FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO




# Synthetic Data Generation for Automated Driving Software-In-The-Loop Testing with CARLA Simulator

**João Paulo Carvalho da Silva Alves**

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Rui Manuel Esteves Araújo

September 23rd, 2022

© João Paulo Carvalho da Silva Alves, 2022

# Abstract

As automated driving research and development (R&D) grows increasingly complex, so do the requirements of virtual testing and the role of simulated environments. Software-in-the-loop testing can enhance its effectiveness by leveraging diverse, realistic, and accurately labelled synthetic sensor data. The advancement of technology, namely in virtual environments, has reduced the gap between what can be perceived as real or not. With that in mind, it has become increasingly important to take advantage of such a leap in the last decade in video game engines and 3D virtual environments, to test simulated data for future real-world applications such as CAR Learning to Act (CARLA), a state-of-the-art open-source simulator platform built upon Unreal Engine, that supports extensive sensor specifications and procedural control over traffic dynamics and environmental conditions.

In this work, CARLA's synthetic data will be generated based on a created script following CARLA's Python API where the information is collected using a simulated RGB camera. This part will be crucial for demonstrating the reasoning of the approach taken with CARLA, as well as the features chosen to simulate in the server, the difficulties faced and the problem solving involved in order to have a stable simulation that not only could withstand long periods of data retrieval but also fully modular in order to easily adapt it when facing different demands of data output from the software that were utilised.

The data retrieved will be subject to visual odometry testing using simultaneous localisation and mapping (SLAM) technology, namely with ORBSLAM2, and lane detection using Continental's framework. As such, the preparation and organisation of the virtual data will be heavily based of the camera's intrinsic and extrinsic parameters, following the concepts involved of the pinhole camera model. The SLAM system, ORBSLAM2, will test the quality of the data provided by CARLA into two different groups of sequences, 500 meters to evaluate the feasibility of synthetic data on the SLAM algorithm and 2000 meters to stress test its effectiveness in larger trajectories. In the final stage, the data extracted from CARLA will be a video with a small duration to showcase the performance of the lane detection neural network of Continental when confronted with synthetic data.

# Resumo

À medida que a pesquisa e o desenvolvimento de condução autónoma se torna cada vez mais complexa, também aumentam os requisitos de testes virtuais e o interesse no papel dos ambientes simulados. A testagem de *software in the loop* pode aumentar a eficácia, ao aproveitar os dados de sensores virtuais na qual se apresentam como diversificados, realistas e rotulados com precisão. O avanço da tecnologia, nomeadamente nos ambientes virtuais, tem diminuído a diferença entre o que pode ser percebido como real ou não. Com isso em mente, tornou-se cada vez mais importante aproveitar o salto da última década em motores de videojogos e ambientes virtuais 3D, para testar os dados simulados para futuras aplicações no mundo real, como CAR Learning to Act (CARLA), uma plataforma de simulação *opensource* de última geração, construído no ambiente Unreal Engine 4, que oferece suporte a extensas especificações de sensores, controlo de procedimentos sobre a dinâmica do tráfego e até de condições no meio simulado.

Neste trabalho, os dados sintéticos do CARLA serão gerados com base num script criado seguindo a API Python do CARLA, onde as informações são colecionadas através de uma câmara RGB simulada. Esta parte será fundamental para demonstrar o raciocínio da abordagem realizada com o CARLA, bem como as funcionalidades escolhidas para simular no servidor da ferramenta, as dificuldades enfrentadas e a resolução de problemas envolvidos de forma a ter uma simulação estável que não só suporta longos períodos de processamento de dados, mas também que seja totalmente modular de forma a adaptá-lo facilmente diante diferentes desafios para retirar dados do software utilizado.

Os dados recolhidos serão sujeitos a testes de odometria visual com recurso à tecnologia de localização e mapeamento simultâneos, nomeadamente com o ORBSLAM2, e deteção de faixa da estrada com recurso ao framework da Continental. Assim, a preparação e organização dos dados virtuais será fortemente baseada nos parâmetros intrínsecos e extrínsecos da câmara, seguindo os conceitos envolvidos no modelo de *pinhole camera*. O sistema SLAM, ORBSLAM2, testará a qualidade dos dados fornecidos pelo CARLA em dois grupos diferentes, 500 metros para avaliar a viabilidade de dados sintéticos no algoritmo SLAM e 2000 metros para testar a sua eficácia em trajetórias maiores. Na etapa final, os dados extraídos do CARLA são transformados num vídeo de curta duração para mostrar o desempenho da rede neural da Continental na detecção das faixas da estrada quando confrontado com dados sintéticos.

# Acknowledgements

The completion of this thesis is the culmination of a long journey, which had the support and the contribution from several people, to whom I am grateful. I express my thanks to: To the teaching staff of Faculdade de Engenharia do Porto throughout the duration of my masters degree. To my supervisor, Prof. Rui Araújo for the availability shown in the process of preparing this document. To my classmates for all the support shown over the years, in particularly to Gonçalo Marinho dos Santos. This work was carried out with the support of the host institution: Continental Engineering Services and supervised at the institution by: Marc Bellinger. The following people collaborated with the supervising team in carrying out this work: Pedro Costa, Simão Pinheiro and Afonso Carvalho. Finally, to my family for their efforts to provide me with a better future, namely to my mother, my father and my grandparents.

João Paulo Carvalho da Silva Alves

*"I dream of a world where the truth is what shapes people's politics, rather than politics shaping what people think is true."*

Neil deGrasse Tyson

# Contents


**1 Introduction** **1**
- 1.1 Purpose of the thesis and its context . . . 1
- 1.2 Goals . . . 1
- 1.3 Structure . . . 2

**2 State of the Art** **3**
- 2.1 Machine Learning and the purpose of Synthetic Data . . . 3
- 2.2 Software-in-the-loop and virtual environments . . . 4
- 2.3 Existing Autonomous Driving simulators and toolboxes . . . 5
- 2.4 Comparison . . . 8
- 2.5 Car Learning to Act . . . 10
  - 2.5.1 SUMMIT, a CARLA based simulator . . . 11
- 2.6 Overview of sensors for Advanced Driver Assistance Systems and Autonomous Driving Functions . . . 11
- 2.7 Synthetic data modelling on virtual environments and on machine learning applications . . . 14
  - 2.7.1 Generative Adversarial Network (GAN) . . . 14
  - 2.7.2 Data Augmentation . . . 15
  - 2.7.3 Domain Randomisation . . . 15
  - 2.7.4 Object Detection . . . 15

**3 Learning and working with CARLA** **17**
- 3.1 Setup . . . 17
- 3.2 Core functionalities . . . 18
  - 3.2.1 The world and the client . . . 18
  - 3.2.2 The actors and their purpose . . . 21
  - 3.2.3 A more detailed overview of the available sensors and Data retrieval . . . 24
  - 3.2.4 The existing maps, their environments and the navigation involved . . . 27
  - 3.2.5 OpenDRIVE in CARLA . . . 28
  - 3.2.6 Recording simulations in CARLA . . . 29
  - 3.2.7 Rendering . . . 29
  - 3.2.8 Synchrony and time-step . . . 30
  - 3.2.9 Traffic Simulation in CARLA . . . 32
- 3.3 Creation of the main script . . . 35
  - 3.3.1 Configuration file . . . 36
  - 3.3.2 Sensor settings . . . 37
  - 3.3.3 Synchrony settings . . . 38
  - 3.3.4 Data retrieval and processing . . . 38

3.3.5 Singlethread to Multithread . . . 39
3.3.6 Weather . . . 40
3.4 Results and analysis . . . 42

**4 Odometry, an extra case of study** **45**
4.1 ORBSLAM2 . . . 45
4.2 Adaptation of the main script . . . 48
4.3 Evaluation Metrics . . . 51
4.4 Results . . . 54
4.4.1 Discussion of the 500 meters sequences . . . 54
4.4.2 Discussion of the 2000 meters sequences . . . 62
4.4.3 Final thoughts . . . 66

**5 Continental Framework** **67**
5.1 The software . . . 67
5.2 Visual results . . . 69
5.3 Extracting the output data . . . 69
5.4 Analysis . . . 72

**6 Conclusions and future work** **75**
6.1 Future work . . . 75

**References** **77**

**A Preliminary work and work-plan** **83**

**B ORBSLAM2 main script** **85**

**C Lane detection main script** **97**

# List of Figures


2.1 Demonstration of SIL with CARLA. . . . . 4
2.2 Image of CARLA, retrieved in [1]. . . . . 10
2.3 Existing oddities on the behavioural of cameras. From left to right: Barrel distortion, a type of distortion where lines that are straight in real life appear to curve inwards, Blur, reduces the quality of the image making it less sharp and Vignette, a reduction of an image brightness or saturation toward the periphery compared to the image centre. Images retrieved in [2]. . . . . 14
2.4 Convolutional Neural Network. . . . . 16

3.1 Project CARLA on PyCharm. . . . . 18
3.2 A simple flowchart, showcasing the very first steps during the creation of a test script based on CARLA's Python API. . . . . 19
3.3 The light differences of the map during different times of the day. . . . . 20
3.4 A showcase of the usage of debug.draw_string function. The green colour represents the center of the lane travelled by the testing vehicle, whilst the remaining colours represent the central points of the adjacent lanes . . . . . 20
3.5 CARLA's coordinates system which is the same as Unreal Engine. . . . . 20
3.6 A showcase of the retrieval of a blueprint and the life cycle of an actor. . . . . 21
3.7 The spectator point of view. . . . . 22
3.8 Traffic signs actors in CARLA, from left to right, stop, yield and traffic lights. . . 22
3.9 Model 3 of Tesla in CARLA. . . . . 23
3.10 A group of pedestrians in CARLA. . . . . 23
3.11 Typical life cycle of a sensor. . . . . 24
3.12 Group of sensor outputs, retrieved in [3]. . . . . 27
3.13 Scenario created by OpenDRIVE feature of CARLA. . . . . 29
3.14 Graphic presets available, left image is the Epic preset and the right image is Low preset. . . . . 30
3.15 Architecture of the TM. . . . . 35
3.16 An overview of the main script. . . . . 36
3.17 An example image to facilitate the understanding of the azimuth angle and altitude angle. . . . . 41
3.18 Images retrieved from the weather generation function. . . . . 42
3.19 Images retrieved from the main script. The bottom images are from the perspective of the sensor implemented, whilst the remaining were taken manually using the spectator. . . . . 43

4.1 ORBSLAM2 analysis frame by frame in search of match points. . . . . 46
4.2 ORBSLAM2 average performance based on KITTI's evaluation, retrieved in [4] . 46

4.3 ORBSLAM2 map viewer. Evolution from left to right of the surroundings and trajectory of the ego vehicle. . . . 47
4.4 Perspective of CARLA and ORBSLAM2. . . . 49
4.5 ORBSLAM2 axis reference. . . . 50
4.6 The green path corresponds to the trajectory based on CARLA's world and the red path is the corrected trajectory, following ORBSLAM2 perspective. . . . 51
4.7 KITTI's metrics, retrieved in [5] . . . 52
4.8 Resulting images of the evaluation script. . . . 53
4.9 Resulting output for Sequence01. . . . 56
4.10 Resulting output for Sequence02. . . . 57
4.11 Resulting output for Sequence03. . . . 58
4.12 Resulting output for Sequence05. . . . 59
4.13 Resulting output for Sequence06. . . . 60
4.14 Resulting output for Sequence04. . . . 61
4.15 Example of the surrounding environment of Sequence03. . . . 61
4.16 Example of the surrounding environment of Sequence04. . . . 61
4.17 Example of the surrounding environment of Sequence05. . . . 62
4.18 Resulting output for Sequence07. . . . 63
4.19 Resulting output for Sequence08. . . . 64
4.20 Resulting output for Sequence09. . . . 65
4.21 Resulting output for Sequence10. . . . 66
4.22 Example of the surrounding environment of Sequence10. . . . 66

5.1 Architecture of a CNN with a SSD detector. . . . 68
5.2 From left to right: a 4x4 grid; anchor boxes and bounding boxes. . . . 68
5.3 Visual results of the lane detection module. . . . 69
5.4 Detected points from Frame 0 by the lane detection algorithm. . . . 70
5.5 Collected waypoints from Frame 0. . . . 70
5.6 Axis conversion from UE to OpenCV standard. . . . 71
5.7 Final result of Equation 5.3 being painted over Frame 0 with the help of OpenCV. 72
5.8 Final group of points that will be used to analyse Continental's framework performance on CARLA's dataset. Upper image is from the Continental framework and lower image is from CARLA. . . . 73
5.9 Showcase of the filled ego lane. The upper image is from the lane detection algorithm and the lower image is from CARLA. . . . 74

A.1 Gantt diagram. . . . 84

# List of Tables

2.1 Comparison of functionalities between the simulators. . . . . 9

3.1 Comparison of the different predefined maps of CARLA. . . . . 28
3.2 The different results of synchrony and time-step. . . . . 31
3.3 The options available for traffic simulation in CARLA. . . . . 34

5.1 Average IoU of each group of frames and total average. . . . . 73

# Symbols and abbreviations

| | |
|---|---|
| CARLA | CAR Learning to Act |
| HIL | Hardware-in-the-loop |
| SIL | Software-in-the-loop |
| KPI | Key Performance Indicator |
| VTS | Virtual Testing Simulation |
| ADAS | Advanced Driver-Assistance Systems |
| API | Application Programming Interface |
| NPC | Non-Player Character |

# Chapter 1

# Introduction

## 1.1 Purpose of the thesis and its context

In the last years, not only the concept of automated cars has been rapidly growing, but also its application in the field. It does not only revolutionise the car industry, it also changes the core of what the driving task is. Since most car accidents are due to human error [6], the implementation of autonomous driving may in future reduce the casualties all around the globe by car accidents. Thankfully, in most cases, human rationality and interpretation of the surrounding environment helps to prevent many accidents from occurring, so it is important to also automate what most drivers do right, risk-free. Various European projects have been made in order to increase the safety of our roads (The European Integrated Project "SAFESPOT" [7]), the economic growth (affordable solutions for improved road safety with low cost advanced driver assistance systems [8]) and the continued innovation of this area (Vision Inspired Driver Assistance Systems, VI-DAS [9]).

The “Framework of Automated Driving System Testable Cases and Scenarios”, a document issued by the National Highway Traffic Safety Administration (NHTSA), a branch of the U.S. Department of Transportation [10], describes the stages of autonomy applied to the driving itself. These stages are represented by the separation of five levels. Current research being done involves the leap from Level 2 (Combined function automation) to Level 3 (Limited self-driving automation) and how the application of simulators and software-in-the-loop environments could help in supplying reliable synthetic information. This advancement in technology would demand much more simulation to efficiently test and verify the enormous amount of functional requirements.

## 1.2 Goals

The main goal for this work is to validate the use of CARLA open-source simulator into a software-in-the-loop testing framework and generate synthetic datasets based on the development of sensors and procedural control over the simulated environment. This thesis will be done in collaboration

with Continental Engineering Services, alongside the Virtual Testing Simulation (VTS) team. As such, it has been established the following set of objectives for the project:

- Contextualisation over virtual environments usage in autonomous driving, mainly on CARLA.
- Generate large-scale synthetic datasets with full automated environmental labelling.
- Apply synthetic sensor data to Continental software-in-the-loop testing framework.
- Validate the use of synthetic data in comparison to real data using driving function Key Performance Indicators (KPI) in both.

For a better perspective of the road map to the established objectives, a more detailed work plan can be seen in the appendix A.

## 1.3 Structure

Besides the introduction, this report will contain the state of the art in the second chapter. The work developed on CARLA is in chapters three, four and five. Finally the conclusion of the document will be present in the sixth and final chapter.

# Chapter 2

# State of the Art

In this chapter, it will be discussed the state of the art of synthetic data, software-in-the-loop testing, existing simulators/virtual environments software, an overview of the current sensors utilised in autonomous driving and finally the existing synthetic data modelling methods.

## 2.1 Machine Learning and the purpose of Synthetic Data

Instance segmentation is a popular computer vision task and a difficult one since its objective is the detection of an object of interest and individually delineating it in a given image. Therefore, it is not surprising that the usage of machine learning is essential for this type of task. However, with the exponential growth of machine learning after the rise of deep learning, the current state-of-the-art models are becoming increasingly impractical [11]. That is, as long as more complex tasks are being demanded, the need for larger datasets also increases which means more time spent on labelling, therefore more money spent. But it does not end there, when the system needs something beyond what is given to him, it reaches a deadlock and to solve it, it requires even more data to be disposed to the system, increasing the cost and the time of the operation. In the context of instance segmentation that means, for example, if a given machine learning model has the purpose of detecting human faces, once it knows the conditions and classes of a given dataset it will take a while to train it, but if one decides to add up even more conditions that the current model is not prepared to handle (e.g. different light angles or different face rotations), it will demand even more labelled examples, increasing its cost and time-consumption. To sum it up, machine learning models are becoming data-intensive and data expensive which, in turn, calls for more efficiency [11].

To reduce the cost of labelling data, many companies use what is called data augmentation. This usage allows the "transformation of the input data that changes target labels in predictable ways" [11], in other words, it is a group of "techniques used to increase the amount of data by adding slightly modified copies of already existing data" [12]. Some authors consider it the first step to synthetic data [11] since it is an adaptation of existing data and the results become quite "synthetic".

So what is synthetic data? Synthetic data is a type of data augmentation, in which data is artificially manufactured rather than being created by real-world events. As some types of data become expensive to gather or are rare (e.g. real-world road events for an autonomous vehicle), synthetic data could in theory help solve the problem that machine learning models are currently experiencing. For instance, synthetic data can be created, with software generators, and used in machine learning while the training is in motion, removing the need to store huge datasets [11]. Of course, one could ask, how reliable can synthetic data be compared with real-world data? In 2017, a Massachusetts Institute of Technology article [13] was published showcasing an experiment done on four groups consisting of a total of 39 data scientists. Their task was to develop predictive models in a crowd-sourced experiment, in which one group was offered the original datasets, while the remaining three were given the synthetic versions. In the end, when all the solutions were compared the difference in performance was not substantial, since only 4 out of the 15 tests showed a significant performance deviation. Finally, the paper suggests that synthetic data can be successful in replacing real data, indicating that there is no significant difference in the work produced by data scientists who used synthetic data as opposed to real data.

## 2.2 Software-in-the-loop and virtual environments

Taking into account the Figure 2.1, let us assume you are a software engineer at a car manufacturing company and have developed a software function that you want to test. One option is to connect the electronic control unit (or any other physical component) to a simulator that simulates your car. This is called hardware-in-the-loop (HIL) testing. Another favourable choice is the idea of software-in-the-loop (SIL) testing where only the software component is applied to a virtual environment (e.g. simulator) for testing with the implementation of additional objects/noise in the virtual environment, for example, vehicle simulation and traffic simulation. In order to close the loop (feedback loop), the simulation can send, for instance, the position of the vehicle in response of commands, such as acceleration.

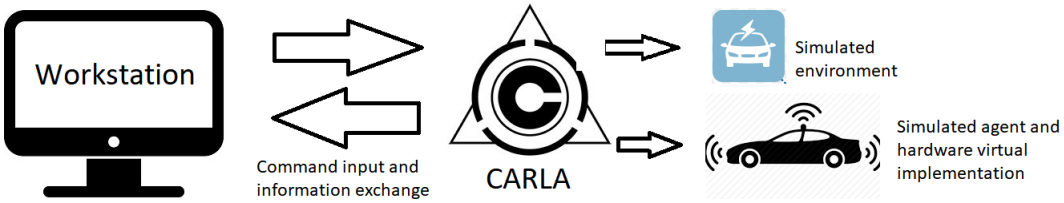



Figure 2.1: Demonstration of SIL with CARLA.

Today, there are multiple simulated environments readily available for developers, from aerial navigation and robotic simulators to indoor and outdoor environments. The latter is the one which this work will be based upon, specially in outdoor urban environments. About outdoor simulated environments for urban cities, various frameworks can be utilised in the context of SIL. The SYNTHetic collection of Imagery and Annotations (SYNTHIA) dataset is one example that consists of a collection of images rendered from a virtual environment (e.g. city) with precise semantic

annotations [14]. Another alternative is Grand Theft Auto 5 (GTA 5) dataset [15] and the various datasets that are based upon the videogame: Precise Synthetic Image and LiDAR (PreSIL), Visual Perception (VIPER) and GTAVision for instance [11][16][17][18]. This paper will however focus on an open-source simulator based on Unreal Engine 4, CARLA [19].

## 2.3 Existing Autonomous Driving simulators and toolboxes

Currently, there are multiple tools and simulators that Advanced Driver-Assistance Systems (ADAS) could benefit from, therefore it is important to differentiate the functionalities of each.

### MATLAB/Simulink

MATLAB/Simulink provides the Automated Driving Toolbox, a package that is not open source, which allows the test of various functionalities namely perception, path planning and vehicle control. In addition, it provides users the chance to develop photo-realistic 3D environments, model and implement sensors and the usage of a built-in visualiser in which the user can view live sensor detection. It has additionally included a *Ground Truth Labeler* application, that automates the labelling of ground truth to train and evaluate perception algorithms. This toolbox additionally supplies reference applications as a basis for developing automated driving functionality, such as forward collision warning, lane keeping assist and automated parking valet [20].

### CarSim

Established in 1996, Mechanical Simulation Corporation provides car, truck and motorcycle simulation packages, one of them named CarSim frequently used by industry. It works as a standalone application, therefore not requiring any other software to perform the simulations. On the other hand, it has a standard interface to other software programs like MATLAB/Simulink and NI LabVIEW. This product allows the implementation of moving objects such as traffic and their customisation, allowing them to have their own embedded animations, like walking pedestrians and pedalling bicyclists. In terms of sensor implementation, CarSim supports up to 99 ADAS sensors however, an extended license is needed to employ them. Each sensor occupies a location established in a part of the simulated vehicle, with a designed aiming direction and sensitivity or, they may be placed in moving objects. Finally, it has the VS Scene Builder Tool in which allows the user to manually construct scenarios for testing [20].

### PreScan

PreScan supplies a framework to simulate and design ADAS and autonomous driving vehicles, allowing manufacturers to evaluate their intelligent systems by producing a diversity of virtual traffic conditions and lifelike environments, put to good use by PreScan's automatic traffic generator. It allows users to customise the sensors suites, control logic and collision warning features. This framework grants the integration of HIL simulation, a regular procedure for assessing Electronic Control Unit (ECU). PreScan is proficient at physics-based calculations of the sensor inputs which, in turn, are integrated with the ECU to test multiple algorithms. Another interesting functionality, is that the signals can also be sent as output to the loop or camera HIL for the driver.

Furthermore, PreScan offers a feature named Vehicle hardware-in-the-loop (VeHIL) laboratory, which allows users to merge and create real and virtual system where the test vehicle is placed on a roller bench and other vehicles are depicted by wheeled robots with an appearance similar to cars. By combining the test vehicle equipped with realist sensors and mobile robots, VeHIL is therefore capable of providing an extra step up on detailed simulations for ADAS.

Finally, it also supports real-time data and GPS vehicle data recordings which is very helpful for situations which are otherwise not easily simulated with synthetic data [20].

### Gazebo

Whilst the previous software were not open source, Gazebo is a scalable, flexible and multi-robot 3D simulator that is of unrestricted use, supporting multiple operating systems such as Linux and Windows. Additionally, although Gazebo is a standalone simulator, it is typically used in conjunction with ROS, allowing the support of modelling different kinds of robots. This software in addition introduces the flexibility of creating indoor or outdoor 3D environments for testing.

The framework of this software relies on three libraries, the physics, the rendering and the communication library. Starting with the physics library, it permits the simulated objects to act realistic in comparison to their real counterparts by allowing the user to specify their physical properties like mass, friction coefficient, velocity and inertia. The default physics engine is Open Dynamic Engine (ODE) but it also tolerates others like Bullet, Simbody and Dynamic Open Source Physics Engine (DART). The rendering library, uses what is called Object-oriented Graphics Rendering Engine (OGRE), which offers the possibility to visualise dynamic 3D objects and scenes. Finally, the communication library allows the interaction between the various elements of Gazebo.

In this 3D simulator, the world file (which represents the 3D scene) is populated with one or multiple models like static objects (a table or a sensor) or dynamic objects. The users can produce objects from the ground up, by defining their visual and physics properties in a Simulation Description File (SDF) format. Alternately, they can specify plugins to conduct the simulation such as a model plugin that controls the model properties. Ultimately, Gazebo has a wide community support with multiple tutorials, documentation and the possibility of sharing and using models already created by different users [20].

### LGSVL Simulator

Developed by LG Electronics America R&D Center, LGSVL is a multi-robot AV simulator that is open source and developed based on the Unity game engine. The simulator leverages the Unity his game engine to generate photo-realistic virtual environments based on High Definition Render Pipeline (HDRP) technology. The engine supplies different functionalities for simulating, namely in the environment, sensor simulation, and vehicle dynamics. In addition it provides a Python application programming interface (API) to control different entities, the customisation of a set of sensors via setting up JSON file that enables the user to specify the intrinsic and extrinsic parameters.

It currently supports cameras, LIDAR, IMU, GPS, radar and the customisation of them, as well various options for instance segmentation and semantic segmentation. The program additionally provides Functional Mockup Interface (FMI) so that it integrates vehicle dynamics model platform to the external third-party dynamics models. Finally, one of the prominent features of LGSVL is exporting HD maps from 3D environments [20].

### DeepDrive

Deepdrive is an open source autonomous car simulator, with a Massachusetts Institute of Technology license. It has some sensor data which can be used for the creation of path-following tasks, using supervised learning or reinforcement learning on autonomous cars. It has compatibility with Gym and Docker. The Docker implementation uses a container for the environment, per client. For the Gym implementation, it may be employed for verifying the validity of changes to their environment, such as changing the reward function for the system.

The program includes a few sensors, namely an RGB camera, a GPS, an inertial measurement unit, IMU, and some location sensors for staying in the centre of the lane and following a specified path. Finally, as stated earlier, the focus of this project seems to be on developing both supervised learning and reinforcement learning models by performing tasks like lane changing, centring the vehicle in the lane, following a path, and generalised autonomous vehicle development [21].

### TORCS

The Open Racing Car Simulator (TORCS) is an open source car racing simulator which provides a sophisticated physics engine, full 3D visualisation, several race tracks and models of vehicles and alternative game modes. The software is composed by three components: the racing simulator, the Python controller and the machine learning which can or not be included.

The racing simulator condenses all the functions about the simulated physics, graphical representation and user interface. The vehicle dynamics are accurately simulated, and the physics

engine takes into account many aspects of the racing cars like traction, aerodynamics and fuel consumption. There are also the racing participants named "bots" that the user can compete with. At each control step, a robot can access the current game state, which includes several information about the vehicle, the race track as well the information about the other cars on the competition. It also can access the control of the car using the gas/brake pedals, the gear stick, and steering wheel.

Finally, it is meaningful to refer that the machine learning algorithms can be developed outside of the platform such as training and data processing being developed in Matlab [22][23].

## 2.4 Comparison

This section brings forth a comparison of the different simulators described previously in section 2.3, with the display of Table 2.1. The simulators will be compared under certain requirements used in [20], such as:

**Open Source**: easier collaboration, collective progress and incorporation of shared projects in the corresponding domain;

**Sensor models supported**: A vital component of self-driving cars, the ability to perceive its surroundings is significantly important. The hardware available must have a vast extension, ranging from camera to LiDAR, ultrasonic sensor, radar, Global Positioning System (GPS) and Inertial Measurement Unit (IMU). To effectively evaluate the system of perception, the software utilised must have sensor models implemented or be able to support input data stream from real sensors;

**Support for different weather conditions**: Modification of the virtual environment is important so that the developer possesses a more varied and robuster data for testing;

**Camera Calibration**: Simultaneous Localisation and Mapping (SLAM) is the computational problem of constructing or updating a map of an unknown environment and tracking simultaneously the agent location within it. To support SLAM applications, the software must supply camera calibration;

**Path Planning**: Planning a path for a mobile agent so that it can move around autonomously without disrupting its surroundings. The simulator should possess a built-in function to develop maps or to import from the outside and the interfaces to program customised algorithms;

**Support for proper vehicle dynamics**: To execute the predicted trajectory, the simulator must have control inputs such as throttle, brake and steering monitored by closed loop control algorithms such as Proportional-Integral-Derivative (PID). Therefore, it is crucial for the developer to be capable to customise vehicle dynamic models and programming the algorithms;

**3D Virtual Environment**: The level of detail of a given simulator may or may not be important depending on the purpose of a project. However, in an effort to test various functional elements of a vehicle such as perception system, the photogenic view of the virtual world must have a diverse set of static objects such as trees, buildings and dynamic objects like various types of vehicles, pedestrians and animals that need to behave realistically. Finally, to achieve graphical fidelity, the simulators can rely on either game engines or High Definition (HD) maps;

**Traffic Infrastructure**: In order to regulate the existent traffic for the safety of all road users in the simulation, it is important for a simulator to possess traffic aids like traffic lights and road signals;

**Traffic Scenario simulation**: In order for the developer to test various "what if" scenarios, it is significant for the software to provide a variety of traffic agents, mechanisms to generate them and the availability to support a significant number of these dynamic agents;

**2D/3D Ground Truth**: For the training of the AI models, the simulator should provide object labels and bounding boxes of the models appearing on-screen.

Table 2.1: Comparison of functionalities between the simulators.

| Requirements | MATLAB/ Simulink | CarSim | PreScan | Gazebo | LGSVL | Deepdrive | TORCS |
|---|---|---|---|---|---|---|---|
| Open Source | No | No | No | Yes | Yes | Yes | Yes |
| Sensor models supported | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| Support for different weather conditions | No | No | Yes | No | Yes | Unknown | No |
| Camera Calibration | Yes | No | Yes | No | No | Unknown | Unknown |
| Path Planning | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| Support for proper vehicle dynamics | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| 3D Virtual Environment | Unknown | Yes | Yes | Yes, Indoor and Outdoor | Yes, Outdoor (Urban) | Yes | Yes, Outdoor |
| Traffic Infrastructure | Yes, Traffic lights model | Yes | Yes | Yes, all kinds of models | Yes | Yes | No |
| Traffic Scenario simulation | Yes | Yes | Yes | No | Yes | Yes | Yes, Racing |
| 2D/3D Ground Truth | Yes | No | No | Unknown | Yes | Yes | No |

It is noteworthy to state some observations on certain simulators. Starting with MATLAB, the software is good at computation, has efficient plot functions and allows the cooperation with other types of software such as CarSim. It is frequent to observe vehicle models from CarSim being developed alongside MATLAB/Simulink on their control algorithms. However, MATLAB/Simulink has limited ability to realistically visualise traffic scenarios, obstacles and pedestrian models.

In comparison with the previous software, PreScan possesses a bigger capacity of simulating its virtual environment, with the inclusion of weather conditions. PreScan developers can also combine it with MATLAB/Simulink so that the modelling process becomes more efficient.

On the other hand, Gazebo is known for its high flexibility in its simulation and its integration with ROS. Another advantage of Gazebo is the inclusion of various sensor models and the capability to create customised sensor models via the plugins. However, by allowing the absolute control over its simulation, the usage of this type of software comes with a cost of time and effort, in which the user must manually create the simulated world, as well its models and physics.

The LGSVL simulator provides high-quality simulation environment that require a capable GPU in order to run at a reasonable performance and frame-rate, therefore a powerful workstation

is needed. It gives a variety of sensors that can be manually enhanced by its users. The backbone of LGSVL simulator is the Unity game engine allowing the user to manually import unique components for its map generation.

## 2.5 Car Learning to Act




Figure 2.2: Image of CARLA, retrieved in [1].

Car Learning to Act (CARLA) (Figure 2.2) is an open-source simulator, built over Unreal Engine 4, that is known for its accessibility for autonomous driving research, serving as a modular and flexible tool that supports training and validation of ADAS systems. The engine provides state-of-the-art rendering quality, physics, basic non-player character (NPC) logic and an ecosystem of inter-operable plugins. To support this functionality, CARLA is designed as a server-client architecture based on Python and C++ source libraries. The server performs the simulation and renders the scene, so in order to have realistic results and good performance, it is recommending a dedicated Graphics Processing Unit (GPU) to run alongside it. The client API is implemented in Python and it has the responsibility for the interaction between the autonomous agent and the server via sockets. The user can, if it is convenient for him, work with C++. The client can send commands, and meta-commands to the server and in response, receive sensor readings. What is important to refer to is that the developer has absolute control over the simulation, making CARLA a very flexible environment to work with [19][20].

The environment in which the simulation is based can be fully customised as referenced in [19], where the authors created two towns, one based for training with 2.9 km of roads and the other for testing with 1.4 km of roads. They also implemented realistic behaviour for the NPCs such as lane following, respecting traffic lights, speed limits and others. For further fidelity, each NPC had a unique combination of clothes, occasionally equipped with a guitar case, a suitcase

or an umbrella. It was also implemented a variety of atmospheric conditions and illumination regimes. The vehicles, buildings and urban layouts are open digital assets that CARLA provides.

The implementation of sensors is also possible for the agent, allowing the use of red, green and blue (RGB) cameras and pseudo-sensors to provide ground-truth depth and semantic segmentation (different in comparison to instance segmentation, in which the whole class is labelled as a group instead of an individual, generally easier to implement). The implementation of the semantic classes is relatively straightforward in the project mentioned: road, lane-marking, traffic sign, sidewalk, fence, pole, wall, building, vegetation, vehicle, pedestrian and others. In addition, CARLA additionally provides a range of measurements associated with the state of the agent and compliance with traffic rules, for example, vehicle orientation, speed, acceleration vector and accumulated impact from collisions.

### 2.5.1 SUMMIT, a CARLA based simulator

SUMMIT is a software based on CARLA that leverages the high-fidelity physics, rendering and sensors in a python-based API. This software uses real-world maps retrieved from online references to allow an unlimited source of complex environments. It repopulates it automatically, by generating crowds composed by different traffic agents with sophisticated and unregulated behaviours.

The crowd simulation algorithms extend an already established framework, GAMMA, that simulates the traffic agents with different geometry, kinematics and behavioural types in a unified velocity-space framework. In SUMMIT, it will also encode topological road contexts such as lanes and pedestrian sidewalks to be as faithfully as possible compared to the real world. This extended version is named Context-GAMMA.

Compared to CARLA, SUMMIT focuses [24] on transferring and implementing real world maps to the 3D scenarios, create unpredictable/unregulated behaviours on the agents, allow denser traffics and continuous support of realistic visuals and sensors. The maps are fetched from OpenStreetMap. It constructs two topological graphs, a lane network for vehicles and a sidewalk network for pedestrians, that will form a representation of the road contexts. Then their behaviour model, Context-GAMMA, will take as an input.

## 2.6 Overview of sensors for Advanced Driver Assistance Systems and Autonomous Driving Functions

Sensors constitute an integral part of ADAS function development and are quite numerous. Their functioning is based on the following tasks: sense, plan and act [25]:

- Sense, because sensors provide information about the surrounding environment, in which the vehicle moves, to the planning unit;
- Plan, which based on sensor inputs, the planning unit calculates the action of the vehicle;

- Act, it considers the result of the planning unit by applying the ego-vehicle moves in the environment accordingly.

A typical classification, based on the intended complexity and application, divides sensor models into three categories: ideal, probabilistic and physical.

Ideal sensor models are often described as sensor models without errors. They detect every object in their specific field of view (FOV) correctly, without false positives or false negatives. Object parameters that influence the perception process and disturbing environmental conditions, like unfavourable weather conditions, are not considered. Input is typically a ground-truth object list and output is a sensor specific object list. Ideal models can also be used for producing ideal sensor raw data [25][26].

Probabilistic sensor models, also referred to as stochastic or data-based, describe uncertainties like weather conditions by adding detection probabilities in comparison to ideal sensor models. It establishes a probabilistic relationship $p$ between sensor output and the given ground-truth. Sensor effects, error sources and the influence of environmental conditions are modelled by means of observations and statistics. Input and output are either object lists or sensor raw data [25][26].

Physical sensor models generate sensor raw data based on physical and mathematical descriptions of the measuring principle and information processing. It includes simulation of electromagnetic wave propagation, interference, diffraction or optical lens properties. Ray-tracing and rendering are popular methods for physical sensor models. Input for this model type is typically a 3D environment including a detailed description of environmental conditions as well as material parameters. However, physical laws can also be applied on sensor raw data as post-processing step. Sensor models might also include object abstractions. This is particularly useful for modelling radar target lists. A vehicle might be represented in a radar sensor model by a few scattering centers or a number of points and plane reflectors. Physical models are the most detailed models [25][26].

### Radar

Radar is a type of sensor that has its own signal source, therefore it is designated as a active sensor [25]. As source, it has one or multiple patches of antennas that are used to emit electromagnetic waves. The automotive radar sensors use Frequency-Modulated Continuous Wave (FMCW) radar to get the information about radial distance $r$ and relative radial velocity $v_r$. The radial distance can be correlated to the time shift $\Delta t$ between the emitted and received FMCW signal, in which $c$ is the speed of light as it is showed in equation 2.1.

$$r = \frac{\Delta t \cdot c}{2} \tag{2.1}$$

The relative radial velocity can be calculated by using frequency shift (Doppler frequency $f_d$) between the emitted and received signal, where $f_c$ is the frequency of the sent signal, showcased in equation 2.2.

$$v_r = \frac{f_d \cdot c}{2 \cdot f_c} \tag{2.2}$$

In Continental Engineering Services there is a wide range of radars to select, however as stated in [25], current state of the art in automotive radar sensors operate with signals of 77 GHz and the 24 GHz operating frequency are being phased out in 2022. This reduces the sample to a few sensors such as: ARS 4-A, ARS 3XX series and ARS 404-21. The radar sensor ARS 404, in particular, has already been subject of research on [26] for the creation of a sensor model for virtual testing.

### LIDAR

LIDAR are also active sensors based on the Time-of-Flight principle (TOF) which a method of measuring the distance between a sensor and an object. A pulse is sent out and reflected at a surface, and the reflection is received by the detector. It is worth mentioning that besides the TOF principle some LIDAR sensors work with modulated continuous waves like the FMCW.

For calculating the distance between the sensor and the surface reflection, the time flight $\Delta t$ between transmitting and receiving the signal is measured, The principle of the equation 2.1 stands the same for this case, in which the distance needs to be divided by two since it comes doubled from source to target and target to source. Automotive LIDAR can be used to extract velocity information of a target as well. Contrary to radar sensors, the Doppler-effect cannot be easily exploited by LIDAR sensors. Therefore, the velocity is calculated by tracking a fixed object point over time.

To conclude, LIDAR can provide a very high resolution in comparison with radar. The range performance, however, depends on the intensity of the emitted light, as well on atmospheric attenuation. Therefore, automotive LIDAR sensors are limited by eye-safety requirements, adverse weather conditions and sunlight. During dusk and dawn, the infrared percentage of the daylight is high compared, so filters are used to minimise the influence of ambient light sources [25].

### Camera

Camera is a passive sensor because it requires an external light source. Light is reflected from different surfaces in the surrounding environment and detected as a digital signal by its sensor chip, in a array of elements called pixels, transforming the 3D environment onto a 2D plane. Camera sensors are monochromatic sensors, therefore, a Colour Filter Array (CFA) is located in front of the sensor to filter specific light components per pixel. In the next step, a RGB image is generated out of the information from the raw image and the applied CFA of the camera. The red pixels in particular are essential to differentiate between headlights and back lights of vehicles [25].

To provide distance measurements by using cameras, a second one has to be added. The stereo cameras extract two different 2D projections of the 3D environment and with that information the

distance of objects can be calculated by using triangulation.

Camera quality suffers in adverse weather conditions and at night, but the use of such sensors excel at object classification and lane detection for instance. For future implementation in virtual environments, it will be important to simulate such anomalies [2] during testing such as: optical distortion, blur and vignetting (Figure 2.3).

Currently there are various Multi Function Camera (MFC) sensors available at Continental. Virtual testing and validation has been already documented, taking into account the previous camera behavioural oddities [2].

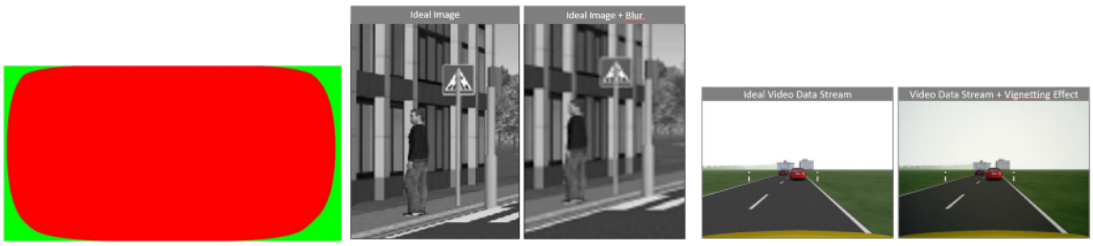



Figure 2.3: Existing oddities on the behavioural of cameras. From left to right: Barrel distortion, a type of distortion where lines that are straight in real life appear to curve inwards, Blur, reduces the quality of the image making it less sharp and Vignette, a reduction of an image brightness or saturation toward the periphery compared to the image centre. Images retrieved in [2].

## 2.7 Synthetic data modelling on virtual environments and on machine learning applications

The discrepancy between the real world and computer generated environments is usually referred as the reality gap [27]. In order to reduce the gap, it has been suggested two possibilities: increasing the similarity between the two domains or implement methods, such as data augmentation and domain randomisation, that are trained in a more generic domain, representative of both simulated and real domains simultaneously.

### 2.7.1 Generative Adversarial Network (GAN)

GAN is an approach to generative modelling by using deep learning methods, such as convolutional neural network (CNN). It is an unsupervised learning task that involves automatically discovering and learning the patterns in input data in such a way that the model can generate new examples that are believable to be drawn from the original set of data.

GANs are used for training a generative model by framing the problem as a supervised learning problem with two sub-models. The generator model that trains the generation of new examples and the discriminator model that tries to classify examples as either real (from the domain) or fake (generated). The two models are trained together until the discriminator model is fooled about half the time, meaning the generator model is generating plausible examples.

GANs are a rapidly changing field, delivering in their ability to generate realistic examples across a range of problem domains, most notably in image-to-image translation tasks such as translating photos of summer to winter or day to night, and in generating photo-realistic photos of objects, scenes, and people that even humans cannot tell are fake [28][29].

### 2.7.2 Data Augmentation

Data augmentation has been briefly described in section 2.1. There are two known methods for data augmentation, offline augmentation and online augmentation. The first, it performs all the necessary transformations beforehand, increasing the size of the dataset. Due to its increase of size this method is recommended for smaller datasets. The second option, is a method more inline with larger datasets. Since it is not possible to afford the explosive increase in size, the transformations are performed on mini-batches that would be fed to the machine learning model. There are also multiple augmentation techniques such as flipping images horizontally and vertically, rotating, scaling outward or inward, cropping, translation along Y or/and X axis and the addition of Gaussian noise. Conditional GANs are also utilised for the creation of even larger datasets just by modifying the existing information [30].

The application of data augmentation in autonomous driving has already been implemented for instance in [31], in which the dataset is enhanced by the introduction of synthetic cars upon the images, increasing the performance of their machine learning model and, in a way, showcasing a combination of synthetic data with real data.

### 2.7.3 Domain Randomisation

Domain randomisation (DR) is a technique for generating training data for machine learning algorithms with the purpose to synthetically create or improve the data, through the introduction of random variances in the environment properties that are not essential to the learning task. For instance, the application of random elements in a virtual environment, as referenced in [32], could be the irregular introduction of 3D models of objects of interest in different positions/orientations and the randomisation of geometric shapes that are to be ignored in the learning process. However, the effectiveness of domain randomisation may become unstable if the amount of randomisation in the objects is inflated in such a way that the distribution is not on par with real world test datasets as shown in [33]. Therefore, it is important to strike a balance in order to not jeopardise the performance of a machine learning model.

To conclude, it is anticipated that, with sufficient diversity in the simulation, the conversion between simulated and real domains will only be regarded as a small disturbance, whilst empowering the network to achieve a more complex behaviour [27].

### 2.7.4 Object Detection

A convolutional neural network (CNN) (Figure 2.4), is a type of neural network that uses convolution, a operation where a filter is applied across small patches on the input layers to extract

features. The result is then sub-sampled and the output is fed into another convolutional level or fed into a neural network [34].

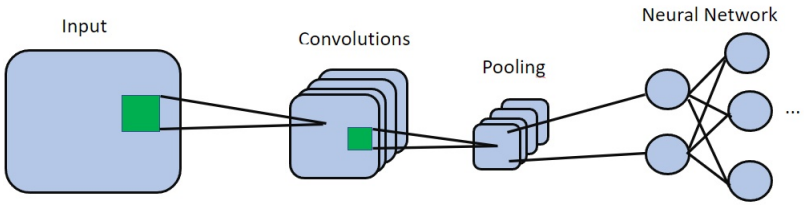



Figure 2.4: Convolutional Neural Network.

There are multiple variations of it, from dilated fully convolutional network (DFCN) applied for synthetic data generation in virtual environments [35] and in Region Based: Faster R-CNN and R-FCN [32], also applied under synthetic data applications. Finally, the Single–Shot Detector (SSD) approach is based on a feed-forward convolutional network that produces a fixed-size collection of bounding boxes and scores for the presence of object class instances in those boxes. The early network layers are based on a standard architecture used for high quality image classification named the base network. It is then followed by a non-maximum suppression step to produce the final detection [36].

# Chapter 3

# Learning and working with CARLA

This chapter will present the setup of CARLA, the work environment, the existing functionalities of the software, the issues faced and the thought process applied for the creation of a baseline framework that may enable future adaptations.

## 3.1 Setup

As stated before, the engine that CARLA uses to simulate, not only it's physics but the graphical aspect as well, is called Unreal Engine 4 (UE4). Thus, CARLA is a performance intensive software which, at the very least, requires a 6 GB GPU (or better) dedicated GPU. Currently, UE4 recommends a minimum of 16 GB of RAM, a Nvidia GTX 1060 graphics card, a solid-state drive (SSD) and a CPU with at least 6 threads [37]. The requirements allowed for not only the setup of CARLA on the workstation at Continental but also on my personal computer, for a more productive work and flexibility in the early stages when the workstation itself was being arranged.

- **Personal Desktop:**
    1. CPU: AMD Ryzen™ 7 2700X
    2. GPU: Nvidia RTX 2060 6 GB
    3. RAM: 16 GB
- **Continental's Workstation:**
    1. CPU: Intel(R) Core(TM) i7-10850H
    2. GPU 1: Nvidia Quadro T1000 with Max-Q design
    3. GPU 2: Nvidia RTX 3090 12 GB
    4. RAM: 32 GB

After the review on the hardware requirements of CARLA, it was the time to prepare for the software installation, namely downloading CARLA's project on GitHub (Version 0.9.13) [38],

Python version 3.7.9 and importing the module *carla* [39]. After those initial steps, the programming tool utilised throughout the project was PyCharm [40], which allowed to set up a virtual environment. With the work environment all set, as shown in Figure 3.1, it arrived the moment to learn the core concepts of CARLA, in the following sub-chapter.

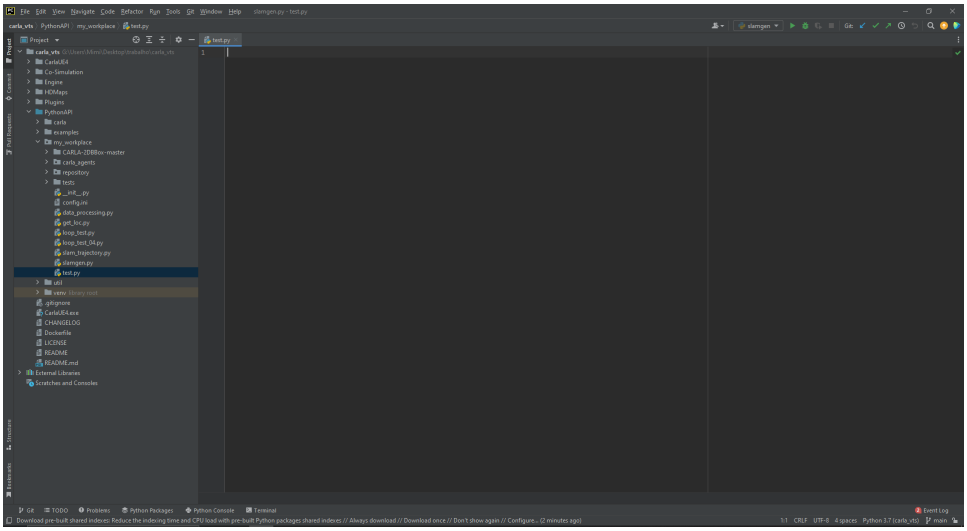


Figure 3.1: Project CARLA on PyCharm.

## 3.2 Core functionalities

This section will serve as an introduction to the core functionalities of CARLA. The information that will be presented in the following sub-chapters was retrieved in CARLA Docs [41].

### 3.2.1 The world and the client

Clients are one of the main elements in the CARLA, since they connect to the server, command changes and retrieve information via Python scripts, allowing to access advanced CARLA modules, features and apply command batches [42]. A basic *modus operandi* of a Python script based on the API from CARLA is presented in Figure 3.2.

By default, it is employed the local host IP and port 2000 to connect with CARLA, but it can be changed by the user. CARLA allows to work with many clients, if there are multiple scripts that need to be running, however, when working with more advanced modules of CARLA, communication will be more complex and caution is advised.

Once the communication between the client and server is presented, it is always recommended to set a time-out so that by limiting all networking operations, these won't block the client forever and an error is returned if connection fails. The following step is for the client to retrieve the world. For that, the user can retrieve the current one present by the server or load a different one if desired. By loading the world and retaining it saved in a variable, the user has a bigger control of

what happens inside the simulation with the usage of multiple *get* methods which help to access, for example, the number of actors in a scenario (*carla.World.get_actors(None)*).

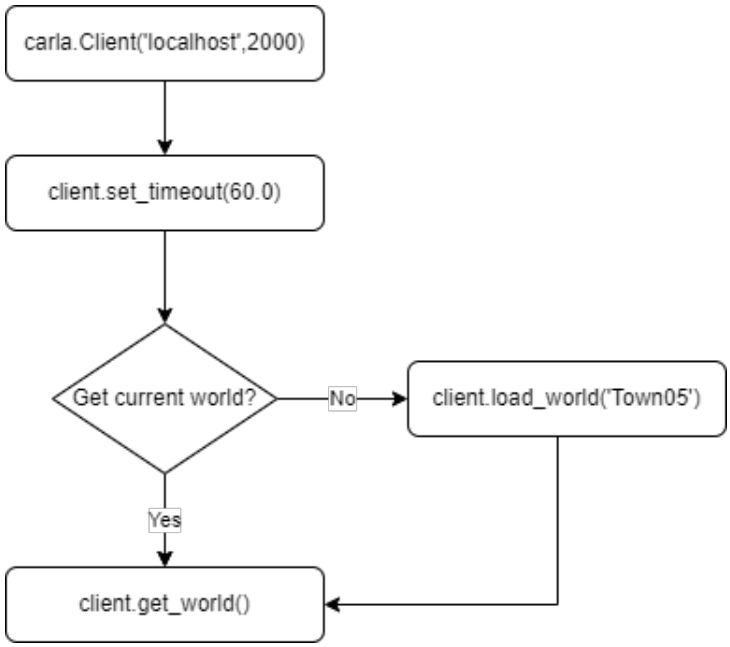



Figure 3.2: A simple flowchart, showcasing the very first steps during the creation of a test script based on CARLA's Python API.

Thus, it is pivotal to get accustomed to this class, *carla.World*, as it is a starting point to access most of the information and general settings:

- **Weather**: The weather contains a set of parameters accessible from the world, which includes sun orientation, cloudiness, wind, fog and much more. There are in addition some weather presets that can be directly applied to the world.

- **Actors**: Spawn actors, get every actor on the scene or one specific, access the spectator actor and much more.

- **Lights**: Street lights automatically turn on when the simulation enters night mode. The lights are placed by the developers of the map, and accessible as *carla.Light* objects. Vehicle lights have to be turned on and off by the user. Each vehicle has a set of lights listed in *carla.VehicleLightState*. However, not all vehicles contain lights integrated (Figure 3.3).

- **Debug**: World objects have a *carla.DebugHelper* object as a public attribute. It allows for various shapes to be drawn during the simulation, so that it can be more straightforward to track multiple events during the simulation (Figure 3.4).

- **World snapshots**: Contain the state of every actor in the simulation at a particular frame with a time reference. The information comes from the same simulation step.

- **World settings**: The world gains access to some advanced configurations for the simulation, these determine rendering conditions, simulation time-steps and synchrony between clients and server. They are accessible from the helper class *carla.WorldSettings*. By default, CARLA runs with the best graphics quality, a variable time-step, and asynchronously.

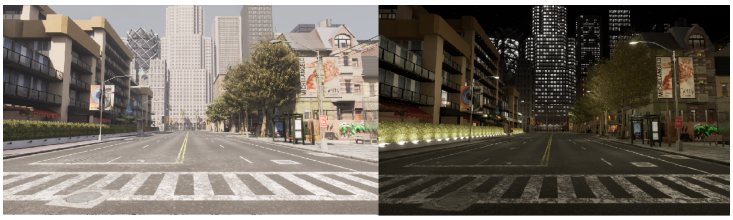


Figure 3.3: The light differences of the map during different times of the day.

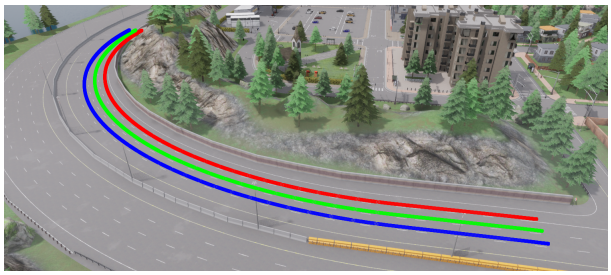


Figure 3.4: A showcase of the usage of debug.draw_string function. The green colour represents the center of the lane travelled by the testing vehicle, whilst the remaining colours represent the central points of the adjacent lanes

Finally, it is important to mention how the coordinates system of CARLA works. Currently, the XYZ axis follows the same orientation as Unreal, which is left-handed (X goes forward, Y goes to the right side and Z goes upwards). CARLA's Python API offers classes such as *carla.Location* which represents a spot in the world with the instance variables: x, y and z and *carla.Rotation* which represents the 3D rotation (pitch, raw and roll) (Figure 3.5). Finally, *carla.Transform* defines a combination of location and rotation and it is useful to quickly get the location and rotation parameters of a given actor in a particular instance of the simulation or set a new position by applying *carla.Transform* with a different location and rotation.

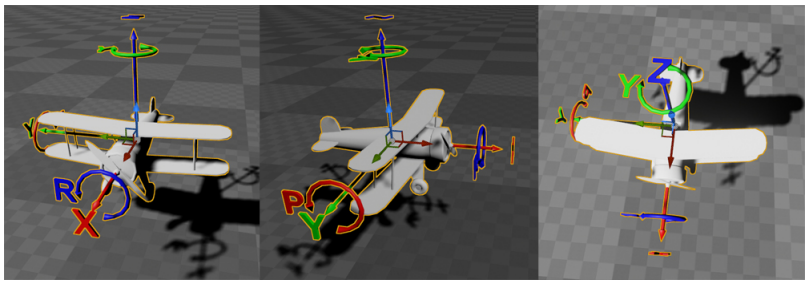



Figure 3.5: CARLA's coordinates system which is the same as Unreal Engine.

### 3.2.2 The actors and their purpose

CARLA offers a vast variety of actors to interact, to customise and to perform actions within the scenario, specified by the user [43]. These objects exist in blueprints exclusively developed for this task and are located in the Blueprint Library. Typically, the usage of actors has a life cycle which involves their spawn, handling and destruction. These objects have incorporated animations and a series of attributes that allows for certain modifications of the actor itself, for example, a vehicle colour, amount of channels in a LIDAR sensor, a walker speed and much more. A typical usage of the blueprint library and the life cycle of an actor is presented in Figure 3.6.

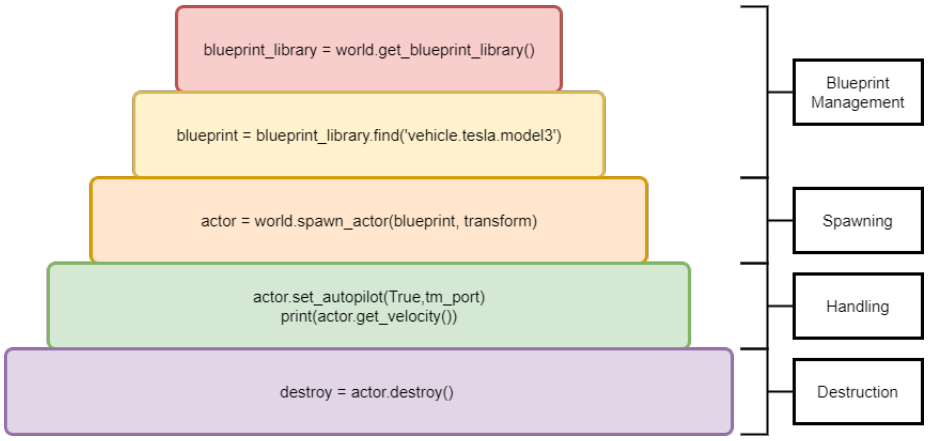



Figure 3.6: A showcase of the retrieval of a blueprint and the life cycle of an actor.

The management of the blueprint library is very flexible, allowing the user to find a certain ID of a model, choose a blueprint at random or filter the library as a whole by using a wildcard pattern [44].

The "spectator" (Figure 3.7) represents a distinctive type of actor in whom it is readily available once the server starts and therefore does not experience the typical life cycle of other actors. It is a functionality of the Unreal Engine and provides the absolute freedom of camera movement for its user to see its surroundings with the typical video game hotkeys WASD to control front, left, back and right movement respectably, QE hotkeys to change the Z elevation (down and up respectably) and the computer mouse to alter the orientation of the camera. It is a significant tool to operate during simulations as it helps to debug any abnormal behaviour and witness the unfolding of the created scenario.

In terms of traffic signs (Figure 3.8), only stops, yields and traffic lights are considered actors in CARLA, the rest of it are only accessible through another class called *carla.Landmark*. The traffic signs represent another type of distinguished actors since they are spawned during the generation of the map, by the information provided by the OpenDRIVE file (Section 3.2.5). Therefore, there is no blueprint available for them and can't be spawned. The way that these signs affect the behaviour of the vehicles when faced by, for example, a red light, is that they are not predefined by a road map but through a bounding box that occupies a certain area that the sign is overlooking to. It transfers the user more control of the simulation by knowing, for instance, if a vehicle is a

located near a certain traffic light and asking to that vehicle the current state of it. Furthermore, traffic lights not only have their unique ID (as all the other actors) but in junctions they have their own group ID, allowing for a more cooperative management of them on the changing of red light to green light and vice versa.



Figure 3.7: The spectator point of view.




Figure 3.8: Traffic signs actors in CARLA, from left to right, stop, yield and traffic lights.

A vehicle in CARLA (Figure 3.9) incorporates internal components that simulate the physics of wheeled vehicles by applying four types of control: *carla.VehicleControl* provides input for driving commands like steering, brake and throttle, *carla.VehiclePhysicsControl* defines physical attributes of the vehicle, *carla.GearPhysicsControl* allows the control of the gears and *carla.Wheel PhysicsControl* which provides the specific control over each wheel. The physics applied to the vehicle are limited by a bounding box that encapsulates it and enables collisions to be detected. Finally, the user can turn on or off the lights of the vehicle in *carla.VehicleLightState* and also enable the autopilot mode, where it will subscribe the vehicle to the Traffic Manager which will be explained in a later topic (Section 3.2.9).

Similar to vehicles, the walker (Figure 3.10) works the same way, where control over them is provided by controllers namely *carla.WalkerControl* and *carla.WalkerBoneControl*. The first one

allows to move the pedestrian around in a certain direction or speed and the second, provides the ability to manipulate its 3D skeleton. Contrary to the case of vehicles, the walkers don't share the autopilot function with them instead, the *carla.WalkerAIController* represents the class that allows the pedestrian to be AI controlled by defining a initialisation, a goal and a speed.

Finally, sensors are actors that produce valuable data about the simulation, have their own blueprints for setting up the attributes, have the ability to attach to a given vehicle and listen to data.




Figure 3.9: Model 3 of Tesla in CARLA.



Figure 3.10: A group of pedestrians in CARLA.

### 3.2.3 A more detailed overview of the available sensors and Data retrieval

As previously mentioned, sensors are actors that retrieve data from their surroundings. For that reason, sensors usually are intertwined with providing a learning environment for driving agents. Hence, how does data retrieval works in CARLA?

To begin with, it is necesssary to define it: *carla.SensorData* is a base class for objects that contain information generated by a *carla.Sensor*. Each of the sensors requires a specific type of sensor data, for example, a RGB camera needs to use *carla.Image* and a LIDAR sensor demands the usage of *carla.LidarMeasurement*. Lastly, it is required to retrieve it on every simulation step or in a certain event, by calling a *listen()* method to manage the data.

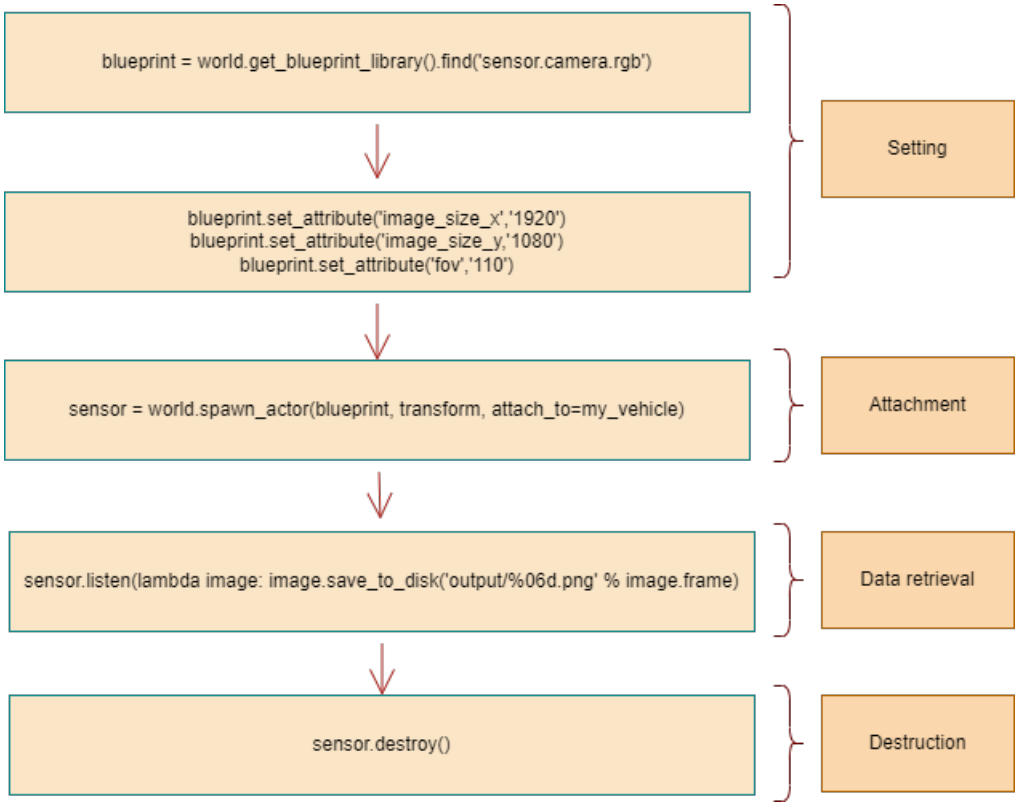



Figure 3.11: Typical life cycle of a sensor.

In Figure 3.11, it is exemplified the distinct stages of a sensor during a simulation. Starting with the setup of the sensor, in this case a RGB camera, it initially follows the similar procedure of every actor during this phase in which the user retrieves the desired blueprint and sets the desired attributes. When spawned, the *attach_to* and *attachment_type* parameters are crucial for the sensor to be attached to a parent actor, normally a vehicle, so that it follows it and gathers data. The attachment type determines how its position is updated regarding the attached vehicle: *Rigid attachment* in which the movement is strict in relation to the location of the object that it is attached to and *SpringArm attachment* where the movement is more smooth with little accelerations and deceleration, useful for recording videos for example. Finally, the *listen* method is called every

time the sensor retrieves data, the argument callback is a lambda function [45] which describes what should the sensor do when information is retrieved. Most sensor data objects have the ability to save the information to disk. Although sensor data might differ between different sensor types, there are some basic attributes that they all share, mainly: the frame number where the measurement took place, the timestamp of the measurement in simulation seconds and the world reference of the sensor at the time of the measurement, *carla.Transform* [46].

The following list represents all the available sensors in CARLA [3]:

- **Depth**: Renders the depth of the elements in the field of view in a grayscale map. The camera provides raw data from the scene, codifying the distance of each pixel to the camera to create a depth map of the elements present on that instance. The image codifies the depth value per pixel operating the three channels of the RGB colour space and converted into a [0,1] float containing the distance and then render it to grayscale (Figure 3.12g).

- **RGB**: Provides clear vision of the surroundings. The RGB camera acts as a standard camera capturing images from the scene. If *enable_postprocess_effects* is enabled, a set of post-process effects is applied to the image to make it more realistic: vignette (darkens the border of the screen), grain jitter (includes noise to the render), bloom (intense lights burn the area around them), auto exposure (modifies the image gamma to simulate eye adaptation to darker or brighter areas), lens flare (simulates the reflection of glittering objects on the lens) and depth of field (blurs objects near or very far away of the camera). The value of the attribute *sensor_tick* contains how fast the sensor captures the data, a value of 1 means that the sensor captures at each second. By default a value of 0.0 means as fast as possible (Figure 3.12b).

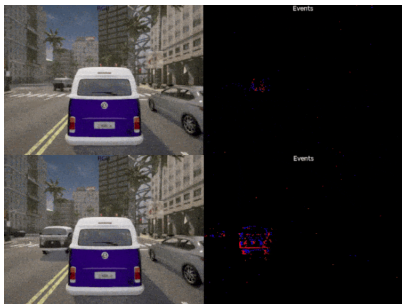
- **Optical Flow**: Renders the motion of every pixel from the camera. The Optical Flow camera captures the motion perceived from the point of view of the camera in which every pixel recorded by this sensor encodes the velocity of that point projected to the image plane (Figure 3.12c).

- **Semantic segmentation**: Renders elements in the field of view with a specific colour according to their tags. The raw image can be retained and converted to *CityScapesPallete* (Figure 3.12d).

- **DVS**: A Dynamic Vision Sensor (DVS) or Event camera is a sensor that instead of capturing intensity images at a established rate, event cameras measure changes of intensity asynchronously, in the form of a stream of events, which encode per-pixel brightness changes. Event cameras are hence sensors that can provide high-quality visual information even in challenging high-speed scenarios and high dynamic range environments due to no motion blur, very high dynamic range and high temporal resolution, enabling new application domains for vision based algorithms (Figure 3.12a).

- **Collision detector**: Retrieves collisions between its parent and other actors. To ensure that collisions with any kind of object are detected, the server creates a dummy actor for environment objects such as buildings so the semantic tag can be retrieved to identify it.
- **Lane invasion detector**: Registers when its parent crosses a lane marking. It uses road data provided by the OpenDRIVE description of the map to determine it by considering the space between wheels.
- **Obstacle detector**: Detects possible obstacles ahead of its parent. In order to anticipate obstacles, the sensor develops a capsular shape ahead of the parent vehicle and resorts it to check for collisions.
- **GNSS**: Retrieves the geolocation of the sensor. This is calculated by adding the metric position to an initial geographical reference location defined within the OpenDRIVE map definition.
- **IMU**: Comprises an accelerometer, a gyroscope, and a compass.
- **LIDAR**: A rotating LIDAR. Generates a 4D point cloud with coordinates and intensity per point to model the surroundings. A LIDAR measurement contains a package with all the points generated during a 1/FPS interval. During this interval the physics are not updated so all the points in a measurement reflect the same static picture of the scene. The information of the LIDAR measurement is encoded 4D points. Being the first three, the space points in x, y and z coordinates and the last one intensity loss during the travel. For a better realism, points in the cloud can be dropped off: General drop-off, proportion of points dropped off randomly and intensity-based drop-off, for each point detected and extra drop-off is performed with a probability based in the computed intensity. This is a convenient way to simulate loss due to external perturbations. Additionally, there is an attribute simulates unexpected deviations that appear in real-life sensors. The rotation of the LIDAR can be tuned to describe a specific angle on every simulation step (using a fixed time-step) (Figure 3.12e).
- **Radar**: The sensor creates a conic view that is translated to a 2D point map of the elements in sight and their speed regarding the sensor. Due to the use of polar coordinates, the points will concentrate around the centre of the view. Points measured are contained in *carla.RadarMeasurement* as an array of *carla.RadarDetection*, which specifies their polar coordinates, distance and velocity.
- **RSS**: Modifies the controller applied to a vehicle according to safety checks. This sensor is disabled by default in CARLA, and it has to be explicitly built to be used.
- **Semantic LIDAR**: A rotating LIDAR. Generates a 3D point cloud with extra information regarding instance and semantic segmentation. Works similarly in comparison to LIDAR, but the raw data includes more information at each point (coordinates of the point, the cosine

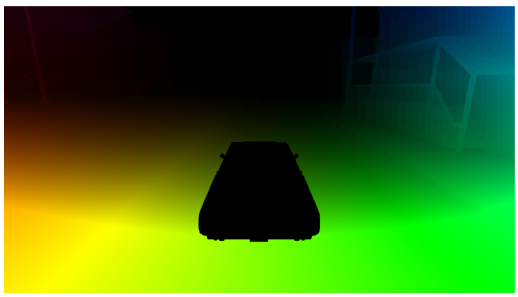
between the angle of incidence and the normal of the surface hit and instance and semantic ground-truth). However, the semantic LIDAR does not include neither intensity, drop-off nor noise model attributes (Figure 3.12f).



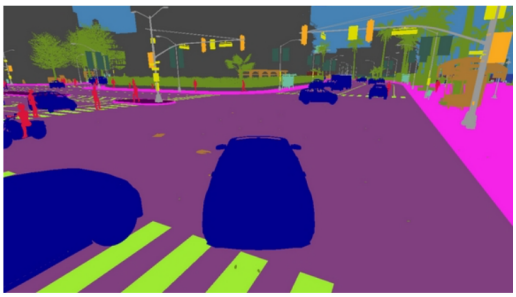

(a) Dynamic Vision Sensor.



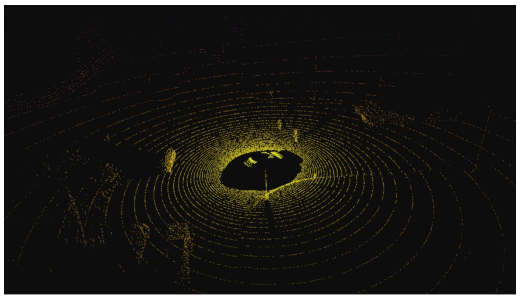
(b) RGB camera.



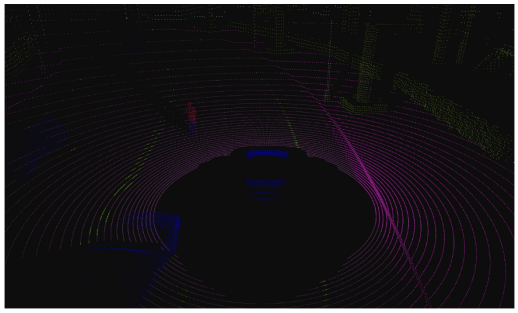
(c) Optical flow camera.



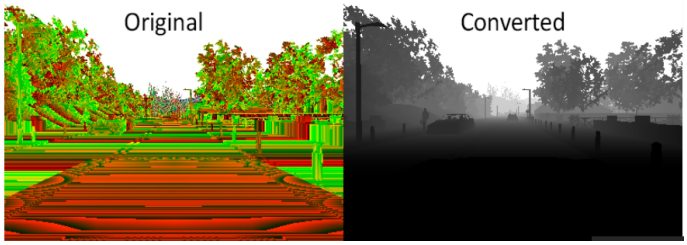
(d) Semantic segmentation camera.



(e) LIDAR sensor.



(f) Semantic LIDAR sensor.




(g) Depth camera.

Figure 3.12: Group of sensor outputs, retrieved in [3].

### 3.2.4 The existing maps, their environments and the navigation involved

A map includes the 3D model of a town and its road definition based on an OpenDRIVE file, a standardised, annotated road definition format [47]. As such, the way the file defines roads, lanes

and junctions, regulates the functionality of the Python API and the reasoning behind its decisions. Therefore, Python API functions proceeds as a high level querying system to navigate through out the map. Each map has a name attribute that matches the currently loaded scenario, for example, in Figure 3.2 *Town05*. In Table 3.1, it is presented the existing CARLA maps [48].

Table 3.1: Comparison of the different predefined maps of CARLA.

| Town | Summary | Type |
|---|---|---|
| Town01 | A basic town layout consisting of T junctions. | Strictly Urban (suburbs) |
| Town02 | Similar to Town01, but smaller. | Strictly Urban (suburbs) |
| Town03 | The most complex town, with a 5-lane junction, a roundabout, unevenness, a tunnel, and more. | Strictly urban |
| Town04 | An infinite loop with a highway and a small town. | Mix of rural and urban |
| Town05 | Squared-grid town with cross junctions and a bridge. It has multiple lanes per direction. Useful to perform lane changes. | Strictly urban |
| Town06 | Long highways with many highway entrances and exits. It also has a Michigan left. | Mix of rural and urban |
| Town07 | A rural environment with narrow roads, barns and hardly any traffic lights. | Strictly rural |
| Town10 | A city environment with different environments such as an avenue or promenade, and more realistic textures. | Strictly urban |

Landmarks are a prominent feature that defines any type of traffic landmark or sign affecting a road. The class *carla.Landmark* facilitates information about which lanes does it affect and when. In other words, the attributes and methods of this class describe the landmark and its area of influence. With that information obtainable, landmarks are normally accessed by *carla.Waypoint* objects, which try to retrieve the regulation of their lane. Waypoints in CARLA are 3D directed points that have their own unique location and orientation in a given road. Therefore, waypoints store the road information in terms of its lane, lane markings, a Boolean to determine if it is inside a junction, lane type, width and lane changing permissions. The lane types, defined by OpenDRIVE standard 1.4, are accessed through *carla.LaneMarking* and contains variables like colour, lane change, type of the lane and width. Finally, everything related to waypoints happens on the client side, which means that it is only needed one communication to the server to get the map object containing the information regarding them.

### 3.2.5 OpenDRIVE in CARLA

As previously referred, the predefined maps in CARLA have its road definition based on OpenDRIVE format. But what if the user wants to incorporate other maps, not available in CARLA, that are based on the real world? That is where the feature OpenDRIVE standalone mode from CARLA can be used. It allows to input any OPENDRIVE file and the simulator generates a road mesh for actors to navigate on. What it means is it procedurally creates a 3D mesh to simulate in a minimalistic manner. The result is a map filled with lanes but the remaining space being void. To prevent the vehicles from falling, lanes are a bit wider at junctions due to a more prominent flow

of vehicles and walls are created at the limits of the road. The remaining objects: traffic lights, stops, yields, sidewalks and crosswalks will be created based on the OpenDRIVE file [49].

The mesh generation is constantly being improved. Instead of producing a single map, separate portions are generated so that, the simulator avoids rendering portions that are not visible and contain issues that may occur on a specific region of the created mesh.

Bellow (Figure 3.13) is the feature being put into practice by replicating the Circunvalação from the city Oporto.



Figure 3.13: Scenario created by OpenDRIVE feature of CARLA.

### 3.2.6 Recording simulations in CARLA

The recorder in CARLA grants the ability to record and reenact a previous simulation registered in a recorder file. The information is compiled in a binary file managed by the client and retrieves on the server side: actors, state of the traffic lights, vehicles and pedestrians positions, and light states from buildings, streets and vehicles. By default the user only retrieves frames where specific events happened like collisions or a vehicle being stuck on the same place by quite some time. The information is organised by the following: timestamp of the event, the actor types (vehicle, walker and others...) and the id of the actors involved. This feature is quite important for debugging since it allows for a simulation playback at any time during the recording, which permits to query the information in a given event and troubleshoot it [50].

### 3.2.7 Rendering

An interesting topic to discuss, the rendering implemented in CARLA allows for a higher fidelity in graphics quality thanks to Unreal Engine 4. However, a main drawback is it requires a powerful workstation to work with if the user wants the simulation to perform not solely on a stable frame rate but also not running into issues with the communication between the client and server due to bottlenecks in the system resources. It is my opinion that this free software could be quite

interesting to work with in academic situations where students could be introduced to the application of machine learning in automotive driving. Unfortunately, where the usage of desktops with integrated graphics card such as Intel is still the norm in classrooms, it stands no chance when operating with software like this. Thankfully, there are some workarounds in CARLA to ease the workload that it does to more weaker computers.

CARLA has two presets for graphics quality (Figure 3.14), Epic (default) and Low where it disables all post-processing, shadows and the drawing distance is set to 50m at maximum. When CARLA is performed under Low preset, the simulations runs significantly faster but with some limitations. The preset is useful if precision is unessential or if the training of agents is under conditions with simpler data and involving close elements [51]. For better customisation, the user can go to Settings/Engine Scalability Settings and add, remove or change the values of each attribute that it follows Unreal nomenclature [52].



Figure 3.14: Graphic presets available, left image is the Epic preset and the right image is Low preset.

Finally, there are two modes that the user can enable to considerably reduce even more the workload of CARLA on the computer, No-rendering mode and Off-screen mode. The No-rendering mode as the name suggests, disables rendering, which means that anything regarding graphics is skipped, facilitating the traffic simulation and road behaviours at higher frequencies. The limitation is that cameras and GPU based sensors will return empty data since the GPU is not being used to render. The off-screen mode disables the display of the simulation but the rendering is still computed therefore enabling for the GPU based sensors to return data.

### 3.2.8 Synchrony and time-step

The simulated world has its own clock and time, controlled by the server [53]. Computing two simulation steps takes real time, normally a few milliseconds. The time spent between the simulation moments is named time-step which can be, for example, a second. For the time-step there are two modes: Variable time-step and Fixed time-step. Variable time-step is the default mode in CARLA, the time step between the two simulation moments will be the time that it takes for the server to compute. Fixed time-step as the name suggests fixes the elapsed time between steps, for

instance if it is set 1/60 seconds, it means that there will be sixty frames per simulated second. By using a constant time increment, it will make it easier to comprehend the moment where it occurred in the simulation and allow to simulate more extended time periods in less real time. Physics sub-stepping must be computed with very low time steps to be precise. To have an optimal simulation, the sub-step delta time should at least be below 0.01666 and ideally below 0.01, a condition that will be needed to take into account later on. If the synchronous mode and time step is set, then the sub-stepping options need to be consistent with the value of the fixed delta seconds (Equation 3.1). By default, CARLA runs in asynchronous mode therefore, the server runs the simulation as fast as possible, without waiting for the client. However, on synchronous mode, the server waits for a client tick, before updating to the next simulation step. Changing between the synchronous and asynchronous mode is just a matter of a Boolean state. The synchronous mode is specially relevant with slow client applications and when synchrony between different sensors is needed. The consequence of not using the synchronous mode is an overflow of information if the client is too slow and the server does not wait, therefore, creating issues such as the client not being able to manage the overflow of information and either it will be lost or mixed. If using multiple sensors with asynchronous enabled, it would became impossible to know if all the sensors are using data from the same moment in the simulation. The following table (Table 3.2) summarises the different outcomes of each mode.

Table 3.2: The different results of synchrony and time-step.

| **Condition** | **Summary** |
|---|---|
| Synchronous mode + variable time-step | Non desirable state since simulation time and physics will not be in synchrony. Physics cannot run properly when the time-step is bigger than 0.1s and if the server has to wait for the client to compute the steps, this is likely to happen. The simulation will be unreliable. |
| Asynchronous mode + variable time-step | Default CARLA state where the client and server are asynchronous. The simulation time flows according to the real time. Reenacting the simulation needs to take into account the precision of the repetition of what has happened and possible differences in time steps between servers. |
| Asynchronous mode + fixed time-step | The server will run as fast as possible and the information retrieved will be clearly related with an exact moment in the simulation. This configuration provides stability to simulate extended periods of time in much less real time, that is, if the server is fast enough. |
| Synchronous mode + fixed time-step | The time step will be fixed, and the server will not compute the following step until the client releases a tick. This is the best mode when synchrony and precision are relevant. |

$$fixed_delta_seconds \leq max_substep_delta_time * max_substeps \tag{3.1}$$

### 3.2.9 Traffic Simulation in CARLA

Traffic simulation is integral to training and testing of autonomous driving stacks. As it can be seen in Table 3.3 [54], CARLA allows to use a decent amount of options for controlling the traffic during the simulation. In this project, it was adopted Traffic Manager (TM) [55] since it is an already integrated module in CARLA, allowing to instantly customise the traffic scenario in the client side from the get-go. Due to that, the remaining content of this subsection will be about how it works and the demonstration of certain customisation parameters. The TM implements general behaviour patterns that must be taken into consideration. For instance, vehicles are not goal-oriented as they follow a dynamically produced trajectory and randomly select a path when approaching a junction. Therefore, it is considered that their path is endless. Target speed is 70% of their current speed limit unless any other value is set. The TM uses its own priority system at junctions and does not follow current traffic regulation which means some issues may arise, for example, vehicles inside a roundabout yielding to a vehicle trying to get in. Currently, the workflow of TM is organised by the following (Figure 3.15):

1. **Store and update the current state of the simulation**: The Agent Lifecycle and State Management (ALSM) scan the world to keep track of all the vehicles and walkers present and to clean up entries for those that no longer exist. All the data are retrieved from the server and is passed through several stages. The ALSM is the only component that makes calls to the server. The vehicle registry contains an array of vehicles on the autopilot (controlled by the TM) and a list of pedestrians and vehicles not on the autopilot (not controlled by the TM). The simulation state is a cache store of the position, velocity, and additional information of all the vehicles and pedestrians in the simulation.

2. **Calculate the movement of every autopilot vehicle**: The TM generates viable commands for all vehicles in the vehicle registry according to the simulation state. Calculations for each vehicle are done separately. These calculations are divided into distinct stages. The control loop makes sure that all calculations are consistent by creating synchronisation barriers between stages. No vehicle moves to the subsequent stage before calculations are finished for all vehicles in the current stage.

    (a) **Localisation Stage**: Paths are forged dynamically using a list of nearby waypoints collected from the In-Memory Map, a simplification of the simulation map as a grid of waypoints. Directions at junctions are chosen randomly. Each vehicle's path is stored and maintained by the Path Buffers & Vehicle Tracking (PBVT) component for easy access and modification in future stages.

    (b) **Collision Stage**: Bounding boxes are extended over each vehicle's path to identify and navigate potential collision hazards.

(c) **Traffic Light Stage**: Similar to the Collision Stage, potential hazards that affect each vehicle's path due to traffic light influence, stop signs, and junction priority are identified.

(d) **Motion Planner Stage**: Vehicle movement is computed based on the defined path. A PID controller determines how to reach the target waypoints. This is then translated into a CARLA command for application in the next step.

(e) **Vehicle Lights Stage**: The vehicle lights switch on/off dynamically based on environmental factors (e.g. sunlight and the presence of fog or rain) and vehicle behaviour (e.g. turning on direction indicators if the vehicle will turn left/right at the next junction, or turn on the stop lights if braking).

3. **Apply the commands in the simulation**: Commands generated in the previous step are collected into the command array and sent to the CARLA server in a batch to be applied in the same frame.

In practical terms, a TM instance is produced by a *carla.Client (tm = client.get _trafficmanager(port))*, passing the port to be used. The default port is 8000. To enable autopilot for a specific vehicle, it is necessary to retrieve the port of the TM instance and set autopilot mode to true: emphvehicle.set_autopilot(True,tm_port). If the TM does not exist, it will create one and if no port is provided it will use the default port, 8000. The TM instance allows altering safety conditions like the minimum distance between vehicles, the desired speed as a percentage to the current limit (default is 70%), reset traffic lights, make a vehicle ignore all other vehicles, walkers and traffic lights, force lane change and enable Hybrid physics mode.

The Hybrid mode allows users to disable most physics calculations for all autopilot vehicles or for autopilot vehicles outside of a certain radius of a vehicle tagged as "hero". Vehicles whose physics are disabled will move by teleportation using basic calculations for linear acceleration to ensure the position updates, vehicle speed remain as realistic as possible and the toggling of physics calculations on vehicles is fluid. This feature removes the vehicle physics bottleneck from the simulator, something that can be a reality when a simulation has a much bigger quantity of vehicles and as a consequence a bigger workload to calculate the physics of each.

The TM is deterministic, which means that it will produce the same results and behaviours under similar conditions. The *Seed_value* is an integer number from which random numbers will be generated. Although the value itself is not relevant, using the same value will always result in the same output. This feature is available in synchronous mode only since in asynchronous mode, there is less control over the simulation and determinism cannot be achieved.

Ultimately, the TM will stop when the client that generated it stops. This is automatically managed by the API. However, when shutting down a TM, the user must destroy the vehicles handled by it, otherwise they will remain immobile on the map. Another point essential to make is that TM is designed to work in synchronous mode, both the CARLA server and TM should be set to synchronous to function properly. If using TM in asynchronous mode, it can lead to unexpected

and undesirable results. However, if it is an absolute necessity, the simulation should run at 20-30 FPS at least.

Table 3.3: The options available for traffic simulation in CARLA.

| **Mode** | **Summary** | **Utility** |
|---|---|---|
| Scenario Runner and OpenSCENARIO | Scenario Runner provides predefined traffic scenarios and additionally allows users to define their own scenarios by using either Python or the OpenSCENARIO 1.0 standard. The primary use of OpenSCENARIO is the description of complex manoeuvres that involve multiple vehicles. Users can see which features of OpenSCENARIO are supported by Scenario Runner here. These features include manoeuvres, actions, conditions and stories. However, the Storyboard.Scenario Runner has to be installed separately from the main CARLA package. | Useful for creating complex traffic scenarios and routes. |
| Traffic Manager | Traffic Manager is a module within CARLA that handles certain vehicles from the client side. Vehicles are registered to Traffic Manager via the carla.Vehicle.set_autopilot method or command.SetAutopilot class. Control of each vehicle is managed through a cycle of distinct stages which each run on a separate thread. | Easy for populating a simulation with realistic urban traffic conditions. Customising traffic behaviours to set specific learning circumstances. |
| Scenic | Scenic is a domain-specific probabilistic programming language for modelling the environments of cyber-physical systems like robots and autonomous cars. It provides a specialised domain to facilitate execution of Scenic scripts on the CARLA simulator. | Generating multiple, diverse scenarios with a single scenario definition. Defining probabilistic policies for dynamic agents to take actions over time in response to the state of the world. |
| SUMO | SUMO is an open source, microscopic, multimodal traffic simulation. In SUMO, each vehicle is modelled explicitly, has its own route and moves individually through the network. Simulations are deterministic by default, but there are various options for introducing randomness. CARLA provides a co-simulation feature with SUMO that allows distribution of tasks between the two simulators. Vehicles can be spawned in CARLA through SUMO, and managed by SUMO much as the Traffic Manager would do. | Exploitation of the capabilities of both CARLA and SUMO in one unique package. |

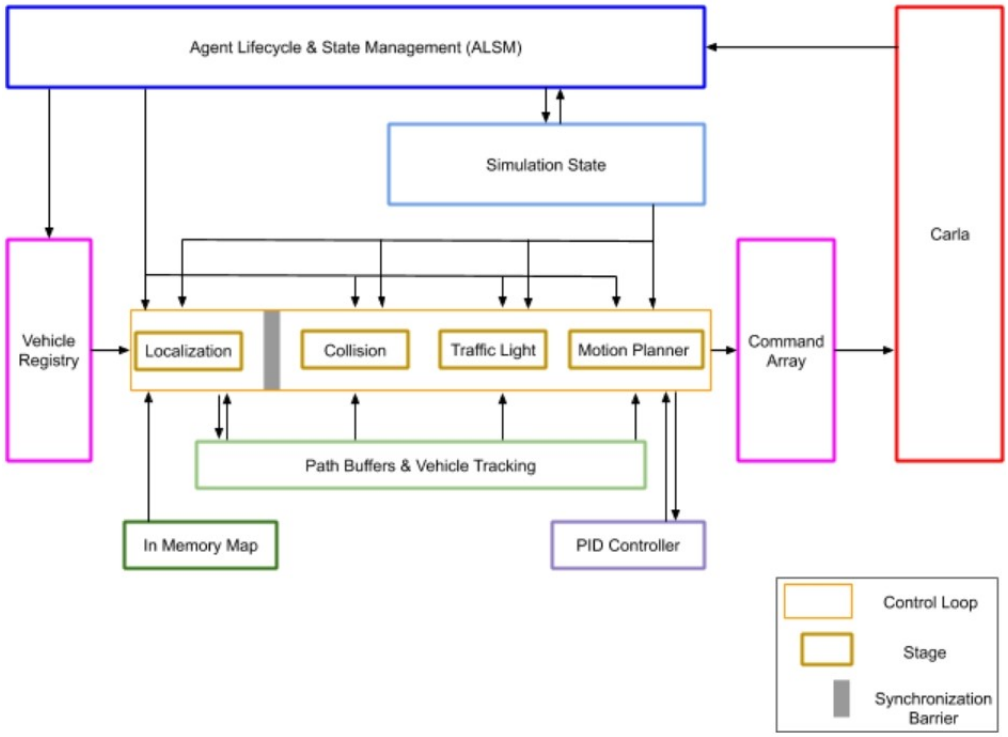



Figure 3.15: Architecture of the TM.

## 3.3 Creation of the main script

The process of developing a script that would serve as the main framework for the work developed, had many iterations before reaching to a final version and moving on to test it on future topics. The Figure 3.16 shows the several stages of the script on a macro level.

As already mentioned in the sub-chapter 3.2.1, the first step that needs to be done is the setup of the client connection to the server followed by choosing the desired map, setting up the TM, synchronisation settings and the weather (Sections 3.3.1, 3.3.3 and 3.3.6 ). Next, the actors are spawned, starting with the vehicles during a for loop to attribute each actor to a certain spawn location and then insert it to the TM's control. The first vehicle actor spawned is always the Ego vehicle that the sensor will be attached to. Next, the sensor is filtered, customised, spawned and attached (Section 3.3.2). Finally, to end the initialisation of the client, walkers are spawned, and an AI controller is attributed to each. Then the script pauses for a second to ensure everyone on the server side is already in motion so that there is no data retrieval when the actors are preparing to start moving.

The second phase prepares the multi-thread and Queue features and the *listen()* method for the sensor that will be tested. The data from the callback function will be retained in a queue (Sections 3.3.4 and 3.3.5). Finally, the subroutine will follow the Ego vehicle and save the images in the disk.

The following sections will introduce key features implemented in the script, some issues faced and their respective solutions.

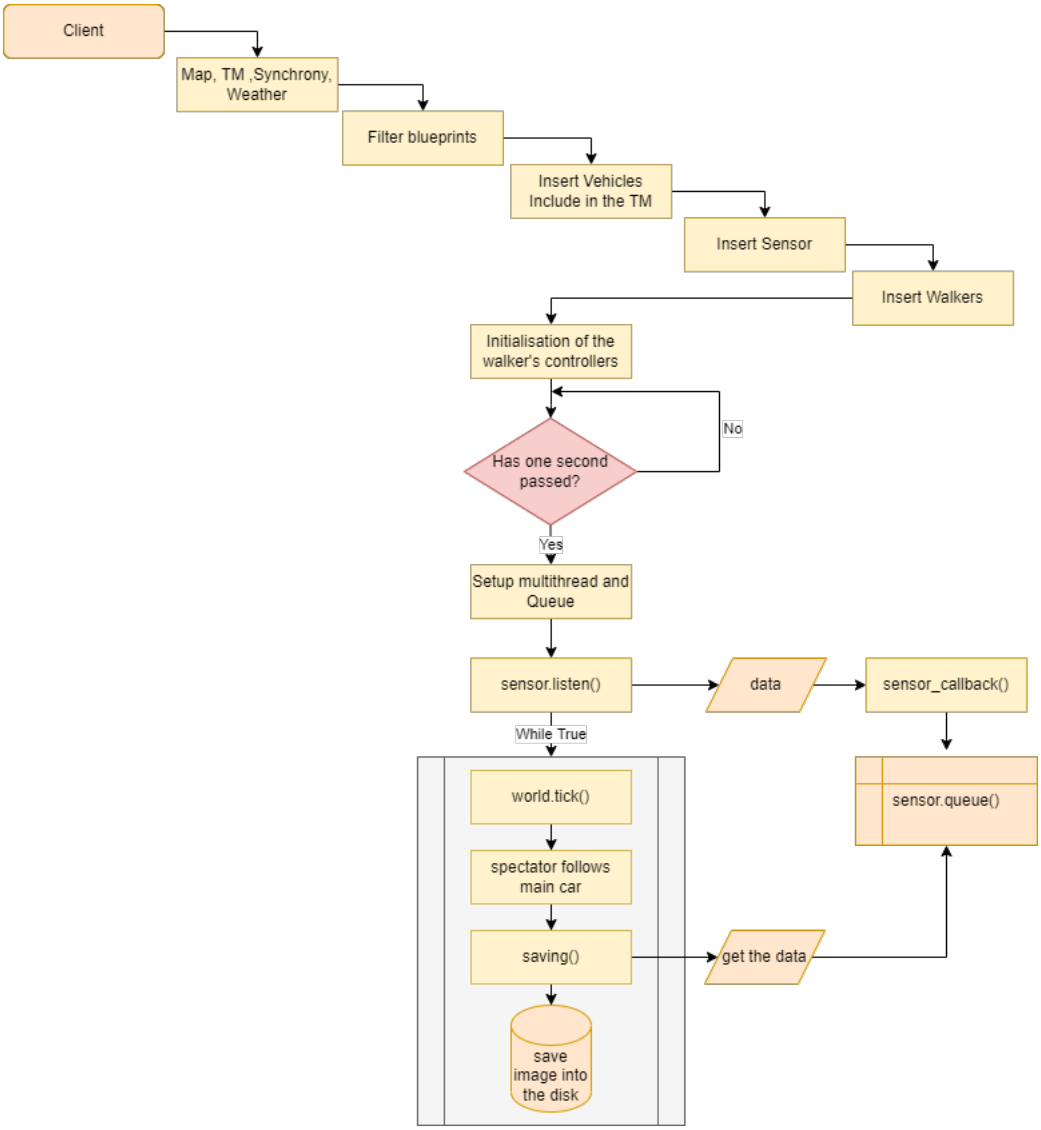



Figure 3.16: An overview of the main script.

### 3.3.1 Configuration file

As it was showcased in the previous sub-chapters, the Python API of CARLA offers a big amount of customisation and settings to alter the generation of the map, actors, sensors and so on. This allowance of altering the attributes of classes and objects in the script can be all over the place, contributing for the confusion when it is needed to alter the script. For that reason, the modification of certain attributes are concentrated on a single INI file [56]. An INI file, is a configuration file that consists of a text-based content with a structure and syntax comprising key-value pairs for properties and sections that organise those properties. As such, it was used the *ConfigParser*

module [57] to prepare the script to read from the created INI file, *config.ini*. The file is divided in four sections: World settings, Vehicle settings, Walker settings and Sensor settings. The first section has familiar proprieties such as TM port (tm_port = 8000), Boolean variables for the type of synchrony in the script (false for synchronous mode, true for asynchronous), hybrid mode and string variables to filter the blueprint library (e.g. *filterw* = *walker.pedestrian.** and *filterv* = *vehicle.**). The next section, Vehicle settings, only specifies the number of vehicle actors that will be in the simulation. The third section, not only defines the number of walker actors but also a percentage of them that will be running and another for how many pedestrians will walk across the road. Finally, the section, Sensor settings, defines the attributes for the sensor that will be generated in the server, see 3.3.2 for more details. To use the sections of the file as an input for the script, the usual workflow is to initiate the module with *parser* = *ConfigParser()*, read the file *parser.read('config.ini')* and then obtain, for example, the number of vehicles specified in the configuration file, *number_of_vehicles* = *parser.getint('vehiclesettings', 'number_of_vehicles')*.

### 3.3.2 Sensor settings

CARLA offers a wide range of sensors and detectors as it was previously demonstrated in the sub-chapter 3.3.2. It was chosen to use the RGB Camera for the following reasons: bigger R&D [2] in the virtual simulators [25][58], the existence of machine learning algorithms of Continental that already interact with this type of input and a better showcase of CARLA, not only in functional terms but also because of its capabilities in graphic fidelity. One of the goals is also to replicate the sensors used by Continental. Due to that, it was decided to study a possible application of the MFC525 monocular camera system.

The MFC525 is a front viewing automotive monocular camera system. It is a standalone system which includes image sensor and signal processing in one box. The camera is a 5th generation ECU including optics, electrical hardware, electrical connector and housing. The MFC525 is a front viewing, one box design, driver assistance camera which is mounted behind the windscreen in the mirror area.

The specifications of the optical module are field of view (FOV) which the values are 110 x 47.5 in degrees, horizontal and vertical respectably, the value of F-number is 1.6 and image resolution 1820 x 940 pixels. Those values are easily replicated on the virtual sensor by the blueprint attributes of the RGB camera: *fov*, *fstop*, *image_size_x* and *image_size_y*.

Now, to analyse the limitations encountered in this integration process. The first one was to setup the vertical FOV directly: in CARLA, the only FOV possible specify is the horizontal FOV, the vertical is automatically defined in accordance. It was decided to maintain the field of view on 110 degrees only for the horizontal view and accept the limitation on CARLA's part. The second one was dealing with the remaining attributes that the blueprint of the RGB Camera offered for a more enrichment on the sensor. This was a practical example about the reality gap and the problems that come with it, 2.7. For the RGB Camera, the blueprint offers attributes such as blur amount, bloom intensity, exposure speed and others. Mostly, those parameters are filled with an integer or a float variable that seem a bit out of reach to factually predict without having

subjectivity involved. At what point can one spot the difference of motion blur intensity between a real camera and a virtual one? Should the value be 0.45 or 0.55? This issue would require to study the Unreal Engine itself to better understand those parameters, which is out of reach for this project at the current time frame. Therefore it was decided to keep the remaining attributes with the default values since, the established ones already ensure a perfectly capable camera to simulate post processing effects, noise, depth of field, blur, exposure and others.

A note about the positioning of the sensor, although it is referenced that the camera should be placed behind the windshield of the vehicle it was decided to place it on top of the vehicle (1.6 meters of height and 1 meters in front of the centre of the vehicle (*sensor_location = carla.Location(1, 0, 1.6)*)) for future work on Chapter 4.

### 3.3.3 Synchrony settings

For the script it was chosen to use synchronous mode with fixed time-step since, as stated by the Table 3.2, precision will be relevant for future applications at a certain tick rate. The time-step is fixed at $\frac{1}{60}$ which means the simulated world will take 60 steps to simulate a single second and adds more flexibility for each preset of the *sensor_tick* attribute (60 FPS: one frame per tick, 30 FPS: one frame per two ticks, 20 FPS: one frame per three ticks and 10 FPS: one frame per six ticks). The setup of the synchronisation in CARLA follows the recommended procedures such as group up the settings that are to be changed and then apply them to the server, calling the command *world.tick()* in a *while()* cycle so that the simulation advances step-by-step and once the script has finished, reset the same settings that were previously tweaked and turn off synchronous mode. Finally, it is important to not forget about the Equation 3.1. Since synchrony is enabled with a fixed step of $\frac{1}{60}$, in order to satisfy the condition for the *max_substep_delta_time* to be bellow the ideal value, 0.01, the optimal value for *max_substep* is $\frac{10}{6}$. However, *max_substep* is an integer variable therefore, the value will be replaced to two, making the final result for *max_substep_delta_time* become $\frac{1}{120}$, which fulfils the requirement of the equation.

### 3.3.4 Data retrieval and processing

The most fundamental aspect of this work, the collection of data from the simulator will be crucial to test it in future chapters. The following code fragment shows the division of this task into three distinctive parts. Initially, the method *sensor.listen()* will call the function *sensor_callback()* every time a new measurement is received. The function *sensor_callback()* will place the information in a queue, *sensor_queue*, which will be useful for multi-threaded tasks. Finally the function *saving()* is called during the loop and will get the raw data, remove the alpha parameter and save it on the output folder.

```
1 def sensor_callback(sensor_data, sensor_queue):
2     sensor_queue.put((sensor_data))
3 (...)
4 def saving(s, x):
5     i = np.array(s.raw_data, dtype='uint8')
```

```
    i2 = i.reshape((IM_HEIGHT, IM_WIDTH, 4))
    i3 = i2[:, :, :3]
    im_rgb = cv2.cvtColor(i3, cv2.COLOR_BGR2GRAY)
    Image.fromarray(im_rgb).save('output/%06d.png' % x)

def main():
    (...)
    sensor.listen(lambda data: sensor_callback(data, sensor_queue))
```

Listing 3.1: Snippet of the code of data retrieval

### 3.3.5 Singlethread to Multithread

This section was created because of an issue during the process of data retrieval. Initially, throughout the course of saving the image to the disk of the computer, it usually took quite a bit of time for that single process to be done. It was discovered that the issue was that the images were being saved by a single thread, creating a general slowdown in the simulation, in which even when the simulation ended, the client was still saving the last images that were temporally kept in memory. The result was that the *listen()* method would pretty much be overflowed with information and the process of saving into the disk couldn't keep up by the time the simulation ended. For example saving 600 images with a resolution of 1080P for a total of 10 seconds from the simulation (60 frames per second), took nine minutes in total. During that time frame, the simulation had already stopped at the 31 seconds mark. Which means that it wasted eight minutes and a half just to save the remaining images to the disk. It gets worse with one minute of the simulation and so on. To combat this, it was decided that the script would utilise as many threads as possible to save the images into the disk. The modules chosen to work with were the *concurrent.futures*[59] and *queue*. In the following code snippet, the *executor* variable stipulates the maximum working threads. If *sensor_queue* is filled (*qsize()*), it will remove an element from the queue and assigned it to a thread. Finally, the thread assigned will run the function *saving()* with the variables *s* and *i* as arguments.

In the end, with this workaround, it only took 40 seconds in total to save all the 600 images with a time efficiency of 1350%.

```
from queue import Queue
from concurrent.futures import ThreadPoolExecutor

def sensor_callback(sensor_data, sensor_queue):
    sensor_queue.put((sensor_data))

def main():
    (...)
    sensor_queue = Queue()
    executor = ThreadPoolExecutor(16)
    (...)
    while True:
```

```
    (...)
        if sensor_queue.qsize() > 0:
                s = sensor_queue.get(True, 0.01)
                f = executor.submit(saving, s, i)
                i = i + 1
```

Listing 3.2: Snippet of the code that deals with multithread

### 3.3.6 Weather

The weather is completely customisable since it exists a class *carla.Weather Parameters* that allows the user to define objects such as lightning, rain, fog and so on. Currently, the conditions that the user can create, only intervene with the RGB Camera. They neither affect the actor's physics and other types of sensors. Each parameter acts independently from the rest, for instance, increasing the rainfall will not create puddles or add humidity to the road. As such, realistic conditions need to be scripted from scratch by the user. The attributes tweaked are the following:

- **Cloudiness**: Values from 0 (clear sky) to 100 (completely covered with clouds).
- **Precipitation**: Changes the rain intensity. Values range from 0 (none) to 100 (heavy rain).
- **Precipitation deposits**: Creation of puddles. Values range from 0 (none) to 100 (road covered with water).
- **Wind intensity**: Controls the strength of the wind. The values range from 0 (none) to 100 (strong wind).
- **Sun azimuth angle**: Values range from 0 to 360 where zero is the origin point in a sphere determined by Unreal Engine (Figure 3.17).
- **Sun altitude angle**: Values range from -90 to 90, midnight to midday respectably.
- **Fog density**: Fog thickness. Values range from 0 to 100.
- **Wetness**: Wetness that affects the RGB Camera. Values range from 0 to 100.

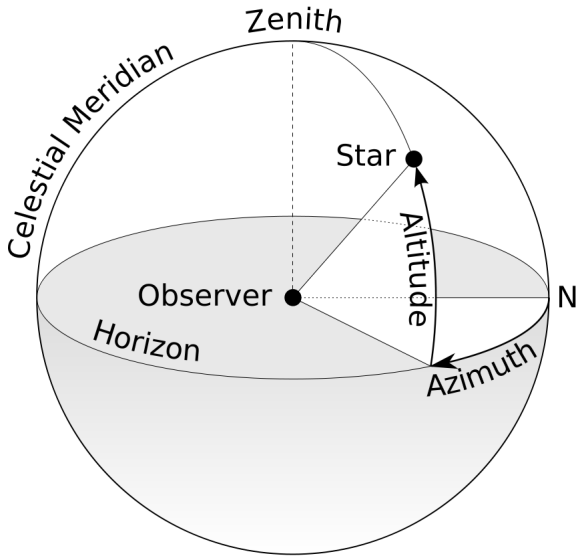



Figure 3.17: An example image to facilitate the understanding of the azimuth angle and altitude angle.

The next code snippet shows the current implementation. Initially, it is defined a random position for the sun. The level of cloudiness determines the existence of precipitation and puddles. The wind is only affected during a storm type of weather and wetness is affected equally by the level of precipitation. Fog density remains unaffected by the previous attributes.

```
class Weather(object):
    def __init__(self, weather):
        self.weather = weather
        self.weather.sun_altitude_angle = 0
        self.weather.sun_azimuth_angle = 0
        self.weather.cloudiness = 0
        self.weather.precipitation = 0
        self.weather.precipitation_deposits = 0
        self.weather.wind_intensity = 0
        self.weather.wetness = 0
        self.weather.sun_azimuth_angle = random.randint(0, 360)
        self.weather.sun_altitude_angle = random.randint(-90, 90)
        self.weather.cloudiness = random.randint(0, 100)
        if 50 <= self.weather.cloudiness <= 100:
            self.weather.precipitation = random.randint(50, 100)
            self.weather.wetness = self.weather.precipitation
            self.weather.precipitation_deposits = self.weather.precipitation
            self.weather.wind_intensity = random.randint(0, 100)
        else:
            if 25 <= self.weather.cloudiness <= 50:
                self.weather.precipitation_deposits = random.randint(0, 100)

        self.weather.fog_density = random.randint(0, 50)

```

```
25 def main():
26     (...)
27     weather = Weather(world.get_weather())
28     world.set_weather(weather.weather)
```

Listing 3.3: Code snippet of the weather customisation

Finally, in Figure 3.18, it is shown a collection of images that demonstrates the results of the previous code.



Figure 3.18: Images retrieved from the weather generation function.

## 3.4 Results and analysis

With the final version of the script, it is possible to create a scenario on any of the available maps offered by CARLA with: multiple pedestrians and vehicles, at random places and with different destinations, different weather types, the ability to collect data and following a procedural generation of domain randomisation. For what CARLA offers, it is a script that serves the main goal of receiving the data retrieved from CARLA so that it can be tested on different software. Finally, the following images (Figure 3.19) are a collection of the output that the script creates on the server side.




Figure 3.19: Images retrieved from the main script. The bottom images are from the perspective of the sensor implemented, whilst the remaining were taken manually using the spectator.

# Chapter 4

# Odometry, an extra case of study

Visual odometry is the process of calculating the position and orientation from a motion sensor's data over time. It is a process also used in many of Continental's products. The concept of optical flow is quite similar to several areas, so I choose to evaluate it by using ORBSLAM2 based on KITTI database which is a state of the art in this particular domain. The goal is for the synthetic data retrieved from CARLA to be tested on SIL environment with the open-source algorithm, ORBSLAM2. The evaluation method used is based on The KITTI Vision Benchmark Suite. This chapter serves as an extra step added by me, in order to have more information on the performance of CARLA's synthetic data.

## 4.1 ORBSLAM2

ORBSLAM2 [60] is a Simultaneous Localisation and Mapping (SLAM) system that operates in real time, in small and large, indoor and outdoor environments for monocular, stereo and RGB-D cameras with feature matching being performed by ORB [61] (Oriented FAST [62] and Rotated BRIEF [63]), a feature that builds on the FAST keypoint detector and BRIEF descriptor. ORB-SLAM2 incorporates three threads that run in parallel: tracking, local mapping and loop closing. The tracking is in charge of localising the camera with every frame and deciding when to insert a new keyframe. It performs first an initial feature matching with the previous frame and optimises the pose. Once there is an initial estimation of the camera pose and feature matching, a local visible map is retrieved. Then, the matches with the local map points are searched by re-projection, and camera pose is optimised. Finally, the tracking thread decides if a new keyframe is inserted. The local mapping processes new keyframes to achieve an optimal reconstruction in the surroundings of the camera pose. New correspondences for unmatched ORB in the new keyframe are searched in connected keyframes in the co-visibility graph to triangulate new points. The local mapping is also in charge of culling redundant keyframes. Summing up, mapping and tracking are used for place recognition to perform frame rate re-localisation and loop detection. The loop closing searches for loops with every new keyframe, by analysing the similarity and accumulated

drift [64]. The Figures 4.3 and 4.1 shows what ORBSLAM2 outputs to the user during its analysis. The output file returns each keyframe's 3D position and the *Euler* angles in quaternions, at a specific time-frame.

Visual SLAM will be performed just by using a monocular camera, allowing for a simple setup to be tested on CARLA and extract data from it. However, it comes with disadvantages that are important to be acknowledged. By using just a monocular camera, depth is not observable therefore, the scale of the map becomes unknown. The initial map cannot be triangulated from the very first frame and will suffer scale drift. Such issues will be tackled in the sub-chapter 4.3 and 4.4. The average performance of ORBSLAm2 with KITTI dataset is in Figure 4.2.




Figure 4.1: ORBSLAM2 analysis frame by frame in search of match points.

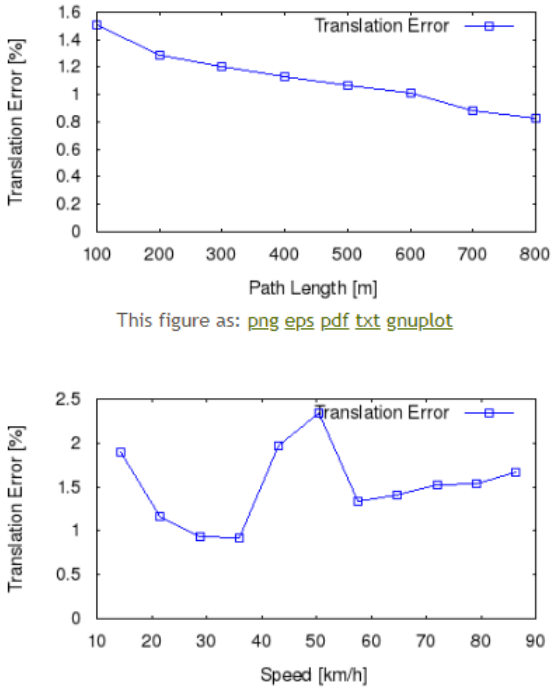



Figure 4.2: ORBSLAM2 average performance based on KITTI's evaluation, retrieved in [4]

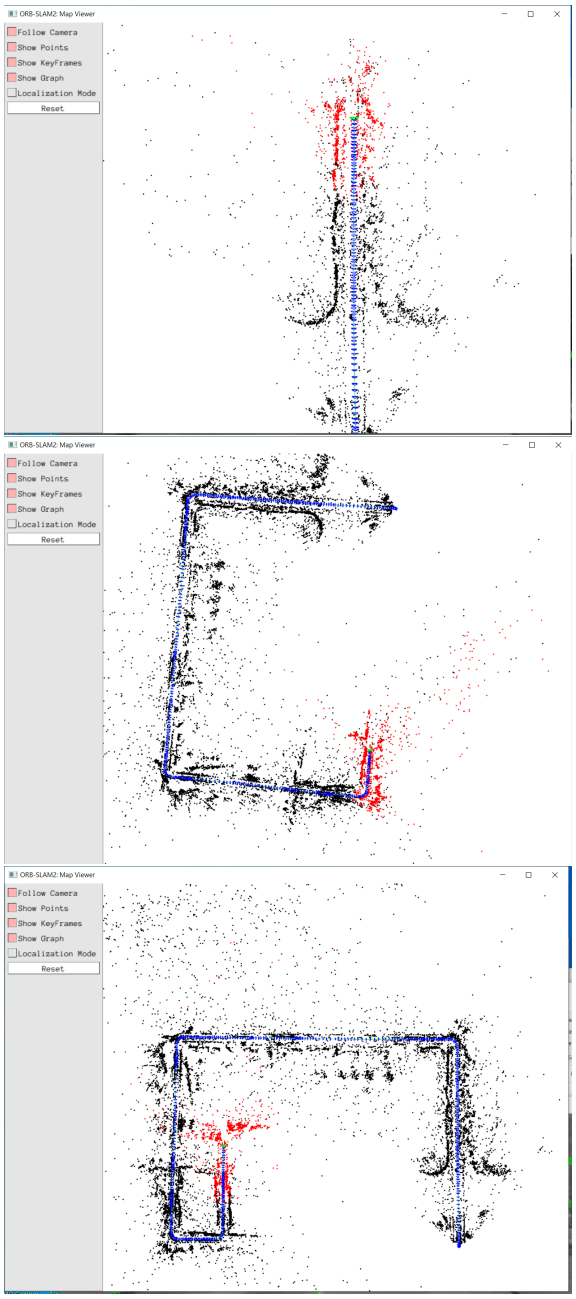



Figure 4.3: ORBSLAM2 map viewer. Evolution from left to right of the surroundings and trajectory of the ego vehicle.

## 4.2 Adaptation of the main script

The script created in the sub-chapter 3.3 needs to be adapted for ORBSLAM2. The additions to the script comes in two groups:

1. **ORBSLAM2 input**: It is required for CARLA's main script to generate a text file with the timestamp of each frame and save the synthetic images into a folder with the name of each file being numerically incremented. The first frame starts with *000000.png*, for instance.

2. **Evaluation**: To compare the results from the generated output file of ORBSLAM2, it is necessary to have a ground truth of the trajectory from the ego vehicle in CARLA to measure deviations created by the analysis of ORBSLAM2.

Starting with the first group, saving the images incrementally has been already implemented, as it can be seen at Section 3.3.4 where the variable $x$ is incremented each time the function *saving()* is called. The time-frame is calculated in each step of the simulation. Each time that the method *sensor_listen()* is called, the current timestamp will be sent along with the sensor data. When attributed to a CPU thread it will save the timestamp in the array *timestamp*. Lastly, when the simulation has ended, it is created a text file (*times.txt*) in which all the existing time-frames from the array will be dumped into it. It is also important to update the camera's extrinsic parameters in ORBSLAM2. The script calculates the focal length ($f_x$ and $f_y$) and the optical centre ($c_x$ and $c_y$), in pixels (Equations 4.1, 4.2 and 4.3).

$$f_x = f_y = image_width/(2.0 * \tan(Field_of_View * \pi/360.0)) \tag{4.1}$$

$$c_y = image_height/2.0 \tag{4.2}$$

$$c_x = image_width/2.0 \tag{4.3}$$

In order to obtain the ground truth of the trajectory, it was made an extensive use of *carla.Transform* to obtain the exact location of the ego vehicle when the method *sensor_listen()* is called. The initial preparation is the same as the timestamp, in which, the coordinates are saved in an array. The following code excerpt shows the code developed for this initial task.

```
1 def sensor_callback(sensor_data, sensor_queue, timer, all_vehicle_actors):
2     ego_location = all_vehicle_actors[0].get_transform()
3     sensor_queue.put((sensor_data, timer, ego_location))
4
5 def main():
6     (...)
7     sensor.listen(lambda data: sensor_callback(data, sensor_queue, timer,
      all_vehicle_actors))
8     (...)
9     while True:
```

```
        if sensor_queue.qsize() > 0:
                s = sensor_queue.get(True, 0.01)
                timestamp.append(s[1])
                if s[2].rotation.yaw < 0:
                    s[2].rotation.yaw = s[2].rotation.yaw + 360
                    location.append(Ego(-s[2].location.x, s[2].location.y, s[2].
    location.z, s[2].rotation.yaw,s[2].rotation.roll, s[2].rotation.pitch))
                    f = executor.submit(saving, s, i)
                    i = i + 1
                timer += 1 / 60
```

Listing 4.1: Code snippet of preparing ground truth data.

In Figure 4.4 and Figure 4.5, it is demonstrated how ORBSLAM2 interprets the first frame it receives in terms of perspective. What is important to emphasise is that what it will be extrapolated as ground truth must not have as a reference CARLA's map coordinates but the location and perspective of the first frame that will be analysed. When ORBSLAM2 analyses the first frame, it will set it as the origin point for the beginning of the projected trajectory. What this means is that the information of the first image, in terms of perspective, will follow along the frontal axis, no matter the perspective of the camera in CARLA (1, 2, 3 and 4).

For that reason, the creation of a ground truth will pass through two stages. Initially, the location of the first frame (variable *origin*) will subtract with all the elements of the array, in order to, move the initial point of the trajectory to the origin, as it can be see in the following code excerpt.

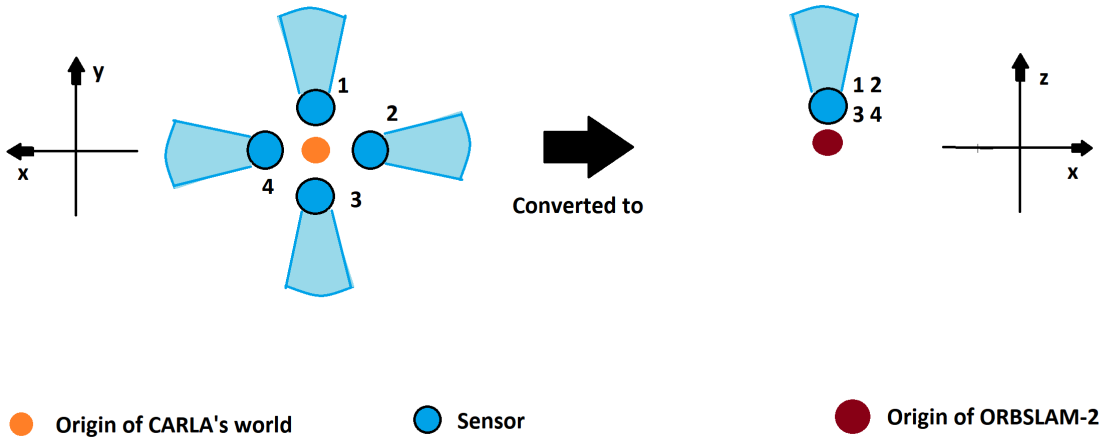



Figure 4.4: Perspective of CARLA and ORBSLAM2.

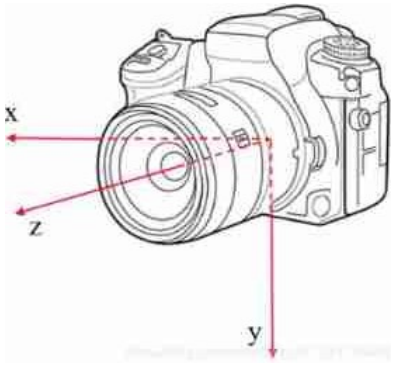



Figure 4.5: ORBSLAM2 axis reference.

```
def main():
    (...)
    reformed_location = []
        origin = location[0]
        i = 0
        for x in location:
            if x is not None:
                reformed_location.append(Ego(x.x - origin.x, x.y - origin.y,
                                             x.z - origin.z, x.yaw,
                                             x.roll,
                                             x.pitch)
```

Listing 4.2: Code snippet about starting the trajectory on the origin.

Next, the following code performs a rotation in the *Euclidean* space to correct the issue of the sensor perspective showcased in the Figure 4.4. The Equations 4.4 and 4.5 are used to correct the initial trajectory points to the correct ones.

$$x' = x\cos(\theta) - y\sin(\theta) \tag{4.4}$$

$$y' = x\sin(\theta) + y\cos(\theta) \tag{4.5}$$

```
def main():
    (...)
    newloc = []

    for x in reformed_location:
        theta = math.radians(location[0].yaw - 90)
        xx = x.x
        yy = x.y
        x1 = xx * math.cos(theta) - yy * math.sin(theta)
        y1 = xx * math.sin(theta) + yy * math.cos(theta)
        newloc.append(Ego(x1, y1,
                          x.z, x.yaw,
                          x.roll,
```

```
14                                  x.pitch))
```

Listing 4.3: Code snippet of the transformation performed.

In the end, the corrected trajectory is saved in a text file, *GT_coordinates.txt*. Each line contains the following: Time-frame, X coordinate, Y coordinate, Z coordinate, Yaw angle, Roll angle and Pitch angle. The Figure 4.6 shows the final result of correcting the trajectory.

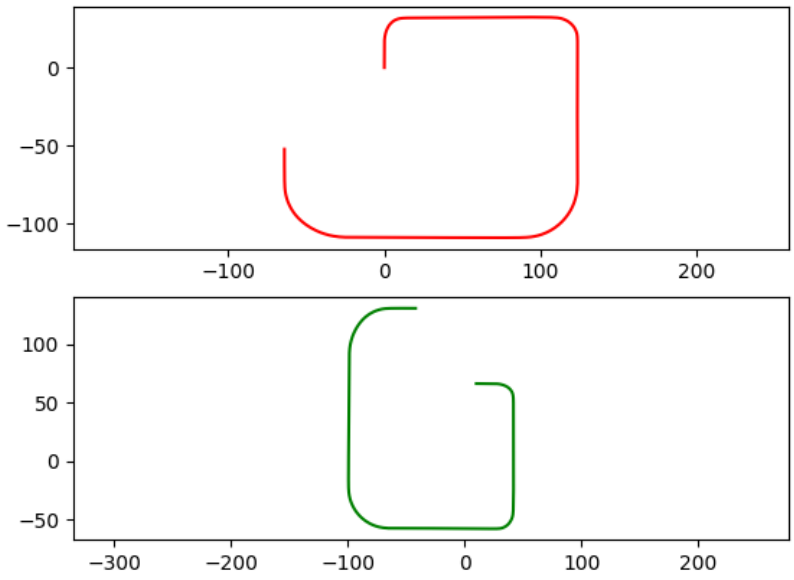



Figure 4.6: The green path corresponds to the trajectory based on CARLA's world and the red path is the corrected trajectory, following ORBSLAM2 perspective.

## 4.3 Evaluation Metrics

The KITTI's evaluation is done by computing the average of all the relative relations at a fixed distance. The translation and rotation error is evaluated as a function of the trajectory length and velocity. This allows for deeper insights into the qualities and failure modes of individual methods. The metrics used stands as it follows (Figure 4.7): F is a set of frames (i, j), $\hat{p} \in SE$ and $p \in SE$ are estimated and true camera poses respectively, $\Theta$ denotes the inverse compositional operator and $\angle$ is the rotation angle.

The script created for the evaluation is based on the following steps:

1. **Retrieve data**: Get the data from the *GT_coordinates.txt* and *KeyFrameTrajectory.txt* and save it.

2. **Correct the Z axis**: ORBSLAM's Z axis corresponds to the Y axis of ORBSLAM2, as shown if FIgure . For simplicity, the data related to ORBSLAM will be organised to X,

$$E_{rot}(\mathcal{F}) = \frac{1}{|\mathcal{F}|} \sum_{(i,j)\in\mathcal{F}} \angle\left[(\hat{\mathbf{p}}_j \ominus \hat{\mathbf{p}}_i) \ominus (\mathbf{p}_j \ominus \mathbf{p}_i)\right]$$

$$E_{trans}(\mathcal{F}) = \frac{1}{|\mathcal{F}|} \sum_{(i,j)\in\mathcal{F}} \left\|(\hat{\mathbf{p}}_j \ominus \hat{\mathbf{p}}_i) \ominus (\mathbf{p}_j \ominus \mathbf{p}_i)\right\|_2$$

Figure 4.7: KITTI's metrics, retrieved in [5]

Z, Y, so that the values X and Z follow the same reference as X and Y from the data of CARLA.

3. **Re-scale**: Since ORBSLAM2 outputs the wrong scale when using monocular mode, it is important to scale up the values for a better correlation with CARLA's values. The scale factor is based on the elapsed distance of trajectory of the ground truth and the of ORB-SLAM. The scale factor will then be applied to the values of ORBSLAM.

4. **Transform into a matrix**: The quaternions from ORBSLAM2 and the euler angles from the ground truth of CARLA are transformed into a rotation matrix and placed together with the XYZ coordinates ($(\psi)$ is the roll angle, $(\phi)$ is the yaw angle and $(\theta)$ is the pitch angle):

$$\begin{bmatrix} [\cos(\phi)\cos(\psi)] & [\sin(\theta)\sin(\phi)\cos(\psi) - \cos(\theta)\sin(\psi)] & [\cos(\theta)\sin(\phi)\cos(\psi) + \sin(\theta)\sin(\psi)] & Tx \\ [\cos(\phi)\sin(\psi)] & [\sin(\theta)\sin(\phi)\sin(\psi) + \cos(\theta)\cos(\psi)] & [\cos(\theta)\sin(\phi)\sin(\psi) - \sin(\theta)\cos(\psi)] & Ty \\ \sin(\phi) & [\sin(\theta)\cos(\phi)] & [\cos(\theta)\cos(\phi)] & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.6}$$

5. **KITTI Evaluation**: As previously mentioned, the goal of the evaluation is to get the average error at each fixed distance. The rotation and translation errors are calculated separately based on length and velocity.

6. **Show the results**: Collect all the data and display it. The Figure 4.8 shows the final result of the evaluation script.

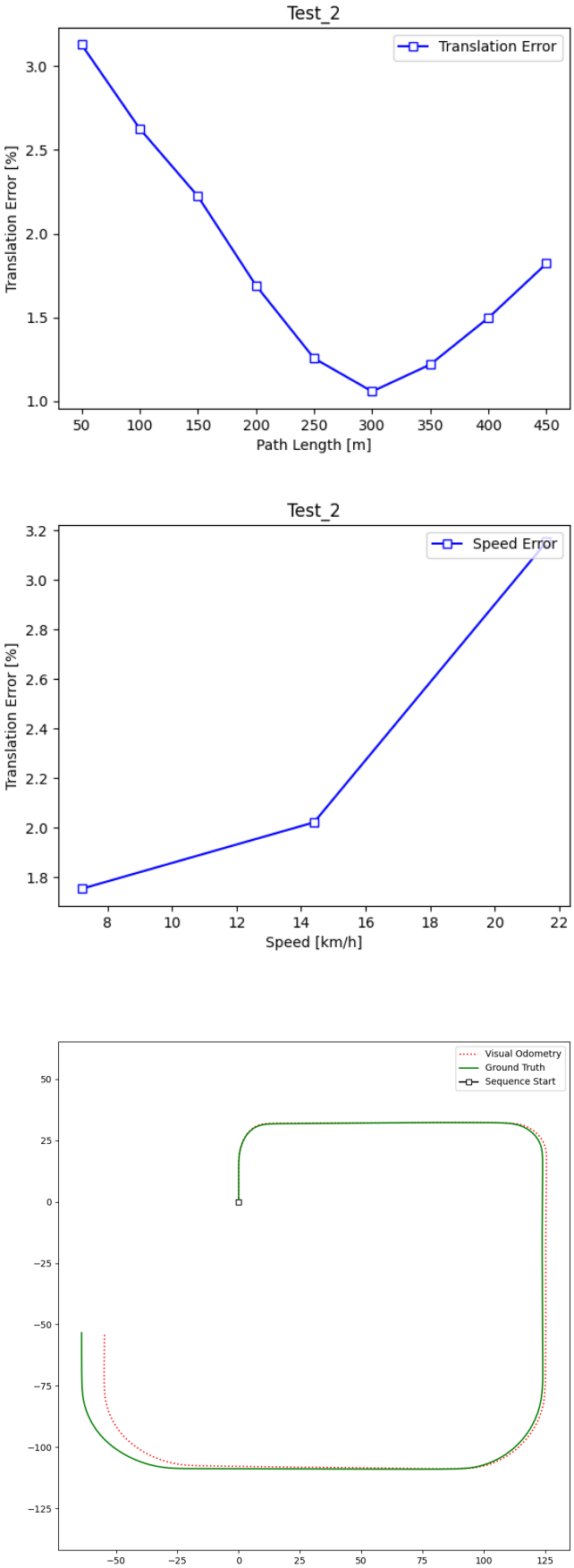



Figure 4.8: Resulting images of the evaluation script.

## 4.4 Results

To tackle some of the problems for using monocular mode, it was decided to evaluate only the translation error on a 2D dimension, in a bird's eye view. That means, the Y axis of ORBSLAM2 and the Z axis of CARLA's ground truth will not be taken into account. Therefore, the matrix that is going to be used for evaluation, will suffer some corrections:

- Yaw, Pitch and Roll angles will be at zero degrees.
- The elevation variation for either ORBSLAM2 and CARLA will be rounded to zero.
- With the previous conditions the resulting matrix will become the following:

$$\begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.7}$$

The results are divided into two groups. The first group has sequences of 500 meters. The second group pushes the simulated sequences to 2000 meters. Sections 4.4.1 and 4.4.2 has recorded data, in accordance with KITTI dataset, at 10 FPS with a width of 1277 pixels and a height of 370 pixels.

### 4.4.1 Discussion of the 500 meters sequences

For a simple starting point for evaluation, a lower distance travelled allows for a much minor impact to the calculated error from the scale drift. Also, by containing the simulation to a few meters, it is much easier to pinpoint situations that impacted the error calculation.

Currently, there are six trajectories analysed: sequence01 (Figure 4.9), Sequence02 (Figure 4.10), sequence03 (Figure 4.11), sequence04 (Figure 4.14), Sequence05 (Figure 4.12) and Sequence06 (Figure 4.13).

The Sequences 1,3,4 and 6 are urban scenarios whilst sequences 2 and 5 are trajectories surrounded with a bigger amount of vegetation, open spaces and, in Sequence 2, highways. Overall, the current performance is to be expected to worsen but in accordance with the average error trend of ORBSLAM2 found in KITTI's website.

For example, the calculation of the movement from point A to B is harder to estimate at higher speeds because of a lower frame-rate and fewer close points to be tracked, as such Sequence02 is a perfect example for this case, since the trajectory is done on a highway. Curiously, Sequence04 failed to be analysed because the trajectory was done in an open space and on a highway with fewer tracking points (Figure 4.16). Therefore, it leads to a complete lack of orientation and an incorrect estimate of the route that was done in CARLA.

The Sequence03 has a bigger translation error, most likely being a significant presence of curves and counter curves, similarly to Sequence05, and junctions with a lot of empty space (Figure 4.15).

Finally, the Sequence05 also has a higher translation error due to the trajectory being surrounded by vegetation (Figure 4.17), like pine trees and wheat fields. Vegetation in CARLA is not the best in terms of textures and unique models, therefore, it is likely that ORBSLAM2 finds it difficult to track each match point located in the vegetation from the previous frame to the next one, due to not having a higher detail in texture, shadows and edges.

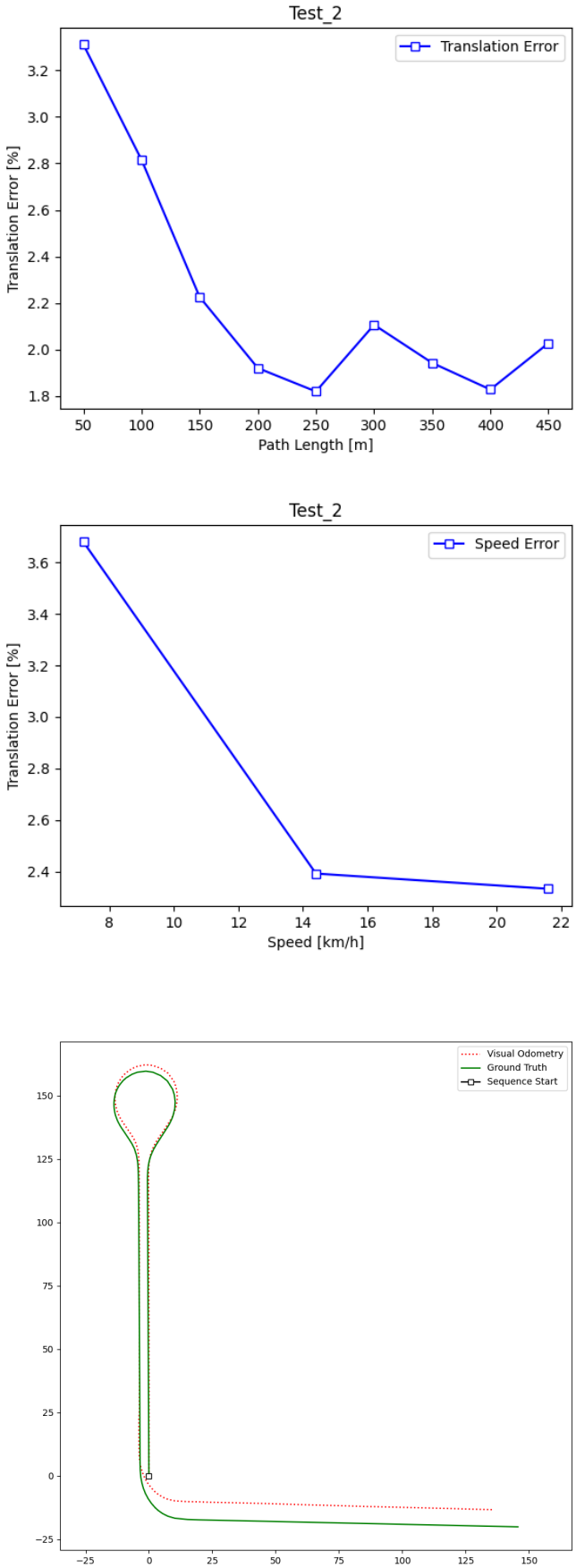



Figure 4.9: Resulting output for Sequence01.

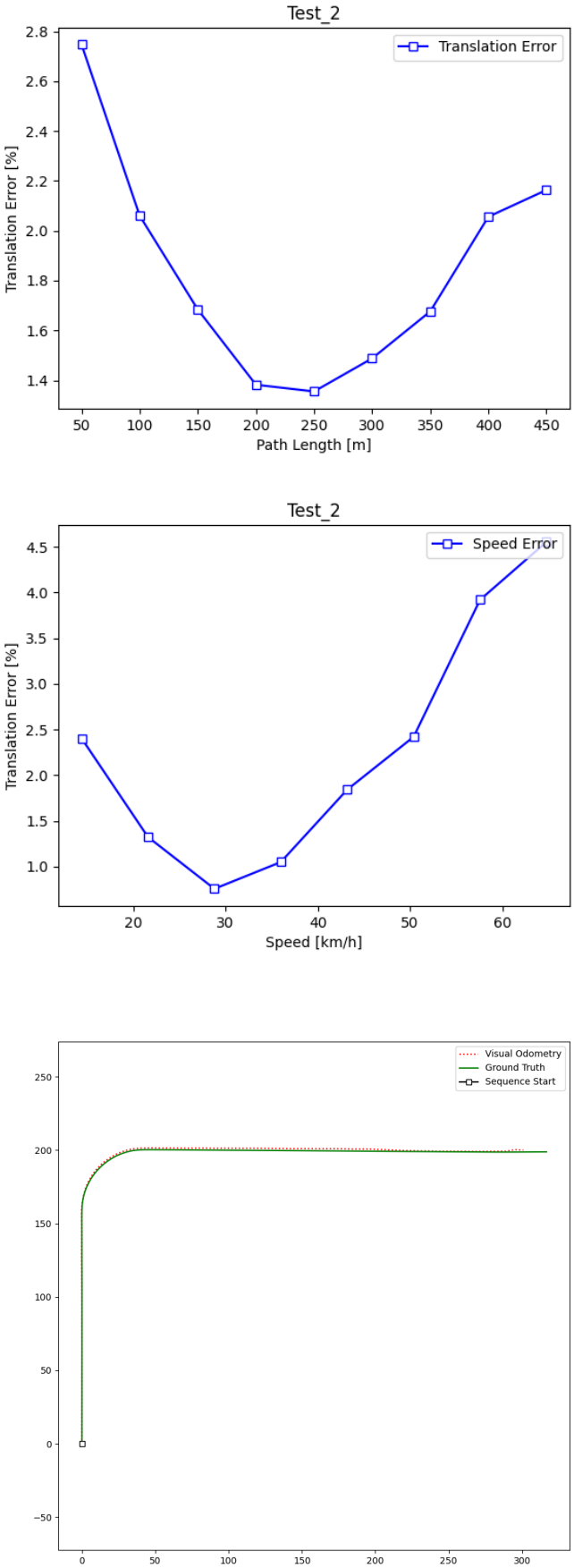



Figure 4.10: Resulting output for Sequence02.

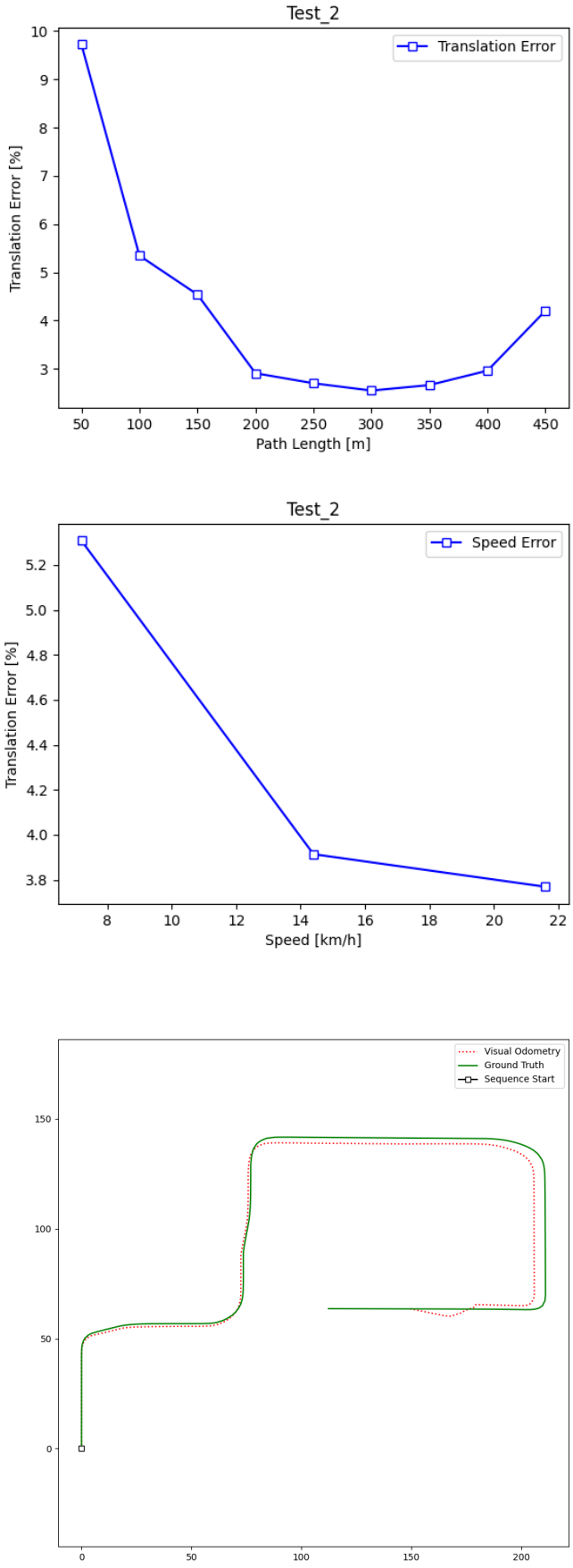



Figure 4.11: Resulting output for Sequence03.

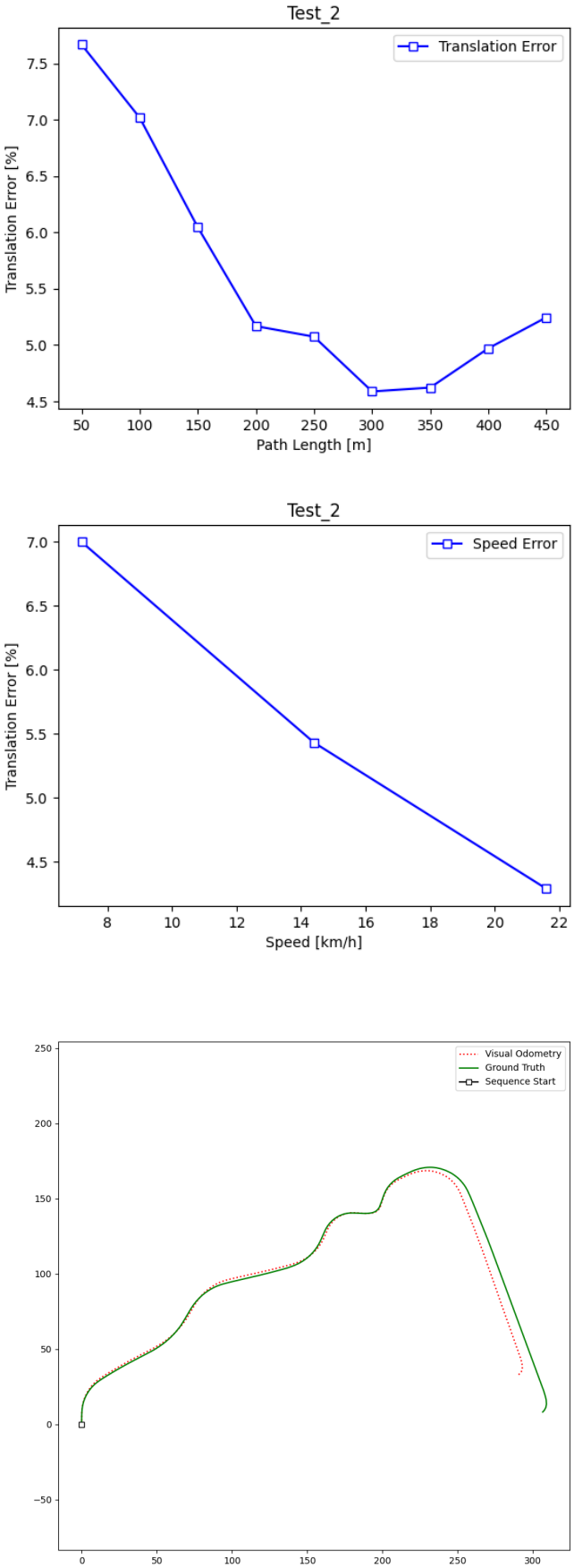
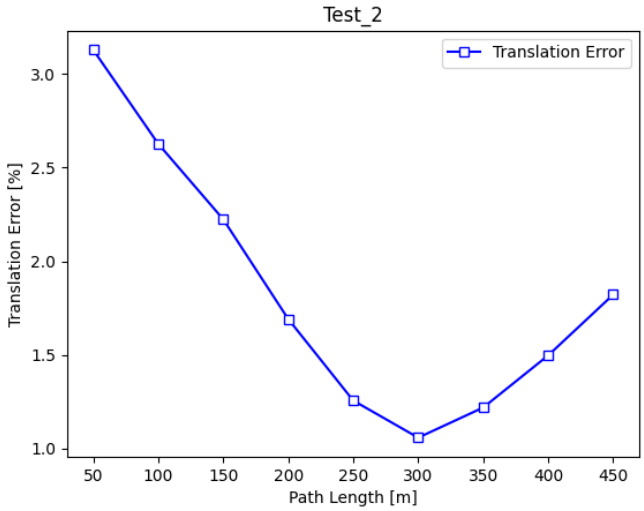
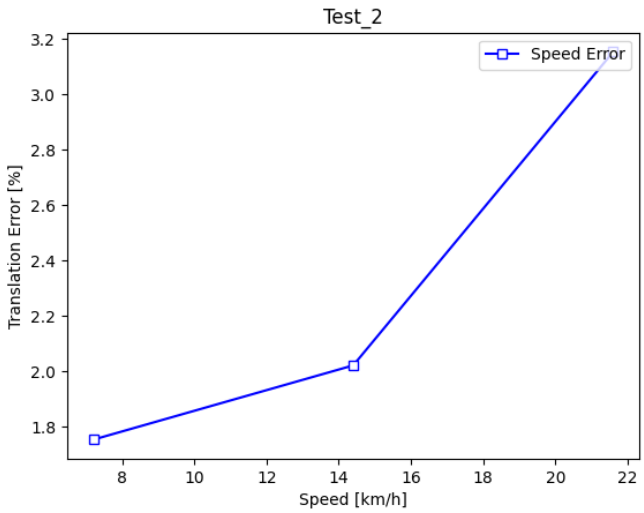



Figure 4.12: Resulting output for Sequence05.



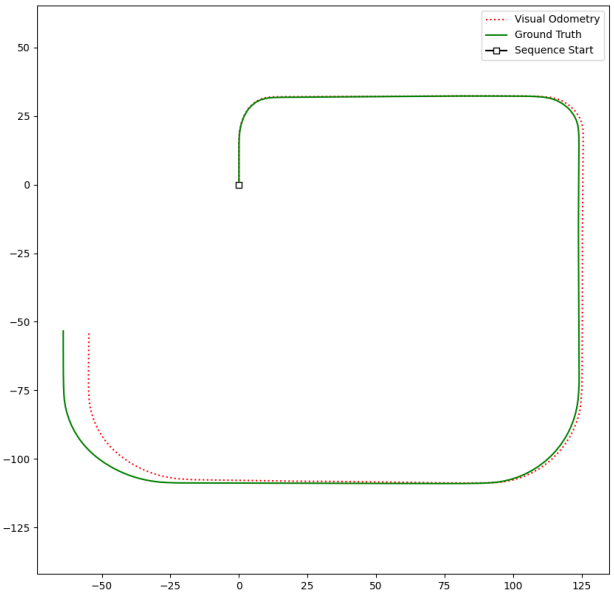




Figure 4.13: Resulting output for Sequence06.

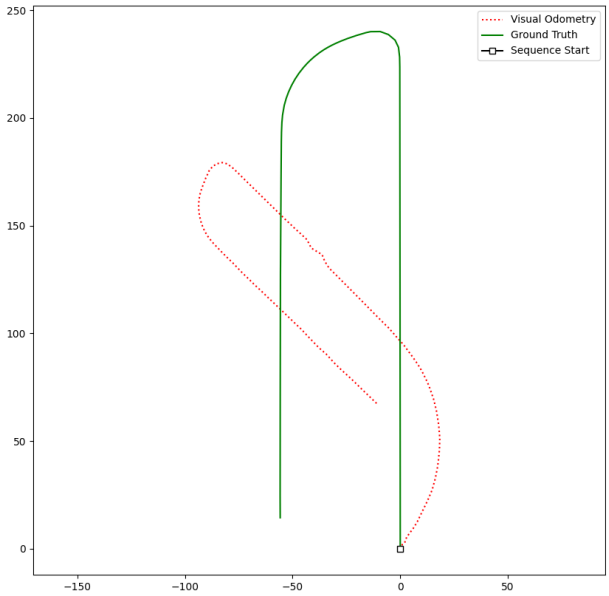



Figure 4.14: Resulting output for Sequence04.



Figure 4.15: Example of the surrounding environment of Sequence03.



Figure 4.16: Example of the surrounding environment of Sequence04.



Figure 4.17: Example of the surrounding environment of Sequence05.

### 4.4.2 Discussion of the 2000 meters sequences

Next, it was decided to test the synthetic data from CARLA on a longer time-frame and distance. It is expected a bigger scale drift due to monocular mode and ORBSLAM2's loop closing feature coming into play as the newer keyframes become similar, allowing ORBSLAM2 to recognise the repetitiveness. The sequences 7 (Figure 4.18) and 8 (Figure 4.19) are based on urban scenarios, whilst sequence 10 (Figure 4.21) is based on a highway environment and sequence 9 (Figure 4.20) on a rural setting.

Overall, it is quite surprising how ORBSLAM2 creates trajectories very similar in comparison with the existing ones from the ground truth. The translation error based on length has a equivalent trend when compared to the 500 m sequences. Because of loop closing and re-mapping techniques, most complex sequences such as these, still have an error akin to the 500 m. Also, the error based on speed is much more detailed since, with larger routes the ego vehicle has the chance to face multiple speed limits for each road that it travels, creating more data diversity.

The most notable example of scale drift happens in Sequence09. Curiously, this sequence is based on the same scenario as Sequence05, with a lot of vegetation and rural environment surrounding the trajectory therefore also affecting the performance on this test in particular.

This set of sequences also has one that completely failed to be analysed and that is Sequence10. The Figure 4.22 shows the starting point where the analysis of ORBSLAM2 started to fail. As stated, open areas seem to be the major culprit for poor performance. Not only that, but the simulated environment itself is a factor as it can be seen from the low quality of edges and textures on the cliff from the left side and poor quality on rock shaped objects from the right side. Finally, the sequence was recorded on a foggy weather reducing even further the quality of the environment itself, specially at bigger distances.

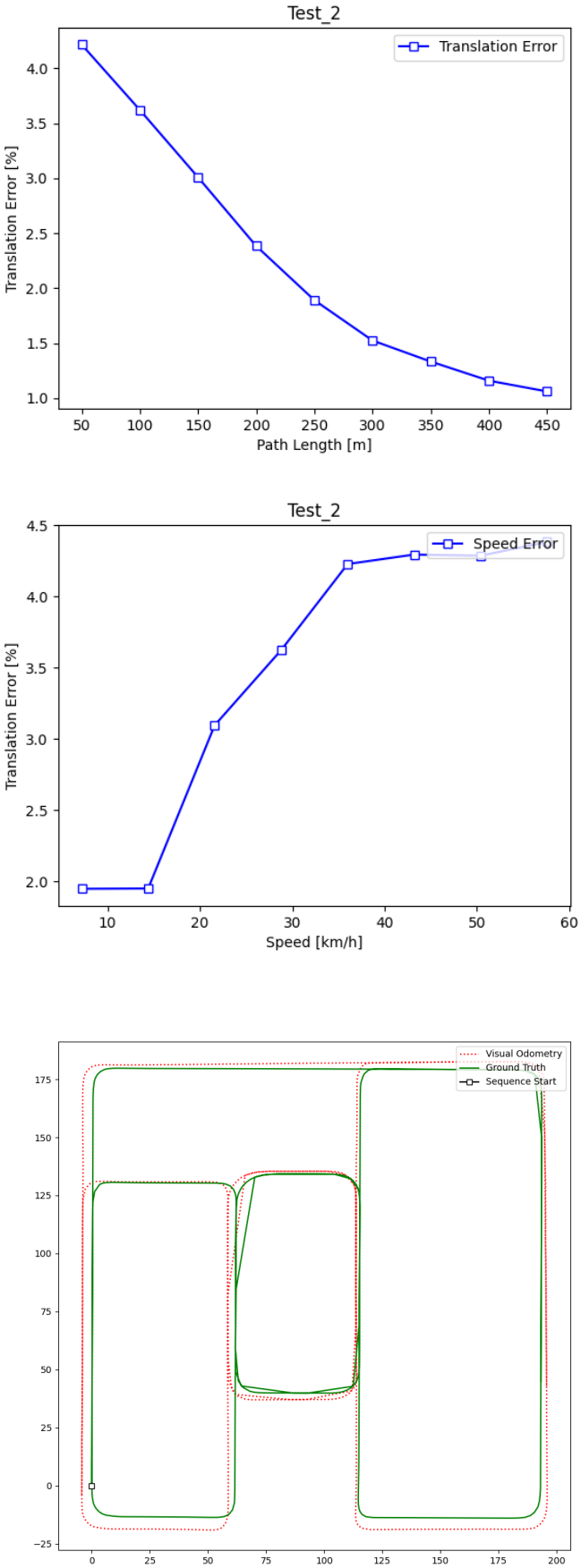



Figure 4.18: Resulting output for Sequence07.

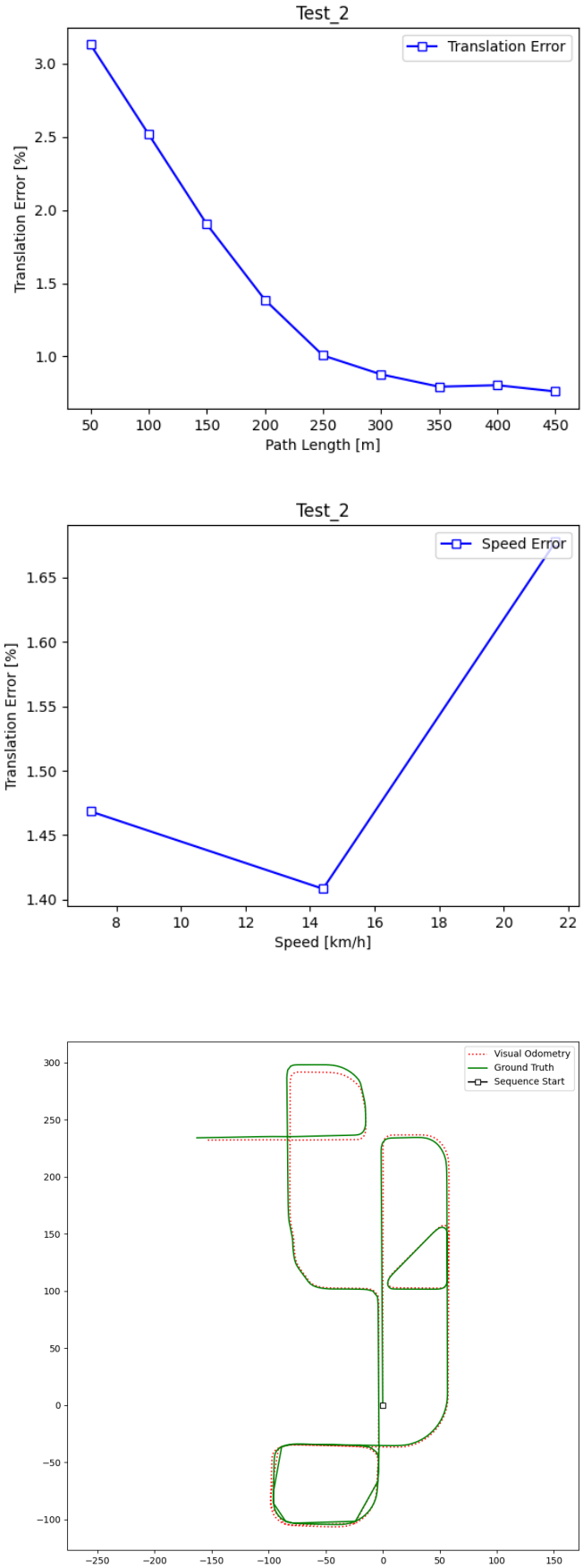



Figure 4.19: Resulting output for Sequence08.

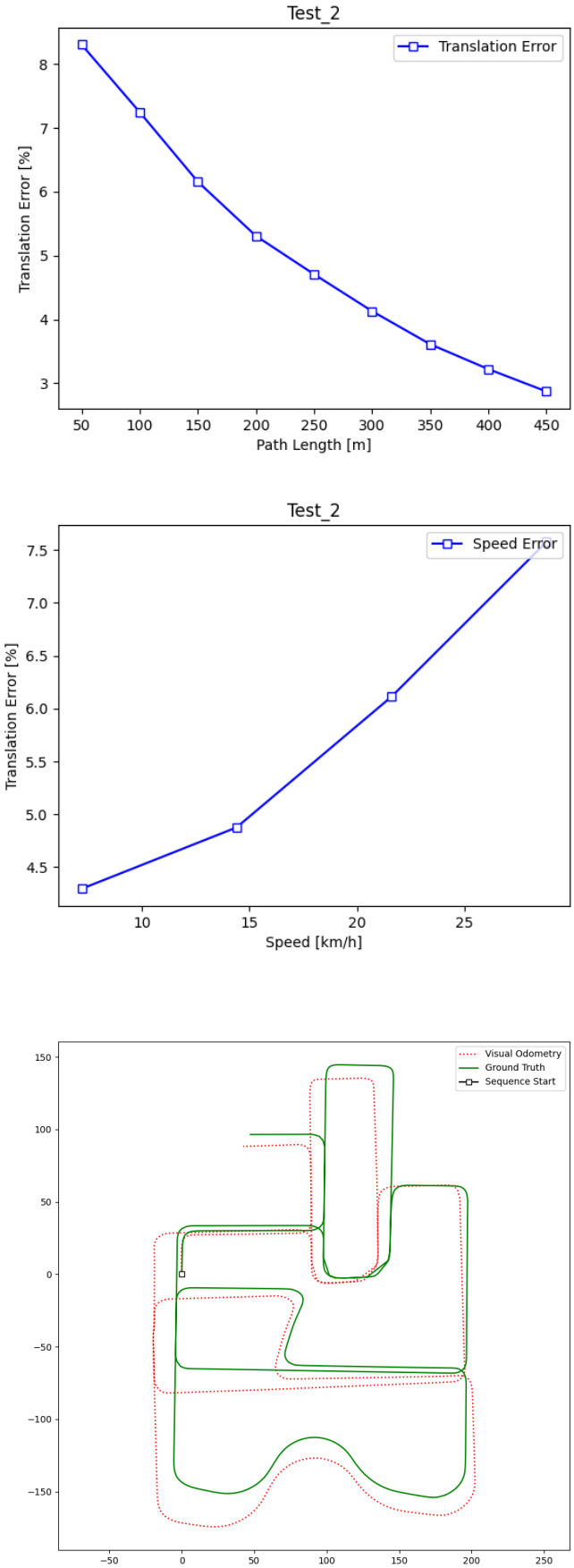



Figure 4.20: Resulting output for Sequence09.

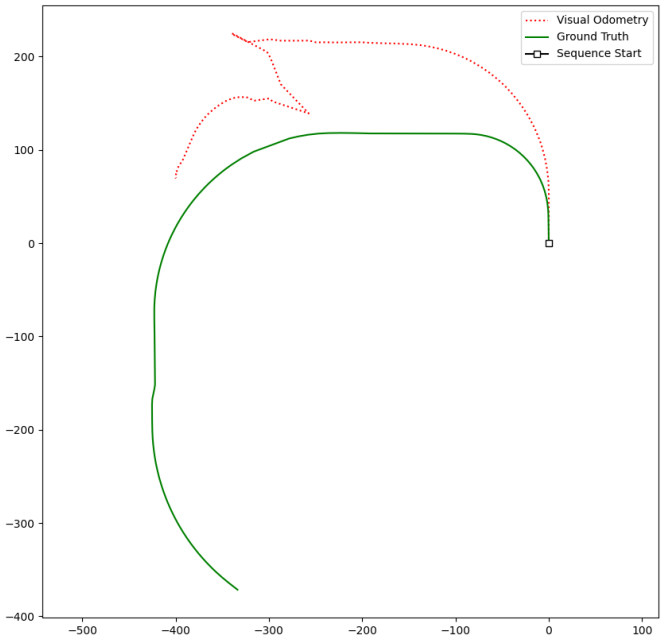



Figure 4.21: Resulting output for Sequence10.

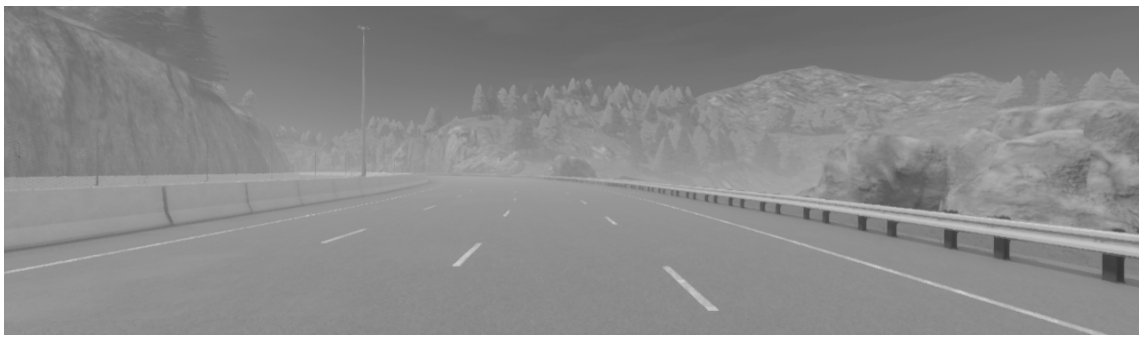


Figure 4.22: Example of the surrounding environment of Sequence10.

### 4.4.3 Final thoughts

The continuation of this task would be to further improve Monocular mode with the addition of a GNSS sensor to correct the drift caused by the usage of a single camera. CARLA's main script would need to be adapted but the foundation for further testing with odometry is there. Furthermore, a different kind of evaluation would be greatly interesting, for example, root-mean-square error (RMSE) would help to understand better in which sections of the trajectory the error was abnormally bigger than its counterparts therefore helping understanding even further what factors affect the quality of synthetic data (shadows, luminosity, edges, textures, and so on).

# Chapter 5

# Continental Framework

This chapter will show the implementation of CARLA's data into the framework of Continental. Initially, it will be explained what consists the software and how it works but in a limited way due to confidential information.

In this chapter, it will be demonstrated the raw results of the algorithm when analysing CARLA's synthetic data, how data is removed from CARLA and from the framework of Continental, in order to be tested. A possible suggestion for evaluating the performance metrics will be displayed in the final sub-chapter.

## 5.1 The software

The Continental's lane detection neural network is a multi-purpose CNN-based image analysis module able for multi-class object detection, semantic segmentation, and lane-marking curve regression. It is now part on most recent Continental's automated driving systems, specifically for camera-based perception tasks. The CNN is an Encoder Network applied with a Single-Shot Detector (SSD) algorithm.

An encoder comprises a network that receives an input and output a feature vector. These feature vector holds the information, the features, that represents the input. The decoder represents once more a network (usually the same network structure as encoder but in opposite orientation) that takes the feature vector from the encoder and gives the best closest match to the actual input or intended output. The encoders are trained with the decoders. The loss function is based on computing the delta between the actual and reconstructed input. The optimiser will try training both encoder and decoder to lower this reconstruction loss. Once trained, the encoder will gives feature vector for input that can be used by the decoder to construct the input with the features that matter the most to make the reconstructed input recognisable as the actual input.

In SSD (Single-Shot Detector) the network is capable to detect all objects within an image in one pass (hence single-shot) through the convolutional neural network (Figure 5.1). SSD divides the image using a grid and each grid cell is responsible for detecting objects, predicting the class and location of an object within that region. If no object is present, it is considered it as the

background class and the location is ignored. Each grid cell in SSD can be assigned with multiple anchor/prior boxes. These anchor boxes are pre-defined and each one is responsible for a size and shape within a grid cell. For example, the swimming pool in the image below corresponds to the taller anchor box while the building corresponds to the wider box.

SSD implements a matching phase while training, to match the appropriate anchor box with the bounding boxes of each ground truth object within an image. Essentially, the anchor box with the most substantial degree of overlap with an object is responsible for predicting that object's class and its location. This property is utilised for improving the network and predicting the detected objects and their locations once the network has been improved. In practice, each anchor box is specified by an aspect ratio and a zoom level.

Not all objects are square in shape. Some are longer and are wider, by varying degrees (Figure 5.2). The SSD architecture allows pre-defined aspect ratios of the anchor boxes to account for this. The ratios parameter can be used to specify the different aspect ratios of the anchor boxes associates with each grid cell at each zoom/scale level.

The receptive field is defined as the region in the input space that a particular CNN's feature is looking at. The receptive field is the key premise of the SSD architecture as it enables to detect objects at various scales and output a narrower bounding box. Because of this, SSD allows us to define a hierarchy of grid cells at various layers. For example, we could use a 4x4 grid to find smaller objects, a 2x2 grid to find mid sized objects and a 1x1 grid to discover objects that cover the entire image [65].

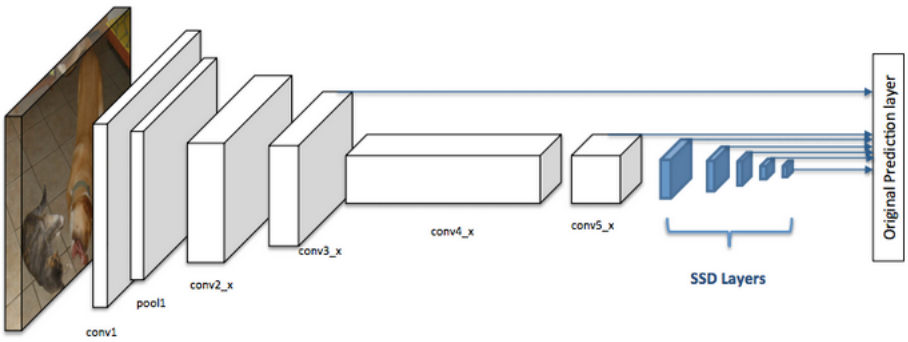



Figure 5.1: Architecture of a CNN with a SSD detector.

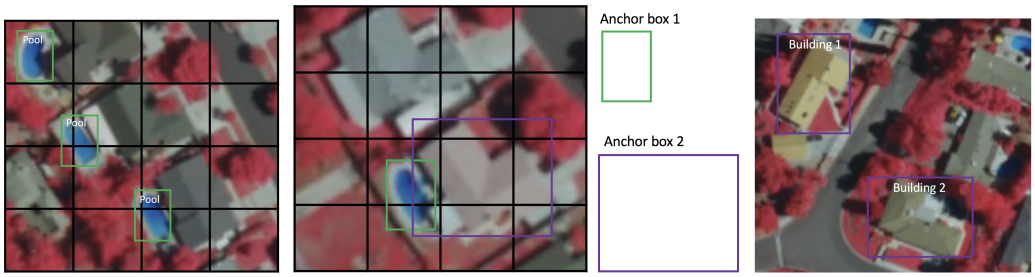



Figure 5.2: From left to right: a 4x4 grid; anchor boxes and bounding boxes.

The lane detection module input data can be sent as a live version or an offline version. The current tests done on it, are based on offline mode. This means that in CARLA, the synthetic data will be retrieved and then transformed from a MP4 extension to a AVI format, instead of being directly injected to the lane detection module. From there, once the AVI file has been selected, the user can choose the algorithm that will output the desired predictions. In this case, the algorithm chosen will detect lanes from the data received from CARLA, frame by frame. The algorithm is capable of detecting not only the ego lane but also the adjacent lanes and if possible road edges. The next sub-chapter will show the visual results of the algorithm when using CARLA's data.

## 5.2 Visual results

As it can be seen in Figure 5.3, the visual results are quite remarkable. Not only it detects both the driving and adjacent lanes but also the road edges. Finally, it serves as a reminder that, it is the first time where virtual information from CARLA is injected in this framework with no modification of the software itself to accommodate CARLA's data.




Figure 5.3: Visual results of the lane detection module.

## 5.3 Extracting the output data

In this section, the data from both software will be extracted for a future evaluation on the performance of the output of Continental's lane detection module when compared with CARLA's ground truth.

Starting with the algorithm of Continental, since the software doesn't directly output data about the 2D points of the detected lanes in each respective frame, it was necessary to modify the software in order to enable us to extract information from it. Once the necessary variables where gathered, namely the x and y pixel coordinates of each detection and the respective frame, they were saved in a text file. The Figure 5.4 shows the results for the first frame, Frame 0.




Figure 5.4: Detected points from Frame 0 by the lane detection algorithm.

For CARLA, to detected a point in a 3D space, redefine it to the camera perspective and convert it to a 2D plane, it is necessary to use concepts from the pinhole camera model, such as intrinsic and extrinsic parameters from the RGB camera. During the simulation, it is collected the waypoints of the lane that the main vehicle is driving and then ask for each waypoint, the respective left lane waypoint and right lane waypoint (Figure 5.5).

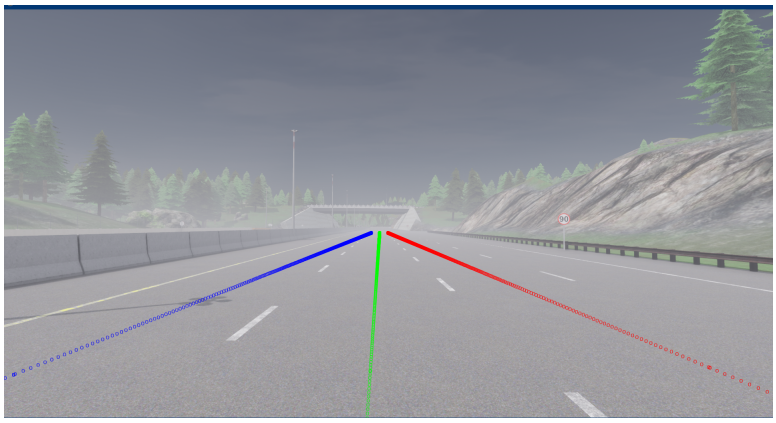


Figure 5.5: Collected waypoints from Frame 0.

Now it is necessary to correct the collected waypoints, so that it represents the left and right lane markings and not the centre of each lane. For that, it is important to collect a sort of offset that corrects the 3D projection of each point. Currently the offset is calculated by getting the vector of the central waypoint, multiplying the lane width and divide the result by two. By correcting the 3D points of the left and right lanes with the calculated offset, the resulting values will be the 3D points representing the lane markings from the left and right side of the driving lane.

The following equation represents the mathematical conversion of each 3D point, $W_p$, to the perspective of the camera, $C_p$ with the help of the Extrinsic parameters of the sensor,$R_c$ (Rotation Matrix) and $T_c$ (3D points).

$$\begin{bmatrix} Cp \\ 1 \end{bmatrix} = \begin{bmatrix} & R_c & & T_c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} Wp \\ 1 \end{bmatrix} \tag{5.1}$$

Next, since the $C_p$ values are from the Unreal Engine 4's coordinate system, it is necessary to convert them to a standard camera coordinate system (the same used by OpenCV for example) (Figure 5.6).

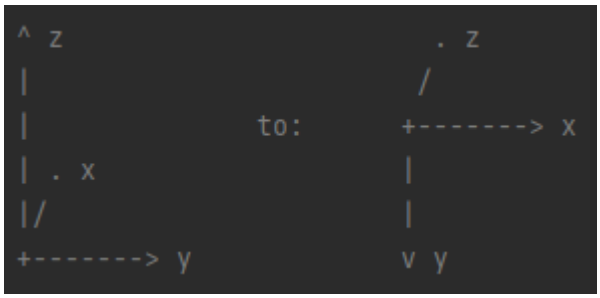



Figure 5.6: Axis conversion from UE to OpenCV standard.

$$\begin{bmatrix} Cp \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \Longrightarrow \begin{bmatrix} Cp \end{bmatrix} = \begin{bmatrix} y \\ -z \\ x \end{bmatrix} \tag{5.2}$$

Finally, the newly converted $C_p$ multiplies with the Intrinsic parameters of the camera and the 3D point is converted to a 2D plane where the values $u$ and $v$ represents the pixel coordinates of the respective 3D position initially retrieved (Figure 5.7).

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{5.3}$$

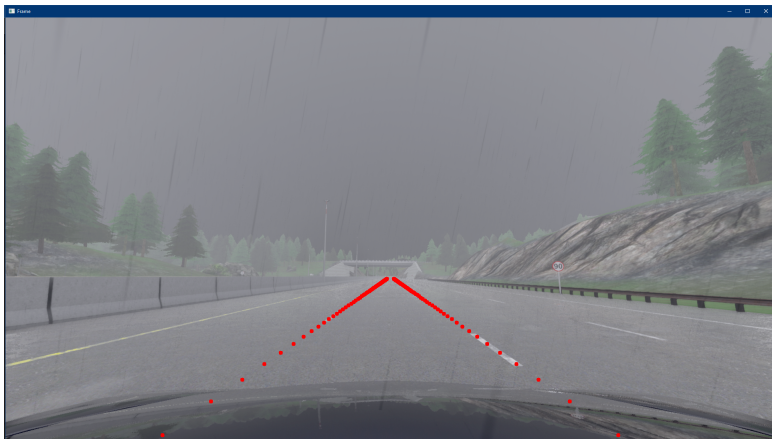


Figure 5.7: Final result of Equation 5.3 being painted over Frame 0 with the help of OpenCV.

## 5.4 Analysis

For the analysis of the algorithm's performance on CARLA's dataset, it was decided to examine the Intersection Over Union (IoU) of the area of the ego lane using the outputs that were retrieved in the previous section. As such, it will be necessary to limit the area of the image that will be analysed by using spline regression.

When we talk about regression, the first things that come to our mind are linear or logistic regression, something that with the current data is not possible. As a result, spline polynomial regression was chosen to derive lane segmentation from the lane detection points. The method of splines consists of dividing up the range of $x$ into segments with joint points called knots. From there, a polynomial of a fixed degree is then fitted to each segment with constraints applied to ensure appropriate continuity at the knots. The application of spline models in lane detection is not new as it can be seen in [66] and [67].

Before applying the spline polynomial regression, it is necessary to remove the isolated points (outliers) that the algorithm detected (Figure 5.4). Since the input dataset is a video of the ego vehicle travelling a straight road, the noise created by the predictions of lane detection can be removed around the horizon and upper part of the image since that region is not important for analysing the data. The resulting images (Figure 5.8) represent the final points from either Continental's algorithm and CARLA that will be examined.

Next, we get the polynomial regression of each group of points (left lane and right lane) with the spline model. The created "straight" lines are then unified by filling the area between them in a triangular shape (basically the representation of the lane) (Figure 5.9).

Since the image is composed of 0 and 1 values, it is applied a Boolean intersection over union with the area of the lane detection output and the area of CARLA's ground truth which enables to get a ratio of occupation between the predicted and the ground truth area.

In the end, CARLA has been successfully integrated with the lane detection framework of Continental and the results are demonstrated in the following table (Table 5.1).

Table 5.1: Average IoU of each group of frames and total average.

| Group of frames (1 second of interval) | Average IoU |
|---|---|
| 1 - 10 | 90.29% |
| 11 - 20 | 94.09% |
| 21 - 30 | 79.47% |
| 31 - 40 | 88.89% |
| 41 - 50 | 87.78% |
| 51 - 60 | 85.78% |
| 61 - 70 | 84.42% |
| 71 - 80 | 74.98% |
| 81 - 90 | 77.67% |
| 91 - 100 | 65.92% |
| Total | 83.28% |




Figure 5.8: Final group of points that will be used to analyse Continental's framework performance on CARLA's dataset. Upper image is from the Continental framework and lower image is from CARLA.

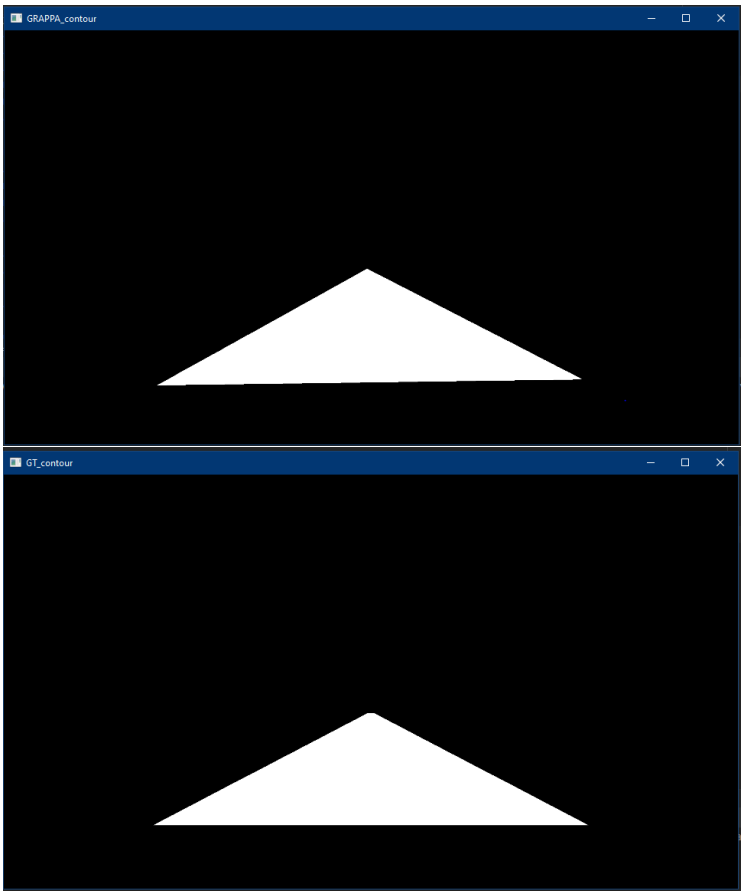



Figure 5.9: Showcase of the filled ego lane. The upper image is from the lane detection algorithm and the lower image is from CARLA.

# Chapter 6

# Conclusions and future work

This thesis introduced some of the main topics associated with autonomous driving like its sensors, as well as their possibility of an implementation into virtual environments in order to solve the problem that current real datasets are facing. Such introduction of concepts were initially foreign and out of my comfort zone, but with the help of both supervisors and the team at Continental, I was able to surpass my initial doubts.

The research into the state of the art of this theme, albeit not fully in depth, helped me in understanding even more the importance of such projects to the scientific community and, in the context of synthetic data generation, a better management of resources by using the digital/virtual world as a means of improving efficiently advanced driver-assistance systems.

Overall, the work developed represents an integration in CARLA of not only the lane detection module of Continental's framework but also with the extra step taken, by using ORBSLAM2 which was not originally planned. There were a lot of issues that needed to be tackled during the adaptation of CARLA's output to be compatible to work with ORBSLAM2 and Continental's framework. In fact, that was one of the aspects that made working on this theme so fulfilling, by applying concepts learnt through out the years as a student and problem solving certain subjects that I didn't fully grasped such as computer vision, the extrinsic and intrinsic parameters of the virtual RGB camera, rotational matrix and pinhole camera model as well technical aspects like multi-threading, synchronisation and data queuing.

Witnessing the completion of the established objectives, it gives a sense of satisfaction on the work done and adds more knowledge of CARLA and its simulation environment that could serve as base for reference in future projects.

## 6.1 Future work

Further development includes the improvement of the main script such as better optimisation or the inclusion of more features from CARLA. The usage of more types of sensors would be critical for a bigger variety of synthetic data, as well as the expansion of procedural generation techniques for a much richer simulated environment. Furthermore, importing maps based on current road

maps as demonstrated in section 3.2.5, would vastly improve the variety of scenarios to be tested for synthetic data retrieval. Such task, however, wouldn't be as simple and it would most likely be very time consuming since the maps would need to be repopulated manually with environment objects such as buildings.

Additionally, for a better adaptation of CARLA to Continental's interest, it might be interesting building CARLA from source for development purposes, allowing for a better customisation and understanding of the software. It would also enable to add extra features or capabilities to CARLA and use the Unreal Editor to create assets or manipulate maps for further improvement over the software.

# References


[1] Sébastien Lozé. Carla democratizes autonomous vehicle R&D with free open-source simulator, 4th of October, 2019. Last acessed on February 20th, 2022. URL: https://www.unrealengine.com/en-US/spotlights/carla-democratizes-autonomous-vehicle-r-d-with-free-open-source-simulator.

[2] Stefan-Alexander Schneider and Kmeid Saad. Camera behavioral model and testbed setups for image-based adas functions. *e i Elektrotechnik und Informationstechnik*, 135, 07 2018. doi:10.1007/s00502-018-0622-7.

[3] Sensors reference. Last accessed February 23rd, 2022. URL: https://carla.readthedocs.io/en/0.9.13/ref_sensors/.

[4] Orb-slam2 evaluation. Last accessed May 23rd, 2022. URL: http://www.cvlibs.net/datasets/kitti/eval_odometry_detail.php?&result=162e869388fd5fd235b88dac7718149241fdc255.

[5] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3354–3361, 2012. doi:10.1109/CVPR.2012.6248074.

[6] Marina Bolotnikova. America's car crash epidemic, September 19th 2021. Last accessed December 20th, 2021. URL: https://www.vox.com/22675358/us-car-deaths-year-traffic-covid-pandemic.

[7] G. Vivo, P. Dalmasso, and F. Vernacchia. The european integrated project "safespot"-how adas applications co-operate for the driving safety. In *2007 IEEE Intelligent Transportation Systems Conference*, pages 624–629, 2007. doi:10.1109/ITSC.2007.4357715.

[8] CORDIS. Low Cost Advanced Driver Assistance Systems (ADAS): A cost affordable solution for improved road safety, 2017. URL: https://cordis.europa.eu/project/id/767509.

[9] CORDIS. Vision inspired driver assistance systems, 2016. URL: https://cordis.europa.eu/project/id/690772.

[10] Eric Thorn, Shawn Kimmel, and Michelle Chaka. *A Framework for Automated Driving System Testable Cases and Scenarios*. National Highway Traffic Safety Administration, 1200 New Jersey Avenue SE, Washington DC, 2018.

[11] Sergey I. Nikolenko. Synthetic data for deep learning. *CoRR*, abs/1909.11512, 2019. URL: http://arxiv.org/abs/1909.11512, arXiv:1909.11512.

[12] A. Takimoglu. What is data augmentation? Techniques, Benefit Examples, April 30th 2021. Last accessed December 20th, 2021. URL: https://research.aimultiple.com/data-augmentation/.

[13] S. Koperniak. Artificial data give the same results as real data without compromising privacy, March 3rd 2017. Last accessed December 20th, 2021. URL: https://news.mit.edu/2017/artificial-data-give-same-results-as-real-data-0303.

[14] German Ros, Laura Sellart, Joanna Materzynska, David Vazquez, and Antonio Lopez. *The synthia dataset: A large collection of synthetic images for semantic segmentation of urban scenes*. IEEE, 2016. URL: https://doi.org/10.1109/CVPR.2016.352.

[15] Stephan R. Richter, Vibhav Vineet, Stefan Roth, and Vladlen Koltun. Playing for data: Ground truth from computer games. *CoRR*, abs/1608.02192, 2016. URL: http://arxiv.org/abs/1608.02192, arXiv:1608.02192.

[16] Braden Hurl, Krzysztof Czarnecki, and Steven L. Waslander. Precise synthetic image and lidar (presil) dataset for autonomous vehicle perception. *CoRR*, abs/1905.00160, 2019. URL: http://arxiv.org/abs/1905.00160, arXiv:1905.00160.

[17] Stephan R. Richter, Zeeshan Hayder, and Vladlen Koltun. Playing for benchmarks. *CoRR*, abs/1709.07322, 2017. URL: http://arxiv.org/abs/1709.07322, arXiv:1709.07322.

[18] Matthew Johnson-Roberson, Charles Barto, Rounak Mehta, Sharath Nittur Sridhar, Karl Rosaen, and Ram Vasudevan. *Driving in the Matrix: Can virtual worlds replace human-generated annotations for real world tasks?* IEEE, 2017. URL: https://doi.org/10.1109/ICRA.2017.7989092.

[19] Alexey Dosovitskiy, Germán Ros, Felipe Codevilla, Antonio M. López, and Vladlen Koltun. CARLA: an open urban driving simulator. *CoRR*, abs/1711.03938, 2017. URL: http://arxiv.org/abs/1711.03938, arXiv:1711.03938.

[20] Prabhjot Kaur, Samira Taghavi, Zhaofeng Tian, and Weisong Shi. A survey on simulators for testing self-driving cars. *CoRR*, abs/2101.05337, 2021. URL: https://arxiv.org/abs/2101.05337, arXiv:2101.05337.

[21] Martin Holen, Kristian Knausgård, and Morten Goodwin. An evaluation of autonomous car simulators and their applicability for supervised and reinforcement learning. In Filippo Sanfilippo, Ole-Christoffer Granmo, Sule Yildirim Yayilgan, and Imran Sarwar Bajwa, editors, *Intelligent Technologies and Applications*, pages 367–379, Cham, 2022. Springer International Publishing.

[22] Daniele Loiacono, Alessandro Prete, Pier Luca Lanzi, and Luigi Cardamone. Learning to overtake in torcs using simple reinforcement learning. In *IEEE Congress on Evolutionary Computation*, pages 1–8, 2010. doi:10.1109/CEC.2010.5586191.

[23] Nuno Calheiros Cruz Vidigal. *Plataforma para o desenvolvimento de algoritmos de condução autónoma de veículos terrestres*. Faculdade de Engenharia da Universidade do Porto, 2018.

[24] Panpan Cai, Yiyuan Lee, Yuanfu Luo, and David Hsu. SUMMIT: A simulator for urban driving in massive mixed traffic. *CoRR*, abs/1911.04074, 2019. URL: http://arxiv.org/abs/1911.04074, arXiv:1911.04074.

[25] Birgit Schlager, Stefan Muckenhuber, Simon Schmidt, Hannes Holzer, Relindis Rott, F. Maier, Kmeid Saad, Martin Kirchengast, Georg Stettinger, Daniel Watzenig, and Jonas Ruebsam. State-of-the-art sensor models for virtual testing of advanced driver assistance systems/autonomous driving functions. *SAE International Journal of Connected and Automated Vehicles*, 3:233–261, 10 2020. doi:10.4271/12-03-03-0018.

[26] Stefan Muckenhuber, Hannes Holzer, Jonas Rübsam, and Georg Stettinger. Object-based sensor model for virtual testing of adas/ad functions. In *2019 IEEE International Conference on Connected Vehicles and Expo (ICCVE)*, pages 1–6, 2019. doi:10.1109/ICCVE45908.2019.8965071.

[27] João Borrego, Atabak Dehban, Rui Figueiredo, Plinio Moreno, Alexandre Bernardino, and José Santos-Victor. Applying domain randomization to synthetic data for object category detection. *CoRR*, abs/1807.09834, 2018. URL: http://arxiv.org/abs/1807.09834, arXiv:1807.09834.

[28] Jason Brownlee. A gentle introduction to generative adversarial networks (gans), July 19th, 2019. Acessed January 20th, 2022. URL: https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/.

[29] Ashish Shrivastava, Tomas Pfister, Oncel Tuzel, Josh Susskind, Wenda Wang, and Russell Webb. Learning from simulated and unsupervised images through adversarial training. *CoRR*, abs/1612.07828, 2016. URL: http://arxiv.org/abs/1612.07828, arXiv:1612.07828.

[30] Arun Gandhi. Data augmentation | how to use deep learning when you have limited data — part 2. May,2021. Acessed February 14th, 2022. URL:https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/.

[31] Hassan Abu Alhaija, Siva Karthik Mustikovela, Lars M. Mescheder, Andreas Geiger, and Carsten Rother. Augmented reality meets computer vision : Efficient data generation for urban driving scenes. *CoRR*, abs/1708.01566, 2017. URL: http://arxiv.org/abs/1708.01566, arXiv:1708.01566.

[32] Jonathan Tremblay, Aayush Prakash, David Acuna, Mark Brophy, Varun Jampani, Cem Anil, Thang To, Eric Cameracci, Shaad Boochoon, and Stan Birchfield. Training deep networks with synthetic data: Bridging the reality gap by domain randomization. *CoRR*, abs/1804.06516, 2018. URL: http://arxiv.org/abs/1804.06516, arXiv:1804.06516.

[33] Svetozar Zarko Valtchev and Jianhong Wu. *Domain randomization for neural network clasifcation*. Journal of Big Data, 2021.

[34] Stanislaw J. Malec. *Semantic segmentation with Carla simulator*. 2021.

[35] Apostolia Tsirikoglou, Joel Kronander, Magnus Wrenninge, and Jonas Unger. Procedural modeling and physically based rendering for synthetic data generation in automotive applications. *CoRR*, abs/1710.06270, 2017. URL: http://arxiv.org/abs/1710.06270, arXiv:1710.06270.

[36] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. *CoRR*,

abs/1512.02325, 2015. URL: http://arxiv.org/abs/1512.02325, arXiv:1512.02325.

[37] Recommended hardware, a detailed guide on picking suitable hardware for unreal engine development, 2021. Last acessed on June 20th, 2022. URL: https://unrealcommunity.wiki/recommended-hardware-x1p9qyg0.

[38] Carla's github. Last accessed February 23rd, 2022. URL: https://github.com/carla-simulator/carla/blob/master/Docs/download.md.

[39] Building CARLA, F.A.Q. Last accessed February 23rd, 2022. URL: https://carla.readthedocs.io/en/0.9.13/build_faq/.

[40] Pycharm, the python ide for professional developers. Last accessed February 23rd, 2022. URL: https://www.jetbrains.com/pycharm/promo/.

[41] CARLA documentation. Last accessed February 23rd, 2022. URL: https://carla.readthedocs.io/en/0.9.13/.

[42] 1st- world and client. Last accessed February 23rd, 2022. URL: https://carla.readthedocs.io/en/0.9.13/core_world/.

[43] 2nd- actors and blueprints. Last accessed February 23rd, 2022. URL: https://carla.readthedocs.io/en/0.9.13/core_actors/.

[44] Wildcards. Last accessed February 23rd, 2022. URL: https://tldp.org/LDP/GNU-Linux-Tools-Summary/html/x11655.htm.

[45] Python lambda functions. Last accessed February 23rd, 2022. URL: https://www.geeksforgeeks.org/python-lambda-anonymous-functions-filter-map-reduce/.

[46] 4th- sensors and data. Last accessed February 23rd, 2022. URL: https://carla.readthedocs.io/en/0.9.13/core_sensors/.

[47] Asam opendrive. Last accessed February 23rd, 2022. URL: https://www.asam.net/standards/detail/opendrive/.

[48] 3rd- maps and navigation. Last accessed February 23rd, 2022. URL: https://carla.readthedocs.io/en/0.9.13/core_map/.

[49] Opendrive standalone mode. Last accessed February 23rd, 2022. URL: https://carla.readthedocs.io/en/latest/adv_opendrive/.

[50] Recorder. Last accessed February 23rd, 2022. URL: https://carla.readthedocs.io/en/0.9.13/adv_recorder/.

[51] Rendering options. Last accessed February 23rd, 2022. URL: https://carla.readthedocs.io/en/0.9.13/adv_rendering_options/.

[52] Scalability reference. Last accessed February 23rd, 2022. URL: https://docs.unrealengine.com/4.27/en-US/TestingAndOptimization/PerformanceAndProfiling/Scalability/ScalabilityReference/.

[53] Synchrony and time-step. Last accessed February 23rd, 2022. URL: https://carla.readthedocs.io/en/0.9.13/adv_synchrony_timestep/.

[54] Traffic simulation in carla. Last accessed February 23rd, 2022. URL: https://carla.readthedocs.io/en/0.9.13/ts_traffic_simulation_overview/.

[55] Traffic manager. Last accessed February 23rd, 2022. URL: https://carla.readthedocs.io/en/0.9.13/adv_traffic_manager/.

[56] Ini file. Last accessed February 23rd, 2022. URL: https://docs.fileformat.com/system/ini/.

[57] configparser - configuration file parser. Last accessed February 23rd, 2022. URL: https://docs.python.org/3/library/configparser.html.

[58] René Molenaar, Arthur van Bilsen, Robin van der Made, and Raymond de Vries. Full spectrum camera simulation for reliable virtual development and validation of adas and automated driving applications. In *2015 IEEE Intelligent Vehicles Symposium (IV)*, pages 47–52, 2015. doi:10.1109/IVS.2015.7225661.

[59] concurrent.futures — launching parallel tasks. Last accessed February 23rd, 2022. URL: https://docs.python.org/3/library/concurrent.futures.html.

[60] Raúl Mur-Artal and Juan D. Tardós. ORB-SLAM2: an open-source SLAM system for monocular, stereo and RGB-D cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262, 2017. doi:10.1109/TRO.2017.2705103.

[61] Kurt Konolige Gary Bradski Ethan Rublee, Vincent Rabaud. ORB: An efficient alternative to SIFT or SURF. *International Conference on Computer Vision*, pages 2564–2571, 2011. doi:10.1109/ICCV.2011.6126544.

[62] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. volume 3951, 07 2006. doi:10.1007/11744023_34.

[63] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. Brief: Binary robust independent elementary features. In Kostas Daniilidis, Petros Maragos, and Nikos Paragios, editors, *Computer Vision – ECCV 2010*, pages 778–792, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[64] Montiel J. M. M. Mur-Artal, Raúl and Juan D. Tardós. ORB-SLAM: a versatile and accurate monocular SLAM system. *IEEE Transactions on Robotics*, 31(5):1147–1163, 2015. doi:10.1109/TRO.2015.2463671.

[65] How single-shot detector (ssd) works? Last accessed June 23rd, 2022. URL: https://developers.arcgis.com/python/guide/how-ssd-works/.

[66] Yue Wang, Dinggang Shen, and Eam Teoh. Lane detection using spline model. *Pattern Recognition Letters*, 21:677–689, 07 2000. doi:10.1016/S0167-8655(00)00021-0.

[67] Huarong Xu, Xiaodong Wang, Hongwu Huang, Keshou Wu, and Qiu Fang. A fast and stable lane detection method based on b-spline curve. In *2009 IEEE 10th International Conference on Computer-Aided Industrial Design Conceptual Design*, pages 1036–1040, 2009. doi:10.1109/CAIDCD.2009.5375392.

# Appendix A

# Preliminary work and work-plan

This chapter will make a brief presentation of the initial work plan. The introduction of an early work plan, although not fully in depth, helps to understand the general tasks that will be done in the upcoming months. The work plan is divided in 4 stages: Setup, Implementation, Generation and Validation.

- **Setup:**
  1. Setup hardware (install GPU and prepare workstation).
  2. Explore CARLA documentation and installation procedures.
  3. Install CARLA and explore the Client-Server architecture through the Python API.
  4. Get used to CARLA functionalities with the standard example scripts.
- **Implementation:**
  1. Implement generic CARLA sensor (camera/radar) in virtual environment with Python API.
  2. Prepare procedural control for controlling the environment.
  3. Expand procedural control to allow automated driving sessions.
  4. Add ground truth labelling to the environment with the previous procedural control.
  5. Develop synthetic data modelling techniques such as domain randomisation.
- **Generation:**
  1. Generate large-scale synthetic datasets with full automated environmental labelling.
  2. Apply synthetic sensor data to Continental software-in-the-loop testing framework.
- **Validation:**
  1. Compare environmental labelling to existing algorithmic detection.
  2. Validate the use of synthetic data in comparison to real data using driving function KPIs in both.

The Figure A.1 shows a preliminary work plan and its respective calendar. The only observation important to refer is that the setup and familiarisation with CARLA has a time-frame of two weeks in order to not consume much of the available time for the remaining tasks which are very important for the validation of this project.

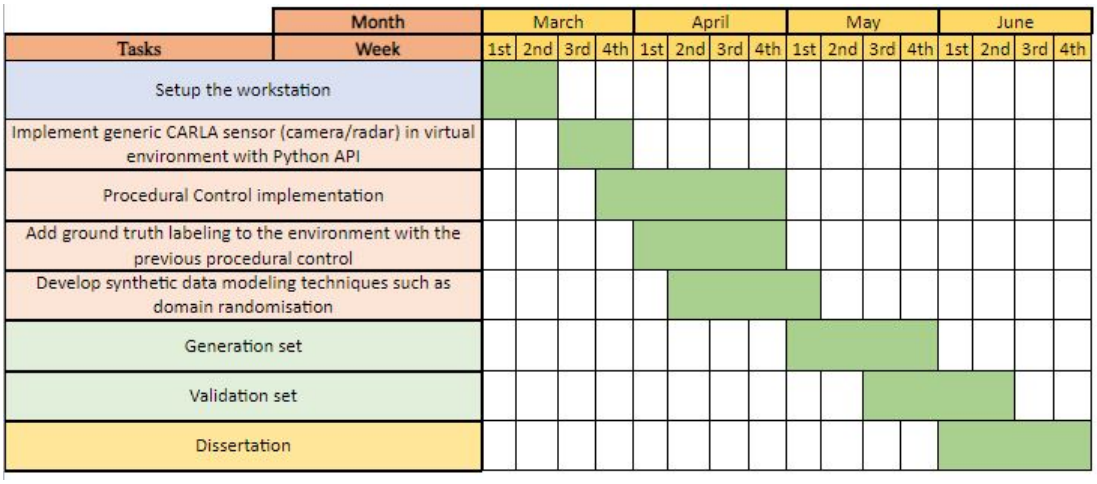



Figure A.1: Gantt diagram.

# Appendix B

# ORBSLAM2 main script

```
import glob
import sys
import carla
import logging
from numpy import random
from queue import Queue
from configparser import ConfigParser
from concurrent.futures import ThreadPoolExecutor
import os
import random
import time
import numpy as np
import cv2
import imageio
from tqdm import tqdm
import math
from PIL import Image


IM_WIDTH = 1277
IM_HEIGHT = 370

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

images = []
video_writer = imageio.get_writer('my_video.mp4', format='FFMPEG', mode='I', fps
    =10)

class Ego:
```

```
    def __init__(self, x, y, z, yaw, pitch, roll):
        self.x = x
        self.y = y
        self.z = z
        self.yaw = yaw
        self.roll = roll
        self.pitch = pitch


def write_images_to_video(images, video_writer):
    print("Writing images to video file...")
    for img in tqdm(images):
        video_writer.append_data(img)
    video_writer.close()


class Weather(object):
    def __init__(self, weather):
        self.weather = weather
        self.weather.sun_altitude_angle = 0
        self.weather.sun_azimuth_angle = 0
        self.weather.cloudiness = 0
        self.weather.precipitation = 0
        self.weather.precipitation_deposits = 0
        self.weather.wind_intensity = 0
        self.weather.wetness = 0
        self.weather.sun_azimuth_angle = random.randint(0, 360)
        self.weather.sun_altitude_angle = random.randint(-90, 90)
        self.weather.cloudiness = random.randint(0, 100)
        if 50 <= self.weather.cloudiness <= 100:
            self.weather.precipitation = random.randint(50, 100)
            self.weather.wetness = self.weather.precipitation
            self.weather.precipitation_deposits = self.weather.precipitation
            self.weather.wind_intensity = random.randint(0, 100)
        else:
            if 25 <= self.weather.cloudiness <= 50:
                self.weather.precipitation_deposits = random.randint(0, 100)
        self.weather.fog_density = random.randint(0, 50)


def get_actor_blueprints(world, filter, generation):
    bps = world.get_blueprint_library().filter(filter)

    if generation.lower() == "all":
        return bps

    # If the filter returns only one bp, we assume that this one needed
    # and therefore, we ignore the generation
```

```
84      if len(bps) == 1:
85          return bps
86
87      try:
88          int_generation = int(generation)
89          # Check if generation is in available generations
90          if int_generation in [1, 2]:
91              bps = [x for x in bps if int(x.get_attribute('generation')) ==
    int_generation]
92              return bps
93          else:
94              print("   Warning! Actor Generation is not valid. No actor will be
    spawned.")
95              return []
96      except:
97          print("   Warning! Actor Generation is not valid. No actor will be spawned.
    ")
98          return []
99
100
101 def sensor_callback(sensor_data, sensor_queue, timer, all_vehicle_actors):
102     ego_location = all_vehicle_actors[0].get_transform()
103     sensor_queue.put((sensor_data, timer, ego_location))
104
105
106 def saving(s, x):
107     i = np.array(s[0].raw_data, dtype='uint8')
108     i2 = i.reshape((IM_HEIGHT, IM_WIDTH, 4))
109     i3 = i2[:, :, :3]
110     im_rgb = cv2.cvtColor(i3, cv2.COLOR_BGR2GRAY)
111     Image.fromarray(im_rgb).save('output/%06d.png' % x)
112     images.append(im_rgb)
113
114
115 def main():
116     os.mkdir(r"G:\Users\Mimi\Desktop\trabalho\carla_vts\PythonAPI\my_workplace\
    output")
117     parser = ConfigParser()
118     parser.read('config.ini')
119     number_of_vehicles = parser.getint('vehiclesettings', 'number_of_vehicles')
120     number_of_walkers = parser.getint('walkersettings', 'number_of_walkers')
121     seed = 20
122
123     logging.basicConfig(format='%(levelname)s: %(message)s', level=logging.INFO)
124
125     vehicles_list = []
126     walkers_list = []
127     all_id = []
```

```
    client = carla.Client(parser.get('worldsettings', 'host'), parser.getint('
worldsettings', 'port'))
    client.set_timeout(120.0)
    synchronous_master = False
    random.seed(seed if seed is not None else int(time.time()))

    try:
        map_choice = random.randint(1, 6)
        if map_choice == 6:
             world = client.load_world('Town10HD')
         else:
             world = client.load_world('Town0%d' % map_choice)

        traffic_manager = client.get_trafficmanager(parser.getint('worldsettings',
'tm_port'))
        traffic_manager.set_global_distance_to_leading_vehicle(2.5)
        if parser.getboolean('worldsettings', 'respawn'):
            traffic_manager.set_respawn_dormant_vehicles(True)
        if parser.getboolean('worldsettings', 'hybrid'):
            traffic_manager.set_hybrid_physics_mode(True)
            traffic_manager.set_hybrid_physics_radius(70.0)
        if seed is not None:
            traffic_manager.set_random_device_seed(seed)

        settings = world.get_settings()
        if not parser.getboolean('worldsettings', 'asynch'):
            traffic_manager.set_synchronous_mode(True)
            if not settings.synchronous_mode:
                synchronous_master = True
                settings.synchronous_mode = True
                settings.fixed_delta_seconds = 1 / 60
                settings.max_substeps = 2

            else:
                synchronous_master = False
        else:
            print("You are currently in asynchronous mode. If this is a traffic
simulation, \
            you could experience some issues. If it's not working correctly, switch
 to synchronous \
            mode by using traffic_manager.set_synchronous_mode(True)")

        if parser.getboolean('worldsettings', 'no_rendering'):
            settings.no_rendering_mode = True
        world.apply_settings(settings)

        weather = Weather(world.get_weather())
        world.set_weather(weather.weather)

```

```
173         blueprints = get_actor_blueprints(world, parser.get('worldsettings', '
    filterv'),
174                                           parser.get('worldsettings', 'generationv'
    ))
175         blueprintsWalkers = get_actor_blueprints(world, parser.get('worldsettings',
     'filterw'),
176                                                  parser.get('worldsettings', '
    generationw'))
177 
178         if parser.getboolean('worldsettings', 'safe'):
179             blueprints = [x for x in blueprints if int(x.get_attribute('
    number_of_wheels')) == 4]
180             blueprints = [x for x in blueprints if not x.id.endswith('microlino')]
181             blueprints = [x for x in blueprints if not x.id.endswith('carlacola')]
182             blueprints = [x for x in blueprints if not x.id.endswith('cybertruck')]
183             blueprints = [x for x in blueprints if not x.id.endswith('t2')]
184             blueprints = [x for x in blueprints if not x.id.endswith('sprinter')]
185             blueprints = [x for x in blueprints if not x.id.endswith('firetruck')]
186             blueprints = [x for x in blueprints if not x.id.endswith('ambulance')]
187 
188         blueprints = sorted(blueprints, key=lambda bp: bp.id)
189 
190         spawn_points = world.get_map().get_spawn_points()
191         number_of_spawn_points = len(spawn_points)
192 
193         if number_of_vehicles < number_of_spawn_points:
194             random.shuffle(spawn_points)
195         elif number_of_vehicles > number_of_spawn_points:
196             msg = 'requested %d vehicles, but could only find %d spawn points'
197             logging.warning(msg, number_of_vehicles, number_of_spawn_points)
198             number_of_vehicles = number_of_spawn_points
199 
200         # @todo cannot import these directly.
201         SpawnActor = carla.command.SpawnActor
202         SetAutopilot = carla.command.SetAutopilot
203         FutureActor = carla.command.FutureActor
204         blueprint_library = world.get_blueprint_library()
205         # --------------
206         # Spawn vehicles
207         # --------------
208         batch = []
209         hero = parser.getboolean('worldsettings', 'hero')
210         for n, transform in enumerate(spawn_points):
211             if n >= number_of_vehicles:
212                 break
213             blueprint = random.choice(blueprints)
214             if hero:
215                 blueprint = blueprint_library.find('vehicle.tesla.model3')
216             if blueprint.has_attribute('color'):
```

```
217                 color = random.choice(blueprint.get_attribute('color').
    recommended_values)
218                 blueprint.set_attribute('color', color)
219             if blueprint.has_attribute('driver_id'):
220                 driver_id = random.choice(blueprint.get_attribute('driver_id').
    recommended_values)
221                 blueprint.set_attribute('driver_id', driver_id)
222             if hero:
223                 blueprint.set_attribute('role_name', 'hero')
224                 hero = False
225             else:
226                 blueprint.set_attribute('role_name', 'autopilot')
227 
228             # spawn the cars and set their autopilot and light state all together
229 
230             x = SpawnActor(blueprint, transform)
231             z = SetAutopilot(FutureActor, True, traffic_manager.get_port())
232             batch.append(x.then(z))
233 
234 
235         for response in client.apply_batch_sync(batch, synchronous_master):
236             if response.error:
237                 logging.error(response.error)
238             else:
239                 vehicles_list.append(response.actor_id)
240 
241         # Set automatic vehicle lights update if specified
242         all_vehicle_actors = world.get_actors(vehicles_list)
243 
244         attr = blueprint_library.find(parser.get('sensorsettings', 'bp'))
245         attr.set_attribute('enable_postprocess_effects', parser.get('sensorsettings
    ', 'enable_postprocess_effects'))
246         attr.set_attribute('fov', parser.get('sensorsettings', 'fov'))
247         attr.set_attribute('fstop', parser.get('sensorsettings', 'fstop'))
248         attr.set_attribute('image_size_x', parser.get('sensorsettings', '
    image_size_x'))
249         attr.set_attribute('image_size_y', parser.get('sensorsettings', '
    image_size_y'))
250         attr.set_attribute('sensor_tick', str(1/10))
251         image_w = float(parser.get('sensorsettings', 'image_size_x'))
252         image_h = float(parser.get('sensorsettings', 'image_size_y'))
253         fov = float(parser.get('sensorsettings', 'fov'))
254         focal = image_w / (2.0 * np.tan(fov * np.pi / 360.0))
255         print("Fx = Fy = %f || Cy =  %f|| Cx = %f||" % (focal, image_h / 2.0,
    image_w / 2.0))
256         sensor_location = carla.Location(1, 0, 1.6)
257         sensor_rotation = carla.Rotation(8.75, 0, 0)
258         sensor_transform = carla.Transform(sensor_location, sensor_rotation)
259 
```

```
260         sensor = world.spawn_actor(attr, sensor_transform, attach_to=
    all_vehicle_actors[0],
261                                    attachment_type=carla.AttachmentType.Rigid)
262 
263         # if parser.getboolean('worldsettings', 'car_lights_on'):
264         #     if 1 <= choice <= 25:
265         #         all_vehicle_actors = world.get_actors(vehicles_list)
266         #         for actor in all_vehicle_actors:
267         #             traffic_manager.update_vehicle_lights(actor, True)
268         #     if 7 <= choice <= 27:
269         #         lights = world.get_lightmanager()
270         #         street = lights.get_all_lights(carla.LightGroup.Street)
271         #         lights.turn_on(street)
272         #     if 19 <= choice <= 25:
273         #         lights = world.get_lightmanager()
274         #         building = lights.get_all_lights(carla.LightGroup.Building)
275         #         lights.turn_on(building)
276 
277         # -------------
278         # Spawn Walkers
279         # -------------
280         percentagePedestriansRunning = parser.getfloat('walkersettings', 'perc_run'
    )  # how many pedestrians will run
281         percentagePedestriansCrossing = parser.getfloat('walkersettings',
282                                                         'perc_cross')  # how many
    pedestrians will walk through the road
283         if parser.getint('worldsettings', 'seedw'):
284             world.set_pedestrians_seed(parser.getint('worldsettings', 'seedw'))
285             random.seed(parser.getint('worldsettings', 'seedw'))
286         # 1. take all the random locations to spawn
287         spawn_points = []
288         for i in range(number_of_walkers):
289             spawn_point = carla.Transform()
290             loc = world.get_random_location_from_navigation()
291             if (loc != None):
292                 spawn_point.location = loc
293                 spawn_points.append(spawn_point)
294         # 2. we spawn the walker object
295         batch = []
296         walker_speed = []
297         for spawn_point in spawn_points:
298             walker_bp = random.choice(blueprintsWalkers)
299             # set as not invincible
300             if walker_bp.has_attribute('is_invincible'):
301                 walker_bp.set_attribute('is_invincible', 'false')
302             # set the max speed
303             if walker_bp.has_attribute('speed'):
304                 if (random.random() > percentagePedestriansRunning):
305                     # walking
```

```
306                     walker_speed.append(walker_bp.get_attribute('speed').
    recommended_values[1])
307                 else:
308                     # running
309                     walker_speed.append(walker_bp.get_attribute('speed').
    recommended_values[2])
310             else:
311                 print("Walker has no speed")
312                 walker_speed.append(0.0)
313             batch.append(SpawnActor(walker_bp, spawn_point))
314         results = client.apply_batch_sync(batch, True)
315         walker_speed2 = []
316         for i in range(len(results)):
317             if results[i].error:
318                 logging.error(results[i].error)
319             else:
320                 walkers_list.append({"id": results[i].actor_id})
321                 walker_speed2.append(walker_speed[i])
322         walker_speed = walker_speed2
323         # 3. we spawn the walker controller
324         batch = []
325         walker_controller_bp = world.get_blueprint_library().find('controller.ai.
    walker')
326         for i in range(len(walkers_list)):
327             batch.append(SpawnActor(walker_controller_bp, carla.Transform(),
    walkers_list[i]["id"]))
328         results = client.apply_batch_sync(batch, True)
329         for i in range(len(results)):
330             if results[i].error:
331                 logging.error(results[i].error)
332             else:
333                 walkers_list[i]["con"] = results[i].actor_id
334         # 4. we put together the walkers and controllers id to get the objects from
     their id
335         for i in range(len(walkers_list)):
336             all_id.append(walkers_list[i]["con"])
337             all_id.append(walkers_list[i]["id"])
338         all_actors = world.get_actors(all_id)
339 
340         # wait for a tick to ensure client receives the last transform of the
    walkers we have just created
341         if parser.getboolean('worldsettings', 'asynch') or not synchronous_master:
342             world.wait_for_tick()
343         else:
344             world.tick()
345 
346         # 5. initialize each controller and set target to walk to (list is [
    controler, actor, controller, actor ...])
347         # set how many pedestrians can cross the road
```

```
348         world.set_pedestrians_cross_factor(percentagePedestriansCrossing)
349         for i in range(0, len(all_id), 2):
350             # start walker
351             all_actors[i].start()
352             # set walk to random point
353             all_actors[i].go_to_location(world.get_random_location_from_navigation
    ())
354             # max speed
355             all_actors[i].set_max_speed(float(walker_speed[int(i / 2)]))
356 
357         print('spawned %d vehicles and %d walkers, press Ctrl+C to exit.' % (len(
    vehicles_list), len(walkers_list)))
358 
359         # Example of how to use Traffic Manager parameters
360         traffic_manager.global_percentage_speed_difference(30.0)
361 
362         timer = 0
363         while True:
364             world.tick()
365             timer += 1 / 60
366             if timer > 1:
367                 break
368 
369         spectator = world.get_spectator()
370 
371         sensor_queue = Queue()
372 
373         timer = 0
374         i = 0
375 
376         executor = ThreadPoolExecutor(16)
377 
378         timestamp = []
379         location = []
380         somador = 0
381 
382         loc_init = sensor.get_transform()
383 
384         sensor.listen(lambda data: sensor_callback(data, sensor_queue, timer,
    all_vehicle_actors))
385 
386         while True:
387             if not parser.getboolean('worldsettings', 'asynch') and
    synchronous_master:
388                 world.tick()
389                 transform = sensor.get_transform()
390                 spectator.set_transform(carla.Transform(transform.location,
391                                                         transform.rotation))
392             else:
```

```
393                 world.wait_for_tick()
394
395             if somador > 340:
396                 break
397
398             if sensor_queue.qsize() > 0:
399                 s = sensor_queue.get(True, 0.01)
400                 timestamp.append(s[1])
401                 location.append(Ego(-s[2].location.x, s[2].location.y, s[2].
    location.z, s[2].rotation.yaw,
402                                     s[2].rotation.roll, s[2].rotation.pitch))
403                 f = executor.submit(saving, s, i)
404                 i = i + 1
405             loc_final = sensor.get_transform()
406             loc_dif = loc_final.location - loc_init.location
407             somador += math.sqrt(loc_dif.x * loc_dif.x + loc_dif.y * loc_dif.y +
    loc_dif.z * loc_dif.z)
408             loc_init = loc_final
409             timer += 1 / 60
410
411     finally:
412         world.tick()
413         sensor.destroy()
414
415         if not parser.getboolean('worldsettings', 'asynch') and synchronous_master:
416             settings = world.get_settings()
417             settings.synchronous_mode = False
418             settings.no_rendering_mode = False
419             settings.fixed_delta_seconds = None
420             settings.max_substep_delta_time = 0.1
421             world.apply_settings(settings)
422
423         print('\ndestroying %d vehicles' % len(vehicles_list))
424         client.apply_batch([carla.command.DestroyActor(x) for x in vehicles_list])
425
426         # stop walker controllers (list is [controller, actor, controller, actor
    ...])
427         for i in range(0, len(all_id), 2):
428             all_actors[i].stop()
429
430         print('\ndestroying %d walkers' % len(walkers_list))
431         client.apply_batch([carla.command.DestroyActor(x) for x in all_id])
432
433         time.sleep(0.5)
434
435         file = open("times.txt", "w+")
436         f = open("times.txt", "w+")
437         time1 = []
438         for x in timestamp:
```

```
            time_origin = timestamp[0]
            if x is not None:
                x = float(x - time_origin)
                scientific_notation = "{:e}".format(x)
                f.write("%s\n" % scientific_notation)
                time1.append(scientific_notation)

        f.close()

        file = open("base_coordinates.txt", "w+")
        f = open("base_coordinates.txt", "w+")
        for x in location:
            if x is not None:
                f.write("%f, %f, %f, %f, %f, %f\n" % (x.x, x.y, x.z, x.yaw, x.roll,
    x.pitch))
        f.close()

        reformed_location = []
        origin = location[0]
        i = 0
        for x in location:
            if x is not None:
                reformed_location.append(Ego(x.x - origin.x, x.y - origin.y,
                                             x.z - origin.z, x.yaw,
                                             x.roll,
                                             x.pitch))
        # <>
        newloc = []

        for x in reformed_location:
            theta = math.radians(location[0].yaw - 90)
            xx = x.x
            yy = x.y
            x1 = xx * math.cos(theta) - yy * math.sin(theta)
            y1 = xx * math.sin(theta) + yy * math.cos(theta)
            newloc.append(Ego(x1, y1,
                              x.z, x.yaw,
                              x.roll,
                              x.pitch))

        file = open("GT_coordinates.txt", "w+")
        f = open("GT_coordinates.txt", "w+")
        i = 0
        for x in newloc:
            if x is not None:
                f.write("%s, %f, %f, %f, %f, %f, %f\n" % (time1[i], x.x, x.y, x.z,
    x.yaw, x.roll, x.pitch))
            i = i + 1
        f.close()
```

```
486
487 if __name__ == '__main__':
488     try:
489         main()
490     except KeyboardInterrupt:
491         pass
492     finally:
493         print('\ndone.')
```

Listing B.1: ORBLSLAM2 code

# Appendix C

# Lane detection main script

```
import glob
import sys
import carla
import numpy
import logging
from numpy import random
from queue import Queue
from configparser import ConfigParser
from concurrent.futures import ThreadPoolExecutor
import os
import random
import time
import numpy as np
import cv2
import atexit
import imageio
from tqdm import tqdm
import math
from PIL import Image

IM_WIDTH = 1920
IM_HEIGHT = 1080

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

images = []
video_writer = imageio.get_writer('my_video.mp4', format='FFMPEG', mode='I', fps
    =10)

```

```
atexit.register(lambda: write_images_to_video(images, video_writer))

def carla_vec_to_np_array(vec):
    return np.array([vec.x,
                     vec.y,
                     vec.z])

class Ego:
    def __init__(self, x, y, z, yaw, roll, pitch):
        self.x = x
        self.y = y
        self.z = z
        self.yaw = yaw
        self.roll = roll
        self.pitch = pitch


def write_images_to_video(images, video_writer):
    print("Writing images to video file...")
    for img in tqdm(images):
        video_writer.append_data(img)
    video_writer.close()


class Weather(object):
    def __init__(self, weather):
        self.weather = weather
        self.weather.sun_altitude_angle = 0
        self.weather.sun_azimuth_angle = 0
        self.weather.cloudiness = 0
        self.weather.precipitation = 0
        self.weather.precipitation_deposits = 0
        self.weather.wind_intensity = 0
        self.weather.wetness = 0
        self.weather.sun_azimuth_angle = random.randint(0, 360)
        self.weather.sun_altitude_angle = random.randint(-90, 90)
        self.weather.cloudiness = random.randint(0, 100)
        if 50 <= self.weather.cloudiness <= 100:
            self.weather.precipitation = random.randint(50, 100)
            self.weather.wetness = self.weather.precipitation
            self.weather.precipitation_deposits = self.weather.precipitation
            self.weather.wind_intensity = random.randint(0, 100)
        else:
            if 25 <= self.weather.cloudiness <= 50:
                self.weather.precipitation_deposits = random.randint(0, 100)
        self.weather.fog_density = random.randint(0, 50)


def get_actor_blueprints(world, filter, generation):
```

```
84      bps = world.get_blueprint_library().filter(filter)
85
86      if generation.lower() == "all":
87          return bps
88
89      # If the filter returns only one bp, we assume that this one needed
90      # and therefore, we ignore the generation
91      if len(bps) == 1:
92          return bps
93
94      try:
95          int_generation = int(generation)
96          # Check if generation is in available generations
97          if int_generation in [1, 2]:
98              bps = [x for x in bps if int(x.get_attribute('generation')) ==
    int_generation]
99              return bps
100         else:
101             print("   Warning! Actor Generation is not valid. No actor will be
    spawned.")
102             return []
103     except:
104         print("   Warning! Actor Generation is not valid. No actor will be spawned.
    ")
105         return []
106
107
108 def sensor_callback(sensor_data, sensor_queue, timer, sensor, world):
109     ego_location = sensor.get_transform()
110     carla.Map
111     waypoint = world.get_map().get_waypoint(sensor.get_location(), project_to_road=
    True,
112                                             lane_type=(carla.LaneType.Driving))
113     offset = (
114             carla_vec_to_np_array(waypoint.transform.get_right_vector())
115             * waypoint.lane_width
116             / 2.0
117     )
118
119     # waypoint_left = waypoint - offset
120     # waypoint_right = waypoint + offset
121     sensor_queue.put((sensor_data, timer, ego_location, waypoint.transform.
    get_matrix(),waypoint.get_left_lane().transform.get_matrix(),waypoint.
    get_right_lane().transform.get_matrix(), offset))
122
123
124 def saving(s, x):
125     i = np.array(s[0].raw_data, dtype='uint8')
126     i2 = i.reshape((IM_HEIGHT, IM_WIDTH, 4))
```

```
    i3 = i2[:, :, :3]
    im_rgb = cv2.cvtColor(i3, cv2.COLOR_BGR2RGB)
    # dim = (1277, 370)
    # resized = cv2.resize(im_rgb, dim, interpolation=cv2.INTER_AREA)
    Image.fromarray(im_rgb).save('output/%d.png' % x)
    # cv2.imwrite('output/%06d.png' % x, im_rgb)
    images.append(im_rgb)


def main():
    # os.mkdir(r"G:\Users\Mimi\Desktop\trabalho\carla_vts\PythonAPI\my_workplace\output")
    parser = ConfigParser()
    parser.read('config.ini')
    number_of_vehicles = parser.getint('vehiclesettings', 'number_of_vehicles')
    number_of_walkers = parser.getint('walkersettings', 'number_of_walkers')
    seed = None

    logging.basicConfig(format='%(levelname)s: %(message)s', level=logging.INFO)

    vehicles_list = []
    walkers_list = []
    all_id = []
    client = carla.Client(parser.get('worldsettings', 'host'), parser.getint('worldsettings', 'port'))
    client.set_timeout(120.0)
    synchronous_master = False
    random.seed(seed if seed is not None else int(time.time()))

    try:
        map_choice = random.randint(1, 6)
        if map_choice == 6:
            world = client.load_world('Town10HD')
        else:
            world = client.load_world('Town0%d' % map_choice)

        world = client.load_world('Town04')
        traffic_manager = client.get_trafficmanager(parser.getint('worldsettings', 'tm_port'))
        traffic_manager.set_global_distance_to_leading_vehicle(2.5)
        if parser.getboolean('worldsettings', 'respawn'):
            traffic_manager.set_respawn_dormant_vehicles(True)
        if parser.getboolean('worldsettings', 'hybrid'):
            traffic_manager.set_hybrid_physics_mode(True)
            traffic_manager.set_hybrid_physics_radius(70.0)
        if seed is not None:
            traffic_manager.set_random_device_seed(seed)

        settings = world.get_settings()
```

```
        if not parser.getboolean('worldsettings', 'asynch'):
            traffic_manager.set_synchronous_mode(True)
            if not settings.synchronous_mode:
                synchronous_master = True
                settings.synchronous_mode = True
                settings.fixed_delta_seconds = 1 / 60
                settings.max_substeps = 2


            else:
                synchronous_master = False
        else:
            print("You are currently in asynchronous mode. If this is a traffic simulation, \
            you could experience some issues. If it's not working correctly, switch to synchronous \
            mode by using traffic_manager.set_synchronous_mode(True)")

        if parser.getboolean('worldsettings', 'no_rendering'):
            settings.no_rendering_mode = True
        world.apply_settings(settings)

        weather = Weather(world.get_weather())
        world.set_weather(weather.weather)

        blueprints = get_actor_blueprints(world, parser.get('worldsettings', 'filterv'),
                                          parser.get('worldsettings', 'generationv'))
        blueprintsWalkers = get_actor_blueprints(world, parser.get('worldsettings', 'filterw'),
                                                 parser.get('worldsettings', 'generationw'))

        if parser.getboolean('worldsettings', 'safe'):
            blueprints = [x for x in blueprints if int(x.get_attribute('number_of_wheels')) == 4]
            blueprints = [x for x in blueprints if not x.id.endswith('microlino')]
            blueprints = [x for x in blueprints if not x.id.endswith('carlacola')]
            blueprints = [x for x in blueprints if not x.id.endswith('cybertruck')]
            blueprints = [x for x in blueprints if not x.id.endswith('t2')]
            blueprints = [x for x in blueprints if not x.id.endswith('sprinter')]
            blueprints = [x for x in blueprints if not x.id.endswith('firetruck')]
            blueprints = [x for x in blueprints if not x.id.endswith('ambulance')]

        blueprints = sorted(blueprints, key=lambda bp: bp.id)

        spawn_points = world.get_map().get_spawn_points()

```

```
        # @todo cannot import these directly.
        SpawnActor = carla.command.SpawnActor
        SetAutopilot = carla.command.SetAutopilot
        FutureActor = carla.command.FutureActor
        blueprint_library = world.get_blueprint_library()
        # --------------
        # Spawn vehicles
        # --------------
        batch = []
        hero = parser.getboolean('worldsettings', 'hero')
        for n, transform in enumerate(spawn_points):
            if n >= number_of_vehicles:
                break
            blueprint = random.choice(blueprints)
            if hero:
                blueprint = blueprint_library.find('vehicle.tesla.model3')
            if blueprint.has_attribute('color'):
                color = random.choice(blueprint.get_attribute('color').
    recommended_values)
                blueprint.set_attribute('color', color)
            if blueprint.has_attribute('driver_id'):
                driver_id = random.choice(blueprint.get_attribute('driver_id').
    recommended_values)
                blueprint.set_attribute('driver_id', driver_id)
            if hero:
                blueprint.set_attribute('role_name', 'hero')
                hero = False
            else:
                blueprint.set_attribute('role_name', 'autopilot')

            # spawn the cars and set their autopilot and light state all together

            x = SpawnActor(blueprint, spawn_points[76])
            z = SetAutopilot(FutureActor, True, traffic_manager.get_port())
            batch.append(x.then(z))

        for response in client.apply_batch_sync(batch, synchronous_master):
            if response.error:
                logging.error(response.error)
            else:
                vehicles_list.append(response.actor_id)

        # Set automatic vehicle lights update if specified
        all_vehicle_actors = world.get_actors(vehicles_list)

        attr = blueprint_library.find(parser.get('sensorsettings', 'bp'))
        attr.set_attribute('enable_postprocess_effects', parser.get('sensorsettings
    ', 'enable_postprocess_effects'))
        attr.set_attribute('fov', parser.get('sensorsettings', 'fov'))
```

```
261         attr.set_attribute('fstop', parser.get('sensorsettings', 'fstop'))
262         attr.set_attribute('image_size_x', parser.get('sensorsettings', '
    image_size_x'))
263         attr.set_attribute('image_size_y', parser.get('sensorsettings', '
    image_size_y'))
264         attr.set_attribute('sensor_tick', str(1 / 10))
265         image_w = float(parser.get('sensorsettings', 'image_size_x'))
266         image_h = float(parser.get('sensorsettings', 'image_size_y'))
267         fov = float(parser.get('sensorsettings', 'fov'))
268         focal = image_w / (2.0 * np.tan(fov * np.pi / 360.0))
269         print("Fx = Fy = %f || Cy =  %f|| Cx = %f||" % (focal, image_h / 2.0,
    image_w / 2.0))
270         sensor_location = carla.Location(1, 0, 1.2)
271         sensor_rotation = carla.Rotation(8.75, 0, 0)
272         sensor_transform = carla.Transform(sensor_location, sensor_rotation)
273 
274         sensor = world.spawn_actor(attr, sensor_transform, attach_to=
    all_vehicle_actors[0],
275                                    attachment_type=carla.AttachmentType.Rigid)
276 
277         # if parser.getboolean('worldsettings', 'car_lights_on'):
278         #     if 1 <= choice <= 25:
279         #         all_vehicle_actors = world.get_actors(vehicles_list)
280         #         for actor in all_vehicle_actors:
281         #             traffic_manager.update_vehicle_lights(actor, True)
282         #     if 7 <= choice <= 27:
283         #         lights = world.get_lightmanager()
284         #         street = lights.get_all_lights(carla.LightGroup.Street)
285         #         lights.turn_on(street)
286         #     if 19 <= choice <= 25:
287         #         lights = world.get_lightmanager()
288         #         building = lights.get_all_lights(carla.LightGroup.Building)
289         #         lights.turn_on(building)
290 
291         # -------------
292         # Spawn Walkers
293         # -------------
294         percentagePedestriansRunning = parser.getfloat('walkersettings', 'perc_run'
    )  # how many pedestrians will run
295         percentagePedestriansCrossing = parser.getfloat('walkersettings',
296                                                         'perc_cross')  # how many
    pedestrians will walk through the road
297         if parser.getint('worldsettings', 'seedw'):
298             world.set_pedestrians_seed(parser.getint('worldsettings', 'seedw'))
299             random.seed(parser.getint('worldsettings', 'seedw'))
300         # 1. take all the random locations to spawn
301         spawn_points = []
302         for i in range(number_of_walkers):
303             spawn_point = carla.Transform()
```

```
304             loc = world.get_random_location_from_navigation()
305             if (loc != None):
306                 spawn_point.location = loc
307                 spawn_points.append(spawn_point)
308         # 2. we spawn the walker object
309         batch = []
310         walker_speed = []
311         for spawn_point in spawn_points:
312             walker_bp = random.choice(blueprintsWalkers)
313             # set as not invincible
314             if walker_bp.has_attribute('is_invincible'):
315                 walker_bp.set_attribute('is_invincible', 'false')
316             # set the max speed
317             if walker_bp.has_attribute('speed'):
318                 if (random.random() > percentagePedestriansRunning):
319                     # walking
320                     walker_speed.append(walker_bp.get_attribute('speed').
    recommended_values[1])
321                 else:
322                     # running
323                     walker_speed.append(walker_bp.get_attribute('speed').
    recommended_values[2])
324             else:
325                 print("Walker has no speed")
326                 walker_speed.append(0.0)
327             batch.append(SpawnActor(walker_bp, spawn_point))
328         results = client.apply_batch_sync(batch, True)
329         walker_speed2 = []
330         for i in range(len(results)):
331             if results[i].error:
332                 logging.error(results[i].error)
333             else:
334                 walkers_list.append({"id": results[i].actor_id})
335                 walker_speed2.append(walker_speed[i])
336         walker_speed = walker_speed2
337         # 3. we spawn the walker controller
338         batch = []
339         walker_controller_bp = world.get_blueprint_library().find('controller.ai.
    walker')
340         for i in range(len(walkers_list)):
341             batch.append(SpawnActor(walker_controller_bp, carla.Transform(),
    walkers_list[i]["id"]))
342         results = client.apply_batch_sync(batch, True)
343         for i in range(len(results)):
344             if results[i].error:
345                 logging.error(results[i].error)
346             else:
347                 walkers_list[i]["con"] = results[i].actor_id
348
```

```
349         # 4. we put together the walkers and controllers id to get the objects from
     their id
350         for i in range(len(walkers_list)):
351             all_id.append(walkers_list[i]["con"])
352             all_id.append(walkers_list[i]["id"])
353         all_actors = world.get_actors(all_id)
354 
355 
356 
357         # wait for a tick to ensure client receives the last transform of the
    walkers we have just created
358         if parser.getboolean('worldsettings', 'asynch') or not synchronous_master:
359             world.wait_for_tick()
360         else:
361             world.tick()
362 
363         # 5. initialize each controller and set target to walk to (list is [
    controler, actor, controller, actor ...])
364         # set how many pedestrians can cross the road
365         world.set_pedestrians_cross_factor(percentagePedestriansCrossing)
366         for i in range(0, len(all_id), 2):
367             # start walker
368             all_actors[i].start()
369             # set walk to random point
370             all_actors[i].go_to_location(world.get_random_location_from_navigation
    ())
371             # max speed
372             all_actors[i].set_max_speed(float(walker_speed[int(i / 2)]))
373 
374         print('spawned %d vehicles and %d walkers, press Ctrl+C to exit.' % (len(
    vehicles_list), len(walkers_list)))
375 
376         # Example of how to use Traffic Manager parameters
377         traffic_manager.global_percentage_speed_difference(30.0)
378 
379         timer = 0
380         while True:
381             world.tick()
382             timer += 1 / 60
383             if timer > 1:
384                 break
385 
386         spectator = world.get_spectator()
387 
388         sensor_queue = Queue()
389 
390         timer = 0
391         i = 0
392 
```

```
393         executor = ThreadPoolExecutor(16)
394 
395         timestamp = []
396         location = []
397         somador = 0
398         loc_init1 = sensor.get_transform()
399         loc_init = sensor.get_transform()
400         wayloc = []
401         wayloc_left = []
402         off = []
403         wayloc_right = []
404         sensor.listen(lambda data: sensor_callback(data, sensor_queue, timer,
    sensor, world))
405 
406         while True:
407             if not parser.getboolean('worldsettings', 'asynch') and
    synchronous_master:
408                 world.tick()
409                 transform = sensor.get_transform()
410                 spectator.set_transform(carla.Transform(transform.location,
411                                                         transform.rotation))
412 
413 
414             else:
415                 world.wait_for_tick()
416 
417             if timer > 15:
418                 break
419 
420             if sensor_queue.qsize() > 0 :
421                 s = sensor_queue.get(True, 0.01)
422                 timestamp.append(s[1])
423                 # if s[2].rotation.yaw < 0:
424                 #     s[2].rotation.yaw = s[2].rotation.yaw + 360
425                 location.append(s[2].get_inverse_matrix())
426                 wayloc.append(s[3])
427                 wayloc_left.append(s[4])
428                 wayloc_right.append(s[5])
429                 off.append(s[6])
430                 if timer < 10:
431                     f = executor.submit(saving, s, i)
432                     i = i + 1
433             loc_final = sensor.get_transform()
434             loc_dif = loc_final.location - loc_init.location
435             somador += math.sqrt(loc_dif.x * loc_dif.x + loc_dif.y * loc_dif.y +
    loc_dif.z * loc_dif.z)
436             loc_init = loc_final
437             timer += 1 / 60
438 
```

```
    finally:
        print(all_vehicle_actors[0].get_transform())
        world.tick()
        sensor.destroy()

        if not parser.getboolean('worldsettings', 'asynch') and synchronous_master:
            settings = world.get_settings()
            settings.synchronous_mode = False
            settings.no_rendering_mode = False
            settings.fixed_delta_seconds = None
            settings.max_substep_delta_time = 0.1
            world.apply_settings(settings)

        print('\ndestroying %d vehicles' % len(vehicles_list))
        client.apply_batch([carla.command.DestroyActor(x) for x in vehicles_list])

        # stop walker controllers (list is [controller, actor, controller, actor
    ...])
        for i in range(0, len(all_id), 2):
            all_actors[i].stop()

        print('\ndestroying %d walkers' % len(walkers_list))
        client.apply_batch([carla.command.DestroyActor(x) for x in all_id])

        time.sleep(0.5)

        f = open("times.txt", "w+")
        time1 = []
        for x in timestamp:
            time_origin = timestamp[0]
            if x is not None:
                x = float(x - time_origin)
                scientific_notation = "{:e}".format(x)
                f.write("%s\n" % scientific_notation)
                time1.append(scientific_notation)

        f.close()

        K = numpy.zeros((3,3))
        K[0][0] = image_w / (2.0 * np.tan(fov * np.pi / 360.0))
        K[1][1] = image_w / (2.0 * np.tan(fov * np.pi / 360.0))
        K[0][2] = image_w / 2.0
        K[1][2] = image_h / 2.0
        K[2][2] = 1

        img_folder_path = 'G:/Users/Mimi/Desktop/trabalho/carla_vts/PythonAPI/
    my_workplace/output'
        dirListing = os.listdir(img_folder_path)
        size = len(dirListing)
```

```
486
487         f1 = open("left_lane.txt", "w+")
488         f2 = open("right_lane.txt", "w+")
489
490         for j in range(size):
491             image = cv2.imread("./output/%d.PNG" %j)
492             f1.write("Frame %d\n" % j)
493             f2.write("Frame %d\n" % j)
494
495             for i in wayloc_left:
496                 world_point = numpy.zeros((4,1))
497                 world_point[0][0] = i[0][3] + off[1][0]
498                 world_point[1][0] = i[1][3] + off[1][1]
499                 world_point[2][0] = i[2][3] + off[1][2]
500                 world_point[3][0] = 1
501                 world_2_camera = location[j]
502                 sensor_points = np.dot(world_2_camera, world_point)
503                 point_in_camera_coords = np.array([
504                     sensor_points[1],
505                     sensor_points[2] * -1,
506                     sensor_points[0]])
507                 points_2d = np.dot(K, point_in_camera_coords)
508                 u_coord, v_coord, _ = tuple(map(int, points_2d/points_2d[2][0]))
509                 if v_coord > 650:
510                     f1.write("%f, %f\n" % (u_coord,v_coord))
511                     image = cv2.circle(image, (u_coord, v_coord), radius=5, color
    =(0, 0, 255), thickness=-1)
512
513             for i in wayloc_right:
514
515                 world_point = numpy.zeros((4,1))
516
517                 world_point[0][0] = i[0][3] - off[1][0]
518                 world_point[1][0] = i[1][3] - off[1][1]
519                 world_point[2][0] = i[2][3] - off[1][2]
520                 world_point[3][0] = 1
521                 world_2_camera = location[j]
522                 sensor_points = np.dot(world_2_camera, world_point)
523                 point_in_camera_coords = np.array([
524                     sensor_points[1],
525                     sensor_points[2] * -1,
526                     sensor_points[0]])
527                 points_2d = np.dot(K, point_in_camera_coords)
528                 u_coord, v_coord, _ = tuple(map(int, points_2d/points_2d[2][0]))
529                 if v_coord>650:
530                     f2.write("%f, %f\n" % (u_coord, v_coord))
531                     image = cv2.circle(image, (u_coord, v_coord), radius=5, color
    =(0, 0, 255), thickness=-1)
532
```

```
533
534             # cv2.imshow('Frame', image)
535             # cv2.waitKey()
536
537         f1.close()
538         f2.close()
539
540
541 if __name__ == '__main__':
542     try:
543         main()
544     except KeyboardInterrupt:
545         pass
546     finally:
547         print('\ndone.')
```

Listing C.1: Lane detection main code