self.sensor.listen(lambda event: CollisionSensor._on_collision(weak_self, event))
435
436     def get_collision_history(self):
437         history = collections.defaultdict(int)
438         for frame, intensity in self.history:
439             history[frame] += intensity
440         return history
441
442     @staticmethod
443     def _on_collision(weak_self, event):
444         self = weak_self()
445         if not self:
446             return
447         actor_type = get_actor_display_name(event.other_actor)
448         self.hud.notification('Collision with %r' % actor_type)
449         impulse = event.normal_impulse
450         intensity = math.sqrt(impulse.x**2 + impulse.y**2 + impulse.z**2)
451         self.history.append((event.frame, intensity))
452         if len(self.history) > 4000:
453             self.history.pop(0)
454
455
456 # ==============================================================================
457 # -- LaneInvasionSensor --------------------------------------------------------
458 # ==============================================================================
459
460
461 class LaneInvasionSensor(object):
462     def __init__(self, parent_actor, hud):
463         self.sensor = None
464
465         # If the spawn object is not a vehicle, we cannot use the Lane Invasion Sensor
466         if parent_actor.type_id.startswith("vehicle."):
467             self._parent = parent_actor
468             self.hud = hud
469             world = self._parent.get_world()
470             bp = world.get_blueprint_library().find('sensor.other.lane_invasion')
471             self.sensor = world.spawn_actor(bp, carla.Transform(), attach_to=self._parent)
472             # We need to pass the lambda a weak reference to self to avoid circular
473             # reference.
474             weak_self = weakref.ref(self)
475             self.sensor.listen(lambda event: LaneInvasionSensor._on_invasion(weak_self, event))
476
477     @staticmethod
478     def _on_invasion(weak_self, event):
479         self = weak_self()
480         if not self:
481             return
482         lane_types = set(x.type for x in event.crossed_lane_markings)
483         text = ['%r' % str(x).split()[-1] for x in lane_types]
484         self.hud.notification('Crossed line %s' % ' and '.join(text))
485
486
487 # ==============================================================================
488 # -- GnssSensor ----------------------------------------------------------------
489 # ==============================================================================
490
491
492 class GnssSensor(object):
493     def __init__(self, parent_actor):
494         self.sensor = None
495         self._parent = parent_actor
496         self.lat = 0.0
497         self.lon = 0.0
498         world = self._parent.get_world()
499         bp = world.get_blueprint_library().find('sensor.other.gnss')
500         self.sensor = world.spawn_actor(bp, carla.Transform(carla.Location(x=1.0, z=2.8)), attach_to=self._parent)
501         # We need to pass the lambda a weak reference to self to avoid circular
502         # reference.
503         weak_self = weakref.ref(self)
504         self.sensor.listen(lambda event: GnssSensor._on_gnss_event(weak_self, event))
505
506     @staticmethod
507     def _on_gnss_event(weak_self, event):
508         self = weak_self()
509         if not self:
510             return
511         self.lat = event.latitude
512         self.lon = event.longitude
513
514
515 # ==============================================================================
516 # -- IMUSensor -----------------------------------------------------------------
517 # ==============================================================================
518
519
520 class IMUSensor(object):
521     def __init__(self, parent_actor):
522         self.sensor = None
523         self._parent = parent_actor
524         self.accelerometer = (0.0, 0.0, 0.0)
525         self.gyroscope = (0.0, 0.0, 0.0)
526         self.compass = 0.0
527         world = self._parent.get_world()
528         bp = world.get_blueprint_library().find('sensor.other.imu')
529         self.sensor = world.spawn_actor(
530             bp, carla.Transform(), attach_to=self._parent)
531         # We need to pass the lambda a weak reference to self to avoid circular
532         # reference.
533         weak_self = weakref.ref(self)
534         self.sensor.listen(
535             lambda sensor_data: IMUSensor._IMU_callback(weak_self, sensor_data))
536
```

```
    @staticmethod
    def _IMU_callback(weak_self, sensor_data):
        self = weak_self()
        if not self:
            return
        limits = (-99.9, 99.9)
        self.accelerometer = (
            max(limits[0], min(limits[1], sensor_data.accelerometer.x)),
            max(limits[0], min(limits[1], sensor_data.accelerometer.y)),
            max(limits[0], min(limits[1], sensor_data.accelerometer.z)))
        self.gyroscope = (
            max(limits[0], min(limits[1], math.degrees(sensor_data.gyroscope.x))),
            max(limits[0], min(limits[1], math.degrees(sensor_data.gyroscope.y))),
            max(limits[0], min(limits[1], math.degrees(sensor_data.gyroscope.z))))
        self.compass = math.degrees(sensor_data.compass)


# ==============================================================================
# -- RadarSensor ---------------------------------------------------------------
# ==============================================================================


class RadarSensor(object):
    def __init__(self, parent_actor):
        self.sensor = None
        self._parent = parent_actor
        bound_x = 0.5 + self._parent.bounding_box.extent.x
        bound_y = 0.5 + self._parent.bounding_box.extent.y
        bound_z = 0.5 + self._parent.bounding_box.extent.z

        self.velocity_range = 7.5 # m/s
        world = self._parent.get_world()
        self.debug = world.debug
        bp = world.get_blueprint_library().find('sensor.other.radar')
        bp.set_attribute('horizontal_fov', str(35))
        bp.set_attribute('vertical_fov', str(20))
        self.sensor = world.spawn_actor(
            bp,
            carla.Transform(
                carla.Location(x=bound_x + 0.05, z=bound_z+0.05),
                carla.Rotation(pitch=5)),
            attach_to=self._parent)
        # We need a weak reference to self to avoid circular reference.
        weak_self = weakref.ref(self)
        self.sensor.listen(
            lambda radar_data: RadarSensor._Radar_callback(weak_self, radar_data))

    @staticmethod
    def _Radar_callback(weak_self, radar_data):
        self = weak_self()
        if not self:
            return
        # To get a numpy [[vel, altitude, azimuth, depth],...[,,,]]:
        # points = np.frombuffer(radar_data.raw_data, dtype=np.dtype('f4'))
        # points = np.reshape(points, (len(radar_data), 4))

        current_rot = radar_data.transform.rotation
        for detect in radar_data:
            azi = math.degrees(detect.azimuth)
            alt = math.degrees(detect.altitude)
            # The 0.25 adjusts a bit the distance so the dots can
            # be properly seen
            fw_vec = carla.Vector3D(x=detect.depth - 0.25)
            carla.Transform(
                carla.Location(),
                carla.Rotation(
                    pitch=current_rot.pitch + alt,
                    yaw=current_rot.yaw + azi,
                    roll=current_rot.roll)).transform(fw_vec)

            def clamp(min_v, max_v, value):
                return max(min_v, min(value, max_v))

            norm_velocity = detect.velocity / self.velocity_range # range [-1, 1]
            r = int(clamp(0.0, 1.0, 1.0 - norm_velocity) * 255.0)
            g = int(clamp(0.0, 1.0, 1.0 - abs(norm_velocity)) * 255.0)
            b = int(abs(clamp(- 1.0, 0.0, - 1.0 - norm_velocity)) * 255.0)
            self.debug.draw_point(
                radar_data.transform.location + fw_vec,
                size=0.075,
                life_time=0.06,
                persistent_lines=False,
                color=carla.Color(r, g, b))

# ==============================================================================
# -- CameraManager -------------------------------------------------------------
# ==============================================================================


class CameraManager(object):
    def __init__(self, parent_actor, hud, gamma_correction):
        self.sensor = None
        self.surface = None
        self._parent = parent_actor
        self.hud = hud
        self.recording = False
        Attachment = carla.AttachmentType

        self._camera_transforms = [
            # Third-person
            (carla.Transform(carla.Location(x=-5.0, z=2.0), carla.Rotation(pitch=6.0)), Attachment.SpringArm),
            # Watch back
            (carla.Transform(carla.Location(x=4.0, z=2.0), carla.Rotation(yaw=0, pitch=6)), Attachment.SpringArm),
            # Watch to the left
            (carla.Transform(carla.Location(x=0, z=2.0, y=3), carla.Rotation(yaw=-90, pitch=0)), Attachment.Rigid),
            # Watch to the right
            (carla.Transform(carla.Location(x=0, z=2.0, y=-3), carla.Rotation(yaw=90, pitch=0)), Attachment.Rigid),
            # First-person
            (carla.Transform(carla.Location(y=-0.42, x=0.03, z=1.2), carla.Rotation(yaw=0, roll=0, pitch=-10)), Attachment.Rigid)]

        self.transform_index = 1
        self.sensors = [
            ['sensor.camera.rgb', cc.Raw, 'Camera RGB', {}],
            ['sensor.camera.depth', cc.Raw, 'Camera Depth (Raw)', {}],
            ['sensor.camera.depth', cc.Depth, 'Camera Depth (Gray Scale)', {}],
            ['sensor.camera.depth', cc.LogarithmicDepth, 'Camera Depth (Logarithmic Gray Scale)', {}],
            ['sensor.camera.semantic_segmentation', cc.Raw, 'Camera Semantic Segmentation (Raw)', {}],
            ['sensor.camera.semantic_segmentation', cc.CityScapesPalette, 'Camera Semantic Segmentation (CityScapes Palette)', {}],
            ['sensor.camera.instance_segmentation', cc.CityScapesPalette, 'Camera Instance Segmentation (CityScapes Palette)', {}],
            ['sensor.camera.instance_segmentation', cc.Raw, 'Camera Instance Segmentation (Raw)', {}],
            ['sensor.lidar.ray_cast', None, 'Lidar (Ray-Cast)', {'range': '50'}],
            ['sensor.camera.dvs', cc.Raw, 'Dynamic Vision Sensor', {}],
            ['sensor.camera.rgb', cc.Raw, 'Camera RGB Distorted',
                {'lens_circle_multiplier': '3.0',
                'lens_circle_falloff': '3.0',
                'chromatic_aberration_intensity': '0.5',
                'chromatic_aberration_offset': '0'}],
            ['sensor.camera.optical_flow', cc.Raw, 'Optical Flow', {}],
        ]
        world = self._parent.get_world()
```

```
        bp_library = world.get_blueprint_library()
        for item in self.sensors:
            bp = bp_library.find(item[0])
            if item[0].startswith('sensor.camera'):
                bp.set_attribute('image_size_x', str(hud.dim[0]))
                bp.set_attribute('image_size_y', str(hud.dim[1]))
                if bp.has_attribute('gamma'):
                    bp.set_attribute('gamma', str(gamma_correction))
                for attr_name, attr_value in item[3].items():
                    bp.set_attribute(attr_name, attr_value)
            elif item[0].startswith('sensor.lidar'):
                self.lidar_range = 50

                for attr_name, attr_value in item[3].items():
                    bp.set_attribute(attr_name, attr_value)
                    if attr_name == 'range':
                        self.lidar_range = float(attr_value)

            item.append(bp)
        self.index = None

    def toggle_camera(self):
        self.transform_index = (self.transform_index + 1) % len(self._camera_transforms)
        self.set_sensor(self.index, notify=False, force_respawn=True)

    def set_sensor(self, index, notify=True, force_respawn=False):
        index = index % len(self.sensors)
        needs_respawn = True if self.index is None else \
            (force_respawn or (self.sensors[index][2] != self.sensors[self.index][2]))
        if needs_respawn:
            if self.sensor is not None:
                self.sensor.destroy()
                self.surface = None
            self.sensor = self._parent.get_world().spawn_actor(
                self.sensors[index][-1],
                self._camera_transforms[self.transform_index][0],
                attach_to=self._parent,
                attachment_type=self._camera_transforms[self.transform_index][1])
            # We need to pass the lambda a weak reference to self to avoid
            # circular reference.
            weak_self = weakref.ref(self)
            self.sensor.listen(lambda image: CameraManager._parse_image(weak_self, image))
        if notify:
            self.hud.notification(self.sensors[index][2])
        self.index = index

    def next_sensor(self):
        self.set_sensor(self.index + 1)

    def toggle_recording(self):
        self.recording = not self.recording
        self.hud.notification('Recording %s' % ('On' if self.recording else 'Off'))

    def render(self, display):
        if self.surface is not None:
            display.blit(self.surface, (0, 0))

    @staticmethod
    def _parse_image(weak_self, image):
        self = weak_self()
        if not self:
            return
        if self.sensors[self.index][0].startswith('sensor.lidar'):
            points = np.frombuffer(image.raw_data, dtype=np.dtype('f4'))
            points = np.reshape(points, (int(points.shape[0] / 4), 4))
            lidar_data = np.array(points[:, :2])
            lidar_data *= min(self.hud.dim) / (2.0 * self.lidar_range)
            lidar_data += (0.5 * self.hud.dim[0], 0.5 * self.hud.dim[1])
            lidar_data = np.fabs(lidar_data)  # pylint: disable=E1111
            lidar_data = lidar_data.astype(np.int32)
            lidar_data = np.reshape(lidar_data, (-1, 2))
            lidar_img_size = (self.hud.dim[0], self.hud.dim[1], 3)
            lidar_img = np.zeros((lidar_img_size), dtype=np.uint8)
            lidar_img[tuple(lidar_data.T)] = (255, 255, 255)
            self.surface = pygame.surfarray.make_surface(lidar_img)
        elif self.sensors[self.index][0].startswith('sensor.camera.dvs'):
            # Example of converting the raw_data from a carla.DVSEventArray
            # sensor into a NumPy array and using it as an image
            dvs_events = np.frombuffer(image.raw_data, dtype=np.dtype([
                ('x', np.uint16), ('y', np.uint16), ('t', np.int64), ('pol', np.bool)]))
            dvs_img = np.zeros((image.height, image.width, 3), dtype=np.uint8)
            # Blue is positive, red is negative
            dvs_img[dvs_events[:]['y'], dvs_events[:]['x'], dvs_events[:]['pol'] * 2] = 255
            self.surface = pygame.surfarray.make_surface(dvs_img.swapaxes(0, 1))
        elif self.sensors[self.index][0] startswith('sensor.camera.optical_flow'):
            image = image.get_color_coded_flow()
            array = np.frombuffer(image.raw_data, dtype=np.dtype("uint8"))
            array = np.reshape(array, (image.height, image.width, 4))
            array = array[:, :, :3]
            array = array[:, :, ::-1]
            self.surface = pygame.surfarray.make_surface(array.swapaxes(0, 1))
        else:
            image.convert(self.sensors[self.index][1])
            array = np.frombuffer(image.raw_data, dtype=np.dtype("uint8"))
            array = np.reshape(array, (image.height, image.width, 4))
            array = array[:, :, :3]
            array = array[:, :, ::-1]
            self.surface = pygame.surfarray.make_surface(array.swapaxes(0, 1))
        if self.recording:
            image.save_to_disk('_out/%08d' % image.frame)


def transform_to_carla_locations(path_locations):
    locations = []
    for loc in path_locations:
        locations.append(carla.Location(x=float(loc["x"]), y=float(loc["y"]), z=float(loc["z"])))
    return locations

def parse_keys(world):
    for event in pygame.event.get():
        if event.type == pygame.QUIT or (event.type == pygame.KEYUP and event.key == K_ESCAPE):
            world.finishScenario()
            return True
        elif event.type == pygame.KEYUP:
            if event.key == K_TAB:
                world.camera_manager.toggle_camera()


def game_loop(args, scenario, path_details):
    pygame.init()
    pygame.font.init()
    world = None
    original_settings = None
    try:
        client = carla.Client(args.host, args.port)
        client.set_timeout(20.0)
        sim_world = client.get_world()
        if args.sync:
            original_settings = sim_world.get_settings()
            settings = sim_world.get_settings()
```

```
            if not settings.synchronous_mode:
                settings.synchronous_mode = True
                settings.fixed_delta_seconds = 0.05
            sim_world.apply_settings(settings)
            traffic_manager = client.get_trafficmanager()
            traffic_manager.set_synchronous_mode(True)

        display = pygame.display.set_mode(
            (args.width, args.height),
            pygame.HWSURFACE | pygame.DOUBLEBUF)
        display.fill((0,0,0))
        pygame.display.flip()
        scenario.setup(traffic_manager, sim_world)
        scenario.spawn(client)
        logger.info("The scenario was set. Spawning the hero vehicle and launching the simulation")
        client.set_timeout(10.0)
        hud = HUD(args.width, args.height)
        try:
            world = World(sim_world, hud, args, scenario, client, path_details)
        except Exception as e:
            logger.error("Could not initiate the simulation world correctly", exc_info=True)
        if args.sync:
            sim_world.tick()
        else:
            sim_world.wait_for_tick()

        clock = pygame.time.Clock()
        map = sim_world.get_map()
        time.sleep(3)
        try:
            locations = transform_to_carla_locations(path_details["path_checkpoints"])
            player = None
            actors = sim_world.get_actors()
            for actor in actors:
                if actor.attributes.get('role_name') == 'hero':
                    player = actor

            # traffic_manager.set_path(player, locations[1:])
            traffic_manager.auto_lane_change(player, True)
            traffic_manager.update_vehicle_lights(player, True)
            agent = BehaviorAgent(player, behavior="aggressive")
            #player.set_autopilot(True)
            world.startScenario()

        except Exception as e:
            logger.error("Could not setup the hero vehicle correctly", exc_info=True)
            raise e

        current_destination_index = 1
        agent.set_destination(locations[current_destination_index])
        while True:
            try:
                if agent.done():
                    if current_destination_index == len(locations)-1:
                        return
                    else:
                        current_destination_index += 1
                        agent.set_destination(locations[current_destination_index])
                player.apply_control(agent.run_step())

                if args.sync:
                    sim_world.tick()
                clock.tick_busy_loop(FPS)
                if world.tick():
                    return
                if parse_keys(world):
                    return
                world.render(display)
                pygame.display.flip()
            except Exception as e:
                raise e
    except Exception as e:
        logger.error("Something went wrong trying to launch the self_driver.py", exc_info=True)
    finally:
        if(scenario.populated):
            scenario.finish(client)
        if(world and world.manager):
            logger.info('Total time simulated: {}s'.format(str(world.simulation_time/1000)))

        if original_settings:
            sim_world.apply_settings(original_settings)

        if world is not None:
            world.destroy()

        pygame.quit()

argparser = argparse.ArgumentParser(
    description='CARLA AI Client')
argparser.add_argument(
    '-v', '--verbose',
    action='store_true',
    dest='debug',
    help='print debug information')
argparser.add_argument(
    '--host',
    metavar='H',
    default='127.0.0.1',
    help='IP of the host server (default: 127.0.0.1)')
argparser.add_argument(
    '-p', '--port',
    metavar='P',
    default=2000,
    type=int,
    help='TCP port to listen to (default: 2000)')
argparser.add_argument(
    '-a', '--autopilot',
    action='store_true',
    help='enable autopilot')
argparser.add_argument(
    '--res',
    metavar='WIDTHxHEIGHT',
    default='1280x720',
    help='window resolution (default: 1280x720)')
argparser.add_argument(
    '--filter', '-f',
    metavar='PATTERN',
    default='vehicle.*',
    help='actor filter (default: "vehicle.*")')
argparser.add_argument(
    '--generation',
    metavar='G',
    default='2',
    help='restrict to certain actor generation (values: "1","2","All" - default: "2")')
argparser.add_argument(
    '--rolename',
    metavar='NAME',
    default='hero',
    help='actor role name (default: "hero")')
argparser.add_argument(
```

```
    '--gamma',
    default=2.2,
    type=float,
    help='Gamma correction of the camera (default: 2.2)')
argparser.add_argument(
    '--sync',
    action='store_true',
    help='Activate synchronous mode execution')
argparser.add_argument(
    '--red',
    default="0",
    help='Red part of the RGB color palette to paint the hero car')
argparser.add_argument(
    '--green',
    default="0",
    help='Green part of the RGB color palette to paint the hero car')
argparser.add_argument(
    '--blue',
    default="0",
    help='Blue part of the RGB color palette to paint the hero car')
argparser.add_argument(
    '-n', '--name',
    metavar='F',
    default="example",
    help='Participant name (name)')
argparser.add_argument(
    '-s', '--scenario',
    metavar='F',
    default="example",
    help='Scenario name (name)')
argparser.add_argument("s_b64")
argparser.add_argument("p_b64")
args = argparser.parse_args()

logger = logging.getLogger("self-driver-thread")
logging.basicConfig(level=logging.INFO, filename="../../data/recordings/{}/session_logs.log".format(args.name), filemode="a", format="%(asctime)s - %(name)s - %(

def main():
    args.width, args.height = [int(x) for x in args.res.split('x')]

    print(__doc__)

    try:

        scenario_b64 = args.s_b64
        path_b64 = args.p_b64

        scenario_bytes = base64.b64decode(scenario_b64.encode())
        path_bytes = base64.b64decode(path_b64.encode())

        scenario = pickle.loads(scenario_bytes)
        path_details = pickle.loads(path_bytes)
        game_loop(args, scenario, path_details)

    except KeyboardInterrupt as e:
        logger.warning('Keyboard interupt while running self_driver.py script')
    except Exception as e:
        logger.error("Something went wrong.", exc_info=True)


if __name__ == '__main__':

    main()
```

File: ./software/carla_scripts/driver_steeringwheel.py

```
#!/usr/bin/env python

# Copyright (c) 2019 Intel Labs
#
# This work is licensed under the terms of the MIT license.
# For a copy, see <https://opensource.org/licenses/MIT>.

# Allows controlling a vehicle with a keyboard. For a simpler and more
# documented example, please take a look at tutorial.py.

"""
Welcome to CARLA manual control with steering wheel Logitech G29.

To drive start by preshing the brake pedal.
Change your wheel_config.ini according to your steering wheel.

To find out the values of your steering wheel use jstest-gtk in Ubuntu.

"""

from __future__ import print_function


# ==============================================================================
# -- find carla module ---------------------------------------------------------
# ==============================================================================


import glob
import os
import sys

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass


# ==============================================================================
# -- imports -------------------------------------------------------------------
# ==============================================================================


import carla

from carla import ColorConverter as cc

import argparse
import collections
import datetime
import logging
import math
import random
import re
import weakref
from recording.manager import Manager
```

```
from config import FPS
import pickle
import base64

if sys.version_info >= (3, 0):

    from configparser import ConfigParser

else:

    from ConfigParser import RawConfigParser as ConfigParser

try:
    import pygame
    from pygame.locals import KMOD_CTRL
    from pygame.locals import KMOD_SHIFT
    from pygame.locals import K_0
    from pygame.locals import K_9
    from pygame.locals import K_BACKQUOTE
    from pygame.locals import K_BACKSPACE
    from pygame.locals import K_COMMA
    from pygame.locals import K_DOWN
    from pygame.locals import K_ESCAPE
    from pygame.locals import K_F1
    from pygame.locals import K_LEFT
    from pygame.locals import K_PERIOD
    from pygame.locals import K_RIGHT
    from pygame.locals import K_SLASH
    from pygame.locals import K_SPACE
    from pygame.locals import K_TAB
    from pygame.locals import K_UP
    from pygame.locals import K_a
    from pygame.locals import K_c
    from pygame.locals import K_d
    from pygame.locals import K_h
    from pygame.locals import K_m
    from pygame.locals import K_p
    from pygame.locals import K_q
    from pygame.locals import K_r
    from pygame.locals import K_s
    from pygame.locals import K_w
except ImportError:
    raise RuntimeError('cannot import pygame, make sure pygame package is installed')

try:
    import numpy as np
except ImportError:
    raise RuntimeError('cannot import numpy, make sure numpy package is installed')


# ==============================================================================
# -- Global functions ----------------------------------------------------------
# ==============================================================================


def find_weather_presets():
    rgx = re.compile('.+?(?:(?<=[a-z])(?=[A-Z])|(?<=[A-Z])(?=[A-Z][a-z])|$)')
    name = lambda x: ' '.join(m.group(0) for m in rgx.finditer(x))
    presets = [x for x in dir(carla.WeatherParameters) if re.match('[A-Z].+', x)]
    return [(getattr(carla.WeatherParameters, x), name(x)) for x in presets]


def get_actor_display_name(actor, truncate=250):
    name = ' '.join(actor.type_id.replace('_', '.').title().split('.')[1:])
    return (name[:truncate - 1] + u'\u2026') if len(name) > truncate else name


# ==============================================================================
# -- World ---------------------------------------------------------------------
# ==============================================================================


class World(object):
    def __init__(self, carla_world, hud, args, scenario, client, path_details):
        self.world = carla_world
        self.sync = args.sync
        self.path_details = path_details
        self.scenario = scenario
        self.client = client
        self.scenario_name = args.scenario
        self.participant_name = args.name

        self.spawn_point = None
        self.init_spawn_point(path_details["start_location"])
        self.hud = hud
        self.player = None
        self.collision_sensor = None
        self.lane_invasion_sensor = None
        self.gnss_sensor = None
        self.camera_manager = None
        self._weather_presets = find_weather_presets()
        self._weather_index = 0
        self._actor_filter = args.filter
        self._gamma = args.gamma
        self.restart(args)

        # All about the simulation
        self.simulation_clock = None
        self.simulation_start_tick = 0
        self.simulation_time = 0
        self.manager = None

        self.world.on_tick(hud.on_world_tick)

    def init_spawn_point(self, start_location):
        try:
            self.spawn_point = carla.Transform(carla.Location(x=start_location["x"], y=start_location["y"], z=start_location["z"]), carla.Rotation(pitch=0.0, ya
        except Exception as e:
            logger.error("Could not initiate spawn location for the driver", e)

    def restart(self, args):
        # Keep same camera config if the camera manager exists.
        cam_index = self.camera_manager.index if self.camera_manager is not None else 0
        cam_pos_index = self.camera_manager.transform_index if self.camera_manager is not None else 0
        # Get a random blueprint.
        blueprint = random.choice(self.world.get_blueprint_library().filter(self._actor_filter))
        blueprint.set_attribute('role_name', 'hero')
        if blueprint.has_attribute('color'):
            blueprint.set_attribute('color', '{},{},{}'.format(args.red, args.green, args.blue))
        # Spawn the player.
        if self.player is not None:
            spawn_point = self.player.get_transform()
            spawn_point.location.z += 2.0
            spawn_point.rotation.roll = 0.0
            spawn_point.rotation.pitch = 0.0
            self.destroy()
            self.player = self.world.try_spawn_actor(blueprint, spawn_point)
        while self.player is None:
            self.player = self.world.try_spawn_actor(blueprint, self.spawn_point)
        # Set up the sensors.
```

```
        self.collision_sensor = CollisionSensor(self.player, self.hud)
        self.lane_invasion_sensor = LaneInvasionSensor(self.player, self.hud)
        self.gnss_sensor = GnssSensor(self.player)
        self.camera_manager = CameraManager(self.player, self.hud, args.gamma)
        self.camera_manager.transform_index = cam_pos_index
        self.camera_manager.set_sensor(cam_index, notify=False)
        actor_type = get_actor_display_name(self.player)
        self.hud.notification(actor_type)

        if self.sync:
            self.world.tick()
        else:
            self.world.wait_for_tick()

    def next_weather(self, reverse=False):
        self._weather_index += -1 if reverse else 1
        self._weather_index %= len(self._weather_presets)
        preset = self._weather_presets[self._weather_index]
        self.hud.notification('Weather: %s' % preset[1])
        self.player.get_world().set_weather(preset[0])

    def startScenario(self):
        if self.simulation_clock is None:
            logger.info("Starting the scenario")
            self.scenario.start()
            self.hud.notification('The simulation has started')
            self.manager = Manager(self.path_details["path_checkpoints"], self.world, self.player, self.participant_name, self.scenario_name)
            #self.collision_sensor = self.manager.collision_sensor
            self.simulation_clock = pygame.time.Clock()
            self.simulation_start_tick = pygame.time.get_ticks()

    def finishScenario(self):
        logger.warning("Finish was NOT REACHED. Terminating the simulation.")
        self.manager.shut_down(False)
        self.scenario.finish(self.client)

    def tick(self):
        if self.simulation_clock is not None:
            self.simulation_clock.tick(60)
            self.simulation_time += self.simulation_clock.get_time()
            if self.manager.record(self.simulation_time):
                return True
            self.hud.tick(self, self.simulation_time/1000)
        else:
            self.hud.tick(self, 0)

    def render(self, display):
        self.camera_manager.render(display)
        self.hud.render(display)

    def destroy(self):
        sensors = [
            self.camera_manager.sensor,
            self.collision_sensor.sensor,
            self.lane_invasion_sensor.sensor,
            self.gnss_sensor.sensor]
        for sensor in sensors:
            if sensor is not None:
                sensor.stop()
                sensor.destroy()
        if self.player is not None:
            self.player.destroy()

# ==============================================================================
# -- DualControl -----------------------------------------------------------
# ==============================================================================


class DualControl(object):
    def __init__(self, world, start_in_autopilot):
        self._autopilot_enabled = start_in_autopilot
        if isinstance(world.player, carla.Vehicle):
            self._control = carla.VehicleControl()
            self._lights = carla.VehicleLightState.NONE
            world.player.set_autopilot(self._autopilot_enabled)
        elif isinstance(world.player, carla.Walker):
            self._control = carla.WalkerControl()
            self._autopilot_enabled = False
            self._rotation = world.player.get_transform().rotation
        else:
            raise NotImplementedError("Actor type not supported")
        self._steer_cache = 0.0
        world.hud.notification("Press 'H' or '?' for help.", seconds=4.0)

        # initialize steering wheel
        pygame.joystick.init()
        self.word = world
        joystick_count = pygame.joystick.get_count()
        if joystick_count > 1:
            raise ValueError("Please Connect Just One Joystick")

        self._joystick = pygame.joystick.Joystick(0)
        self._joystick.init()

        self._parser = ConfigParser()
        self._parser.read('wheel_config.ini')
        self._steer_idx = int(
            self._parser.get('G29 Racing Wheel', 'steering_wheel'))
        self._throttle_idx = int(
            self._parser.get('G29 Racing Wheel', 'throttle'))
        self._brake_idx = int(self._parser.get('G29 Racing Wheel', 'brake'))
        self._reverse_idx = int(self._parser.get('G29 Racing Wheel', 'reverse'))
        self._handbrake_idx = int(
            self._parser.get('G29 Racing Wheel', 'handbrake'))

    def parse_events(self, client, world, clock, sync_mode):
        if isinstance(self._control, carla.VehicleControl):
            current_lights = self._lights
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                return True
            elif event.type == pygame.JOYBUTTONDOWN:
                if event.button == 1:
                    world.hud.toggle_info()
                elif event.button == 19:
                    world.camera_manager.toggle_camera()
                elif event.button == 19:
                    world.camera_manager.toggle_camera_backward()
                elif event.button == self._reverse_idx:
                    self._control.gear = 1 if self._control.reverse else -1
                elif event.button == 23:
                    if not self._lights & carla.VehicleLightState.Position:
                        world.hud.notification("Position lights")
                        current_lights |= carla.VehicleLightState.Position
                    else:
                        world.hud.notification("Low beam lights")
                        current_lights |= carla.VehicleLightState.LowBeam
                    if self._lights & carla.VehicleLightState.LowBeam:
                        world.hud.notification("Fog lights")
                        current_lights |= carla.VehicleLightState.Fog
```

```
                    if self._lights & carla.VehicleLightState.Fog:
                        world.hud.notification("Lights off")
                        current_lights ^= carla.VehicleLightState.Position
                        current_lights ^= carla.VehicleLightState.LowBeam
                        current_lights ^= carla.VehicleLightState.Fog
                elif event.button == 3 and not world.simulation_clock:
                    world.startScenario()
                elif event.button == 3 and world.simulation_clock:
                    world.finishScenario()
                    return True
                elif event.button == 5:
                    current_lights ^= carla.VehicleLightState.LeftBlinker
                elif event.button == 4:
                    current_lights ^= carla.VehicleLightState.RightBlinker

            elif event.type == pygame.KEYUP:
                if self._is_quit_shortcut(event.key):
                    return True
                elif event.key == K_BACKSPACE and not world.simulation_clock:
                    world.startScenario()
                elif event.key == K_BACKSPACE and world.simulation_clock:
                    world.finishScenario()
                    return True
                elif event.key == K_F1:
                    world.hud.toggle_info()
                elif event.key == K_h or (event.key == K_SLASH and pygame.key.get_mods() & KMOD_SHIFT):
                    world.hud.help.toggle()
                elif event.key == K_TAB:
                    world.camera_manager.toggle_camera()
                elif event.key == K_c and pygame.key.get_mods() & KMOD_SHIFT:
                    world.next_weather(reverse=True)
                elif event.key == K_c:
                    world.next_weather()
                elif event.key == K_BACKQUOTE:
                    world.camera_manager.next_sensor()
                elif event.key > K_0 and event.key <= K_9:
                    world.camera_manager.set_sensor(event.key - 1 - K_0)
                elif event.key == K_r:
                    world.camera_manager.toggle_recording()
                if isinstance(self._control, carla.VehicleControl):
                    if event.key == K_q:
                        self._control.gear = 1 if self._control.reverse else -1
                    elif event.key == K_m:
                        self._control.manual_gear_shift = not self._control.manual_gear_shift
                        self._control.gear = world.player.get_control().gear
                        world.hud.notification('%s Transmission' %
                                               ('Manual' if self._control.manual_gear_shift else 'Automatic'))
                    elif self._control.manual_gear_shift and event.key == K_COMMA:
                        self._control.gear = max(-1, self._control.gear - 1)
                    elif self._control.manual_gear_shift and event.key == K_PERIOD:
                        self._control.gear = self._control.gear + 1
                    elif event.key == K_p:
                        self._autopilot_enabled = not self._autopilot_enabled
                        world.player.set_autopilot(self._autopilot_enabled)
                        world.hud.notification('Autopilot %s' % ('On' if self._autopilot_enabled else 'Off'))

        if not self._autopilot_enabled:
            if isinstance(self._control, carla.VehicleControl):
                self._parse_vehicle_keys(pygame.key.get_pressed(), clock.get_time())
                self._parse_vehicle_wheel()
                self._control.reverse = self._control.gear < 0

                if self._control.brake:
                    current_lights |= carla.VehicleLightState.Brake
                else: # Remove the Brake flag
                    current_lights &= ~carla.VehicleLightState.Brake
                if self._control.reverse:
                    current_lights |= carla.VehicleLightState.Reverse
                else: # Remove the Reverse flag
                    current_lights &= ~carla.VehicleLightState.Reverse
                if current_lights != self._lights: # Change the light state only if necessary
                    self._lights = current_lights
                    world.player.set_light_state(carla.VehicleLightState(self._lights))

            elif isinstance(self._control, carla.WalkerControl):
                self._parse_walker_keys(pygame.key.get_pressed(), clock.get_time())
            world.player.apply_control(self._control)

    def _parse_vehicle_keys(self, keys, milliseconds):
        self._control.throttle = 1.0 if keys[K_UP] or keys[K_w] else 0.0
        steer_increment = 5e-4 * milliseconds
        if keys[K_LEFT] or keys[K_a]:
            self._steer_cache -= steer_increment
        elif keys[K_RIGHT] or keys[K_d]:
            self._steer_cache += steer_increment
        else:
            self._steer_cache = 0.0
        self._steer_cache = min(0.7, max(-0.7, self._steer_cache))
        self._control.steer = round(self._steer_cache, 1)
        self._control.brake = 1.0 if keys[K_DOWN] or keys[K_s] else 0.0
        self._control.hand_brake = keys[K_SPACE]

    def _parse_vehicle_wheel(self):
        numAxes = self._joystick.get_numaxes()
        jsInputs = [float(self._joystick.get_axis(i)) for i in range(numAxes)]
        # print (jsInputs)
        jsButtons = [float(self._joystick.get_button(i)) for i in
                     range(self._joystick.get_numbuttons())]

        # Custom function to map range of inputs [1, -1] to outputs [0, 1] i.e 1 from inputs means nothing is pressed
        # For the steering, it seems fine as it is
        K1 = 1.0  # 0.55
        steerCmd = K1 * math.tan(1.1 * jsInputs[self._steer_idx])

        K2 = 1.6  # 1.6
        throttleCmd = K2 + (2.05 * math.log10(
            -0.7 * jsInputs[self._throttle_idx] + 1.4) - 1.2) / 0.92
        if throttleCmd <= 0:
            throttleCmd = 0
        elif throttleCmd > 1:
            throttleCmd = 1

        brakeCmd = 1.6 + (2.05 * math.log10(
            -0.7 * jsInputs[self._brake_idx] + 1.4) - 1.2) / 0.92

        if brakeCmd <= 0:
            brakeCmd = 0
        elif brakeCmd > 1:
            brakeCmd = 1

        self._control.steer = steerCmd
        self._control.brake = brakeCmd
        self._control.throttle = throttleCmd

        #toggle = jsButtons[self._reverse_idx]

        self._control.hand_brake = bool(jsButtons[self._handbrake_idx])

    def _parse_walker_keys(self, keys, milliseconds):
        self._control.speed = 0.0
```

```
        if keys[K_DOWN] or keys[K_s]:
            self._control.speed = 0.0
        if keys[K_LEFT] or keys[K_a]:
            self._control.speed = .01
            self._rotation.yaw -= 0.08 * milliseconds
        if keys[K_RIGHT] or keys[K_d]:
            self._control.speed = .01
            self._rotation.yaw += 0.08 * milliseconds
        if keys[K_UP] or keys[K_w]:
            self._control.speed = 5.556 if pygame.key.get_mods() & KMOD_SHIFT else 2.778
        self._control.jump = keys[K_SPACE]
        self._rotation.yaw = round(self._rotation.yaw, 1)
        self._control.direction = self._rotation.get_forward_vector()

    @staticmethod
    def _is_quit_shortcut(key):
        return (key == K_ESCAPE) or (key == K_q and pygame.key.get_mods() & KMOD_CTRL)


# ==============================================================================
# -- HUD -----------------------------------------------------------------------
# ==============================================================================


class HUD(object):
    def __init__(self, width, height):
        self.dim = (width, height)
        font = pygame.font.Font(pygame.font.get_default_font(), 20)
        font_name = 'courier' if os.name == 'nt' else 'mono'
        fonts = [x for x in pygame.font.get_fonts() if font_name in x]
        default_font = 'ubuntumono'
        mono = default_font if default_font in fonts else fonts[0]
        mono = pygame.font.match_font(mono)
        self._font_mono = pygame.font.Font(mono, 12 if os.name == 'nt' else 14)
        self._notifications = FadingText(font, (width, 40), (0, height - 40))
        self.help = HelpText(pygame.font.Font(mono, 24), width, height)
        self.server_fps = 0
        self.frame = 0
        self.simulation_time = 0
        self._show_info = True
        self._info_text = []
        self._server_clock = pygame.time.Clock()

    def on_world_tick(self, timestamp):
        self._server_clock.tick()
        self.server_fps = self._server_clock.get_fps()
        self.frame = timestamp.frame
        self.simulation_time = timestamp.elapsed_seconds

    def tick(self, world, time):
        self._notifications.tick(world, time)
        if not self._show_info:
            return
        t = world.player.get_transform()
        v = world.player.get_velocity()
        c = world.player.get_control()
        heading = 'N' if abs(t.rotation.yaw) < 89.5 else ''
        heading += 'S' if abs(t.rotation.yaw) > 90.5 else ''
        heading += 'E' if 179.5 > t.rotation.yaw > 0.5 else ''
        heading += 'W' if -0.5 > t.rotation.yaw > -179.5 else ''
        colhist = world.collision_sensor.get_collision_history()
        collision = [colhist[x + self.frame - 200] for x in range(0, 200)]
        max_col = max(1.0, max(collision))
        collision = [x / max_col for x in collision]
        vehicles = world.world.get_actors().filter('vehicle.*')
        self._info_text = [
            'Vehicle: % 20s' % get_actor_display_name(world.player, truncate=20),
            'Route time: % 12s' % datetime.timedelta(seconds=int(time)),
            '',
            'Speed:   % 15.0f km/h' % (3.6 * math.sqrt(v.x**2 + v.y**2 + v.z**2)),
            '']
        if isinstance(c, carla.VehicleControl):
            self._info_text += [
                ('Throttle:', c.throttle, 0.0, 1.0),
                ('Steer:', c.steer, -1.0, 1.0),
                ('Brake:', c.brake, 0.0, 1.0),
                ('Reverse:', c.reverse),
                ('Hand brake:', c.hand_brake),
                ('Manual:', c.manual_gear_shift),
                'Gear:        %s' % {-1: 'R', 0: 'N'}.get(c.gear, c.gear)]
        elif isinstance(c, carla.WalkerControl):
            self._info_text += [
                ('Speed:', c.speed, 0.0, 5.556),
                ('Jump:', c.jump)]
        self._info_text += [
            '',
            'Collision:',
            collision,
            '']

    def toggle_info(self):
        self._show_info = not self._show_info

    def notification(self, text, seconds=2.0):
        self._notifications.set_text(text, seconds=seconds)

    def error(self, text):
        self._notifications.set_text('Error: %s' % text, (255, 0, 0))

    def render(self, display):
        if self._show_info:
            info_surface = pygame.Surface((220, self.dim[1]))
            info_surface.set_alpha(100)
            display.blit(info_surface, (0, 0))
            v_offset = 4
            bar_h_offset = 100
            bar_width = 106
            for item in self._info_text:
                if v_offset + 18 > self.dim[1]:
                    break
                if isinstance(item, list):
                    if len(item) > 1:
                        points = [(x + 8, v_offset + 8 + (1.0 - y) * 30) for x, y in enumerate(item)]
                        pygame.draw.lines(display, (255, 136, 0), False, points, 2)
                    item = None
                    v_offset += 18
                elif isinstance(item, tuple):
                    if isinstance(item[1], bool):
                        rect = pygame.Rect((bar_h_offset, v_offset + 8), (6, 6))
                        pygame.draw.rect(display, (255, 255, 255), rect, 0 if item[1] else 1)
                    else:
                        rect_border = pygame.Rect((bar_h_offset, v_offset + 8), (bar_width, 6))
                        pygame.draw.rect(display, (255, 255, 255), rect_border, 1)
                        f = (item[1] - item[2]) / (item[3] - item[2])
                        if item[2] < 0.0:
                            rect = pygame.Rect((bar_h_offset + f * (bar_width - 6), v_offset + 8), (6, 6))
                        else:
                            rect = pygame.Rect((bar_h_offset, v_offset + 8), (f * bar_width, 6))
                        pygame.draw.rect(display, (255, 255, 255), rect)
                    item = item[0]
```

```
                if item:  # At this point has to be a str.
                    surface = self._font_mono.render(item, True, (255, 255, 255))
                    display.blit(surface, (8, v_offset))
                v_offset += 18
        self._notifications.render(display)
        self.help.render(display)


# ==============================================================================
# -- FadingText ----------------------------------------------------------------
# ==============================================================================


class FadingText(object):
    def __init__(self, font, dim, pos):
        self.font = font
        self.dim = dim
        self.pos = pos
        self.seconds_left = 0
        self.surface = pygame.Surface(self.dim)

    def set_text(self, text, color=(255, 255, 255), seconds=2.0):
        text_texture = self.font.render(text, True, color)
        self.surface = pygame.Surface(self.dim)
        self.seconds_left = seconds
        self.surface.fill((0, 0, 0, 0))
        self.surface.blit(text_texture, (10, 11))

    def tick(self, _, time):
        delta_seconds = 1e-3 * time
        self.seconds_left = max(0.0, self.seconds_left - delta_seconds)
        self.surface.set_alpha(500.0 * self.seconds_left)

    def render(self, display):
        display.blit(self.surface, self.pos)


# ==============================================================================
# -- HelpText ------------------------------------------------------------------
# ==============================================================================


class HelpText(object):
    def __init__(self, font, width, height):
        lines = __doc__.split('\n')
        self.font = font
        self.dim = (680, len(lines) * 22 + 12)
        self.pos = (0.5 * width - 0.5 * self.dim[0], 0.5 * height - 0.5 * self.dim[1])
        self.seconds_left = 0
        self.surface = pygame.Surface(self.dim)
        self.surface.fill((0, 0, 0, 0))
        for n, line in enumerate(lines):
            text_texture = self.font.render(line, True, (255, 255, 255))
            self.surface.blit(text_texture, (22, n * 22))
            self._render = False
        self.surface.set_alpha(220)

    def toggle(self):
        self._render = not self._render

    def render(self, display):
        if self._render:
            display.blit(self.surface, self.pos)


# ==============================================================================
# -- CollisionSensor -----------------------------------------------------------
# ==============================================================================


class CollisionSensor(object):
    def __init__(self, parent_actor, hud):
        self.sensor = None
        self.history = []
        self._parent = parent_actor
        self.hud = hud
        world = self._parent.get_world()
        bp = world.get_blueprint_library().find('sensor.other.collision')
        self.sensor = world.spawn_actor(bp, carla.Transform(), attach_to=self._parent)
        # We need to pass the lambda a weak reference to self to avoid circular
        # reference.
        weak_self = weakref.ref(self)
        self.sensor.listen(lambda event: CollisionSensor._on_collision(weak_self, event))

    def get_collision_history(self):
        history = collections.defaultdict(int)
        for frame, intensity in self.history:
            history[frame] += intensity
        return history

    @staticmethod
    def _on_collision(weak_self, event):
        self = weak_self()
        if not self:
            return
        actor_type = get_actor_display_name(event.other_actor)
        self.hud.notification('Collision with %r' % actor_type)
        impulse = event.normal_impulse
        intensity = math.sqrt(impulse.x**2 + impulse.y**2 + impulse.z**2)
        self.history.append((event.frame, intensity))
        if len(self.history) > 4000:
            self.history.pop(0)


# ==============================================================================
# -- LaneInvasionSensor --------------------------------------------------------
# ==============================================================================


class LaneInvasionSensor(object):
    def __init__(self, parent_actor, hud):
        self.sensor = None
        self._parent = parent_actor
        self.hud = hud
        world = self._parent.get_world()
        bp = world.get_blueprint_library().find('sensor.other.lane_invasion')
        self.sensor = world.spawn_actor(bp, carla.Transform(), attach_to=self._parent)
        # We need to pass the lambda a weak reference to self to avoid circular
        # reference.
        weak_self = weakref.ref(self)
        self.sensor.listen(lambda event: LaneInvasionSensor._on_invasion(weak_self, event))

    @staticmethod
    def _on_invasion(weak_self, event):
        self = weak_self()
        if not self:
            return
        lane_types = set(x.type for x in event.crossed_lane_markings)
        text = ['%r' % str(x).split()[-1] for x in lane_types]
        self.hud.notification('Crossed line %s' % ' and '.join(text))
```

```
710
711 # ==============================================================================
712 # -- GnssSensor ----------------------------------------------------------------
713 # ==============================================================================
714
715
716 class GnssSensor(object):
717     def __init__(self, parent_actor):
718         self.sensor = None
719         self._parent = parent_actor
720         self.lat = 0.0
721         self.lon = 0.0
722         world = self._parent.get_world()
723         bp = world.get_blueprint_library().find('sensor.other.gnss')
724         self.sensor = world.spawn_actor(bp, carla.Transform(carla.Location(x=1.0, z=2.8)), attach_to=self._parent)
725         # We need to pass the lambda a weak reference to self to avoid circular
726         # reference.
727         weak_self = weakref.ref(self)
728         self.sensor.listen(lambda event: GnssSensor._on_gnss_event(weak_self, event))
729
730     @staticmethod
731     def _on_gnss_event(weak_self, event):
732         self = weak_self()
733         if not self:
734             return
735         self.lat = event.latitude
736         self.lon = event.longitude
737
738
739 # ==============================================================================
740 # -- CameraManager -------------------------------------------------------------
741 # ==============================================================================
742
743
744 class CameraManager(object):
745     def __init__(self, parent_actor, hud, gamma_correction):
746         self.sensor = None
747         self.surface = None
748         self._parent = parent_actor
749         self.hud = hud
750         self.recording = False
751         self._camera_transforms = [
752             carla.Transform(carla.Location(x=-5.5, z=2.8), carla.Rotation(pitch=-15)),
753             carla.Transform(carla.Location(x=1.6, z=1.7))]
754         self.transform_index = 1
755         self.sensors = [
756             ['sensor.camera.rgb', cc.Raw, 'Camera RGB'],
757             ['sensor.camera.depth', cc.Raw, 'Camera Depth (Raw)'],
758             ['sensor.camera.depth', cc.Depth, 'Camera Depth (Gray Scale)'],
759             ['sensor.camera.depth', cc.LogarithmicDepth, 'Camera Depth (Logarithmic Gray Scale)'],
760             ['sensor.camera.semantic_segmentation', cc.Raw, 'Camera Semantic Segmentation (Raw)'],
761             ['sensor.camera.semantic_segmentation', cc.CityScapesPalette,
762                 'Camera Semantic Segmentation (CityScapes Palette)'],
763             ['sensor.lidar.ray_cast', None, 'Lidar (Ray-Cast)']]
764         world = self._parent.get_world()
765         bp_library = world.get_blueprint_library()
766         for item in self.sensors:
767             bp = bp_library.find(item[0])
768             if item[0].startswith('sensor.camera'):
769                 bp.set_attribute('image_size_x', str(hud.dim[0]))
770                 bp.set_attribute('image_size_y', str(hud.dim[1]))
771             elif item[0].startswith('sensor.lidar'):
772                 bp.set_attribute('range', '50')
773             item.append(bp)
774         self.index = None
775
776     def toggle_camera(self):
777         self.transform_index = (self.transform_index + 1) % len(self._camera_transforms)
778         self.sensor.set_transform(self._camera_transforms[self.transform_index])
779
780     def toggle_camera_backward(self):
781         self.transform_index = (self.transform_index -1 ) % len(self._camera_transforms)
782         self.sensor.set_transform(self._camera_transforms[self.transform_index])
783
784     def set_sensor(self, index, notify=True):
785         index = index % len(self.sensors)
786         needs_respawn = True if self.index is None \
787             else self.sensors[index][0] != self.sensors[self.index][0]
788         if needs_respawn:
789             if self.sensor is not None:
790                 self.sensor.destroy()
791                 self.surface = None
792             self.sensor = self._parent.get_world().spawn_actor(
793                 self.sensors[index][-1],
794                 self._camera_transforms[self.transform_index],
795                 attach_to=self._parent)
796             # We need to pass the lambda a weak reference to self to avoid
797             # circular reference.
798             weak_self = weakref.ref(self)
799             self.sensor.listen(lambda image: CameraManager._parse_image(weak_self, image))
800         if notify:
801             self.hud.notification(self.sensors[index][2])
802         self.index = index
803
804     def next_sensor(self):
805         self.set_sensor(self.index + 1)
806
807     def toggle_recording(self):
808         self.recording = not self.recording
809         self.hud.notification('Recording %s' % ('On' if self.recording else 'Off'))
810
811     def render(self, display):
812         if self.surface is not None:
813             display.blit(self.surface, (0, 0))
814
815     @staticmethod
816     def _parse_image(weak_self, image):
817         self = weak_self()
818         if not self:
819             return
820         if self.sensors[self.index][0].startswith('sensor.lidar'):
821             points = np.frombuffer(image.raw_data, dtype=np.dtype('f4'))
822             points = np.reshape(points, (int(points.shape[0] / 4), 4))
823             lidar_data = np.array(points[:, :2])
824             lidar_data *= min(self.hud.dim) / 100.0
825             lidar_data += (0.5 * self.hud.dim[0], 0.5 * self.hud.dim[1])
826             lidar_data = np.fabs(lidar_data) # pylint: disable=E1111
827             lidar_data = lidar_data.astype(np.int32)
828             lidar_data = np.reshape(lidar_data, (-1, 2))
829             lidar_img_size = (self.hud.dim[0], self.hud.dim[1], 3)
830             lidar_img = np.zeros(lidar_img_size)
831             lidar_img[tuple(lidar_data.T)] = (255, 255, 255)
832             self.surface = pygame.surfarray.make_surface(lidar_img)
833         else:
834             image.convert(self.sensors[self.index][1])
835             array = np.frombuffer(image.raw_data, dtype=np.dtype("uint8"))
836             array = np.reshape(array, (image.height, image.width, 4))
837             array = array[:, :, :3]
838             array = array[:, :, ::-1]
839             self.surface = pygame.surfarray.make_surface(array.swapaxes(0, 1))
```

```
        if self.recording:
            image.save_to_disk('_out/%08d' % image.frame)


# ==============================================================================
# -- game_loop() ---------------------------------------------------------------
# ==============================================================================


def game_loop(args, scenario, path_details):
    pygame.init()
    pygame.font.init()
    world = None
    original_settings = None
    try:
        client = carla.Client(args.host, args.port)
        client.set_timeout(20.0)
        sim_world = client.get_world()
        if args.sync:
            original_settings = sim_world.get_settings()
            settings = sim_world.get_settings()
            if not settings.synchronous_mode:
                settings.synchronous_mode = True
                settings.fixed_delta_seconds = 0.05
            sim_world.apply_settings(settings)
            traffic_manager = client.get_trafficmanager()
            traffic_manager.set_synchronous_mode(True)

        display = pygame.display.set_mode(
            (args.width, args.height),
            pygame.HWSURFACE | pygame.DOUBLEBUF)

        display.fill((0,0,0))
        pygame.display.flip()
        ### MY PART ###
        scenario.setup(traffic_manager, sim_world)
        scenario.spawn(client)
        logger.info("The scenario was set. Spawning the hero vehicle and launching the simulation")
        ###   ###   ###
        hud = HUD(args.width, args.height)
        try:
            world = World(sim_world, hud, args, scenario, client, path_details)
            controller = DualControl(world, args.autopilot)
        except Exeption as e:
            logger.error(e)
        if args.sync:
            sim_world.tick()
        else:
            sim_world.wait_for_tick()

        clock = pygame.time.Clock()
        while True:
            try:
                if args.sync:
                    sim_world.tick()
                clock.tick_busy_loop(FPS)
                if controller.parse_events(client, world, clock, args.sync):
                    return
                if world.tick():
                    return
                world.render(display)
                pygame.display.flip()
            except Exception as e:
                logger.error(e)
    except Exception as e:
        logger.error("Something went wrong trying to launch the driver.py", e)
    finally:
        ### MY PART ###
        if(scenario.populated):
            scenario.finish(client)
        if(world and world.manager):
            logger.info('Total time simulated: {}s'.format(str(world.simulation_time/1000)))
        ###   ###   ###

        if original_settings:
            sim_world.apply_settings(original_settings)

        if world is not None:
            world.destroy()

        pygame.quit()


# ==============================================================================
# -- main() --------------------------------------------------------------------
# ==============================================================================

argparser = argparse.ArgumentParser(
    description='CARLA Manual Control Client')
argparser.add_argument(
    '-v', '--verbose',
    action='store_true',
    dest='debug',
    help='print debug information')
argparser.add_argument(
    '--host',
    metavar='H',
    default='127.0.0.1',
    help='IP of the host server (default: 127.0.0.1)')
argparser.add_argument(
    '-p', '--port',
    metavar='P',
    default=2000,
    type=int,
    help='TCP port to listen to (default: 2000)')
argparser.add_argument(
    '-a', '--autopilot',
    action='store_true',
    help='enable autopilot')
argparser.add_argument(
    '--res',
    metavar='WIDTHxHEIGHT',
    default='1280x720',
    help='window resolution (default: 1280x720)')
argparser.add_argument(
    '--filter', '-f',
    metavar='PATTERN',
    default='vehicle.*',
    help='actor filter (default: "vehicle.*")')
argparser.add_argument(
'--generation',
metavar='G',
default='2',
help='restrict to certain actor generation (values: "1","2","All" - default: "2")')
argparser.add_argument(
    '--rolename',
    metavar='NAME',
    default='hero',
    help='actor role name (default: "hero")')
argparser.add_argument(
```

```
    '--gamma',
    default=2.2,
    type=float,
    help='Gamma correction of the camera (default: 2.2)')
argparser.add_argument(
    '--sync',
    action='store_true',
    help='Activate synchronous mode execution')
argparser.add_argument(
    '--red',
    default="0",
    help='Red part of the RGB color palette to paint the hero car')
argparser.add_argument(
    '--green',
    default="0",
    help='Green part of the RGB color palette to paint the hero car')
argparser.add_argument(
    '--blue',
    default="0",
    help='Blue part of the RGB color palette to paint the hero car')
argparser.add_argument(
    '-n', '--name',
    metavar='F',
    default="example",
    help='Participant name (name)')
argparser.add_argument(
    '-s', '--scenario',
    metavar='F',
    default="example",
    help='Scenario name (name)')
argparser.add_argument("s_b64")
argparser.add_argument("p_b64")
args = argparser.parse_args()

logger = logging.getLogger("driver-thread")
logging.basicConfig(level=logging.INFO, filename="../../data/recordings/{}/session_logs.log".format(args.name), filemode="a", format="%(asctime)s - %(name)s - %

def main():
    args.width, args.height = [int(x) for x in args.res.split('x')]

    print(__doc__)

    try:

        scenario_b64 = args.s_b64
        path_b64 = args.p_b64

        scenario_bytes = base64.b64decode(scenario_b64.encode())
        path_bytes = base64.b64decode(path_b64.encode())

        scenario = pickle.loads(scenario_bytes)
        path_details = pickle.loads(path_bytes)
        game_loop(args, scenario, path_details)

    except KeyboardInterrupt as e:
        logger.warning('Keyboard interupt while running driver.py script', e)
    except Exception as e:
        logger.error("Something went wrong.", e)


if __name__ == '__main__':

    main()
```

File: ./software/carla_scripts/simulation_manager.py

```
import json
import logging
import sys
from scenarios.scenario_factory import ScenarioFactory
from scenarios.path_builder import PathBuilder
import glob
import os
import time
import pickle
import subprocess
import base64
from threading import Thread

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla


logger = logging.getLogger(__name__)

class SimulationManager():

    def __init__(self, scenario_name, path_name, participant_name, vehicle_blueprint, randomization_seed, ai):
        self.vehicle_blueprint = vehicle_blueprint
        self.scenario_name = scenario_name
        self.path_name = path_name
        self.participant_name = participant_name
        self.ai = ai
        self.seed = randomization_seed
        self.scenario = None

    def begin_simulation(self):
        try:
            scenario_data = self.load_scenario()
            path_details = self.determine_path_details(scenario_data)
            self.change_map(path_details["town"])
            self.change_weather(scenario_data)
            self.scenario = ScenarioFactory.make_scenario(scenario_data, path_details, self.seed)
            self.launch_driving_mode(self.scenario, path_details)

        except Exception as e:
            raise e

    # Either generate the path or get it from file if such is defined
    def determine_path_details(self, scenario_data):
        if self.path_name:
            details = self.read_path_details()
        else:
            logger.info("No predefined details to load. The details will be generated automatically.")
            details = PathBuilder.generate_details(scenario_data)
        return details

```

```
    # Read path details to predefine the scenario if the path is specified
    def read_path_details(self):
        logger.info("Loading predefined details for the scenario")
        file_path = "../../data/paths/{}.json".format(self.path_name)
        try:
            with open(file_path) as file:
                details = json.load(file)
                logger.info("Successfully read additional scenario details from {}.json data".format(self.path_name))
                return details
        except FileNotFoundError as e:
            logger.error("Path {} file could not be found.".format(self.path_name), exc_info=True)
            raise e
        except json.JSONDecodeError as e:
            logger.error("The file {}.json could not be decoded as JSON".format(self.path_name), exc_info=True)
            raise e


    # Get a file indicating scenario parameters
    def load_scenario(self):
        logger.info("Loading {} data".format(self.scenario_name))
        file_path = "../../data/scenario_generation_data/generated_scenarios/{}.json".format(self.scenario_name)
        try:
            with open(file_path) as file:
                scenario_data = json.load(file)
                logger.info("Successfully read scenario {}.json data".format(self.scenario_name))
                return scenario_data
        except FileNotFoundError as e:
            logger.error("Scenario {} file could not be found.".format(self.scenario_name), exc_info=True)
            raise e
        except json.JSONDecodeError as e:
            logger.error("The file {}.json could not be decoded as JSON".format(self.scenario_name), exc_info=True)
            raise e

    # Used to kill all the actors in the simulation
    def close_scenario(self):
        client = carla.Client('127.0.0.1', 2000)
        client.set_timeout(20.0)
        if(self.scenario is not None and self.scenario.populated):
            logger.info("The scenario was running. Finishing it.")
            self.scenario.finish(client)

    # Change the weather in the simulation
    def change_weather(self, scenario_data):
        try:
            logger.info("Changing the weather")
            client = carla.Client('localhost', 2000)
            world = client.get_world()
            weather = carla.WeatherParameters(
                cloudiness = scenario_data["cloudiness"],
                precipitation = scenario_data["precipitation"],
                precipitation_deposits = scenario_data["precipitation_deposits"],
                wind_intensity = scenario_data["wind_intensity"],
                sun_azimuth_angle = scenario_data["sun_azimuth_angle"],
                sun_altitude_angle = scenario_data["sun_altitude_angle"],
                fog_density = scenario_data["fog_density"],
                fog_distance = scenario_data["fog_distance"],
                wetness = scenario_data["wetness"],
                fog_falloff = scenario_data["fog_falloff"],
                scattering_intensity = scenario_data["scattering_intensity"],
                mie_scattering_scale = scenario_data["mie_scattering_scale"],
                rayleigh_scattering_scale = scenario_data["rayleigh_scattering_scale"],
                # dust_storm = scenario_data["dust_storm"],
            )

            world.set_weather(weather)
            logger.info("Finished changing the weather")
        except Exception as e:
            logger.error("Something wrong happened trying to change the weather. Aborting...", exc_info=True)
            raise e

    # Change the simulation map
    def change_map(self, map):
        try:
            logger.info("Changing the map to {}.".format(map))
            client = carla.Client('localhost', 2000)
            client.set_timeout(10.0)
            world = client.load_world(map)
            # Give 2 sec for the map to load
            time.sleep(2)
            logger.info("Finished changing the map")
        except Exception as e:
            logger.exception("Something wrong happened trying to change the map. Aborting...", exc_info=True)
            raise e


    # Launch the first person driving mode
    def launch_driving_mode(self, scenario, path_details):
        scenario_bytes = pickle.dumps(scenario)
        path_bytes = pickle.dumps(path_details)

        s_b64 = base64.b64encode(scenario_bytes).decode()
        p_b64 = base64.b64encode(path_bytes).decode()
        try:
            if self.ai:
                logger.info("Launching self_driver.py script as a subprocess to let AI implementation drive the scenarios.")
                process = subprocess.run(["python3", "self_driver.py", s_b64, p_b64, "-n{}".format(self.participant_name), "--sync", "-s{}".format(self.scenario_
            else:
                logger.info("Launching driver.py script as a subprocess to drive the generated scenario.")
                process = subprocess.run(["python3", "driver_keyboard.py", s_b64, p_b64, "-n{}".format(self.participant_name), "--sync", "-s{}".format(self.scena
                # process = subprocess.run(["python3", "driver_steeringwheel.py", s_b64, p_b64, "-n{}".format(self.participant_name), "--sync", "-s{}".format(self
            # Uncomment to print statemennts coming from driver.py to the terminal
            # print(process.stdout.decode(), process.stderr.decode())
        except Exception as e:
            logger.error("Could not launch driving script properly", exc_info=True)
            raise e

    # # MULTI-THREADING

    # # Launch the first person driving mode
    # def launch_driving_mode(self, scenario, path_details):
    #     try:
    #         flag = [False]

    #         threads = [
    #             Thread(target=simulate, args=[scenario, path_details, self.participant_name, self.scenario_name, self.vehicle_blueprint, self.ai, flag]),
    #             Thread(target=draw_lanes, args=[path_details["path_checkpoints"], self.participant_name]),
    #             Thread(target=draw_hero, args=[self.participant_name])
    #         ]

    #         for thread in threads:
    #             thread.start()

    #         while True:
    #             time.sleep(1)
    #             if flag[0]:
    #                 for thread in threads:
    #                     thread.join()
    #                 break
    #     except Exception as e:
    #         logger.error("Something went wrong when starting the threads.", e)
```

```

def simulate(scenario, path_details, participant_name, scenario_name, vehicle_blueprint, ai, flag):
    try:
        scenario_bytes = pickle.dumps(scenario)
        path_bytes = pickle.dumps(path_details)

        s_b64 = base64.b64encode(scenario_bytes).decode()
        p_b64 = base64.b64encode(path_bytes).decode()
        if ai:
            logger.info("Launching self_driver.py script as a subprocess to let AI implementation drive the scenarios.")
            simulating = subprocess.Popen(["python3", "self_driver.py", s_b64, p_b64, "-n{}".format(participant_name), "--sync", "-s{}".format(scenario_name), "-
        else:
            logger.info("Launching driver.py script as a subprocess to drive the generated scenario.")
            simulating = subprocess.Popen(["python3", "driver.py", s_b64, p_b64, "-n{}".format(participant_name), "--sync", "-s{}".format(scenario_name), "-f{}".

    except Exception as e:
        logger.error("Could not the simulation", e)

def draw_lanes(locations, participant_name):
    try:
        locations_bytes = pickle.dumps(locations)
        locations_b64 = base64.b64encode(locations_bytes).decode()

        log_path = "../../data/recordings/{}/session_logs.log".format(participant_name)

        logger.info("Launching draw_lines.py script as a subprocess to mark the lanes and the driver on the map.")
        drawing = subprocess.Popen(["python3", "helper_scripts/draw_lanes.py", locations_b64, "-l{}".format(log_path)])
    except Exception as e:
        logger.error("Could not start drawing lines", e)

def draw_hero(participant_name):
    try:
        log_path = "../../data/recordings/{}/session_logs.log".format(participant_name)

        logger.info("Launching draw_hero.py script as a subprocess to mark the location of the participant on the map.")
        drawing = subprocess.Popen(["python3", "helper_scripts/draw_hero.py", "-l{}".format(log_path)])
    except Exception as e:
        logger.error("Could not start hero vehicle", e)
```

File: ./software/carla_scripts/scenarios/scenario_tester.py

```
import glob
import os
import sys
import argparse
import argparse

try:
    sys.path.append('..')
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla
from scenarios import ScenarioOne

argparser = argparse.ArgumentParser(description=__doc__)
argparser.add_argument(
    '-n', '--name',
    default='scenario1',
    help='Specify the name of the scenario')
args = argparser.parse_args()

# When a new scenario needs to be simulated
# Its subclass has to be imported and used in this function to retrieve it in the main
def get_scenario(traffic_manager, sim_world):
    if(args.name == "scenario1"):
        return ScenarioOne(traffic_manager, sim_world)
    else:
        return ScenarioOne(traffic_manager, sim_world)

# Used to run and test if the scenario is working as expected
# Spawns all the actors, runs them and then destroys
def main():
    try:
        print("Beginning to simulate {}".format(args.name))
        client = carla.Client('localhost', 2000)
        client.set_timeout(20.0)
        sim_world = client.get_world()
        original_settings = sim_world.get_settings()
        settings = sim_world.get_settings()
        if not settings.synchronous_mode:
            settings.synchronous_mode = True
            settings.fixed_delta_seconds = 0.05
        sim_world.apply_settings(settings)

        traffic_manager = client.get_trafficmanager()
        traffic_manager.set_synchronous_mode(True)
        scenario = get_scenario(traffic_manager, sim_world)
        scenario.spawn(client)
        scenario.start()
        while True:
            sim_world.tick()
    except KeyboardInterrupt:
        print("Finished simulating the scenario.")
    except:
        print("Something wrong happened trying to simulate the scenario.")
    finally:
        scenario.finish(client)
        sim_world.apply_settings(original_settings)

if __name__ == '__main__':
    main()
```

File: ./software/carla_scripts/scenarios/scenario_factory.py

```
from scenarios.scenario import Scenario
import sys
import os
import glob
import logging

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla

logger = logging.getLogger(__name__)


class ScenarioFactory():

    @staticmethod
    def make_scenario(parameters, path_details, seed):
        logger.info("Creating a scenario according to the generated path and scenario details")
        return Scenario(parameters, path_details, seed)

```

File: ./software/carla_scripts/scenarios/change_weather.py

```
import glob
import os
import sys
import time
import random
import argparse

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla

import logging
logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO, filename="log.log", filemode="a", format="%(asctime)s - %(name)s - %(levelname)s - %(message)s")

argparser = argparse.ArgumentParser(description=__doc__)
argparser.add_argument(
    '-n', '--name',
    default='scenario1',
    help='Specify the name of the scenario')
argparser.add_argument(
    '--log',
    action='store_true',
    help='Activate logging')
args = argparser.parse_args()

# Logging is used only when maps are changed during the simulations (logged to the participant's directory)
if args.log:
    logging.basicConfig(level=logging.INFO, filename='/recording/recordings/{}/session_logs.log'.format(args.name), filemode="a", format="%(asctime)s - %(levelnam
    logger = logging.getLogger("ChangeWeather")

# Log INFO message
def inform(message):
    logger.info(message) if args.log else print(message)

def main():
    try:
        inform("Changing the weather for {}.".format(args.name))
        if args.name == "scenario1":
            pass
        else:
            pass
        inform("Finished changing the weather")
    except KeyboardInterrupt:
        message = "Manually exited the weather changing script."
        logger.warn(message) if args.log else print(message)
    except:
        message = "Something wrong happened trying to change the weather. Aborting..."
        logger.exception(message) if args.log else print(message)


if __name__ == '__main__':
    main()
```

File: ./software/carla_scripts/scenarios/change_map.py

```
import glob
import os
import sys
import time
import random
import argparse

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla

argparser = argparse.ArgumentParser(description=__doc__)
argparser.add_argument(
    '-m', '--map',
    default='Town04',
    help='Specify the name of the map')
argparser.add_argument(
    '--name',
    default="ParticipantA",
    help='Name of the participant')
args = argparser.parse_args()


def main():
    try:
        print("Changing the map to {}.".format(args.map))
        client = carla.Client('localhost', 2000)
        world = client.load_world('{}'.format(args.map))
        time.sleep(10)
        print("Finished changing the map")
    except KeyboardInterrupt:
        message = "Manually exited the weather changing script."
        print(message)
    except Exception as e:
        message = "Something wrong happened trying to change the map. Aborting..."
        print(message)

if __name__ == '__main__':
    main()
```

File: ./software/carla_scripts/scenarios/scenario.py

```
import carla
import random
import numpy as np
import logging

logger= logging.getLogger(__name__)


class Scenario():

    def __init__(self, parameters, path_details, seed):
        self.scenario_parameters = parameters
        self.path_details = path_details
        # Used for reproducable randomizarion
        self.seed = seed

        self.vehicles_list = []
        self.walkers_list = []
        self.two_wheel_vehicles_list = []
        self.all_id = []
        self.all_actors = []

        self.traffic_manager = None
        self.synchronous_master = False
        self.world = None
        self.populated = False

        self.walker_speed = None

    def spawn(self, client):
        logger.info("Spawning the scenario actors")
        try:
            blueprints_vehicles_all = get_actor_blueprints(self.world, "vehicle.*", "All")

            blueprints_vehicles = get_safe_vehicle_blueprints(blueprints_vehicles_all)
            blueprints_walkers = get_actor_blueprints(self.world, "walker.pedestrian.*", "2")
            blueprints_two_wheel_vehicles = get_two_wheel_vehicle_blueprints(blueprints_vehicles_all)
        except Exception as e:
            logger.error("An error occured trying to get the blueprints", exc_info=True)
            raise e


        try:
            # Read all the parameters from the scenario object
            parameters = self.scenario_parameters
            number_of_pedestrians = int(parameters["number_of_pedestrians"])
            number_of_vehicles = int(parameters["number_of_vehicles"])
            number_of_two_wheel_vehicles = int(parameters["number_of_two_wheel_vehicles"])
            proportion_of_speeding_vehicles = float(parameters["proportion_of_speeding_vehicles"])
            proportion_of_vehicles_without_lights = float(parameters["proportion_of_vehicles_without_lights"])
            proportion_of_light_ignoring_vehicles = float(parameters["proportion_of_light_ignoring_vehicles"])
            light_ignoring_percent = float(parameters["light_ignoring_percent"])
            proportion_of_sign_ignoring_vehicles = float(parameters["proportion_of_sign_ignoring_vehicles"])
            sign_ignoring_percent = float(parameters["sign_ignoring_percent"])
            proportion_of_vehicle_ignoring_vehicles = float(parameters["proportion_of_vehicle_ignoring_vehicles"])
            vehicle_ignoring_percent = float(parameters["vehicle_ignoring_percent"])
            proportion_of_walker_ignoring_vehicles = float(parameters["proportion_of_walker_ignoring_vehicles"])
            walker_ignoring_percent = float(parameters["walker_ignoring_percent"])
            proportion_of_keeping_right_vehicles = float(parameters["proportion_of_keeping_right_vehicles"])
            keeping_right_percent = float(parameters["keeping_right_percent"])
            proportion_of_lane_changing_vehicles = float(parameters["proportion_of_lane_changing_vehicles"])
            lane_change_percent = float(parameters["lane_change_percent"])
            proportion_of_misbehaving_pedestrians = float(parameters["proportion_of_misbehaving_pedestrians"])
            proportion_of_running_pedestrians = float(parameters["proportion_of_running_pedestrians"])
            proportion_of_road_crossing_pedestrians = float(parameters["proportion_of_road_crossing_pedestrians"])
        except Exception as e:
            logger.error("An error occured trying to get retrieve scenario parameters", exc_info=True)
            raise e


        # ALL NECCESSARY PARAMETERS WERE EXTRACTED

        try:
            SpawnActor = carla.command.SpawnActor
            SetAutopilot = carla.command.SetAutopilot
            FutureActor = carla.command.FutureActor

```

```
            start = self.path_details["start_location"]
            hero_spawn_location = carla.Location(x=float(start["x"]), y=float(start["y"]), z=float(start["z"]))

            # Removes ones too close to the player's spawn location
            spawn_points = self.world.get_map().get_spawn_points()
            spawn_points = filter_spawn_points(spawn_points, hero_spawn_location, 5)
            number_of_spawn_points = len(spawn_points)

            total_number_of_vehicles = (number_of_vehicles + number_of_two_wheel_vehicles)
            if total_number_of_vehicles <= number_of_spawn_points:
                random.shuffle(spawn_points)

            else:
                logger.warning("There are too many vehicles for too few spawn points. Reducing the number of vehicles. There are {} spawn points for vehicles in t
                difference = total_number_of_vehicles - number_of_spawn_points
                if (number_of_vehicles - difference) >= 0:
                    number_of_vehicles = (number_of_vehicles - difference)
                else:
                    # Over is going to be negative
                    over = (number_of_vehicles - difference)
                    number_of_two_wheel_vehicles = number_of_two_wheel_vehicles + over

            total_number_of_vehicles = (number_of_vehicles + number_of_two_wheel_vehicles)
            vehicle_spawn_points, two_wheel_vehicle_spawn_points = divide_list(spawn_points, number_of_vehicles, number_of_two_wheel_vehicles)

            # Make all vehicles aim for the speed limit
            self.traffic_manager.global_percentage_speed_difference(0)
        except Exception as e:
            logger.error("An error occured trying to get the spawn points", exc_info=True)
            raise e

        try:
            # --------------
            # Spawn vehicles
            # --------------
            batch = []
            for n, transform in enumerate(vehicle_spawn_points):
                if n >= number_of_vehicles:
                    break
                blueprint = random.choice(blueprints_vehicles)
                if blueprint.has_attribute('color'):
                    color = random.choice(blueprint.get_attribute('color').recommended_values)
                    blueprint.set_attribute('color', color)
                if blueprint.has_attribute('driver_id'):
                    driver_id = random.choice(blueprint.get_attribute('driver_id').recommended_values)
                    blueprint.set_attribute('driver_id', driver_id)
                blueprint.set_attribute('role_name', 'autopilot')
                # Spawn the cars and set their autopilot and light state all together
                batch.append(SpawnActor(blueprint, transform)
                    .then(SetAutopilot(FutureActor, False, self.traffic_manager.get_port())))

            for response in client.apply_batch_sync(batch, self.synchronous_master):
                if response.error:
                    logger.error(response.error)
                else:
                    self.vehicles_list.append(response.actor_id)

            logger.info("The {}/{} vehicles were successfully spawned".format(len(self.vehicles_list), number_of_vehicles))

        except Exception as e:
            logger.error("Error creating vehicles on the map.", exc_info=True)
            raise e

        try:
            # --------------
            # Spawn two-wheel vehicles
            # --------------
            batch = []
            for n, transform in enumerate(two_wheel_vehicle_spawn_points):
                if n >= number_of_two_wheel_vehicles:
                    break
                blueprint = random.choice(blueprints_two_wheel_vehicles)
                if blueprint.has_attribute('color'):
                    color = random.choice(blueprint.get_attribute('color').recommended_values)
                    blueprint.set_attribute('color', color)
                if blueprint.has_attribute('driver_id'):
                    driver_id = random.choice(blueprint.get_attribute('driver_id').recommended_values)
                    blueprint.set_attribute('driver_id', driver_id)
                blueprint.set_attribute('role_name', 'autopilot')
                # Spawn the cars and set their autopilot and light state all together
                batch.append(SpawnActor(blueprint, transform)
                    .then(SetAutopilot(FutureActor, False, self.traffic_manager.get_port())))

            for response in client.apply_batch_sync(batch, self.synchronous_master):
                if response.error:
                    logger.error(response.error)
                else:
                    self.two_wheel_vehicles_list.append(response.actor_id)

            logger.info("The {}/{} two-wheel vehicles were successfully spawned".format(len(self.two_wheel_vehicles_list), number_of_two_wheel_vehicles))


        except Exception as e:
            logger.error("Error creating two-wheel vehicles on the map.", exc_info=True)
            raise e

        try:
            logger.info("Setting the vehicle behavior according to the scenario parameters")
            ### SETUP VEHICLE BEHAVIOUR

            all_vehicles = self.vehicles_list + self.two_wheel_vehicles_list
            amount_vehicles = len(all_vehicles)
            all_vehicle_actors = self.world.get_actors(all_vehicles)


            # Distance
            for actor in all_vehicle_actors:
                # Select a distance to keep between 1 and 5 metres randomly.
                diff = random.randint(1, 5)
                self.traffic_manager.auto_lane_change(actor, True)
                self.traffic_manager.distance_to_leading_vehicle(actor, diff)

            # Set not update vehicle lights for certain vehicles
            amount_of = int(np.round(amount_vehicles * proportion_of_vehicles_without_lights))
            random_vehicles = random.sample(all_vehicles, amount_of)
            for v in random_vehicles:
                actor = self.world.get_actor(v)
                self.traffic_manager.update_vehicle_lights(actor, False)

            # Update the lights for the rest
            for v in all_vehicles:
                if(v not in random_vehicles):
                    actor = self.world.get_actor(v)
                    self.traffic_manager.update_vehicle_lights(actor, True)


            # Speeding
            amount_of = int(np.round(amount_vehicles * proportion_of_speeding_vehicles))
            random_vehicles = random.sample(all_vehicles, amount_of)
            for v in random_vehicles:
```

```
                actor = self.world.get_actor(v)
                # Select a speeding increase randomly
                diff = random.randint(-30, -1)
                self.traffic_manager.vehicle_percentage_speed_difference(actor, diff)

            # Light ignoring
            amount_of = int(np.round(amount_vehicles * proportion_of_light_ignoring_vehicles))
            random_vehicles = random.sample(all_vehicles, amount_of)
            for v in random_vehicles:
                actor = self.world.get_actor(v)
                self.traffic_manager.ignore_lights_percentage(actor, light_ignoring_percent)

            # Sign ignoring
            amount_of = int(np.round(amount_vehicles * proportion_of_sign_ignoring_vehicles))
            random_vehicles = random.sample(all_vehicles, amount_of)
            for v in random_vehicles:
                actor = self.world.get_actor(v)
                self.traffic_manager.ignore_signs_percentage(actor, sign_ignoring_percent)

            # Vehicle ignoring
            amount_of = int(np.round(amount_vehicles * proportion_of_vehicle_ignoring_vehicles))
            random_vehicles = random.sample(all_vehicles, amount_of)
            for v in random_vehicles:
                actor = self.world.get_actor(v)
                self.traffic_manager.ignore_vehicles_percentage(actor, vehicle_ignoring_percent)

            # Walker ignoring
            amount_of = int(np.round(amount_vehicles * proportion_of_walker_ignoring_vehicles))
            random_vehicles = random.sample(all_vehicles, amount_of)
            for v in random_vehicles:
                actor = self.world.get_actor(v)
                self.traffic_manager.ignore_walkers_percentage(actor, walker_ignoring_percent)

            # Keep right rule
            amount_of = int(np.round(amount_vehicles * proportion_of_keeping_right_vehicles))
            random_vehicles = random.sample(all_vehicles, amount_of)
            for v in random_vehicles:
                actor = self.world.get_actor(v)
                self.traffic_manager.keep_right_rule_percentage(actor, keeping_right_percent)

            # Lane changing left
            amount_of = int(np.round(amount_vehicles * proportion_of_lane_changing_vehicles))
            random_vehicles = random.sample(all_vehicles, amount_of)
            for v in random_vehicles:
                actor = self.world.get_actor(v)
                self.traffic_manager.random_left_lanechange_percentage(actor, lane_change_percent)

            # Lane changing right
            amount_of = int(np.round(amount_vehicles * proportion_of_lane_changing_vehicles))
            random_vehicles = random.sample(all_vehicles, amount_of)
            for v in random_vehicles:
                actor = self.world.get_actor(v)
                self.traffic_manager.random_right_lanechange_percentage(actor, lane_change_percent)

            logger.info("The behaviour of vehicles was successfully set")

        except Exception as e:
            logger.error("Error setting the behaviour of vehicles.", exc_info=True)
            raise e


        try:
            # -------------
            # Spawn Walkers
            # -------------
            # Random locations to spawn

            number_of_pedestrians_initial = number_of_pedestrians

            try:
                spawn_points = []
                for i in range(number_of_pedestrians):
                    spawn_point = carla.Transform()
                    loc = self.world.get_random_location_from_navigation()
                    if (loc != None):
                        spawn_point.location = loc
                        spawn_points.append(spawn_point)

                if(len(spawn_points) >= number_of_pedestrians):
                    logger.debug("There is a required amount of spawn points for pedestrians on the map.")
                else:
                    number_of_pedestrians = len(spawn_points)
            except Exception as e:
                logger.error(e)

            # Spawn walker objects
            walker_speed = []
            walker_speed2 = []
            logger.debug("Need to spawn {} pedestrians".format(number_of_pedestrians))
            for i in range(number_of_pedestrians):

                # Modify the blueprint
                walker_bp = random.choice(blueprints_walkers)
                # set as not invincible
                if walker_bp.has_attribute('is_invincible'):
                    walker_bp.set_attribute('is_invincible', 'false')
                # set the max speed
                if walker_bp.has_attribute('speed'):
                    if (random.random() > proportion_of_running_pedestrians):
                        # walking
                        walker_speed.append(walker_bp.get_attribute('speed').recommended_values[1])
                    else:
                        # running
                        walker_speed.append(walker_bp.get_attribute('speed').recommended_values[2])
                else:
                    walker_speed.append(0.0)

                logger.debug("The blueprint was set.")
                try:
                    # Try to spawn the pedestrian 10000 times and
                    y = 0
                    not_spawned = True
                    while(not_spawned and y < 10000):
                        spawn_point = carla.Transform()
                        loc = self.world.get_random_location_from_navigation()
                        if (loc != None):
                            spawn_point.location = loc
                            batch = []
                            batch.append(SpawnActor(walker_bp, spawn_point))
                            results = client.apply_batch_sync(batch, True)
                            if results[0].error:
                                logger.debug(results[0].error)
                                y += 1
                            else:
                                not_spawned = False
                                self.walkers_list.append({"id": results[0].actor_id})
                                walker_speed2.append(walker_speed[i])
                                break
                        else:
                            y += 1
```

```
                except Exception as e:
                    logger.error(e)
                logger.debug("There were {} respawns for this walker".format(y))


            logger.info("Successfully spawned {}/{} walkers.".format(len(self.walkers_list), number_of_pedestrians_initial))
            self.walker_speed = walker_speed2

            # Spawn the walker controller
            batch = []
            walker_controller_bp = self.world.get_blueprint_library().find('controller.ai.walker')
            for i in range(len(self.walkers_list)):
                batch.append(SpawnActor(walker_controller_bp, carla.Transform(), self.walkers_list[i]["id"]))
            results = client.apply_batch_sync(batch, True)
            for i in range(len(results)):
                if results[i].error:
                    logger.error(results[i].error)
                else:
                    self.walkers_list[i]["con"] = results[i].actor_id

            # Put together the walkers and controllers id to get the objects from their id
            for i in range(len(self.walkers_list)):
                self.all_id.append(self.walkers_list[i]["con"])
                self.all_id.append(self.walkers_list[i]["id"])
            self.all_actors = self.world.get_actors(self.all_id)

        except Exception as e:
            logger.error("Error creating walkers on the map.", exc_info=True)
            raise e

        self.world.tick()
        self.world.set_pedestrians_cross_factor(proportion_of_road_crossing_pedestrians)
        self.populated = True

    # Setup the traffic manager and other parameters before the simulation
    def setup(self, traffic_manager, world):
        logger.info("Setting up the scenario")
        self.traffic_manager = traffic_manager
        self.world = world
        self.traffic_manager.set_global_distance_to_leading_vehicle(2.5)
        self.traffic_manager.set_random_device_seed(self.seed)
        self.world.set_pedestrians_seed(self.seed)
        random.seed(self.seed)
        self.traffic_manager.set_hybrid_physics_mode(True)
        self.traffic_manager.set_hybrid_physics_radius(70.0)

        settings = self.world.get_settings()
        self.traffic_manager.set_synchronous_mode(True)
        if not settings.synchronous_mode:
            self.synchronous_master = True
            settings.synchronous_mode = True
            settings.fixed_delta_seconds = 0.05
        else:
            self.synchronous_master = False

    # Unfreeze all actors and let them move as specified
    def start(self):
        logger.info("Unfreezing all actors in the simulation.")
        port = self.traffic_manager.get_port()

        # Unfreeze vehicles
        for v in self.vehicles_list:
            actor = self.world.get_actor(v)
            actor.set_autopilot(True, port)

        # Unfreeze two-wheel vehicles
        for b in self.two_wheel_vehicles_list:
            actor = self.world.get_actor(b)
            actor.set_autopilot(True, port)

        # Unfreeze walkers
        for i in range(0, len(self.all_id), 2):
            self.all_actors[i].start()
            self.all_actors[i].go_to_location(self.world.get_random_location_from_navigation())
            self.all_actors[i].set_max_speed(float(self.walker_speed[int(i/2)]))

    # Change settings back to default and destroy all actors
    def finish(self, client):
        logger.info("Finishing the scenario")
        if self.synchronous_master:
            settings = self.world.get_settings()
            settings.synchronous_mode = False
            settings.no_rendering_mode = False
            settings.fixed_delta_seconds = None
            self.world.apply_settings(settings)

        logger.info('Destroying {} vehicles'.format(len(self.vehicles_list)))
        client.apply_batch([carla.command.DestroyActor(x) for x in self.vehicles_list])

        logger.info('Destroying {} two-wheel vehicles'.format(len(self.two_wheel_vehicles_list)))
        client.apply_batch([carla.command.DestroyActor(x) for x in self.two_wheel_vehicles_list])
        # Stop walker controllers (list is [controller, actor, controller, actor ...])
        for i in range(0, len(self.all_id), 2):
            self.all_actors[i].stop()
        logger.info('Destroying {} walkers'.format(len(self.walkers_list)))
        client.apply_batch([carla.command.DestroyActor(x) for x in self.all_id])

        self.populated = False


def get_actor_blueprints(world, filter, generation):
    bps = world.get_blueprint_library().filter(filter)
    if generation.lower() == "all":
        return bps

    # If the filter returns only one bp, we assume that this one needed
    # and therefore, we ignore the generation
    if len(bps) == 1:
        return bps
    try:
        int_generation = int(generation)
        # Check if generation is in available generations
        if int_generation in [1, 2]:
            bps = [x for x in bps if int(x.get_attribute('generation')) == int_generation]
            return bps
        else:
            logger.warning("Actor Generation is not valid. No actor will be spawned.")
            return []
    except:
        logger.warning("Actor Generation is not valid. No actor will be spawned.")
        return []

def get_safe_vehicle_blueprints(blueprints):
    blueprints = [x for x in blueprints if int(x.get_attribute('number_of_wheels')) == 4]
    blueprints = [x for x in blueprints if not x.id.endswith('microlino')]
    blueprints = [x for x in blueprints if not x.id.endswith('carlacola')]
    blueprints = [x for x in blueprints if not x.id.endswith('cybertruck')]
    blueprints = [x for x in blueprints if not x.id.endswith('t2')]
    blueprints = [x for x in blueprints if not x.id.endswith('sprinter')]
    blueprints = [x for x in blueprints if not x.id.endswith('firetruck')]
```

```
    blueprints = [x for x in blueprints if not x.id.endswith('ambulance')]
    blueprints = sorted(blueprints, key=lambda bp: bp.id)
    return blueprints

def get_two_wheel_vehicle_blueprints(blueprints):
    return [x for x in blueprints if int(x.get_attribute('number_of_wheels')) == 2]

def distanceBetweenTwoLocations(start, finish):
    first = np.array((start.x, start.y))
    second = np.array((finish.x, finish.y))
    return np.linalg.norm(first-second)

def filter_spawn_points(spawn_points, hero_spawn_location, min_distance):
    try:
        filtered_spawn_points = []
        for point in spawn_points:
            distance = distanceBetweenTwoLocations(point.location, hero_spawn_location)
            # If the spawn location is further than two metres from the player's location, add to the filtered ones
            if distance > min_distance:
                filtered_spawn_points.append(point)
        return filtered_spawn_points
    except Exception as e:
        logger.error(e)

def divide_list(input_list, x, y):
    first_list = input_list[:x]
    second_list = input_list[x:x+y]
    return first_list, second_list
```

File: ./software/carla_scripts/scenarios/path_builder.py

```
import logging
import xml.etree.ElementTree as ET
import os
import random
import json

logger = logging.getLogger(__name__)
# How many times to try and recreate the path if it keeps on failing or reaching a dead-end before
MAX_SEARCH_LENGTH = 10000
path_to_save_path = "../../data/paths/"

class PathBuilder():

    @staticmethod
    # Used to find the path, the map and the start location automatically
    def generate_details(scenario_data, save_filename = None):
        try:
            map, road_elements = PathBuilder.find_map_and_path(scenario_data)
        except Exception as e:
            logger.exception("Something went wrong generating the path", e)
            raise Exception
        try:
            start_location = PathBuilder.find_start_location(road_elements[0])
            path_details = PathBuilder.form_json(map, road_elements, start_location, save_filename)
            logger.info(path_details)
            if save_filename:
                logger.info("Path was saved to file {}.json in paths directory amongst other data".format(save_filename))
            return path_details
        except Exception as e:
            logger.exception("Something went wrong combining the path object", e)
            raise Exception

    @staticmethod
    # Used to return the newly generated path and everything in a json-like python structure
    # And save it, if the save filename is specified
    def form_json(map, road_elements, start_location, save_filename):
        # Build a json-like file
        # Save if needed and return
        locations = []
        for i in range(len(road_elements)):
            if(i == len(road_elements)-1):
                locations.append(PathBuilder.get_road_location_object(road_elements[i], finish_road=True))
            else:
                locations.append(PathBuilder.get_road_location_object(road_elements[i]))
        attributes = {
            "path_checkpoints": locations,
            "start_location": start_location,
            "town": map.replace(".xodr", "")
        }
        if save_filename:
            with open("{}{}.json".format(path_to_save_path, save_filename), "w+") as f:
                json.dump(attributes, f, indent=4)
        return attributes

    @staticmethod
    # Used to find and return a start location among the given path_checkpoints
    def find_start_location(road_element):
        return PathBuilder.get_road_location_object(road_element, yaw=True)


    @staticmethod
    # Extract the location from the road object
    def get_road_location_object(road, finish_road = None, yaw = None):
        geometries = road.find("planView").findall("geometry")
        geometry = geometries[len(geometries)-1] if finish_road else geometries[0]
        x = float(geometry.attrib["x"])
        y = float(geometry.attrib["y"])
        rotation = float(geometry.attrib["hdg"])
        # Determine z according to the height in that place
        # yaw determine from the road_element
        z = 0.4
        start_location = {
            "x": x,
            "y": y,
            "z": z,
            "yaw": rotation
        }
        return start_location

    @staticmethod
    # Used to find the appropriate city
    def find_map_and_path(parameters):
        # Look for an appropriate map iterating through all files in maps/opendrive_format drirectory
        logger.info("Looking for a map that could be used in the scenario and that satisfies the scenario parameters.")
        map_filenames = os.listdir("../../data/maps/opendrive_format")
        already_considered_maps = []
        for i in range(len(map_filenames)):
            # Loop through all mapfiles randomly and if the match is found, return
            unconsidered_maps = [file for file in map_filenames if file not in already_considered_maps and file.endswith(".xodr")]
            random_map = random.sample(unconsidered_maps, 1)[0]
            processed_map = PathBuilder.process_map(random_map, parameters)
            if processed_map:
                return random_map, processed_map
            else:
                logger.warning("The algorithm was unable to find a path given the scenario parameters in the map {}".format(random_map))
```

```
                already_considered_maps.append(random_map)
                if i == (len(map_filenames)-1):
                    logger.warning("The search was exhausted and no valid path was found for the given scenario and all the map files. RETURNING NONE.")
        # If this place was reached, it is impossible to create a map
        return None

    @staticmethod
    # Process the map file to check if it is appropriate for this scenario
    def process_map(filename, parameters):
        logger.info("Checking if {} is a valid map for this scenario".format(filename))
        path_to_file = "../../data/maps/opendrive_format/{}".format(filename)
        tree = ET.parse(path_to_file)
        root = tree.getroot()

        # Get all road and junction elements from the XML
        roads = []
        junctions = []
        for road in root.findall("./road"):
            roads.append(road)
        for junction in root.findall("./junction"):
            junctions.append(junction)

        verified = PathBuilder.check_if_parameters_are_satisfied(parameters, roads, junctions, filename)
        if not verified: return None

        ## MAKE PATH
        #print("Working on map: {}".format(filename))
        logger.info("Working on map: {}".format(filename))
        return PathBuilder.make_path(roads, junctions, parameters)


    @staticmethod
    # Checks if the scenario parameters are satisfied in the map
    def check_if_parameters_are_satisfied(parameters, roads, junctions, filename):
        # Check if there are enough junctions in the map
        if len(junctions) >= parameters["number_of_junctions"]:
            logger.info("Junction parameter is satisfied")
        else:
            logger.warning("There are not enough junctions in {} to accomodate the scenario.".format(filename))
            return None

        # Check if there is distance requirement is satisfied
        total_road_length = 0.0
        for road in roads:
            total_road_length += float(road.attrib["length"])
        if total_road_length >= float(parameters["distance_in_metres"]):
            logger.info("Distance parameter is satisfied")
        else:
            logger.warning("There is not enough unique drivable road to accomodate the scenario {}".format(filename))
            return None
        return True

    @staticmethod
    # Used to run iteration and try to create a path from the given road and junction objects
    def make_path(roads, junctions, parameters):
        go = True
        i = 0
        path = None
        # Try to create a path for a MAX_SEARCH_LENGTH times
        while(go and i < MAX_SEARCH_LENGTH):
            temp = None
            try:
                temp = PathBuilder.get_path(roads, junctions, parameters["number_of_junctions"], parameters["distance_in_metres"])
            except Exception as e:
                logger.warning("Something went weong trying to create the path. Retrying.")
            if temp:
                logger.info("The path was successfully generated")
                go = False
                path = temp
            i += 1
        if not path:
            logger.warning("No success in {} tries to create a path".format(MAX_SEARCH_LENGTH))
            # print("No success in {} tries...".format(i))
        return path

    @staticmethod
    # Used to generate a random path over the map
    def get_path(roads, junctions, max_number_of_junctions_allowed, min_distance):
        path = []
        chosen_road_queue = []
        current_road = random.choice(roads)
        path.append(current_road)
        id, length, predecessor, successor = PathBuilder.analyse_road_element(current_road)
        chosen_road_queue.append(id)
        # print("Id of the first one: {}".format(id))
        current_junctions = 0
        current_length = length
        while current_length < min_distance and current_junctions < max_number_of_junctions_allowed:
            if len(chosen_road_queue) > 7 and chosen_road_queue[-8:].count(id) >= 3:
                # Check if the same road id appeared among the last 8 roads chosen
                # It means that the algorithm got stuck and now is moving back and forth
                return None
            # print("Current length: {}".format(current_length))
            # print("Already used roads: {}".format(chosen_road_queue))
            if successor[0] == "junction":
                # print("Junction approached. Its id is: {}".format(successor[1]))
                current_junctions += 1
                junction = PathBuilder.get_the_right_junction(junctions, successor[1])
                all_exits = PathBuilder.analyse_junction_and_get_possible_exits(junction, id, chosen_road_queue)
                # If a dead-end was reached (no more exits)
                if len(all_exits) == 0:
                    # Return a failure, because a dead-end was reached
                    return None
                # print("All possible exits from this are: {}".format(all_exits))
                random_exit = random.choice(all_exits)
                new_road_to_take = PathBuilder.get_the_right_road(roads, random_exit)
                id, length, predecessor, successor = PathBuilder.analyse_road_element(new_road_to_take)
                # print("The chosen way is: {}".format(id))
                current_length += length
                chosen_road_queue.append(id)
            else:
                new_road_to_take = PathBuilder.get_the_right_road(roads, successor[1])
                id, length, predecessor, successor = PathBuilder.analyse_road_element(new_road_to_take)
                # print("The chosen way is: {}".format(id))
                current_length += length
                chosen_road_queue.append(id)
                path.append(new_road_to_take)
            # print("\n")

        return path


    @staticmethod
    # Used to parse the xml road element and retrieve useful data
    def analyse_road_element(road):
        id = road.attrib["id"]
        length = float(road.attrib["length"])
        link = road.findall("link")[0]
        predecessor = (link.find("predecessor").attrib["elementType"], link.find("predecessor").attrib["elementId"])
        successor = (link.find("successor").attrib["elementType"], link.find("successor").attrib["elementId"])
```

```
        return id, length, predecessor, successor

    @staticmethod
    # Used to
    def analyse_junction_and_get_possible_exits(junction, incoming_road_id, already_used_ids):
        # First try to get new roads to drive on
        possible_next_roads = []
        id = junction.attrib["id"]
        for connection in junction.findall("connection"):
            if connection.attrib["incomingRoad"] == incoming_road_id and connection.attrib["connectingRoad"] not in already_used_ids:
                possible_next_roads.append(connection.attrib["connectingRoad"])

        # If there are no unique roads, allow picking of old ones
        if len(possible_next_roads) == 0:
            for connection in junction.findall("connection"):
                if connection.attrib["incomingRoad"] == incoming_road_id:
                    possible_next_roads.append(connection.attrib["connectingRoad"])
        # print("Possible next moves: ".format(possible_next_roads))
        return possible_next_roads

    @staticmethod
    # Used to find the right junction object from all junctions using its id
    def get_the_right_junction(junctions, id):
        for junction in junctions:
            if junction.attrib["id"] == id:
                return junction
        return None

    @staticmethod
    # Used to find the right road object from all road objects using its id
    def get_the_right_road(roads, id):
        for road in roads:
            if road.attrib["id"] == id:
                return road
        return None
```

File: ./software/carla_scripts/clean.sh

```
#!/bin/bash
find . -name __pycache__ -type d -exec rm -rf {} +

find . -name ".*" -type f -delete

find . -name "*.sh" -exec chmod +x {} \;
```

File: ./software/carla_scripts/wheel_config.ini

```
[G29 Racing Wheel]
steering_wheel = 0
throttle = 2
brake = 3
reverse = 6
handbrake = 4
```

File: ./software/carla_scripts/driver_keyboard.py

```
#!/usr/bin/env python

# Copyright (c) 2019 Computer Vision Center (CVC) at the Universitat Autonoma de
# Barcelona (UAB).
#
# This work is licensed under the terms of the MIT license.
# For a copy, see <https://opensource.org/licenses/MIT>.

# Allows controlling a vehicle with a keyboard. For a simpler and more
# documented example, please take a look at tutorial.py.
"""
Welcome to CARLA manual control.

Use ARROWS or WASD keys for control.

    W            : throttle
    S            : brake
    A/D          : steer left/right
    Q            : toggle reverse
    Space        : hand-brake
    P            : toggle autopilot
    M            : toggle manual transmission
    ,/.          : gear up/down

    L            : toggle next light type
    SHIFT + L    : toggle high beam
    Z/X          : toggle right/left blinker
    I            : toggle interior light

    TAB          : change camera position

    ENTER        : start route

    F1           : toggle HUD
    H/?          : toggle help
    ESC          : quit
"""

from __future__ import print_function



# ==============================================================================
# -- find carla module ---------------------------------------------------------
# ==============================================================================


import glob
import os
import sys
import pickle
from scenarios.scenario import Scenario
import base64

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass
```

```
63
64
65  # ==============================================================================
66  # -- imports -------------------------------------------------------------------
67  # ==============================================================================
68
69
70  import carla
71
72  from carla import ColorConverter as cc
73
74  import argparse
75  import collections
76  import datetime
77  import logging
78  import math
79  import random
80  import re
81  import weakref
82  from recording.manager import Manager
83  from config import FPS
84
85  try:
86      import pygame
87      from pygame import KMOD_CTRL
88      from pygame import KMOD_SHIFT
89      from pygame import K_0
90      from pygame import K_9
91      from pygame import K_BACKQUOTE
92      from pygame import K_BACKSPACE
93      from pygame import K_DOWN
94      from pygame import K_ESCAPE
95      from pygame import K_COMMA
96      from pygame import K_F1
97      from pygame import K_LEFT
98      from pygame import K_PERIOD
99      from pygame import K_RIGHT
100     from pygame import K_SLASH
101     from pygame import K_SPACE
102     from pygame import K_TAB
103     from pygame import K_UP
104     from pygame import K_a
105     from pygame import K_b
106     from pygame import K_c
107     from pygame import K_d
108     from pygame import K_g
109     from pygame import K_h
110     from pygame import K_i
111
112     from pygame import K_l
113     from pygame import K_m
114     from pygame import K_n
115     from pygame import K_o
116     from pygame import K_p
117     from pygame import K_q
118     from pygame import K_r
119     from pygame import K_s
120     from pygame import K_t
121     from pygame import K_v
122     from pygame import K_w
123     from pygame import K_x
124     from pygame import K_z
125     from pygame import K_MINUS
126     from pygame import K_EQUALS
127 except ImportError:
128     raise RuntimeError('cannot import pygame, make sure pygame package is installed')
129
130 try:
131     import numpy as np
132 except ImportError:
133     raise RuntimeError('cannot import numpy, make sure numpy package is installed')
134
135
136 # ==============================================================================
137 # -- Global functions ----------------------------------------------------------
138 # ==============================================================================
139
140
141 def find_weather_presets():
142     rgx = re.compile('.+?(?:(?<=[a-z])(?=[A-Z])|(?<=[A-Z])(?=[A-Z][a-z])|$)')
143     name = lambda x: ' '.join(m.group(0) for m in rgx.finditer(x))
144     presets = [x for x in dir(carla.WeatherParameters) if re.match('[A-Z].+', x)]
145     return [(getattr(carla.WeatherParameters, x), name(x)) for x in presets]
146
147
148 def get_actor_display_name(actor, truncate=250):
149     name = ' '.join(actor.type_id.replace('_', '.').title().split('.')[1:])
150     return (name[:truncate - 1] + u'\u2026') if len(name) > truncate else name
151
152 def get_actor_blueprints(world, filter, generation):
153     bps = world.get_blueprint_library().filter(filter)
154     if generation.lower() == "all":
155         return bps
156
157     # If the filter returns only one bp, we assume that this one needed
158     # and therefore, we ignore the generation
159     if len(bps) == 1:
160         return bps
161
162     try:
163         int_generation = int(generation)
164         # Check if generation is in available generations
165         if int_generation in [1, 2]:
166             bps = [x for x in bps if int(x.get_attribute('generation')) == int_generation]
167             return bps
168         else:
169             print("   Warning! Actor Generation is not valid. No actor will be spawned.")
170             return []
171     except:
172         print("   Warning! Actor Generation is not valid. No actor will be spawned.")
173         return []
174
175
176 # ==============================================================================
177 # -- World ---------------------------------------------------------------------
178 # ==============================================================================
179
180
181 class World(object):
182     def __init__(self, carla_world, hud, args, scenario, client, path_details):
183         self.world = carla_world
184         self.sync = args.sync
185         self.actor_role_name = args.rolename
186         self.path_details = path_details
187         self.scenario = scenario
188         self.client = client
189         self.scenario_name = args.scenario
190         self.participant_name = args.name
191         try:
192             self.map = self.world.get_map()
```

```
        except RuntimeError as error:
            print('RuntimeError: {}'.format(error))
            print('  The server could not send the OpenDRIVE (.xodr) file:')
            print('  Make sure it exists, has the same name of your town, and is correct.')
            sys.exit(1)
        self.spawn_point = None
        self.init_spawn_point(path_details["start_location"])
        self.hud = hud
        self.player = None
        self.collision_sensor = None
        self.lane_invasion_sensor = None
        self.gnss_sensor = None
        self.imu_sensor = None
        self.radar_sensor = None
        self.camera_manager = None
        self._weather_presets = find_weather_presets()
        self._weather_index = 0
        self._actor_filter = args.filter
        self._actor_generation = args.generation
        self._gamma = args.gamma
        self.restart(args)
        self.recording_enabled = 0

        # All about the simulation
        self.simulation_clock = None
        self.simulation_start_tick = 0
        self.simulation_time = 0
        self.manager = None

        self.recording_start = 0
        self.constant_velocity_enabled = False
        self.show_vehicle_telemetry = False
        self.doors_are_open = False
        self.current_map_layer = 0
        self.map_layer_names = [
            carla.MapLayer.NONE,
            carla.MapLayer.Buildings,
            carla.MapLayer.Decals,
            carla.MapLayer.Foliage,
            carla.MapLayer.Ground,
            carla.MapLayer.ParkedVehicles,
            carla.MapLayer.Particles,
            carla.MapLayer.Props,
            carla.MapLayer.StreetLights,
            carla.MapLayer.Walls,
            carla.MapLayer.All
        ]

    def init_spawn_point(self, start_location):
        try:
            self.spawn_point = carla.Transform(carla.Location(x=start_location["x"], y=start_location["y"], z=start_location["z"]), carla.Rotation(pitch=0.0, ya
        except Exception as e:
            logger.error("Could not initiate spawn location for the driver", e)
    def restart(self, args):
        self.player_max_speed = 1.589
        self.player_max_speed_fast = 3.713
        # Keep same camera config if the camera manager exists.
        cam_index = self.camera_manager.index if self.camera_manager is not None else 0
        cam_pos_index = self.camera_manager.transform_index if self.camera_manager is not None else 0
        # Get a random blueprint.
        blueprint = random.choice(get_actor_blueprints(self.world, self._actor_filter, self._actor_generation))
        blueprint.set_attribute('role_name', self.actor_role_name)
        if blueprint.has_attribute('color'):
            blueprint.set_attribute('color', '{},{},{}'.format(args.red, args.green, args.blue))
        if blueprint.has_attribute('driver_id'):
            driver_id = random.choice(blueprint.get_attribute('driver_id').recommended_values)
            blueprint.set_attribute('driver_id', driver_id)
        if blueprint.has_attribute('is_invincible'):
            blueprint.set_attribute('is_invincible', 'true')
        # set the max speed
        if blueprint.has_attribute('speed'):
            self.player_max_speed = float(blueprint.get_attribute('speed').recommended_values[1])
            self.player_max_speed_fast = float(blueprint.get_attribute('speed').recommended_values[2])

        # Spawn the player.
        if self.player is not None:
            spawn_point = self.player.get_transform()
            spawn_point.location.z += 2.0
            spawn_point.rotation.roll = 0.0
            spawn_point.rotation.pitch = 0.0
            self.destroy()
            self.player = self.world.try_spawn_actor(blueprint, spawn_point)
            self.show_vehicle_telemetry = False
            self.modify_vehicle_physics(self.player)
        while self.player is None:
            if not self.map.get_spawn_points():
                print('There are no spawn points available in your map/town.')
                print('Please add some Vehicle Spawn Point to your UE4 scene.')
                sys.exit(1)
            if self.spawn_point is None:
                spawn_points = self.map.get_spawn_points()
                self.spawn_point = random.choice(spawn_points) if spawn_points else carla.Transform()
            self.player = self.world.try_spawn_actor(blueprint, self.spawn_point)
            self.show_vehicle_telemetry = False
            self.modify_vehicle_physics(self.player)
        # Set up the sensors.
        self.collision_sensor = CollisionSensor(self.player, self.hud)
        self.lane_invasion_sensor = LaneInvasionSensor(self.player, self.hud)
        self.gnss_sensor = GnssSensor(self.player)
        self.imu_sensor = IMUSensor(self.player)
        self.camera_manager = CameraManager(self.player, self.hud, self._gamma)
        self.camera_manager.transform_index = cam_pos_index
        self.camera_manager.set_sensor(cam_index, notify=False)
        actor_type = get_actor_display_name(self.player)
        self.hud.notification(actor_type)

        if self.sync:
            self.world.tick()
        else:
            self.world.wait_for_tick()

    def toggle_radar(self):
        if self.radar_sensor is None:
            self.radar_sensor = RadarSensor(self.player)
        elif self.radar_sensor.sensor is not None:
            self.radar_sensor.sensor.destroy()
            self.radar_sensor = None

    def modify_vehicle_physics(self, actor):
        #If actor is not a vehicle, we cannot use the physics control
        try:
            physics_control = actor.get_physics_control()
            physics_control.use_sweep_wheel_collision = True
            actor.apply_physics_control(physics_control)
        except Exception:
            pass

    def startScenario(self):
        if self.simulation_clock is None:
            logger.info("Starting the scenario")
```

```
            self.scenario.start()
            self.hud.notification('The simulation has started')
            self.manager = Manager(self.path_details["path_checkpoints"], self.world, self.player, self.participant_name, self.scenario_name)
            #self.collision_sensor = self.manager.collision_sensor
            self.simulation_clock = pygame.time.Clock()
            self.simulation_start_tick = pygame.time.get_ticks()

    def finishScenario(self):
        logger.warning("Finish was NOT REACHED. Terminating the simulation.")
        self.manager.shut_down(False)
        self.scenario.finish(self.client)

    def tick(self):
        if self.simulation_clock is not None:
            self.simulation_clock.tick(60)
            self.simulation_time += self.simulation_clock.get_time()
            if self.manager.record(self.simulation_time):
                return True
            self.hud.tick(self, self.simulation_time/1000)
        else:
            self.hud.tick(self, 0)

    def render(self, display):
        self.camera_manager.render(display)
        self.hud.render(display)

    def destroy_sensors(self):
        self.camera_manager.sensor.destroy()
        self.camera_manager.sensor = None
        self.camera_manager.index = None

    def destroy(self):
        if self.radar_sensor is not None:
            self.toggle_radar()
        sensors = [
            self.camera_manager.sensor,
            self.collision_sensor.sensor,
            self.lane_invasion_sensor.sensor,
            self.gnss_sensor.sensor,
            self.imu_sensor.sensor]
        for sensor in sensors:
            if sensor is not None:
                sensor.stop()
                sensor.destroy()
        if self.player is not None:
            self.player.destroy()


# ==============================================================================
# -- KeyboardControl -----------------------------------------------------------
# ==============================================================================


class KeyboardControl(object):
    """Class that handles keyboard input."""
    def __init__(self, world, start_in_autopilot):
        self._autopilot_enabled = start_in_autopilot
        if isinstance(world.player, carla.Vehicle):
            self._control = carla.VehicleControl()
            self._lights = carla.VehicleLightState.NONE
            world.player.set_autopilot(self._autopilot_enabled)
            world.player.set_light_state(self._lights)
        elif isinstance(world.player, carla.Walker):
            self._control = carla.WalkerControl()
            self._autopilot_enabled = False
            self._rotation = world.player.get_transform().rotation
        else:
            raise NotImplementedError("Actor type not supported")
        self._steer_cache = 0.0
        world.hud.notification("Press 'H' or '?' for help.", seconds=4.0)

    def parse_events(self, client, world, clock, sync_mode):
        if isinstance(self._control, carla.VehicleControl):
            current_lights = self._lights
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                return True
            elif event.type == pygame.KEYUP:
                if self._is_quit_shortcut(event.key):
                    return True
                elif event.key == K_F1:
                    world.hud.toggle_info()
                elif event.key == K_h or (event.key == K_SLASH and pygame.key.get_mods() & KMOD_SHIFT):
                    world.hud.help.toggle()
                elif event.key == K_TAB:
                    world.camera_manager.toggle_camera()
                elif event.key == K_BACKSPACE and not world.simulation_clock:
                    world.startScenario()
                elif event.key == K_BACKSPACE and world.simulation_clock:
                    world.finishScenario()
                    return True
                # Driving key scenarios
                if isinstance(self._control, carla.VehicleControl):
                    if event.key == K_q:
                        self._control.gear = 1 if self._control.reverse else -1
                    elif event.key == K_m:
                        self._control.manual_gear_shift = not self._control.manual_gear_shift
                        self._control.gear = world.player.get_control().gear
                        world.hud.notification('%s Transmission' %
                                               ('Manual' if self._control.manual_gear_shift else 'Automatic'))
                    elif self._control.manual_gear_shift and event.key == K_COMMA:
                        self._control.gear = max(-1, self._control.gear - 1)
                    elif self._control.manual_gear_shift and event.key == K_PERIOD:
                        self._control.gear = self._control.gear + 1
                    elif event.key == K_p and not pygame.key.get_mods() & KMOD_CTRL:
                        if not self._autopilot_enabled and not sync_mode:
                            print("WARNING: You are currently in asynchronous mode and could "
                                  "experience some issues with the traffic simulation")
                        self._autopilot_enabled = not self._autopilot_enabled
                        world.player.set_autopilot(self._autopilot_enabled)
                        world.hud.notification(
                            'Autopilot %s' % ('On' if self._autopilot_enabled else 'Off'))
                    elif event.key == K_l and pygame.key.get_mods() & KMOD_CTRL:
                        current_lights ^= carla.VehicleLightState.Special1
                    elif event.key == K_l and pygame.key.get_mods() & KMOD_SHIFT:
                        current_lights ^= carla.VehicleLightState.HighBeam
                    elif event.key == K_l:
                        # Use 'L' key to switch between lights:
                        # closed -> position -> low beam -> fog
                        if not self._lights & carla.VehicleLightState.Position:
                            world.hud.notification("Position lights")
                            current_lights |= carla.VehicleLightState.Position
                        else:
                            world.hud.notification("Low beam lights")
                            current_lights |= carla.VehicleLightState.LowBeam
                        if self._lights & carla.VehicleLightState.LowBeam:
                            world.hud.notification("Fog lights")
                            current_lights |= carla.VehicleLightState.Fog
                        if self._lights & carla.VehicleLightState.Fog:
                            world.hud.notification("Lights off")
```

```
                        current_lights ^= carla.VehicleLightState.Position
                        current_lights ^= carla.VehicleLightState.LowBeam
                        current_lights ^= carla.VehicleLightState.Fog
                    elif event.key == K_i:
                        current_lights ^= carla.VehicleLightState.Interior
                    elif event.key == K_z:
                        current_lights ^= carla.VehicleLightState.LeftBlinker
                    elif event.key == K_x:
                        current_lights ^= carla.VehicleLightState.RightBlinker

        if not self._autopilot_enabled:
            if isinstance(self._control, carla.VehicleControl):
                self._parse_vehicle_keys(pygame.key.get_pressed(), clock.get_time())
                self._control.reverse = self._control.gear < 0
                # Set automatic control-related vehicle lights
                if self._control.brake:
                    current_lights |= carla.VehicleLightState.Brake
                else: # Remove the Brake flag
                    current_lights &= ~carla.VehicleLightState.Brake
                if self._control.reverse:
                    current_lights |= carla.VehicleLightState.Reverse
                else: # Remove the Reverse flag
                    current_lights &= ~carla.VehicleLightState.Reverse
                if current_lights != self._lights: # Change the light state only if necessary
                    self._lights = current_lights
                    world.player.set_light_state(carla.VehicleLightState(self._lights))
            elif isinstance(self._control, carla.WalkerControl):
                self._parse_walker_keys(pygame.key.get_pressed(), clock.get_time(), world)
            world.player.apply_control(self._control)

    def _parse_vehicle_keys(self, keys, milliseconds):
        if keys[K_UP] or keys[K_w]:
            self._control.throttle = min(self._control.throttle + 0.01, 1.00)
        else:
            self._control.throttle = 0.0

        if keys[K_DOWN] or keys[K_s]:
            self._control.brake = min(self._control.brake + 0.2, 1)
        else:
            self._control.brake = 0

        steer_increment = 5e-4 * milliseconds
        if keys[K_LEFT] or keys[K_a]:
            if self._steer_cache > 0:
                self._steer_cache = 0
            else:
                self._steer_cache -= steer_increment
        elif keys[K_RIGHT] or keys[K_d]:
            if self._steer_cache < 0:
                self._steer_cache = 0
            else:
                self._steer_cache += steer_increment
        else:
            self._steer_cache = 0.0
        self._steer_cache = min(0.7, max(-0.7, self._steer_cache))
        self._control.steer = round(self._steer_cache, 1)
        self._control.hand_brake = keys[K_SPACE]

    def _parse_walker_keys(self, keys, milliseconds, world):
        self._control.speed = 0.0
        if keys[K_DOWN] or keys[K_s]:
            self._control.speed = 0.0
        if keys[K_LEFT] or keys[K_a]:
            self._control.speed = .01
            self._rotation.yaw -= 0.08 * milliseconds
        if keys[K_RIGHT] or keys[K_d]:
            self._control.speed = .01
            self._rotation.yaw += 0.08 * milliseconds
        if keys[K_UP] or keys[K_w]:
            self._control.speed = world.player_max_speed_fast if pygame.key.get_mods() & KMOD_SHIFT else world.player_max_speed
        self._control.jump = keys[K_SPACE]
        self._rotation.yaw = round(self._rotation.yaw, 1)
        self._control.direction = self._rotation.get_forward_vector()

    @staticmethod
    def _is_quit_shortcut(key):
        return (key == K_ESCAPE) or (key == K_q and pygame.key.get_mods() & KMOD_CTRL)


# ==============================================================================
# -- HUD -----------------------------------------------------------------------
# ==============================================================================


class HUD(object):
    def __init__(self, width, height):
        self.dim = (width, height)
        font = pygame.font.Font(pygame.font.get_default_font(), 20)
        font_name = 'courier' if os.name == 'nt' else 'mono'
        fonts = [x for x in pygame.font.get_fonts() if font_name in x]
        default_font = 'ubuntumono'
        mono = default_font if default_font in fonts else fonts[0]
        mono = pygame.font.match_font(mono)
        self._font_mono = pygame.font.Font(mono, 12 if os.name == 'nt' else 14)
        self._notifications = FadingText(font, (width, 40), (0, height - 40))
        self.help = HelpText(pygame.font.Font(mono, 16), width, height)
        self.server_fps = 0
        self.frame = 0
        self.simulation_time = 0
        self._show_info = True
        self._info_text = []
        self._server_clock = pygame.time.Clock()

    def on_world_tick(self, timestamp):
        self._server_clock.tick()
        self.server_fps = self._server_clock.get_fps()
        self.frame = timestamp.frame
        self.simulation_time = timestamp.elapsed_seconds

    def tick(self, world, time):
        self._notifications.tick(world, time)
        if not self._show_info:
            return
        t = world.player.get_transform()
        v = world.player.get_velocity()
        c = world.player.get_control()
        compass = world.imu_sensor.compass
        heading = 'N' if compass > 270.5 or compass < 89.5 else ''
        heading += 'S' if 90.5 < compass < 269.5 else ''
        heading += 'E' if 0.5 < compass < 179.5 else ''
        heading += 'W' if 180.5 < compass < 359.5 else ''
        colhist = world.collision_sensor.get_collision_history()
        collision = [colhist[x + self.frame - 200] for x in range(0, 200)]
        max_col = max(1.0, max(collision))
        collision = [x / max_col for x in collision]
        vehicles = world.world.get_actors().filter('vehicle.*')
        self._info_text = [
            'Vehicle: % 20s' % get_actor_display_name(world.player, truncate=20),
            'Route time: % 12s' % datetime.timedelta(seconds=int(time)),
            '',
```

```
            'Speed:   % 15.0f km/h' % (3.6 * math.sqrt(v.x**2 + v.y**2 + v.z**2)),
            '']
        if isinstance(c, carla.VehicleControl):
            self._info_text += [
                ('Throttle:', c.throttle, 0.0, 1.0),
                ('Steer:', c.steer, -1.0, 1.0),
                ('Brake:', c.brake, 0.0, 1.0),
                ('Reverse:', c.reverse),
                ('Hand brake:', c.hand_brake),
                ('Manual:', c.manual_gear_shift),
                'Gear:        %s' % {-1: 'R', 0: 'N'}.get(c.gear, c.gear)]
        self._info_text += [
            '',
            'Collision:',
            collision,
            '']

    def toggle_info(self):
        self._show_info = not self._show_info

    def notification(self, text, seconds=2.0):
        self._notifications.set_text(text, seconds=seconds)

    def error(self, text):
        self._notifications.set_text('Error: %s' % text, (255, 0, 0))

    def render(self, display):
        if self._show_info:
            info_surface = pygame.Surface((220, self.dim[1]))
            info_surface.set_alpha(100)
            display.blit(info_surface, (0, 0))
            v_offset = 4
            bar_h_offset = 100
            bar_width = 106
            for item in self._info_text:
                if v_offset + 18 > self.dim[1]:
                    break
                if isinstance(item, list):
                    if len(item) > 1:
                        points = [(x + 8, v_offset + 8 + (1.0 - y) * 30) for x, y in enumerate(item)]
                        pygame.draw.lines(display, (255, 136, 0), False, points, 2)
                    item = None
                    v_offset += 18
                elif isinstance(item, tuple):
                    if isinstance(item[1], bool):
                        rect = pygame.Rect((bar_h_offset, v_offset + 8), (6, 6))
                        pygame.draw.rect(display, (255, 255, 255), rect, 0 if item[1] else 1)
                    else:
                        rect_border = pygame.Rect((bar_h_offset, v_offset + 8), (bar_width, 6))
                        pygame.draw.rect(display, (255, 255, 255), rect_border, 1)
                        f = (item[1] - item[2]) / (item[3] - item[2])
                        if item[2] < 0.0:
                            rect = pygame.Rect((bar_h_offset + f * (bar_width - 6), v_offset + 8), (6, 6))
                        else:
                            rect = pygame.Rect((bar_h_offset, v_offset + 8), (f * bar_width, 6))
                        pygame.draw.rect(display, (255, 255, 255), rect)
                    item = item[0]
                if item:  # At this point has to be a str.
                    surface = self._font_mono.render(item, True, (255, 255, 255))
                    display.blit(surface, (8, v_offset))
                v_offset += 18
        self._notifications.render(display)
        self.help.render(display)


# ==============================================================================
# -- FadingText ----------------------------------------------------------------
# ==============================================================================


class FadingText(object):
    def __init__(self, font, dim, pos):
        self.font = font
        self.dim = dim
        self.pos = pos
        self.seconds_left = 0
        self.surface = pygame.Surface(self.dim)

    def set_text(self, text, color=(255, 255, 255), seconds=2.0):
        text_texture = self.font.render(text, True, color)
        self.surface = pygame.Surface(self.dim)
        self.seconds_left = seconds
        self.surface.fill((0, 0, 0, 0))
        self.surface.blit(text_texture, (10, 11))

    def tick(self, _, time):
        delta_seconds = 1e-3 * time
        self.seconds_left = max(0.0, self.seconds_left - delta_seconds)
        self.surface.set_alpha(500.0 * self.seconds_left)

    def render(self, display):
        display.blit(self.surface, self.pos)


# ==============================================================================
# -- HelpText ------------------------------------------------------------------
# ==============================================================================


class HelpText(object):
    """Helper class to handle text output using pygame"""
    def __init__(self, font, width, height):
        lines = __doc__.split('\n')
        self.font = font
        self.line_space = 18
        self.dim = (780, len(lines) * self.line_space + 12)
        self.pos = (0.5 * width - 0.5 * self.dim[0], 0.5 * height - 0.5 * self.dim[1])
        self.seconds_left = 0
        self.surface = pygame.Surface(self.dim)
        self.surface.fill((0, 0, 0, 0))
        for n, line in enumerate(lines):
            text_texture = self.font.render(line, True, (255, 255, 255))
            self.surface.blit(text_texture, (22, n * self.line_space))
            self._render = False
        self.surface.set_alpha(220)

    def toggle(self):
        self._render = not self._render

    def render(self, display):
        if self._render:
            display.blit(self.surface, self.pos)



# ==============================================================================
# -- CollisionSensor -----------------------------------------------------------
# ==============================================================================


```

```
class CollisionSensor(object):
    def __init__(self, parent_actor, hud):
        self.sensor = None
        self.history = []
        self._parent = parent_actor
        self.hud = hud
        world = self._parent.get_world()
        bp = world.get_blueprint_library().find('sensor.other.collision')
        self.sensor = world.spawn_actor(bp, carla.Transform(), attach_to=self._parent)
        # We need to pass the lambda a weak reference to self to avoid circular
        # reference.
        weak_self = weakref.ref(self)
        self.sensor.listen(lambda event: CollisionSensor._on_collision(weak_self, event))

    def get_collision_history(self):
        history = collections.defaultdict(int)
        for frame, intensity in self.history:
            history[frame] += intensity
        return history

    @staticmethod
    def _on_collision(weak_self, event):
        self = weak_self()
        if not self:
            return
        actor_type = get_actor_display_name(event.other_actor)
        self.hud.notification('Collision with %r' % actor_type)
        impulse = event.normal_impulse
        intensity = math.sqrt(impulse.x**2 + impulse.y**2 + impulse.z**2)
        self.history.append((event.frame, intensity))
        if len(self.history) > 4000:
            self.history.pop(0)


# ==============================================================================
# -- LaneInvasionSensor --------------------------------------------------------
# ==============================================================================


class LaneInvasionSensor(object):
    def __init__(self, parent_actor, hud):
        self.sensor = None

        # If the spawn object is not a vehicle, we cannot use the Lane Invasion Sensor
        if parent_actor.type_id.startswith("vehicle."):
            self._parent = parent_actor
            self.hud = hud
            world = self._parent.get_world()
            bp = world.get_blueprint_library().find('sensor.other.lane_invasion')
            self.sensor = world.spawn_actor(bp, carla.Transform(), attach_to=self._parent)
            # We need to pass the lambda a weak reference to self to avoid circular
            # reference.
            weak_self = weakref.ref(self)
            self.sensor.listen(lambda event: LaneInvasionSensor._on_invasion(weak_self, event))

    @staticmethod
    def _on_invasion(weak_self, event):
        self = weak_self()
        if not self:
            return
        lane_types = set(x.type for x in event.crossed_lane_markings)
        text = ['%r' % str(x).split()[-1] for x in lane_types]
        self.hud.notification('Crossed line %s' % ' and '.join(text))


# ==============================================================================
# -- GnssSensor ----------------------------------------------------------------
# ==============================================================================


class GnssSensor(object):
    def __init__(self, parent_actor):
        self.sensor = None
        self._parent = parent_actor
        self.lat = 0.0
        self.lon = 0.0
        world = self._parent.get_world()
        bp = world.get_blueprint_library().find('sensor.other.gnss')
        self.sensor = world.spawn_actor(bp, carla.Transform(carla.Location(x=1.0, z=2.8)), attach_to=self._parent)
        # We need to pass the lambda a weak reference to self to avoid circular
        # reference.
        weak_self = weakref.ref(self)
        self.sensor.listen(lambda event: GnssSensor._on_gnss_event(weak_self, event))

    @staticmethod
    def _on_gnss_event(weak_self, event):
        self = weak_self()
        if not self:
            return
        self.lat = event.latitude
        self.lon = event.longitude


# ==============================================================================
# -- IMUSensor -----------------------------------------------------------------
# ==============================================================================


class IMUSensor(object):
    def __init__(self, parent_actor):
        self.sensor = None
        self._parent = parent_actor
        self.accelerometer = (0.0, 0.0, 0.0)
        self.gyroscope = (0.0, 0.0, 0.0)
        self.compass = 0.0
        world = self._parent.get_world()
        bp = world.get_blueprint_library().find('sensor.other.imu')
        self.sensor = world.spawn_actor(
            bp, carla.Transform(), attach_to=self._parent)
        # We need to pass the lambda a weak reference to self to avoid circular
        # reference.
        weak_self = weakref.ref(self)
        self.sensor.listen(
            lambda sensor_data: IMUSensor._IMU_callback(weak_self, sensor_data))

    @staticmethod
    def _IMU_callback(weak_self, sensor_data):
        self = weak_self()
        if not self:
            return
        limits = (-99.9, 99.9)
        self.accelerometer = (
            max(limits[0], min(limits[1], sensor_data.accelerometer.x)),
            max(limits[0], min(limits[1], sensor_data.accelerometer.y)),
            max(limits[0], min(limits[1], sensor_data.accelerometer.z)))
        self.gyroscope = (
            max(limits[0], min(limits[1], math.degrees(sensor_data.gyroscope.x))),
            max(limits[0], min(limits[1], math.degrees(sensor_data.gyroscope.y))),
            max(limits[0], min(limits[1], math.degrees(sensor_data.gyroscope.z))))
        self.compass = math.degrees(sensor_data.compass)
```

```
843
844
845 # ==============================================================================
846 # -- RadarSensor ---------------------------------------------------------------
847 # ==============================================================================
848
849
850 class RadarSensor(object):
851     def __init__(self, parent_actor):
852         self.sensor = None
853         self._parent = parent_actor
854         bound_x = 0.5 + self._parent.bounding_box.extent.x
855         bound_y = 0.5 + self._parent.bounding_box.extent.y
856         bound_z = 0.5 + self._parent.bounding_box.extent.z
857
858         self.velocity_range = 7.5 # m/s
859         world = self._parent.get_world()
860         self.debug = world.debug
861         bp = world.get_blueprint_library().find('sensor.other.radar')
862         bp.set_attribute('horizontal_fov', str(35))
863         bp.set_attribute('vertical_fov', str(20))
864         self.sensor = world.spawn_actor(
865             bp,
866             carla.Transform(
867                 carla.Location(x=bound_x + 0.05, z=bound_z+0.05),
868                 carla.Rotation(pitch=5)),
869             attach_to=self._parent)
870         # We need a weak reference to self to avoid circular reference.
871         weak_self = weakref.ref(self)
872         self.sensor.listen(
873             lambda radar_data: RadarSensor._Radar_callback(weak_self, radar_data))
874
875     @staticmethod
876     def _Radar_callback(weak_self, radar_data):
877         self = weak_self()
878         if not self:
879             return
880         # To get a numpy [[vel, altitude, azimuth, depth],...[,,,]]:
881         # points = np.frombuffer(radar_data.raw_data, dtype=np.dtype('f4'))
882         # points = np.reshape(points, (len(radar_data), 4))
883
884         current_rot = radar_data.transform.rotation
885         for detect in radar_data:
886             azi = math.degrees(detect.azimuth)
887             alt = math.degrees(detect.altitude)
888             # The 0.25 adjusts a bit the distance so the dots can
889             # be properly seen
890             fw_vec = carla.Vector3D(x=detect.depth - 0.25)
891             carla.Transform(
892                 carla.Location(),
893                 carla.Rotation(
894                     pitch=current_rot.pitch + alt,
895                     yaw=current_rot.yaw + azi,
896                     roll=current_rot.roll)).transform(fw_vec)
897
898             def clamp(min_v, max_v, value):
899                 return max(min_v, min(value, max_v))
900
901             norm_velocity = detect.velocity / self.velocity_range # range [-1, 1]
902             r = int(clamp(0.0, 1.0, 1.0 - norm_velocity) * 255.0)
903             g = int(clamp(0.0, 1.0, 1.0 - abs(norm_velocity)) * 255.0)
904             b = int(abs(clamp(- 1.0, 0.0, - 1.0 - norm_velocity)) * 255.0)
905             self.debug.draw_point(
906                 radar_data.transform.location + fw_vec,
907                 size=0.075,
908                 life_time=0.06,
909                 persistent_lines=False,
910                 color=carla.Color(r, g, b))
911
912 # ==============================================================================
913 # -- CameraManager -------------------------------------------------------------
914 # ==============================================================================
915
916
917 class CameraManager(object):
918     def __init__(self, parent_actor, hud, gamma_correction):
919         self.sensor = None
920         self.surface = None
921         self._parent = parent_actor
922         self.hud = hud
923         self.recording = False
924         bound_x = 0.5 + self._parent.bounding_box.extent.x
925         bound_y = 0.5 + self._parent.bounding_box.extent.y
926         bound_z = 0.5 + self._parent.bounding_box.extent.z
927         Attachment = carla.AttachmentType
928
929         self._camera_transforms = [
930             # Third-person
931             (carla.Transform(carla.Location(x=-5.0, z=2.0), carla.Rotation(pitch=6.0)), Attachment.SpringArm),
932             # Watch back
933             (carla.Transform(carla.Location(x=4.0, z=2.0), carla.Rotation(yaw=0, pitch=6)), Attachment.SpringArm),
934             # Watch to the left
935             (carla.Transform(carla.Location(x=0, z=2.0, y=3), carla.Rotation(yaw=-90, pitch=0)), Attachment.Rigid),
936             # Watch to the right
937             (carla.Transform(carla.Location(x=0, z=2.0, y=-3), carla.Rotation(yaw=90, pitch=0)), Attachment.Rigid),
938             # First-person
939             (carla.Transform(carla.Location(y=-0.42, x=0.03, z=1.2), carla.Rotation(yaw=0, roll=0, pitch=-10)), Attachment.Rigid)]
940
941         self.transform_index = 1
942         self.sensors = [
943             ['sensor.camera.rgb', cc.Raw, 'Camera RGB', {}],
944             ['sensor.camera.depth', cc.Raw, 'Camera Depth (Raw)', {}],
945             ['sensor.camera.depth', cc.Depth, 'Camera Depth (Gray Scale)', {}],
946             ['sensor.camera.depth', cc.LogarithmicDepth, 'Camera Depth (Logarithmic Gray Scale)', {}],
947             ['sensor.camera.semantic_segmentation', cc.Raw, 'Camera Semantic Segmentation (Raw)', {}],
948             ['sensor.camera.semantic_segmentation', cc.CityScapesPalette, 'Camera Semantic Segmentation (CityScapes Palette)', {}],
949             ['sensor.camera.instance_segmentation', cc.CityScapesPalette, 'Camera Instance Segmentation (CityScapes Palette)', {}],
950             ['sensor.camera.instance_segmentation', cc.Raw, 'Camera Instance Segmentation (Raw)', {}],
951             ['sensor.lidar.ray_cast', None, 'Lidar (Ray-Cast)', {'range': '50'}],
952             ['sensor.camera.dvs', cc.Raw, 'Dynamic Vision Sensor', {}],
953             ['sensor.camera.rgb', cc.Raw, 'Camera RGB Distorted',
954                 {'lens_circle_multiplier': '3.0',
955                 'lens_circle_falloff': '3.0',
956                 'chromatic_aberration_intensity': '0.5',
957                 'chromatic_aberration_offset': '0'}],
958             ['sensor.camera.optical_flow', cc.Raw, 'Optical Flow', {}],
959         ]
960         world = self._parent.get_world()
961         bp_library = world.get_blueprint_library()
962         for item in self.sensors:
963             bp = bp_library.find(item[0])
964             if item[0].startswith('sensor.camera'):
965                 bp.set_attribute('image_size_x', str(hud.dim[0]))
966                 bp.set_attribute('image_size_y', str(hud.dim[1]))
967                 if bp.has_attribute('gamma'):
968                     bp.set_attribute('gamma', str(gamma_correction))
969                 for attr_name, attr_value in item[3].items():
970                     bp.set_attribute(attr_name, attr_value)
971             elif item[0].startswith('sensor.lidar'):
972                 self.lidar_range = 50
```

```
                for attr_name, attr_value in item[3].items():
                    bp.set_attribute(attr_name, attr_value)
                    if attr_name == 'range':
                        self.lidar_range = float(attr_value)

            item.append(bp)
        self.index = None

    def toggle_camera(self):
        self.transform_index = (self.transform_index + 1) % len(self._camera_transforms)
        self.set_sensor(self.index, notify=False, force_respawn=True)

    def set_sensor(self, index, notify=True, force_respawn=False):
        index = index % len(self.sensors)
        needs_respawn = True if self.index is None else \
            (force_respawn or (self.sensors[index][2] != self.sensors[self.index][2]))
        if needs_respawn:
            if self.sensor is not None:
                self.sensor.destroy()
                self.surface = None
            self.sensor = self._parent.get_world().spawn_actor(
                self.sensors[index][-1],
                self._camera_transforms[self.transform_index][0],
                attach_to=self._parent,
                attachment_type=self._camera_transforms[self.transform_index][1])
            # We need to pass the lambda a weak reference to self to avoid
            # circular reference.
            weak_self = weakref.ref(self)
            self.sensor.listen(lambda image: CameraManager._parse_image(weak_self, image))
        if notify:
            self.hud.notification(self.sensors[index][2])
        self.index = index

    def next_sensor(self):
        self.set_sensor(self.index + 1)

    def toggle_recording(self):
        self.recording = not self.recording
        self.hud.notification('Recording %s' % ('On' if self.recording else 'Off'))

    def render(self, display):
        if self.surface is not None:
            display.blit(self.surface, (0, 0))

    @staticmethod
    def _parse_image(weak_self, image):
        self = weak_self()
        if not self:
            return
        if self.sensors[self.index][0].startswith('sensor.lidar'):
            points = np.frombuffer(image.raw_data, dtype=np.dtype('f4'))
            points = np.reshape(points, (int(points.shape[0] / 4), 4))
            lidar_data = np.array(points[:, :2])
            lidar_data *= min(self.hud.dim) / (2.0 * self.lidar_range)
            lidar_data += (0.5 * self.hud.dim[0], 0.5 * self.hud.dim[1])
            lidar_data = np.fabs(lidar_data)  # pylint: disable=E1111
            lidar_data = lidar_data.astype(np.int32)
            lidar_data = np.reshape(lidar_data, (-1, 2))
            lidar_img_size = (self.hud.dim[0], self.hud.dim[1], 3)
            lidar_img = np.zeros((lidar_img_size), dtype=np.uint8)
            lidar_img[tuple(lidar_data.T)] = (255, 255, 255)
            self.surface = pygame.surfarray.make_surface(lidar_img)
        elif self.sensors[self.index][0].startswith('sensor.camera.dvs'):
            # Example of converting the raw_data from a carla.DVSEventArray
            # sensor into a NumPy array and using it as an image
            dvs_events = np.frombuffer(image.raw_data, dtype=np.dtype([
                ('x', np.uint16), ('y', np.uint16), ('t', np.int64), ('pol', np.bool)]))
            dvs_img = np.zeros((image.height, image.width, 3), dtype=np.uint8)
            # Blue is positive, red is negative
            dvs_img[dvs_events[:]['y'], dvs_events[:]['x'], dvs_events[:]['pol'] * 2] = 255
            self.surface = pygame.surfarray.make_surface(dvs_img.swapaxes(0, 1))
        elif self.sensors[self.index][0].startswith('sensor.camera.optical_flow'):
            image = image.get_color_coded_flow()
            array = np.frombuffer(image.raw_data, dtype=np.dtype("uint8"))
            array = np.reshape(array, (image.height, image.width, 4))
            array = array[:, :, :3]
            array = array[:, :, ::-1]
            self.surface = pygame.surfarray.make_surface(array.swapaxes(0, 1))
        else:
            image.convert(self.sensors[self.index][1])
            array = np.frombuffer(image.raw_data, dtype=np.dtype("uint8"))
            array = np.reshape(array, (image.height, image.width, 4))
            array = array[:, :, :3]
            array = array[:, :, ::-1]
            self.surface = pygame.surfarray.make_surface(array.swapaxes(0, 1))
        if self.recording:
            image.save_to_disk('_out/%08d' % image.frame)


# ==============================================================================
# -- game_loop() ---------------------------------------------------------------
# ==============================================================================


def game_loop(args, scenario, path_details):
    pygame.init()
    pygame.font.init()
    world = None
    original_settings = None
    try:
        client = carla.Client(args.host, args.port)
        client.set_timeout(20.0)
        sim_world = client.get_world()
        if args.sync:
            original_settings = sim_world.get_settings()
            settings = sim_world.get_settings()
            if not settings.synchronous_mode:
                settings.synchronous_mode = True
                settings.fixed_delta_seconds = 0.05
            sim_world.apply_settings(settings)
            traffic_manager = client.get_trafficmanager()
            traffic_manager.set_synchronous_mode(True)

        if args.autopilot and not sim_world.get_settings().synchronous_mode:
            print("WARNING: You are currently in asynchronous mode and could "
                  "experience some issues with the traffic simulation")

        display = pygame.display.set_mode(
            (args.width, args.height),
            pygame.HWSURFACE | pygame.DOUBLEBUF)
        display.fill((0,0,0))
        pygame.display.flip()
        ### MY PART ###
        scenario.setup(traffic_manager, sim_world)
        scenario.spawn(client)
        logger.info("The scenario was set. Spawning the hero vehicle and launching the simulation")
        ###   ###   ###
        client.set_timeout(10.0)
        hud = HUD(args.width, args.height)
```

```
        world = World(sim_world, hud, args, scenario, client, path_details)
        controller = KeyboardControl(world, args.autopilot)
        if args.sync:
            sim_world.tick()
        else:
            sim_world.wait_for_tick()

        clock = pygame.time.Clock()
        while True:
            try:
                if args.sync:
                    sim_world.tick()
                clock.tick_busy_loop(FPS)
                if controller.parse_events(client, world, clock, args.sync):
                    return
                if world.tick():
                    return
                world.render(display)
                pygame.display.flip()
            except Exception as e:
                logger.error(e)
    except Exception as e:
        logger.error("Something went wrong trying to launch the driver.py", e)
    finally:
        ### MY PART ###
        if(scenario.populated):
            scenario.finish(client)
        if(world and world.manager):
            logger.info('Total time simulated: {}s'.format(str(world.simulation_time/1000)))
        ###   ###   ###

        if original_settings:
            sim_world.apply_settings(original_settings)

        if (world and world.recording_enabled):
            client.stop_recorder()

        if world is not None:
            world.destroy()

        pygame.quit()

# ==============================================================================
# -- main() --------------------------------------------------------------------
# ==============================================================================

argparser = argparse.ArgumentParser(
    description='CARLA Manual Control Client')
argparser.add_argument(
    '-v', '--verbose',
    action='store_true',
    dest='debug',
    help='print debug information')
argparser.add_argument(
    '--host',
    metavar='H',
    default='127.0.0.1',
    help='IP of the host server (default: 127.0.0.1)')
argparser.add_argument(
    '-p', '--port',
    metavar='P',
    default=2000,
    type=int,
    help='TCP port to listen to (default: 2000)')
argparser.add_argument(
    '-a', '--autopilot',
    action='store_true',
    help='enable autopilot')
argparser.add_argument(
    '--res',
    metavar='WIDTHxHEIGHT',
    default='1280x720',
    help='window resolution (default: 1280x720)')
argparser.add_argument(
    '--filter', '-f',
    metavar='PATTERN',
    default='vehicle.*',
    help='actor filter (default: "vehicle.*")')
argparser.add_argument(
    '--generation',
    metavar='G',
    default='2',
    help='restrict to certain actor generation (values: "1","2","All" - default: "2")')
argparser.add_argument(
    '--rolename',
    metavar='NAME',
    default='hero',
    help='actor role name (default: "hero")')
argparser.add_argument(
    '--gamma',
    default=2.2,
    type=float,
    help='Gamma correction of the camera (default: 2.2)')
argparser.add_argument(
    '--sync',
    action='store_true',
    help='Activate synchronous mode execution')
argparser.add_argument(
    '--red',
    default="0",
    help='Red part of the RGB color palette to paint the hero car')
argparser.add_argument(
    '--green',
    default="0",
    help='Green part of the RGB color palette to paint the hero car')
argparser.add_argument(
    '--blue',
    default="0",
    help='Blue part of the RGB color palette to paint the hero car')
argparser.add_argument(
    '-n', '--name',
    metavar='F',
    default="example",
    help='Participant name (name)')
argparser.add_argument(
    '-s', '--scenario',
    metavar='F',
    default="example",
    help='Scenario name (name)')
argparser.add_argument("s_b64")
argparser.add_argument("p_b64")
args = argparser.parse_args()

logger = logging.getLogger("driver-thread")
logging.basicConfig(level=logging.INFO, filename="../../data/recordings/{}/session_logs.log".format(args.name), filemode="a", format="%(asctime)s - %(name)s - %

def main():
    args.width, args.height = [int(x) for x in args.res.split('x')]

    print(__doc__)
```

```

    try:

        scenario_b64 = args.s_b64
        path_b64 = args.p_b64

        scenario_bytes = base64.b64decode(scenario_b64.encode())
        path_bytes = base64.b64decode(path_b64.encode())

        scenario = pickle.loads(scenario_bytes)
        path_details = pickle.loads(path_bytes)
        game_loop(args, scenario, path_details)

    except KeyboardInterrupt as e:
        logger.warning('Keyboard interupt while running driver.py script', e)
    except Exception as e:
        logger.error("Something went wrong.", e)


if __name__ == '__main__':

    main()
```

File: ./software/carla_scripts/helper_scripts/print_coordinates.py

```
import glob
import os
import sys
import time
import random

from agents.navigation.behavior_agent import BehaviorAgent, BasicAgent

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla

# Used to print coordinates of the spectator (the one flying in the simulator and observing the world) to the terminal
# Mainly used to assist in scenario designing to know exact coordinates of specific locations
def main():
    client = carla.Client('localhost', 2000)
    client.set_timeout(2.0)
    world = client.get_world()
    try:
        print("Started printing spectator's coordinates.")
        while(True):
            spectator = world.get_spectator().get_transform()
            coordinates_string = "{}, {}, {}  |  x, y, z".format('%.3f' % spectator.location.x, '%.3f' % spectator.location.y, '%.3f' % spectator.location.z)
            print(coordinates_string)
            time.sleep(1)
    except KeyboardInterrupt:
        print("Stopped printing coordinates.")
    except:
        print("Something wrong happened trying to print the spectator's coordinates.")


if __name__ == '__main__':
    main()
```

File: ./software/carla_scripts/helper_scripts/draw_lanes_terminal.py

```python
import glob
import os
import sys
import time
import random
import logging
import argparse
import numpy
import json
import base64
import pickle
sys.path.append("..")
from classes.route import Route
from config import *
from agents.navigation.global_route_planner import GlobalRoutePlanner

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla

argparser = argparse.ArgumentParser(
        description=__doc__)
argparser.add_argument(
    '-p', '--path',
    metavar='F',
    default="path1",
    help='Path filename (path1)')
args = argparser.parse_args()

# Reads file and extracts locations
def get_route(map):
    waypoints = []
    grp = GlobalRoutePlanner(map, SAMPLING_RESOLUTION)
    locations = read_path_file()

    for i in range(len(locations)-1):
        waypoints.extend(grp.trace_route(locations[i], locations[i+1]))
    return Route(waypoints)

def read_path_file():
    file_path = "../../../data/paths/{}.json".format(args.path)
    try:
        with open(file_path) as file:
            data = json.load(file)
            locations = []
            for loc in data["path_checkpoints"]:
                locations.append(carla.Location(x=float(loc["x"]), y=float(loc["y"]), z=float(loc["z"])))
            return locations
    except Exception as e:
        print("Could not read the file {}".format(args.path))


# Used to draw the specified path on the current map
def main():
    client = carla.Client("localhost", 2000)
    client.set_timeout(10)
    world = client.get_world()
    map = world.get_map()


    route = get_route(map)
    try:
        print("Started drawing {}.".format(args.path))
        finished = False
        all_waypoints = route.get_all_waypoints()
        start_color = carla.Color(r=0, g=0, b=255)
        finish_color = carla.Color(r=255, g=0, b=0)
        num_waypoints = len(all_waypoints)
        while(not finished):

            # Draw lanes that can be driven
            for i, waypoint in enumerate(all_waypoints):
                color_progress = float(i) / (num_waypoints - 1)

                r = (finish_color.r - start_color.r) / (num_waypoints - 1)
                g = (finish_color.g - start_color.g) / (num_waypoints - 1)
                b = (finish_color.b - start_color.b) / (num_waypoints - 1)
                color = carla.Color(
                    r = int(start_color.r + i * r),
                    g = int(start_color.g + i * g),
                    b = int(start_color.b + i * b)
                )
                world.debug.draw_string(waypoint.transform.location, 'o', draw_shadow=False,color=color, life_time=10,persistent_lines=True)

            time.sleep(9)
    except KeyboardInterrupt:
        print("Stopped drawing the path.")
    except Exception as e:
        print("Something wrong happened trying to draw the path. Aborting...", e)

if __name__ == '__main__':
    main()
```

File: ./software/carla_scripts/helper_scripts/save_opendrive_map.py

```python
#!/usr/bin/env python

# Copyright (c) 2019 Computer Vision Center (CVC) at the Universitat Autonoma de
# Barcelona (UAB).
#
# This work is licensed under the terms of the MIT license.
# For a copy, see <https://opensource.org/licenses/MIT>.

import glob
import os
import sys

import carla

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

# Used to save the current map in the simulation to an .xml file
def main():

    try:
        client = carla.Client('127.0.0.1', 2000)
        client.set_timeout(10)
        world = client.get_world()
        map = world.get_map()

        name = map.name.rsplit('/', 1)[-1]
        world.get_map().save_to_disk("maps/{}.xml".format(name))

    except Exception as e:
        print("Something wrong happened:", e)
    finally:
        print("Finished")


if __name__ == '__main__':
    main()
```

File: ./software/carla_scripts/helper_scripts/hero_info.py

```python
import glob
import os
import sys
import time
import random

from agents.navigation.behavior_agent import BehaviorAgent, BasicAgent

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla

# Used to print the bounding box of the "hero" vehicle
def main():
    client = carla.Client("localhost", 2000)
    client.set_timeout(10)
    world = client.get_world()
    try:
        # Get the player
        player = None
        actors = world.get_actors()
        for actor in actors:
            if actor.attributes.get('role_name') == 'hero':
                player = actor

        box = player.bounding_box.extent
        print("The bounding box (x, y, z):")
        print(box.x)
        print(box.y)
        print(str(box.z) + "\n")
    except Exception as e:
        print("Something wrong happened trying to get the bounding box of the actor named hero:", e)

if __name__ == '__main__':
    main()
```

File: ./software/carla_scripts/helper_scripts/draw_points_terminal.py

```python
import glob
import os
import sys
import time
import random
import logging
import argparse
import numpy
import json
import base64
import pickle
sys.path.append("..")
from classes.route import Route
from config import *
from agents.navigation.global_route_planner import GlobalRoutePlanner

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla

argparser = argparse.ArgumentParser(
        description=__doc__)
argparser.add_argument(
    '-p', '--path',
    metavar='F',
    default="path1",
    help='Path filename (path1)')
args = argparser.parse_args()

def read_path_file():
    file_path = "../../../data/paths/{}.json".format(args.path)
    try:
        with open(file_path) as file:
            data = json.load(file)
            locations = []
            for loc in data["path_checkpoints"]:
                locations.append(carla.Location(x=float(loc["x"]), y=float(loc["y"]), z=float(loc["z"])))
            return locations
    except Exception as e:
        print("Could not read the file {}".format(args.path))


# Used to draw the specified path on the current map
def main():
    client = carla.Client("localhost", 2000)
    client.set_timeout(10)
    world = client.get_world()
    map = world.get_map()

    locations = read_path_file()
    try:
        finished = False
        while(not finished):

            for l in locations[1:]:
                world.debug.draw_string(l, 'o000o', draw_shadow=False,color=carla.Color(r=0, g=0, b=255), life_time=10,persistent_lines=True)

            world.debug.draw_string(locations[0], 'o000o', draw_shadow=False,color=carla.Color(r=255, g=0, b=0), life_time=10,persistent_lines=True)
            time.sleep(5)
    except KeyboardInterrupt:
        print("Stopped drawing the path.")
    except Exception as e:
        print("Something wrong happened trying to draw the path. Aborting...", e)

if __name__ == '__main__':
    main()
```

File: ./software/carla_scripts/helper_scripts/draw_hero.py

```python
import glob
import os
import sys
import time
import random
import logging
import argparse
import numpy
import json
import base64
import pickle
sys.path.append("..")
from classes.route import Route
from config import *
from agents.navigation.global_route_planner import GlobalRoutePlanner

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla

argparser = argparse.ArgumentParser(
        description=__doc__)
argparser.add_argument(
    '-l', '--log',
    help='Log path')
args = argparser.parse_args()

logger = None
if args.log:
    logger = logging.getLogger("draw_hero-thread")
    logging.basicConfig(level=logging.INFO, filename=args.log, filemode="a", format="%(asctime)s - %(name)s - %(levelname)s - %(message)s")

# Used to draw the specified path on the current map
def main():
    client = carla.Client("localhost", 2000)
    client.set_timeout(10)
    world = client.get_world()

    try:
        message = "Begin marking the player's location in green"
        if logger is not None: logger.info(message)
        else: print(message)
        # Every step
        finished = False
        while(not finished):
            # Get the player
            player = None
            actors = world.get_actors()
            for actor in actors:
                if actor.attributes.get('role_name') == 'hero':
                    player = actor
            if player:
                world.debug.draw_string(player.get_location(), 'O', draw_shadow=False,color=carla.Color(r=0, g=255, b=0), life_time=0.2,persistent_lines=True)
            else:
                time.sleep(2)
            time.sleep(0.1)
    except KeyboardInterrupt:
        if logger is None: print("Stopped marking the player's location.")
    except Exception as e:
        message = "Something wrong happened trying to mark the player's location. Aborting..."
        if logger is not None: logger.error(message, e)
        else:  print(message)
    finally:
        message = "Stopped drawing lanes."
        if logger is not None: logger.info(message)
        else: print(message)

if __name__ == '__main__':
    main()
```

File: ./software/carla_scripts/helper_scripts/draw_lanes.py

```python
import glob
import os
import sys
import time
import random
import logging
import argparse
import numpy
import json
import base64
import pickle
sys.path.append("..")
from classes.route import Route
from config import *
from agents.navigation.global_route_planner import GlobalRoutePlanner

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla

argparser = argparse.ArgumentParser(
        description=__doc__)
argparser.add_argument(
    '-p', '--path',
    metavar='F',
    default="path1",
    help='Path filename (path1)')
argparser.add_argument(
    '-l', '--log',
    help='Log path')
argparser.add_argument("locations")
args = argparser.parse_args()

logger = None
if args.log:
    logger = logging.getLogger("draw_lanes-thread")
    logging.basicConfig(level=logging.INFO, filename=args.log, filemode="a", format="%(asctime)s - %(name)s - %(levelname)s - %(message)s")


def extract_locations(path_details):
    locations = []
    for loc in path_details:
        locations.append(carla.Location(x=float(loc["x"]), y=float(loc["y"]), z=float(loc["z"])))
    return locations

# Reads file and extracts locations
def get_route(map, details):
    waypoints = []
    grp = GlobalRoutePlanner(map, SAMPLING_RESOLUTION)
    if details is not None: locations = extract_locations(details)
    else: locations = read_path_file

    for i in range(len(locations)-1):
        waypoints.extend(grp.trace_route(locations[i], locations[i+1]))
    return Route(waypoints)

def read_path_file():
    file_path = "../../data/paths/{}.json".format(args.path)
    try:
        with open(file_path) as file:
            data = json.load(file)
            locations = []
            for loc in data["path_checkpoints"]:
                locations.append(carla.Location(x=float(loc["x"]), y=float(loc["y"]), z=float(loc["z"])))
            return locations
    except Exception as e:
        print("Could not read the file {}".format(args.path))


# Used to draw the specified path on the current map
def main():
    client = carla.Client("localhost", 2000)
    client.set_timeout(10)
    world = client.get_world()
    map = world.get_map()

    locations = None
    if args.locations:
        locs = args.locations
        locs_decoded = base64.b64decode(locs.encode())
        locations = pickle.loads(locs_decoded)
    route = get_route(map, locations)
    try:
        if logger is not None: logger.info("Started drawing the path")
        else: print("Started drawing {}.".format(args.path))
        # Every step
        finished = False
        while(not finished):
            all_waypoints = route.get_all_waypoints()

            # Draw lanes that can be driven
            for waypoint in all_waypoints:
                world.debug.draw_string(waypoint.transform.location, 'o', draw_shadow=False,color=carla.Color(r=0, g=0, b=255), life_time=10,persistent_lines=Tru

            # Draw finish line
            for waypoint in route.finish_lane_waypoints:
                world.debug.draw_string(waypoint.transform.location, 'o', draw_shadow=False,color=carla.Color(r=255, g=0, b=0), life_time=10,persistent_lines=Tru
            time.sleep(9)
    except KeyboardInterrupt:
        if logger is None: print("Stopped drawing the path.")
    except Exception as e:
        if logger is not None: logger.error("Something wrong happened trying to draw the path. Aborting...", e)
        else:  print("Something wrong happened trying to draw the path. Aborting...", e)
    finally:
        if logger is not None: logger.info("Stopped drawing lanes.")

if __name__ == '__main__':
    main()
```

File: ./software/python_libraries/requirements.txt

```
antlr4-python3-runtime==4.9.3
appnope==0.1.3
asgiref==3.5.2
asttokens==2.2.1
backcall==0.2.0
certifi==2022.12.7
charset-normalizer==3.0.1
click==8.1.3
```

```
commonroad-all==0.0.1
commonroad-drivability-checker==2022.2.1
commonroad-io==2022.3
commonroad-route-planner==2022.3
commonroad-scenario-designer==0.6.0
commonroad-vehicle-models==3.0.2
contourpy==1.0.7
cycler==0.11.0
decorator==5.1.1
Django==3.2
docker==6.0.1
enum34==1.1.10
executing==1.2.0
ffmpeg==1.4
fonttools==4.38.0
grpcio==1.51.1
idna==3.4
ipython==8.9.0
iso3166==2.1.1
jedi==0.18.2
joblib==1.2.0
kiwisolver==1.4.4
libsumo==1.15.0
lxml==4.9.2
matplotlib==3.6.3
matplotlib-inline==0.1.6
mercantile==1.2.1
networkx==3.0
numpy==1.24.1
omegaconf==2.3.0
opendrive2lanelet==1.2.1
ordered-set==4.1.0
packaging==23.0
parso==0.8.3
pexpect==4.8.0
pickleshare==0.7.5
Pillow==9.4.0
prompt-toolkit==3.0.36
protobuf==3.20.1
ptyprocess==0.7.0
pure-eval==0.2.2
Pygments==2.14.0
pyparsing==3.0.9
pyproj==3.4.1
PyQt5==5.15.8
PyQt5-Qt5==5.15.2
PyQt5-sip==12.11.1
python-dateutil==2.8.2
pytz==2022.5
PyYAML==6.0
requests==2.28.2
Rtree==1.0.1
scikit-learn==1.2.0
scipy==1.10.0
Shapely==1.8.5
six==1.16.0
sqlparse==0.4.3
stack-data==0.6.2
sumocr==2023.1
sumolib==1.15.0
threadpoolctl==3.1.0
tqdm==4.64.1
traci==1.15.0
traitlets==5.9.0
urllib3==1.26.14
utm==0.7.0
wcwidth==0.2.6
websocket-client==1.5.0
```

File: ./software/python_libraries/README.md

```
## Introduction
This directory contains the `crdesigner` python library and the requirements.txt file containing a list of python libraries needed to work with this framework.

`crdesigner` is a modified version of the `crdesigner` library (version 0.6.0). This version was the most recent release available at the time of the research.
More about crdesigner can be found [here](https://gitlab.lrz.de/tum-cps/commonroad-scenario-designer).

## Modification
The original `commonroad-scenario-designer` library was modified because it could not read input map files correctly and could not draw a list of specific points c

## Requirements
The `requirements.txt` file lists the dependencies required to use this project.
These dependencies can be installed by creating a virtual environment at the root of this project and running the command
``pip install -r requirements.txt``
```

File: ./software/README.md

```
# Software directory

This directory is part of the CS Bachelor's thesis on the topic "Comparing human and AI behavior in simulated driving scenarios". It contains all the software impl

## Technicalities

The software is working with [CARLA](https://carla.org) driving simulator.

- CARLA Version 0.9.13
- CARLA API Version 0.9.13
- OS: Ubuntu 18.04 and 20.04
- Python 3.6.9

All the required python libraries are listed [here](./python_libraries/requirements.txt).

## Structure

The following is the structure of this software bundle. More details can be found in the respective directories.

- [CARLA scripts](./carla_scripts/) - contains scripts for scenario processing and simulations.
- [Data analytics tools](./data_analysis/) - contains tools for parsing and analysing recorded data.
- [Python libraries](./python_libraries/) - required libraries for the software.
- [Scenario generation tool](./generating_scenarios/) - machine learning algorithm able to estimate the parameters of new scernarios.
- [Testing](./testing/) - directory containing various tests of software components.
```

File: ./software/testing/README.md

```
The approach to testing...
```

File: ./software/generating_scenarios/README.md

```
## General information

This directory contains scripts needed to generate new driving scenarios.

The data used in learning is stored in the [learning_data](../../data/scenario_generation_data/learning_data/) directory.

Newly generated scenarios go to the [generated_scenarios](../../data/scenario_generation_data/generated_scenarios/) directory.

The `regressor` file contains the algorithm used to train the model and estimate new scenarios.

To create a new scenario, run the `create_scenario.py` file giving flags `-d` specifying the difficulty of the scenario, `-s` specifying the sun's altitude, `-w` s

Example: `python create_scenario.py -d 200 -w 20 -s 30 -f new_scenario`
```

File: ./software/generating_scenarios/regressor.py

```python
import pandas as pd
import numpy as np
from sklearn.multioutput import MultiOutputRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
import json

SCENARIO_GENERATION_DATA_PATH = "../../data/scenario_generation_data"

# This class is responsible for initialising the MultiOutputRegressor algorithm which
# is trained on the data contained in scenarios.csv file and is able to estimate new
# scenario parameters given difficulty
class Regressor():

    def __init__(self, randomise):
        # First read the .csv file containing scenario entries
        scenarios_dataframe = pd.read_csv("{}/learning_data/scenarios.csv".format(SCENARIO_GENERATION_DATA_PATH), delimiter = ",")
        random_state = 10 if randomise else None
        # Create a new dataset with only the difficulty as a feature and the rest of the attributes as targets
        X = scenarios_dataframe[["difficulty", "wetness", "sun_altitude_angle"]]
        y = scenarios_dataframe.drop(["difficulty", "wetness", "sun_altitude_angle"], axis=1)
        self.attribute_names = y.columns
        # Split the data into training and testing sets using the standard 80-20 approach
        self.X_train, self.X_test, self.y_train, self.y_test = train_test_split(X, y, test_size=0.1, random_state=random_state)
        # Train the multi-target regression model
        self.model = MultiOutputRegressor(RandomForestRegressor(n_estimators=100, random_state=random_state))
        self.model.fit(self.X_train.values, self.y_train)

    # This method will create a new scenario given the difficulty and then save it to the file
    def create_new_scenario(self, difficulty, wetness, sun, filename):
        attributes = self.predict_scenario_attributes(difficulty, wetness, sun)
        self.write_scenario_to_file(attributes, filename)

    # Given the difficulty, predict what attributes a scenario could have
    def predict_scenario_attributes(self, difficulty, wetness, sun):
        arr = np.array([difficulty, wetness, sun]).reshape(1, -1)
        attributes = self.model.predict(arr)[0]
        rounding = pd.read_csv("{}/learning_data/rounding.csv".format(SCENARIO_GENERATION_DATA_PATH))
        maximums = pd.read_csv("{}/learning_data/maximums.csv".format(SCENARIO_GENERATION_DATA_PATH))
        minimums = pd.read_csv("{}/learning_data/minimums.csv".format(SCENARIO_GENERATION_DATA_PATH))
        rounded_attributes = []
        # For each predicted value, check if it is not exceeding the maximum allowed value and is not lower than the mimum allowed value
        # Also round the final value accordingly
        list_of_attribute_names = self.attribute_names.tolist()
        for i in range(len(attributes)):
            value = self.format_attrbute_value(value = float(attributes[i]), minimum = float(minimums.loc[0, list_of_attribute_names[i]]), maximum = maximums.loc[
            rounded_attributes.append(value)

        # Also add difficulty, wetness and sun_altitude_angle to the final scenario file
        dictionary = dict(zip(self.attribute_names, rounded_attributes))
        dictionary["difficulty"] = self.format_attrbute_value(value = float(difficulty), minimum = float(minimums.loc[0, "difficulty"]), maximum = maximums.loc[0,
        dictionary["wetness"] = self.format_attrbute_value(value = float(wetness), minimum = float(minimums.loc[0, "wetness"]), maximum = maximums.loc[0, "wetness
        dictionary["sun_altitude_angle"] = self.format_attrbute_value(value = float(sun), minimum = float(minimums.loc[0, "sun_altitude_angle"]), maximum = maximu
        return dictionary

    def format_attrbute_value(self, value, minimum, maximum, roundn):
        if value < minimum:
            value = minimum
        if maximum == "inf":
            pass
        else:
            if value > float(maximum):
                value = float(maximum)
        value = int(np.round(value)) if roundn == 0 else np.round(value, roundn)
        return value

    # Create a .json file and save the attribute values there
    def write_scenario_to_file(self, attributes, filename):
        with open("{}/generated_scenarios/{}.json".format(SCENARIO_GENERATION_DATA_PATH, filename), "w+") as f:
            json.dump(attributes, f, indent=4)
```

File: ./software/generating_scenarios/create_scenario.py

```python
from regressor import Regressor
import argparse

# The purpose of this script is to create new scenario.json file according to the difficulty, wetness amd sun altitude angle
# and save it in the file specified in the arguments
def main():
    parser = argparse.ArgumentParser(description='Create a new scenario based on difficulty.')
    parser.add_argument('--difficulty', '-d', type=float, default='70', help='Difficulty value')
    parser.add_argument('--sun', '-s', type=float, default='68', help='Sun altitude angle')
    parser.add_argument('--wetness', '-w', type=float, default='10', help='Wetness value')
    parser.add_argument('--filename', '-f', type=str, default='example', help='Filename to save the scenario')
    parser.add_argument('--random', '-r', action='store_false', help='Everytime build and train the algortithm differently')
    args = parser.parse_args()

    try:
        regressor = Regressor(args.random)
        regressor.create_new_scenario(args.difficulty, args.wetness, args.sun, args.filename)
        print("New scenario was created and save into {}.json file.".format(args.filename))
    except Exception as e:
        print("An error occurred trying to create a new scenario:", str(e))

if __name__ == '__main__':
    main()
```

File: ./README.md

```markdown
# Testing vehicle safety in simulated driving scenarios

This repository holds data and code related to the CS Bachelor's thesis.

- [Forms](./forms/)
- [Software bundle](./software)
- [Data storage](./data)

---

I verify that I am the sole author of this project, except where explicitly stated to the contrary.

**Author:**

> Vakaris Paulaviius

**Date:**

> 2023-04-03
```

File: ./.gitignore

```
# Created by https://www.toptal.com/developers/gitignore/api/windows,linux,macos,intellij,pycharm,visualstudiocode,python
# Edit at https://www.toptal.com/developers/gitignore?templates=windows,linux,macos,intellij,pycharm,visualstudiocode,python

### Intellij ###
# Covers JetBrains IDEs: IntelliJ, RubyMine, PhpStorm, AppCode, PyCharm, CLion, Android Studio, WebStorm and Rider
# Reference: https://intellij-support.jetbrains.com/hc/en-us/articles/206544839

# User-specific stuff
.idea/**/workspace.xml
.idea/**/tasks.xml
.idea/**/usage.statistics.xml
.idea/**/dictionaries
.idea/**/shelf

# AWS User-specific
.idea/**/aws.xml

# Generated files
.idea/**/contentModel.xml

# Sensitive or high-churn files
.idea/**/dataSources/
.idea/**/dataSources.ids
.idea/**/dataSources.local.xml
.idea/**/sqlDataSources.xml
.idea/**/dynamic.xml
.idea/**/uiDesigner.xml
.idea/**/dbnavigator.xml

# Gradle
.idea/**/gradle.xml
.idea/**/libraries

# Gradle and Maven with auto-import
# When using Gradle or Maven with auto-import, you should exclude module files,
# since they will be recreated, and may cause churn. Uncomment if using
# auto-import.
# .idea/artifacts
# .idea/compiler.xml
# .idea/jarRepositories.xml
# .idea/modules.xml
# .idea/*.iml
# .idea/modules
# *.iml
# *.ipr

# CMake
cmake-build-*/

# Mongo Explorer plugin
.idea/**/mongoSettings.xml

# File-based project format
*.iws

# IntelliJ
out/

# mpeltonen/sbt-idea plugin
.idea_modules/
```

```
# JIRA plugin
atlassian-ide-plugin.xml

# Cursive Clojure plugin
.idea/replstate.xml

# SonarLint plugin
.idea/sonarlint/

# Crashlytics plugin (for Android Studio and IntelliJ)
com_crashlytics_export_strings.xml
crashlytics.properties
crashlytics-build.properties
fabric.properties

# Editor-based Rest Client
.idea/httpRequests

# Android studio 3.1+ serialized cache file
.idea/caches/build_file_checksums.ser

### Intellij Patch ###
# Comment Reason: https://github.com/joeblau/gitignore.io/issues/186#issuecomment-215987721

# *.iml
# modules.xml
# .idea/misc.xml
# *.ipr

# Sonarlint plugin
# https://plugins.jetbrains.com/plugin/7973-sonarlint
.idea/**/sonarlint/

# SonarQube Plugin
# https://plugins.jetbrains.com/plugin/7238-sonarqube-community-plugin
.idea/**/sonarIssues.xml

# Markdown Navigator plugin
# https://plugins.jetbrains.com/plugin/7896-markdown-navigator-enhanced
.idea/**/markdown-navigator.xml
.idea/**/markdown-navigator-enh.xml
.idea/**/markdown-navigator/

# Cache file creation bug
# See https://youtrack.jetbrains.com/issue/JBR-2257
.idea/$CACHE_FILE$

# CodeStream plugin
# https://plugins.jetbrains.com/plugin/12206-codestream
.idea/codestream.xml

# Azure Toolkit for IntelliJ plugin
# https://plugins.jetbrains.com/plugin/8053-azure-toolkit-for-intellij
.idea/**/azureSettings.xml

### Linux ###
*~

# temporary files which can be created if a process still has a handle open of a deleted file
.fuse_hidden*

# KDE directory preferences
.directory

# Linux trash folder which might appear on any partition or disk
.Trash-*

# .nfs files are created when an open file is removed but is still being accessed
.nfs*

### macOS ###
# General
.DS_Store
.AppleDouble
.LSOverride

# Icon must end with two \r
Icon


# Thumbnails
._*

# Files that might appear in the root of a volume
.DocumentRevisions-V100
.fseventsd
.Spotlight-V100
.TemporaryItems
.Trashes
.VolumeIcon.icns
.com.apple.timemachine.donotpresent

# Directories potentially created on remote AFP share
.AppleDB
.AppleDesktop
Network Trash Folder
Temporary Items
.apdisk

### macOS Patch ###
# iCloud generated files
*.icloud

### PyCharm ###
# Covers JetBrains IDEs: IntelliJ, RubyMine, PhpStorm, AppCode, PyCharm, CLion, Android Studio, WebStorm and Rider
# Reference: https://intellij-support.jetbrains.com/hc/en-us/articles/206544839

# User-specific stuff
```

```
# AWS User-specific

# Generated files

# Sensitive or high-churn files

# Gradle

# Gradle and Maven with auto-import
# When using Gradle or Maven with auto-import, you should exclude module files,
# since they will be recreated, and may cause churn. Uncomment if using
# auto-import.
# .idea/artifacts
# .idea/compiler.xml
# .idea/jarRepositories.xml
# .idea/modules.xml
# .idea/*.iml
# .idea/modules
# *.iml
# *.ipr

# CMake

# Mongo Explorer plugin

# File-based project format

# IntelliJ

# mpeltonen/sbt-idea plugin

# JIRA plugin

# Cursive Clojure plugin

# SonarLint plugin

# Crashlytics plugin (for Android Studio and IntelliJ)

# Editor-based Rest Client

# Android studio 3.1+ serialized cache file

### PyCharm Patch ###
# Comment Reason: https://github.com/joeblau/gitignore.io/issues/186#issuecomment-215987721

# *.iml
# modules.xml
# .idea/misc.xml
# *.ipr

# Sonarlint plugin
# https://plugins.jetbrains.com/plugin/7973-sonarlint

# SonarQube Plugin
# https://plugins.jetbrains.com/plugin/7238-sonarqube-community-plugin

# Markdown Navigator plugin
# https://plugins.jetbrains.com/plugin/7896-markdown-navigator-enhanced

# Cache file creation bug
# See https://youtrack.jetbrains.com/issue/JBR-2257

# CodeStream plugin
# https://plugins.jetbrains.com/plugin/12206-codestream

# Azure Toolkit for IntelliJ plugin
# https://plugins.jetbrains.com/plugin/8053-azure-toolkit-for-intellij

### Python ###
# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class

# C extensions
*.so

# Distribution / packaging
.Python
build/
develop-eggs/
dist/
downloads/
eggs/
.eggs/
lib/
lib64/
parts/
sdist/
var/
wheels/
share/python-wheels/
*.egg-info/
.installed.cfg
*.egg
MANIFEST

# PyInstaller
# Usually these files are written by a python script from a template
# before PyInstaller builds the exe, so as to inject date/other infos into it.
*.manifest
*.spec

# Installer logs
pip-log.txt
pip-delete-this-directory.txt
```

```
# Unit test / coverage reports
htmlcov/
.tox/
.nox/
.coverage
.coverage.*
.cache
nosetests.xml
coverage.xml
*.cover
*.py,cover
.hypothesis/
.pytest_cache/
cover/

# Translations
*.mo
*.pot

# Django stuff:
*.log
local_settings.py
db.sqlite3
db.sqlite3-journal

# Flask stuff:
instance/
.webassets-cache

# Scrapy stuff:
.scrapy

# Sphinx documentation
docs/_build/

# PyBuilder
.pybuilder/
target/

# Jupyter Notebook
.ipynb_checkpoints

# IPython
profile_default/
ipython_config.py

# pyenv
# For a library or package, you might want to ignore these files since the code is
# intended to run in multiple environments; otherwise, check them in:
# .python-version

# pipenv
# According to pypa/pipenv#598, it is recommended to include Pipfile.lock in version control.
# However, in case of collaboration, if having platform-specific dependencies or dependencies
# having no cross-platform support, pipenv may install dependencies that don't work, or not
# install all needed dependencies.
#Pipfile.lock

# poetry
# Similar to Pipfile.lock, it is generally recommended to include poetry.lock in version control.
# This is especially recommended for binary packages to ensure reproducibility, and is more
# commonly ignored for libraries.
# https://python-poetry.org/docs/basic-usage/#commit-your-poetrylock-file-to-version-control
#poetry.lock

# pdm
# Similar to Pipfile.lock, it is generally recommended to include pdm.lock in version control.
#pdm.lock
# pdm stores project-wide configurations in .pdm.toml, but it is recommended to not include it
# in version control.
# https://pdm.fming.dev/#use-with-ide
.pdm.toml

# PEP 582; used by e.g. github.com/David-OConnor/pyflow and github.com/pdm-project/pdm
__pypackages__/

# Celery stuff
celerybeat-schedule
celerybeat.pid

# SageMath parsed files
*.sage.py

# Environments
.env
.venv
env/
venv/
ENV/
env.bak/
venv.bak/

# Spyder project settings
.spyderproject
.spyproject

# Rope project settings
.ropeproject

# mkdocs documentation
/site

# mypy
.mypy_cache/
.dmypy.json
dmypy.json
```

```
# Pyre type checker
.pyre/

# pytype static type analyzer
.pytype/

# Cython debug symbols
cython_debug/

# PyCharm
# JetBrains specific template is maintained in a separate JetBrains.gitignore that can
# be found at https://github.com/github/gitignore/blob/main/Global/JetBrains.gitignore
# and can be added to the global gitignore or merged into this file. For a more nuclear
# option (not recommended) you can uncomment the following to ignore the entire idea folder.
#.idea/

### Python Patch ###
# Poetry local configuration file - https://python-poetry.org/docs/configuration/#local-configuration
poetry.toml

# ruff
.ruff_cache/

# LSP config files
pyrightconfig.json

### VisualStudioCode ###
.vscode/*
!.vscode/settings.json
!.vscode/tasks.json
!.vscode/launch.json
!.vscode/extensions.json
!.vscode/*.code-snippets

# Local History for Visual Studio Code
.history/

# Built Visual Studio Code Extensions
*.vsix

### VisualStudioCode Patch ###
# Ignore all local history of files
.history
.ionide

### Windows ###
# Windows thumbnail cache files
Thumbs.db
Thumbs.db:encryptable
ehthumbs.db
ehthumbs_vista.db

# Dump file
*.stackdump

# Folder config file
[Dd]esktop.ini

# Recycle Bin used on file shares
$RECYCLE.BIN/

# Windows Installer files
*.cab
*.msi
*.msix
*.msm
*.msp

# Windows shortcuts
*.lnk

# End of https://www.toptal.com/developers/gitignore/api/windows,linux,macos,intellij,pycharm,visualstudiocode,python
```

File: ./.code2pdf

```
:directories:
- .git
- data
- forms
- software/python_libraries/crdesigner

:files:
- .DS_Store
- __init__.py
- analysis_parameters.json
- scenario_list.json
```