POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Virtual vehicle sensor's set-up and synchronous logging from CARLA Simulator

Supervisors

Prof. Massimo VIOLANTE

Dott. Alessandro TESSUTI

Dott. Simone MARAGLIULO

Candidate

Federico STELLA

APRIL 2025

Abstract

Improving road safety is a global priority, with the World Health Organization identifying human error as the main cause of accidents. In response to this, the Zero Vision project has been launched with the ambition of completely eliminating road deaths by the year 2050 through the mandatory introduction of Advanced Driver Assistance Systems (ADAS) technologies and the development of autonomous driving systems. Before being deployed in vehicles, these systems must undergo rigid testing and validation phases to ensure their effectiveness and safety among the variety of conditions under which they may operate. This process requires the collection of huge amounts of data, even for complex driving scenarios. As such collection is costly and risky when dealing with borderline situations, the work proposed in this thesis aims to develop a fully customizable virtual environment for the generation of synthetic data which can be used to integrate real-world data in the validation phase, enabling the safe and controlled reproduction of the most difficult scenarios to be captured on the road. The platform allows complete customization of every aspect of the simulation, including weather, traffic conditions and detailed configuration of each virtual sensor installed on the simulated vehicle. In addition, the data collected will be organized according to one of the most widely used automotive standards, ensuring compatibility with validation and development processes already adopted in the industry. Although synthetic data can not completely replace real data collected on the road, through this work, an effective methodology is proposed to accelerate the validation of ADAS systems, contributing to the progress towards autonomous driving.

Table of Contents

List of Tables

List of Figures

Acronyms

ACC
Adaptive Cruise Control

AD
Autonomous Drive

ADAS
Advanced Driver Assistance Systems

ADDW
Advanced Driver Distraction Warning

AI
Artificial Intelligence

API
Application Programming Interface

AV
Autonomous Vehicles

CARLA
Car Learning to Act

CPU
Central Processing Unit

FPS
Frames per second

GNSS

Global Navigation Satellite Systems

GPS

Global Positioning System

GSR

General Safety Regulation

GPU

Graphics Processing Unit

IMU

Inertial Measurement Unit

LiDAR

Light Detection And Ranging

LKA

Lane Keeping Assistant

NHTSA

National Highway Traffic Safety Administration

RADAR

Radio Detection And Ranging

WHO

World Health Organization

Chapter 1

Introduction

The automotive industry is experiencing a paradigm shift with the rapid development of Advanced Driver Assistance Systems (ADAS) and Autonomous Vehicles (AV). As these technologies evolve, the demand for extensive and high-quality data to train and validate algorithms that underlie their functionality has grown exponentially.
Traditional data collection methods, which are based on vehicle sensorization, present significant challenges, including high costs, logistic complexities, and the time-consuming nature of collecting real-world data across different driving conditions. In response to these challenges, a data-driven approach has emerged as a viable solution. By leveraging the power of high-fidelity digital twins, the industry is now able to simulate real-world environments with unprecedented accuracy. These digital replicas of physical assets and environments provide a reliable and cost-effective alternative to physical sensor data, offering the ability to generate huge amounts of simulation-based data under a wide range of scenarios and conditions.

This thesis explores the pivotal role of digital twins in the development of ADAS/AV, providing an instrument to improve the performance, safety, and reliability of traditional data collection methodologies for ADAS validation.

1.1 Thesis genesis

As expressed in the World Health Organization (WHO) Global Status Report on Road Safety 2018 [1] the number of road traffic deaths reached 1.35 million in 2016, making road traffic injuries the leading cause of death for people aged 5 - 29 years. According to these studies, human errors are estimated to be involved in about 95% of road accidents causing, in addition to deaths, tens of millions of non-fatal injuries every year. As the Head of the WHO office in Montenegro said, *"Road traffic injuries are not just 'accidents'. They have risk factors, predictors and*

determinants and are therefore preventable"[2]. Road safety is thus seen as a shared responsibility in which urgent action is needed, with the aim of the Vision Zero project, which prevents zero causalities by 2050. An important step in this direction is the introduction of technologies on vehicles that can intervene in potentially risky situations.
Due to the General Safety Regulation (GSR [3]), the demand for ADAS in vehicles is expected to increase significantly in the coming years. The GSR introduces stricter safety standards aimed to reduce road accidents and to protect drivers, passengers, and pedestrians. As a result, automakers are required to integrate more ADAS technologies.

The GSR will be implemented in three phases from July 2022 to January 2029, as shown in Figure 1.1:

1st phase of implementation (A/B)		**2nd phase of implementation (C)**		**3rd phase of implementation (D)**	
✓ *Intelligent speed assistance (ISA)**		✓ *Advanced emergency braking for pedestrians and cyclists (cars and vans)*		✓ *Direct vision requirements (trucks and buses)*	
✓ *Emergency lane keeping (cars and vans)*		✓ *Advanced driver distraction warning*		✓ *Event data recorder (trucks and buses)**	
✓ *Advanced emergency braking for stationary/moving vehicles (cars and vans)*		✓ *Enlarged head impact zone (cars and vans)*		*Pedestrian protection for small series:*	
✓ *Event data recorder (cars and vans)**		✓ *Tyres in worn condition*		➢ *mid-2028 (new types)*	
✓ *Driver drowsiness and attention warning**		✓ *Event data recorder (for automated vehicles)*		➢ *mid-2034 (new vehicles)*	
✓ *Alcohol interlock installation facilitation**		✓ *Driver availability monitoring (for automated vehicles)*			
✓ *Emergency stop signal**		✓ *Platooning (for automated trucks)*			
✓ *Reversing detection**					
✓ *Blind spot information system (trucks and buses)*					
✓ *Pedestrians and cyclists collision warning (trucks and buses)*					
✓ *Tyre pressure monitoring system (vans, trucks and buses)*					
✓ *Cybersecurity & software update*					
new types	**new vehicles/parts**	**new types**	**new vehicles/tyres**	**new types**	**new vehicles**
6 July 2022	**7 July 2024**	**7 July 2024**	**7 July 2026**	**7 Jan 2026**	**7 Jan 2029**
Supplementary legislation to be adopted by: **6 April 2021**		**7 April 2023**		**7 September 2024**	
* Detailed technical requirements to be set out in Delegated Acts.					

Figure 1.1: Implementation phases of General Safety Regulation [4]

Each of these systems must undergo validation before being integrated into a vehicle. The testing and validation process requires a large amount of data to accurately represent a wide range of possible driving scenarios. These data are essential for properly training the machine learning algorithms that power the ADAS functionalities, ensuring their reliability and effectiveness under real world conditions.
This thesis work focuses on the development of a completely customizable virtual environment designed for the generation of synthetic data adapted for autonomous

driving systems. These scenarios are intended to reproduce a wide range of driving situations, traffic and environmental conditions that can be accurately controlled and manipulated to simulate real-world driving experiences. The synthetic data generated within this environment are then properly formatted to comply with the most widely accepted and standardized datasets used in the validation and testing sector for autonomous driving systems. The final objective is to obtain data that can be integrated into existing workflows, enabling comprehensive testing and validation of autonomous systems under diverse and highly realistic conditions.

1.2 ADAS of interest

This thesis project focuses on ADAS systems that are capable of monitoring and collecting information about the external environment in which the vehicle operates. These systems play a critical role in improving safety by continuously detecting and mitigating potential hazards in real time. This objective is achieved by evaluating the presence of pedestrians, other vehicles, cyclists and obstacles, reducing risks to both the driver and other road users, preventing accidents before they occur. For this reason, in the simulation environment will be collected only data related to external interaction of the vehicle with the surrounding, neglecting all the in-cabin information about the driver condition.

The sensors that will be implemented are briefly discussed here and then explored in depth in later sections:

- **RGB Camera:** it views the world within the visible spectrum and gives highly detailed visual information about the environment. It finds a wide range of applications in computer vision tasks such as object detection, scene recognition, and semantic
- **LiDAR:** is a technology that measures the range to an object by sending out laser pulses and looking at the reflected light. The resulting high-precision 3D point clouds it produces are very useful for mapping, localization, and obstacle detection in complicated environments.
- **Radar:** detects the velocity and range of objects in the most adverse weather conditions, such as rain, fog, or dust, using radio waves. The applications of radar are unlimited in tracking moving objects, especially vehicles and pedestrians.
- **Global Navigation Satellite System:** it is based on satellite signals and provides data on positioning characterized by accurate geolocation and navigation information. It plays a very key role in autonomous systems, including some applications such as global positioning and route planning.

- **Inertial Measurement Unit:** it provides information on vehicle acceleration, rotation, and orientation. It usually includes accelerometers and gyroscopes and is quite crucial for the estimation of motion and stability of the vehicle, especially when GPS is not available.

1.3 Contribution: Reply - Concept Quality

The research and developing processes presented in this thesis work was carried out as part of an internship at the Reply - Concept Quality company, which provided all the necessary tools for the development of the platform and the achievement of the set goals. The support and experience of its team of experts, allowed the project challenges to be addressed with a structured approach. The collaborative work environment fostered continuous learning, encouraging the adoption of best practices and the exploration of different methodologies to enhance the simulation framework.

In addition, the company collaborated on the development of a custom front-end tool created specifically to meet the implementation requirements that emerged during project development. This tool played a key role in facilitating interaction with the simulation environment by providing the ability to customize various aspects of the simulation in a simple and intuitive manner.

Chapter 2

Autonomous Drive: A New Era of Mobility

The continuing evolution of automotive technology aims to the objective of completely autonomous vehicles, where the driver becomes in effect a passenger in the vehicle, without the need of any manual intervention during its journey, leading to several benefits in terms of road safety, traffic management, fuel efficiency, and greater accessibility to mobility.

Although ADAS are not still Autonomous Driving (AD) systems, they play a crucial role in preparing vehicles for full autonomy. Beyond the direct safety benefits, ADAS serves as a critical testing ground for core autonomous technologies such as advanced sensors, machine learning algorithms and vehicle-to-everything (V2X) communication. In the deployment of ADAS features, automakers and researchers get valuable data to further improve perception, decision-making, and control systems. This iterative development process will help to build not only the transition from partial to full autonomy, but also the consumer trust in automation. By training drivers to operate on a car with ADAS features, they will be more prepared to use a vehicle with full autonomy, making ADAS an essential stepping stone in the evolution of autonomous vehicles.

2.1 Levels of Automation

The Society of Autonomous Engineers (SAE) ha set 6 levels of Driving Automation, with the first three levels of support features (levels 0, 1 and 2) that require the driver to be involved and constantly monitoring the functionalities offered by the systems installed, and the last three levels (levels 3, 4 and 5) with real automated driving features.

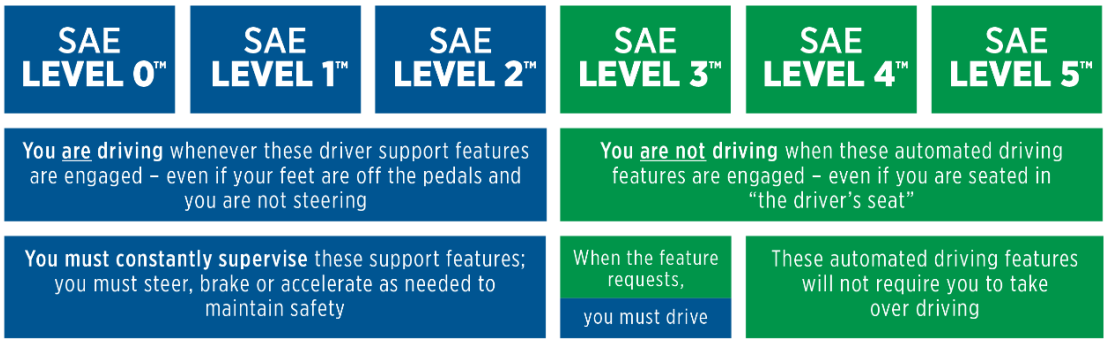

Figure 2.1: SAEJ3016 Levels of Driving Automation [5]

- **Level 0: No Driving Automation**
  The most basic level of autonomous driving, just a step up from traditional driving where the car is completely under the control of the driver. There are no autonomous systems implemented, but just some safety features such as backup cameras, collision warnings, blind spot warnings, and emergency breaking that must be constantly supervised and require the intervention of driver, who remains responsible for all aspects of driving.

- **Level 1: Driver Assistance**
  The car is able to operate autonomously in certain situations, but the driver is always required to be in control of the vehicle. The features of this level of automation can provide steering input to keeps the car in the middle of its lane and brake/acceleration to support the driver on maintaining a set distance from the car in front of it. Some common examples of Level I autonomous driving include adaptive cruise control (ACC) and Lane Keeping Assist (LKA). The vehicle is equipped with sensors aimed to detect objects around the car. If there is a vehicle or obstacle in the way, ACC will slow down or stop the car, and LKA will adjust the steering to keep the car in the lane.

- **Level 2: Partial Driving Automation**
  Characterized by the introduction of more advanced ADAS which can now control the longitudinal and lateral dynamics at the same time. For example,

lane keeping assist and ACC are combined in one system. However, the driver has still the responsibility for every behavior of the vehicle and must monitor the system at all times, touching the steering wheel regularly, and being able to intervene immediately if necessary.

- **Level 3: Conditional Driving Automation**
  Starting from this stage, the driver does not need to constantly monitor the systems while the vehicle temporarily takes over the driving task. Within certain limits other activities can be engaged by the human (e.g. read a message, text or drink a cup of coffee) while the car is in autonomous mode. Even on this level, if the system can no longer operate, the driver will be warned for a period and must be able to resume all aspects of the driving task.

- **Level 4: High Driving Automation**
  Under certain conditions (e.g. defined route, driving on the highway, etc.), the vehicle operates completely autonomously and does not require anymore the capability for the human driver to intervene. The vehicle is able to eventually reach a safe state in autonomous way if a malfunction on the systems equipped happens, without representing an hazard for the driver or for other road users. The driver however, has still the possibility to take control of the vehicle manually.

- **Level 5: Full Driving Automation**
  The autonomy of the vehicle is no longer subject to conditions. The car can operate completely autonomously anywhere in road traffic and under all conditions while the human can perform any kind of different activities without any risk. The vehicle is capable to perform a combination of several tasks simultaneously, such as navigating, changing lanes and avoiding obstacles. The human input is no longer required, so these vehicles are not anymore equipped with a steering wheel or a gas or brake pedal. The human driver becomes in effect a passenger

2.2 Cooperative Approach for AD

Beyond vehicles, a large cooperative ecosystem is needed to realize the full potential in the field of autonomous driving. This includes seamless interaction with external infrastructures, such as intelligent traffic signals, roadway sensors, and high definition mapping systems that provide critical information about the environment and enable situational awareness in real time. For example, integrated infrastructure warns the autonomous vehicle about possible hazards, changes in traffic patterns, or construction zones to maintain or improve its safety and efficiency of operation. Furthermore, safe operation of autonomous vehicles requires cooperation with other

road users, such as human drivers, cyclists, and pedestrians, in shared environments. This calls for systems designed to interpret and forecast human behaviors while properly communicating the vehicle's intentions. For example, an AV has to recognize and react to a pedestrian trying to cross the street or understand the action of a driver changing lanes in close proximity.

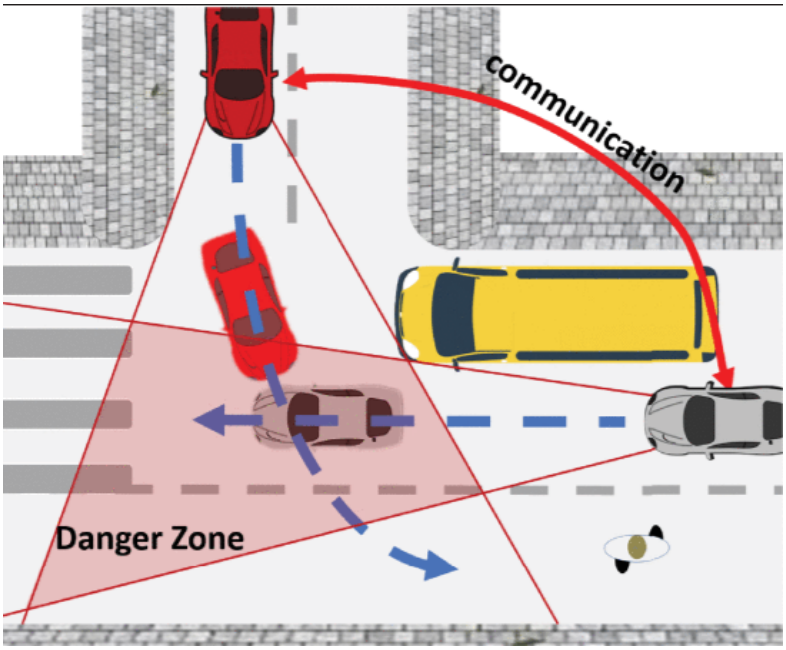

Figure 2.2: Vehicle coordination relies on tight interaction between control, communication, and sensing [6]

Different types of communication are then required to achieve the completely AD [6].

- **Infrastructure to Vehicle Communication (I2V):** AVs can transmit and receive information from static infrastructure devices in the environment. I2V communication involves not only monitoring vehicles but also other road users such as cyclists and pedestrians. Pedestrian monitoring is an essential element of AD in an urban environment. Many streets today are equipped with 24/7 surveillance cameras for traffic monitoring and security, and the big data from these cameras' networks provide beneficial data for AVs. However, today, most AV control relies on the vehicle's onboard sensors alone, yet fusion with additional external data will increase performance for pedestrian tracking and accident avoidance.
- **Communication Between AVs and Pedestrians (V2P):** The communication of the AVs with other road users (especially pedestrians) is a

fundamental step to achieve safe AD within the urban environment. AVs should be able to recognize pedestrian's actions and intentions with the help of enhanced sensors and AI technology while responding with suitable means such as visual displays, sounds or light patterns. For example, an AV moving towards a crosswalk could activate external lights to indicate that it has spotted a pedestrian and is giving way. The necessity of a common standard, or 'language', for this type of communication is still an argument of research.

- **Vehicle to Vehicle Communication (V2V) :** Because of its decentralized structure, this communication is more complex to carry out. V2V relies on the exchange of information among the vehicles located within a certain area. This turn in necessity of some communications technology and protocol. Issues that require focus include, but are not limited to, communication latency losses, incomplete readings, security and safety concerns. Be mindful that AVs communicating between themselves can greatly benefit traffic performance, but it does have risks as well concerned to cyber security.

2.3 Benefits

Although vehicles traceable to levels above three (Figure 2.1) are not yet completely developed and ready to enter the customer market, the benefits that these technologies can bring to everyday life are well foreseeable, as expressed by National Highway Traffic Safety Administration (NHTSA) studies [7].

- Safety: this is one of the automation's biggest benefits. Safety is not intended only for the passengers of the vehicle but also for every other road user such as pedestrian, bicyclists and so on. Higher levels of automation will remove completely the human error from the chain of events that can lead to a crash, achieving the Vision Zero project [1.1]

- Mobility: access to mobility is expanded to a larger category of users in autonomy such as for seniors or people with disabilities. The equity will be considered and addressed throughout the Autonomous Driving infrastructures and vehicle design processes.

- Economic: motor vehicle crashes cost billions each year. Eliminating the majority of vehicle crashes through technology these costs will be reduced significantly.

- Environmental: as the automotive industry continues to evolve toward further automation and electrification, both are promising ways in which safety and environmentally friendly practices can improve. Automation in vehicles will

likely alter the need for individualized parking spaces and lots, increasing the use of automated ride-share and shuttle fleets, which may change land use drastically. Moreover, electrification of vehicles could bring about opportunities to enhance efficiency with a decrease in personal driving, leading to further decreases in air pollutants emitted from the transportation sector.

2.4 Current Gaps

Several gaps must be addressed to advance higher levels of automation seen in Section 2.1. Starting from I2V communication, the development of advanced infrastructures is necessary to support the huge amount of data shared, together with the implementation of data fusion and analysis algorithms able to perform hazards identification, trajectories prediction and failures recognition. Moreover, including pedestrian in the communication network is a challenge that requires further development due to the possibility of receiving information only through small wearable devices. Furthermore, the presence of wireless communication can lead to vulnerabilities caused by the risk of cyber attacks, so reliable and secure data sharing must be applied. Finally, ethical considerations are still an argument of discussion: since every decision an autonomous vehicle makes has to be intentionally programmed, how should it respond in a no-win scenario? For example, in a crash situation where no matter what is done, there is a great likelihood of somebody being harmed, it can be programmed to prioritize the safety of the driver and passengers, pedestrians or other drivers.

Chapter 3

ADAS Validation

Validation of Advanced Driver Assistance Systems (ADAS) is a critical aspect in the automotive industry as it directly impacts the safety, reliability, and effectiveness of these technologies which are based on an intricate combination of sensors, cameras, radar and algorithms. Due to their complexity, a scientific validation framework is required to perform an accurate test in order to identify potential failure modes, improving system robustness before large-scale deployment.

The main points to be followed during validation and test phases are expressed below (Source [8]).

- **Safety Assurance:** the most important objective of ADAS validation is to ensure that the features such as automatic braking, lanekeeping, and collision avoidance will function safely and accurately within an increasing horizon of driving scenarios. This involves testing ADAS under variable conditions of traffic density, road type, and weather conditions to minimize accidents due to faulty systems, misinterpretation by sensors, or software bugs. By confirming a system's handling of edge cases and critical situations, ADAS validation prevents possibly lethal failures and increases overall vehicle safety.

- **Performance evaluation:** whatever the influence of external factors, ADAS must show a high degree of reliability and precision. Performance testing is an important part of validation since it will include a performance assessment of the systems by detailed testing that checks the precision of object detection, the interpretation of sensor information, and the execution of appropriate timely responses. This ensures that it functions optimally across all operating conditions and remains highly reliable throughout the vehicle's life.

- **Regulatory Compliance:** ADAS are subject to stringent safety standards and regulations of different national and international regulatory bodies. These ensure functional safety, reliability, and effectiveness in real-world conditions.

One of the most important regulations in this field is ISO 26262, which provides a way to guarantee functional safety and gives methods for estimating the risk associated with the usage of the E/E parts and then establish a set of procedures that allows to produce an output of the system that is functionally safe. Proper validation ensures that these tough safety requirements are met, helping to avoid possible legal matters, recalls, or penalties, and instills confidence in stakeholders and consumers.

- **User Experience:** a key part of the ADAS validation process involves the evaluation of usability and aspects of the human-machine interface. The goal is to ensure a smooth and intuitive driving experience, allowing users to interact with the system easily and without distraction. This process includes extensive testing to ensure that features are easily accessible, interfaces are clear and responsive, and system warnings are understandable and timely, thus helping to improve driver safety and confidence in using ADAS.

3.1 Types of testing

Traditionally, validation relies on several well-established methodologies whose choice depends on both the characteristics of the device under test and the particular phase of its development. In Figure 3.1 are reported the principal approaches, ordered according to the typical time sequence of the development process, under the name convention of X-In-The-Loop. As the terms already suggest, these methods rely on an iterative process, where the 'X' describes which component of the system is under test and validation. That process - from earlier design phases to actual vehicle testing - is crucial to ensure that the system operates reliably and efficiently [9].

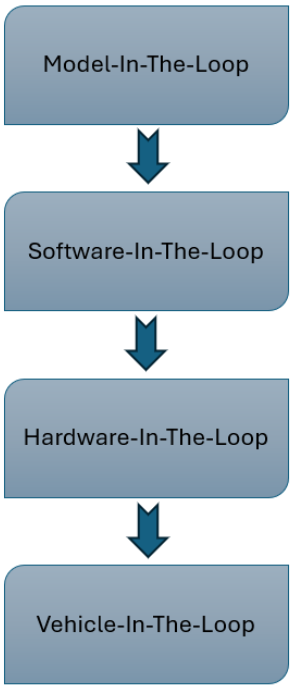

Figure 3.1: X-In-The-Loop approaches

- Model-In-The-Loop: it is used to optimize the behavior of the designed model and to verify that it is implementing all the requested functionality. Everything is running on the development PC that contains both controller and plant models, defined using the same specification language and they interacts according with the typical closed-loop chain.

- Software-In-The-Loop: the testing entity transitions from a block model to code and verifies that it is consistent with the specified model. It is co-simulated on the development PC with the model of the plant, giving an idea of whether the control logic can be converted to code and if it is hardware implementable.

- Hardware-In-The-Loop: the software is deployed in the real ECU and it is connected to other cooperating electronic devices that are still simulated in real time machines. Its possible to verify whether the software is still behaving as expected on the final hardware that is much less capable than the development PC.

- Vehicle-In-The-Loop: the developed device is deployed and tested directly in the target vehicle which, inside a laboratory, is stimulated with external inputs to check the whole behavior.

3.2 The necessity of Virtual Validation

As ADAS and AD evolves, traditional testing methods have proven to be insufficient to cover the full range of possible scenarios. The increasing complexity of these systems, which are characterized by the integration of advanced sensors, artificial intelligence and increasingly elaborate control algorithms, requires the adoption of more efficient and secure validation strategies. Within this context, virtual validation is a fundamental tool that supports traditional approaches in guaranteeing the system's reliability and safety before implementing it in an actual vehicle.
This approach uses high-fidelity simulation environments to measure the performance, reliability and robustness of ADAS functionality in a wide range of conditions that would be difficult, expensive and dangerous to be replicated in physical testing. Virtual validation enables comprehensive testing of system behavior prior to physical testing by simulating different driving scenarios such as complex urban environments, adverse weather conditions and rare but critical edge cases.

3.2.1 Bridging the Gap Between Virtual and Real

Although virtual validation offers many advantages, real-world testing remains a crucial stage in the verification process of ADAS and AD systems. [10]
Elements such as tire wear, slight sensor misalignments or the unpredictability of human behavior can represent difficult variables to be completely captured in a simulated environment. Road tests are therefore critical to ensure the reliability of systems subjected to real-world conditions.
An innovative approach to reduce the gap between real-world and virtual testing is **Sensor Stimulation**. This technique consists of injecting synthetic data directly into the sensors in order to reproduce the inputs that the vehicle would receive under real-world driving conditions but in a controlled environment, enabling highly realistic testing without the risks and limitations of road tests. With this approach it is possible to analyze system's response without the need to involve physical vehicles.
An example would be to check how the vehicle reacts if, inside a laboratory, its LiDAR sensor is stimulated with artificially generated data to simulate the presence of moving obstacles, such as a cyclist approaching the vehicle sideways.

3.3 Proposed Virtual Valdiation Approach

The project presented in this thesis focuses on the generation of synthetic data using the Driver-In-The-Loop approach. Through the use of a driving simulator, a virtual sensor-equipped vehicle can be controlled within a virtual environment, accurately replicating the variety of scenarios required for the validation phase. The data collected in this environment will then be formatted according to one of the most popular large-scale data sets for autonomous driving. This makes it possible to produce realistic synthetic data, which can then be injected into the devices to be tested, ensuring in-depth analysis of their performance under conditions that are difficult to replicate in the real world.
To achieve this goal, it was essential to identify two key elements:

1. A **simulation environment** capable of providing all the necessary components to realistically reproduce urban driving scenarios, ensuring accurate interactions between sensors and environmental elements.

2. A **structured data format** that allows the generated synthetic data to be seamlessly integrated with real-world datasets, minimizing the need for extensive adaptations and ensuring compatibility with existing validation pipelines.

Chapter 4

The Simulation Environment

To ensure the generation of realistic synthetic data, a crucial role is represented by the choice of the virtual environment in which to conduct the simulation. It must be able to accurately reproduce real driving conditions, including a variety of scenarios representative of the different situations that an ADAS system might face. This involves accurate modeling of elements such as road geometry, dynamic vehicle behavior, the presence of pedestrians and cyclists, the possibility of varying weather and lighting conditions. In addition, the simulator must support realistic interaction between vehicle sensors and the surrounding environment, allowing the system's response to different stresses to be accurately assessed. A well-designed virtual environment allows a wide range of tests to be covered, ensuring a more comprehensive and effective data generation, while reducing the need for complex and expensive on-road testing. For the purpose of this thesis, CARLA Simulator was chosen.

The official CARLA documentation (Source [11]) will be used as reference in the following sections to describe the main features used in the development of this project.

4.1 CARLA: Car Learning to Act

CARLA is an open-source simulator for autonomous urban driving [12]. Leveraging the potential of Unreal Engine 4 enables highly realistic and customizable virtual environments to support the training, prototyping and development of autonomous driving systems. The simulation platform provides a flexible configuration for each element of the offered sensor suite, giving the possibility to generate signals that are critical to driving strategy development, such as GPS coordinates, IMU data, LiDAR and RADAR information, comprehensive collision and infraction reports.

Moreover, many environmental parameters, such as time of day and weather conditions, can be configured inside the simulation, as illustrated in Figure 4.1.

CARLA is the optimal environment for the purpose of this thesis as it provides a rich and highly realistic platform for the simulation of complex urban scenarios, providing a number of fundamental elements for a realistic representation of the urban system such as pedestrians, which can be configured to move realistically by following random or predefined trajectories, road signs of various types that regulate traffic flow and so much more. The simulation also includes road markings that define lanes, pedestrian crossings, stop lines and everything that is essential for the proper functioning of autonomous vehicle perception systems. Traffic lights, which are also fully configurable, allow the testing of algorithms for detecting and interpreting illuminated signs, while intersection management allows the analysis of interaction between vehicles, pedestrians and road infrastructure. Finally, CARLA realistically simulates road dynamics, making it possible to reproduce traffic behavior in different scenarios with different levels of congestion and priority rules.

Figure 4.1: Some of the configurable elements subjected to different weather conditions [12]

4.1.1 Client-Server Architecture

CARLA is based on a Client-Server architecture in which the server plays the main role in managing the simulation, rendering scenes, processing data coming from sensors, computing physics and transmitting information to clients. On the other hand, the client consists of several modules that control the logic of the agents in the simulated environment. To implement this client-server architecture, CARLA makes use of API written in Python that allows communication through the use of sockets, ensuring dynamic and efficient interaction between the different elements of the simulation. The interaction takes place through the exchange of commands and meta-commands. Commands are used to directly control the main vehicle, while meta-commands are used to modify the behavior of the server, configure sensors and define the characteristics of the environment.

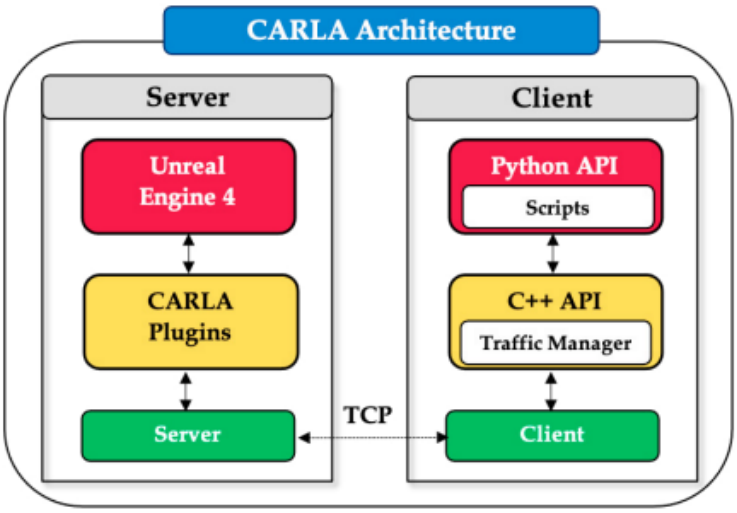

Figure 4.2: Client-Server communication in CARLA [13]

Communication can be implemented through two possible modalities [11]:

- Synchronous Mode: the client, through Python code, controls the flow of the simulation, determining when the server must update the state of the environment.
- Asynchronous Mode: server runs the simulation as fast as possible, handling client requests when possible.

The choice of the communication mode has a crucial role for guaranteeing the synchrony between the data generated from all the sensors involved in the simulation. Efficient management of information transmission is essential to ensure

the consistency and reliability of the acquired data.
This aspect will be explored in detail in Section 6.2, where all elements of the implemented simulation will be explained in depth, with a focus on sensor management, data synchronization, and environment setup.

4.1.2 Actors

An actor in CARLA is any element able to perform actions within the simulation and to interact with other elements in the environment. Actors can be dynamic, such as vehicles and pedestrians, or static, such as traffic lights and traffic signals. Actors are introduced to the simulation using blueprints, which are predefined models that contain animations, physical properties and a range of customizable attributes. Blueprints allow the user to modify specific parameters of each actor, such as vehicle type, color, driving behavior or pedestrian path, to match the simulation to the specific needs.

The actors managed through the API developed for this thesis are briefly introduced below.

- Vehicles: each vehicle that is generated in the CARLA world. A distinction can be made between:
  - Ego Vehicle: refers to the vehicle which is the focus of the simulation. It is possible to drive it manually via user controls. All the sensors necessary for data collection are attached to it.
  - Traffic Vehicles: all other vehicles in the simulation generated to populate the roads in order to simulate traffic conditions and the interaction of Ego vehicle with other vehicles. They can not be controlled manually, but their behavior is managed in autopilot mode by the Traffic Manager (4.1.3).
- Sensors: list of sensors attached to the Ego Vehicle which generates raw data. [11]
  - RGB Camera: «Provides clear vision of the surroundings. Looks like a normal photo of the scene».
  - LiDAR: «Generates a 4D point cloud with coordinates and intensity per point to model the surroundings».
  - RADAR: «2D point map modelling elements in sight and their movement regarding the sensor».
  - IMU: «Comprises an accelerometer, a gyroscope, and a compass».
  - GNSS: «Retrieves the geolocation of the sensor».

- Spectator: used to move the view of the simulator window. During all simulations performed it is placed inside the Ego Vehicle providing a driver point of view, just behind the steering wheel.
- Walkers: pedestrians generated to populate the town. They work in a similar way to vehicles, but do not have an autopilot mode. Their movement is handled by AI controllers that allow them to reach random points on the map.

Traffic signs and traffic lights are also actors that could be managed and modified through API. However, for the simulations performed during this thesis work they were not modified, maintaining the default characteristics of the reference maps used.

4.1.3 Traffic Manager

The Traffic Manager is a CARLA module designed to manage vehicles traffic within the simulation in an autonomous and realistic way. It makes possible to automatically control the behavior of vehicles not driven by the user, regulating aspects such as speed, trajectories and compliance with traffic rules. Thanks to the Traffic Manager, it is possible to simulate complex scenarios with several vehicles interacting with each other, respecting traffic lights, signals and priorities. In addition, the user can configure various parameters to customize the behavior of the vehicles, including safety distance, attitude to overtake, compliance with traffic rules.

None of these aspects were changed from the default values in the simulations performed.

Chapter 5

The Standard for Dataset

The data generated in the virtual environment introduced in Chapter 4, as widely discussed, are an essential resource for the validation process of the systems being developed. However, in order to be effectively used, they must be properly collected, organized and formatted according to the most widely accepted and standardized datasets used in the validation and testing sector for autonomous driving systems. This is crucial as it influences the compatibility of the synthetic data with other analysis tools and the capability to incorporate them into automated processing pipelines. Moreover, it makes easier to match the simulation results with real data, improving the virtual validation accuracy.
For this reason, the choice of format in which to store the synthetic data generated by simulations plays a central role in the development of this project. Several factors must be taken into account, such as the ability to faithfully represent the information collected, the ease of access and manipulation of data and the compatibility with software and tools already used to analyze and validate autonomous driving systems.

A nuScenes-compliant format will be used to store the synthetic data generated by the simulations performed in this thesis.

5.1 nuScenes

nuScenes is a large-scale public dataset developed by Motional to support research in autonomous driving and computer vision. [14]
The dataset consists of 1.000 driving scenes of 20 seconds each, collected in Boston and Singapore since they are known for being cities with dense traffic and challenging driving conditions. The scenes collected were then manually selected to include a wide range of different situations and traffic conditions, providing an highly representative dataset of real-world driving conditions. With accurate 3D bounding boxes at a frequency of 2 Hz across the entire dataset, nuScenes also provides

detailed annotations for 23 object classes, giving in addition information on object attributes, such as visibility, activity and pose, to improve the understanding of the road's context. In constrast to the other datasets which focus mainly on specific sensors (typically for image-based object recognition), a unique feature of nuScenes is that it aims to provide a more complete picture of autonomous perception by integrating data from the entire sensor suite of an autonomous vehicle.

5.1.1 Data collection

The data has been collected over a total of 15 hours of driving sessions, during which a variety of traffic scenarios were recorded. The most significant and diversified scenes were then manually selected to ensure a wide variety of situations and improve the representativeness of the dataset. The collection was carried out using two identical vehicles equipped with a wide range of sensors, as shown in Figure 5.1.

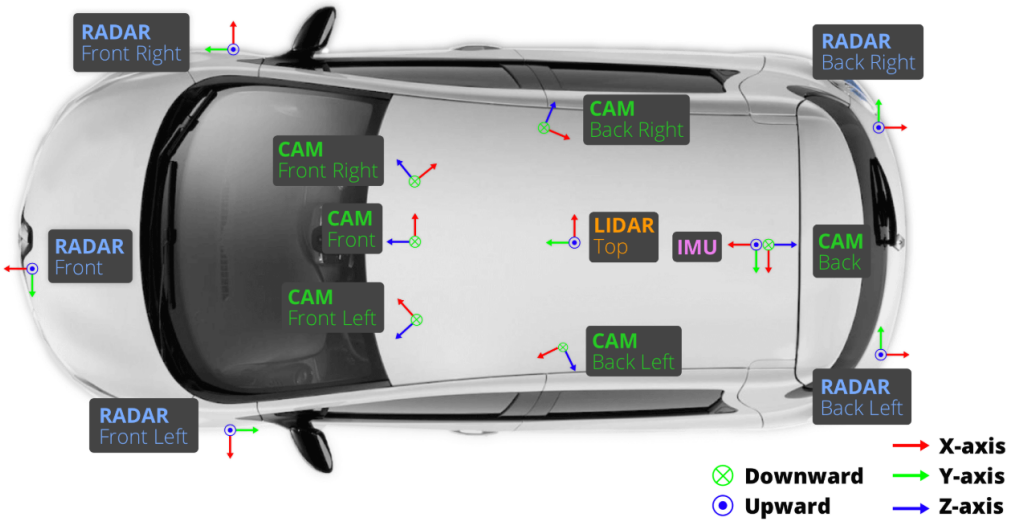

Figure 5.1: Sensor's configuration on the cars used for data collection [14]

Within CARLA simulator, the Ego Vehicle used for data collection was configured to replicate exactly the same layout (in terms of position and orientation) of the sensors used in the nuScenes dataset. This choice led to synthetic data as close as possible to the real data, ensuring the consistency of the information collected and facilitating the integration between simulated and real data.
Each virtual sensor has a set of configurable attributes that can be modified by the user before each simulation to customize the data collection for specific needs. If no changes are made, the sensors will maintain their default values, which have been

set to be, within the implementation's limits, as close as possible to the original configuration used in the nuScenes data collection, reported in Table 5.1.

Sensor	**Capture Frequency**	**Technical specifications**
LIDAR	20 Hz	32 beams, 1080 points per ring, 32 channels, 360° Horizontal FOV, +10°/-30° Vertical FOV, 80 m Range
RADAR	13 Hz	250 m Range
RGB Camera	12 Hz	1600×1200 px Resolution

Table 5.1: nuScenes sensor's specifications [14]

5.1.2 Data format

All the information acquired during the data collection phase are stored in a relational database (Figure 5.2), which is structured to ensure a consistent and accessible organization of the data. The different tables in the database are linked together by means of tokens, which are unique alphanumeric strings that allow the integrity and traceability of information to be maintained in relation to each driving session. This approach allows a comprehensive representation of the entire collected dataset.

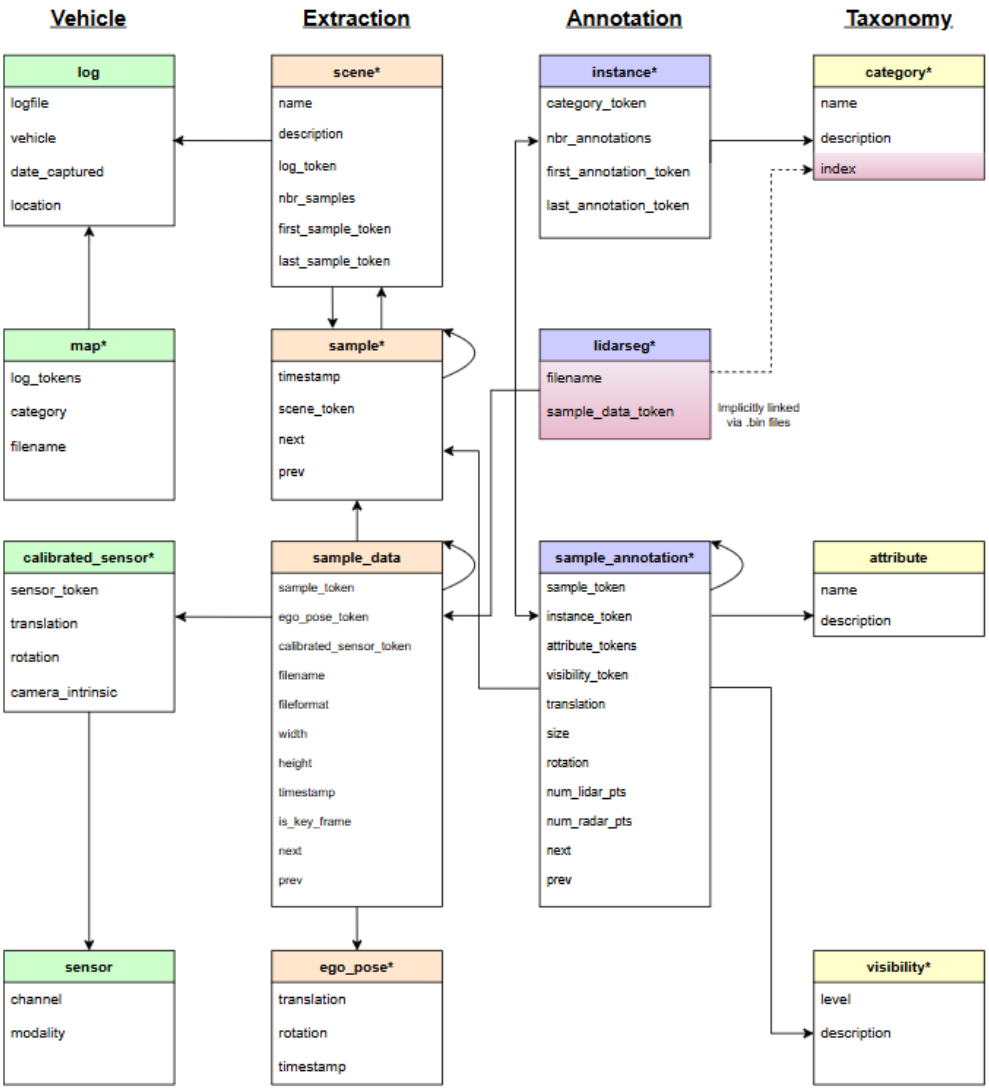

Figure 5.2: nuScenes relational database schema [14]

The structure of the database can be divided into four categories:

- Vehicle: contains all the generic information about the driving session including date, location, type of vehicle and the suite of sensors installed on the car.
- Extraction: contains information about the scenes collected with references to each sensor's output acquired and the global position of the nuScenes vehicle at every time instance.
- Annotation: contains information related to the bounding boxes of the objects

recognized in the scene, storing dimensions, positions, sensor's interaction and attributes to provide an annotation.

- Taxonomy: static attributes that are associated to the objects captured among the scenes to detail their characteristics.

Each table in the database is represented by a JSON file, providing a clear and organized structure for the collected data. The same structure is used for the synthetic data generated by simulations performed in CARLA Simulator. At the end of each execution, the system automatically generates the JSON files corresponding to each database table. These files contain information related to the data collected in the simulation, organized according to the same structure seen in Figure 5.2. This approach ensures consistency between real and simulated data.

5.1.3 Data annotation

In the nuScenes dataset, annotation is performed after data collection by extracting synchronized keyframes at 2 Hz and sending them to a team of experienced annotators for labeling. Each object in the dataset is associated with a semantic category, a 3D bounding box and specific attributes for each frame in which it appears. To ensure high accuracy, several validation steps are included in this process. The list of categories for which annotation is provided in nuScenes is reported in the Table 5.2.

On the other hand, in the project developed with CARLA simulator, the annotation phase is managed directly during the simulation, getting information about the detected objects by reading the attributes of the blueprints present in the virtual environment. This approach produces an already labeled dataset in an immediate and structured way, avoiding the need for a successive manual annotation phase. Moreover, the annotation categories which can be retrived in the virtual environment are more extensive and detailed, including additional information such as the specific model of the vehicles and other attributes that would be difficult to extract by manual annotation. This provides a richer dataset containing metadata that would be complex and time-consuming to annotate manually, thus improving the quality and depth of information available for training and validation of perception algorithms.

Category
animal
human.pedestrian.adult
human.pedestrian.child
human.pedestrian.construction_worker
human.pedestrian.personal_mobility
human.pedestrian.police_officer
human.pedestrian.stroller
human.pedestrian.wheelchair
movable_object.barrier
movable_object.debris
movable_object.pushable_pullable
movable_object.trafficcone
static_object.bicycle_rack
vehicle.bicycle
vehicle.bus.bendy
vehicle.bus.rigid
vehicle.car
vehicle.construction
vehicle.emergency.ambulance
vehicle.emergency.police
vehicle.motorcycle
vehicle.trailer
vehicle.truck

Table 5.2: Annotations categories in nuScenes [14]

Chapter 6

Synthetic Data Generation

This chapter aims to analyze in detail all the development phases of the platform designed for the generation of synthetic data in nuScenes format. Starting with a general overview of the overall framework, are then explored in detail the technical choices implemented to allow the end user to generate all the necessary data in an immediate and automated manner, according with the specific needs. The goal is to make the process as efficient and accessible as possible, thus facilitating the validation of autonomous driving systems through an accurate and standards-compliant synthetic dataset in nuScenes.

6.1 Overview

The purpose of this project is to allow the user to obtain synthetic data related to customized scenarios. For this reason, the pipeline starts with a front-end that allows to configure every detail of the simulation. Here it is possible to select the sensors to be activated on the vehicle, to define their technical parameters and to precisely manage the virtual environment in which the simulation will take place (weather and traffic conditions, type of control for the ego vehicle, etc.). Once all the desired settings have been selected, the simulation can be started.

The simulation can be monitored from a Pygame window, which provides the driver's viewpoint and allows to control the vehicle (detailed in section 6.1.2).

Figure 6.1: Driver's point of view using Pygame window

6.1.1 System's architecture

The whole system is based on a modular architecture in which different scripts, developed in Python, retrieve information from the front-end to handle specific aspects of the simulation.
A conceptual diagram showing the main modules developed and the inputs they receive is shown in Figure 6.2.

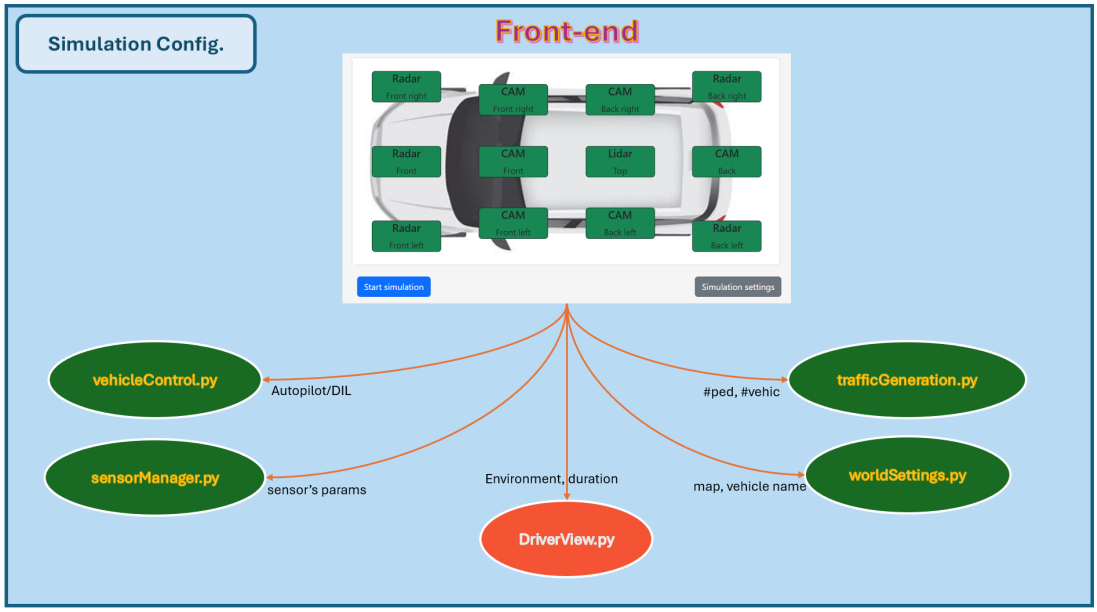

Figure 6.2: Set-up of the simulation through Front-End

- *DriverView.py*: contains the main loop that runs the whole simulation and provides a driver-in-the-loop view via a Pygame window. This script starts the simulation, properly initializes the virtual environment and calls the other Python modules dedicated to specific functions.
- *worldSettings.py*: manages general information of the running simulation such as date and time of start. It also maintains global data structures that can be simultaneously accessed by other scripts to retrieve information needed for their operation.
- *vehicleControl.py*: handle the control of the ego vehicle. It manages the activation of the autopilot mode and implements functions for manual control of the vehicle via keyboard or steering wheel. It also defines some physical properties to enhance the realistic interaction of the vehicle with the environment.
- *sensorManager.py*: it is responsible for generating on the reference vehicle the sensors according to the configuration set on the front-end and implements all the functions for extracting and manipulating the raw data provided as output by each virtual sensor.
- *trafficGeneration.py*: manages the traffic in the simulation environment. In particular, it is responsible for generating pedestrians and other vehicles to populate the scenario in which data are collected. It manages the other dynamic actors that may interact with the ego vehicle during its journey.

During simulation, the collected data are progressively processed and stored in a volatile data structure accurately designed to facilitate its access and subsequent manipulation in the post-simulation phase. Once the data collection is complete, the main script (*DriverView.py*) performs the operations necessary to successfully terminate the simulation and then invokes the *nuScenes.py* module. The latter, developed with reference to the nuScenes database shown in Figure 5.2, reads all the data stored during the simulation phase and proceeds to create the dataset. The data are thus organized in the respective folders according to the source sensors, while the JSON files are generated according to the dataset tables described in Section 5.1.2.

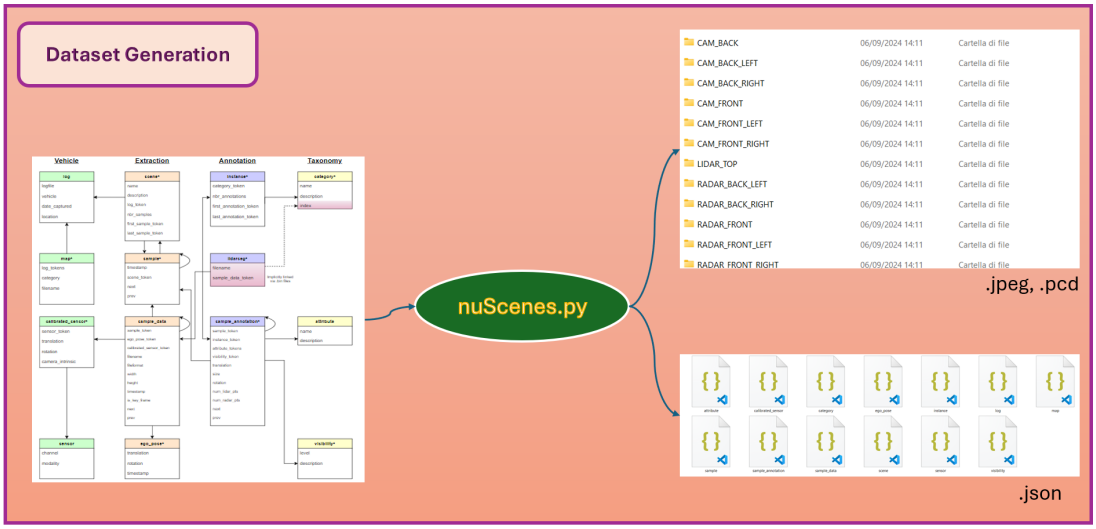

Figure 6.3: Data generation process through python script

6.1.2 Driving modalities

Two different driving modes have been implemented to control the vehicle:

1. Autopilot: vehicle's control is fully managed by the Traffic Manager, discussed in section 4.1.3. In this mode, the vehicle moves autonomously within the map, following predefined behaviors to respect traffic rules such as the right of way, traffic lights rules and speed limits.

2. Manual control: the user can directly control the vehicle, managing various aspects such as acceleration, braking, steering, handbrake and reverse gear. This mode can be selected during front-end configuration and is managed by the *vehicleControl.py* script.

For a more realistic experience, the vehicle can be manually controlled using a driver-in-the-loop approach, taking advantage of the Logitech G29 Driving Force Racing Wheel ® system's features. [1]

Figure 6.4: Logitech G29 Driving Force Racing Wheel ®

The interaction between steering wheel and the vehicle in CARLA Simulator is managed through Pygame. The content of *"manual_control_steeringwheel.py"* provided by official CARLA documentation [15] has been used as a starting point.

The association between the steering wheel's physical buttons (and axes, such as pedals) is recorded in a configuration file named config.ini (Figure 6.5).
This file is then read through a parser in *vehicleControl.py* and, after initialization of the joystick, each index is stored in a variable that is successively used to effectively send inputs to the Ego Vehicle, as shown in Figure 6.6

[1] *"Logitech, Logi, and their logos are trademarks or registered trademarks of Logitech Europe S.A. and/or its affiliates in the United States and/or other countries."*

config.ini

```
[CONFIG]
steering_wheel = 0
throttle = 1
brake = 2
handbrake = 10
reverse = 11
autopilot = 24
```

Figure 6.5: config.ini

```
pygame.joystick.init()
joystick_count = pygame.joystick.get_count()
# print(joystick_count)
if joystick_count < 1:
    raise ValueError("Please Connect a Joystick")
elif joystick_count > 1:
    raise ValueError("Please Connect only One Joystick")

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

#Setup the variables and the button mapping provided by the .ini file

self._parser = ConfigParser()
self._parser.read('config.ini')
self._steer_idx = int(
    self._parser.get('CONFIG', 'steering_wheel'))
self._throttle_idx = int(
    self._parser.get('CONFIG', 'throttle'))
self._brake_idx = int(self._parser.get('CONFIG', 'brake'))
self._reverse_idx = int(self._parser.get('CONFIG', 'reverse'))
self._handbrake_idx = int(
    self._parser.get('CONFIG', 'handbrake'))
self._autopilot_idx = int(
    self._parser.get('CONFIG', 'autopilot'))
```

Figure 6.6: Joystick initialization and config parser

The raw signals provided by the controller required some processing in order to achieve a realistic response during simulation.

In particular:

- Steering Wheel: a tangent function has been applied to get a more sensitive control at larger angles, while maintaining a smoother response near zero.

- Pedals: a logarithmic function has been applied to ensure smoother acceleration at low speeds, avoiding excessively sharp responses, and greater sensitivity at high speeds.

In both cases, appropriate correction coefficients were applied to maintain the values in the correct range. These scaling factors were experimentally tuned to achieve a more reliable interaction between the physical controller and the dynamic behavior of the vehicle in the simulator. The corresponding code is shown in the Figure 6.7

```
def joyControl(self):
    numAxes = self._joystick.get_numaxes()
    jsInputs = [float(self._joystick.get_axis(i)) for i in range(numAxes)]
    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
    K1 = 1.0
    steerCmd = K1 * math.tan(0.8 * jsInputs[self._steer_idx])          #constant 0.8 to maintain the steer in the correct range

    #Mapping of the raw throttle data from range [-1,1] to range [0,1] with corrections
    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

    #Mapping of the raw braking data from range [-1,1] to range [0,1] with corrections
    brakeCmd = K2 + (5 * math.log10(-1 * 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
```

Figure 6.7: joyControl function for manual drive using steering wheel

A keyboard driving mode is also provided for running simulations when steering wheel is not available. In this case, the control is carried out by means of directional arrows or the key combination (W,A,S,D). Adjustments have also been made here to improve the driving experience.

```python
def keyControl(self):

    keys = pygame.key.get_pressed()

    if keys[pygame.K_UP] or keys[pygame.K_w]:
        self._control.throttle = min(self._control.throttle + 0.05, 1.0)
    else:
        self._control.throttle = 0.0

    if keys[pygame.K_DOWN] or keys[pygame.K_s]:
        self._control.brake = min(self._control.brake + 0.2, 1.0)
    else:
        self._control.brake = 0.0

    if keys[pygame.K_LEFT] or keys[pygame.K_a]:
        self._control.steer = max(self._control.steer - 0.05, -1.0)
    elif keys[pygame.K_RIGHT] or keys[pygame.K_d]:
        self._control.steer = min(self._control.steer + 0.05, 1.0)
    else:
        self._control.steer = 0.0
```

Figure 6.8: keyControl function for manual drive using keyboard

6.2 Environment set-up

This section focuses on the initial configuration phase of the simulation environment. In particular, it describes in detail all the options available via the front-end and how they are effectively implemented in the simulation through Python scripts.

6.2.1 Front-end

Since the objective is to simulate the nuScenes data collection process described in Section 5.1.1, the same model of Figure 5.1 is used as a reference for the virtual vehicle configuration.
Each sensor can be completely customized in its technical characteristics and, if considered unnecessary for the purposes of the simulation of interest, can also be deactivated. In this way, the sensor will not be generated on the virtual vehicle, reducing the computational load of the simulation and improving its performance by focusing only on the elements of real interest.

An example of this is shown in Figure 6.9, where some of the sensors (highlighted in red) are disabled.

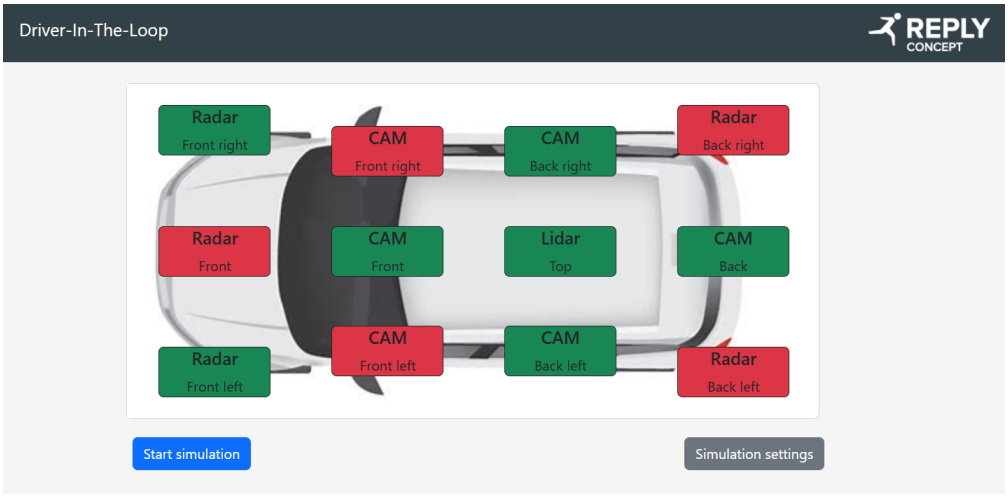

Figure 6.9: Virtual vehicle's sensor configuration from Front-End

The list of specific characteristics depends on the sensor to be modified: clicking on it opens a drop-down menu that allows to modify the data acquisition properties and the position of the sensor itself on the virtual vehicle. Below are the parameters for the three types of sensor that can be customized for the considered model. If no changes are made, the values that are most consistent with the nuScenes configuration will be used by default.

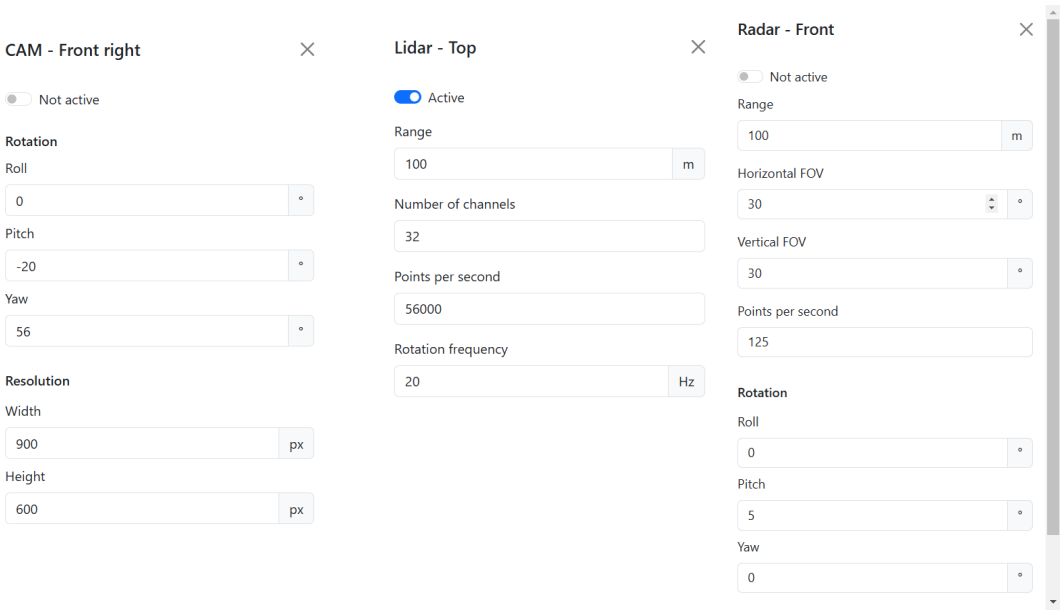

Figure 6.10: Configuration parameters for each type of sensor through Front-End

All other aspects related to the simulation environment can be selected from a dedicated menu in which time of day, weather conditions, number of vehicles and pedestrians to be generated, duration of the simulation and the vehicle driving mode can be specified. It is also possible to select one of the CARLA Simulator maps on which the simulation will take place.

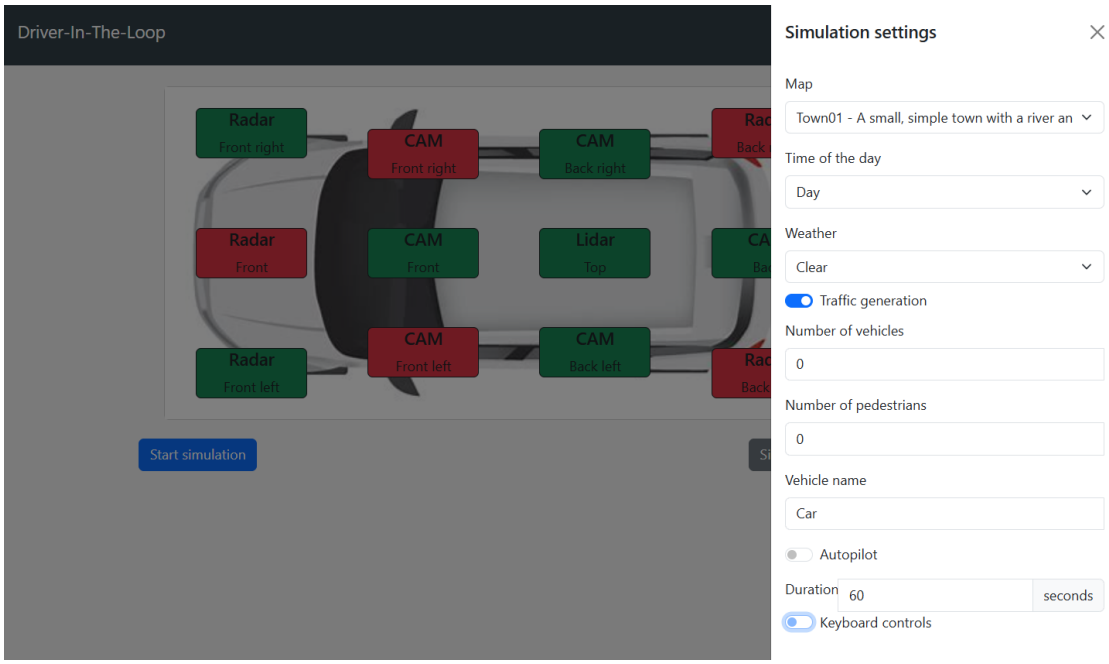

Figure 6.11: Environment settings through Front-End

The list of available maps with a brief description is reported in Table 6.1

Town	Description
Town01	"A small, simple town with a river and several bridges."
Town02	"A small simple town with a mixture of residential and commercial buildings."
Town03	"A larger, urban map with a roundabout and large junctions."
Town04	"A small town embedded in the mountains with a special "figure of 8" infinite highway.
Town05	"Squared-grid town with cross junctions and a bridge. It has multiple lanes per direction. Useful to perform lane changes."
Town06	"Long many lane highways with many highway entrances and exits. It also has a Michigan left."
Town07	"A rural environment with narrow roads, corn, barns and hardly any traffic lights."
Town10	"A downtown urban environment with skyscrapers, residential buildings and an ocean promenade."
Town11	"A Large Map that is undecorated. Serves as a proof of concept for the Large Maps feature."
Town12	"A Large Map with numerous different regions, including high-rise, residential and rural environments."

Table 6.1: Maps available in the simulator from CARLA Official documentation [11]

Once all data have been entered according to requirements, the simulation can be started using the appropriate button. Upon pressing it, the system will automatically generate two files in JSON format, containing the chosen configuration parameters:

- sensors.json: collects all information about the sensors selected for simulation.
- simulation.json: includes the general settings of the simulated environment.

These files will then be processed by the *worldSettings.py* script, which will store the data within an instance of a dedicated class. In this way, the information will be simultaneously accessible by other Python scripts, allowing them to customize the simulation according to the preferences set by the user in the Front-End.

6.2.2 Ego Vehicle set-up

All aspects related to the integration of the virtual sensors required for the simulation into the vehicle are handled by the *sensorManager.py* script.
To match the desired configuration specified in the Front-End, for each sensor two aspects are managed:

1. Technical characteristics: depending on the type of sensor, the corresponding blueprint is identified in the library provided by CARLA Simulator. This allows access to a series of attributes that are modified to configure the technical specifications for data acquisition according to the configuration specified in the Front-End. If the user does not make any change, by defaul the system will provide values corresponding to the real sensors used in the data acquisition phase of nuScenes, as seen in Table 5.1.
   The blueprint attributes modified for each sensor are listed in the following Table:

Radar	Lidar	RGB Camera
horizontal_fov	channels	image_size_x
vertical_fov	rotation_frequency	image_size_y
points_per_second	points_per_second	
range	range	

Table 6.2: Blueprint attributes of interest

2. Positioning: concerning the positions in the Ego vehicle, if the user does not make any changes, the system is designed to automatically apply the default values to replicate the nuScenes configuration shown in Figure 6.12.

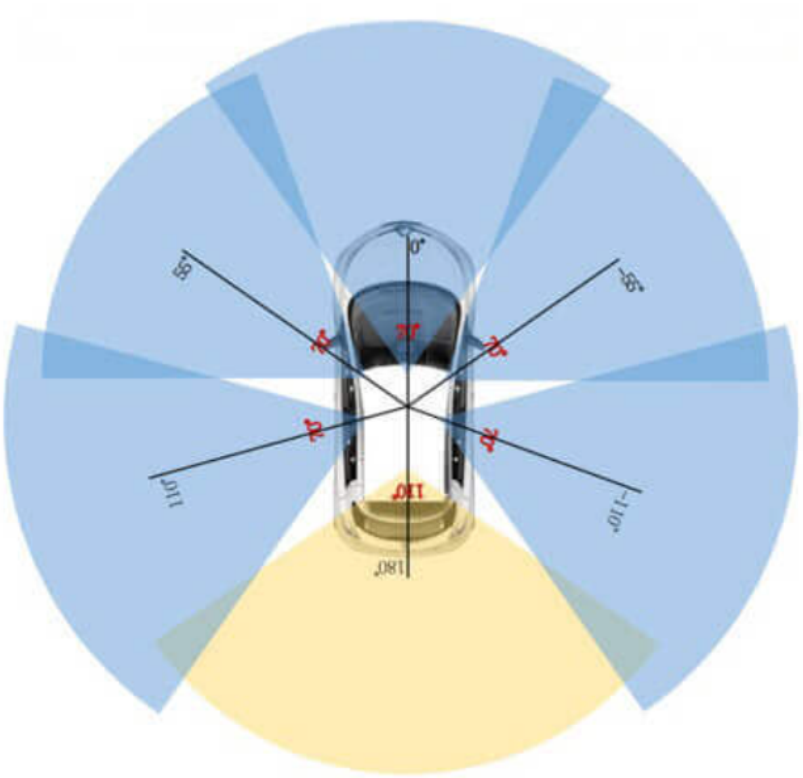

Figure 6.12: nuScenes sensor's orientation [14]

Position and orientation of each sensor are precisely imposed by using instances of the *carla.Transform* class, belonging to the CARLA Python API.

carla.Transform: «Class that defines a transformation, a combination of location and rotation, without scaling»

Instance Variables:

- Location: describes a point in the coordinate system.
- Rotation: describes a rotation for an object.

Since CARLA rotations are described according to the Unreal Engine's reference system, based on left-hand and Z-up conventions, the rotations must be adjusted. In particular, the Yaw values must be reversed in sign to ensure correct alignment with the nuScenes model.

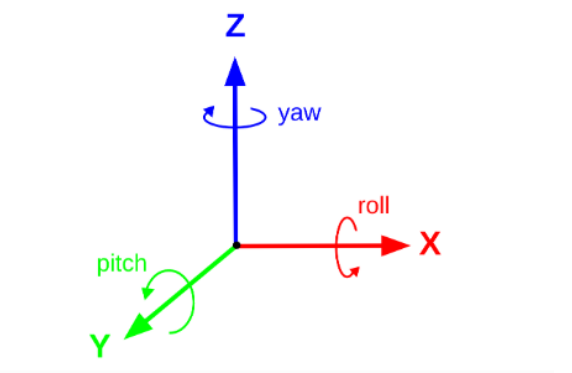

Figure 6.13: Unreal Engine's left handed coordinate system with rotations.[11]

Once the blueprint is correctly configured and the transformation is defined via the Transform object, the sensor can actually be created using the *spawn_actor* method.

world.spawn_actor: «The method will create, return and spawn an actor into the world. The actor will need an available blueprint to be created and a transform (location and rotation). It can also be attached to a parent with a certain attachment type.»

Instance Variables:

- blueprint: "The reference from which the actor will be created."
- transform: "The location and orientation the actor will be spawned with."
- attach_to: "The parent object that the spawned actor will follow around."

In addition to the parameters already described, it is necessary to specify an attribute indicating the element to which the sensor is attached. In the context of the simulation, since the goal is to generate a virtual vehicle equipped with sensors, all devices will be attached to the ego vehicle. The *transform.Location* variables are thus referenced to the center of the vehicle.

6.2.3 Weather and Light conditions

Atmospheric conditions play an important role in this project as they have a direct impact on the quality of the data collected during the simulation. This aspect is managed through the *environment.py* module provided by CARLA Simulator [16]. Any customization specified in the front-end is provided as input to this

script, which is called at the beginning of *DriverView.py*. All changes to the entire simulated world are applied before proceeding with any other operations.

It is given the possibility to select both the time of day, with the resulting changes in sunlight, and the weather conditions. Each available scenario is obtained by modifying the *world.Weather* object provided by CARLA, which includes a set of parameters accessible from the world that can be modified in order to create presets that faithfully reproduce the desired conditions.

- **Time of the day:** lighting of the scene is controlled by the position of the sun. The parameters used to realize the options available to the user are shown in the following table.

Preset	**Altitude Angle**	**Azimuth Angle**	**Description**
Day	45.0°	0.0°	Represents daylight conditions with the sun positioned at a high altitude, ensuring maximum illumination.
Night	-90.0°	0.0°	Simulates night conditions where the sun is completely below the horizon, resulting in darkness.
Sunset	0.5°	0.0°	Creates a sunset effect, with the sun near the horizon, generating long shadows and a warm lighting effect.

Table 6.3: Time of the day presets based on sun position

- **Weather conditions:** these are reproduced by accurately tuning a series of parameters described in the table below.

Parameter	Clear	Overcast	Rain
Cloudiness (%)	10.0	80.0	100.0
Precipitation (%)	0.0	0.0	80.0
Precipitation Deposits (%)	0.0	0.0	90.0
Wind Intensity (%)	5.0	50.0	100.0
Fog Density (%)	0.0	2.0	7.0
Fog Distance	0.0	0.75	0.75
Fog Falloff	0.2	0.1	0.1
Wetness (%)	0.0	10.0	100.0
Scattering Intensity	0.0	0.0	0.0
Mie Scattering Scale	0.0	0.03	0.03
Rayleigh Scattering Scale	0.0331	0.0331	0.0331

Table 6.4: Weather condition presets realized by modifying various environmental parameters.

All the possible atmospheric conditions which can be reproduced by combining weather conditions with sun position are illustrated in Figure 6.14.
A description of each image is provided in the corresponding cell of the following table.

	Clear	Overcast	Rain
Day	*Clear Day*	*Overcast Day*	*Rainy Day*
Sunset	*Clear Sunset*	*Overcast Sunset*	*Rainy Sunset*
Night	*Clear Night*	*Overcast Night*	*Rainy Night*

Table 6.5: Possible combinations of time and weather conditions

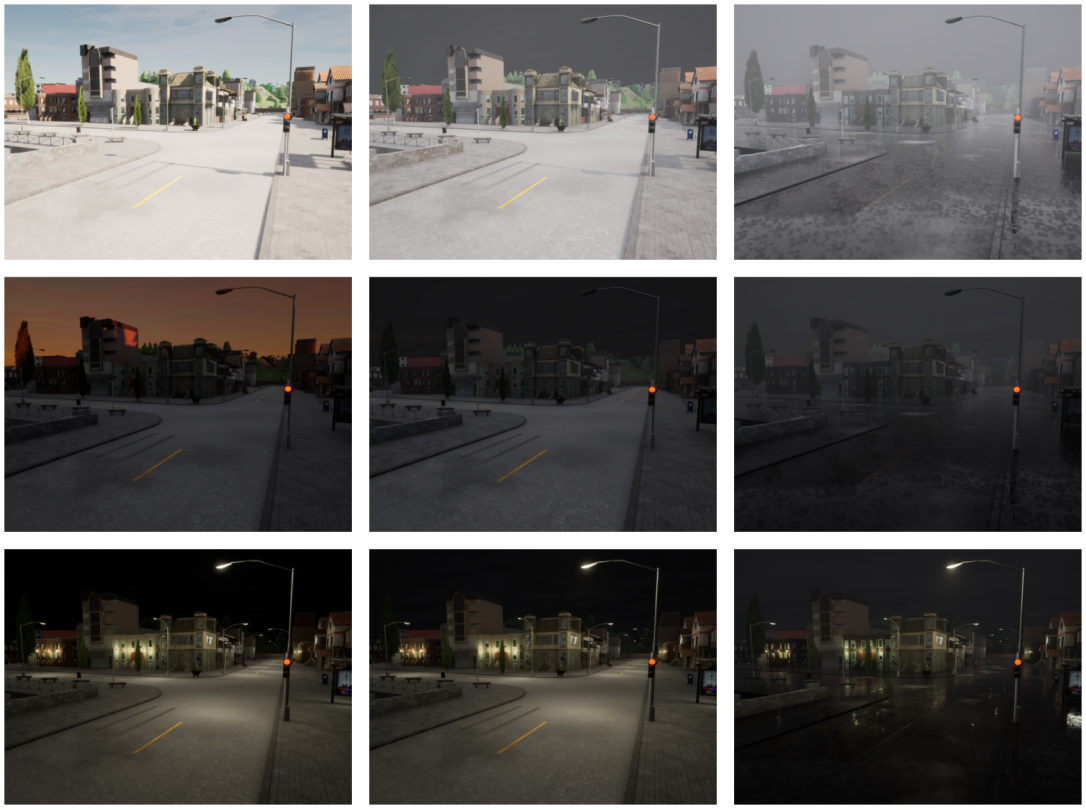

Figure 6.14: Screenshot of the different atmospheric conditions

6.2.4 Traffic generation

An essential element in ensuring a simulation environment that can realistically reproduce the dynamics of real driving is the presence of sufficient vehicle and pedestrian traffic to reproduce complex scenarios in which the ego vehicle interacts with other road users. This is essential to collect significant data on the interactions between the vehicle's sensors and the surrounding elements, which can be used for testing the perception, planning and control algorithms under realistic and varying conditions.
During the initial configuration phase, the user can regulate the intensity of the simulated traffic by selecting the number of pedestrians and vehicles to be generated in the virtual environment. In this way, it is possible to configure the scenario to be reproduced according to specific test requirements.
As explained in Section 4.1.2, each generated vehicle or pedestrian represents an actor associated with a blueprint, so it can be fully configured in terms of appearance parameters, behavior, and interaction with the environment. For example,

vehicles can follow pre-defined routes or be guided autonomously by the Traffic Manager, which simulates real traffic behavior, while pedestrians can cross the road at specific points or move randomly around the map.

In order to correctly generate new actors, the first step is to define the effectively accessible points at which they can appear. These are different depending on the map chosen to host the simulation.
Each Town listed in Table 6.1 is a CARLA object that can be accessed using the command *world.get_map()* and then consent to extrapolate the list of recommended spawn points using the *.get_spawn_points()* method, which returns instances of the *carla.Transform* class for each of them.

```
spawn_points = carlaWorld.world.get_map().get_spawn_points()

#Make a random array of spawnpoints
random.shuffle(spawn_points)
```

Figure 6.15: Python code to get a shuffled list of spawnpoints

Once obtained the list of available spawnpoints for the selected town, is possible to proceed with the traffic generation:

- **Vehicles**: all available vehicle blueprints from the CARLA library are filtered. Then, based on the number of vehicles specified by the user via the front-end, several blueprints are randomly selected to ensure variety in the simulation. Each vehicle is then assigned to one of the available spawn points. The generation process is performed using the *try_spawn_actor* method. This tries to place the vehicle at the assigned point. However, the generation may fail if the spawn point is not free. If successful, the vehicle is added to the list of active vehicles and configured in autopilot mode. Its behavior is thus automatically managed by the Traffic Manager, as described in Section 4.1.3. The corresponding code, extracted from *trafficGeneration.py* script, is reported on Figure 6.16

```python
vehicle_blueprints = carlaWorld.blueprint_library.filter('vehicle.*')

for i in range(min(number_of_vehicles, len(spawn_points))):
    blueprint = random.choice(vehicle_blueprints)
    transform = spawn_points[i]
    vehicle = carlaWorld.world.try_spawn_actor(blueprint, transform)
    if vehicle != None:
        carlaWorld.vehicle_list.append(vehicle)
        vehicle.set_autopilot(True)

print("\n %d vehicles spawned" % len(carlaWorld.vehicle_list))
```

Figure 6.16: Vehicle's generation code (from *trafficGeneration.py*)

- **Pedestrians**: follows a similar logic seen for vehicles, but introduces a new type of actor: the *carla.WalkerAiController*. This component is essential for assigning control of the walkers to the AI. Each generated pedestrian must be associated with a WalkerAiController that manages its movement. The effective control is performed using the *.go_to_location()* method, which allows a specific destination to be assigned. Suitable locations for pedestrian traffic, such as sidewalks, can be obtained using the *.get_random_location_from_navigation()* method, similar to the selection of spawn points.
  In addition to route management, the *WalkerAiController* allows to define the maximum speed that each pedestrian can reach. To increase the realism and variability of the simulation, a random value is assigned to each pedestrian.
  The code that implements this logic is shown in Figure 6.17.

```python
blueprintsWalkers = carlaWorld.world.get_blueprint_library().filter("walker.pedestrian.*")
walker_controller_bp = carlaWorld.world.get_blueprint_library().find('controller.ai.walker')

for i in range(number_of_walkers):
    spawn_point = carla.Transform()
    spawn_point.location = carlaWorld.world.get_random_location_from_navigation()
    if (spawn_point.location != None):
        spawn_points.append(spawn_point)

for spawn_point in spawn_points:
    walker_bp = random.choice(blueprintsWalkers)
    walker = carlaWorld.world.try_spawn_actor(walker_bp, spawn_point)
    if walker != None:
        carlaWorld.walker_list.append(walker)

carlaWorld.world.tick()

for elem in carlaWorld.walker_list:
    controller = carlaWorld.world.spawn_actor(walker_controller_bp,carla.Transform(),elem)
    controller.start()
    controller.go_to_location(carlaWorld.world.get_random_location_from_navigation())
    controller.set_max_speed(1 + random.random())
    carlaWorld.controller_list.append(controller)

print("\n %d walkers spawned" % len(carlaWorld.walker_list))
```

Figure 6.17: Walker's generation code (from *trafficGeneration.py*)

6.3 Data Collection

This section examines the data collection mechanisms implemented in simulation time to directly structure all the collected information in a consistent and accessible manner, thus facilitating the successive creation of the final dataset. The sensor synchronization system, which is essential to ensure that no relevant data are lost during the simulation, is explored. Furthermore, the modeling processes applied to the acquired raw data are also discussed, with the aim of transforming them into a format compatible with the nuScenes standard.

6.3.1 Synchronous configuration

The wide presence of actors in the simulation environment requires careful management of synchronization between them. This aspect becomes essential when dealing with the effective collection of virtual sensor outputs. In order to obtain an accurate and coherent dataset, it is necessary to ensure that each sensor data is related to the same time frame, also avoiding any kind of information loss. A mismatch in the temporal alignment between sensors could in fact compromise the integrity of the dataset, causing discrepancies between the information obtained from different sources, thus complicating the further phases. For this reason, the synchronization system ensures that all outputs are recorded simultaneously, with the aim of obtaining a complete and detailed snapshot of the scene at each instant of the simulation.
As mentioned in Section 4.1.1, the CARLA Simulator is based on a client-server architecture, where all simulation activities are managed by the server (the CARLA world), while interaction with it takes place via Python scripts executed by the client. By default, this type of communication takes place in asynchronous mode, which means that the server runs as fast as possible, while the client interacts with it to retrieve information only when it is ready to proceed. However, this approach presents a significant criticality: if the processing time of the client is longer than that of the server, some instances of the simulation may be lost, compromising the consistency and integrity of the collected data.
To avoid this problem, the synchronous communication mode was adopted in the development of this thesis.

- **Synchronous Mode:** the server only advances the simulation when it receives an explicit signal from the client, thus ensuring a perfect time alignment between the events and the collected data.
  After each frame processing, the simulation stops until the client explicitly requests to continue with the next instance via the *world.tick()* command. In this way, the client can carry out all the necessary operations on the data at

its own rate, and when it is ready, it can invoke the command again to take the simulation one step ahead. The server will then update the state of the simulation by processing the operations of all the actors in the scene, such as the movement of vehicles and pedestrians, or the new set of sensor outputs. The *world.tick()* method also returns an identifier of the processed frame, which is essential to verify the alignment and completeness of the received data.

Another aspect to take into account is the time interval between two simulation steps. By default, with each tick received, the server advances the simulation by a certain amount of time which is determined dynamically on the basis of the computing time required to complete its operations. While this approach may be suitable for asynchronous execution, it is inadequate for synchronous mode because the server has to wait for the client to compute the steps, potentially resulting in non reliable simulation.
To address this issue, a **fixed time-step** approach has been adopted, where the simulation advances in uniform time intervals defined a priori, ensuring that every update of the simulation progresses by the same predefined duration.

In figure 6.18 is reported the code related to the configuration described above.

```
#Set synch mode and returns the original settings
def set_synchronousMode(self,client):
        original_settings = self.world.get_settings()
        settings = self.world.get_settings()
        settings.synchronous_mode = True # Enables synchronous mode
        settings.fixed_delta_seconds = 1/self.FPS
        self.world.apply_settings(settings)
        client.reload_world(False)

        return original_settings
```

Figure 6.18: Synchronous mode, fixed-step configuration

The fixed-step value is defined through *fixed_delta_seconds* attribute of CARLA world. Its value is calculated to guarantee a predefined number of frames per second, so that the simulation advances as far as necessary to meet the requirements.
For example, if a value of 20 FPS is set then, according to

$$fixed\_delta\_seconds = \frac{1}{20} = 0.05s$$

for each world.tick(), the server will advance the simulation by 50 ms.

Through this approach, the speed of the simulation is entirely controlled by the client, who can adapt it to his own computing capabilities. It also permit to reconstruct a temporally consistent ground-truth, where all sensor readings, actor positions, and environmental states are perfectly aligned for each simulation frame.

6.3.2 Data structure

Once the server has been configured to generate new data in accordance with the client's dependencies, it is necessary to ensure that all the information required to build the synthetic dataset is correctly captured and organized.
To provide a graphically clean simulation from a driver-in-the-loop perspective, the server processes all instances of the simulation without loss of information through the synchronous approach described above. However, for the purposes of this work, it is not necessary to record data for every single instant of time. In fact, the official nuScenes dataset provides annotations at a frequency of 2 Hz. (Source [14])
To align with this configuration and avoid storing superfluous data, the collection of information will be carried out by imposing a **sampling time** of 0.5 s, saving only the data produced at these time intervals. This ensures an optimized dataset in terms of size and consistency with the nuScenes format, without compromising the quality and integrity of the information.

Storing procedure: At the beginning of the simulation, an empty vector named *data_list* is initialized, which is used as a container for the recorded data instances. In each instance of interest, determined by the sampling time defined previously, an object of the *DataLog* class is created (Figure 6.19). This class is designed to capture all relevant information, including sensor outputs, the simulation timestamp and details related to the Ego Vehicle and other actors involved in the scene.

```
class DataLog:
    def __init__(self,sample):
        self._sample = sample
        self._timestamp = None
        self._IMU_val = None
        self._GNSS_val = None
        self._RAD_val = []
        self._RGB_val = []
        self._LID_val = []
        self._egoTransform = None
        self._actors = []
```

Figure 6.19: DataLog class used to store data in simulation-time

Before proceeding to the next simulation step with the *world.tick()* command, the corresponding *DataLog* object is added to the end of the *data_list*. All data are thus stored in simulation-time according to the schema reported in Figure 6.20. This provides a structured and chronologically ordered collection of data, facilitating subsequent processing and conversion operations.

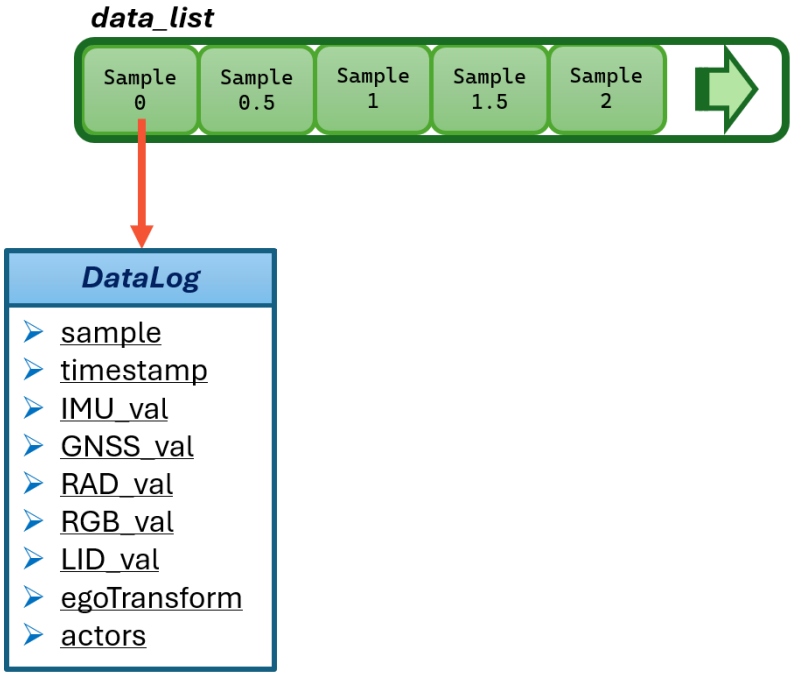

Figure 6.20: Data structure used to collect data in simulation-time

Each virtual sensor always provides a reference to the time at which the raw data was generated. This is the content of the *timestamp* attribute, defined as: «Timestamp of the measurement in simulation seconds since the beginning of the episode.» (Source [11]).
Time alignment is checked by comparing the timestamp of the acquired data with the timestamp recorded in the *DataLog* object at the time of its creation. This ensures that all information relating to the same moment in the simulation is stored consistently, eliminating time discrepancies between different sensors and guaranteeing the quality of the final dataset. This approach is applied to the output of all sensors involved in the simulation.
An example, for the case of IMU data, is shown in the figure 6.21.

```
if (self.settings.data_list[-1])._sample == round(imu_data.timestamp,4):
    (self.settings.data_list[-1])._IMU_val = self.accelerometer
```

Figure 6.21: Example of storing an IMU output in the corresponding DataLog object

6.3.3 Raw data processing

Each sensor introduced into the simulation has a listener method, which is automatically executed each time a new set of data is generated. This mechanism ensures that no information is lost and that any acquired data is immediately processed. Each listener invokes a dedicated function for raw data processing, designed to transform the stream of information received from the sensor into a structured and usable format. This step is essential to ensure the coherence of the collected data and to allow effective management of the information for the final creation of the dataset.
The data processing operations implemented for each sensor are described below.

- **RGB Camera**: the output provided is a *carla.Image* object that defines an image of 32-bit BGRA colors. The list of attributes which can be accessed through this object is reported in the following table.

Attribute	Type	Description
width	int	«Image width in pixels.»
height	int	«Image height in pixels.»
fov	float	«Horizontal field of view in degrees.»
raw_data	bytes	«Flattened array of pixel data, use reshape to create an image array.»

Table 6.6: RGB Camera output attributes [17]

The image is first converted, preserving the original colors, without any processing. The raw data are then extracted and transformed into a NumPy array, properly modeled to correctly reflect the dimensions of the image. At this point, the alpha channel, which represents transparency, is removed, leaving only the three basic channels: red, green and blue. Finally, to obtain an RGB image as a result, the channels are inverted and the processed image is transformed into a PyGame surface ready to be saved.
The corresponding code is reported below:

```python
def process_rgb(self,image):

    image.convert(carla.ColorConverter.Raw)
    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))
```

Figure 6.22: RGB Camera processing operations

- **LiDAR Sensor:** the output provided is a *carla.LidarMeasurement* containing a package with all the points generated during a "static picture" of the scene. The list of attributes which can be accessed through this object is reported in the following table.

Attribute	Type	Description
channels	int	«Number of lasers shot.»
horizontal_angle	float (*rad*)	«Horizontal angle the LIDAR is rotated at the time of the measurement.»
fov	float	«Horizontal field of view in degrees.»
raw_data	bytes	«Received list of 4D points. Each point consists of [x,y,z] coordinates plus the intensity computed for that point.»

Table 6.7: LiDAR output attributes [17]

The raw data are initially converted into a NumPy array and reshaped into four columns, each corresponding to one of the components of the XYZI format.
The ring index is then calculated, which is a value that identifies the vertical layer from which each LiDAR point originates. The number of rings depends on the number of sensor channels selected during sensor configuration. Each ring represents a different vertical scan layer. To assign the correct index to each point, each channel is iterated and the number of points acquired is calculated. The channel index is then replicated for the number of points belonging to that level and stored in an array. Finally, the LiDAR data are reorganised into a more complete structure, combining column by column the spatial information (x, y, z), intensity and ring index.
The code which implements what explained above is reported in the following figure.

```python
def process_lidar(self,lidar_data):


    points = np.frombuffer(lidar_data.raw_data, dtype=np.float32).reshape(-1, 4)

    num_channels = lidar_data.channels

    ring_indices = []

    for channel in range(num_channels):
        point_count = lidar_data.get_point_count(channel)

        ring_indices.extend([channel] * point_count)
    ring_indices = np.array(ring_indices)

    points_with_ring = np.column_stack((points, ring_indices))
```

Figure 6.23: LiDAR Sensor processing operations

- **RADAR Sensor:** the output provided is a *carla.RadarMeasurement* containing an array of *carla.RadarDetection*, which specifies their polar coordinates, distance and velocity.

Attribute	Type	Description
altitude	float (rad)	«Altitude angle of the detection.»
azimuth	float (rad)	«Azimuth angle of the detection.»
depth	float (m)	«Distance from the sensor to the detection position.»
velocity	float $(\frac{m}{s})$	«The velocity of the detected object towards the sensor.»

Table 6.8: RADAR output attributes [17]

The azimuth, elevation and depth values extracted from each detected point are first mapped into Cartesian coordinates using the polar coordinate conversion formula showed below

```
x = r .* cos(elevation) .* cos(azimuth)
y = r .* cos(elevation) .* sin(azimuth)
z = r .* sin(elevation)
```

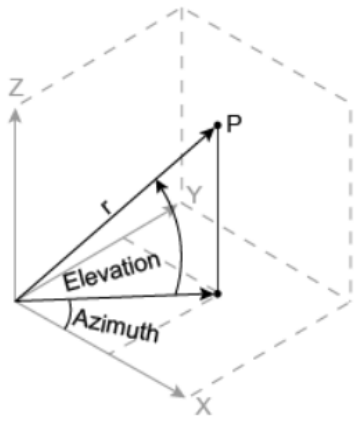

Figure 6.24: Mapping from Spherical coordinates to three-dimensional Cartesian coordinates [18]

Then the velocity components along the *vx_comp* and *vy_comp* axes are also calculated, using the measured velocity and decomposing it according to the azimuth and altitude of the object. This provides a more detailed spatial representation of the movement of objects detected by the radar, which is essential for analyzing the dynamics of the scene. The Python code which implements what explained above is showed in figure 6.25

```python
def process_rad(self,radar_data):

    for detect in radar_data:
        velocity = detect.velocity
        altitude = detect.altitude
        azimuth = detect.azimuth
        depth = detect.depth




        x = depth * np.cos(azimuth) * np.cos(altitude)
        y = depth * np.sin(azimuth) * np.cos(altitude)
        z = depth * np.sin(altitude)

        vx_comp = vx = velocity * np.cos(azimuth) * np.cos(altitude)
        vy_comp = vy = velocity * np.sin(azimuth) * np.cos(altitude)
```

Figure 6.25: RADAR Sensor processing operations

- **IMU Sensor:** provides as output a *carla.IMUMeasurement* object. This is accessed to retrive the following information:

Attribute	Type	Description
accelerometer	*carla.Vector3D* $(\frac{m}{s^2})$	«Linear acceleration.»
gyroscope	*carla.Vector3D* $(\frac{rad}{s})$	«Angular velocity.»

Table 6.9: IMU output attributes [17]

To ensure the reliability of the data acquired and to avoid values that could cause inconsistencies in the data set, each measurement is compared with predefined limit values and, if necessary, adjusted so that they are not exceeded. Accelerometer data (along the x, y and z axes) are acquired and limited using the min and max function to ensure they remain within a manageable range, avoiding extreme values. The same process is applied to the gyroscope data, after converting the angle values from radians to degrees using *math.degrees()*. The corresponding code is reported in the following figure

```
def process_imu(self,imu_data):
    limits = (-99.9, 99.9)
    self.accelerometer = (
        max(limits[0], min(limits[1], imu_data.accelerometer.x)),
        max(limits[0], min(limits[1], imu_data.accelerometer.y)),
        max(limits[0], min(limits[1], imu_data.accelerometer.z)))

    self.gyroscope = (
        max(limits[0], min(limits[1], math.degrees(imu_data.gyroscope.x))),
        max(limits[0], min(limits[1], math.degrees(imu_data.gyroscope.y))),
        max(limits[0], min(limits[1], math.degrees(imu_data.gyroscope.z))))
```

Figure 6.26: IMU Sensor processing operations

- **GNSS Sensor:** the object retrived is a *carla.GNSSMeasurement* which reports the position of the sensor through the following attributes:

Attribute	Type	Description
altitude	float (m)	«Height regarding ground level.»
latitude	float (deg)	«North/South value of a point on the map.»
longitude	float (deg)	«West/East value of a point on the map.»

Table 6.10: GNSS output attributes [17]

No further operations are performed on the raw data provided by this type of sensor.

6.4 Dataset generation

This section analyzes the dataset generation procedure. The data collected during the simulation phase, stored according to the scheme shown in Figure 6.20, are now reorganized to produce an output consistent with that offered by the nuScenes dataset. This operation is only performed after the entire simulation has been completed, ensuring that all necessary information has been correctly captured and all aspects of the simulation have been properly terminated.
The final dataset will be composed of two main parts. The first one includes every output sample produced by the sensors involved in the simulation. Data from the RGB camera, LiDAR and Radar are stored in their appropriate format to maintain the collected information's quality and integrity.
A series of JSON files, in the second part, define how the different samples are linked and introduces additional metadata for annotations. These files organize the dataset structure to allow coherent reconstruction of each simulated scene.

6.4.1 Output samples

A dedicated folder containing all the samples acquired during the simulation is automatically created for each sensor that needs its output to be stored, as reported in the following figure.

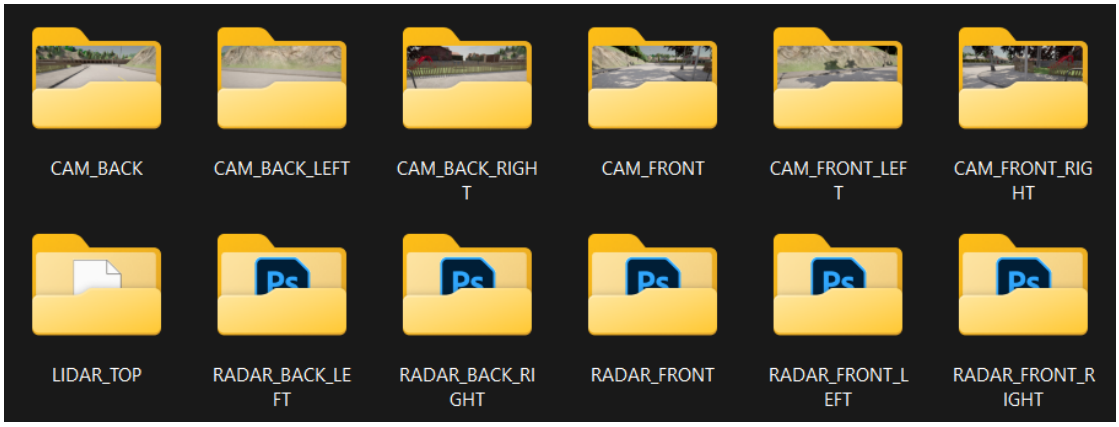

Figure 6.27: Data folder structure

Each sample file is renamed in a structured manner, including essential information such as the type of sensor, the vehicle on which it is installed, and the date and time of acquisition. This ensures a clear organization and allows easy access to the data.
The choice of output format has been carefully considered for each sensor, taking into account both data quality and compatibility with the nuScenes dataset:

- **LiDAR Output:** samples are saved in *.pcd.bin* format, without further processing. This is sufficient to accurately represent the information acquired from the LiDAR sensor and to maintains compatibility with the format used in the original nuScenes collection.
- **RGB Output:** images are stored in *.jpg* format, taking advantage of lossy compression. This technique reduces the file size, allowing optimized storage management and improved simulation performance, without significantly compromising the visual quality required for analysis.
- **RADAR Output:** data are stored according to the *Point Cloud Data v0.7* standard. This format ensures that all essential radar information is retained and structured in a manner consistent with the reference dataset, facilitating integration and comparative analysis with real data.

6.4.2 JSON files

All information captured during the simulation is organized in a relational database, structured according to the schema introduced in Section 5.1.2. For each table in the schema, a corresponding JSON file is created to store the associated information. Using the function shown in Figure 6.28, a 32-character alphanumeric token is generated for all items entered into the database to ensure their unique identification.

```python
# Generates an alphanumeric unique token and append it to a token_list
def tokenGen(token_list):

    token_length = 32
    sample_str = "abcdefghijklmnopqrstuvwxyz0123456789"

    generated_string = ''.join(random.choices(sample_str, k = token_length))

    while generated_string in token_list:

        generated_string = ''.join(random.choices(sample_str, k = token_length))

    token_list.append(generated_string)
    return generated_string
```

Figure 6.28: Token generation function

For each table of the schema, a Python class is designed to hold the data of interest, ensuring consistency with the relational structure of the dataset. In each class object, the data collected during the simulation are first processed and reorganized according to the nuScenes format. Then, they are placed in a *dictionary* type variable, which facilitates their management and serialization. Each object of a class represents a row in the corresponding table and it is stored into a dedicated vector which represents a specific table in the database.
An extraction of this process is reported below, showing a Python class (Figure 6.29) designed for reconstruction of the *Log Table* belonging to the nuScenes scheme (Figure 6.30)

```python
class log_Obj:
    def __init__(self,carlaWorld):

        self.token = tokenGen(carlaWorld.token_list)
        self.logfile = f"{carlaWorld.vehicleName}-{carlaWorld.date}"
        self.vehicle = carlaWorld.vehicleName
        self.date_captured = str(date.today())
        self.location = carlaWorld.map

        self.table_dict = {
                "token": self.token,
                "logfile": self.logfile,
                "vehicle": self.vehicle,
                "date_captured": self.date_captured,
                "location": self.location
                }
```

Figure 6.29: Log database table - Python class

```
log {
   "token":                   <str> -- Unique record identifier.
   "logfile":                 <str> -- Log file name.
   "vehicle":                 <str> -- Vehicle name.
   "date_captured":           <str> -- Date (YYYY-MM-DD).
   "location":                <str> -- Area where log was captured, e.g. singapore-onenorth.
}
```

Figure 6.30: Log database table - nuScenes scheme [14]

For each element, the content of the *Dictionary* variable is extracted and written to the corresponding JSON file.
The whole dataset is implemented by iterating this procedure on all the other tables specified in the nuScenes schema.

Chapter 7

Conclusions and Future Developments

The final objective maintained during the development phases of this thesis was to replicate in a virtual environment the data collection process adopted by the *Motional* team of *nuScenes*, and to develop a methodology that would allow standardized datasets to be obtained in a safe and controlled manner. This approach has been studied for integration with existing validation platforms, ensuring compatibility with methodologies already used to deal with real data. The implemented process includes a structured pipeline that handles the generation, organization and annotation of data acquired in the simulation. The collected data are converted into a format consistent with industry standards, enabling efficient management and easy integration into existing validation workflows. During the development phases, analysis of the system's performance and capabilities identified both the strengths and limitations of this approach compared to traditional data acquisition. The use of synthetic data revealed a promising solution for improving ADAS technology development and testing processes, but also highlighted some critical issues that need to be addressed for effective integration with real data. The results obtained, reported in the following sections, suggest that the developed framework can be a valid support for the validation of ADAS, reducing the need for road tests and offering greater flexibility in the creation of test scenarios. However, in order for this technology to be adopted on a large scale, some aspects need to be further investigated by improving the fidelity of the simulation and by exploring strategies to better combine synthetic data with real data.

7.1 Advantages of using Synthetic Data

Compared to traditional data collection on the road, at each stage of the development process, the approach presented in this thesis has highlighted numerous advantages, providing effective solutions to overcome many of the limitations of real-world data collection. The benefits emerged not only improve the efficiency and security of the process but also ensure greater flexibility of the generated datasets, making them more suitable for the ADAS validation use cases.
The main benefits include:

- **Cost-efficiency:** the significant cost reduction compared to real-world data collection is one of the main advantages that emerged. The latter requires a significant investment in resources, including vehicles equipped with advanced sensors, data acquisition hardware, fuel and specialized personnel to operate and maintain the entire system. For example, the vehicle used to collect the data in the nuScenes dataset is equipped with an expensive suite of sensors, including six RGB cameras, five radars and a 32-channel LiDAR.
  In contrast, synthetic data generated in virtual environments eliminate the need to physically purchase and integrate such sensors, thus avoiding high sensorization costs.
  Furthermore, simulation allows detailed data sets to be obtained without the expense of sensor maintenance, periodic calibration and component wear, further reducing operational costs.
  The absence of physical vehicles also eliminates fuel consumption and logistics costs, making synthetic data generation a cost-effective and highly scalable solution.

- **Safety:** the use of virtual scenarios completely eliminates the dangers associated with road testing, including the possibility of collisions, unexpected malfunctions and dangerous interactions with other road users that could expose to risks not only the operating team, but also pedestrians, cyclists and other vehicles. Through the virtual approach presented, these dangers are completely eliminated, allowing rigorous and repeatable testing to take place in a controlled and safe environment, without any risk to people or property. In addition, using simulations it is also possible to reproduce and then test rare or dangerous situations, such as extreme weather conditions, sensor failures or emergency scenarios that would be risky and difficult to recreate in the real world.

- **Time saving:** real world data collection, as already discussed, is a long and complex process, requiring the organization of road test sessions with vehicles and personnel management, and the subsequent analysis of acquired data.

In contrast, in a virtual environment, data generation is fast and scalable, allowing large amounts of information to be collected in drastically reduced time. With synthetic data, it is also possible to extrapolate all the information necessary for data annotation, as the simulation has direct access to the characteristics of each object in the scene. This makes it possible to automate the annotation process, generating ready-to-use datasets and thus significantly accelerating the model development and validation cycle.

- **Diversity and Flexibility:** through the front-end described in Section 6.2.1, it is possible to customize every aspect of the simulation, allowing it to vary in a controlled manner. This approach overcomes one of the main limitations of real-world data collection, where the variability of conditions is linked to external factors that cannot always be controlled. Furthermore, the integration of the Driver-In-The-Loop mode allows the user to actively interact with the simulation and replicate specific scenarios according to validation needs. Thanks to this flexibility, it is possible to obtain a diverse and balanced dataset, including both realistic scenarios and edge cases that are difficult to capture in the real world. This is particularly effective for the training and validation of ADAS systems, which require a wide range of situations to ensure robustness and effectiveness in real-world operational contexts.

- **Standardization:** the reorganization process according to the *nuScenes* standard ensures compliance with one of the most widely used formats in the automotive industry, making the generated datasets easily usable in existing validation pipelines that are already optimized for working with real data, without requiring substantial changes to processing and analysis systems. The ability to take advantage of a standardized format also reduces the risk of conversion errors and facilitates comparison between synthetic and real data. This improves the reliability of validation process but also highlights the aspects where further optimization work is needed.

7.2 Limitations Found

During the development and final testing of the platform, several limitations of this approach were identified. Due to the virtual nature of the environment, the level of detail in some aspects of the information generated resulted to be less than that provided by real data.
Virtual sensors operate under ideal and perfectly calibrated conditions, regardless of slight misalignment due to mounting tolerances, vibrations, or small shocks. This leads to a discrepancy between synthetic and real data, reducing the ability to validate perception algorithms in more realistic scenarios, where calibration errors

are inevitable.
Moreover, technical limitations related to the simulator used and to the computational capabilities of the hardware on which the simulation run, also affect the fidelity and complexity of the scenes reproduced.
In particular, for the CARLA version used in the development of this thesis (v0.9.15) the following limitations have emerged:

- **Parked vehicles:** vehicles parked at the side of the road are represented as static meshes, unlike the dynamic vehicles used to simulate traffic, which are instead actors in their own right and associated with blueprints. This means that it is impossible to extract any information about them, such as position, size or orientation. This limitation has a significant impact on the generation of virtual ground truth, as the bounding boxes of parked vehicles are not captured by the simulation. The virtual dataset will therefore be incomplete compared to one generated in the real world, where static objects along the road can provide crucial information for validating ADAS.

- **RADAR output:** by analyzing the RADAR sensor output provided by nuScenes emerged that the information generated by the corresponding virtual sensor in CARLA is not sufficient to reconstruct in detail all aspects of sampling. From the dataset released by *Motional Team* in March 2019, an example of nuScenes radar output is reported in the following figure.

```
# .PCD v0.7 - Point Cloud Data file format
VERSION 0.7
FIELDS x y z dyn_prop id rcs vx vy vx_comp vy_comp is_quality_valid ambig_state x_rms y_rms invalid_state pdh0 vx_rms vy_rms
SIZE 4 4 4 1 2 4 4 4 4 4 1 1 1 1 1 1 1 1
TYPE F F F I I F F F F F I I I I I I I I
COUNT 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
WIDTH 125
HEIGHT 1
VIEWPOINT 0 0 0 1 0 0 0
POINTS 125
DATA binary
```

Figure 7.1: nuScenes_mini RADAR output fields (March 2019 [14])

The file format used for each RADAR sample is *PCD v0.7*. It is composed of different fields needed to add more details in the recorded output. Most of these do not have a reference in CARLA and consequently in each virtual output they have been set with a null value.
In particular, the missing information concerns the following fields:

Field	Description
invalid_state	state of Cluster validity state.
dynProp	Dynamic property of cluster to indicate if is moving or not.
ambig_state	State of Doppler (radial velocity) ambiguity solution.
pdh0	False alarm probability of cluster (i.e. probability of being an artefact caused by multipath or similar).

Table 7.1: Synthetic RADAR data missing fields (from nuScenes-devkit) [19]

- **Hardware performance:** by default, simulations are set to replicate the *nuScenes* sensor configuration seen in figure 5.1. This involves the simultaneous use of six RGB sensors plus an additional one inserted inside the vehicle to provide the driver's point of view. Computing this kind of sensors in CARLA simulator is a very expensive operation done by the server which necessarily compromise the performance of the simulation.
  When using camera sensors, images are generated in the GPU and then they are copied to the CPU in order to be sent to the client. The GPU to CPU copy is an expensive operation which stalls both devices. The larger the images or number of them, the longer the stall occurs. For this reason, each RGB sensor spawned in the CARLA world causes a significant FPS drop in the simulation, leading to the risk of sampling loss.
  In order to analyze this behaviour, a series of tests were carried out in which a certain FPS value was imposed on the server (via synchronous mode) and then a series of RGB sensors were progressively introduced to check the FPS drop they caused.
  The hardware used for the benchmark tests and the results obtained are reported below.

CPU	RAM	GPU
i9-14th	32GB	RTX 4080 12GB

Table 7.2: Hardware used for benchmark tests

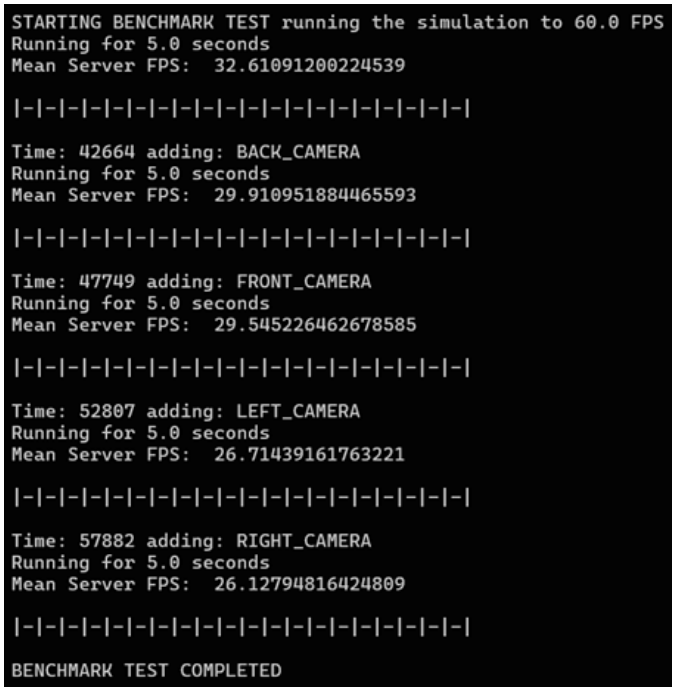

Figure 7.2: Benchmark test with 60 FPS

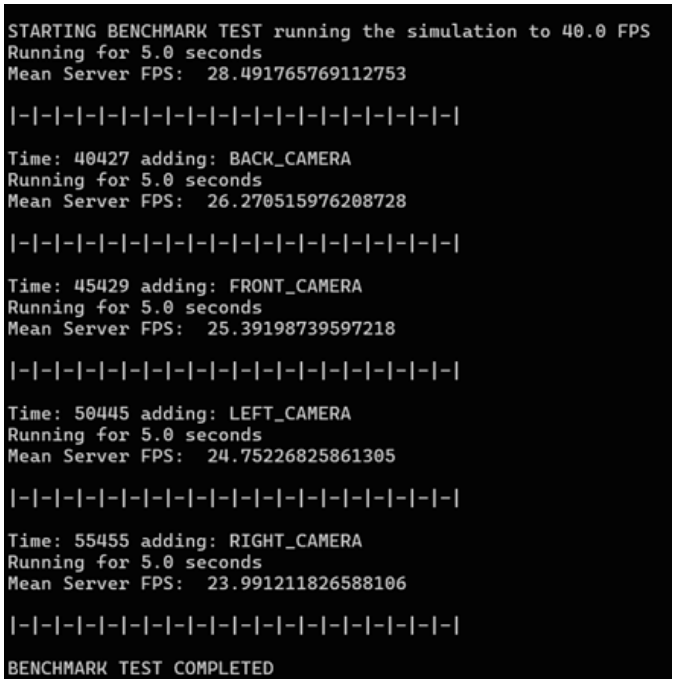

Figure 7.3: Benchmark test with 40 FPS

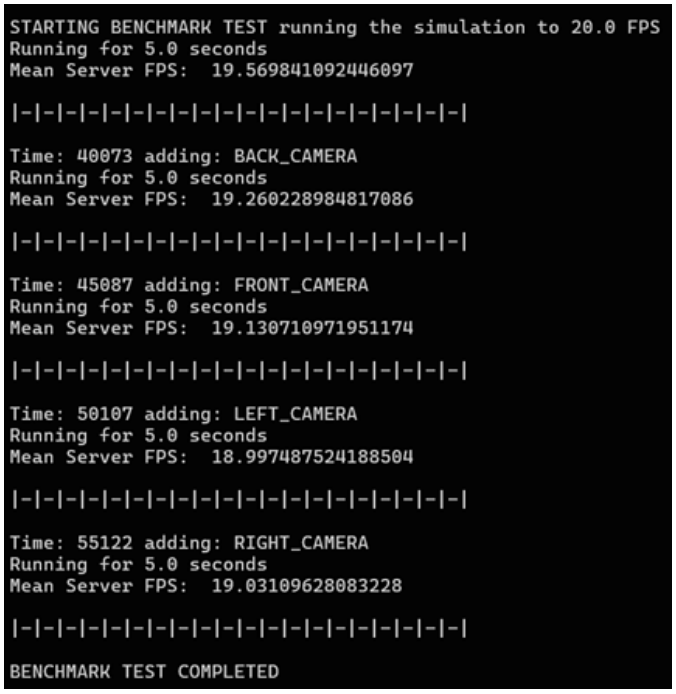

Figure 7.4: Benchmark test with 20 FPS

Results: With only one RGB sensor (used for driver's point of view) the server immediately drops to 30 FPS. By adding other four RGB sensors (used for data logging) the server can run around to 20/25 FPS.
Since, as explained, the bottleneck is GPU to CPU communication, a possible solution to improve the performances (even adding more RGB sensors) is to use the Carla Multi-GPU feature [20] in which the user can launch multiple secondary servers, each utilizing a dedicated GPU to handle rendering tasks for the primary server which is responsible for managing the distribution of user-defined sensors across the available secondary servers, optimizing workload allocation.

7.3 Future Developments

In order to maximize its potential, the project presented in this thesis leaves room for future developments that can overcome the identified limitations.
A key aspect will be to reduce the discrepancy between the synthetic data generated in the virtual environment and the real data collected in the field, also known as the Sim2Real gap. This will be essential in order to make optimal use of the synthetic datasets in validation processes without the need for complex adjustments.

Another goal will be the implementation of real-time visualization for collected data during simulation, enabling immediate analysis and facilitating the debugging and optimization of generated scenarios.
Finally, the possibilities for analysis and use of the collected data can be extended by implementing integration with some external frameworks to ensure greater interoperability with established tools in the field.

7.3.1 Sim2Real gap - NVIDIA Omniverse

Through the Omniverse Unreal Engine plug-in [21], it will be possible to easily introduce SimReady content into the simulation, i.e., 3D objects with precise physical properties, behaviors and associated data streams to accurately represent the real world [22]. In this way, vehicles already configured with working lights, doors and wheels, as well as props, can be used immediately to decorate CARLA maps, allowing virtual scenarios to be generated with much greater visual fidelity, improving the quality of images perceived by virtual sensors. In addition, support for co-simulation with NVIDIA DriveSim enables more accurate simulation of dynamic vehicle and sensor behaviour, ensuring the generation of increasingly representative synthetic data.

7.3.2 Real-Time visualization - CarlaViz plugin

CarlaViz is a plugin that allows the simulation to be viewed in a web browser. The integration of CARLAviz into this project would allow for direct real-time monitoring of all the actors involved in the simulation, thus improving the debugging efficiency. The ability to simultaneously view data from sensors such as LIDAR, cameras, RADAR and GPS provides a clear overview of the information captured, making it easier to detect anomalies. (Source [23])

7.3.3 Dataset analysis - Rerun

Rerun is an open source framework designed for multi-sensor data visualization and analysis. Providing an efficient infrastructure to handle large amounts of data, it can be integrated into this project for a dynamic exploration of the collected data, filtering and navigating the information through the advanced graphical representation tools offered.(Source [24])

Bibliography

[1] World Health Organization. *Global Status Report on Road Safety 2018*. Dec. 2018. URL: https://www.who.int/publications/i/item/9789241565684 (cit. on p. 1).

[2] Vlatko Otasevic. *Road traffic injuries are not accidents. They are preventable.* Nov. 2018. URL: https://montenegro.un.org/en/41896-road-traffic-injuries-are-not-accidents-they-are-preventable (cit. on p. 2).

[3] *Vehicle General Safety Regulation*. Nov. 2019. URL: https://eur-lex.europa.eu/eli/reg/2019/2144/oj (cit. on p. 2).

[4] *United Nations Economic Commission for Europe*. URL: https://wiki.unece.org (cit. on p. 2).

[5] SAE International. *SAE Levels of Driving Automation*. May 2021. URL: https://www.sae.org/blog/sae-j3016-update (cit. on p. 6).

[6] Shlomi Hacohen, Oded Medina, and Shraga Shoval. «Autonomous Driving: A Survey of Technological Gaps Using Google Scholar and Web of Science Trend Analysis». In: *IEEE Transactions on Intelligent Transportation Systems* 23.11 (2022), pp. 21241–21258. DOI: 10.1109/TITS.2022.3172442 (cit. on p. 8).

[7] NHTSA. *The Evolution of Automated Safety Technologies*. URL: https://www.nhtsa.gov/vehicle-safety/automated-vehicles-safety (cit. on p. 9).

[8] Dorleco. *ADAS Testing and Validation*. May 2024. URL: https://dorleco.com/adas-testing-and-validation/ (cit. on p. 11).

[9] Michał Pietruch, Andrzej Mlyniec, and Andrzej Wetula. «An overview and review of testing methods for the verification and validation of ADAS, active safety systems, and autonomous driving». In: *Mininig - Informatics Automation and Electrical Engineering* 1 (541) (Jan. 2020), pp. 19–36. DOI: 10.7494/miag.2020.1.541.19 (cit. on p. 12).

[10] Vaibhav. *ADAS Validation: Challenges and Methodologies in Advanced Driver Systems Testing*. Feb. 2024. URL: https://www.embitel.com/blog/embedded-blog/adas-validation-challenges-and-methodologies-in-advanced-driver-systems-testing (cit. on p. 14).

[11] CARLA Team. *CARLA Documentation*. URL: https://carla.readthedocs.io/en/latest (cit. on pp. 16, 18, 19, 37, 40, 49).

[12] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. «CARLA: An Open Urban Driving Simulator». In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 1–16 (cit. on pp. 16, 17).

[13] Sumbal Malik, Manzoor Ahmed Khan, and Hesham El-Sayed. «CARLA: Car Learning to Act — An Inside Out». In: *Procedia Computer Science* 198 (2022). 12th International Conference on Emerging Ubiquitous Systems and Pervasive Networks / 11th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare, pp. 742–749. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2021.12.316. URL: https://www.sciencedirect.com/science/article/pii/S1877050921025552 (cit. on p. 18).

[14] Holger Caesar et al. «nuScenes: A multimodal dataset for autonomous driving». In: *arXiv preprint arXiv:1903.11027* (2019) (cit. on pp. 21–24, 26, 39, 48, 58, 62).

[15] CARLA Simulator. *Manual Control with Steering Wheel Script*. Accessed: 2025-01-10. 2025. URL: https://github.com/carla-simulator/carla/blob/master/PythonAPI/examples/manual_control_steeringwheel.py (cit. on p. 31).

[16] CARLA Simulator. *Environment Control Script*. Accessed: 2025-01-10. 2025. URL: https://github.com/carla-simulator/carla/blob/master/PythonAPI/util/environment.py (cit. on p. 40).

[17] CARLA Team. *CARLA Sensor Reference*. Accessed: February 10, 2025. 2024. URL: https://carla.readthedocs.io/en/latest/ref_sensors/ (cit. on pp. 50–52, 54, 55).

[18] The MathWorks Inc. *sph2cart*. Accessed: 2025-03-07. 2025. URL: https://it.mathworks.com/help/matlab/ref/sph2cart.html (cit. on p. 53).

[19] nuTonomy. *nuScenes DevKit*. Accessed: 2025-03-25. 2025. URL: https://github.com/nutonomy/nuscenes-devkit (cit. on p. 63).

[20] CARLA Team. *Advanced Multi-GPU Rendering in CARLA*. Accessed: March 25, 2025. 2024. URL: https://carla.readthedocs.io/en/latest/adv_multigpu/ (cit. on p. 65).

[21] CARLA Simulator. *Ecosystem SimReady*. Accessed: 2025-03-05. 2025. URL: https://carla.readthedocs.io/en/latest/ecosys_simready/ (cit. on p. 66).

[22] NVIDIA. *Omniverse SimReady Assets*. Accessed: 2025-03-05. 2025. URL: https://developer.nvidia.com/omniverse/simready-assets (cit. on p. 66).

[23] CARLA Team. *CARLAviz Plugin Documentation*. Accessed: March 5, 2025. 2024. URL: https://carla.readthedocs.io/en/latest/plugins_carlaviz/ (cit. on p. 66).

[24] Rerun Development Team. *Rerun: A Visualization SDK for Multimodal Data*. Available from https://www.rerun.io/ and https://github.com/rerun-io/rerun. Online, 2024. URL: https://www.rerun.io (cit. on p. 66).