Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2019

Pedro Marques Ferreira da Silva

Estudo e Adaptação do Simulador de Condução Autónoma CARLA para o ATLASCAR2
Study and Adaptation of the Autonomous Driving Simulator CARLA for the ATLASCAR2

Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2019

Pedro Marques Ferreira da Silva

Estudo e Adaptação do simulador de condução autónoma CARLA para o ATLASCAR2
Study and Adaptation of the Autonomous Driving Simulator CARLA for the ATLASCAR2

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica de Paulo Miguel de Jesus Dias, Professor Auxiliar do Departamento Electrónica, Telecomunicações e Informática da Universidade de Aveiro e de Vítor Manuel Ferreira dos Santos, Professor Associado do Departamento de Engenharia Mecânica da Universidade de Aveiro.

o júri / the jury

presidente / president	**Prof. Doutor José Nuno Panelas Nunes Lau.** Professor Auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro.
vogais / examiners committee	**Doutor João Manuel Leite da Silva.** Investigador Sénior da empresa Altran Portugal (arguente).
	Prof. Doutor Paulo Miguel de Jesus Dias. Professor Auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro (orientador).

agradecimentos / acknowledgements

Em primeiro lugar quero agradecer à minha familia em especial aos meus pais pela educação que me deram e por me terem acompanhado neste trajecto académico. Quero agradecer aos professores Paulo Dias e Vitor Santos pela sua orientação e por me terem guiado neste projecto e também quero deixar um agradecimento ao professor Miguel Riem Oliveira por estar sempre disposto a ajudar em qualquer altura.
Por fim, quero agradecer aos meus amigos e aos meus colegas do DETI que me apoiaram estes anos todos e me fizeram evoluir não só como trabalhador e agora como engenheiro mas sobretudo como pessoa.

Palavras-Chave Condução autónoma; Veículos autónomos; Simuladores de condução autónoma; ATLASCAR; CARLA; AD; ADAS; LIDAR; Detecção de objetos; Processamento de Imagem.

Resumo

No âmbito do projecto do ATLASCAR2, esta dissertação baseia-se no estudo e integração do simulador já existente para assistência à condução autónoma CARLA, que implementa uma interface baseada em ROS para replicar o setup do ATLASCAR2 dentro da simulação. A ideia do uso de um simulador de condução autónoma como CARLA foi proposta a fim de simplificar a tarefa de aquisição de dados para o ATLASCAR2 visto que esta é uma tarefa que se vai tornando cada vez mais díficil devido a factores como a complexidade ao nível do setup e a calibração dos sensores instalados no ATLASCAR2, assim como outros factores como a interface de hardware e o tempo que é necessário para fazer uma aquisição de dados usando o ATLASCAR2. Esta ferramenta consegue produzir cenários próximos da realidade e pode ser usada para testar os algoritmos que vão ser usados no ATLASCAR2 em ambientes controlados oferencendo um método de validação que pode ser usado para avaliar o desempenho destes algoritmos nesses ambientes antes de os testar na plataforma real. O processo de replicação do setup do ATLASCAR2 e os algoritmos envolvidos no CARLA serão descritos com mais detalhe durante a dissertação que inclui secções onde é feita a descrição do processo de replicação do setup e dos algoritmos, a demonstração de resultados produzidos através da implementação do setup do ATLASCAR2 no CARLA, assim como alguns resultados produzidos de algumas experiências com dados simulados pelo CARLA que inclui a utilização de algoritmos de visão e outros algoritmos que estão a ser utilizados no ATLASCAR2.

Keywords Autonomous Driving; Autonomous Vehicles; Autonomous Driving Simulators; ATLASCAR; CARLA; AD; ADAS; LIDAR; Object Detection; Image Processing.

Abstract

Within the scope of the ATLASCAR2 project, this dissertation is based on studying and integrating the already existing autonomous driving assistance simulator named CARLA that implements an interface based on ROS to replicate the ATLASCAR2 setup in the simulation. The idea of using an autonomous driving simulator was proposed as a way to simplify the data aquisition process for the ATLASCAR2 since this process keeps on getting more and more difficult due to factors such as the complexity in the setup and the calibration processes of the installed sensors on the ATLASCAR2, as well as other factors such as the hardware interface and the time that is required to perform a single data aquisition using the ATLASCAR2. This tool can produce realistic scenarios and can be used for testing out the algorithms that are going to be implemented in the ATLASCAR2 in controlled environments, offering a degree of ground truth for these algorithms that can be used to evaluate the performance in these environments before implementing them in the real platform. The replication of the ATLASCAR2 setup process as well as the algorithms involved in CARLA will be discussed in further detail during this dissertation which include sections talking about the replication process and the algorithms involved, showing the results of the ATLASCAR2 setup implementation in CARLA as well as some other results produced from experiments with CARLA simulated data which include the use of computer vision algorithms as well as other algorithms that are currently being used in the ATLASCAR2.

Contents

List of Figures

List of Tables

Listings

Glossary

AD	Autonomous Driving. 1
ADAS	Advanced Driving Assistance System. 1
AI	Artificial Intelligence. 15
ATC	Advanced Technologies Center. 11
CARLA	Car Learning How to Act. 3
DARPA	Defense Advanced Research Projects Agency. 9
FOV	Field of View. 34
GPU	Graphical Processor Unit. 4
GUI	Graphical User Interface. 27
HMI	Human Machine Interaction. 20
KITTI	Karlsruhe Institute of Technology and Toyota Technological Institute. 64
LAR	Laboratory of Automation and Robotics. 1
LIDAR	Light Detection and Ranging. 2
OpenCV	Open Source Computer Vision Library. 7
PCL	Point Cloud Library. 7
RCA	Radio Corporation of America. 9
ROS	Robot Operating System. 7
SCOT	Shared Computer-Operated Transit. 11
SMART	Singapore-MIT Alliance for Research and Technology. 11
UE4	Unreal Engine 4. 30

Chapter 1

Introduction

In the fields of Advanced Driving Assistance Systems (ADASs) and Autonomous Driving (AD), there have been some technological studies that have kept on growing these past few decades in both the automobile industry and the academic environment.
It is also important to note that in AD and ADAS the use of autonomous driving simulators is becoming more and more significant as they prove to be important tools for testing and evaluating the algorithms that are going to be implemented in the autonomous vehicle before implementing these algorithms in the real vehicle. These simulators can produce realistic scenarios that are often used for data acquisition in datasets that in conjunction with data labelling can be used as input for learning algorithms and work on their results.
The purpose of this dissertation is focused on the adaptation of an autonomous driving simulator named CARLA in the context of the ATLASCAR2 project and test the results of the simulation. These results can later on be used for research of methods to register data using the cameras and the LIDAR sensors of the ATLASCAR2 installed in the simulated vehicle. This registered data is important for the vehicle to know its surroundings and later on create models of the objects present in CARLA and therefore computer vision algorithms used for object detection and visual perception must be tested with data registered from CARLA in order to create these models. For this reason, a tool for testing out these algorithms using frames gathered from the CARLA simulation will be developed.
This dissertation will also be used to evaluate algorithms that are currently being implemented in the ATLASCAR2, evaluating the performance of these algorithms in the scenarios provided by the CARLA simulator. This tool can also be used to test the developed algorithms before implementing them in the ATLASCAR2 offering then the possibility to alternate between simulated scenarios that can be controlled and real world scenarios.

1.1 ATLASCAR Project

The ATLASCAR is one of the projects developed from the ATLAS project, which is a project developed by the Laboratory of Automation and Robotics (LAR) at the Department of Mechanical Engineering of the University of Aveiro, Portugal. The focus of the ATLAS project was to develop and enable the proliferation of advanced sensing and active systems designed for implementation in automobiles and affine platforms. These advanced active systems keep on being improved, or newly developed, using the data acquired from vision, laser and other sensors.

The ATLAS team has a vast experience with autonomous navigation in controlled environments and this is especially shown in projects such as the ATLASCAR1 which was a full sized prototype equipped with several state of the art sensors(LARlabs 2019[1]).
Currently, the ATLAS project is now evolving to deal with real road scenarios with the ATLASCAR2 project being the main model which is the new full sized prototype being used for research purposes equipped with Light Detection and Ranging (LIDAR) sensors and a camera set. The ATLAS project was created in 2003 and it started with robot prototypes (figure 1.1) that were later developed to participate at AD competitions taking place at the Portuguese National Robotics Festival. From this project, three small-sized platform robots were built, which were very successfull having won some prizes in other robotics competitions.
As time goes on, the project grew evolving into full-sized prototypes: the ATLASCARs, ATLASCAR1 (figure 1.2) being the first full-sized platform and it was based on a Ford Escort Station Wagon, and the ATLASCAR2 (figure 1.3) which is a full-sized platform based on a Mitsubishi i-MiEV.
The ATLASCAR1 was equipped with several LIDAR sensors as well as several cameras. The data acquired about its environment was gathered by the scanners present in the car which would then process building perception into the car allowing it to move and perform tasks autonomously, like moving and executing maneuvers in small and controlled places. In the end, the ATLASCAR1 brought some very interesting and succesfull results which were then adapted to the new full-sized platform of the ATLAS project which is the ATLASCAR2 and this was vehicle model used for research in this dissertation.
The ATLASCAR2 is equipped with various state of the art LIDAR sensors and a Point Grey Camera, and it is also a full electric vehicle, unlike the ATLASCAR1, which means it is easier to modify, test and control unlike the previous version.

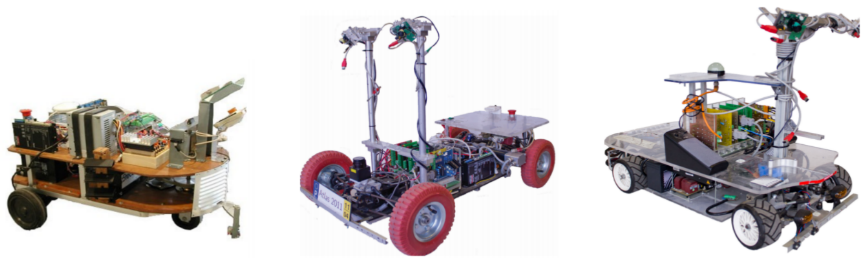


Figure 1.1: ATLAS project prototypes.[2]

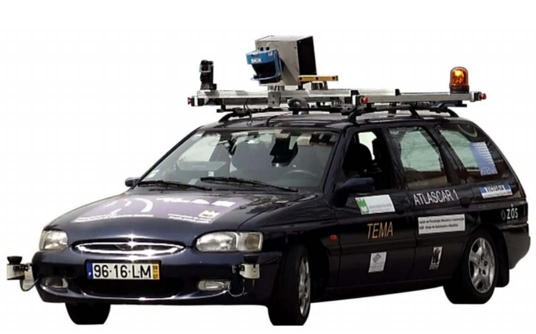


Figure 1.2: ATLASCAR1 model based on the Ford Escort platform.[3]

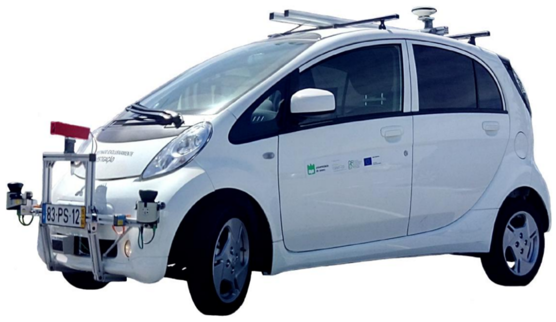


Figure 1.3: ATLASCAR2 model based on the Mitsubishi i-MiEV platform.[4]

1.2 CARLA Project

The Car Learning How to Act (CARLA)[5] is an open-source simulator used for autonomous driving research, developed from the ground up to support development, training, prototyping and validation of autonomous driving models in urban driving systems and these driving models include both perception and control models. This platform was designed by a development team in the Computer Vision Center in Barcelona. In addition to the open-source code and protocols, CARLA provides open digital assets (urban layouts, buildings, vehicles, pedestrians, street signs, etc. . .) that were created for this purpose and can be used freely.
Uniquely, the content of urban environments provided with CARLA is also free. The content was created from scratch by a dedicated team of digital artists employed for this purpose.
The simulation platform supports flexible setup of sensor suites and provides signals that can be used to train driving strategies, such as GPS coordinates, speed, acceleration and detailed data on collisions and other infractions. A wide range of environmental conditions can be specified, including weather and time of day. A number of these environmental conditions are illustrated in Figure 1.4.

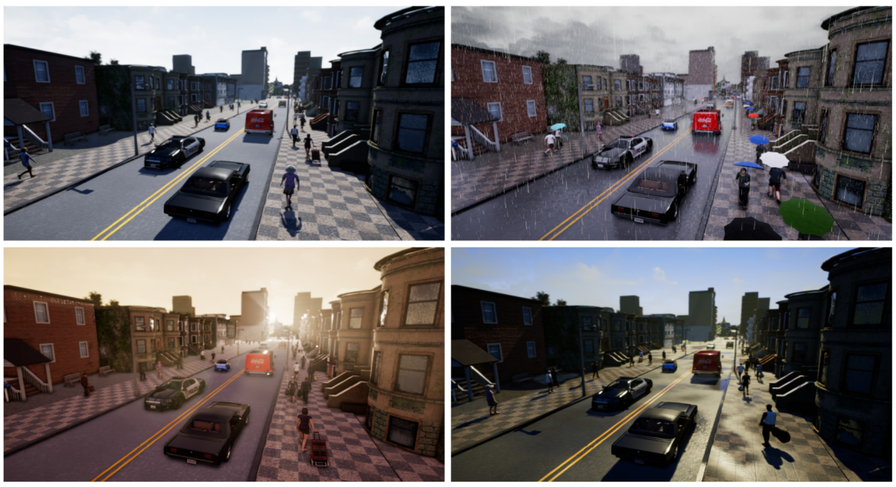


Figure 1.4: Carla Environment.

This simulation platform consist of two main modules, the CARLA simulator and the CARLA Python API module (as described in figure 1.5), in which the simulator does most of the heavy work, controls the logic, physics and rendering of all the actors and sensors present in the scene.
This simulator requires a powerful machine with a dedicated Graphical Processor Unit (GPU) in order to run.
The CARLA Python API is a module that can be used to import Python scripts and it is used to provide an interface for controlling the simulator and retrieve data from this simulator. With the Python API, it is possible to control any vehicle in the simulation, attach sensors to it and read back the data that these sensors can generate. Most of the aspects of the simulation can be controlled via the Python API, and the objective of this module is to have more and more aspects available via this module in future releases.

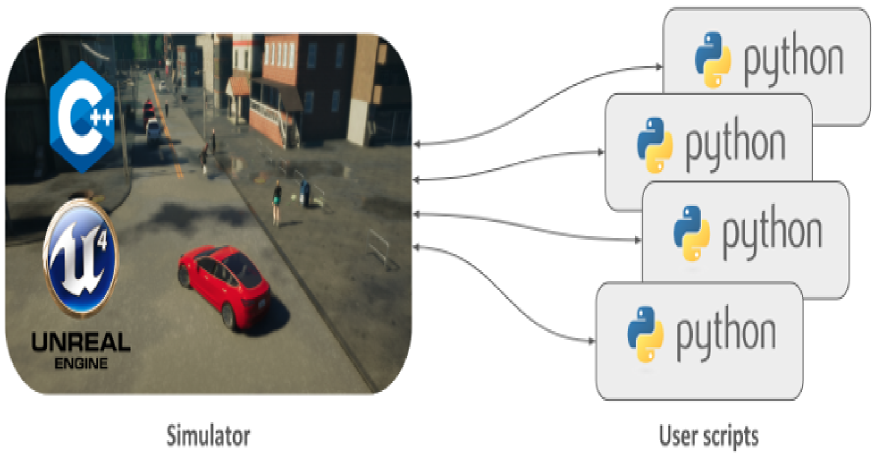


Figure 1.5: Carla Modules.

The Python API is the one responsible for establishing connection and future interaction between the autonomous agent and the server present in the simulator running the simulation via sockets. The client can use this API to send commands, or meta-commands to the server in order to receive sensor messages in return. The command messages are used to control the vehicle to perform a certain action such as steering, accelerating and braking and the meta-command messages are used to control the environmental conditions of the simulation such as weather conditions, illumination and the density of cars and pedestrians present in the simulation. When the server present in the simulator is reset, the agent can be re-initialized and the vehicle will be spawned in a new location specified by the Python API.

The environment used by the CARLA simulator incorporates a set of 3D models of static objects such as buildings, vegetation, traffic signs, infrastructure, as well as other dynamic objects such as vehicles and pedestrians. All of these models were rigorously designed in order to reconcile factors such as visual quality and rendering speed, using low-weight geometric models and textures to maintain a degree of visual realism by carefully crafting the materials and making use of a variable level of detail. The sizes of these 3D models were made in a common scale in order to reflect those of real objects.

The sensors implemented in CARLA allow for flexible configuration of the agent's sensor suite. At the beginning of this dissertation, the sensors were limited to RGB cameras and pseudo-sensors that provide some level of ground-truth depth and semantic segmentation. The camera sensors are illustrated in figure 1.6 and the LIDAR range-based sensors are illustrated in figure 1.7.

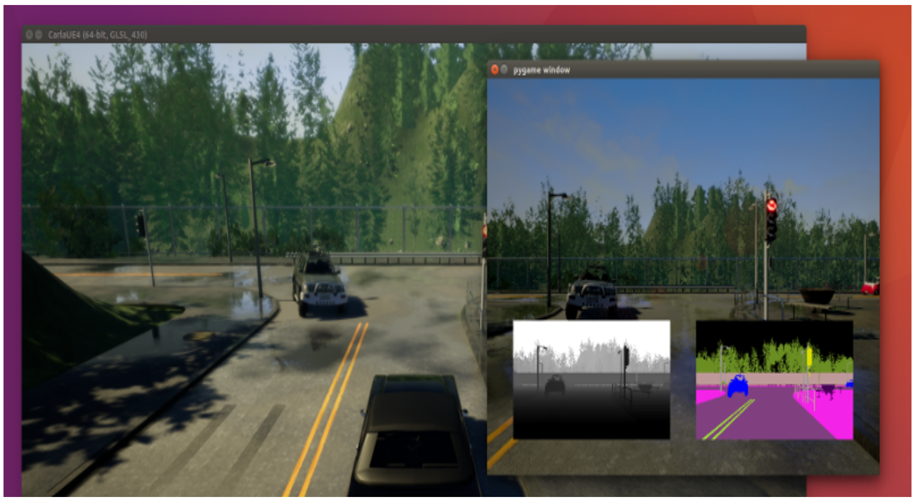


Figure 1.6: Carla Sensor Modalities.[6]

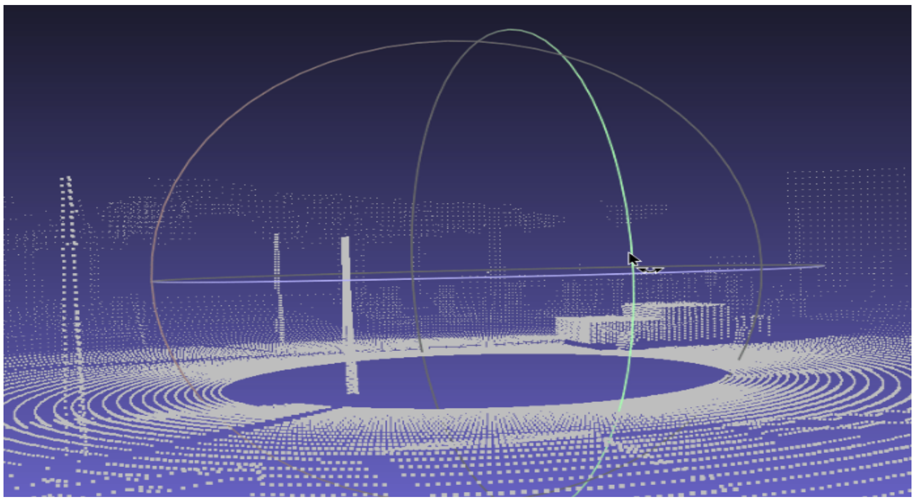


Figure 1.7: Carla Range-Based Sensors.[6]

This was the autonomous driving simulator chosen for the purposes of this dissertation because of its capability of rendering artificial but realistic scenarios with a high level of detail and for being a useful and a safe testing ground that can be used for testing new ideas for the ATLASCAR2 project. How this simulator was adapted to suit the ATLASCAR2 project will be explained later in the dissertation.

1.3 Main Motivation

In the field of technology, AD and ADAS keep on making more and more use of autonomous driving simulators in order to test their algorithms, testing their results in various training scenarios before deploying them in real world scenarios. One of the main reasons these simulators are used is to create training datasets for the learning algorithms that use images and range-based sensor data as input templates in order to recreate object models. These object models can later be used to track an object between a set of images and obtain a set of samples of the target object's location or they can simply be used to automatically recognize target objects based on the information given by those object models in order to feed the learning algorithm. Another reason is to perform initial tests on the simulator before deploying these algorithms in the real platform and compare the results. Finally, another reason is simply to facilitate the data acquisition process because since the model runs on top of the simulator this process can run much smoother when compared to the real world when sometimes it takes hours to create a simple dataset. This data recording process runs much smoother with a simulator since we don't have to worry about issues mainly caused by the hardware interface of the vehicle such as the sensor setup and calibration, vehicle damage and basic time constraints. All of these are problems that can happen in the real world and can greatly affect the data acquisition process, but these same problems can be easily fixed using a simulator in which you can control every aspect of the scenario you are creating be it the environmental conditions, the vehicle setup, the vehicle traffic, etc...
In the end of this dissertation, the ATLASCAR2 is expected to have a properly simulated sensor setup implemented in the CARLA simulator which will perform proper data acquisition using its sensors which can be used to create datasets for the ATLASCAR2. This sensor setup can be used for replicating everyday scenarios of the ATLASCAR2 and it can also be used for testing out algorithms and evaluating their performance before deploying these algorithms in the real ATLASCAR2.

1.4 Main Objectives

The main objectives for this dissertation are, firstly, to test the CARLA simulator and check its viability as a simulator for the ATLASCAR project, checking out all the possible scenarios that can be provided by this simulator.
Secondly, implement the setup of the ATLASCAR2 into the CARLA simulation and see how it interacts in the CARLA environment and record the data provided by the ATLASCAR2 sensors namely the LIDAR sensors.
Thirdly, implement some computer vision algorithms using CARLA simulated data as a tool that can be used to detect and track objects present in the CARLA world.
Finally, implement algorithms that are currently being used in the ATLASCAR2, evaluate the performance of these algorithms in the CARLA world and check if these results are valid for comparison with the results provided by the real platform.
In conclusion, the main objective of this dissertation is to provide a properly functional simulator for the ATLASCAR2 that can be used to replicate its sensors and the same scenarios that are used to test the vehicle and also store data information about the objects present in the CARLA world that can later on be used to validate algorithms that perform tasks such as object detection, tracking and labelling.

1.5 Document Structure

This document is composed of six chapters including the introduction.
In the second chapter, a literature review detailing the most significant milestones in the history of autonomous driving is presented as well as a research on some autonomous driving simulators, describing some of the related work previously done on these platforms as well as detailing some of the most significant milestones in the history of these platforms.
In the third chapter, the experimental structure of this dissertation will be described depicting the hardware setup (ATLASCAR2 and sensors) and the software setup tools such as the Robot Operating System (ROS), the Open Source Computer Vision Library (OpenCV) and the Point Cloud Library (PCL) that were used, as well as the software setup of the CARLA simulator.
In the fourth chapter, the implementation of the ATLASCAR2 setup into the CARLA simulation will be explained presenting its features, the base algorithm and the different ROS nodes used to implement this simulation.
In the fifth chapter, the results of the sensor setup configuration in CARLA are detailed as well as other experiments with CARLA simulated data. These experiments include 3 algorithms in which the first one can be used for object detection, tracking and labelling of objects present in the CARLA simulation, the second one is used strictly for object detection and the third one is used to detect the lane using road visual perception. The results and the development of these experiments will be explained, firstly by describing how each one of the algorithms operates, displaying some results of these algorithms with data simulated by CARLA and, in the end, comparing these results with results of these algorithms with data provided by the ATLASCAR2 and see how the results from CARLA match up with the results from the ATLASCAR2.
In the final chapter, the conclusions of this dissertation are presented as well as some of the future work proposed that is relation to the overall scope of this dissertation.

Chapter 2

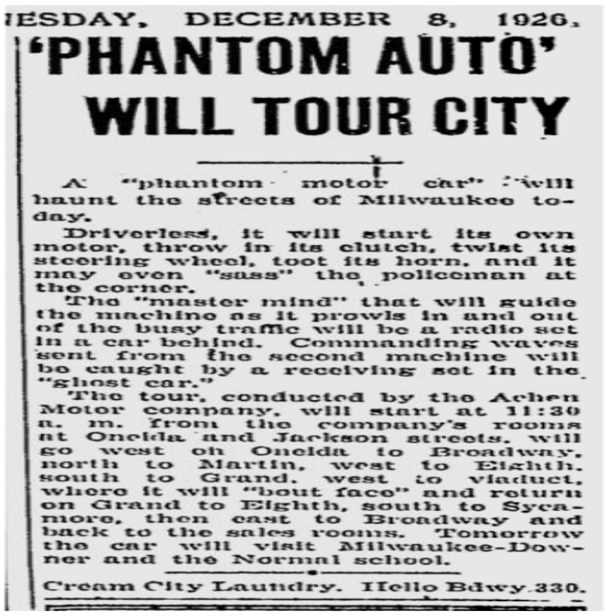
Literature Review

This chapter will elaborate upon some of the milestones in the history of autonomous driving and some of the previous work involved with CARLA as well as some other examples of autonomous driving simulators that are being used in ADAS.

2.1 Main Milestones in the History of Autonomous Driving

At the long-term, motorized road transportation has led to the accidental deaths of over 200,000 US citizens in the 1920s, with the greatest number of these deaths being pedestrians (Kröger 2016)[7].
The idea of substituting error-prone humans with techonology has been around for some time and the basic concept for this idea was practically suggested itself. The first registered experiments for this have been conducted since the 1920's (The Milwaukee Sentinel 1926) in Milwaukee (see figure 2.1)[8]. One of these experiments was a 1926 Chandler that was equipped with a transmitting antennae that was radio-controlled by a second car that followed it around.

During the 1950s, promising trials in AD took place, namely General Motors(GM) conducted experiments in miniature models alongside the electronic manufacturer Radio Corporation of America (RCA). These two companies later went on to develop a full size system that was successfully demonstrated by completing a test route of one mile (Kröger 2016)[7].
During the 1980s, a pioneer known as Ernst Dickmanns designed a vision-guided Mercedes Benz along with the engineering team of Bundeswehr University of Munich. This model managed to achieve a speed of 63 km/h on streets with no traffic. In the late 1980s, projects with bot LIDAR scanners and computer vision algorithms were carried out and in 1989 the first experiments with vehicles making use of neural networks were conducted (Pomerleau 1989)[9]. From this point onwards, various autonomous vehicle competitions have been held, one of these being a first long distance competition for driverless cars called the Defense Advanced Research Projects Agency (DARPA) Grand Challenge (DARPA 2019)[10]. This event was open to teams and organizations from around the world and these teams have participated from high schools, universities, businesses and other organizations, bringing a whole new wide variety of technological skills to the event. This challenge also offered high value money prizes to the winners and because the reward was so high, the contest brought various state-of-the-art autonomous vehicles that showcased solutions implemented in other



ESDAY, DECEMBER 8, 1926.

'PHANTOM AUTO' WILL TOUR CITY

A "phantom motor car" will haunt the streets of Milwaukee today.

Driverless, it will start its own motor, throw in its clutch, twist its steering wheel, toot its horn, and it may even "sass" the policeman at the corner.

The "master mind" that will guide the machine as it prowls in and out of the busy traffic will be a radio set in a car behind. Commanding waves sent from the second machine will be caught by a receiving set in the "ghost car."

The tour, conducted by the Achen Motor company, will start at 11:30 a. m. from the company's rooms at Oneida and Jackson streets, will go west on Oneida to Broadway, north to Martin, west to Eighth, south to Grand, west to viaduct, where it will "bout face" and return on Grand to Eighth, south to Sycamore, then east to Broadway and back to the sales rooms. Tomorrow the car will visit Milwaukee-Downer and the Normal school.

Cream City Laundry. Hello Bdwy.330.

Figure 2.1: The Milwaukee Sentinel 1926, 8 December - Phantom Auto Will Tour City.[8]

platforms featuring new ideas that used the most recent technologies (Montemerlo et al. 2006 and Thrun et al. 2007).[11]
Since then, many other companies and research organizations have been developing various prototype cars. In the past decade, various electric motored cars have emerged and new opportunities for AD and ADAS research have appeared. One of these vehicles was the Waymo by Google.
Waymo, the Google self-driving car project, begun its testing by testing driverless cars without someone at the driver position. This project was started in 2009 and it managed to count more than 5 million miles self driven. Since then, Google has recently partnered with Jaguar and designed a model that was a self-driving Jaguar I-PACEs seen in figure 2.2. Some test on the newest self-driving Waymo's vehicle have been conducted during last year 2018 (Waymo 2018).[12]

Figure 2.2: Waymo's Jaguar I-PACE Waymo 2018.[12]

Another example of these vehicles is an autonomous vehicle project known as the Uber Advanced Technologies Center (ATC) car based on an hybrid Ford Fusion seen in figure 2.3. This vehicle was equipped with state of the art LIDAR scanners, and several vision-based sensors and radars.

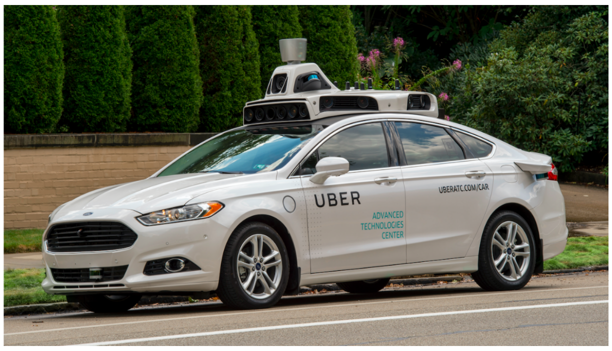


Figure 2.3: Ford Fusion Uber ATC car (Uber Advanced Technologies Group 2018).[13]

Audi has also released its new model A8 seen in figure 2.4 and the company has stated that they would be the first manufacturer to use laser scanners in addition to cameras and other sensors used in autonomous driving. This model was designed to a level 3 autonomous driving, which means, it is capable of self-driving with the expectation that the human driver will respond appropriately to a request to intervene. The Audi AI traffic jam pilot takes over the driving task in slow-moving traffic up to 60 km/h (Audi MediaCenter 2018[14] and Andreas Herrmann, Walter Brenner 2018[15]).

Figure 2.4: Audi A8 vehicle (Audi MediaCenter 2018).[14]

Another interesting example is the Shared Computer-Operated Transit (SCOT) vehicle seen in figure 2.5, conducted by the Singapore-MIT Alliance for Research and Technology (SMART) (SMART) (Singapore-MIT Alliance for Research and Technology 2018)[16]. Like the ATLASCAR 2, SCOT is also a Mitsubishi i-MiEV model used for ADAS and AD research at SMART and it is designed for operations on public roads (Andreas Herrmann, Walter Brenner 2018)[15]. The SCOT vehicle also relies on LIDAR sensors similar to ATLASCAR 2 (Teo 2014)[17].

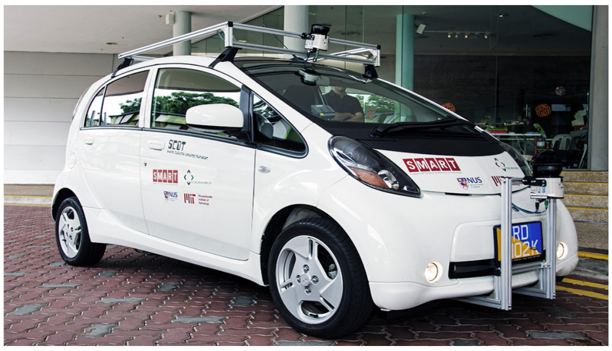
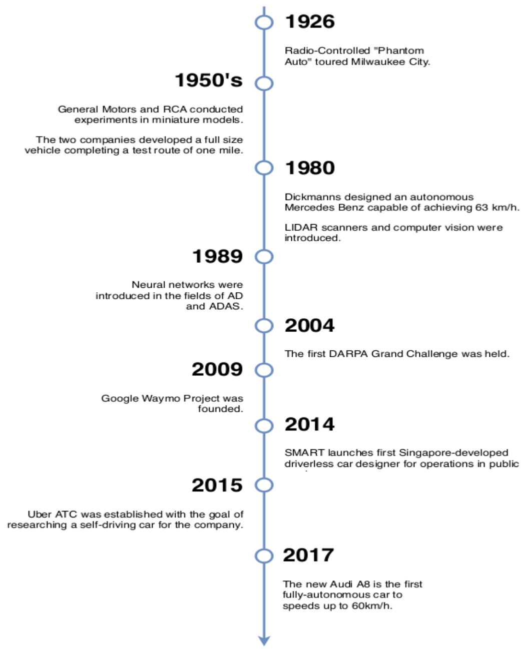


Figure 2.5: SCOT - Shared Computer-Operated Transit vehicle (Singapore-MIT Alliance for Research and Technology 2018).[16]

Like the University of Aveiro, many other universities and many other research institutes have studied the AD and ADAS paradigms. So, in resume, there have been many milestones in autonomous driving linked to AD and ADAS paradigms and a timelined with the referred milestones can be seen in figure 2.6.



1926

Radio-Controlled "Phantom Auto" toured Milwaukee City.

1950's

General Motors and RCA conducted experiments in miniature models.

The two companies developed a full size vehicle completing a test route of one mile.

1980

Dickmanns designed an autonomous Mercedes Benz capable of achieving 63 km/h.

LIDAR scanners and computer vision were introduced.

1989

Neural networks were introduced in the fields of AD and ADAS.

2004

The first DARPA Grand Challenge was held.

2009

Google Waymo Project was founded.

2014

SMART launches first Singapore-developed driverless car designer for operations in public

2015

Uber ATC was established with the goal of researching a self-driving car for the company.

2017

The new Audi A8 is the first fully-autonomous car to speeds up to 60km/h.

Figure 2.6: Timeline with some milestones in the history of autonomous driving up to 2017.

Nowadays, the autonomous driving industry has evolved a lot namely with companies like Tesla that has offered new technological updates to the autonomous driving industry. Each one of the new vehicles that are being currently released by Tesla come equipped with state of the art advanced hardware (seen in figure 2.7) such as 8 surrounding cameras that provide a 360° visibility radius around the vehicle at a distance of about 250 meters, 12 ultrasonic sensors that are used to detect rigid/flexible objects present in the road at double the distance of the camera setup and a ultrasonic radar that is used to provide information about the surrounding environmental conditions such as strong rain, fog, dust or even other vehicles that are in the front of the vehicle.

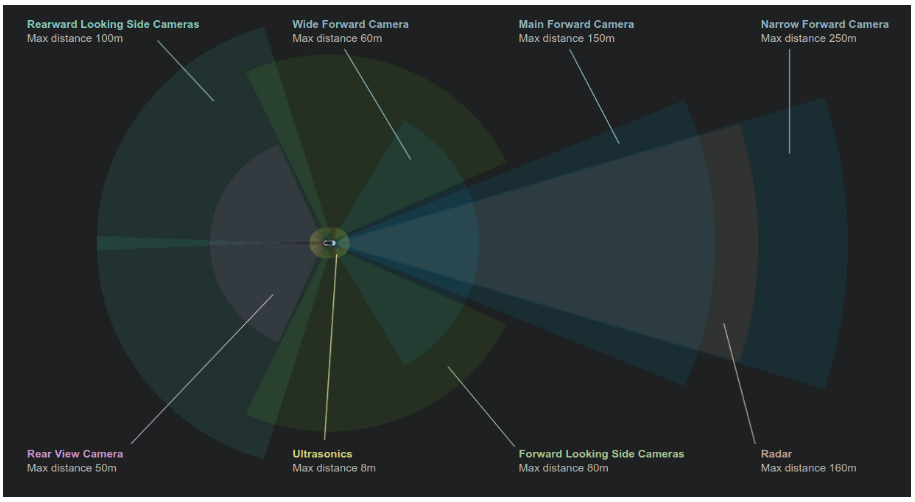


Figure 2.7: Sensor Setup of a Tesla Vehicle.[18]

All of these sensors are used in the autopilot system of these vehicles as seen in figure 2.8. This system was designed to assist the driver with the most burdensome parts of driving and introduced new features into the world of autonomous vehicles such as advanced safety, continuous software upgrades and other convenience features that majorly improved the existing functionality of autonomous vehicles making them more capable over time. The autopilot system developed by Tesla was designed to perform actions on the vehicle such as steering, accelerating and braking automatically without the need of input from the driver as long as the car is driving within its own lane. However the current features of this autopilot system require active driver supervision namely for emergency and safety measures and thus this system does not make the vehicle fully autonomous. Other factors prevent this vehicle from being fully autonomous such as the level of trust people can have on an autonomous vehicle like this one (Murat Dikmen, Catherine Burns 2017)[19] that although this trust has increased over the years this fact still remains an issue that holds back the future of autonomous driving.

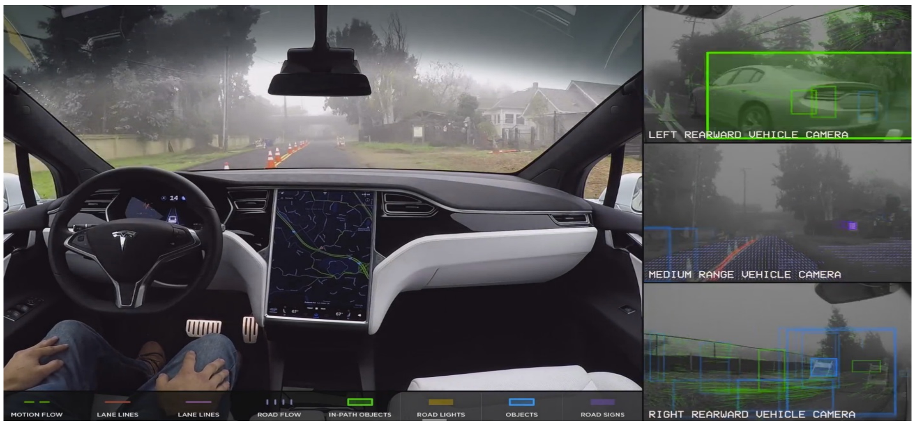

LEFT REARWARD VEHICLE CAMERA
MEDIUM RANGE VEHICLE CAMERA
RIGHT REARWARD VEHICLE CAMERA
MOTION FLOW
LANE LINES
LANE LINES
ROAD FLOW
IN-PATH OBJECTS
ROAD LIGHTS
OBJECTS
ROAD SIGNS

Figure 2.8: Tesla Autopilot Interface.[20]

2.2 Autonomous Driving Simulators used in ADAS

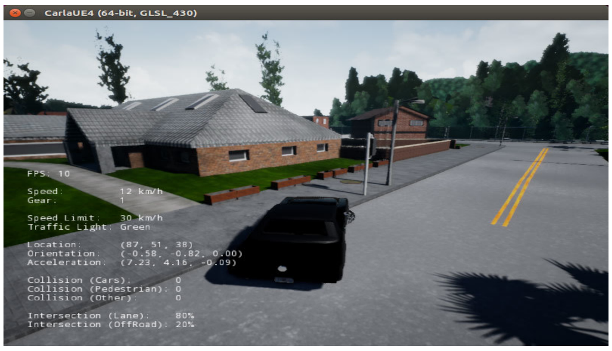
This section will elaborate upon some of the autonomous driving simulators that were relevant for the reasearch for this dissertation.

2.2.1 CARLA

Driving simulators are not new and therefore numerous realistic driving and racing simulators have been created, many of these being designed for gaming and many autonomous driving groups have used these systems in order to test their technology. From this rising surge of driving simulators, a research and development team present in the Computer Vision Center in Barcelona decided to create its own simulator as a mean to test their skills in autonomous driving technology and improve their algorithms and this simulator went on to be known as CARLA[21]. This simulator is currently on its 0.9.5 version, as seen in figure 2.9, and has been receiving very positive feedback from its community since the time it was conceived.
The CARLA (Car Learning to Act) system, described in the introduction segment, simulates a wide range of driving conditions and repeats dangerous situations endlessly to help its learning aspect. The team involved in this project has already used these situations to evaluate the performance of several different approaches to autonomous driving, some of them being used in other driving simulators. But none of these simulators managed to provide the kind of feedback that autonomous driving systems need in order to train efficiently. This level of feedback information is especially important in reinforcement learning which was one of the methods used to train CARLA, as seen in figure 2.10, because it is only based on this information that it is possible to reward or penalize the model's actions in order for it to learn properly. These systems also do not allow significant control over driving conditions or the actions on other agents. The racing simulators do not usually allow crossing traffic or pedestrians and the city simulators like Grand Theft Auto do not give control over the weather, the behavior of the other cars, traffic signals and pedestrians, cyclist and so on.
It was at this point that Dosovitskiy[5], one of the lead heads of the project and his team decided to create their own driving simulator, which came to be known as CARLA. This simulator offered a library of assets that could be arranged into towns under various weather and lighting conditions. The library included 40 different buildings, 16 animated vehicle models and 40 animated pedestrians.

The team has since used these to create two towns with several kilometers of drivable roads and then tested three different approaches to training self-driving systems and these approaches were evaluated in controlled scenarios of increasing difficulty improving the performance of its training model.

Figure 2.9: Carla World Version 0.9.5.[22]



```
FPS: 10

Speed:          12 km/h
Gear:           1

Speed Limit:    30 km/h
Traffic Light:  Green

Location:       (87, 51, 38)
Orientation:    (-0.58, -0.82, 0.00)
Acceleration:   (7.23, 4.16, -0.09)

Collision (Cars):       0
Collision (Pedestrian): 0
Collision (Other):      0

Intersection (Lane):    80%
Intersection (OffRoad): 20%
```

Figure 2.10: Carla Simulation Autonomous Driving with Reinforcement Learning.[23]

2.2.2 COGNATA

Cognata is a virtual reality simulator that uses a state-of-the-art deep learning simulation engine that leverages reality-grade city meshes combined with DNN (deep neural networks) and Artificial Intelligence (AI) capabilities in order to generate its environment. One of these environments can be seen in figure 2.11.
The Cognata's virtual reality simulator engine enables autonomous car manufactures and ADAS system providers to run thousands of different scenarios based on various geographic locations and driver behaviors, and sharing of the road with other users. Each mile driven on the simulator is equal to hundreds of miles driven regularly as it constantly creates use cases used to train the AI driving system in order to reduce the time used to reach its maturity.
The Cognata's platform can be used for autonomous analytics and since each sensor is ground-truth-ed this platform can be used in order to generate, regenerate and focus on failure cases until the solution is perfected. The platform can be used to complete regression tests in one click and can be used to control the timeline access to all raw sensor data.
The interface of this analytics platform can be seen in figure 2.12.
This platform started as a startup and now it is involved in parternships with many car manufacturating companies such as German car maker Audi AG.[24].

Figure 2.11: Cognata World Description.[25]

Figure 2.12: Cognata Platform Analytics.

2.2.3 NVIDIA DRIVE

For the purposes of autonomous driving research, NVIDIA developed the NVIDIA DRIVE which is a scalable AI platform used for autonomous driving. Their interest on this matter reflect on the idea that autonomous vehicles will transform the way we live, work, and play, creating safer and more efficient roads and in order to realize these revolutionary benefits, the cars of the future will require a massive amount of computational horsepower. Tapping into long decades of experience in AI, NVIDIA decided to launch NVIDIA DRIVE hardware and software solutions that hope to deliver industry-leading performance to help automakers, truck makers, tier 1 suppliers, and startups to make autonomous driving a reality.
The main hardware used in NVIDIA DRIVE[26] mainly correlates to their various NVIDIA DRIVE AGX platforms that are scalable, open autonomous vehicle computing platforms that serves as the brain of the operation and are used for their autonomous vehicle models. These platforms are refered to as being the only hardware platforms of its kind, delivering high-performance, energy-efficient computing for functionally safe AI-powered self-driving. One of these platforms can be seen in figure 2.13.

In terms of simulation, the NVIDIA DRIVE implemented a photorealistic simulation that was consider to be a safe, scalable solution for testing and validating a self-driving platform before it hits the road. This platform used a constellation data center as a solution to integrate the various powerfull GPUs and the NVIDIA DRIVE AGX hardware platforms. From this, advanced visualization software running on GPUs was able to simulate cameras, radar and

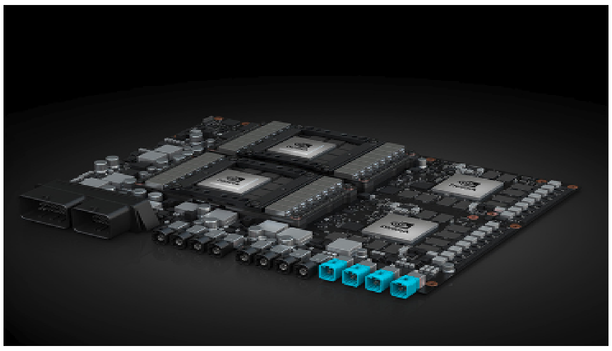


Figure 2.13: NVIDIA DRIVE AGX Computing Platform.[27]

lidar as inputs to NVIDIA DRIVE AGX platforms which were able to process the data as if it were actually driving on the road and from this same idea a scalable system capable of generating billions of miles of diverse autonomous vehicle testing scenarios to validate hardware and software platforms in the loop prior to its deployment was created. The results of this simulation can be seen in figure 2.14 and the user interface developed to test the simulation can be seen in figure 2.15.

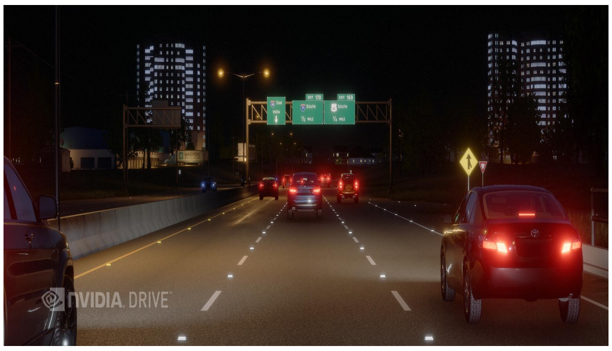


Figure 2.14: NVIDIA DRIVE Simulation.

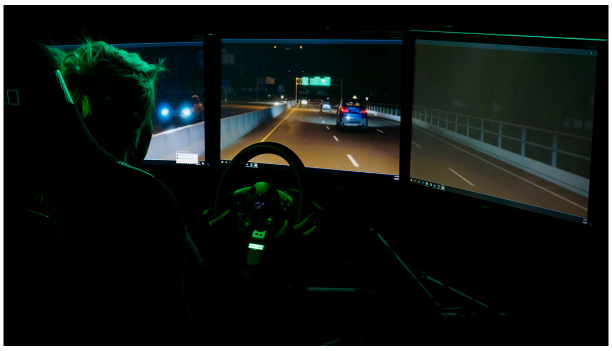


Figure 2.15: NVIDIA DRIVE Simulation With User Interface.

With the surrounding perception and highly accurate localization provided by the simulation, the first development in the NVIDIA DRIVE AutoPilot software development kit was introduced. This software package presented features such as supervised self-driving on the

highway including adaptive cruise control and lane keeping.
In hindsight, the overall purpose of the NVIDIA DRIVE AutoPilot was to provide an industry-leading driver-assistance experience, handling challenges both inside and outside of the vehicle. This package was considered to be the world's first commercially available Level 2+ Automated Driving System of its kind[28], intergrating the high-performance of the NVIDIA DRIVE AGX platforms with its driving software. This package was also used to let manufactures bring sophisticated driver assistance features, such as intelligent cockpits and visualization capabilities to the passenger cars and the commercial trucks.
This software is currently being developed and they hope to add the ability to use virtual and augmented reality to show the driver a visualization of the surrounding environment as sensed by the vehicle and its planned route, as well as detailed visualization of the NVIDIA DRIVE driver-monitoring system.
The different stages in the evolution of the NVIDIA DRIVE AutoPilot platform can be seen in figure 2.16 and the further developments of this platform with virtual and augmented reality can be seen in figure 2.17.

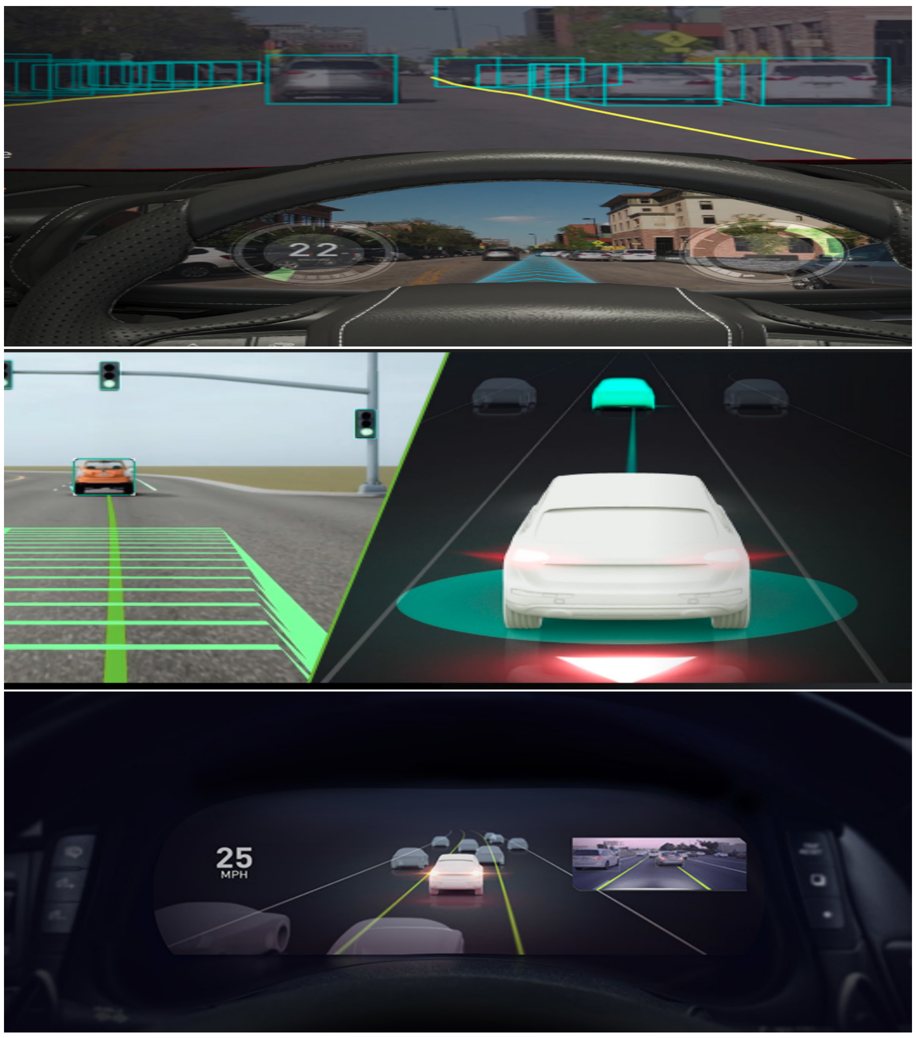


Figure 2.16: NVIDIA DRIVE AutoPilot software platform evolution.

Figure 2.17: NVIDIA DRIVE Simulation With Virtual and Augmented Reality.[28]

2.2.4 Other simulators

Here are some of the other simulators included in the research for this dissertation.

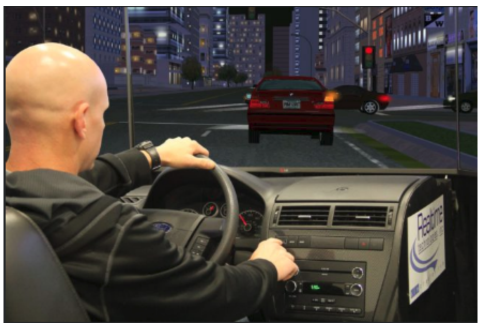
SimDriver

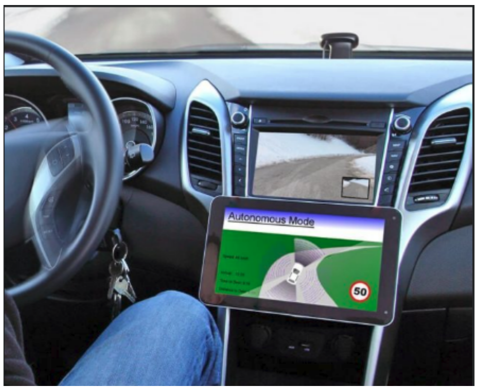
The SimDriver simulator is a component of the SimCreator DX program[29] and it was conducted as an autonomous vehicle control solution for evaluating human interaction with automated vehicles in both city and freeway driving environments.
This simulator is focused on understanding the relationship between the human driver and the autonomous vehicle and it provides the technology required to accurately measure the driver's behavior with one of the main behavior aspects being the driver attentiveness. All of this was implemented with the focus on developing a highly accurate driver model for the purposes of improving self-driving cars and other ADAS or connected vehicle technology. Some of these tecnhological enhancements being played can be seen in figure 2.18.

Figure 2.18: SimDriver Platform.[30]

The SimDriver was designed to help create scenarios that test the driver's ability to take manual control of an autonomous vehicle at a moment's notice. Once the simulation starts and the SimDriver code is loaded, the vehicle is started under manual control but the autonomous driving mode can be engaged through the cruise control button on the steering wheel or another button of choice. Once this method is activated, the vehicle is controlled through JavaScript commands and the vehicle stops accepting input from the driver.
The SimDriver will keep the vehicle on the road and maintain a specified headway distance or

(a) Autonomous Mode


(b) Manual Mode



(c) Design Applications

Figure 2.19: SimDrive simulator scenarios with design applications.

desired velocity. However, this simulator will also provide scenarios in which it may become necessary for a human to take manual control over the vehicle and this done for improving the accuracy of the model of human attentiveness or alertness which can be crucial in implementing safety features into ADAS. The autonomous mode can be disengaged at any point during the simulation and the driver can immediatly regain control of the steering wheel and the accelerator/brake pedals. Some of the scenarios implemented in this simulation can be seen in figure 2.19.

SCANeR^{TM}

The SCANeR^{TM} studio[31] is a comprehensive software suite created for automotive and transport simulation addressing features for both testing and driving for ADAS, autonomous vehicles, Human Machine Interaction (HMI) and other headlight use cases. The studio interface can be seen in figure 2.20. This studio provides a simulation tool with all the tools and models necessary in order to build an ultra-realistic virtual world and these tools include vehicle dynamics, road environments, traffic, sensors and real or virtual drivers as seen in figure 2.21. This studio is often used in conjuction with compact simulator platforms, as seen in figure 2.22, which provide the driver with realistic driving interfaces along with a large field of vision at a low cost and it is also used for massive simulation purposes in which autonomous driving functions are endlessly tested and validated and big chunks of simulated data are produced, as seen in figure 2.23.

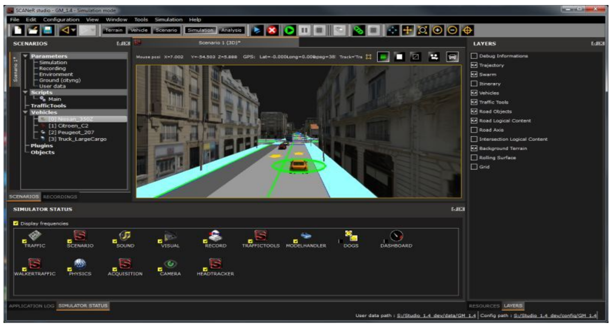


Figure 2.20: SCANeR^{TM} studio visual interface.

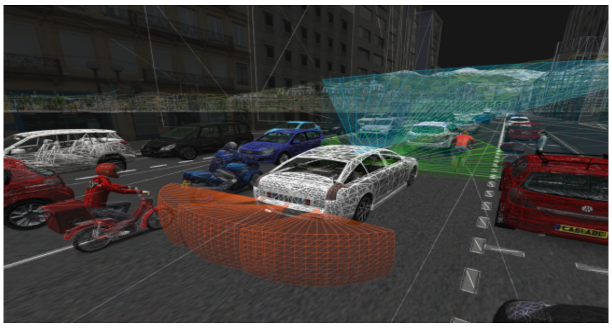


Figure 2.21: SCANeR^{TM} studio vehicle objects.

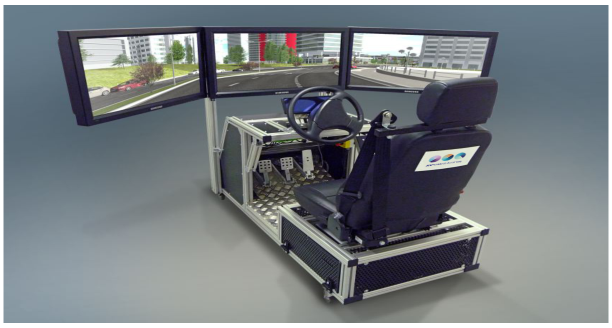


Figure 2.22: SCANeR^{TM} studio compact simulator setup.

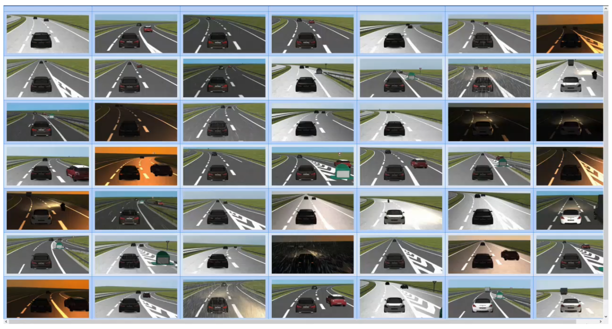


Figure 2.23: SCANeR^{TM} studio simulated data.

2.3 Overview

In order to successfully test the software and hardware tools that are going to be implemented in autonomous driving systems it is always important to implement these tools in a safe and realistic working environment and in order to produce this kind of environment, various autonomous driving research teams often make use of autonomous driving simulators to create the environment they want to use in order to test their algorithms.
Despite the title, several of the presented platforms are driving simulators marginally oriented for autonomous driving and it is important to note that these simulators are used to replicate real world scenarios in order to test their algorithms before testing them in the real world, although these scenarios do not directly translate into real world scenarios where damaging events can occur.
The scenarios provided by the simulator should not be used in order to replace real world scenarios since it is the job of the simulator to produce scenarios that are the closest thing you can get to real world scenarios and it is always productive to compare these scenarios with real ones in order to see how they can improve.
It is important that these scenarios need to be well tested under different conditions which can be controlled during the course of the simulation in order to train the model and see how it adapts to the innumerous conditions specified by the user. It is also important to reflect on the cost problems that may rise on using a simulator like this, namely budget and horsepower problems, that can affect the implementation of your autonomous driving system.
For these reasons and mainly due to the fact that CARLA is an open source simulator that is currently releasing new software releases and it provides a client API with which you can control aspects in the simulated environment directly, is why we opted to use CARLA for the purposes of this dissertation since the other autonomous driving simulators discussed previously are either not open source or required tremendous horsepower and money in order to implement them or they simply require a compact hardware platform in order to run the simulation.

Chapter 3

Experimental Infrastructure

This section will describe the hardware and software frameworks used in the development of this dissertation.
The hardware used was mainly focused on the ATLASCAR2 and its sensors: two SICK LMS151 LIDAR sensors, one SICK LD-MRS LIDAR sensor, a PointGrey Zebra 2 camera and two PointGrey FL3-GE-28S4-C cameras.
The central framework used is centered around ROS and it is used to receive messages from the simulated sensors of the ATLASCAR2 in CARLA. The data received will then be processed using the OpenCV for images and other several libraries will be used for the range-based sensors such as the PCL library.
The software framework used is centered around CARLA and its various modules.

3.1 ATLASCAR2

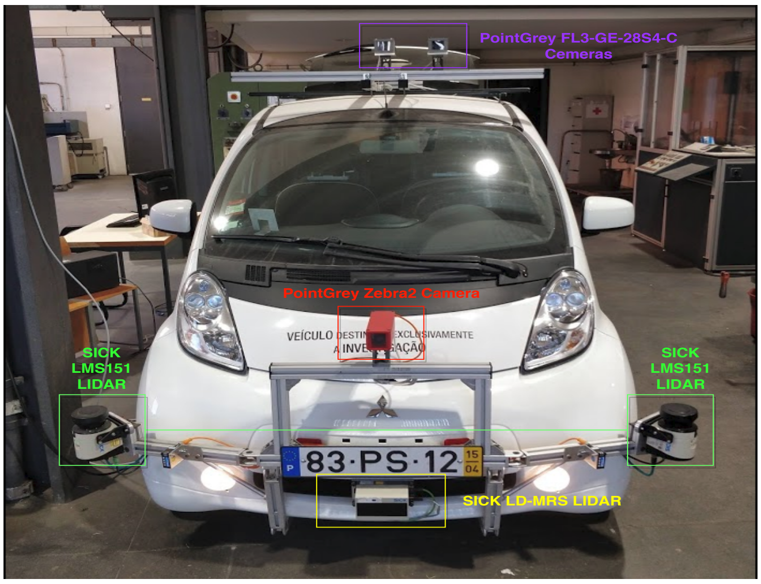


Figure 3.1: The ATLASCAR 2 based on the Mitsubishi i-MiEV platform equipped with PointGrey cameras and several LIDAR sensors.

As seen in figure 3.1, the ATLASCAR2 structure is based on the platform of the 2015 Mitsubishi i-MiEV[32], which is a full electric vehicle and it involves a battery that is used to power the engine as well the camera setup and the other sensors. The main characteristics of this model can be seen in the following table 3.1.

Mitsubishi i-MiEV Specifications	**Unit**	**Value**
Wheelbase	mm	2550
Track (Front/Rear)	mm	1310/1270
Vehicle Weight	kg	1450
Engine	-	Electric
Electric Energy Consumption	Wh/km	135
Electric Range (NEDC)	km	150
Maximum Speed	km/h	130
Minimum Turning Radius	m	4.5
Maximum Power Output	kW	49
Maximum Torque	Nm	180
Traction Battery Type	-	Lithium-ion Battery
Traction Battery Voltage	V	330
Traction Battery Energy	kWh	16
Regular charging (AC 230V 1 phase) 8A	hrs	10

Table 3.1: Mitsubishi i-MiEV technical specifications (MITSUBISHI MOTORS 2019).[32]

3.2 LIDAR Sensors

The sensors that the ATLASCAR2 has equipped correspond to two SICK LMS151 LIDAR, a SICK LD-MRS LIDAR, a PointGrey Zebra 2 Camera and a PointGrey FL3-GE-28S4-C Camera. The LIDAR sensors have been mounted in the front of the car with a aluminum infrastructure designed by Correia in 2017.[33]
These devices are then connected to a network switch installed in the car to which a computer can be plugged to receive the data from the sensors.

3.2.1 SICK LMS151

The SICK LMS151 (seen in figure 3.2) is a LIDAR sensor meant to be used outdoors. This sensor is a planar infrared scanner with a large planar aperture angle often used in robotics and in AD fields for its high scanning frequency and its operating range. This scanner is also able to scan distances through fog, glass and dust (multi-echo technology). This scanner is provided with an Ethernet TCP/IP interface with a high data transmission rate (SICK 2019).[34]
For the purposes of this project, the SICK LMS151 sensor will operate at a frequency of 50Hz with an angle increment of 0.5 between readings and each message will transmit a total of 540 points per scan in polar coordinates (r, θ) (SICK 2019).[34]

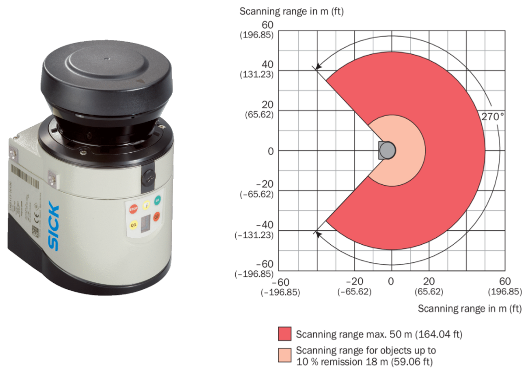


Figure 3.2: The SICK LMS151 LIDAR and its operating range (SICK 2019).[34]

SICK LMS151 Specifications	Values
Field of Application	Outdoors
Laser Class	1 (IEC 60825-1:2014, EN 60825-1:2014)
Aperture Angle	270
Scanning Frequency	25Hz / 50 Hz
Angular Resolution	0.25 / 0.5
Operating Range	0.5m ... 50m
Maximum Range with 10 percent reflectivity	18 m
Amount of Evaluated Echoes	2
Data Transmission Rate	10/100 MBit/s

Table 3.2: SICK LMS151 Sensor Specifications (SICK 2019).[34]

3.2.2 SICK LD-MRS

The SICK LD-MRS (figure 3.3) is a LIDAR sensor also meant to be used outdoors, but this sensor features 4 planar infrared scanners with 0.8 vertical aperture angle between each plan, offering tri-dimensional point clouds. This sensor also provides high scanning frequencies and long operating range up to 300 meters and is also provided with an Ethernet TCP/IP interface with a high data transmission rate (SICK 2019).[34]

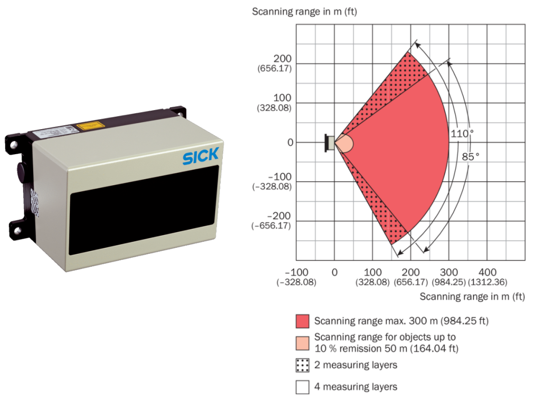


Figure 3.3: The SICK LD-MRS LIDAR sensor and its operating range (SICK 2019).[34]

SICK LD-MRS Specifications	Values
Field of Application	Outdoors
Laser Class	1 (IEC 60825-1:2014, EN 60825-1:2014)
Scanner Planes	4 measuring planes
Aperture Angle	85
Total Aperture	110
Scanning Frequency	12.5Hz / 50 Hz
Angular Resolution	0.125/ 0.25 / 0.5
Operating Range	0.5m ... 300m
Maximum Range with 10 percent reflectivity	50 m
Amount of Evaluated Echoes	3
Data Transmission Rate	100 MBit/s

Table 3.3: SICK LD-MRS Sensor Specifications (SICK 2019).[34]

For the purposes of this project, the SICK LD-MRS sensor will operate at a frequency 50Hz with an angle increment of 0.5 between readings and each message will transmit a total of 200 points per scan in polar coordinates (r, θ) for each plan and since this scanner offers four planar scans, the final point cloud will transmit a total of 800 points (SICK 2019).[34]

3.3 PointGrey Cameras

The PointGrey Zebra 2 Camera (seen in figure 3.4) is a high resolution camera with a Sony ICX274 sensor. The PointGrey FL3-GE-28S4-C Camera (seen in figure 3.4) is a high resolution camera with a Sony ICX687 sensor. Both of these cameras features a GigE Interface and both of these models are highly configurable to fulfill any particular utilization needs (PointGrey 2019).[35]
The PointGrey Zebra 2 Camera was inserted in a case made with 3D printing and it was designed by Correia in 2017[33] and the PointGrey FL3-GE-28S4-C Cameras were also inserted in a case made with 3D printing and they were introduced by Tiago Almeida in 2019[36].
Other relevant specifications can be found in table 3.4.

(a) PointGrey Zebra 2 Camera

(b) PointGrey FL3-GE-28S4-C Camera

Figure 3.4: The PointGrey Camera Modules (PointGrey 2019).[35].

PointGrey Camera Specifications	Zebra 2 Values	FL3-GE-28S4-C Values
Resolution	1624x1224	1928x1448
Maximum Frame Rate	25 FPS with HD-SDI	14 FPS
Megapixels	2.0 MP	2.7MP
Chroma	Color	Monochrome
Sensor	Sony ICX274 CCD	Sony ICX687 CCD
Image Buffer	32 MB	32 MB
Interface	GigE PoE, HD-SDI	GigE Vision

Table 3.4: PointGrey Camera Specifications for both camera modules (PointGrey 2019).[35]

Taking into consideration that the camera working at maximum resolution and at a high frame rate would cause the network bandwidth and the image processing times to increase, for the purposes of this project the frame rate of both camera modules was set to 10 fps in order to guarantee a balance between image quality and processing optimization.

3.4 Software Tools

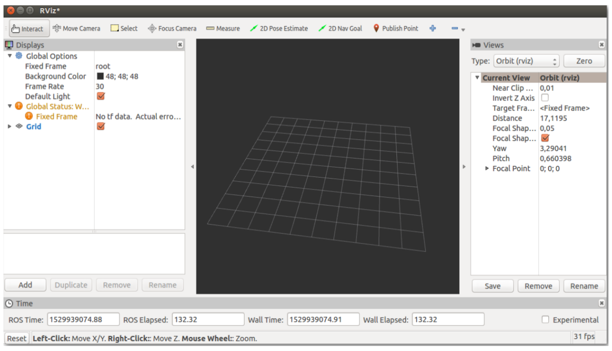
This section will be used to elaborate upon the software tools used in this dissertation. The basic architecture of these tools are centered around the ROS framework. Many ROS tools and features were used as rviz, rosbags, roslaunch and rosrun. The main nodes developed in this work were mainly focused on ROS-CARLA integration and the handling of the data information was done using libraries such as PCL and OpenCV.
All of these software tools were developed in Python and C++.

3.4.1 ROS (Robot Operating System)

The central framework used in this dissertation is based on ROS Melodic on Ubuntu 18.04.[37] The ROS framework although it is not really an operating system, it serves as a robotics middleware. This middleware offers open-source services designed for developers that need the hardware abstraction and the low-level device control.
The architectures in ROS are centered around applications called rosnodes which are used to communicate with each other creating a graph message-passing processes that can be viewed using the rqt_graph tool. Not only that, ROS can also be used to receive data packets and transform them into messages that contain data from the sensors from both CARLA and the ATLASCAR2. It is also possible to manipulate the data using ROS nodes and other tools.

RVIZ

The rviz is a standard ROS tool used for 3D visualization and it is one of the most important tools as it will be used to visualize the data from both the CARLA world and the ATLASCAR2 either directly in real time or in the case of ATLASCAR2 by connecting a computer to the car or using rosbags. This tool will also prove to be important especially in debugging since this tool can be utilized to analyze pointclouds using the RVIZ Graphical User Interface (GUI) seen in figure 3.5.[38] These point clouds are sensor messages presented in the PointCloud2 format which is the format designed as a ROS Message and this is the preferred format used for ROS applications.[38]



Global Options
Fixed Frame: root
Background Color: ■ 48; 48; 48
Frame Rate: 30
Default Light: ✓
Global Status: W...
Fixed Frame: No tf data. Actual erro...
Grid: ✓

Type: Orbit (rviz)
Current View: Orbit (rviz)
Near Clip ...: 0,01
Invert Z Axis: ☐
Target Fra...: <Fixed Frame>
Distance: 17,1195
Focal Shap...: 0,05
Focal Shap...: ✓
Yaw: 3,29041
Pitch: 0,660398
Focal Point: 0; 0; 0

ROS Time: 1529939074.88 ROS Elapsed: 132.32 Wall Time: 1529939074.91 Wall Elapsed: 132.32 ☐ Experimental

Left-Click: Move X/Y. **Right-Click:**: Move Z. **Mouse Wheel:**: Zoom.

Figure 3.5: ROS Rviz GUI (ROS Wiki).[38]

Rosbag

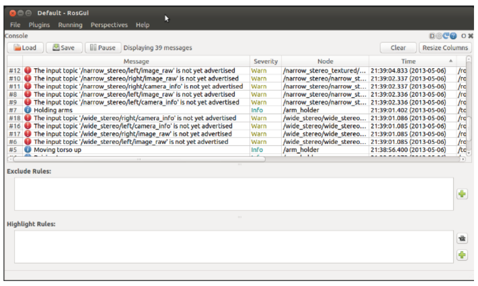
The rosbag tool corresponds to a file in ROS which is used to play recorded ROS messages from past events. While ROS is running on a device, or several devices, multiple topics related to those devices can be subscribed at once to be recorded into a rosbag.
The advantage of using this tool is to replicate working environments off-line and therefore this tool will prove to be extremely important in recording the environment from CARLA. Several rosbags were recorded throughout the development process of this project for detection, tracking and labelling purposes.
In ROS, the rosbag tool package contains a set of tools that can be used for recording and playing back messages from ROS topics. It is intended to be used in high performance and avoid deserialization and reserialization of the messages and it also features command line tools than can be used to work with rosbag file as well as code APIs that can be used to read, write and manipulate data from rosbags (ROS Wiki).[39]

Roslaunch

The roslaunch tool corresponds to files that can be used to launch multiple ROS nodes. A roslaunch file[40] sets up the roscore (ROS master node), sets the parameters via the Parameter Server and it can also be used execute other roslaunch files.
This tool includes options to automatically re-spawn ROS processes that have already died and takes in one or more XML configuration files (with the .launch extension) that are used to specify the parameters to set and what ROS nodes are going to be launched.
For example, a rosbag can be given as parameter of a roslaunch file which opens the rviz tools with a previous defined configuration in order to visualize the data from the rosbag file.
In this project, the roslaunch tool can be used to launch roslaunch files that are used to launch various ROS nodes in the CARLA simulation and manipulate the data from these ROS nodes. The transformations between the ROS and the CARLA simulation are set up using the ATLAS-CARLA integration package developed in this project and the static transform publishers implemented by ROS. The roslaunch files also prove to be advantageous namely in setting up multiple drivers needed to bring up several devices equipped in the ATLASCAR2. Since these sensors are connected to a network switch, the parameters given in the driver's roslaunch file are the IP address of each one of the devices and the driver will receive packets from the sensors and remap them into the ROS format.

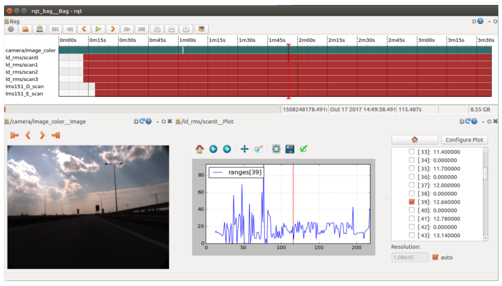
Rqt

The rqt tool corresponds to a software framework of ROS that implements various GUI tools in the form of plugins. The main plugin tools used in this project are the rqt_console which is used as viewer that displays the messages that are being published in the `rosout` ROS topic; the rqt_bag which is used to replay and display data from rosbag files and the rqt_graph which displays a graph with all the active ROS nodes along with the active ROS topics. The GUI of some these tools are shown in figure 3.6.



	Message	Severity	Node	Time
#12	The input topic '/narrow_stereo/left/image_raw' is not yet advertised	Warn	/narrow_stereo_textured/...	21:39:04.833 (2013-05-06)
#10	The input topic '/narrow_stereo/right/image_raw' is not yet advertised	Warn	/narrow_stereo/narrow_st...	21:39:02.337 (2013-05-06)
#11	The input topic '/narrow_stereo/right/camera_info' is not yet advertised	Warn	/narrow_stereo/narrow_st...	21:39:02.337 (2013-05-06)
#8	The input topic '/narrow_stereo/left/image_raw' is not yet advertised	Warn	/narrow_stereo/narrow_st...	21:39:02.336 (2013-05-06)
#9	The input topic '/narrow_stereo/left/camera_info' is not yet advertised	Warn	/narrow_stereo/narrow_st...	21:39:02.336 (2013-05-06)
#7	Holding arms	Info	/arm_holder	21:39:01.402 (2013-05-06)
#18	The input topic '/wide_stereo/right/camera_info' is not yet advertised	Warn	/wide_stereo/wide_stereo...	21:39:01.086 (2013-05-06)
#16	The input topic '/wide_stereo/left/camera_info' is not yet advertised	Warn	/wide_stereo/wide_stereo...	21:39:01.085 (2013-05-06)
#17	The input topic '/wide_stereo/right/image_raw' is not yet advertised	Warn	/wide_stereo/wide_stereo...	21:39:01.085 (2013-05-06)
#6	The input topic '/wide_stereo/left/image_raw' is not yet advertised	Warn	/wide_stereo/wide_stereo...	21:39:01.085 (2013-05-06)
#5	Moving torso up	Info	/arm_holder	21:38:56.400 (2013-05-06)

(a) rqt_console GUI

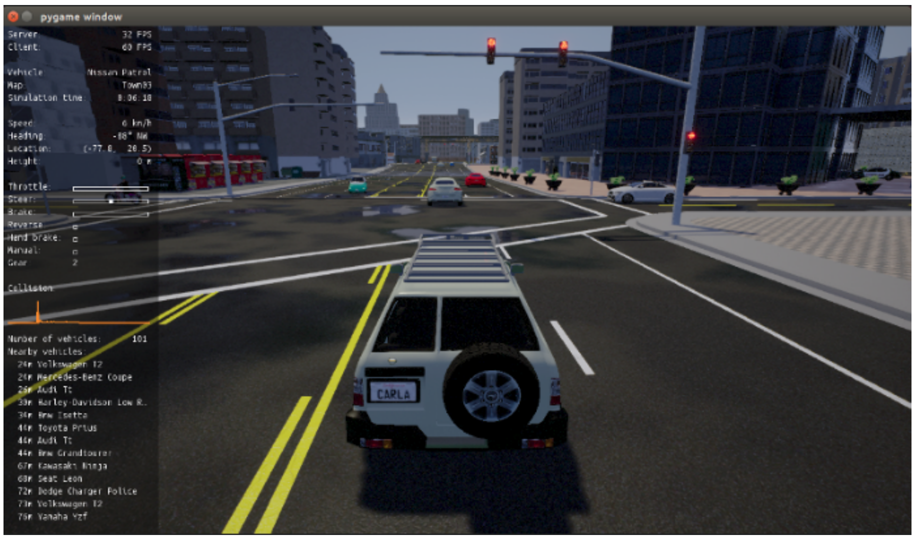


(b) rqt_bag GUI

Figure 3.6: ROS rqt tools GUI (ROS Wiki).[41]

3.5 Pygame

The Pygame library[42] is a free and open-source python programming language library used for making multimedia applications like games built on top of the excellent Simple DirectMedia Layer(SDL) library.[43]
The pygame is highly portable and runs on nearly every platform and operating system and it is used in many platforms used for gaming, art, music, sound, video and multimedia purposes. The pygame window used for this project is presented in figure 3.7.



```
Server:              32 FPS
Client:              63 FPS

Vehicle:       Nissan Patrol
Map:                  Town03
Simulation time:     0:06:18

Speed:                6 km/h
Heading:             -88° NW
Location:      (-77.0,  20.5)
Height:                  0 m

Throttle:
Steer:
Brake:
Reverse:
Hand brake:
Manual:
Gear:                      2

Collision:

Number of vehicles:      101
Nearby vehicles:
  24m Volkswagen T2
  27m Mercedes-Benz Coupe
  29m Audi TT
  33m Harley-Davidson Low R.
  38m Bmw Isetta
  41m Toyota Prius
  44m Audi TT
  44m Bmw Grandtourer
  67m Kawasaki Ninja
  68m Seat Leon
  72m Dodge Charger Police
  73m Volkswagen T2
  78m Yamaha Yzf
```

Figure 3.7: Pygame Window Example used in CARLA.[42]

3.6 OpenCV Library

The OpenCV (Open Source Computer Vision Library)[44] is an open-source BSD-licensed library that includes several hundreds of computer vision algorithms, The OpenCV has a modular structure, which means that the package includes several shared or static libraries, such as Core Functionality (core) module, which is a compact module used for defining basic data structures, including the dense multi-dimensional array Mat and basic functions used by all the other models; the Image Processing (imgproc) module, which is an image processing module that includes linear and non-linear image filtering, geometrical image transformations (resize, affine and perspective warping, generic table-based remapping), color space conversion, histograms, and so on; the Video Analysis (video) module, which is a video analysis module that includes motion estimation, background substraction and object tracking algorithms; the Camera Calibration and 3D Reconstruction (calib3d) module, which is a basic multiple-view geometry algorithms, single and stereo camera calibration, object pose estimation, stereo correspondence algorithms, and elements of 3D reconstruction; the 2D Features Framework (features2d) module, which is used for salient feature detection, and description matches; the Object Detection (objdetect) module, which is used for detection of objects anbd instances of the predefined classes (for example, faces, eyes, mugs, people, cars, and so on); the High-Level GUI (highgui) module that is an easy-to-use interface for simple UI capabilities and the Video I/O module that is an easy-to-use interface that can be used for video capturing and video codecs. There also some other helper modules such as the FLANN, Google test wrappers, Python bindings and others.

3.7 PCL Library

The PCL (Point Cloud Library) is a libary used to implement methods to process point-cloud data. The PCL framework contains numerous state of the art algorithms including filter, feature estimation, surface reconstruction, registration, model fitting and segmentation (Point Cloud Library 2019)[45]. Some of these algorithms will be used throughout this dissertation mainly to be used in conjunction with point clouds provided by CARLA.

3.8 CARLA Simulation

This section will elaborate upon some of the components developed to be used in the CARLA Simulation.

3.8.1 CARLA Simulation Engine

The simulation engine used by CARLA has been build for flexibility and realism in the rendering and physics simulation and it serves as an open-source layer over the Unreal Engine 4 (UE4), enabling future extensions passed down by its community. This engine provides state-of-the-art-rendering quality, realistic physics, basic NPC logic components and an ecosystem of interoperable plugins. Also, this engine is free and non-commercial for use. The CARLA simulates an open dynamic world and provides a simple interface between the world and the controlling agent which is the one who interacts with the world. In order to support this functionality, the CARLA was designed based on a server-client model, where the server runs the simulation, renders the scene and provides data information to the client. Once the engine establishes a network connection with a host IP address (e.g: 127.0.0.1), it opens a window with the rendered scene as seen in figure 3.8. In order to establish communication with the client, a client API implemented in Python was developed and it's responsible for the establishing connection and future interaction between the autonomous agent and the server via sockets. This client, via this API, can establish a connection with the server via a server port (e.g: 2000) and from this point onward can then send commands, or meta-commands to the server and receive sensor messages in return. The commands used to control the vehicle include steering, accelerating and braking commands. The meta-properties used to control the environment include modifying the sensor suite and the environmental properties used to control the environment including weather conditions, illumination and car/pedestrian density. When the server is reset, the agent is re-initialized at a new location specified by the client API.

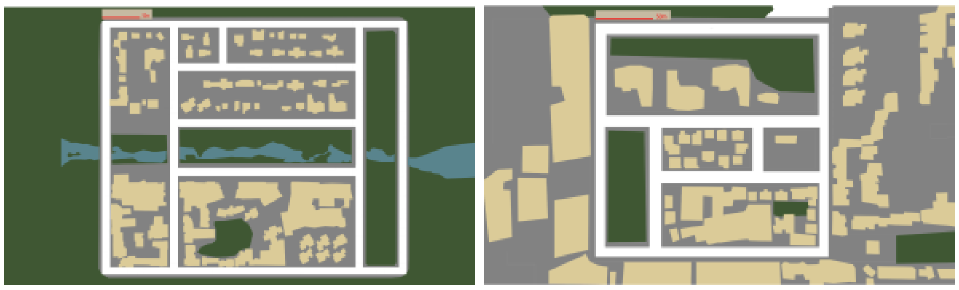
Figure 3.8: CARLA-0.9.5 Simulation Engine.

3.8.2 CARLA Simulation Environment

The environment used by CARLA is composed of 3D models of static objects such as buildings, vegetation, traffic signs, infrastructure, as well as dynamic objects such as vehicles and pedestrians.
All models were carefully designed to reconcile visual quality and rendering speed, using low-weight geometric models and textures, maintaining visual realism by carefully crafting the materials and making use of a variable of level of detail.
All 3D models were made in a common scale, and their sizes reflected those of real objects and at the time of writing of this dissertation, the asset library includes 40 different buildings, 16 animated vehicle models and 50 animated pedestrian models.
These assets were later on used to build urban environments by first laying out the roads and the sidewalks. Secondly, manually placing houses, vegetation, terrain and traffic infrastructure onto the environment and finally specifying locations where dynamic objects can appear (spawn). From these assets, a set of towns were designed by the artists over at the CARLA research and development team, with the main towns being:

- **Town 1:** total of 2.9 km of drivable roads;

- **Town 2:** total of 1.4 km of drivable roads;



(a) Town 1 (b) Town 2

Figure 3.9: Town Maps used in the CARLA simulation.

The maps of these towns can be seen in figure 3.9.
One of the main challenges faced by the developers of this platform was configurating the behavior of the non-player characters during the simulation, which is an important key used for realism. The non-player vehicles on the standard UE4 vehicle model (PhysXVehicles) and their kinematic parameters were adjusted for realism.
A basic controller was developed for basic non-player vehicle behavior such as lane following, respecting traffic lights, speed limits and intersection making decisions. Vehicles and pedestrians can also detect and avoid each other giving the possibility for many other advanced non-player vehicle controllers. Some of the vehicles and pedestrians used in the simulation are shown in figure 3.10.
The pedestrians can navigate the streets according to a town-specific navigation map, which conveys a certain location-based cost. This cost is designed to encourage pedestrians to walk along the sidewalks and along the marked road crossings, but this also allows them to cross roads at any point. Pedestrians can also wander around the town in accordance with the map, avoiding each other and trying to avoid vehicles. If a vehicle collides with a pedestrian, this pedestrian is then deleted from the simulation and a new pedestrian is spawned at a different location according to a certain timestamp. In order to increase the level of visual diversity, the non-player characters will appear with a random appearance when they are added to the simulation, in which each pedestrian is clothed in a random outfit sample from a pre-specified wardrobe and this pedestrian can be optionally equipped with one or more following items such as smartphones, shopping bags, guitar cases, suitcases, rolling bags or umbrellas. The vehicles are rendered randomly according to a model-specific set of materials.
A variety of atmospheric conditions and illumination regimes were implemented in the course of the CARLA simulation. These differ in the position and color of the sun, the intensity and color used for sky radiation, as well as ambient occlusion, atmospheric fog, cloudiness and precipitation. Currently, the simulator supports two lighting conditions - midday and sunset - as well as nine weather conditions based on weather sets, differing in cloud cover, level of precipitation and the presence of puddles in the streets. This results in a total of 18 illumination-weather combinations. Some of these weather conditions are illustrated in figure 3.11.
These weather conditions are important because they will affect how the vehicle will behave during the course of the simulation. For example, if it is raining a lot, the simulated vehicle will be forced to reduce its velocity.

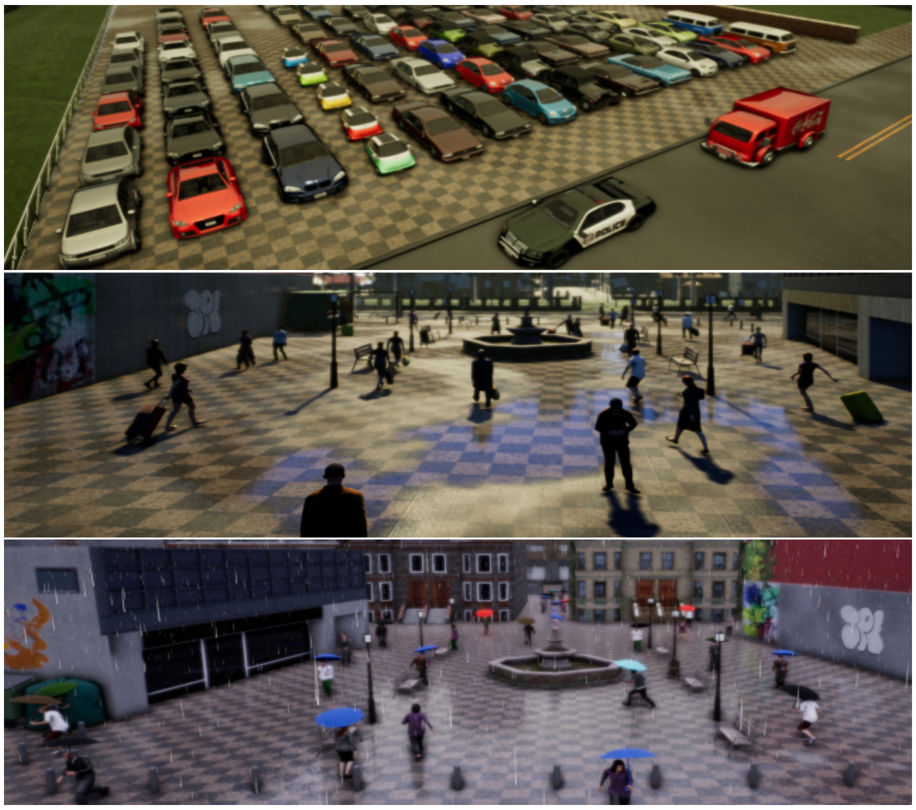


Figure 3.10: CARLA-0.9.5 Vehicles and Pedestrians.

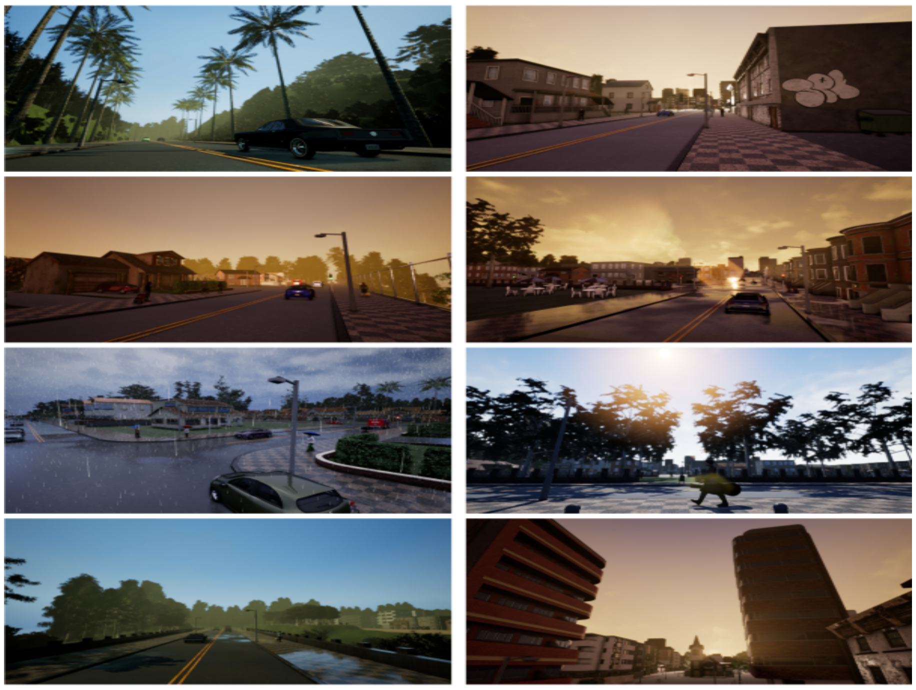


Figure 3.11: CARLA-0.9.5 Weather Conditions.

3.8.3 CARLA Simulation Sensors

The sensors used by CARLA allow for flexible configuration of the agent's sensor suite and at the start of this dissertation, the sensors were limited to RGB cameras and pseudo-sensors that provide some ground-truth and semantic segmentation. At the time of writting of this dissertation the CARLA simulator now displays 3 camera models (RGB, Depth and Segmentation), LIDAR ray-cast and Odometry sensors, a GPS module and a LaneInvasion and CollisionEvent sensors.
The camera sensors are illustrated in figure 3.12 and the LIDAR results are illustrated in figure 3.13.

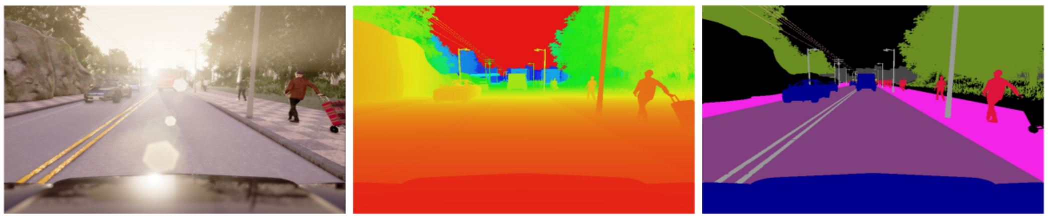


Figure 3.12: CARLA-0.9.5 Simulation Camera Sensors.

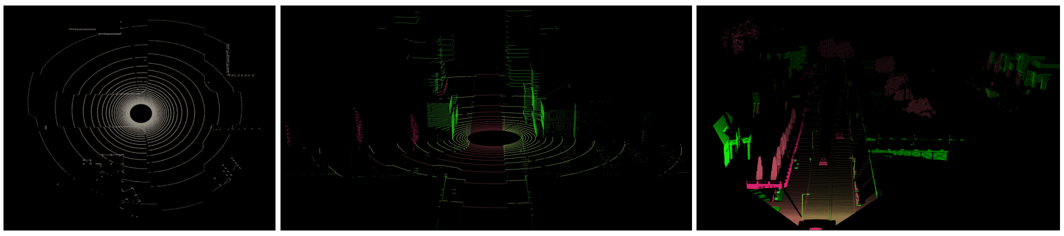


Figure 3.13: CARLA-0.9.5 Simulation LIDAR Results.

The number of sensors used, their type and their position were previously parameters that were established in the vehicle blueprint designed in UE4 but now these parameters can be specified by the client using the client API and the sensor parameters include 3D location, 3D orientation with respect to the car's coordinate system, its Field of View (FOV) and the depth of said field. The CARLA's semantic segmentation pseudo-sensor provides 12 semantic classes such as: road, lane-marking, traffic sign, sidewalk, fence, pole, wall, building, vegetation, vehicle, pedestrian and other. In addition to sensor and pseudo-sensor readings, the CARLA also provides a range of measurements associated with the state of the agent and compilance with traffic rules. The measurements used to translate the agent's state include vehicle location and orientation with respect to the world coordinate system (GPS/Compass), speed, acceleration and accumulated impact from collisions. The measurement used to translate traffic rules include the vehicle's footprint percentage that impinges on the wrong-way lanes or the sidewalks, as well as the states of the traffic lights and the speed limit at the current location of the vehicle. Finally, the CARLA also provides access to the exact locations and bouding boxes of all the dynamic objects present in the environment and these signals played an important role in training and evaluating the driving process and it will also proved to be important later on in this dissertation.

3.9 CARLA Autonomous Driving

The CARLA is a platform used to support development, training and detailed performance analysis used for autonomous driving systems and this platform provides two driving methods that can be used to control the vehicle in CARLA. One of these methods is based on manual control in which the vehicle is controlled using the WASD keys of the keyboard and the other method is based on an autopilot module that was developed by CARLA's development team. Both of the driving methods were used during the course of this dissertation as a way to evaluate the vehicle's performance in the CARLA simulation and during the research part of this dissertation, it was revealed that the autopilot developed in CARLA was evaluated according to three approaches to autonomous driving.
The first approach was a modular pipeline that relied on dedicated subsystems for visual perception, planning and control, and the architecture involved with this approach is in line with most existing autonomous driving systems. The second approach was based on a deep network trained end-to-end via imitation learning and this approach was based on a long line of investigation that has recently attracted renewed interest and finally the third approach was based on another deep network but this one was trained end-to-end via reinforcement learning.
All three of these approaches make use of an agent that interacts with the environment over discrete time steps in which in each time step, the agent gets an observation value and based on that value produces a certain action. This action, based on the observation values, could be one of 3 exclusive actions: steering, throttle or brake actions. The observation value was determined by a tuple of sensory inputs which included high-dimensional sensory readings, such as color images and depth maps, and lower-dimensional sensory readings such as speed and GPS. In addition to this, all of these approaches made use of a plan provided by a high-level topological planner that takes into consideration the current position of the agent and current location of the goal as inputs and based on these inputs uses an algorithm to provide a high-level plan to the agent in order for him to follow and reach his goal. This plan was designed to advice the agent in order for him to perform actions such as turn left, turn right or keep straight at intersections. However, it was found out that this plan does not provide a trajectory for the agent to follow nor does it contain any geometric information of relevancy to the agent and because of this it is possible to assume that the plan used in all of these approaches is a weaker form of any plan that can be given by any common GPS navigation applications which guide human drivers and autonomous vehicles present in the real world and they do not provide metric maps.
The development team conducted a series of experiments using each one of these approaches on the very same agent, not fine-tunning anything separetely for each scenario, but increasing the level of difficulty on each trial phase. These experiments were conducted in both towns (Town 1 and Town2) with different weather conditions.
In the end, the overall performance of all methods was not perfect, not even on simple tasks such as driving in a straight line and the success rate of these methods further declined as long as the level of difficulty increased. However it was found out that it was easier to generalize the agent to new weather conditions than it is to generalize the agent to new town conditions mainly due to the fact of the maps made for both of these towns being very different.
The results from these experiments served to showcase the susceptibility of end-to-end approaches to rare events such as braking or swerving to avoid hitting a pedestrian which were in rare occurrence during the training phase of this model and while CARLA can be used

to increase the frequency of such events during training to support end-to-end approaches, deeper advances in machine learning algorithms and model architectures are necessary for significant improvements in the robustness of this model.
As it stands, the autopilot module in CARLA is currently being developed and needs more testing in other environments and based on this idea, the CARLA's reasearch and development team have launched the CARLA Autonomous Driving Challenge[46] which is a challenge in which various teams can submit their own self-contained autonomous driving agents to compete in the CARLA world as docker images. These agents would then be evaluated on the task of safe driving, which requires the agents to reach a target destination following a pre-defined route without traffic infractions. During driving, these agents will have to face challenging traffic situations modeled after different traffic scenarios according to a collection of routes and for each one of these routes there will be repetitions under different weather and traffic conditions. The goal of this challenge is to gauge the overall performance of the agents in order to find which agent has the best training method and this will depend on the percentage of routes successfully completed without any critical infractions such as crashing against vehicles or any other infractions such as running a red light.
A figure illustrating a route example of this challenge, highlighting the starting point (blue dot) and the target destination (red dot) along with the road trajectory(green curve) can be seen figure 3.14 and a figure illustrating two of the many traffic conditions in the CARLA Autonomous Driving Challenge can be seen in figure 3.15.

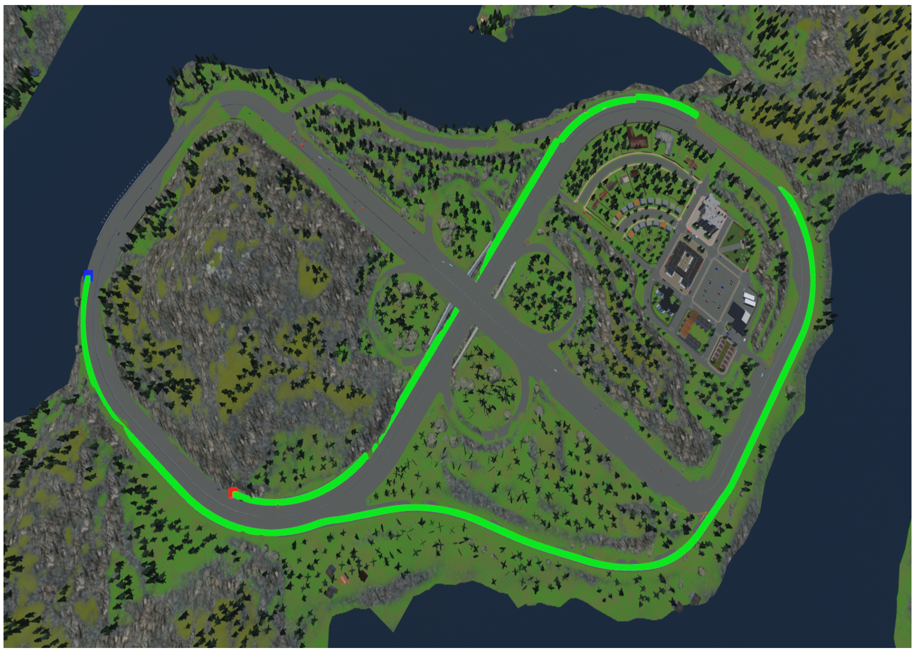


Figure 3.14: Illustration of a route, highlighting the starting point (blue dot) and the target point (red dot) along with the trajectory (green curve) in the CARLA Autonomous Driving Challenge.[46]

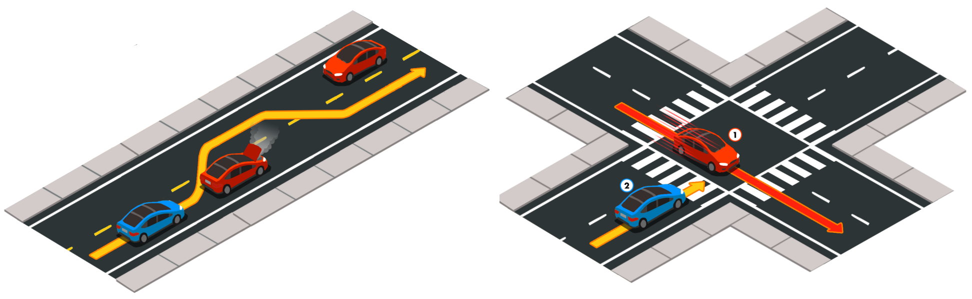


Figure 3.15: Illustration of two traffic situations in the CARLA Autonomous Driving Challenge.[46]

3.9.1 CARLA ROS Bridge Package

One can argue that the CARLA was chosen as the autonomous driving simulator used for this dissertation based on it being open source and being capable of displaying very good, complex and realistic scenarios without very powerfull hardware but the main reason for this is the fact that CARLA can be used in conjunction with ROS which is perfect since the programs that run on the ATLASCAR2 also run on ROS.
The integration between ROS and CARLA is achieved via the `carla_ros_bridge` package developed by the CARLA development team at the CVC in Barcelona.[47]
This package serves as an interface that connects CARLA 0.9.X to the ROS framework showing the results present in the CARLA world with RVIZ and this package will be presented as a middlepoint that connects the simulation and the Pygame used to control the simulated vehicle using the `manual_control.py` script present in the CARLA Python API. This script will also spawn a vehicle from a random blueprint from CARLA with a set of sensors attached to it along with a certain rolename and if this rolename is within the list of rolenames specified by the ROS parameter file present in the `carla_ros_bridge`, this vehicle will be interpreted as being the controlled vehicle and all the relevant ROS topics of the vehicle will be available in the `carla_ros_bridge`.

This package is used for handling ROS dependencies, marking the objects present in the CARLA simulation, such as cars or pedestrians using RVIZ markers and it can also be used to record rosbag files using the rosbag command line. This package will use ROS AckermannDrive messages that implement the Ackermann steering principle[48] that is based on the idea that the inner wheel (which is the wheel closest to the Instantaneous Center of Rotation (ICR) point of the vehicle) should steer for a bigger angle than the outer wheel allowing the vehicle to rotate around the middle point between the rear wheel and its axis and causes the inner wheel to travel with a slower speed than the outer wheel. The Ackermann driving mechanism allows the rear wheels to have no slip angle which is defined by the orientation of the local velocity vector relative to the wheel's symmetry plane, and requires that the ICR point of the vehicle to be on a straight line defined by the rear wheels' axis.
The ROS Ackermann Drive messages were needed in order for the vehicle markers to move according to the vehicle objects present in the simulation and the Ackermann Drive mechanism used to control the vehicle objects is in display in figure 3.16.

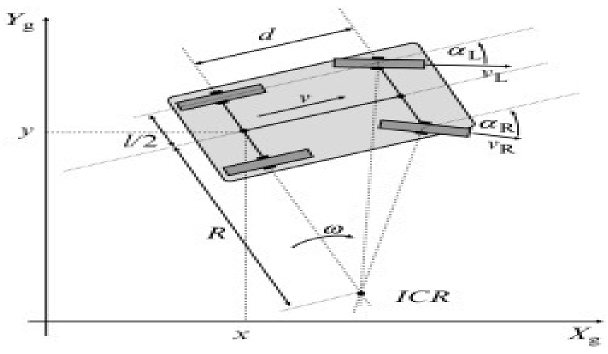
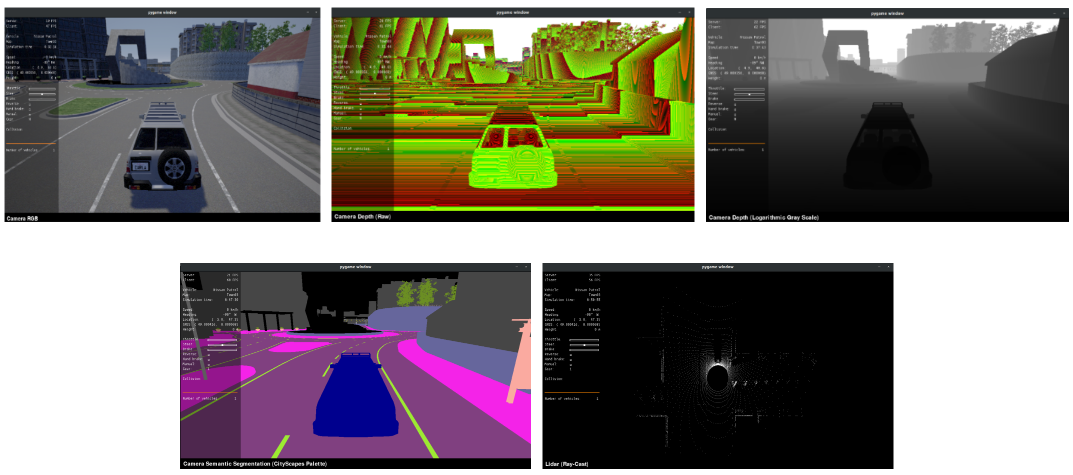


Figure 3.16: Ackermann Driving Mechanism.

The Pygame window used to control the simulated vehicle will spawn this vehicle in simulation along with list of stats about the vehicle and a CameraManager model which will display each sensor reading from each one of the sensors attached to that vehicle in a dfferent window, be it the different camera results or the LIDAR results as seen in figure 3.17.



pygame window
Camera RGB
Camera Depth (Raw)
Camera Depth (Logarithmic Gray Scale)
Camera Semantic Segmentation (CityScapes Palette)
Lidar (Ray-Cast)

Figure 3.17: Sensor Readings from the Sensors attached in the CARLA simulated vehicle

Depending on which sensor reading is being presented in the CameraManager window, the ROS topic being published will change as will the results being shown in RVIZ. For example, if the CameraManager window is displaying the LIDAR results, then it will display a rosgraph similar to figure 3.18 and the results from the ROS topic will be presented in RVIZ (figure 3.19).

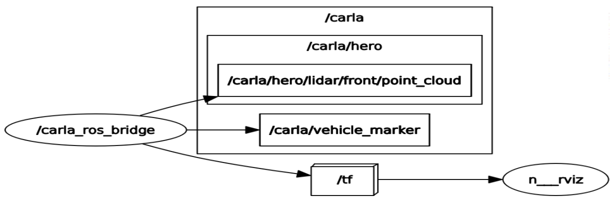


Figure 3.18: ROS Graph Published via LIDAR Results.

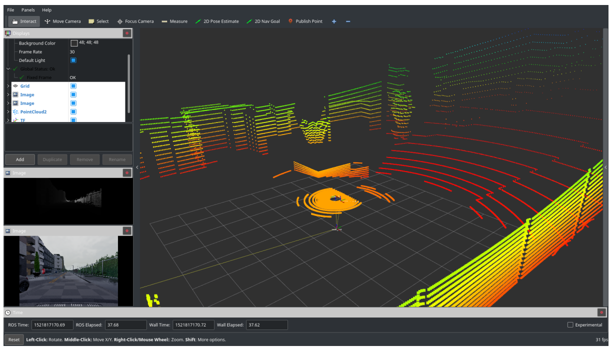


Figure 3.19: LIDAR Results from CARLA Simulation Illustrated in RVIZ present in the ROS Bridge.

In order to setup this package, it is necessary to first configure a ROS environment by creating a catkin workspace and install the `carla_ros_bridge` package.
Then, in order to actually use the package, the simulator must be executed by running the shell script present in the simulator package. This simulator will run a server in which the client will be connected to using the client API.
The `carla_ros_bridge` will also need to have access to the CARLA library and this access is established by exporting a PYTHONPATH via an .egg file present in the CARLA repository.
After all of this is set and done, it is safe to start the `carla_ros_bridge` package via a roslaunch file that reads the setup parameters via another .yaml file and from this point forward this package will have access to the ROS topics of the simulated vehicle while at the same time being able to present the results from the ROS topics of the simulation in RVIZ.
An overall diagram depecting the placement of this package in the simulation process is represented in figure 3.20 and the ROS topics being published with its visualization results being shown in RVIZ is presented in figure 3.21.

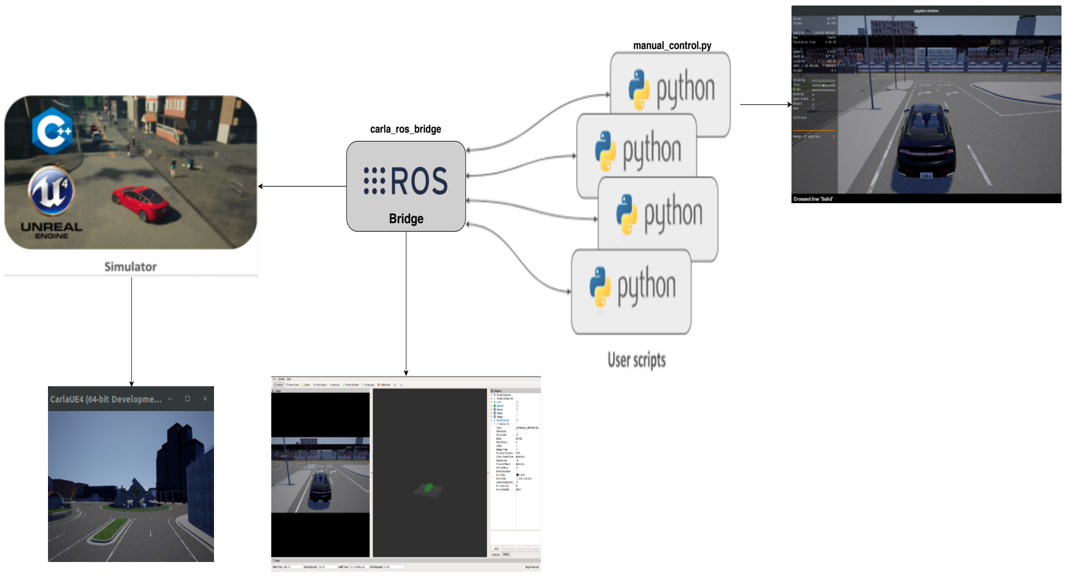
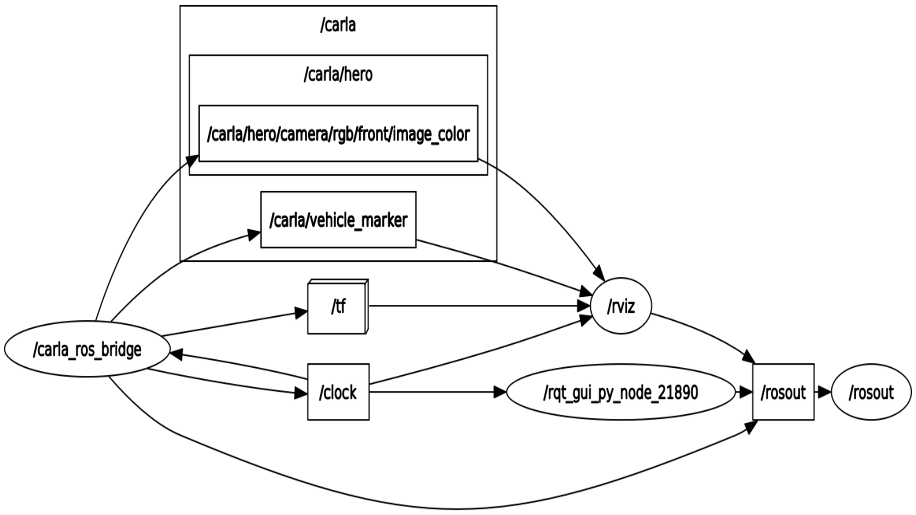
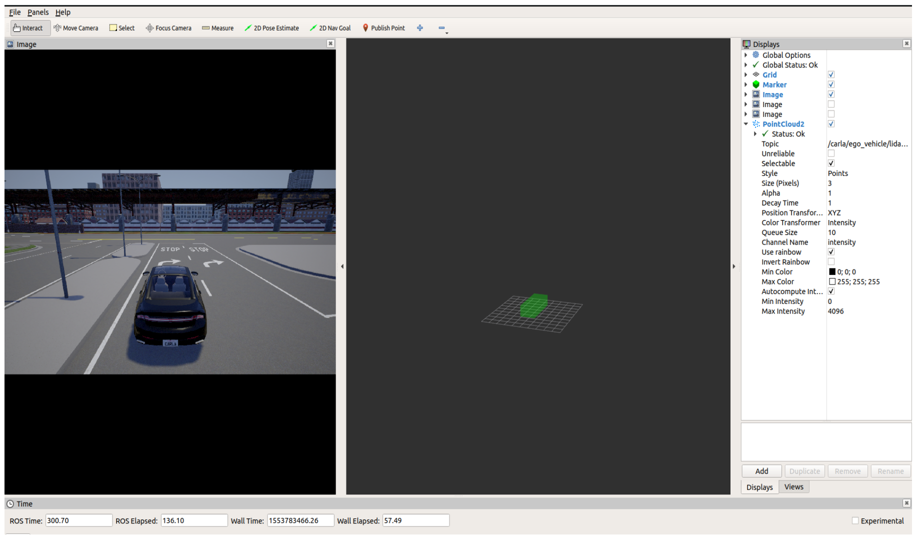


Figure 3.20: CARLA ROS Bridge Package Placement in the CARLA simulation presenting the results of the simulation in RVIZ.





Displays

Global Options	
Global Status: Ok	
Grid	✓
Marker	✓
Image	✓
Image	
Image	
PointCloud2	✓
Status: Ok	
Topic	/carla/ego_vehicle/lida...
Unreliable	
Selectable	✓
Style	Points
Size (Pixels)	3
Alpha	1
Decay Time	1
Position Transfor...	XYZ
Color Transformer	Intensity
Queue Size	10
Channel Name	intensity
Use rainbow	✓
Invert Rainbow	
Min Color	■ 0; 0; 0
Max Color	□ 255; 255; 255
Autocompute Int...	✓
Min Intensity	0
Max Intensity	4096

ROS Time: 300.70 ROS Elapsed: 136.10 Wall Time: 1553783466.26 Wall Elapsed: 57.49

Figure 3.21: CARLA ROS Bridge Rosgraph with the ROS Topics of the simulation with its results visualized in RVIZ with the green marker representing the controlled vehicle.

3.9.2 CARLA ROS Bridge Overview

The purpose of this package is to create a bridge based on ROS for the CARLA simulator and the Pygame in which you can visualize the occurring events currently happening in the simulation in RVIZ.
As it currently stands it was not possible to change the vehicle using the `carla_ros_bridge` package alone therefore other ROS nodes were developed in order to realize this task.
The next chapter in this dissertation will elaborate upon some of the major changes made on this package implemented during the course of this dissertation and the implementation of the ATLASCAR2 setup in the CARLA world.

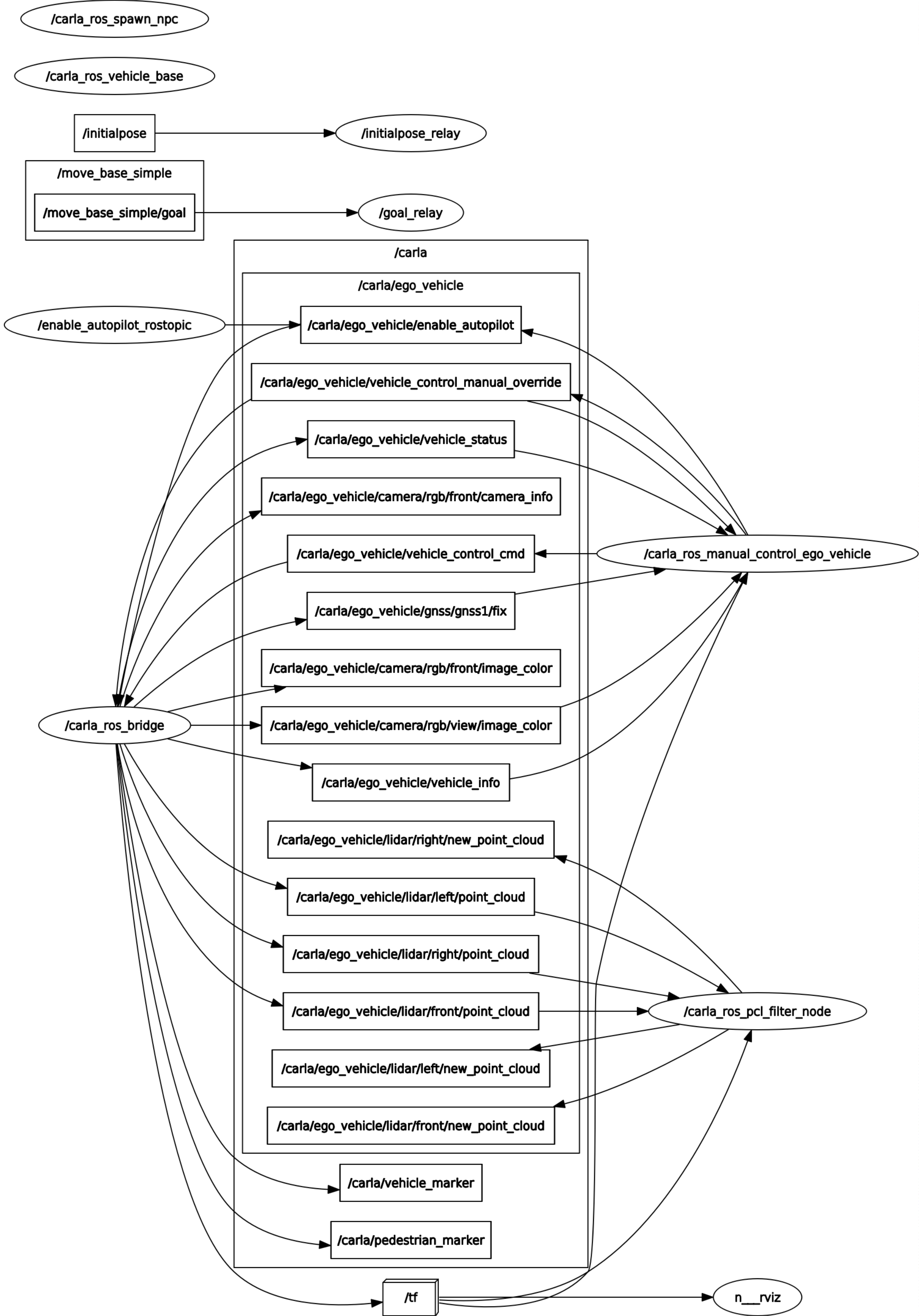
Chapter 4

Adaptation of the ROS Bridge Package And Implementation of the ATLASCAR2 Setup

The use of sensors for perception to recognize the environment is one of the most, if not the most important requirement for vehicles in autonomous driving. A car equipped with several sensors needs to know the position of each sensor so that the readings can be aligned with each perception obtained. This factor is very important in completing one of the main tasks of this dissertation which was to adapt the ATLASCAR2 setup in a vehicle using the CARLA simulator and test its results in the CARLA world.

4.1 New ROS Bridge Architecture

In the following page, a graph is shown with a series of ROS topics from different ROS nodes used in this package. As stated before, the `carla_ros_bridge` package alone was not enough to change the vehicle setup which was one of the main points of this dissertation and so this package needed to be implemented in a different architecture with a set of ROS nodes each one designed for a specific purpose. The `carla_ros_bridge` establishes a connection between the different nodes as well as the server and creates a series of ROS topics that are going to be used to pass information about the vehicle and that includes information about the sensor data and the current status of the vehicle. The `carla_ros_manual_control` node spawns a Pygame window which can be used to control the vehicle in the simulation and the `carla_ros_vehicle` node is the one in charge of the setup of the vehicle and spawning this vehicle in the simulation. The `carla_ros_pcl_filter` node is used to treat the information regarding the LIDAR sensors of the vehicle and adapt them to the context of the ATLASCAR2. The following sections will be used to describe the progress involved during the development of the `ros_bridge` package starting with the ATLASCAR2 sensor setup implementation. However, in order to talk about sensor setup implementation, we first have to talk about the vehicle blueprints and how they work in CARLA.


/carla_ros_spawn_npc
/carla_ros_vehicle_base
/initialpose
/initialpose_relay
/move_base_simple
/move_base_simple/goal
/goal_relay
/carla
/carla/ego_vehicle
/enable_autopilot_rostopic
/carla/ego_vehicle/enable_autopilot
/carla/ego_vehicle/vehicle_control_manual_override
/carla/ego_vehicle/vehicle_status
/carla/ego_vehicle/camera/rgb/front/camera_info
/carla/ego_vehicle/vehicle_control_cmd
/carla_ros_manual_control_ego_vehicle
/carla/ego_vehicle/gnss/gnss1/fix
/carla/ego_vehicle/camera/rgb/front/image_color
/carla_ros_bridge
/carla/ego_vehicle/camera/rgb/view/image_color
/carla/ego_vehicle/vehicle_info
/carla/ego_vehicle/lidar/right/new_point_cloud
/carla/ego_vehicle/lidar/left/point_cloud
/carla/ego_vehicle/lidar/right/point_cloud
/carla/ego_vehicle/lidar/front/point_cloud
/carla_ros_pcl_filter_node
/carla/ego_vehicle/lidar/left/new_point_cloud
/carla/ego_vehicle/lidar/front/new_point_cloud
/carla/vehicle_marker
/carla/pedestrian_marker
/tf
n___rviz

4.2 Vehicle Blueprints in CARLA

This section will be used to describe how the vehicle blueprints work in CARLA.
In CARLA, all the vehicle models are defined in blueprints which contains the information necessary (e.g: vehicle color, sensor information) to create the vehicle and spawn it as new actor in the CARLA world and this blueprint has an ID that uniquely identifies it and all the actor instances created with the blueprint (e.g: "vehicle.nissan.patrol", "sensor.camera.depth").
The vehicle blueprint is one of these blueprints and the way that the sensor setup works in CARLA is that the vehicle sensors are already attached to vehicle when you design the vehicle blueprint in UE4 and because of this it is not possible to add new sensors or change the position of these sensors without changing the model using the UE4 editor.
This vehicle blueprint is special since it has a set of other blueprints attached to it, as seen in figure 4.1, such as the skeletal mesh which is a set of polygons composed to make up the surface of the animated object combined with a hierarchical set of interconnected bones which is responsible for the animation of the object; the physics asset which is used to define the physics and the collision parameters used by the skeletal mesh; the animation blueprint which is responsible for controlling the animations of the skeletal mesh; a tire configuration data asset and one or more wheel blueprints, depending on how many wheels the model has, which are used to define the data information about the wheels.
The most important component of this scheme is the vehicle blueprint which is the one that is going to be read by the simulator in order to spawn the vehicle with the components defined in the blueprint in the simulated environment. This blueprint defines every single attribute present in the vehicle when it is spawned in CARLA, be it the colour, the number of wheels and even the sensor setup list present in the vehicle. Naturally, in order to implement the ATLASCAR2 setup in the CARLA module certain changes had to be made in the setup of the vehicle namely the sensor setup. This setup was described as a list of vehicle attributes present in the vehicle blueprint that when launched in CARLA would spawn the sensors together with the vehicle in a certain position.

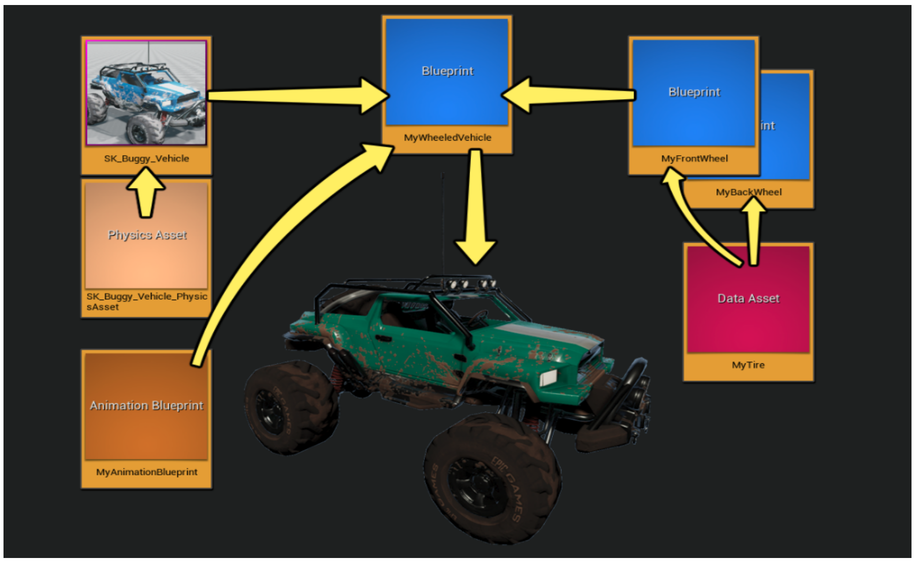


Figure 4.1: CARLA Vehicle Model Components in Unreal Engine 4.

4.3 ATLASCAR2 Sensor Setup Implementation in CARLA

This section will be used to describe the implementation of the new sensor setup of the CARLA vehicle used in order to mirror the setup of the ATLASCAR2.
In order to change the sensor setup implementation, a ROS node called `carla_ros_vehicle` was developed and the purpose of this node is to read every single attribute present in the vehicle blueprint except the sensor setup list associated with that blueprint. This node will also create a new blueprint with every attribute of the previous blueprint except the sensors. Naturally, the next thing to do was to create a configuration file for the sensor setup. This file corresponds to a .JSON file that specifies each and every sensor present in the vehicle along with each and every attribute of that sensor such as the type which is the blueprint ID of the sensor (ex:sensor.camera.rgb), the id (ex:front camera, left lidar), the position which follows the (x,-y,z) coordinate system, the orientation, the width and height as well as the FOV of the sensor. A layout of the sensor configuration file is described in listing 4.1.

```json
 {
  "sensors": [
    {"type": "sensor.camera.rgb",
      "id": "front",
      "x": 2.0, "y": 0.0, "z": 1.0, "roll": 0.0, "pitch": 0.0, "yaw": 0.0,
      "width": 800,
      "height": 600,
      "fov": 100},
    {"type": "sensor.camera.rgb",
      "id": "view",
      "x": -4.5, "y": 0.0, "z": 2.8, "roll": 0.0, "pitch": -20.0, "yaw": 0.0,
      "width": 800,
      "height": 600,
      "fov": 100},
    {"type": "sensor.lidar.ray_cast",
      "id": "front",
      "x": 2.0, "y": 0.0, "z": 2.4, "roll": 0.0, "pitch": 0.0, "yaw": 0.0,
      "fov": 50}
  ]
}
```

Listing 4.1: Sensor Setup .JSON First Configuration.

Using this file, the task of adding more sensors to the vehicle becomes more flexible since it is possible to add as many sensors as desired to the vehicle, as long as the sensors types are blueprint IDs recognized by the CARLA module, and it is also possible to change each one of the sensor attributes of the sensor in order to achieve the best sensor setup configuration. As seen in the listing 4.1 the sensor attributes such as the type, the id, the position, the rotation, the width, the height and the field of view must be specified in the configuration. The width and height camera attributes must be aligned with the image resolution determined when the simulator engine is started and these cameras will work at a frame rate 10fps which is the benchmark frame rate provided when you start the simulator engine.
In order to mirror the ATLASCAR2 camera setup, it was opted to only use the RGB camera models provided by CARLA. In fact, two RGB cameras were used each with its own parameters describing its position, rotation and its calibration. This form factor was used in order to give a different perspective for the vehicle object in which one of the cameras is used as a "view" camera that displays the vehicle in its entirety and the objects around the vehicle from

an outside perspective and the other camera is used as a "front" camera that displays the objects present in front of the vehicle from an inside perspective close to the driver's position.

In order to mirror the ATLASCAR2 lidar setup, it was opted first to just use one LIDAR sensor and see how it behaved. This sensor was placed in front of the vehicle with a FOV of 50 ° in order to fit in the maximum range for the FOV of the LIDAR sensor that the CARLA allowed at the time which was around 100°. It is important to note that a FOV of 85 ° was also tested but the FOV of 50 ° presented cleaner results mainly due to factors such as the limited range of the sensor. It is also important to note that this FOV was only horizontal. The results of the RGB cameras can be seen in figure 4.2 and the results from the LIDAR sensor can be seen in figure 4.3.

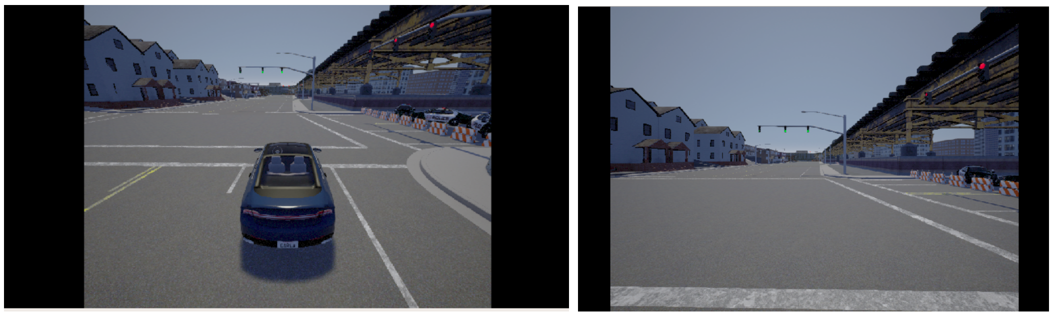

(a) Camera View (b) Camera Front

Figure 4.2: RGB camera results visualized in RVIZ.

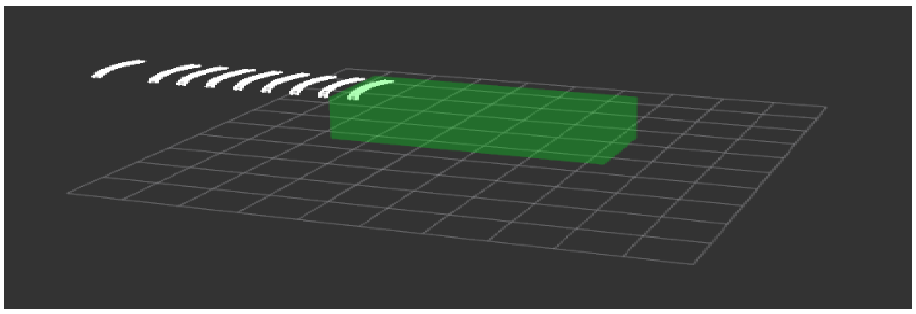
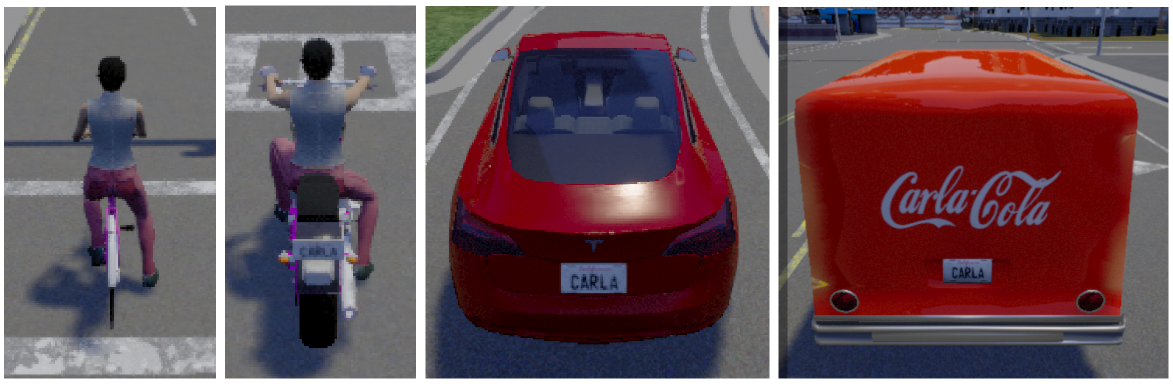


Figure 4.3: LIDAR data results visualized in RVIZ with a horizontal FOV of 50°.

It is important to note that the LIDAR sensor had to be positioned in a place in which it would not be obstructed by the vehicle model itself since these LIDAR sensors are very sensitive and they sometimes do not show the correct results in RVIZ mainly because the sensor readings are being excluded because of the size of the vehicle. Also these results are from an older version of the CARLA simulator (CARLA-0.9.1)[22] which had many limitations mainly in the LIDAR sensors which were not fully developed and it had issues with the GPU which in turn did not allowed us to visualize the correct results in RVIZ. Also, in this version, we couldn't specify certain parameters partaining to the LIDAR sensors such as the number of channels and its range. These problems were later solved by upgrading the simulator to its newest version which at the time of writting of this dissertation is the CARLA-0.9.5.

4.4 Vehicle Control using ROS and a Pygame Window

After the sensor setup implementation and the configuration of the sensor attributes have been completed, the sensor setup selection process can begin in which the `carla_ros_vehicle` node mentioned previously reads the sensor setup list present in the configuration file, selects the sensors and sets them up as new vehicle attributes in the blueprint that is going to be used to spawn the vehicle in the CARLA world. Once the vehicle setup is all set and done, the vehicle blueprint is sent to the CARLA simulator and the simulator spawns the vehicle that is specified in the blueprint according to a list of spawning points that is provided by the simulator. In order to freely control this vehicle, a ROS node named `carla_ros_manual_control` is used to control the vehicle from CARLA. This node is a ROS node that is used to control the vehicle from CARLA via sockets and ROS messages, and it was developed based on the `manual.py` script present in the PythonAPI of the CARLA simulator. All the data is received and published via ROS topics and it allows for manual steering as well as automatic steering which uses machine learning algorithms to compute the correct path.
This node also spawns a Pygame window that can be used to control the vehicle from CARLA once the `carla_ros_bridge` node establishes a connection between the server present in the CARLA simulator and the client. The vehicle blueprints used by these packages are from random models from CARLA (seen in figure 4.4), each one with a different size and this factor proved to be useful in order to test results in different sensor positions and see if there were major differences in the sensor readings depending on the simulated model.



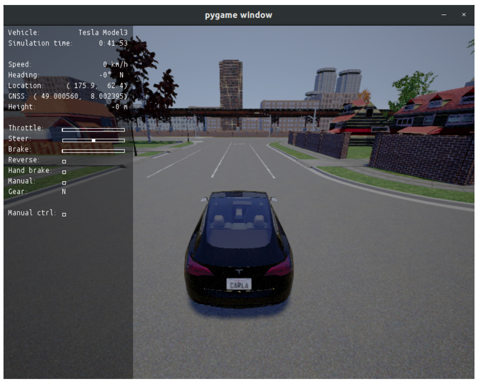
(a) Bike (b) Motorcycle (c) Car (d) Truck

Figure 4.4: CARLA Vehicle Models with Different Sizes and Dynamic Sensor Positions.

As seen in figure 4.5, once the `carla_ros_bridge` node establishes a connection between the CARLA simulator and the client, the vehicle spawned in the `carla_ros_vehicle` node will appear in the Pygame window and from this point forward, this node can be used to control the vehicle via the Pygame using WASD keys from the keyboard. By clicking on the H key, it will open a help list of all the commands that can be used in the Pygame and by clicking on the C key, the weather conditions present in the simulation can be altered in order to achieve the desired weather. The help list is shown in figure 4.6 and the same pygame with different weather conditions is presented in figure 4.7.

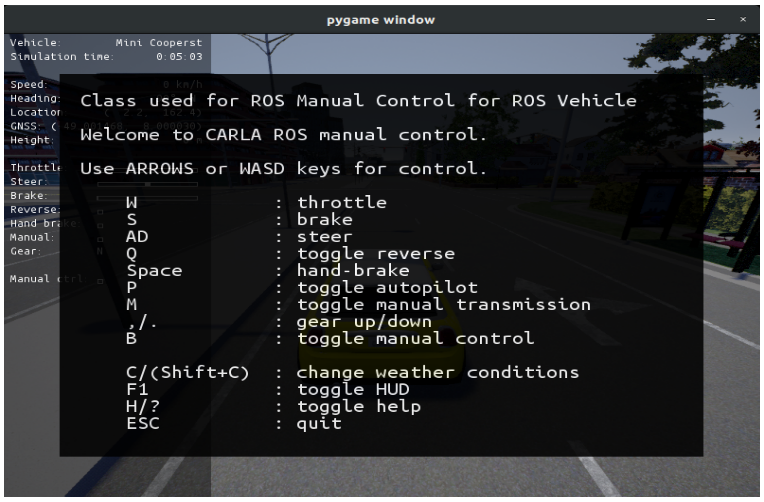
Press 'H' or '?' for help.

(a) Pygame Window without CARLA Connection



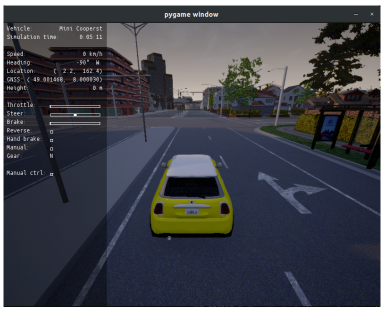
(b) Pygame Window with CARLA Connection

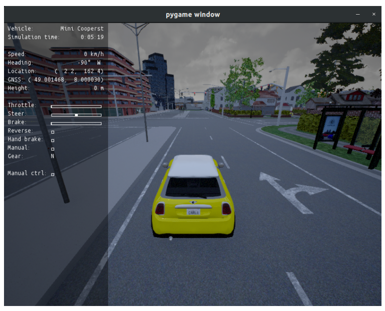
Figure 4.5: CARLA ROS Manual Control Pygame Connection.



```
Class used for ROS Manual Control for ROS Vehicle

Welcome to CARLA ROS manual control.

Use ARROWS or WASD keys for control.

    W              : throttle
    S              : brake
    AD             : steer
    Q              : toggle reverse
    Space          : hand-brake
    P              : toggle autopilot
    M              : toggle manual transmission
    ,/.            : gear up/down
    B              : toggle manual control

    C/(Shift+C)    : change weather conditions
    F1             : toggle HUD
    H/?            : toggle help
    ESC            : quit
```

Figure 4.6: Pygame Help Guide.


(a) Sunset Weather


(b) Cloud Weather

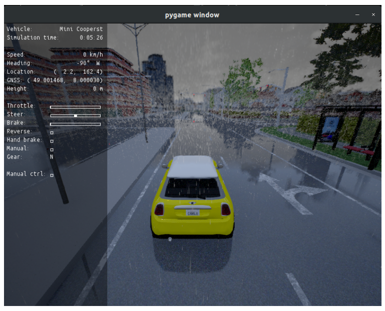

(c) Rainy Weather

Figure 4.7: Different weather conditions displayed in the same Pygame Window.

In addition to this, in the top left corner of the pygame window it is possible to see a list of stats with information about the vehicle such as the name of the model, such as the vehicle acceleration and its velocity, and so by analyzing the results provided by the Pygame window it is possible to check the vehicle's status as the vehicle is being controlled by the user.
This module provides two methods to control the vehicle: a manual control and auto pilot. The pygame automatically starts with the manual control method and it can alternate to auto pilot by pressing the P key from the keyboard and you can return back to the manual control mode by pressing the M key from the keyboard.

4.5 ATLASCAR2 Sensor Configuration with New Version of CARLA.

By upgrading to the CARLA 0.9.5 version, there were some major improvements done in the sensor department namely the LIDAR sensors. This module can now be handled differently by specifying new attributes to the sensor which help to make this module more robust. These attributes are the range of the sensor which is the maximum measurement distance measured in centimeters, the number of lasers integrated in the sensor, the number of points generated by all lasers per second, the rotation frequency and the number of seconds between each sensor capture (number of ticks). The FOV attribute of sensor was also altered to a vertical FOV which was divided into two sections: an upper FOV and a lower FOV.
This sensor now simulates a rotating LIDAR implemented using ray-casting, in which the points are computed by adding a laser for each channel distributed in the vertical FOV and the rotation is simulated by computing the horizontal angle that the LIDAR rotated in the frame and calculating the ray-cast for each point that each laser was supposed to generate the frame by applying the following equation:

$$ray\_cast = \frac{points\_per\_second}{frames\_per\_second \times number\_of\_channels} \tag{4.1}$$

A table with the attributes of the LIDAR sensor along with its default values according to the CARLA documentation is presented in table 4.1 and the new .JSON layout format for the sensor configuration file can be seen in listing 4.2.

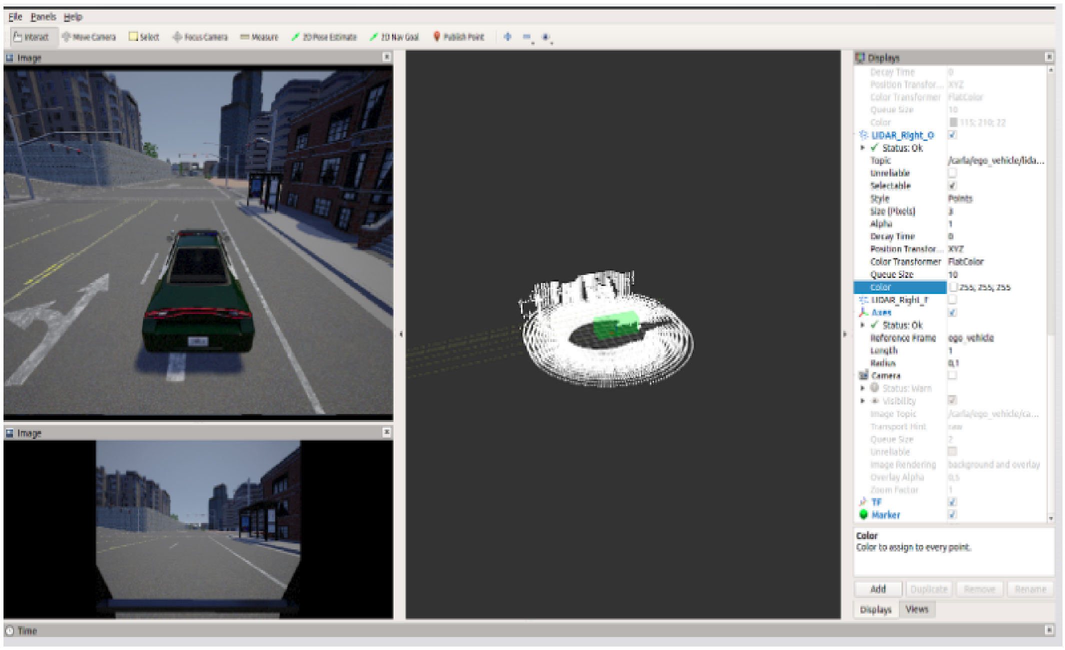
LIDAR Blueprints Attributes	**Types**	**Default Values**	**Description**
channels	int	32	Number of lasers
range	float	1000	Maximum measurement distance in centimeters
points_per_second	int	56000	Points generated by all lasers per second
rotation_frequency	float	10.0	Lidar rotation frequency
upper_fov	float	10.0	Angle in degrees of the upper most laser
lower_fov	float	-30.0	Angle in degrees of the lower most laser
sensor_tick	float	0.0	Seconds between sensor captures (ticks)

Table 4.1: LIDAR Blueprint Attributes From Carla Documentation.[6]

```
{"sensors": [
    {"type": "sensor.camera.rgb", "id": "front",
      "x": 2.0, "y": 0.0, "z": 1.0, "roll": 0.0, "pitch": 0.0, "yaw": 0.0,
      "width": 800, "height": 600, "fov": 100},
    {"type": "sensor.camera.rgb", "id": "view",
      "x": −4.5, "y": 0.0, "z": 2.8, "roll": 0.0, "pitch": −20.0, "yaw": 0.0,
      "width": 800, "height": 600, "fov": 100},
    {"type": "sensor.lidar.ray_cast", "id": "front",
      "x": 2.0, "y": 0.0, "z": 2.4, "roll": 0.0, "pitch": 0.0, "yaw": 0.0,
      "range": 1000, "channels": 32,
      "points_per_second": 56000,
      "upper_fov": 10.0, "lower_fov": −30.0,
      "rotation_frequency": 10},
    {"type": "sensor.lidar.ray_cast", "id": "left",
      "x": 2.0, "y": −1.3, "z": 2.4, "roll": 0.0, "pitch": 0.0, "yaw": 0.0,
      "range": 1000, "channels": 32,
      "points_per_second": 56000,
      "upper_fov": 10.0, "lower_fov": −30.0,
      "rotation_frequency": 10},
    {"type": "sensor.lidar.ray_cast", "id": "right",
      "x": 2.0, "y": 1.3, "z": 2.4, "roll": 0.0, "pitch": 0.0, "yaw": 0.0,
      "range": 1000, "channels": 32,
      "points_per_second": 56000,
      "upper_fov": 10.0, "lower_fov": −30.0,
      "rotation_frequency": 10}]
}
```

Listing 4.2: Sensor Setup .JSON Second Configuration.

The RVIZ results from this setup configuration can be seen in figure 4.8.



```
Decay Time              0
Position Transfor...    XYZ
Color Transformer       FlatColor
Queue Size              10
Color                   115; 210; 22
LIDAR_Right_O           ✓
  Status: Ok
  Topic                 /carla/ego_vehicle/lida...
  Unreliable
  Selectable            ✓
  Style                 Points
  Size (Pixels)         3
  Alpha                 1
  Decay Time            0
  Position Transfor...  XYZ
  Color Transformer     FlatColor
  Queue Size            10
  Color                 255; 255; 255
LIDAR_Right_F
Axes                    ✓
  Status: Ok
  Reference Frame       ego_vehicle
  Length                1
  Radius                0,1
Camera
  Status: Warn
  Visibility            ✓
  Image Topic           /carla/ego_vehicle/ca...
  Transport Hint        raw
  Queue Size            2
  Unreliable
  Image Rendering       background and overlay
  Overlay Alpha         0,5
  Zoom Factor           1
TF                      ✓
Marker                  ✓

Color
Color to assign to every point.
```

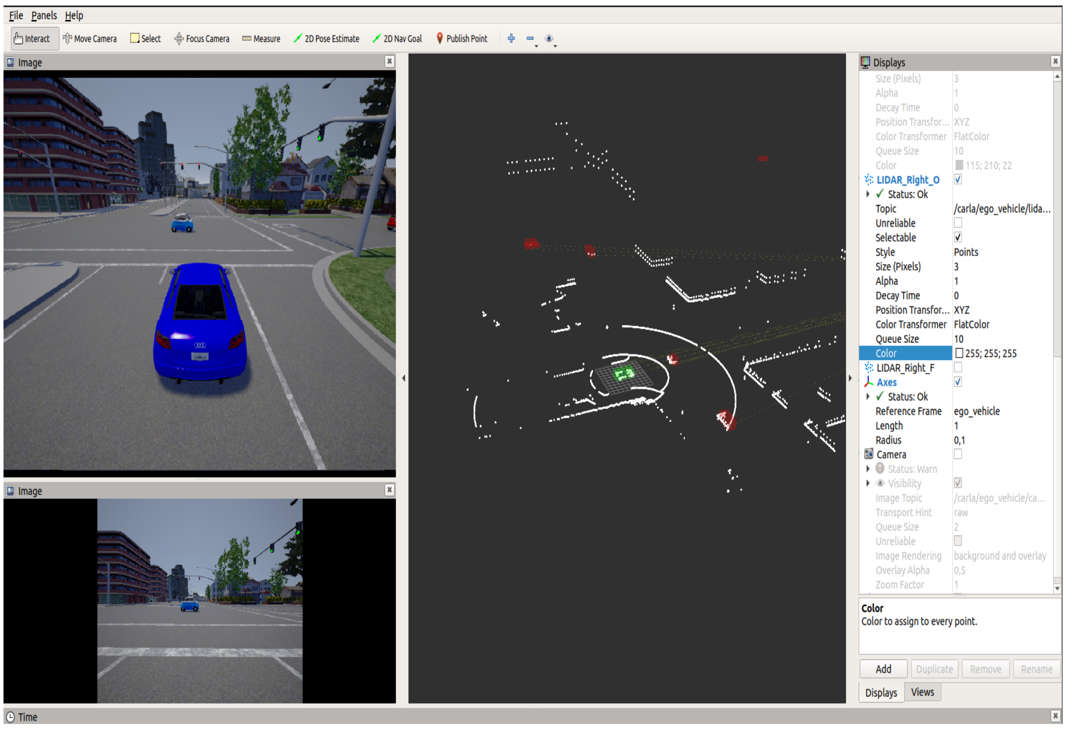
Figure 4.8: RVIZ Results from second configuration with 2 cameras (front + view) and 3 LIDAR sensors (front + left + right).

From this point onward the only thing left to do was to change the attributes of the LIDAR sensor in order to suit the specifications of the ATLASCAR2.
The SICK LD-MRS LIDAR sensor has a range that goes from 0.5 meters to a maximum of 300 meters and so we decided to use a range of 250 meters which amounts to a range of 25000 centimeters and for the case of the SICK LMS151, the maximum range for this sensor is 50 meters which amounts to a range of 5000 centimeters. The SICK LD-MRS model features 4 planar infrared scanners with a 0.8° vertical aperture angle between each plan and this form factor was able to be replicated by messing around with the FOV attribute of the LIDAR sensor giving an 1.6° vertical angle to the upper and lower FOVs giving it a 3.2° angle to the FOV of the sensor. The SICK LMS151 only features 1 planar infrared scanner and this scanner has no vertical FOV. Both of these sensors will operate at a frequency of 50Hz, meaning the sensor readings will be captured with a interval of 0.02s and each message will transmit a total of 200 points per scan giving a final point cloud with 800 points in the case of the SICK LD-MRS sensor and for the SICK LMS151 LIDAR it will transmit a final point cloud with 540 points since it only has one laser scan and each message transmits a total of 540 points per scan. With these values it is possible to calculate the value for the `points_per_second` attribute by dividing the total number of points by the time (in seconds) of the sensor tick and so the SICK LD-MRS will generate 40000 points by all lasers per second and the SICK LMS151 will generate 27000 points by all lasers per second.
Since both of these sensors will operate at a frequency of 50Hz, the rotation frequency for each one of these models will also be 50Hz and this value was chosen in order to match the scanning frequencies of each sensor.
The value for each attribute in the blueprint is presented in table 4.2.
The third .JSON layout format of the sensor configuration file can be seen in listing 4.3 and the RVIZ results for this configuration can be seen in figure 4.9.

LIDAR Blueprints Attributes	**Types**	**LD-MRS**	**LMS151**	**Description**
channels	int	4	1	Number of lasers
range	float	25000	5000	Maximum measurement distance in centimeters
points_per_second	int	40000	27000	Points generated by all lasers per second
rotation_frequency	float	50.0	50.0	Lidar rotation frequency
upper_fov	float	1.6	0.0	Angle in degrees of the upper most laser
lower_fov	float	-1.6	0.0	Angle in degrees of the lower most laser
sensor_tick	float	0.02	0.02	Seconds between sensor captures (ticks)

Table 4.2: LIDAR SICK LD-MRS & SICK LMS151 Blueprint Attributes.

```
{"sensors": [
  {"type": "sensor.camera.rgb", "id": "front",
    "x": 2.0, "y": 0.0, "z": 1.0, "roll": 0.0, "pitch": 0.0, "yaw": 0.0,
    "width": 800, "height": 600, "fov": 100},
  {"type": "sensor.camera.rgb", "id": "view",
    "x": −4.5, "y": 0.0, "z": 2.8, "roll": 0.0, "pitch": −20.0, "yaw": 0.0,
    "width": 800, "height": 600, "fov": 100},
  {"type": "sensor.lidar.ray_cast", "id": "front",
    "x": 2.0, "y": 0.0, "z": 2.4, "roll": 0.0, "pitch": 0.0, "yaw": 0.0,
    "range": 25000, "channels": 4,
    "points_per_second": 40000,
    "upper_fov": 1.6, "lower_fov": −1.6,
    "rotation_frequency": 50, "sensor_tick": 0.02},
  {"type": "sensor.lidar.ray_cast", "id": "left",
    "x": 2.0, "y": −1.3, "z": 2.4, "roll": 0.0, "pitch": 0.0, "yaw": 0.0,
    "range": 5000, "channels": 1,
    "points_per_second": 27000,
    "upper_fov": 0, "lower_fov": 0,
    "rotation_frequency": 50, "sensor_tick": 0.02},
  {"type": "sensor.lidar.ray_cast", "id": "right",
    "x": 2.0, "y": 1.3, "z": 2.4, "roll": 0.0, "pitch": 0.0, "yaw": 0.0,
    "range": 5000, "channels": 1,
    "points_per_second": 27000,
    "upper_fov": 0, "lower_fov": 0,
    "rotation_frequency": 50, "sensor_tick": 0.02}]
}
```
Listing 4.3: Sensor Setup .JSON Third Configuration.



Size (Pixels)	3
Alpha	1
Decay Time	0
Position Transfor...	XYZ
Color Transformer	FlatColor
Queue Size	10
Color	▨ 115; 210; 22
LIDAR_Right_O	✓
✓ Status: Ok	
Topic	/carla/ego_vehicle/lida...
Unreliable	☐
Selectable	✓
Style	Points
Size (Pixels)	3
Alpha	1
Decay Time	0
Position Transfor...	XYZ
Color Transformer	FlatColor
Queue Size	10
Color	☐ 255; 255; 255
LIDAR_Right_F	☐
Axes	✓
✓ Status: Ok	
Reference Frame	ego_vehicle
Length	1
Radius	0,1
Camera	☐
Status: Warn	
Visibility	✓
Image Topic	/carla/ego_vehicle/ca...
Transport Hint	raw
Queue Size	2
Unreliable	☐
Image Rendering	background and overlay
Overlay Alpha	0,5
Zoom Factor	1

Color
Color to assign to every point.

Figure 4.9: RVIZ results from third configuration with 2 cameras (front + view) and 3 LIDAR sensors (front + left + right).

4.6 Configuring the Field of View of the LIDAR Sensors

Each LIDAR sensor present in CARLA delivers a point cloud with a 360° horizontal FOV which does not match up with the setup of the ATLASCAR2 since each LIDAR sensor in this setup has a different angle setup for the horizontal and vertical fields of view. It is also important to note that CARLA does not yet have a sensor attribute to fix both fields of view according to a given angle. The field of view attribute of the LIDAR sensor correlates to a vertical field of view (upper field + lower field) and what we need to change is the horizontal field of view of the LIDAR according to the angle specified by the real LIDAR sensor. Therefore, these pointclouds had to be rearranged in order to match the angle setup of the LIDAR sensors of the ATLASCAR2.
This section describes the development process of the point cloud filtering algorithm as well as describe some of the tools used in order to record and visualize these point clouds as a way to check the results of the algorithm as it was being developed.

4.6.1 Point Cloud Filtering Algorithm

An overview of the point cloud filtering algorithm is presented in figure 4.10.

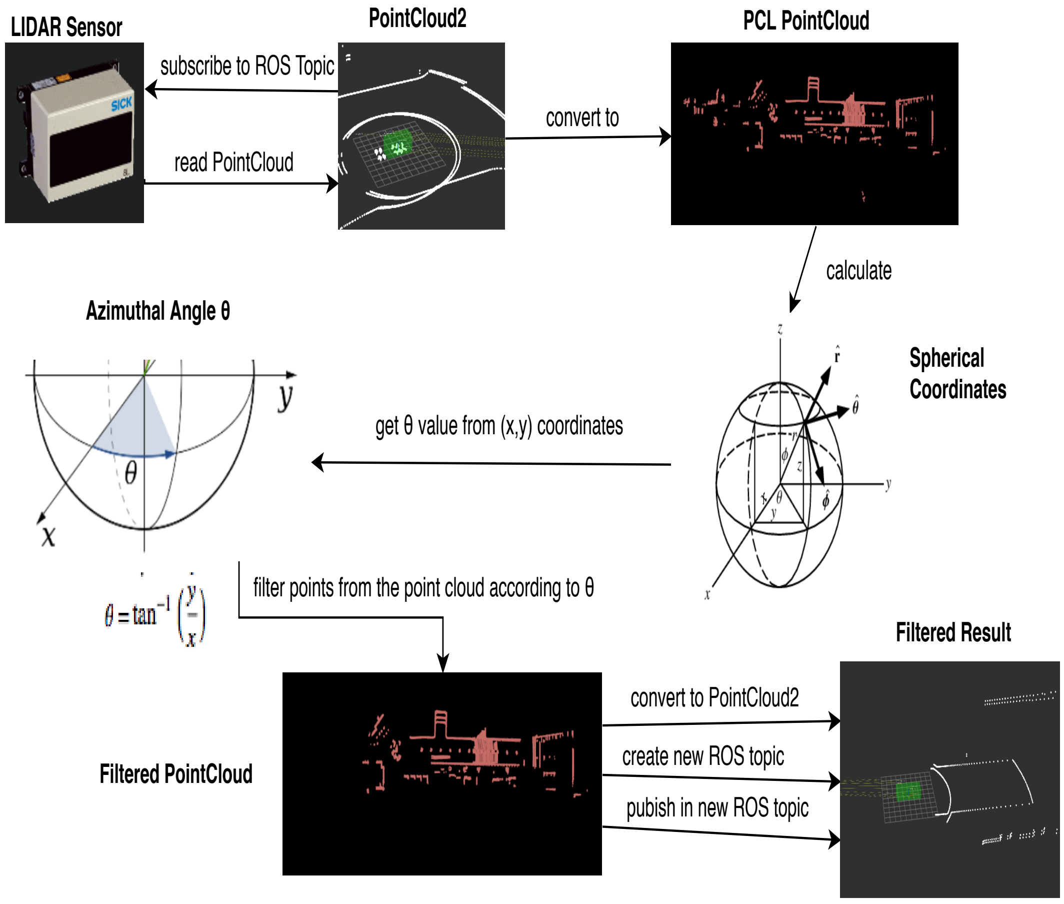


Figure 4.10: Point Cloud Filtering Algorithm.

The development of the point cloud filtering starts by first processing and analyzing the point cloud messages from the ROS topics of the LIDAR sensors. Since ROS does not allow changes in the point cloud's format while it is in the message format, it is necessary to convert this message into a PCL point cloud. Once this is done we can proceed with the filtering process by calculating the spherical coordinates from the cartesian coordinates provided by each point in the point cloud in order to determine the azimuthal angle $\boldsymbol{\theta}$ between the points. This angle has a range between 0 and $2\boldsymbol{\pi}$ and rotates around the z axis as seen in figure 4.11.

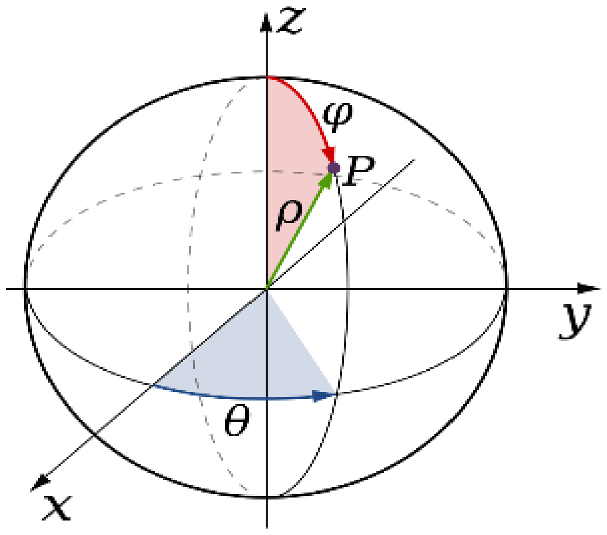


Figure 4.11: Spherical Coordinates System.[49]

The spherical coordinates ($\boldsymbol{\rho}$,$\boldsymbol{\theta}$,$\boldsymbol{\varphi}$) are related to the cartesian coordinates($\boldsymbol{x}$,$\boldsymbol{y}$,$\boldsymbol{z}$) by applying these formulas:

$$\rho = \sqrt{x^2 + y^2 + z^2} \qquad \theta = \tan^{-1}(\frac{y}{x}) \qquad \varphi = \cos^{-1}(\frac{z}{\rho})$$

in which the value we need is the $\boldsymbol{\theta}$ value which is the azimuthal angle that determines the radius for the horizontal field of view.
Having the value of $\boldsymbol{\theta}$ calculated, we can filter out the points in which the $\boldsymbol{\theta}$ value does not fit in the angle range of the LIDAR sensor of the ATLASCAR2 setup. The points which managed to fit in the angle range are then put on another point cloud and this point cloud will showcase the final result of the filtering process.
In this project we considered three angle ranges: one is for the front SICK LD-MRS model sensor and the other two ranges are for the left and right SICK LMS151 model sensors.
The SICK LD-MRS model has an horizontal aperture angle that varies between 85° and 110° depending on the frequency. Since we opted to work on a lower frequency we assumed that the maximum angle would be 90 ° giving an [-$\boldsymbol{\pi}$/4, $\boldsymbol{\pi}$/4] angle range in radians for $\boldsymbol{\theta}$.
The SICK LMS151 model contains a horizontal aperture angle of 270° (depending on the frequency). Since the horizontal angle of the vehicle is equal to 0 and the ATLASCAR2 possesses this sensor model on both right and left sides of this vehicle it was chosen an [-$\boldsymbol{\pi}$, 0] angle range in radians for $\boldsymbol{\theta}$ in the case of the right sensor and it was chosen an [0, $\boldsymbol{\pi}$] angle range in radians for $\boldsymbol{\theta}$ in the case of the left sensor.

A drawing depicting the different angle ranges used in each sensor is presented in figure 4.12, in which the red lines depict the angle range used for the front SICK LD-MRS sensor, the green lines depict the angle range used for the left SICK LMS151 sensor and the yellow lines depict the angle range used for the right SICK LMS151 sensor.

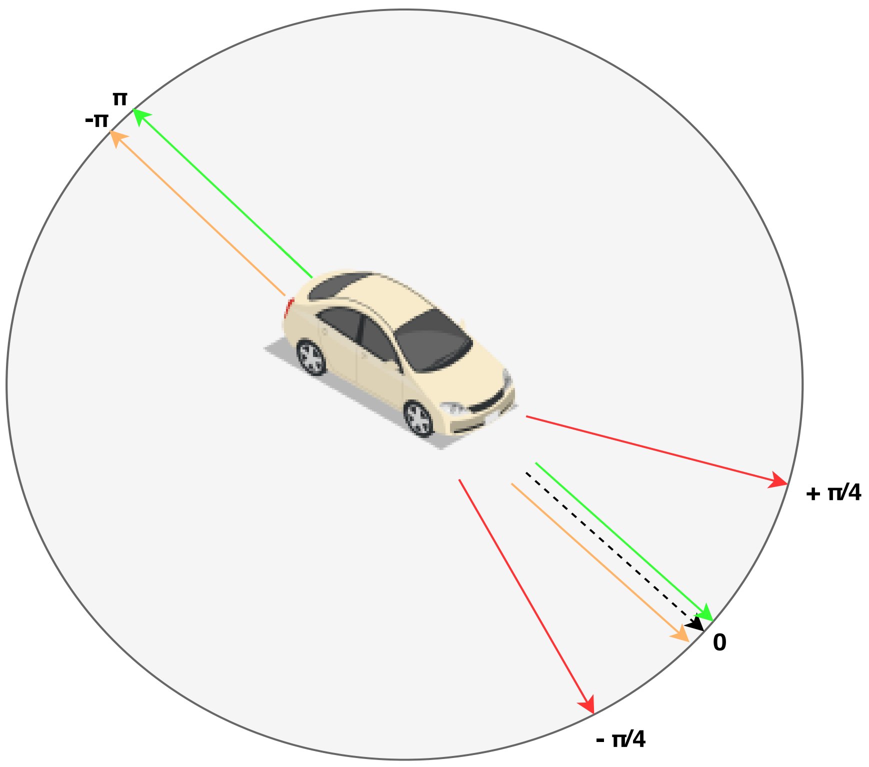
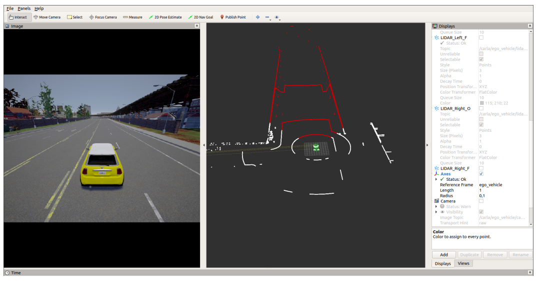


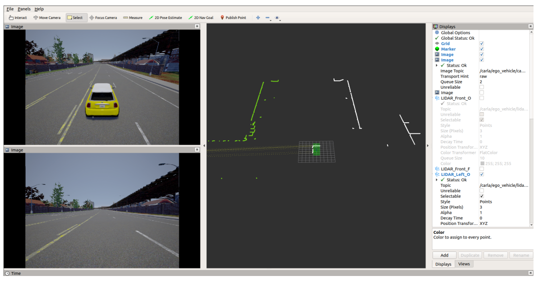
Figure 4.12: Angle $\boldsymbol{\theta}$ range system in radians for the LIDAR sensors of the vehicle module.

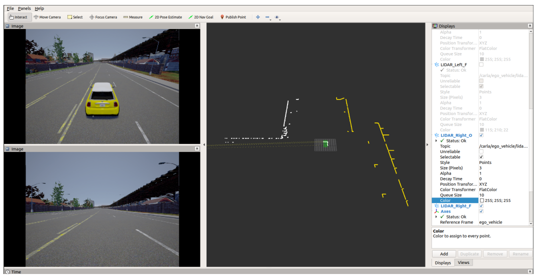
In order to visualize the results in RVIZ, the PCL pointcloud must be converted again to the PointCloud2 format and inserted into a ROS message with a given header to be published in a new ROS topic. The results of the filtering algorithm for each LIDAR visualized in RVIZ are shown in figure 4.13 and the values used for the range of the azimuthal angle $\boldsymbol{\theta}$ in radians for each sensor are depicted in table 4.3.

Sensor Point Cloud	Azimuthal Angle $\boldsymbol{\theta}$ Range Values in Radians
Front LIDAR (Red)	$-\pi/4 \leq \boldsymbol{\theta} \leq \pi/4$
Left LIDAR (Green)	$0 \leq \boldsymbol{\theta} \leq \pi$
Right LIDAR (Yellow)	$-\pi \leq \boldsymbol{\theta} \leq 0$

Table 4.3: Azimuthal Angle $\boldsymbol{\theta}$ Values used in the Point Cloud Filtering Process.


(a) Front Filter Results (Original - White; Filtered - Red)


(b) Left Filter Results (Original - White; Filtered - Green)


(c) Right Filter Results (Original - White; Filtered - Yellow)

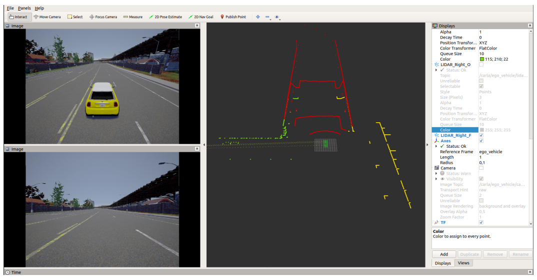

(d) Front + Left + Right Filter (Front - Red; Left - Green; Right - Yellow)

Figure 4.13: CARLA-RVIZ Filtering Results Visualization.

The results shown in RVIZ proved to be reasonable in showcasing the end results of the point cloud filtering process. However, we could not prove that this method was a 100% accurate since there was no guarantee that there were no "necessary" points that were being left out of the point cloud when we applied the filtering. Even in RVIZ, depending on the frame rate, it was very difficult to distinguish if certain points should be on the filtered point cloud or not, according to the vehicle's position and orientation, and if so should the angle range be altered or not.
In order to help solve this problem, we need to compare each point cloud in each frame and see if the combination of the filtered results match those of the original and for that we used two packages in which one is used to record an history log with all the point clouds from the sensors and save them in folders and the other package is used to visualize the point clouds in order to check if the results match or not.

4.6.2 PCL Recording Package

This package corresponds to a ROS node used to record the point clouds from each one of the LIDAR sensors and save these clouds in .PCD format in a certain folder. The purpose of this package is to make an history log with all the point clouds from each one of the LIDAR sensors in a certain session and this is done by first converting the PointCloud2 message from the LIDAR sensor to a PCL point cloud based on the transform from the frame ID, then using a PCDWriter to write the point cloud result in a .PCD file and finally saving the results in a pointcloud directory.
The way this package works is described in figure 4.14.

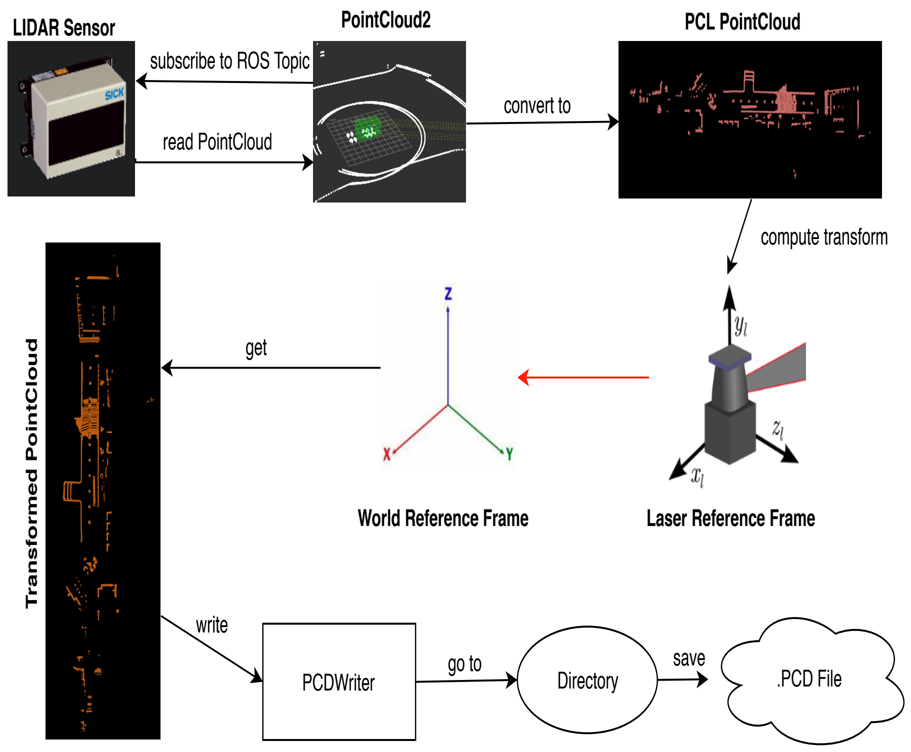


Figure 4.14: Point Cloud Recording Process.

This package will record the point cloud results from each LIDAR sensor according to the timestamp present in the sensor tick of that sensor and these point clouds are put in the world frame of reference in order to have a common reference in each one of the point clouds. With this package it is possible to record each cloud in a file and later see if we can spot major differences in these point clouds.

4.6.3 PCL Visualizing Package

This package corresponds to a ROS node used to visualize the map result from each one of the point clouds from the LIDAR sensors that are saved in the pointcloud directory using a PCL viewer. The maps were used in order to visualize the point clouds in its entirety and see if the cloud was missing any points or simply to see if each cloud from each sensor had

points that should not have been included in the pointcloud.
This node executes 3 shell scripts (one for each sensor) that concatenates the points from each one of the point cloud files present in the directory using the `pcl_concatenate_points_pcd` command to make sure that the number of points in each cloud stays the same between sensor readings; the `pcl_voxel_grid` command to filter the duplicated results according to a given leaf size and reduce the overall size of the pointcloud making the cloud easier to visualize; and finally show the map result in a PCDViewer window using the `pcl_viewer` command. All of these commands are granted by the `pcl_tools` package.[45]
The structure of this shell script file can be seen in listing 4.4 and the map result for each one of the LIDAR sensors can be seen in figure 4.15.

Listing 4.4: PCL visualizer Shell Script File for Front LIDAR PointCloud Directory

```
#!/usr/bin/env bash
# Move to front LIDAR cloud directory
echo "Moving to front LIDAR cloud directory"
echo " "
cd /home/pedro/catkin_ws/src/ros_bridge/pointclouds/front/
sleep 5
echo " "
# Concatenate point cloud files
echo "Concatenating point cloud files"
echo " "
pcl_concatenate_points_pcd *.pcd
sleep 5
echo " "
# Filter duplicate results
echo "Filtering duplicated results"
echo " "
pcl_voxel_grid -leaf 0.1,0.1,0.1 output.pcd map.pcd
sleep 5
echo " "
# Show result
echo "Showing Point Cloud Map Result for Front Lidar Sensor"
echo " "
pcl_viewer map.pcd
sleep 5
echo " "
```

By using this script file it is possible to check and see if there are any points being left out when we apply the filtering process between sensor readings and this way it is possible to check if the algorithm is working correctly by comparing the original map result with the filtered parts obtained from the filtering algorithm.
On a side note, it is possible to see the vehicle course trajectory accross the map during the recording, by checking the areas of the map in which you can see greater levels of point density and show the map results in a PCDViewer as seen in figure 4.16. This, however, only works with the original point clouds provided by the front LIDAR sensor with a FOV of 360° which gives the points on the ground from the lower planes of this sensor, and this alternative was not fully explored with the filtered point clouds from the filtered LIDAR sensors.

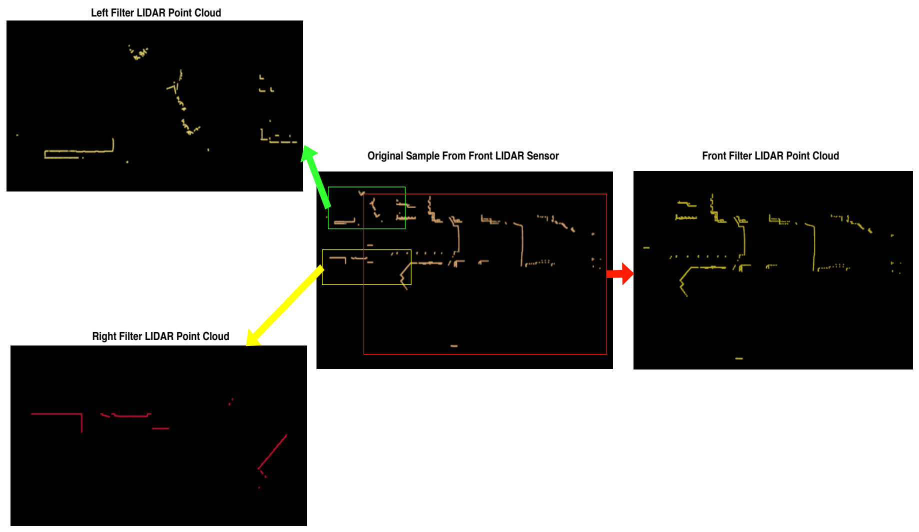

Left Filter LIDAR Point Cloud
Original Sample From Front LIDAR Sensor
Front Filter LIDAR Point Cloud
Right Filter LIDAR Point Cloud

Figure 4.15: Point Cloud Filtering Results visualized in PCDViewer in different scales.

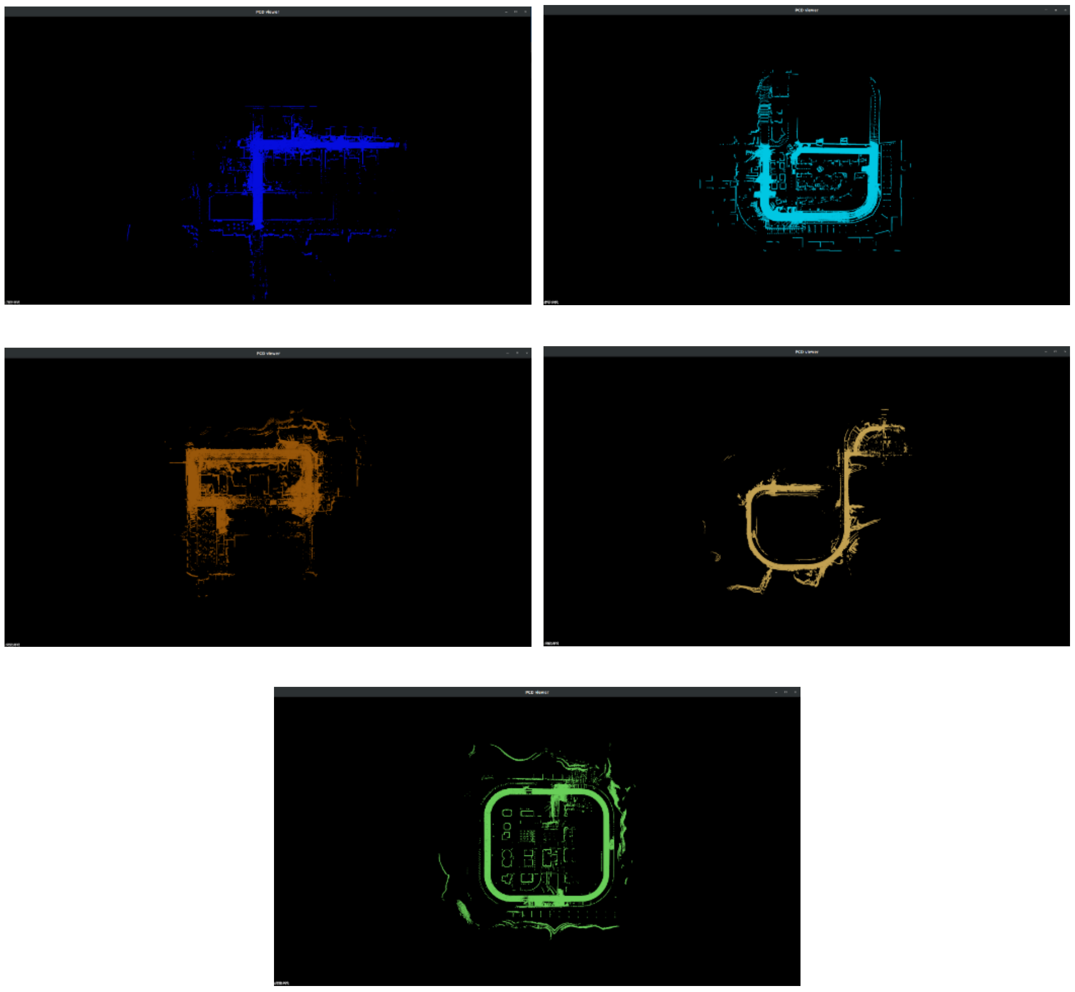


Figure 4.16: Different Point Cloud Town Maps with Vehicle Course Trajectories.

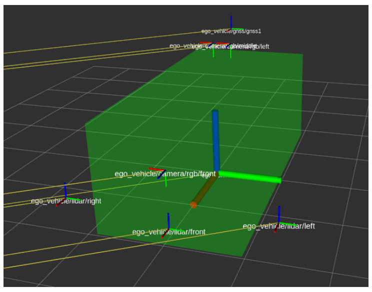
Chapter 5

Experiments with CARLA simulated data

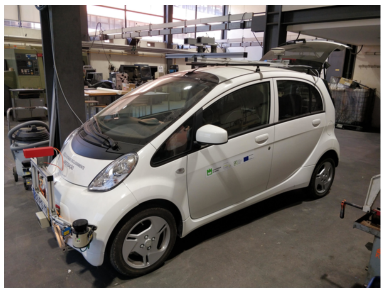
This chapter will be used to illustrate the results of the ATLASCAR sensor setup configuration in CARLA as well as some other results relating to experiments done with CARLA simulated data. These experiments were made in order to validate the use of the CARLA simulator and they include algorithms that can be used to evaluate the performance of algorithms that have been implemented in the ATLASCAR such as semi-automatic detection of target objects, basic object detection and road visual perception.

5.1 Sensor Setup Configuration

The figure 5.1 presents an illustration of the ATLASCAR sensor setup configuration visualized using RVIZ. The localization of each one of the sensors must be in correlation to the moving axis of the vehicle center and their position should mirror the real deal as much as possible. However this isn't truly possible, because of the scaling method used in CARLA and for that reason the x, y and z coordinates have to be rearranged in order to fit the vehicle model and produce results that mirror those of the ATLASCAR sensors. The other sensor attributes must be defined based on the sensor modules used in the ATLASCAR and calibration results used on the ATLASCAR as it was the case for the cameras.



(a) ATLASCAR2 sensor setup model

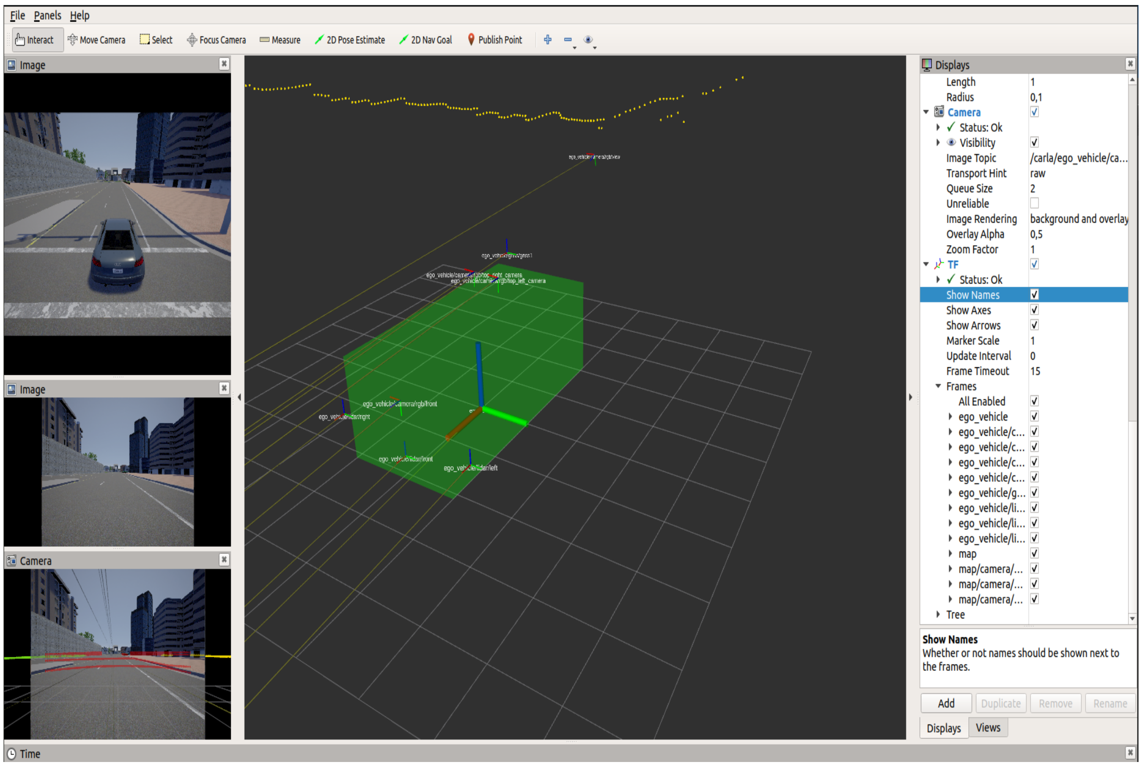


(b) ATLASCAR2 real setup model

Figure 5.1: ATLASCAR Sensor Setup Configuration Result with Transformation Frames.

In listing 5.1 a snippet file with the final sensor configuration is presented. This file corresponds to a JSON file that contains every sensor with its specified attributes for the vehicle that is to be spawned in CARLA. Each sensor present in the file will publish its own transformation matrix relatively to the `map` transformation reference which is aligned with the vehicle center using a frame publisher. The camera calibration parameters presented in the top right and left cameras are values from an extrinsic calibration that was used in the real ATLASCAR cameras. The results from this sensor configuration are displayed in figure 5.2 and the transformation tree for the vehicle sensor configuration is presented in the following page.

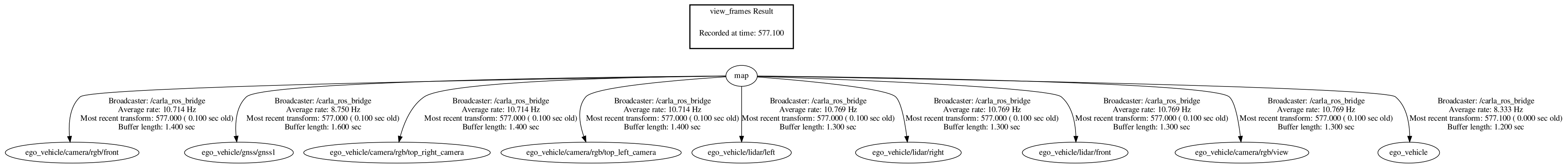
```json
{
  "sensors": [
    {"type": "sensor.camera.rgb",
      "id": "front",
      "x": 2.0, "y": 0.0, "z": 1.0, "roll": 0.0, "pitch": 0.0, "yaw": 0.0,
      "width": 800,
      "height": 600,
      "fov": 100},
    {"type": "sensor.camera.rgb",
      "id": "top_right_camera",
      "x": 0.0, "y": 0.0, "z": 2.0, "roll": 3.08517033374988, "pitch":
    0.0114432071771162, "yaw": -1.58683126505267,
      "width": 964,
      "height": 724,
      "fov": 100},
    {"type": "sensor.camera.rgb",
      "id": "top_left_camera",
      "x": 0.0, "y": -0.5, "z": 2.0, "roll": 3.09315055757511, "pitch":
    0.0108160149890114, "yaw": -1.60435715276337,
      "width": 964,
      "height": 724,
      "fov": 100},
    {"type": "sensor.camera.rgb",
      "id": "view",
      "x": -4.5, "y": 0.0, "z": 2.8, "roll": 0.0, "pitch": -20.0, "yaw": 0.0,
      "width": 800,
      "height": 600,
      "fov": 100},
    {"type": "sensor.lidar.ray_cast",
      "id": "front",
      "x": 2.0, "y": 0.0, "z": 0.25, "roll": 0.0, "pitch": 0.0, "yaw": 0.0,
      "range": 25000,
      "channels": 4,
      "points_per_second": 40000,
      "upper_fov": 1.6,
      "lower_fov": -1.6,
      "rotation_frequency": 20,
      "sensor_tick": 0.02},
    {"type": "sensor.lidar.ray_cast",
      "id": "left",
      "x": 2.0, "y": -1.3, "z": 0.5, "roll": 0.0, "pitch": 0.0, "yaw": 0.0,
      "range": 5000,
      "channels": 1,
      "points_per_second": 27000,
      "upper_fov": 0,
      "lower_fov": 0,
      "rotation_frequency": 25,
      "sensor_tick": 0.02},
    {"type": "sensor.lidar.ray_cast",
      "id": "right",
      "x": 2.0, "y": 1.3, "z": 0.5, "roll": 0.0, "pitch": 0.0, "yaw": 0.0,
      "range": 5000,
      "channels": 1,
      "points_per_second": 27000,
      "upper_fov": 0,
      "lower_fov": 0,
      "rotation_frequency": 25,
      "sensor_tick": 0.02},
    {"type": "sensor.other.gnss",
      "id": "gnss1",
      "x": −1.0, "y": 0.0, "z": 2.0},
    {"type": "sensor.other.collision",
      "id": "collision1",
      "x": 0.0, "y": 0.0, "z": 0.0},
    {"type": "sensor.other.lane_invasion",
      "id": "laneinvasion1",
      "x": 0.0, "y": 0.0, "z": 0.0}
  ]
}
```
Listing 5.1: Snippet file with .JSON Final Sensor Setup Configuration.



Length	1
Radius	0,1
Camera	✓
Status: Ok	
Visibility	✓
Image Topic	/carla/ego_vehicle/ca...
Transport Hint	raw
Queue Size	2
Unreliable	☐
Image Rendering	background and overlay
Overlay Alpha	0,5
Zoom Factor	1
TF	✓
Status: Ok	
Show Names	✓
Show Axes	✓
Show Arrows	✓
Marker Scale	1
Update Interval	0
Frame Timeout	15
Frames	
All Enabled	✓
ego_vehicle	✓
ego_vehicle/c...	✓
ego_vehicle/c...	✓
ego_vehicle/c...	✓
ego_vehicle/c...	✓
ego_vehicle/g...	✓
ego_vehicle/li...	✓
ego_vehicle/li...	✓
ego_vehicle/li...	✓
map	✓
map/camera/...	✓
map/camera/...	✓
map/camera/...	✓
Tree	

Show Names
Whether or not names should be shown next to the frames.

Figure 5.2: CARLA Sensor Configuration shown in RVIZ with Camera and LIDAR results.

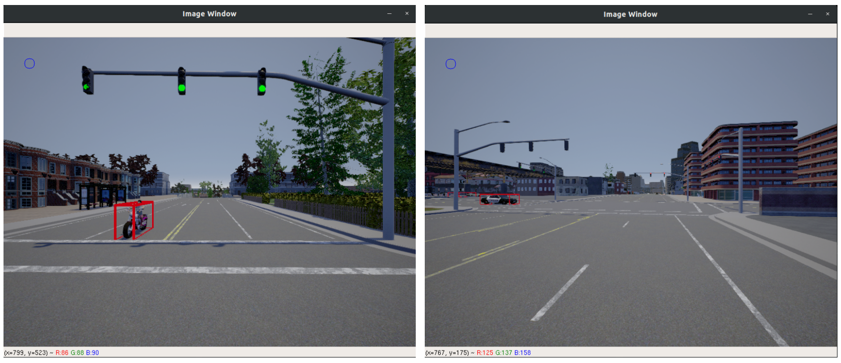

view_frames Result
Recorded at time: 577.100
map
Broadcaster: /carla_ros_bridge
Average rate: 10.714 Hz
Most recent transform: 577.000 (0.100 sec old)
Buffer length: 1.400 sec
ego_vehicle/camera/rgb/front
Broadcaster: /carla_ros_bridge
Average rate: 8.750 Hz
Most recent transform: 577.000 (0.100 sec old)
Buffer length: 1.600 sec
ego_vehicle/gnss/gnss1
Broadcaster: /carla_ros_bridge
Average rate: 10.714 Hz
Most recent transform: 577.000 (0.100 sec old)
Buffer length: 1.400 sec
ego_vehicle/camera/rgb/top_right_camera
Broadcaster: /carla_ros_bridge
Average rate: 10.714 Hz
Most recent transform: 577.000 (0.100 sec old)
Buffer length: 1.400 sec
ego_vehicle/camera/rgb/top_left_camera
Broadcaster: /carla_ros_bridge
Average rate: 10.769 Hz
Most recent transform: 577.000 (0.100 sec old)
Buffer length: 1.300 sec
ego_vehicle/lidar/left
Broadcaster: /carla_ros_bridge
Average rate: 10.769 Hz
Most recent transform: 577.000 (0.100 sec old)
Buffer length: 1.300 sec
ego_vehicle/lidar/right
Broadcaster: /carla_ros_bridge
Average rate: 10.769 Hz
Most recent transform: 577.000 (0.100 sec old)
Buffer length: 1.300 sec
ego_vehicle/lidar/front
Broadcaster: /carla_ros_bridge
Average rate: 10.769 Hz
Most recent transform: 577.000 (0.100 sec old)
Buffer length: 1.300 sec
ego_vehicle/camera/rgb/view
Broadcaster: /carla_ros_bridge
Average rate: 8.333 Hz
Most recent transform: 577.100 (0.000 sec old)
Buffer length: 1.200 sec
ego_vehicle

5.2 Simple Examples Using Carla Simulated Data

This section will be used to present some examples of uses of CARLA simulated data that can be used for object detection and visual perception algorithms.

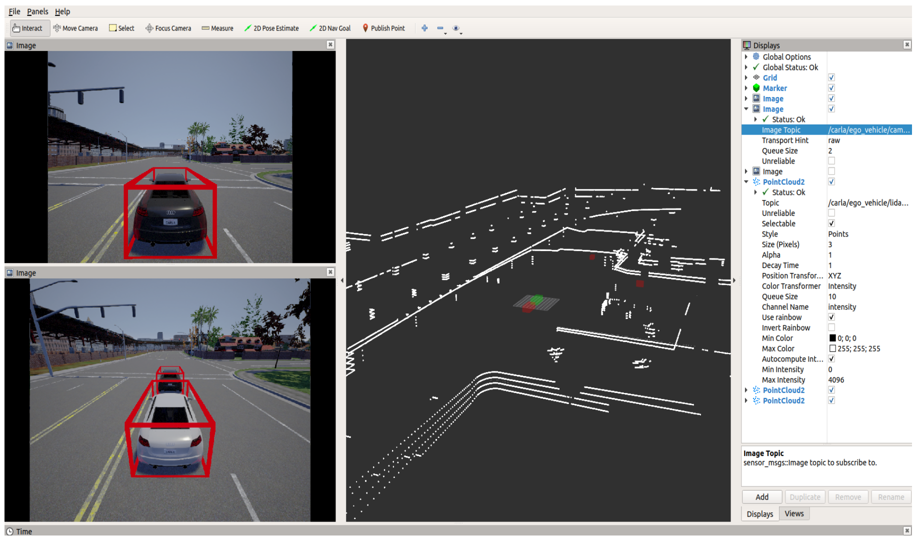
5.2.1 Bounding Box Activation for Dataset Creation

In order to perform object detection, tracking and labelling of objects present in the CARLA world, a ROS node was created which its function is to activate the bounding boxes present in each object model of CARLA.
These bounding boxes are designed using functions present in the CARLA module and these bounding boxes will keep up with the vehicle object according to the present frame.
Some of these bounding boxes can be seen in figure 5.3 and the results can be seen in figure 5.4.



```
(x=799, y=523) ~ R:86 G:88 B:90
(x=767, y=175) ~ R:125 G:137 B:158
```

Figure 5.3: CARLA Bounding Boxes.



```
Displays
▸ Global Options
▸ ✓ Global Status: Ok
▸ Grid                    ✓
▸ Marker                  ✓
▸ Image                   ✓
▾ Image                   ✓
   ▸ ✓ Status: Ok
   Image Topic            /carla/ego_vehicle/cam...
   Transport Hint         raw
   Queue Size             2
   Unreliable             ☐
▸ Image                   ☐
▾ PointCloud2             ✓
   ▸ ✓ Status: Ok
   Topic                  /carla/ego_vehicle/lida...
   Unreliable             ☐
   Selectable             ✓
   Style                  Points
   Size (Pixels)          3
   Alpha                  1
   Decay Time             1
   Position Transfor...   XYZ
   Color Transformer      Intensity
   Queue Size             10
   Channel Name           intensity
   Use rainbow            ✓
   Invert Rainbow         ☐
   Min Color              ■ 0; 0; 0
   Max Color              □ 255; 255; 255
   Autocompute Int...     ✓
   Min Intensity          0
   Max Intensity          4096
▸ PointCloud2             ✓
▸ PointCloud2             ✓

Image Topic
sensor_msgs::Image topic to subscribe to.
```

Figure 5.4: CARLA Bounding Boxes Results shown in RVIZ.

Bounding Box to File

Based on the information from these bounding boxes, it is possible to gather basic information relating to the objects surrounding the vehicle such as their position and rotation in relation to the world's coordinate system, their blueprint ID as well as the extent of their bounding box. The first two parameters can be accessed by checking the location and the rotation attributes of the vehicle transform which outputs the (x,y,z) coordinates and the (yaw) rotation of the vehicle. The extent of the bounding box give us a set of coordinates (bx,by,bz) which can be used to calculate the volume of the bounding box and with that volume we can separate the objects into four labelling classes: bicycle, motorcycle, car and truck. The vehicle model has an attribute which allows us to check the number of wheels it has and based on that information we can either compare the volume of the bounding box of the vehicle model with the average volume of a bicycle model in CARLA which is around $0.18cm^3$ if the vehicle is a 2-wheeled model or compare the volume of the bounding box with the average volume of a car model in CARLA which is around $2.17cm^3$. The use of labels in relation to the volume of the bounding box is shown in the following tables.
All of this information is dumped in a .JSON dataset whose format can be seen in listing 5.2.

Bounding Box Volume	Label
$\leq 0.18\text{cm}^3$	Bicycle
$> 0.18\text{cm}^3$	Motorcycle

Table 5.1: 2-Wheeled Vehicles.

Bounding Box Volume	Label
$\leq 2.17\text{cm}^3$	Car
$> 2.17\text{cm}^3$	Truck

Table 5.2: 4-Wheeled Vehicles.

Table 5.3: Bounding Box Labelling Process.

```
{
  "vehicles": [
  {"yaw": 178.757781982, "y": 58.7389602661, "x": 219.094940186, "model": '
    vehicle.nissan.patrol', "z": −0.0561580248177, "bx": 2.68459653854, "class":
     'car', "bz": 0.707549750805, "by": 0.916354358196}, {"yaw": 89.6200866699,
    "y": 75.060508728, "x": −9.86885166168, "model": 'vehicle.bmw.isetta', "z":
    0.0423052981496, "bx": 1.90109634399, "class": 'car', "bz": 0.737500309944,
    "by": 0.939999997616},
  {"yaw": 91.389465332, "y": 80.9183654785, "x": 233.150924683, "model": '
    vehicle.ninja.kawasaki', "z": 0.0724878311157, "bx": 0.921720683575, "class"
    : 'motorcycle', "bz": 0.795036911964, "by": 0.249023392797},
  {"yaw": −0.150054946542, "y": 203.937469482, "x": 38.2016067505, "model": '
    vehicle.ninja.kawasaki', "z": 0.0507690794766, "bx": 0.754661381245, "class"
    : 'motorcycle', "bz": 0.76531547308, "by": 0.432970315218},
  {"yaw": −87.9634170532, "y": −120.209236145, "x": 84.7773208618, "model": '
    vehicle.audi.tt', "z": 7.98664188385, "bx": 1.83442378044, "class": 'car', "
    bz": 0.761606872082, "by": 0.832001924515}
  ...
  ]
}
```

Listing 5.2: CARLA Bounding Box Dataset Format.

This format was chosen in order to mirror the ones used in the Karlsruhe Institute of Technology and Toyota Technological Institute (KITTI)[50] and Berkeley DeepDrive[51] datasets. Using this approach, it is possible to get a level of ground-truth to the object detection, tracking and labelling processes that can be used to rate the overall performance of other algorithms that are implemented with the same objective such as the Semi-Automatic Automatic Labelling and Object Tracking algorithm developed by Nuno Silva in 2018.[52]

5.2.2 Template Matching Algorithm with Templates from CARLA

Another way to perform object detection and tracking of objects present in the CARLA world, is using another ROS node that was created which its function is to implement the Template Matching algorithm using template objects from the CARLA world to design bounding boxes in each object model of CARLA.
The template matching algorithm[44] is a technique used in computer vision for finding areas in an image that match (are similar) to a template image (patch image). The resulting image is calculated by iterating the source image and comparing the template image with the area in that position in the image and for each position a score is assigned representing how good the match is in that position in the image. If this score passes a given threshold, the algorithm will design a 2D bounding box around the object identifying the template object in the image. Other computer vision algorithms were tested such as Optical Flow but the results provided by these algorithms were not good for the problem at hand. The Optical Flow method failed because this method is meant to be used when the camera is static whereas in this work the camera is constantly moving along with the vehicle sometimes assuming a different position in each frame taken from the camera.

Templates from CARLA simulation

The template objects used in the Template Matching algorithm are template objects taken from the CARLA world. These templates are cropped from the image sequences received from CARLA using OpenCV mouse events to cropp the template image from the source image leaving the cropping form representing the region of interest of the template object designed in the source image. The cropping process of the template object can be seen in figure 5.5.

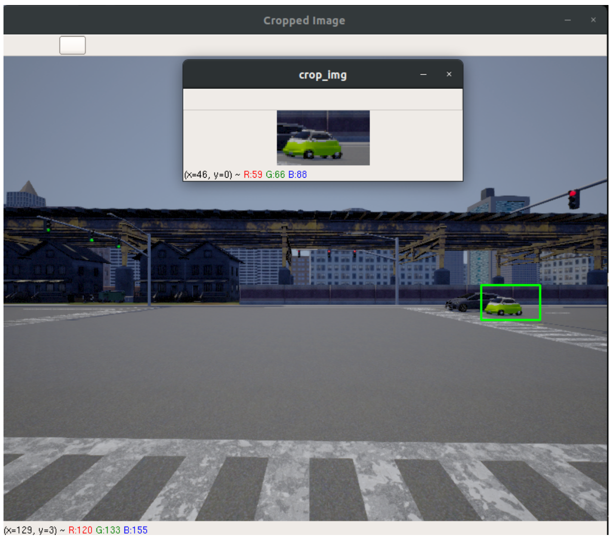

Cropped Image
crop_img
(x=46, y=0) ~ R:59 G:66 B:88
(x=129, y=3) ~ R:120 G:133 B:155

Figure 5.5: Template object cropped out of the source image.

Template Matching in CARLA with 2D Bounding Boxes

Once the template objects are cropped out from the images, we can proceed with the implementation of the Template Matching algorithm using the OpenCV's `matchTemplate` function using the cropped image as the template and the image sequences received from CARLA as the source images drawing a 2D bounding box around the object detected from the Template Matching. The results can be seen in figure 5.6.

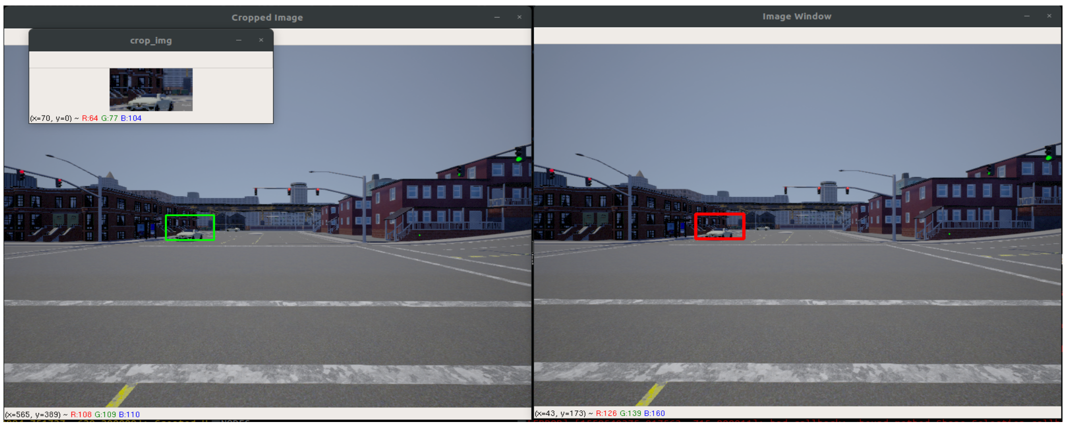
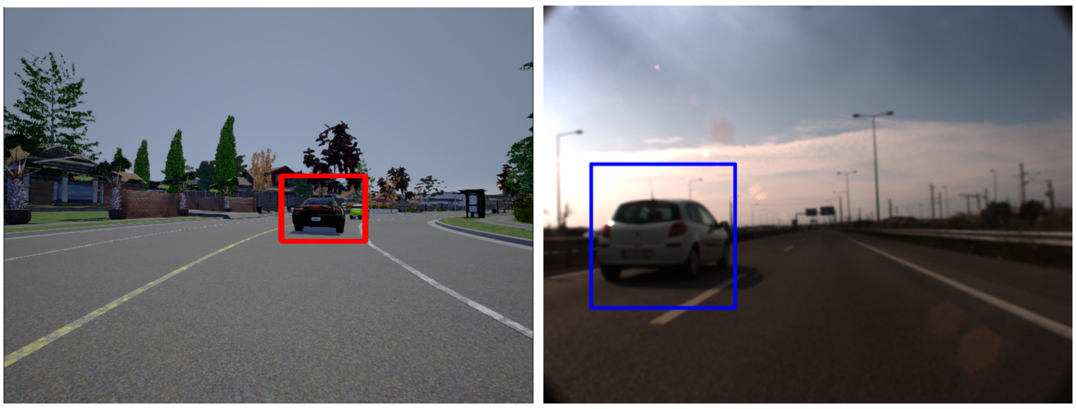

Cropped Image
crop_img
(x=70, y=0) ~ R:64 G:77 B:104
(x=565, y=389) ~ R:108 G:109 B:110
Image Window
(x=43, y=173) ~ R:126 G:139 B:160

Figure 5.6: Template Matching with 2D Bounding Boxes.



(a) Template Matching With Frames From CARLA

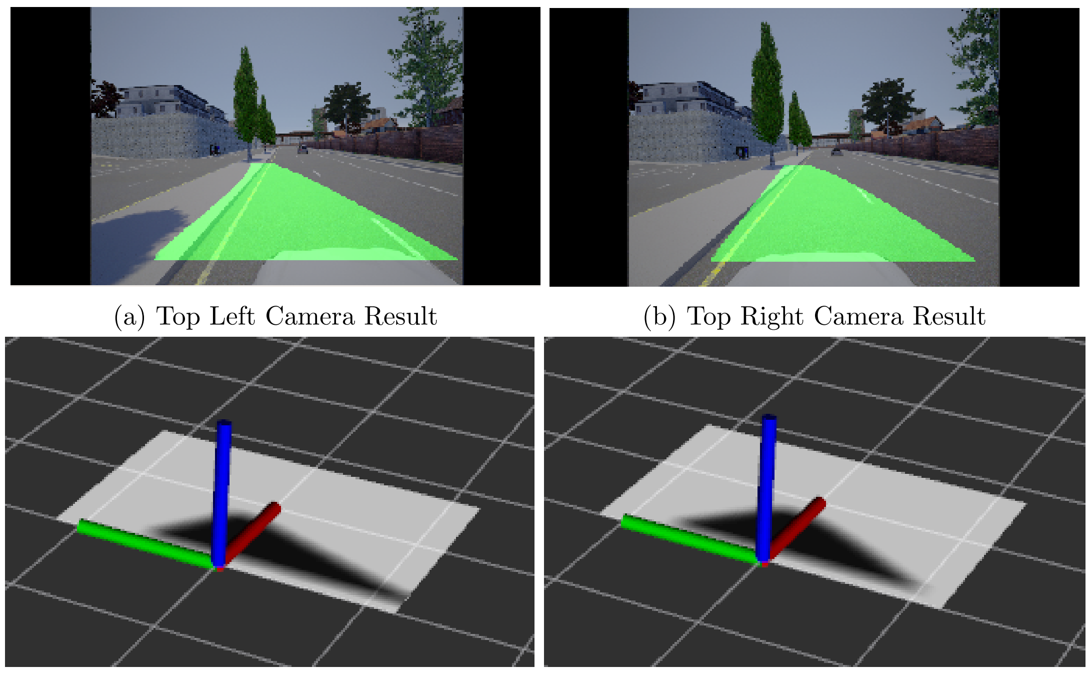
(b) Template Matching With Frames From ATLASCAR2

Figure 5.7: Template Matching Comparison.

By looking at figure 5.7, it is possible to conclude that the Template Matching algorithm works well in frame scenarios provided by CARLA and frame scenarios provided by the ATLASCAR2 and one can argue that this algorithm performs better in CARLA maybe due to the rendering quality provided by the cameras in CARLA or simply for the fact that with this tool it is possible to control the size of the bounding box without the need of calculating the distance to the object which is helpful in tracking smaller sized objects in an image frame. However, this still presents some problems since this process requires a steady hand and a lot of patience in order to create the bounding boxes but these problems can later on be solved by evolving the frame-by-frame selection to an automatic process.

5.2.3 Road Visual Perception Algorithm

Using data simulated by the CARLA simulator it is possible to replicate the ATLASCAR setup as well as illustrate the possibility of applying algorithms that are being developed in the ATLASCAR with simulated data provided by CARLA.
The Road Visual Perception Algorithm implemented by Tiago Almeida[36] is one of the algorithms that is currently being tested in the ATLASCAR and this algorithm uses the top cameras present in the vehicle to perform lane detection on the road based on information provided by those two cameras. This algorithm uses a combination of two other algorithms: a Lane Detection algorithm provided by Nsteel[53] that uses basic computer vision algorithms to check the road markings and detect the lane and another one which is the Advanced Lane Detection algorithm provided by HsucheChiang[53] which uses advanced computer vision algorithms mixed in with deep learning in order to detect the road markings and detect the lane.
The results of this algorithm with camera simulated data from CARLA are presented in figure 5.8.



(a) Top Left Camera Result

(b) Top Right Camera Result

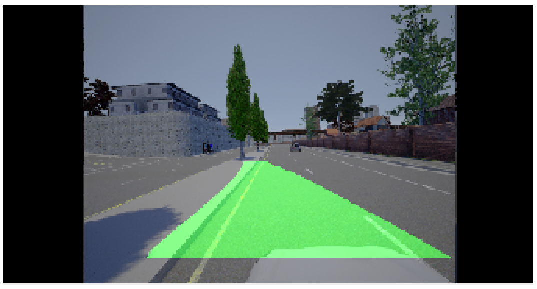
(c) Top Left Probablistic Map

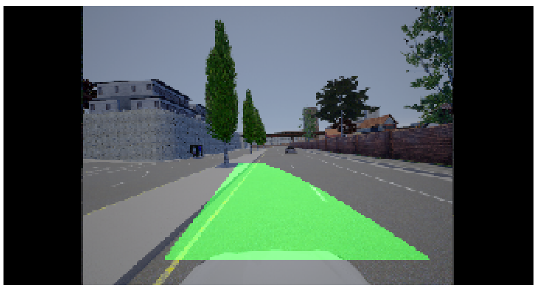
(d) Top Right Probablistic Map

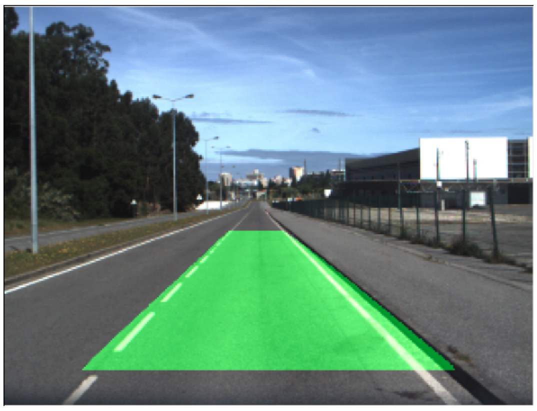
Figure 5.8: Road Visual Perception Algorithm Results using CARLA simulated data.

The probablistic maps are used show the probability of having road markings present on the road and these are used to establish the limits of the lane and based on those limits construct a map over said lane.

The results of the algorithm with simulated data in comparison with the results of the algorithm with data provided by the ATLASCAR can be seen in figure 5.9.


(a) Top Left Camera Result - CARLA


(b) Top Right Camera Result - CARLA


(c) Top Left Camera Result - ATLASCAR

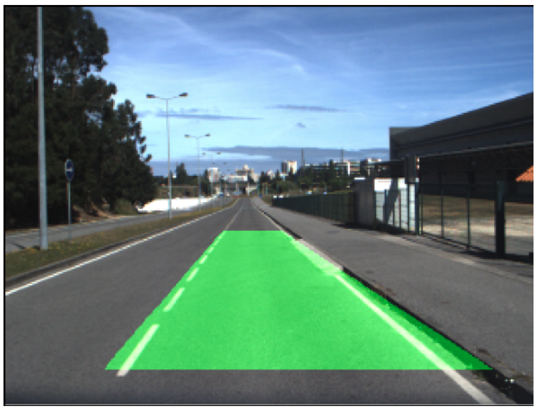

(d) Top Right Camera Result - ATLASCAR

Figure 5.9: Road Visual Perception Algorithm Results with CARLA simulated data and data from the ATLASCAR2.

It is important to note that this algorithm was applied using a deep learning algorithm to perform the road lane detection and it learned that data from real scenarios provided by the ATLASCAR2. Taking this into consideration, it is possible to understand why some of the results with simulated data can fall short when compared with real scenarios. Other factors such as the rendering of the road lines can also be a factor for the overall performance of the algorithm. However, these results are still valid for measuring the performance and accuracy of this algorithm in simulated environments and these results are also valid for comparison with results from the real world.

Chapter 6

Conclusions and Future Work

This chapter serves as a closing point to the work done during the course of this dissertation regarding the adaptation of the CARLA simulator as a tool that can be used to perform data acquisition and can be used to test out the ATLASCAR2 algorithms in CARLA.
This chapter is also used to elaborate upon some future works that can be proposed based of the work accomplished in this dissertation as long as it maintains in the scope of the ATLASCAR2 project and the paradigms of AD and ADAS.

6.1 Conclusions

The research in the fields of AD and ADAS is proven to be more and more important for car manufactures. Building perception of the environment surrounding the vehicle and the control of the different environmental conditions surrounding the vehicle are some of the key factors used for creating systems such as anti collision, path planning, etc...
For this reason, many research teams in autonomous driving chose to test their algorithms using an autonomous driving simulator in which the conditions surrounding the vehicle can be controlled. These algorithms are implemented first in a simulator to test out their performance and behavior in different environmental conditions before implementing them in the real world where there are scenarios that are simply out of our control.
CARLA is one of many autonomous driving simulators used for the purpose of training autonomous vehicles and testing out the algorithms that are going to be used for autonomous driving and it provides some state-of-the-art rendering scenarios which can now be used in the context of the ATLASCAR project by replicating the sensor setup configuration of the vehicle and simulate it using CARLA.
The sensor setup configuration part was achieved by creating a .JSON configuration with all the sensors present in the vehicle detailing each one of the attributes of each sensor and with this file it is possible parameterize the number of sensors present in the vehicle as well as their working conditions. The field of view of the LIDAR sensors needed to be configured in order to suit the LIDAR sensors of the ATLASCAR and for that reason a tool for filtering out points from the original point clouds from the LIDAR sensors was developed and these results were tested using PCL point cloud recording and visualization packages.
With this configuration, it is possible to use CARLA as a way to test out the algorithms that are going to be implemented in the ATLASCAR by first providing some use cases of some algorithms used for object detection, tracking and labelling as well as visual perception

and providing some tools that can be used as ground truth for validation of some of these algorithms which is something that the ATLASCAR did not have until this point. The use cases were depicted from experiments with CARLA simulated data such as the one with the Template Matching algorithm and the Road Visual Perception algorithm and the tools that can be used as ground truth were depicted in the experiment with the bounding boxes provided by CARLA which can be used to create datasets and in turn these datasets can be used for validation of other datasets that we wish to create using some other method be it semi-automatic labelling or plain-old object tracking.
The main objective of this dissertation was to replicate the setup of the ATLASCAR2, replicate its scenarios using a simulated environment and check if the results were valid enough to perform data acquisition for the ATLASCAR2 using this simulator. This objective was achieved by replicating the sensor setup of the ATLASCAR2 in a vehicle model in CARLA and playing this vehicle in the simulation. By analyzing the results of each sensor it is possible to classify these results as valid to perform data acquisition for the ATLASCAR2 using this model.
In the end, the CARLA simulator can now be used as a tool for developing new algorithms and can be used as an external tool for running tests without the need of using the ATLASCAR vehicle. The results of the tests can be used to evaluate the algorithms, test out their performance and see if they can improve.

6.2 Future Work

There are many possibilities of future work for the ATLASCAR2 regarding AD and ADAS based on the implementation of an autonomous driving simulator. Since this simulator can produce data with some level of ground truth this method can be used to evaluate the performance of the algorithms before testing them out on the ATLASCAR2. Some interesting ideas that are worth mentioning would be implementating a module that can perform fusion between the camera and LIDAR sensor data in order to track the objects around the CARLA world and create bounding boxes surrouding those objects and based on that create datasets based on the position and the blueprint ID and see how they can compare to the bounding box results provided by CARLA.
Creating a hardware interface, mirroring the one used for the ATLASCAR2, to be used to control the vehicle in the simulation provided by the autonomous driving simulator would also be a good idea since we can add the driver's behavior input into the equation and see how it behaves in the simulation.
Another interesting idea for a project would be working on a navigation method for the vehicle in which the vehicle follows another object around the world and this can be achieved by implementing a machine learning algorithm based on image templates and labelled image datasets provided by CARLA to create an autonomous driving method that can follow labelled objects around the CARLA world.
The scope of this dissertation was to replicate scenarios of the ATLASCAR2 in order to perform data acquisition to test out the algorithms but it is possible to expand this in numerous ways based on the desired objectives. One of these ways can be using augmented reality in order to develop a detection, tracking and labelled system or using machine learning to improve the object detection process.

Chapter 7

References

[1] *LAR (2019) - lar - laboratory for automation and robotics.* http://lars.mec.ua.pt/, Accessed: 2019-04-11.

[2] *ATLAS Prototypes (2019) - atlas competitions.* http://atlas.web.ua.pt/competitions.html, Accessed: 2019-04-11.

[3] *ATLASCAR 1 (2019) - atlascar research.* http://atlas.web.ua.pt/atlascar.html, Accessed: 2019-04-11.

[4] J. Pereira, *Quadro Elétrico ATLASCAR-2.* Aveiro University: Aveiro University Industrial Automation Engineering, 2017, Accessed: 2019-04-11. [Online]. Available: http://lars.mec.ua.pt/public/LAR%20Projects/SystemDevelopment/2017_JosePereira/PEA_Relatorio_QE_71985.pdf.

[5] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "Carla: An open urban driving simulator", Nov. 2017, Accessed: 2019-05-02. [Online]. Available: http://proceedings.mlr.press/v78/dosovitskiy17a/dosovitskiy17a.pdf.

[6] *CARLA SIMULATOR (2019) cameras and sensors*, https://carla.readthedocs.io/en/latest/cameras_and_sensors/, Accessed: 2019-05-02.

[7] F. Kröger, "Automated driving in its social, historical and cultural contexts", in. May 2016, pp. 41–68, Accessed: 2019-04-11, ISBN: 978-3-662-48845-4. DOI: 10.1007/978-3-662-48847-8_3.

[8] *Milwaukee Sentinel (1926) "phantom auto will tour the city"*, https://news.google.com/newspapers?id=unBQAAAAIBAJ&sjid=QQ8EAAAAIBAJ&pg=7304,3766749, Accessed: 2019-04-11.

[9] D. Pomerleau, "Alvinn: An autonomous land vehicle in a neural network", Accessed: 2019-04-11, vol. 1, Jan. 1988, pp. 305–313. [Online]. Available: http://papers.nips.cc/paper/95-alvinn-an-autonomous-land-vehicle-in-a-neural-network.pdf.

[10] *DARPA (2019) the darpa grand challenge*, https://www.darpa.mil/about-us/timeline/-grand-challenge-for-autonomous-vehicles, Accessed: 2019-04-11.

[11] M. Montemerlo, S. Thrun, H. Dahlkamp, D. Stavens, and S. Strohband, "Winning the darpa grand challenge with an ai robot.", Accessed: 2019-04-11, Jan. 2006. [Online]. Available: http://ai.stanford.edu/~dstavens/aaai06/montemerlo_etal_aaai06.pdf.

[12] *Waymo (2019) - waymo*, https://waymo.com/, Accessed: 2019-05-29.

[13] *Uber Advanced Techonologies Group (2019) - uber atg*, https://www.uber.com/info/atg/, Accessed: 2019-04-15.

[14] *Audi (2018) - audi mediacenter audi a8*, https://www.audi-mediacenter.com/en/press-releases/audi-at-the-iaa-2017-autonomous-driving-in-three-steps-9311, Accessed: 2019-05-29.

[15] A. Herrmann, W. Brenner, and R. Stadler, *Autonomous Driving: How the Driverless Revolution Will Change the World*. Apr. 2018, Accessed: 2019-04-15, ISBN: 978-1-78714-834-5. DOI: 10.1108/9781787148338.

[16] *Singapore-MIT Alliance for Research and Technology (2019) - smart - singapore-mit alliance for research and technology*. https://smart.mit.edu/, Accessed: 2019-04-16.

[17] P. Teo, *SMART launches first Singapore-developed driverless car designed for operations on public roads*. SMART MIT: SMART, 2014, Accessed: 2019-04-11. [Online]. Available: https://smart.mit.edu/images/pdf/news/2014/Driverless_Car_NR_270114_Final.pdf.

[18] *TESLA (2019) a look at tesla's new autopilot hardware suite: 8 cameras, 1 radar, ultrasonics & new supercomputer*, https://electrek.co/2016/10/20/tesla-new-autopilot-hardware-suite-camera-nvidia-tesla-vision/, Accessed: 2019-04-11.

[19] M. Dikmen and C. Burns, "Trust in autonomous vehicles: The case of tesla autopilot and summon", Accessed: 2019-04-11, Oct. 2017. DOI: 10.1109/SMC.2017.8122757.

[20] *TESLA (2019) tesla autopilot*, https://www.tesla.com/en_GB/autopilot?redirect=no, Accessed: 2019-04-11.

[21] *CARLA SIMULATOR (2019) - carla open-source simulator for autonomous driving research*. http://carla.org/, Accessed: 2019-04-25.

[22] *CARLA SIMULATOR (2019) - releases*, carla-0.9.5-http://carla.org/2019/04/03/release-0.9.5/; carla-0.9.1-http://carla.org/2018/11/16/release-0.9.1/, Accessed: 2019-04-25.

[23] *CARLA SIMULATOR (2018) reinforcement learning for autonomous driving in carla*, https://ai4sig.org/2018/08/carla-reinforcement-learning/, Accessed: 2019-05-02.

[24] *Cognata Autonomous Driving Simulator (2018) audi partners with israeli startup cognata for autonomous car development*, http://nocamels.com/2018/06/audi-cognata-autonomous-vehicles/, Accessed: 2019-05-10.

[25] *Cognata Autonomous Driving Simulator (2019) deep learning autonomous and adas simulation*, https://www.cognata.com/, Accessed: 2019-05-10.

[26] *NVIDIA DRIVE(2019) nvidia drive platform - scalable ai platform for autonomous driving*, https://www.nvidia.com/en-us/self-driving-cars/drive-platform/, Accessed: 2019-05-12.

[27] *NVIDIA DRIVE AGX(2019) nvidia drive agx computing platform*, https://www.nvidia.com/en-us/self-driving-cars/drive-platform/hardware/, Accessed: 2019-05-12.

[28] *NVIDIA DRIVE NEWS(2019) nvidia news outlets*, https://nvidianews.nvidia.com/news/nvidia-introduces-drive-autopilot-worlds-first-commercially-available-level-2+-automated-driving-system https://nvidianews.nvidia.com/news/nvidia-introduces-drive-constellation-simulation-system-to-safely-drive-autonomous-vehicles-billions-of-miles-in-virtual-reality, Accessed: 2019-05-12.

[29] *SimCreator DX (2019) simcreator: Real time simulation modeling system*, https://www.faac.com/realtime-technologies/products/simcreator/, Accessed: 2019-06-19.

[30] *SimDriver (2019) simdriver: Autonomous vehicle control solution*, http://faac.wpengine.com/realtime-technologies/products/simdriver/, Accessed: 2019-06-19.

[31] *SCANeR^{TM}(2019) scaner^{TM}: Avsimulation solution*, https://www.avsimulation.fr/solutions/ https://www.avsimulation.fr/applications/, Accessed: 2019-06-22.

[32] *MITSUBISHI MOTORS (2019) specifications - i miev*, https://www.mitsubishi-motors.com/en/showroom/i-miev/specifications/, Accessed: 2019-04-11.

[33] J. Correia, "Visual and depth perception unit for atlascar2", Accessed: 2019-05-29, Aveiro University, 2017, https://ria.ua.pt/handle/10773/22498.

[34] *SICK (2019) LD-MRS400001/LMS151-10100 - detection and ranging solutions*, LD-MRS - https://www.sick.com/de/en/detection-and-ranging-solutions/3d-lidar-sensors/ld-mrs/ld-mrs400001/p/p112355, LMS151 - https://www.sick.com/de/en/detection-and-ranging-solutions/2d-lidar-sensors/lms1xx/lms151-10100/p/p141840, Accessed: 2019-05-29.

[35] *Point Grey Camera Specifications(2019)*, Zebra2 - https://www.flir.com/support/products/zebra2-gige/, FL3-GE-28S4-C - https://www.newegg.com/p/0XR-00KX-00001, Accessed: 2019-05-29.

[36] T. Almeida, "Multi-camera and multi-algortithm architecture for visual perception onboard the atlascar2", Accessed: 2019-05-29, Aveiro University, 2019, unpublished thesis.

[37] *ROS Melodic(2019) ros melodic morenia*, http://wiki.ros.org/melodic, Accessed: 2019-05-01.

[38] *RVIZ (2019) ros.org documentation: Rviz tool with point cloud messages*, http://wiki.ros.org/rviz, http://docs.ros.org/melodic/api/sensor_msgs/html/msg/PointCloud2.html, Accessed: 2019-05-01.

[39] *Rosbag(2019) ros.org documentation: Rosbag tool*, http://wiki.ros.org/rosbag, Accessed: 2019-05-01.

[40] *Roslaunch (2019) ros.org documentation: Roslaunch tool*, http://wiki.ros.org/roslaunch, Accessed: 2019-05-01.

[41] *RQT (2019) ros.org documentation: Rqt tool*, http://wiki.ros.org/rqt, Accessed: 2019-05-01.

[42] *PYGAME pygame front page documentation*, https://www.pygame.org/docs/, Accessed: 2019-05-29.

[43] *SDL simple directmedia layer*, http://www.libsdl.org/, Accessed: 2019-05-29.

[44] *OpenCV (2019) open source computer vision library documentation*, https://docs.opencv.org/ref/2.4.13.7/ https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html https://www.docs.opencv.org/2.4/doc/tutorials/imgproc/histograms/template_matching/template_matching.html, Accessed: 2019-05-08.

[45] *PCL (2019) pcl - point cloud library*, http://www.pointclouds.org/ https://www.howtoinstall.co/en/ubuntu/xenial/pcl-tools http://wiki.ros.org/pcl/Tutorials, Accessed: 2019-05-08.

[46] *CARLA SIMULATOR (2019) carla autonomous driving challenge*, https://carlachallenge.org/, Accessed: 2019-05-29.

[47] *CARLA SIMULATOR (2019) carla ros bridge*, https://github.com/carla-simulator/ros-bridge/tree/master/carla_ros_bridge, Accessed: 2019-04-25.

[48] *Ackermann Function (2019) documentation.* https://www.encyclopediaofmath.org/index.php/Ackermann_function http://wiki.ros.org/ackermann_msgs, Accessed: 2019-04-30.

[49] *Spherical Coordinates (2019) spherical coordinates – from wolfram mathworld*, http://mathworld.wolfram.com/SphericalCoordinates.html, Accessed: 2019-05-16.

[50] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The kitti dataset", *The International Journal of Robotics Research*, vol. 32, pp. 1231–1237, Sep. 2013, Accessed: 2019-05-29. DOI: 10.1177/0278364913491297.

[51] F. Yu *et al.*, "Bdd100k: A diverse driving video database with scalable annotation tooling", May 2018, Accessed: 2019-05-29. [Online]. Available: https://arxiv.org/abs/1805.04687.

[52] N. Silva, "Semi-automatic labelling and tracking of targets for autonomous driving", Accessed: 2019-05-29, Aveiro University, 2018, https://ria.ua.pt/handle/10773/25902.

[53] *Lane Detection Repositories (2019) - by nstell and hsuchechiang*, https://github.com/Nsteel/Lane_Detector https://github.com/HsucheChiang/Advanced_Lane_Detection, Accessed: 2019-05-29.

[54] *CARLA SIMULATOR (2019) add gnss support commit message*, https://github.com/carla-simulator/carla/commit/99c04092, Accessed: 2019-05-29.

Appendices

Appendix A

ATLASCAR Vehicle Module in Unreal Engine 4

A.1 Creating the ATLASCAR Vehicle Module in UE4 and add it to CARLA

During the course of this dissertation, it was verified that it is possible to add a new vehicle asset module to the CARLA simulator by creating the vehicle model in Unreal Engine and the initial idea was to create the vehicle module of the ATLASCAR2 in UE4 and then add that model in the CARLA repository. This however required a lot of experties in UE4 namely in creating the animations for the model. Not to mention the time required to craft each component of this model and put it running in CARLA but the process should look somewhat like what is described in figure A.1.
In order to resolve this issue, we decided to focus on replicating the sensor setup of the ATLASCAR and put it in models already provided by CARLA. This way it is possible to test the sensor results with different vehicle blueprints and see if there are major differences in the results depending on the model that was used.

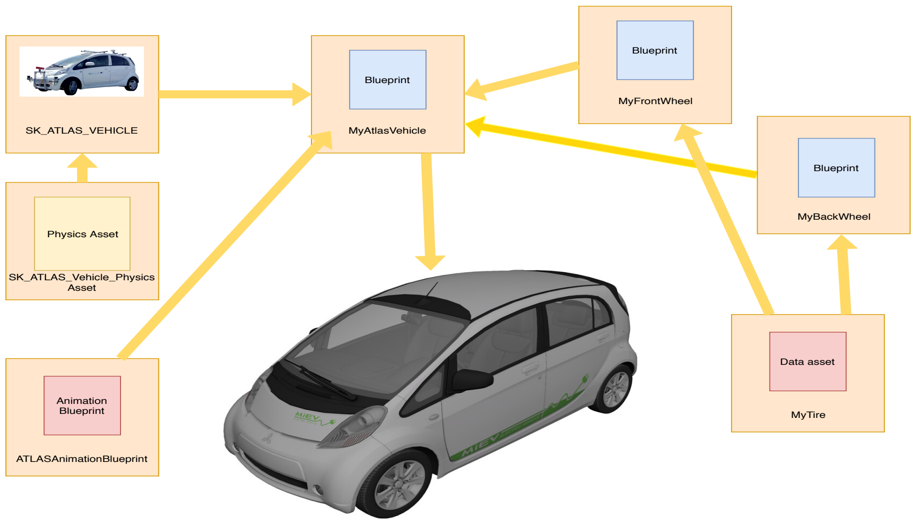


Figure A.1: ATLASCAR2 Vehicle Model Components in Unreal Engine 4.

A.2 Adding New Sensor Types to the CARLA Setup

It is possible to a add new sensor types to the CARLA module by creating a new sensor actor by first creating a new blueprint in UE4 for that sensor, add that blueprint to the vehicle model in Unreal Engine and then add this new model to the simulator. However this process is not so easy to accomplish as the CARLA simulator only accepts known sensor types to the simulator. The only way to achieve this is to add a sensor serializer object for the new sensor actor. The sensor serializer receives the data from the sensor actor and this object comunicates with the simulator via sensor event messages. It is important that when creating the sensor actor class it has a callback method that can be used to gather the data relative to the sensor. The sensor serializer can be achieved by adding a serializer class to the LibCarla repository with two methods, one to serialize the data and another to deserialize the data. The final step is to register both classes in the sensor registry of the CARLA repository and create a sensor event message in order to establish a communication between the sensor actor and the simulator. This same message must also be added to the `carla_ros_bridge` package in order to establish a connection between the sensor actor, ROS and the CARLA simulator. [54] The process design to add sensor in the CARLA setup should look somewhat like what is described in figure A.2.

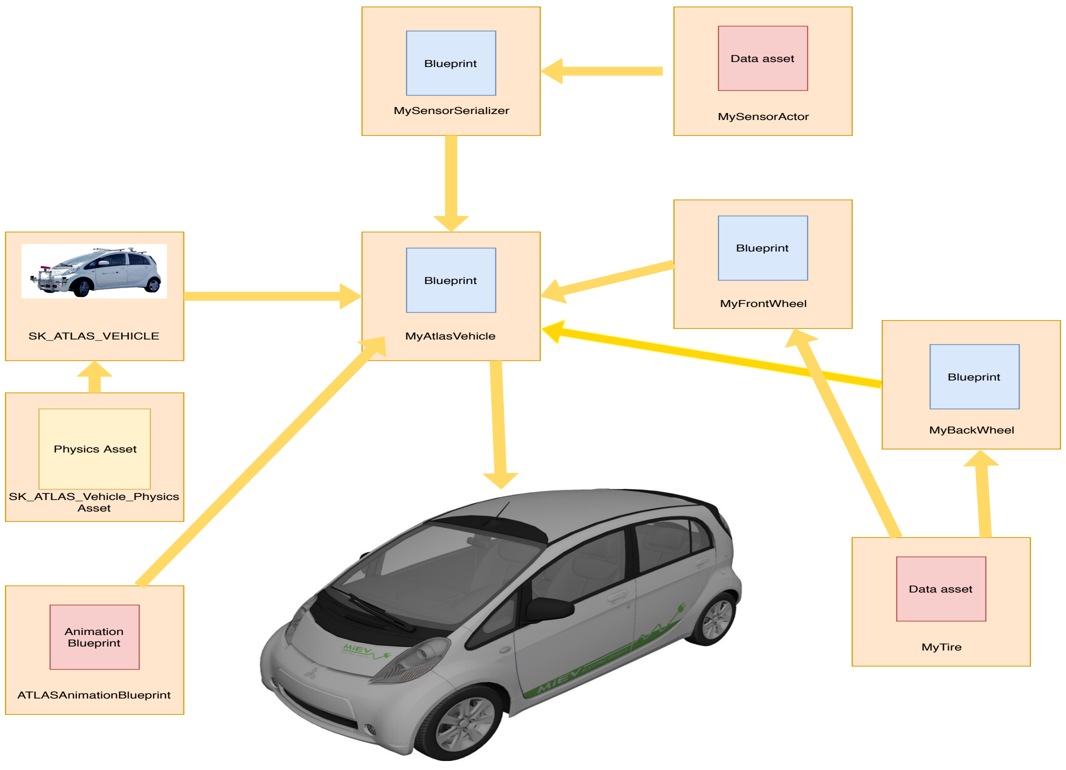


Figure A.2: ATLASCAR2 Vehicle Model Components in Unreal Engine 4 with New Sensor.

Since the basic sensors used in the ATLASCAR (cameras and LIDAR sensors) already exist in CARLA, there was no need to create new sensor types and add them to CARLA, but this section can prove useful for creating new sensor types and see their results before implementing these sensors in the real vehicle.

Appendix B

ROS Bridge Package

The ROS Bridge Package is a ROS package built for the ATLASCAR2 that provides a bridge to the CARLA simulator via sockets and ROS messages with the aim of providing a platform that can be used to gather data assets for the ATLASCAR as well an environment that can be used to test the algorithms before implementing them in the real platform.

This package features several nodes:

- ***carla_ros_bridge*:** Core of the package used to establish a connection between the client and the CARLA server via sockets and ROS messages.
- ***carla_ros_bridge_msgs*:** Stores the ROS messages used in this package.
- ***carla_ros_bridge_lifecycle*:** Controls the lifecycle of the ROS bridge package.
- ***carla_ros_vehicle*:** Replicates the vehicle setup via the sensors.json file and spawns the vehicle in the simulation.
- ***carla_ros_manual_control*:** Opens a Pygame window which can be used to control the vehicle in the simulation.
- ***carla_ackermann_control*:** Converts AckermannDrive messages to control the vehicle using a PID controller.
- ***carla_ros_spawn_npc*:** Spawns NPC objects into the simulation.
- ***carla_ros_pcl_filter*:** Filters the results from the point clouds of the LIDAR sensors to fit and match the LIDAR sensors used in the ATLASCAR.
- ***carla_ros_pcl_recorder*:** Records the point clouds from each one of the LIDAR sensors.
- ***carla_ros_pcl_visualizer*:** visualize the recorded point clouds from the LIDAR sensors.
- ***carla_ros_object_tracking*:** Activates the bounding boxes present in the CARLA objects and create datasets based on the information given by those datasets.
- ***carla_ros_template_matching*:** Implements the Template Matching algorithm using template objects cropped from CARLA.

- ***carla_waypoint_publisher*:** Sets waypoints for the CARLA vehicle.

In order to visualize the functionalities of this package certain ***roslaunch*** files in the ***carla_ros_bridge*** package were created:

- ***client.launch*:** This file is used to establish a normal connection between the client and the server present in the CARLA simulator.

- ***client_with_rviz.launch*:** This file is used to launch the RVIZ tool and play the data that is being recorded in the simulation or plays the data recorded in rosbags.

- ***client_with_rqt.launch*:** This file is used to launch the RQT tool and visualize the ROS messages that are being published during the simulation.

B.1 Install CARLA Python API

In order to use the client Python API provided by CARLA you need to install it in your machine by exporting the PYTHONPATH to the .egg file containing the CARLA library by using the following command:

```
export PYTHONPATH=$PYTHONPATH:<path/to/carla/>/PythonAPI/<.egg>
```

B.2 Run CARLA Simulator Engine

In order to run the CARLA Simulator Engine you must go the CARLA repository and execute the CarlaUE4.sh shell file using the following command:

```
./CarlaUE4.sh -windowed -ResX=320 -ResY=240 -benchmark -fps=10
```

This will open a simulator window which is going to be the main server used for the simulation and from this point onward the server will wait until the client establishes a connection.

B.3 Start the ROS Bridge

The carla_ros_bridge node is the core of the ROS bridge package and this node can be launched with the following commands:

```
# start ros bridge
roslaunch carla_ros_bridge client.launch
# start ros bridge with RVIZ
roslaunch carla_ros_bridge client_with_rviz.launch
# start ros bridge with RQT
roslaunch carla_ros_bridge client_with_rqt.launch
# starts ros bridge with a vehicle setup and a pygame window
roslaunch carla_ros_bridge client_with_vehicle_example.launch
```

B.4 Spawn NPC Objects in the CARLA simulation

In order to spawn NPC objects in the CARLA simulation launch the carla_ros_spawn_npc node by using the following command:

```
roslaunch carla_ros_spawn_npc spawn_npc.launch
```

B.5 CARLA ROS Vehicle Setup Configuration

The carla_ros_vehicle node is responsible for establishing the setup of the vehicle by settings up the sensors with its attributes in the `sensors.json` file. To launch this setup into CARLA you can do it using the following command:

```
# start ros vehicle
roslaunch carla_ros_vehicle carla_ros_vehicle.launch
# start ros vehicle together with client_with_vehicle_example
# launch file from carla_ros_bridge
roslaunch carla_ros_vehicle carla_ros_vehicle_with_client.launch
```

B.6 CARLA ROS Manual Control Pygame Interface

The carla_ros_manual_control node can spawn a Pygame window which can be used to control the vehicle during the simulation and you can launch this node using the following command:

```
# start ros manual control
roslaunch carla_ros_manual_control manual_control.launch
```

The Pygame interface uses keyboard inputs:

- ***WASD*** : Use these keys to control the vehicle.
- ***M*** : Toogle manual transmission.
- ***P*** : Toogle auto pilot.
- ***C*** : Change weather conditions.
- ***Q*** : Toggle reverse.
- ***Space*** : Hand-brake.
- ***H*** : Toggle help.

B.7 Spawn Vehicle with ATLASCAR in CARLA.

In order to spawn the vehicle with the ATLASCAR sensor setup use the following command:

```
# launch carla_ros_pcl_filter together with
# client_with_vehicle_example from carla_ros_bridge
roslaunch carla_ros_pcl_filter pcl_filter_with_client.launch
```

B.8 Record Point Clouds from the LIDAR sensors

You can record the point clouds from each one of the LIDAR sensors by using the following command:

```
# launch carla_ros_pcl_recorder node
roslaunch carla_ros_pcl_recorder pcl_recorder.launch
```

This will save the point clouds in .PCD format in a directory corresponding to the LIDAR sensor from which the point cloud was extracted.

B.9 Visualize Point Clouds from the LIDAR sensors

You can visualize the recorded point clouds by using the following command:

```
# launch carla_ros_pcl_visualizer node
roslaunch carla_ros_pcl_visualizer pcl_visualizer.launch.
```

This will open a PCDViewer window which can be used to visualize the point cloud in its entirety.

B.10 Activate CARLA Bounding Boxes to create datasets

In order to activate the bounding boxes present in each object in CARLA to create datasets use the following command:

```
# activate CARLA bounding boxes
roslaunch carla_ros_object_tracking object_tracking.launch
```

This will store the information about the objects in a `data.json` file in a datasets folder.

B.11 CARLA Template Matching

In order to use the template matching algorithm launch the carla_ros_template_matching node with the following command:

```
# do template matching with template from CARLA.
roslaunch carla_ros_template_matching shape_selection.launch
```

This will open an OpenCV window in which you can hold the left mouse click to form the size of the bounding box that is going to be implemented in the Template Matching algorithm.

B.12 Code Repository

All the code samples used for this dissertation is available on GitHub to be used by any possible contributors in the ATLASCAR project:

- ***ros_bridge***: `https://github.com/lardemua/ros_bridge`