Mechatronic Engineering

Perception algorithms analysis for autonomous drive application in simulated environment

Candidate:

Niccolò Mariani

******

Supervisor(s):

Prof. Massimo Violante

Ing. Luigi Spasiano

Politecnico di Torino

October 2024

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Niccolò Mariani
October 2024

I would like to dedicate this thesis to my loving parents

Abstract

Over the last period, the advancement of autonomous driving technology has made huge steps forward with regard to perception algorithms, i.e. the tools capable of translating sensory data into understandable data. This thesis project aims to delve into the different perception techniques present on the autonomous driving market. In particular, to achieve this goal, CARLA, a cutting-edge open source system for this topic, was used: the main focus is therefore to develop a perception algorithm, obtaining the best possible results in terms of performance within a simulated environment as similar as possible to a complex real-life condition.

Understanding and being able to interpret the surrounding environment in real time is one of the most important aspects of autonomous driving, because all the subsequent choices made by the machine itself and consequently the success of the algorithm depend on it. With regard to this in particular, the CARLA platform was chosen because it is able to provide a world within which to make your own simulations that is totally customizable, and therefore can be made as realistic as possible. The fundamental library of this project is OpenCV, because, through various image manipulation tools, it allows you to process with great accuracy and precision the different frames captured during the experiments, and consequently allow a detection of obstacles or surrounding objects.

The project was divided into different parts, each functional to the next and in general to the success of the whole, starting from the study of the interface features through which it was possible to simulate everything, i.e. the CARLA application programming interface (API), then moving on to the feasibility and in what terms of the OpenCV library, analyzing the different possibilities such as the possibility of using pre-existing techniques such as object edge detection or visual understanding functions, all tools of fundamental importance in the selected field.

In addition to the pure development of a perception algorithm aimed at recognizing road lanes, one of the aspects that have been carried out in parallel is the desire to learn aspects that seem peripheral in software development, but that in reality play a fundamental role in the success of the project and above all in its understanding in the eyes of an outsider: in particular, the use of a correct version control system of the program via Git has been examined, as well as the search for maintaining excellent documentation and therefore in a broader sense the ability to apply the right methodologies for managing a project.

The modularity and maintainability of the code were the prerogatives of the entire project, in order to make it reusable for future projects. In this sense, automatic testing methods were used to make everything even more professional, thus integrating a code that is both functional and functional, but also scalable for other projects or situations.

In conclusion, the results of the thesis give the possibility to explore new scenarios within the implementation of perception algorithms for Lane Detection, thus promoting the development of different technologies compatible with this. Through the use of simulation tools and modern and advanced libraries, this project aims to set the limit beyond the current "normality", exploring new concepts regarding autonomous driving, thus promoting new improvements in this field.

Contents

List of Figures ix

List of Tables xi

1 Introduction 1

1.1 Contextualization of the problem 1

1.2 Advanced driver - assistance system 3

1.3 Part of the testing relating to autonomous driving: road tests and simulations 5

1.4 CARLA: an open-source autonomous driving simulator 7

1.5 Correlation and motivation of the thesis 9

2 Theoretical and Technological Foundations 10

2.1 Description of the CARLA simulator 10

2.1.1 Main features of CARLA 12

2.1.2 Architecture by CARLA 14

2.2 Introduction to OpenCV 16

2.2.1 History and development of OpenCV 16

2.2.2 Main Features and Applications 17

2.2.3 Integration of OpenCV with Python and Other Libraries . . 19

2.2.4 Using OpenCV in Autonomous Driving Applications 20

3 Objective and phases of the project **21**

3.1 Description of General Objectives 21

3.2 Kickoff and hardware setup 23

3.3 Individual studying 25

3.4 Development phase 28

3.4.1 Camera autopilot integration 28

3.4.2 Development and Implementation of Image Thresholding Modules: The Case of frame_threshold.py 31

3.4.3 Development and Implementation of Perspective Transformation Modules: The Case of eye_perspective.py 37

4 Analysis of Implementation Strategies for Lane Detection **44**

4.1 Introduction to the attempts made 44

4.2 First Attempt: Lane Detection Using Horizontal Segmentation . . . 45

4.3 Second attempt: Lane Detection using the continuity of the white pixels 49

4.4 Ultimate Choice for Lane Detection 50

4.4.1 Implementing of the camera_callback() function 52

5 Client Code Implementation and Description **59**

5.1 Introduction to ‘`client.py`‘ 59

5.2 General structure of the code 60

5.3 Accurate description of the behavior of the ‘main()‘ function 63

6 Evaluation of results and development prospects **70**

6.1 Results achieved 70

6.2 Conclusions 74

References **82**

Appendix A Threshold_code **83**

Appendix B Perspective_Transform_code **86**

Appendix C Camera_callback **96**

Appendix D client.py **99**

List of Figures

1.1 Example of a scenario in CARLA 8

2.1 Example of a maps in CARLA 12

3.1 Mercedes coupè 2020, the chosen vehicle 30

3.2 Test in CARLA with the semantic segmentation camera 30

3.3 Camera frame at the chosen position 31

3.4 Before (left) and after (right) the application of the threshold, test image 35

3.5 Threshold applied in the CARLA simulation environment 36

3.6 Original image (left, input) and transformed image (right, output) of the perspective transformation 39

3.7 Result of the view from above 43

4.1 Detection result by scanning horizontal lines 46

4.2 Interpolation with the curve 48

4.3 Interpolation with the polygon 48

4.4 Interpolation with the new method 50

4.5 Results of the ‘`camera_callback`‘ function: the final image on the left, the functional polygons on the right 57

6.1 Highway view, first attempt 71

6.2 Linear urban context frame 72

6.3 Situation on a curve in an urban context 73

6.4 Situation on a curve in an urban context with no data for Lane Detection 76

List of Tables

3.1 Description of different towns 29

3.2 Code metrics for the frame_threshold.py file 35

3.3 Code metrics for the eye_perspective.py file 42

5.1 Code metrics for the client.py file 68

Chapter 1

Introduction

1.1 Contextualization of the problem

Autonomous driving currently represents one of the biggest challenges in both the industrial and academic fields. The reasons are many: the possibility of driving without human intervention could potentially revolutionize the concept of mobility as we understand it today.

The benefits are of different types, starting from safety, as a machine will always be more reliable, if the necessary studies are carried out, compared to the possibility of human error. Furthermore, the hope is that with the introduction of autonomous driving, traffic efficiency can be significantly improved thanks to the possibility of communication between the different elements that compose it.

The last fundamental aspect to take into consideration is certainly the environmental one. The ambition to drastically reduce the circulation of combustion cars by 2030 (across Europe, more than 65 percent of new cars sold are expected to be fully electric by 2030), makes the transition in car production an issue of fundamental importance.

The systems for detecting the environment surrounding the car for autonomous driving must be able to respond and understand numerous circumstances and situations, starting from the detection and recognition of obstacles, the prediction of the behavior of other cars and more, the driving in more complex environments than normal, even when operating in climatic conditions that are not always standard.

All these variables must be studied, both in their individuality, but above all in their collective nature, as each of the above-mentioned situations depends on and influences the others.

1.2 Advanced driver - assistance system

These tasks require the integration of different technologies, including computer vision and everything related to Advanced driver-assistance systems (ADAS), which are the hardware and software components that automate a driver's responsibilities.

These tasks require the integration of several technologies, including computer vision and everything related to advanced driver assistance systems (ADAS), which are the hardware and software components that automate the driver's responsibilities. ADAS systems play a crucial role in the attempt to transition towards full driving autonomy, improving vehicle safety and facilitating the gradual transition from human-driven to self-driving cars.

(Quote Evaluation of ADAS systems) ADAS technologies can be organized, based on their operation, into 6 main categories:

- stability system: the aim is to prevent loss of control or similar phenomena
- longitudinal Dynamics Control: pursuit of longitudinal safety, i.e. avoiding rear-end collisions or similar;
- lateral Dynamics Control: pursuit of lateral safety and therefore preventing any situations of skidding from the lane;
- warning System: these are systems that warn the driver in a timely manner to send an alarm signal to stimulate a reaction from the driver;
- driver Status Monitoring: such as drowsiness or distraction detection systems to identify any anomalous driver behaviour;
- V2V: all ADAS whose operation is based on the exchange of information between vehicles.

By providing real-time data and automated responses to various driving scenarios, ADAS helps mitigate the risks associated with human error, which is a leading cause of road accidents. Furthermore, the data collected by ADAS can be used to further enhance machine learning algorithms and improve the overall performance of autonomous driving systems.

As the automotive industry moves towards the goal of fully autonomous vehicles, the role of ADAS becomes increasingly important. These systems not only serve as building blocks for autonomous technology, but also prepare public and regulatory bodies for the eventual adoption of self-driving cars.

The main studies regarding ADAS systems refer to the following categories:

- Reduction of road collisions: the objective is to estimate the effectiveness regarding the reduction of collisions;
- Benefit/Cost Analysis: estimate of the relationship or important indications on the economic advantage that could arise from the use of these systems;
- Performance: studies regarding the performance of the systems and the possible benefits they can bring to road safety;
- Acceptance: studies whose objective is to know the acceptance or in general the perception of those who drive vehicles regarding ADAS technologies.

1.3 Part of the testing relating to autonomous driving: road tests and simulations

Despite the concrete tests regarding autonomous driving, the limitations regarding road tests are numerous: first of all the high cost as testing autonomous driving vehicles (AV) on public city roads means requiring and obtaining huge investments financial both in vehicles but also and above all in all secondary but fundamentally important infrastructures. These expenses can include the other accessory test vehicles, the qualified personnel necessary to supervise the tests and the possibility of having normally public places reserved for this situation.

Furthermore, precisely because these are tests aimed at improving functionality and avoiding any future damage, the possibility of error is certainly high and consequently the possibility of causing harm to personnel and agents external to the simulation. These errors can be caused by sudden climate changes, irregular behavior of the vehicle or users. These events can in many cases cause physical damage to the people involved but not only that, but also the loss of large sums of money to repair the damage caused.

Then considering the multitude of tests necessary to cover the entire breadth of possible cases, it makes road tests even longer and often unfeasible.

Taking into consideration all these different problems and above all slowdowns that may exist in the testing phase, driving simulators have become increasingly popular, becoming more and more similar to reality but with an enormous possibility of addressing the problems mentioned above. They allow you to recreate complex driving conditions, investigating a multitude of different situations without leaving your desk and above all giving you the possibility of recreating high-risk situations without danger to anyone, while still obtaining reliable results.

Above all, simulated systems allow the speed in changing something that is not correct, viewing the data in real time and monitoring the changes made, making adjustments where necessary. The data generated by the simulations can then easily be fed to machine learning models in order to further refine the algorithms.

In conclusion, although on-road testing remains an essential component of the development of autonomous vehicles, the limitations associated with it highlight the importance of using simulators. These tools provide a safe, convenient and

efficient means of testing, ultimately accelerating progress towards achieving fully autonomous driving.

1.4 CARLA: an open-source autonomous driving simulator

To deal with the challenges and limitations of road tests that we talked about before, advanced driving simulators have been created and one in particular fully reflects the needs for testing autonomous driving cars: CARLA (Car Learning to Act) is an open source simulator whose purpose is the development, training and validation of autonomous driving systems. In particular, it offers a simulated environment with a large variety of variables capable of making each simulation personal, from atmospheric conditions to the most disparate road situations to the choice of vehicle on which to carry out the tests: it is undoubtedly a possibility to drastically reduce costs regarding certain specific tests.

The replica of the simulated environment is a faithful reconstruction of the real world. The situations in which tests can be carried out are of various types, starting simply from the possibility of varying the weather conditions: this is the peculiarity of the simulated systems and in particular of CARLA, as it allows rapid but substantial changes to be made.

The complexity of simulation situations can be addressed quickly and safely, as a circumstance such as a pedestrian crossing, if it were not simulated, would entail high and above all unnecessary risks as they are easy to replicate in systems such as the one we are talking about . This capability allows for extensive testing and validation of perception algorithms, ensuring that autonomous vehicles can detect and respond to potential hazards accurately and in a timely manner.

Another aspect of fundamental importance is the variety of sensors commonly used on self-driving vehicles, starting from standard cameras, up to LIDAR and radar with high accuracy and high performance in correlation with the real world. By providing realistic data from sensors, CARLA allows developers to quickly, cheaply and safely refine the performance of object perception and detection algorithms.

A factor of fundamental importance regarding CARLA specifically, being a totally open source platform, allows for strong collaboration between the developers who decide to use it, allowing the community to share information and their experience completely free of charge.

Finally, unlike standard tests carried out on the road or in real situations, the use of CARLA allows you to significantly reduce costs, requiring only relatively low and accessible computing power: it is from the cost - effectiveness ratio that the power of the simulators and in particular in this case, by CARLA.

In conclusion, CARLA represents a powerful tool for overcoming the limits of road tests. By providing a safe, realistic and cost-effective simulation environment, CARLA enables the comprehensive testing and development of perception algorithms essential for autonomous driving. Its accessibility and flexibility make it an invaluable resource to advance the field and bring us closer to the mass use of autonomous vehicles.

Fig. 1.1 Example of a scenario in CARLA

1.5 Correlation and motivation of the thesis

Taking into consideration all the aspects mentioned and the aim of the thesis project, namely that of pursuing and obtaining the perception algorithm to assess its performance and suitability for autonomous driving tasks, CARLA acts as a fundamental tool for analyzing and evaluating the different possible and feasible algorithms for autonomous driving systems. The thesis aims to evaluate these road lane recognition algorithms in the simulation environment, trying to find techniques that allow greater performance in different scenarios.

The possibility of a variety of situations but at the same time the simplicity with which even substantial changes can be made are the main reasons that led to the choice of this simulator.

Through comprehensive experimentation in CARLA, the thesis attempts to provide insights into the strengths and limitations of different perception algorithms. By comparing their performance parameters, including detection rates, false positives and computational efficiency, the study aims to highlight which techniques are best suited to achieve reliable and safe autonomous driving capabilities.

The possibility of simulating scenarios that are even very distant and otherwise difficult to reach allows you to further evaluate the robustness and resilience of the algorithms in unexpected situations.

Ultimately, using CARLA as the primary testing platform, this thesis seeks to provide valuable findings and recommendations to advance the development of perception algorithms in autonomous driving systems.

Chapter 2

Theoretical and Technological Foundations

2.1 Description of the CARLA simulator

The CARLA simulator has been taken into strong consideration in particular since, in 2018, the Toyota Research Institute decided to donate one hundred thousand dollars to accelerate the development of the Car Learning to Act simulator: a simulator for autonomous driving that makes open source its distinctive feature, where all those who want to give their contribution to research can do so completely remotely and free of charge.

This is a project integrated with the Github platform, developed primarily by the CVC - Computer Vision Center - of the Universitat Autonoma de Barcellona (UAB), in Spain. As Toyota explains, the project is focused on "the development, training and validation of autonomous driving systems in urban centers" and is designed to "ensure the reliability of autonomous vehicles in a myriad of situations that are not always testable in the real world" , or at least, not with the same simplicity.

The CARLA project was created mainly to make research in this field 'democratic'.

Since its creation, CARLA has been continuously improved and updated thanks to the contribution of a large community of researchers and developers from around

the world. Its growing popularity and use have made it one of the main tools for simulating and testing autonomous driving systems.

The main purposes of the simulator are each aimed at the advancement of autonomous driving technology, and in particular they are the following:

- Facilitating Research and Development: system aimed at supporting research in the development and validation of algorithms aimed exclusively at autonomous driving: the environment is realistic and controlled, efficiency is the most important distinguishing factor;
- Realistic and Detailed Simulation: the similarity between the simulated world and the real world is such thanks to the possibility of choosing dozens of variables within the simulator: urban and rural scenarios, different weather conditions and above all different traffic conditions. The robustness and adaptability of the results is a consequence;
- Safety and Risk Reduction: The testing phases have always been the most expensive phase in the development of a project: with CARLA a solution to this problem was found, allowing the simulation of dangerous circumstances or circumstances that are difficult to replicate in real life. The risk of accidents is therefore reduced to almost zero;
- Reduced Costs: researchers can conduct extensive testing without the high costs associated with real-world testing, allowing resources to be allocated more efficiently;
- Collaboration and Sharing: given the open - source nature of the simulator, the great strength of CARLA is inherent in its idea when it was created: collaboration between academic institutions, companies and independent developers. Sharing becomes the discriminating factor when choosing which tool to use;
- Training and Education: CARLA is also used as an educational tool to train the next generation of engineers and researchers in the field of autonomous driving. By providing an accessible and versatile simulation environment, the simulator allows students to acquire practical and theoretical skills in a realistic context.

2.1.1 Main features of CARLA

The strong point of CARLA is certainly the possibility of having a highly detailed and realistic simulation and above all the possibility of making your simulation extremely personal and unique. The main features of the simulation environment are:

- Maps: CARLA provides a multitude of predefined maps that give the possibility to vary between different urban, rural or even highway scenarios. Urban maps in particular are designed to represent complex situations such as traffic intersections, traffic lights, horizontal and vertical signs, and different types of buildings.

  The most interesting feature, however, is the possibility of creating customized maps for even more specific scenarios using the tools provided by CARLA for their creation.

Fig. 2.1 Example of a maps in CARLA

- Weather: CARLA allows you to modify the weather conditions both in a standard way, before running your application, and in real time, including sun, rain, fog, snow and different brightness levels: although it may seem only for context, it is a feature of fundamental importance to be able to manage every scenario of daily and non-daily life;

Furthermore, CARLA provides the opportunity to choose between a large variety of vehicles and secondly allows you to also include the presence of pedestrians along the roads, in order to make the simulation even more truthful:

- Vehicles: the possibility of choice regarding vehicles leaves total freedom to the user to adapt the simulation to their needs; Each vehicle can be customized in terms of driving dynamics and physical characteristics, allowing algorithms to be tested on different types of vehicles and configurations;
- Pedestrians: CARLA has pedestrian models with different movement and behavior characteristics. Pedestrians can walk, run and cross streets realistically, providing complex scenarios to test the ability of autonomous vehicles to detect and react to pedestrians.

A factor of fundamental importance is certainly the accuracy of the replicas of the sensors for self-driving cars: these instruments can be configured and positioned on the vehicles in a totally customizable way to collect the data necessary for the perception and navigation algorithms:

- cameras: the cameras that can be chosen are for example the RGB one, or depth camera and infrared cameras: they can be positioned anywhere on the machine, according to needs;
- LIDAR: they are sensors that emit laser pulses to detect objects and measure distances: the data provided by LIDARs can be fundamental for the creation of maps and above all for the precise detection of obstacles;
- GPS: thanks to this tool, localization data can be collected in a precise and simple way: the data is very important for control systems;
- Radar: RADAR systems are of fundamental importance with regard to the real-time recognition of moving objects, such as other vehicles, but not only, and for measuring their relative speed;
- Depth sensors and semantic segmentation: they can be considered as some sort of cameras capable of providing detailed information on the environment, in particular regarding the depth of objects and the semantic classification of the scene, providing fundamental data for the distance between the different agents that animate the simulation.

2.1.2 Architecture by CARLA

The CARLA (Car Learning to Act) architecture is conceived and designed to be flexible but above all modular, i.e. composed of units that can be added, eliminated or modified without having repercussions on the rest of the system, in order to allow developers to use realistic systems. The structure of the software architecture is based on multiple specific levels aimed at collaborating for an accurate simulation.

The physics of the simulated world is managed entirely by the core of the architecture, i.e. by the simulation engine that uses Unreal Engine for three-dimensional environments.

Above this level there are other modules to manage other simulation logic.

The key levels or modules of CARLA are as follows:

- Simulation Engine: module entirely based, as previously mentioned, on Unreal Engine and is the component that manages the graphics and interaction between objects in the simulated world. It is also responsible for vehicle dynamics;
- Server Carla: communication management module between the simulation engine and external clients of all types; in this sense it is solely responsible for the synchronization of data between the different factors that make up the system and for the management of client requests;
- Client Python API: interface to interact with the CARLA server. Users can use Python scripts to control vehicles, configure sensors, and collect data from the simulation. This API is essential for integration with other machine learning and data processing tools;
- Vehicle Modules: fundamental for the realism of the simulation as they are responsible for the vehicle dynamics models, controllers for autonomous driving and above all artificial intelligence models with which to simulate different situations.

The main programming languages that can be used with CARLA are Python and C++: the first of the two is the one mainly used for interaction with CARLA, as the Client is written in this language. The choice of Python is due to its simplicity

combined with immense power in data processing, as well as the enormous breadth of tool libraries for example for machine learning or artificial intelligence. As regards the most critical and complex parts of the simulation engine and CARLA server, C++ is used as it guarantees excellent performance and is useful for the precise control it offers over system resources, essential in particular for the management of 3D graphics and especially real-time physics. In summary, CARLA's architecture is designed to be highly modular and scalable, allowing developers to customize and extend the platform according to their specific needs. The combined use of Python and C++ provides a balance between ease of use and high performance.

2.2 Introduction to OpenCV

OpenCV (Open Source Computer Vision Library) is an open-source library for computer vision and not only that, but also for machine learning. The purpose for which it was designed is to provide a common structure for computer vision applications and in particular to speed up the use of technologies based on artificial intelligence or similar. OpenCV is classified as one of the most complete and well-stocked libraries in this specific field and for this reason one of the most used, thanks to the enormous possibility of different functionalities and the simplicity with which it can integrate with other technologies in the field in question.

2.2.1 History and development of OpenCV

OpenCV was initially created by Intel (designs, manufactures and sells computer components and related products for business and consumer markets) in 1999 as part of a research project related to image processing and in particular regarding computer vision; the primary objective was to be able to have a standard library for everyone for artificial vision purposes and which could be used both in the industrial sector, by Intel itself, but also in the academic sector, in order to broaden the segment of people to turn to: the choice to make the project totally open - source has further facilitated these two issues mentioned above, facilitating the use of this library in different contexts and above all continuously improving its functionality and its robustness, thanks especially to the developers of Worldwide.

Immediately experiencing great success and great approval, OpenCV received important and fundamental contributions from large companies, one above all WIllow Garage (robotics research lab and technology incubator devoted to developing hardware and open source software for personal robotics applications), one of main offices where, among others, the ROS project, Robot Operating System, open-source robotics middleware suite, was developed. Although ROS is not an operating system but a set of software frameworks for robot software development: this step was fundamental in the growth of the project because it helped integrate OpenCV into robotic applications.

Since 2012, the administration was then handed over to OpenCV.org, an organization aimed at improving and developing the library. Since its first release, in

2000, OpenCV has had regular updates and substantial improvements over time, so much so that in 2012 it exceeded 2 million downloads, an incredible number for those years: to date, an estimated 200 million downloads, with a which is around 500 thousand per month.

Furthermore, after 2012, given the great trust it found both among professionals and among ordinary sporadic users, it obtained the support of huge companies such as Microsoft and Google, maintaining its finances solid and the possibility of further expanding its features.

2.2.2 Main Features and Applications

OpenCV presents itself on the IT market as one of the most used but above all appreciated libraries with regards to artificial vision and image processing: the main objective has always been to provide a common structure for these applications in order to accelerate the use of perceptions based on machine learning in research but also in the development of commercial products.

The vastness of application possibilities and the potential of OpenCV represent some of the distinctive characteristics of this infrastructure. This library allows for quick and intuitive manipulation of images, allowing developers and others to read and work on images with extreme ease.

One of the prominent operations of this library is the conversion of images between different color spaces, such as RGB, which is an additive color model in which the red, green and blue primary colors of light are added together in various ways to reproduce a broad array of colors, grayscale and HSV, cylindrical-coordinate representations of points in an RGB color model.

Another fundamental aspect that makes OpenCV even more useful is its ease and robustness regarding the image filtering system. Filters can be useful for different purposes, starting from noise reduction which can make an image better, sharper but above all easier to work with, or for example to extract other characteristics of the same.

An example of fundamental importance is certainly the Gaussian filter which is often used for noise reduction before tackling further analyzes of any kind. Another important feature, especially in the field of object and obstacle recognition in

autonomous driving, is the recognition of their edges, implemented thanks to the Canny algorithm; this technique is also of fundamental importance for robotics.

Geometric transformations are one of the peculiarities of OpenCV as the library offers tools to resize, rotate or deform images, including more complex tools for reconstructing an artificial image thanks to transforms, such as for the Eye Perspective View.

OpenCV also includes powerful tools for object detection. Using machine learning and deep learning techniques, OpenCV can be trained to recognize and localize specific objects within an image or video. This ability is critical in applications such as facial recognition, surveillance and autonomous driving systems.

The detection of lines and geometric shapes represents once again a strong point in favor of OpenCV, as, through the Hough transform, an extraction technique used in the field of digital image processing. In its classic form it is based on the recognition of the lines of an image, but it has also been extended to the recognition of other arbitrarily defined shapes, the library is able to recognize straight and or curved lines with extreme precision: this aspect is of fundamental importance regarding the analysis of road images.

The integrations with libraries and external functions are then very simple and intuitive and make it possible to further expand the capabilities available to the developer.

The OpenCV architecture is generally based on modules, each of which has its own main functionality but can be used both individually and with other modules.

- Core Module: main module of OpenCV, responsible for both the basic data structures and the main image processing functions: it also includes the necessary mathematical operations and functions for the image input/output process;
- Image Processing module: responsible for the functions aimed at manipulating and analyzing images, such as filtering, geometric transformation or other types we talked about above;
- Video Module: similarly to the Image Processing Module, it is responsible for video processing, including algorithms for compression and decompression, object detection and other functions aimed at stabilizing images in real time;

- Object Detection Module: exploiting techniques such as machine learning or deep learning, it allows the use of very complex and advanced algorithms for object detection;
- Calibration and RD Reconstruction Module: provide methods for calibrating images coming from cameras and others, with the possibility of correcting errors but above all it allows the reconstruction and therefore the use of two-dimensional images, but translating them in order to obtain three-dimensional images;
- Machine Learning Module: tools for training and predicting machine learning algorithms, to facilitate the process of using them;
- Contrib Module: it is a module within which contributions from the community that uses OpenCV are collected, where algorithms or similar are collected which however are not yet part of the core of the library.

In summary, OpenCV is an extremely versatile and powerful library that offers a wide range of features for image processing and computer vision.

2.2.3 Integration of OpenCV with Python and Other Libraries

The integration of OpenCV with Python represents one of the most prolific and versatile combinations in the field of computer vision and image processing; Python is a programming language widely used in web applications, software development, data science, and machine learning (ML). Developers use Python because it is efficient and easy to learn and can run on different platforms. Thanks to the presence of numerous external Python libraries, it is possible to integrate OpenCV with them in order to further expand the already vast capabilities.

An example could be the use of libraries such as TensorFlow and PyTorch with regards to machine learning, where the documentation and specific wrappers make the integration between the different parts even easier.

Another example is the use of the NumPy library, an open source library for the Python programming language, which adds support for large multidimensional matrices and arrays along with a vast collection of high-level mathematical functions to be able to operate efficiently on these data structures: this cohesion made

simple and intuitive allows you to perform complex operations on images or videos, significantly improving the performance and scalability of applications.

2.2.4 Using OpenCV in Autonomous Driving Applications

Autonomous driving is undoubtedly one of the greatest challenges for humans in the next decade, and in this sense OpenCV could play a crucial role in its success.

The image processing capacity of the library and the possibility of developing and implementing visual perception algorithms, of fundamental importance for the functioning of the technologies necessary for autonomous driving.

This library certainly allows us to tackle the problem of road lane detection in a modular and systematic way: the algorithms are multiple and use in particular the Hough Transform which allows us to identify and trace the lanes on the road; furthermore, OpenCV allows us to apply filters to improve the quality of image perception and above all to reduce noise, significantly improving performance.

Another very important aspect concerns object detection through the use of convolutional neural networks (CNN), a type of feed-forward artificial neural network in which the connectivity pattern between neurons is inspired by the organization of the animal visual cortex, whose individual neurons are arranged in such a way as to respond to the overlapping regions that tile the visual field. Additional libraries, such as TensorFlow or PyTorch, certainly help in this aspect.

This information is very important for the correct navigation of vehicles.

Another aspect of fundamental importance is the management of sensors and the data that these tools return; OpenCV greatly facilitates these operations, allowing you to align and superimpose data from different sources to obtain a 360-degree view of the situation.

In conclusion, the library allows you to manage and process many features necessary for the success of autonomous driving.

Chapter 3

Objective and phases of the project

3.1 Description of General Objectives

The underlying theme of this master's thesis is the analysis and subsequent implementation of perception algorithms in order to be used for autonomous driving, all tested in simulated environments. In particular, the project is committed to using a road scenario that is as true to life as possible and similar to reality. CARLA, the simulator to which it refers, allows an in-depth analysis of the performance of algorithms but in safe and replicable contexts.

Another aspect of great importance for the project is being able to acquire technical but above all methodological skills for software development and project management. For this purpose, the Python programming language was used, versatile and with exponential growth during learning. Some versioning tools such as BitBucket and Git were also explored, as well as the practice of software documentation and design techniques.

At the end of the project, in order to be able to evaluate the performance of the perception algorithms used, it was necessary to measure and analyze the performance through specific indicators; the use of Key Performance Indicators (KPIs) allows you to obtain a specific value for different factors on a scale of goodness that allows even a less experienced reader to appreciate the precision of the code.

Finally, the project of these months also includes familiarization with development environments and tools such as the Linux operating system, the necessary

practice with bash commands and scripting to manage some of its activities and the use of external tools such as Confluence for the step-by-step management of the code drafting and more.

Within the thesis project, in line with what was also defined by the host company, a specific strategy was adopted to divide the work into very distinct phases, each of which with a precise purpose: this approach allows the work to be conducted in systematically, ensuring that each of the five phases is completed. In particular, the first phase, that of the "Kick off and hardware setup", involves the initial organization, specifically of the laptop provided, followed by a fairly long period of individual study, coinciding with the second phase of the project, of fundamental importance to familiarize with the work tools. The third phase is aimed at the development of the actual project, with a specific focus on the code level. The fourth phase is dedicated to finalizing the project, both in terms of documentation and reporting of results. The last phase instead focuses on writing the thesis and preparing the final discussion.

The division of the project into small tasks, in this project but also in many others, allows all those who work there to monitor the progress step by step, giving the possibility of solving problems before they become too large.

3.2 Kickoff and hardware setup

This section has the task of describing the first tasks carried out during the initial setup phase: the goal is to be able to provide a 360-degree view of the processes followed and the choices made.

In particular, the Kickoff phase included the definition of the work tools to be used, such as the company laptop, the choice of the computer language to be used, the planning of the activities to be followed and the objectives to be achieved in the medium and long term.

The hardware was immediately set up thanks to the help of the internal IT of the Luxoft company, configuring the laptop according to the needs of the project.

The phase to which the most time was dedicated was the one relating to the creation of the team behind the project, starting from the main tutor, the engineer Luigi Spasiano and the technical and experienced support of Nicola Sabino. Working closely from the very beginning with these two prominent figures of the company made it possible to achieve the objectives of the project. The weekly and sometimes daily meetings greatly facilitated communication and the smooth step-by-step progression of the work.

The work methodology used throughout the duration of the project within the company follows the directives of the employees themselves, thus identifying as such. The AGILE methodology represents a project management method, particularly related to the IT environment, which tends to enhance flexibility, collaboration between the different parts of the team and above all the free communication of ideas and proposals between them, in order to free up change aimed at improvement. Initially it was designed solely for the software sector, but in recent years it has gained more ground also in environments such as design but also research and development, as it allows to respond effectively to the speed of current innovation [1].

The AGILE Manifesto, drawn up in 2001, represents a turning point in the idea of software development and project management. This document defines the fundamental values and principles that guide this working methodology, which is designed to respond effectively and rapidly to transformations and to improve the final product. This Manifesto is composed of twelve key principles:

1. `Customer Satisfaction`: satisfy the customer by delivering valuable software early and continuously;
2. `Welcoming Changes`: accept changes in requirements even during the project, using these changes to improve the product and meet customer needs;
3. `Frequent Delivery of Working Software`: delivery of a product periodically in a short time to obtain rapid feedback and over a short period of work, in order to avoid incurring major development errors;
4. `Daily Collaboration`: step-by-step group work so that each participant is always kept up to date with the project's developments and news;
5. `Projects Based on Motivated Individuals`: motivation and trust are the pillars for a successful project;
6. `Face-to-Face Communication`: conversation and direct dialogue between team members to avoid misunderstandings;
7. `Measuring Progress`: a functioning product is the main measure on which to calibrate the success of the project;
8. `Sustainable Development`: This method allows for continuous development at a constant pace;
9. `Technical Excellence and Good Design`: attention to the refinement of the project;
10. `Simplicity`: linearity in order to make the work of the different components feasible;
11. `Self-Organizing Architectures`: the aim is to find the best method within the working group itself, without any necessary external impositions;
12. `Reflection and Adaptation`: adjustments to working methods periodically, in order to change if necessary for improvement.

The use of these principles within the project at Luxoft made it possible to have a balanced work period in terms of workload but also emotional state, influencing the ways of planning the work. These decisions improved the overall quality of the final result [1].

3.3 Individual studying

The purpose of this second phase was to obtain a solid theoretical and practical basis to be able to tackle the project, from different points of view, both with regard to the programming language and the CARLA environment. If it was necessary to gain confidence with the simulation software, it was also necessary to dedicate the right amount of effort to be able to become familiar with this method with regard to the management and use of the company's AGILE system. The ability to plan the work and objectives with the team, the ability to carefully choose what to focus on and what it was not necessary to waste too much time on and above all to understand where the critical issues to be thinned out were, were some of the fundamental points of this phase.

Initially, it was thought that it would be possible to tackle a study of the C++ language first, in order to be able to delve into the aspects of the development of the software in question with high accuracy, being able to go into depth: this language would have given the possibility of having control over the low-level details of the algorithm. However, despite the numerous advantages that C++ gives, the language was found to be rather difficult with regard to syntax and memory management. The reasoning that led to the change of direction towards Python is mainly due to its exponential learning curve since, especially in terms of timing, it guarantees better performance. Python allows the use of a decidedly simpler and more intuitive syntax, streamlining the code and relieving the developer of some issues not strictly connected to the project in question. Another aspect of fundamental importance that tipped the scales in the direction of Python is undoubtedly its ease of accessibility with external libraries such as OpenCV and NumPy, suitable for image processing and consequently for the implementation of perception algorithms.

This choice proved to be a winning one, as it gave the possibility of concentrating energies on the development of the algorithm without having to deal with the critical issues related to the complexity of the C++ language.

Within this phase it was of fundamental importance to also address other technical and theoretical aspects in order to obtain a good result of the project: three main environments required greater attention, namely the CARLA API, the Linux environment and the study of the modern situation of lane detection, useful for taking stock of what is already present in terms of technologies and approaches in this area.

An API (Application Programming Interface) is a cluster of rules and details that together allow different software to communicate with each other effectively. APIs therefore specify the methods and formats through which communication between applications and software is possible.

APIs can be divided based on their use and their level of access:

- `system API`: interface between the operating system and individual applications such as the Windows API which allow applications to be related to the OS features;
- `library API`: communication interface to use predefined software libraries, for specific functions, methods or procedures called by developers without the need to work directly;
- `web API`: communication between application systems and network-based services, such as the Internet;
- `third-party APIs`: equipped directly by external services in order to give the possibility of using the required functions without having to worry about how to create them from scratch, but only about how to integrate them.

Thanks to APIs, modularity becomes of fundamental importance, i.e. the principle of software design and organization in which a complex system is divided into smaller and well-defined parts called "modules", making systems simple that would otherwise be much more complex [2]. The CARLA API in this sense can be included within the Web API category as it is capable of relating a network to the simulator in real time: similarly, however, it can be understood as capable of providing methods and classes that can be used exactly as libraries thanks to programming languages such as Python, and consequently consider these features as a library API. In any case, the study of this API was fundamental as it gave the possibility of understanding how to interact advantageously with the simulated world and exploit all its possibilities. The CARLA API [3] also provides a multitude of examples from which to draw inspiration and above all some very useful clients if you do not want to mess with those aspects and use other people's scripts that work correctly, as has been done for some aspects of the project such as that of guidance within the world.

The second aspect of great importance in the initial period was the basic learning of the Linux operating system and its features. Most of the development tools and

especially the libraries used during project development are optimized for Unix - like environments. Among other things, tools such as the basic and advanced bash commands for managing files and in particular directories were carefully analyzed and secondly an important amount of time was dedicated to versioning thanks to Git and Docker. Linux gave the possibility to keep the project within a stable environment which allowed the perception algorithms to be implemented and tested extremely efficiently.

Finally, the last phase was dedicated to collecting information and documents relating to the Lane Detection project, considering the current situation available online, looking for the strengths and weaknesses of pre-existing projects. In this sense, in fact, this particular function refers to one of the main functions to which an autonomously driving machine must be able to respond, both in standard cases, so to speak, but also in more complicated cases such as in the presence of traffic, low light or bad road markings. By studying and observing the methodologies and study circumstances currently present, it has been possible to define the critical issues and strengths, while also identifying some shortcomings: currently online it is very easy to find Lane Detection systems that are perfectly functional and also relatively simple. to be implemented but mainly dedicated to driving on the motorway, a decidedly restricted research environment both in terms of the speeds and possible situations that must be faced, but also for a matter of curvatures which, unlike the city environment, are much lighter and easy to detect. Due to this lack, the desire to create an algorithm capable of detecting road lanes in a more chaotic and complex environment such as that of the city was decided at this stage.

3.4 Development phase

The third phase of the project, called Development phase, lasting approximately three months, is the central part and the heart of the project as it was the period in which previous theoretical knowledge, literature studies up to that moment and the practical development skills, implemented and tested in order to obtain efficient and functioning perception algorithms within the CARLA simulator. The transposition of the design specifications into a functioning software solution was therefore the greatest challenge, not only at a purely computer language level, but also purely at a geometric - mathematical level.

This phase was divided into different sub-phases such as the design of the system, the writing of the code itself and the consequent need to integrate these parts within the simulator and finally the testing and validation phase, which is also complicated but extremely necessary .

During the Development Phase, particular attention was paid to the use of best practices in terms of code management, documentation, and versioning, thus ensuring that the project met the required quality standards.

In the following subchapters, the design choices that have been made will be explained and analysed, also including natural errors and second thoughts in these development phases, also going against the current with theoretical choices made previously, as during the development phase problems and critical issues not analyzed before, or on the contrary recognizing potential not previously considered.

3.4.1 Camera autopilot integration

Starting the development phase, the first fundamental part was the choice of the setup in its most specific parts, necessary to establish starting points from which to then begin to specifically develop the perception algorithm. First of all, it was necessary to define the world to rely on: this was important and crucial because the realism and repeatability of the project depends on it. The CARLA simulator allows you to use different worlds, each of which has its own specific characteristics useful in certain situations [4].

After an analysis of already existing perception algorithms, which mainly focus on specific driving situations such as driving on the motorway where you are almost

Town	Summary
Town 01	A small, simple town with a river and several bridges.
Town 02	A small simple town with a mixture of residential and commercial buildings.
Town 03	A larger, urban map with a roundabout and large junctions.
Town 04	A small town embedded in the mountains with a special infinite highway.
Town 05	Squared-grid town with cross junctions and a bridge. It has multiple lanes per direction. Useful to perform lane changes.
Town 06	Long many lane highways with many highway entrances and exits. It also has a Michigan left.
Town 07	A rural environment with narrow roads, corn, barns and hardly any traffic lights.
Town 08	Secret "unseen" town used for the Leaderboard challenge.
Town 09	Secret "unseen" town used for the Leaderboard challenge.
Town 10	A downtown urban environment with skyscrapers, residential buildings and an ocean promenade.
Town 11	A Large Map that is undecorated. Serves as a proof of concept for the Large Maps feature.
Town 12	A Large Map with numerous different regions, including high-rise, residential and rural environments.

Table 3.1 Description of different towns

on a straight line or with short and "soft" curves, an urban world that includes signals was chosen both horizontal and vertical roads and the presence of other cars. This environment, despite requiring greater attention to various details, gave the opportunity to address problems which, once resolved, gave light to new and interesting ideas, which led to the achievement of an excellent final result. Furthermore, we tried to reproduce an environment as faithful as possible to reality and the everyday life of an average customer.

Then the virtual car from which to use and on which to mount the camera was then chosen. The CARLA simulation environment provides a wide range of vehicles, each with different characteristics, once again allowing for a wide range of choices to satisfy any developer need: even the dynamics and kinematics of the cars themselves are taken care of detail. For this project in particular it was decided to use a classic sedan, in order to remain as standard as possible without going into detail regarding the heights and general measurements of the vehicle. This model,

Fig. 3.1 Mercedes coupè 2020, the chosen vehicle

intended as a common small car, was chosen for the possibility of comparing the results obtained with other studies and research analyzed in the literature. As regards the selection of the camera, CARLA offers various alternatives, each dedicated to particular objectives. Initially, the Semantic Segmentation Camera was tested, which includes an automatic classification of pixels within certain ranges, which were then all considered equal in order to have figures with uniform colours: although this could have significantly helped the success of the project, in the same way it differs greatly from reality as it allows automatic detection of obstacles or other objects but only within the simulation, differing greatly from how it is normally done for lane detection or in general object recognition. Once this type of camera was discarded, it

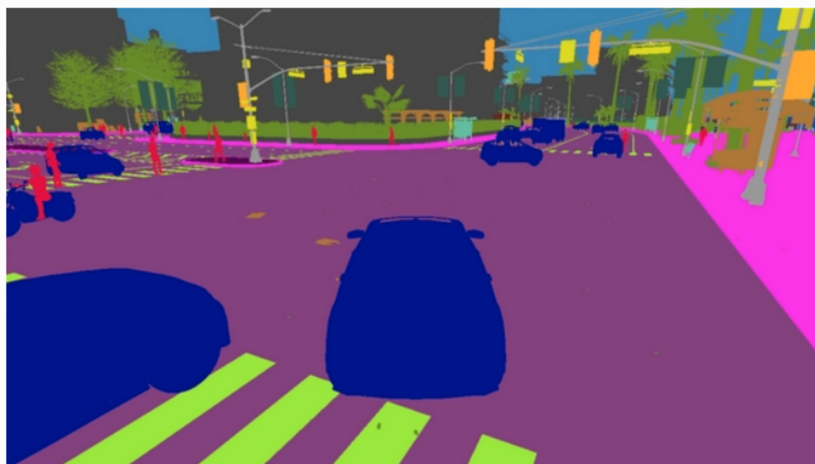


Fig. 3.2 Test in CARLA with the semantic segmentation camera

was decided to take inspiration from the dash cam, in particular regarding the central position, just below the rear-view mirror, in such a way as to be able to have a broad vision and good resolution; the only change was its inclination, slightly downwards to point towards the road. This configuration has proven to be ideal for acquiring images useful for detecting road lanes, as shown in figure 3.3. The last part of the

Fig. 3.3 Camera frame at the chosen position

set up was related to the integration between the server, i.e. the part relating to the management of the CARLA simulation world, and the client, i.e. the part in which the choices regarding the perception algorithm, with the related calculations and decisions in order to perform Lane Detection correctly.

This part of integration is of fundamental importance as it is the basis for correctly taking individual frames and consequently being able to process them correctly.

3.4.2 Development and Implementation of Image Thresholding Modules: The Case of frame_threshold.py

The thresholding to be applied to images is a key step within computer vision and in particular in image processing. This technique consists in the conversion of an image to gray tones, i.e. each pixel of any color is converted to a very precise gray scale so that each shade means that this pixel is more or less dark: once converted in this sense the The grayscale image is converted into a binary image, thus defining a threshold above which it is decided to make that pixel "active", or similarly in our

case considered as white, and below the threshold the pixel is considered as "off", or similarly in our case considered as black.

This method is of vital importance in the recognition of objects and their contours because it allows to "highlight" in a certain sense only the factors of importance for the algorithm and to "delete" everything that is not needed for the purposes of a good outcome of work.

This module, called ‘frame_threshold.py‘ therefore has the task of applying a simple thresholding to the single frame that is processed during the running of the application, in such a way as to obtain a black and white image with only white, or almost, the elements of our interest.

This work was possible thanks to the implementation of OpenCV which made this operation simple, intuitive and above all replicable. The main purposes of this module are three: the efficiency of the module, the flexibility with which it can be applied to different environments and frames, the ability to save the results autonomously and without the need for external intervention.

As explained within the official OpenCV documentation, there are several possibilities with this library to obtain a good result: simple, adaptive thresholding and the Otsu method.

For the purposes of the project it was decided to use simple thresholding, i.e. the application of a very precise fixed threshold value, decided a priori through attempts to obtain the most useful value in the simulation in CARLA: all the pixels with RGB value above above this threshold they are raised to the maximum level, 255, i.e. white, while all values below the threshold value are brought back to the minimum value, 0, i.e. black.

The use of the ‘cv2.threshold()‘ function is well supported by the OpenCV library, giving the possibility of versatile management for different computer vision applications. Isolating relevant elements in CARLA’s simulated environment makes object detection easy [5].

The code of the submodule that has been developed is divided into several parts, in particular:

- `Importing libraries`: this script uses some libraries such as ‘os‘, ‘logging‘ and ‘cv2‘ of OpenCV in order to correctly manage the acquisition of files and

consequently of individual frames so as to be able to correctly process the images;

- `Logging setup`: during the development phase of this submodule it was necessary to carry out debugging phases, and for this reason the logging was configured to observe the flow of the application step by step in order to identify any errors;

- `Validation of input parameters`: after a first draft of the code it was also necessary to insert an input control area so that it reflects all the necessary parameters before the processing process is carried out, so that if the input is not correct in some points of view, see the format or otherwise, an error is reported even before proceeding with the rest of the application;

- `Application of thresholding`: the image is previously converted to grayscale as explained previously, and then a simple thresholding is applied, setting the threshold value to 205, a value chosen following several attempts which led to the choice of this value for the circumstances of the simulation and the need for required accuracy;

- `Preview and save image`: finally, if enabled, a series of images is made to appear which represent the original image, the grayscale one and finally the fully processed image with thresholding applied: once this is done, the last image is saved within a specific path, also thanks to one of the libraries initially mentioned.

The code used for the analysis is reported in the Appendix A.

Once the code had been developed, for this submodule and for the entire project, it was then necessary to insert a quality control phase of the same: in particular, within the software development, in order to ensure good quality of the code, Priority is given to readability, maintainability and in general to overall performance. After a careful analysis of the available tools, it was chosen to use Radon in order to measure the different metrics of the code. Radon fits into the software development landscape as a Python tool capable of providing quantitative and qualitative metrics for evaluating the quality of the source code, analyzing various aspects: these metrics make further analysis possible in order to understand the complexity of the code via numerical values [6].

The main metrics to refer to are:

- `Lines of Code (LOC)`: total number of lines within the code considered, including all possible diversities, empty lines, comments and pure code;
- `Logical Lines of Code (LLOC)`: number of lines of logical code, i.e. those lines of code within which an action is actually performed or a decision made within the program: empty lines and comments are therefore excluded, which do not they are part of the logic of the algorithm in question and therefore do not contribute to the real behavior of the program;
- `Source Lines of Code (SLOC)`: these are the lines of source code, i.e. those that can possibly also include lines of multiple declarations included on the same line: this is a value similar to the LLOC;
- `Comments`: textual lines of code where the behavior of the code is declared or simplified in words: a very important part of the code to make it easy to read for an external user who needs to understand how it works quickly and intuitively;
- `Percentage of comments`: total number of lines of comments compared to lines of total code, in order to provide a value of how much such code can be considered understandable.

The results obtained denote a code with a good density of comments aimed at understanding it and a good percentage of lines dedicated to documentation. The code appears compact with a good balance between lines of code and documentation.

Metric	Value
Lines of Code (LOC)	85
Logical Lines of Code (LLOC)	38
Source Lines of Code (SLOC)	39
Total Comments	7
Single Comments	7
Multi-line Comments	17
Blank Lines	22
C % L (Comments on LOC)	8%
C % S (Comments on SLOC)	18%
C + M % L (Comments + Multi-line on LOC)	28%

Table 3.2 Code metrics for the `frame_threshold.py` file

In conclusion, to validate the correct functioning of the submodule and to ensure that the chosen threshold could work, it was tested on a test image. This step was of fundamental importance to ensure the correct behavior of the submodule before applying it in CARLA. Below is an image, figure 3.4, showing the initial image on the left and the final image on the right with thresholding applied: this test highlights how the module is able to separate the areas of interest from the background.

Fig. 3.4 Before (left) and after (right) the application of the threshold, test image

Having seen the correct behavior of the module relating to thresholding, it was then applied in the CARLA simulation environment, obtaining an excellent result as shown in the image 3.5:

Fig. 3.5 Threshold applied in the CARLA simulation environment

3.4.3 Development and Implementation of Perspective Transformation Modules: The Case of eye_perspective.py

Following the correct use of the previous submodule, responsible for manipulating the image pixels, it was necessary to work further on it. In particular, in the field of autonomous driving, one of the main objectives is to have a clear and correct view of the road, through visual sensors, cameras, LIDAR, radar and other systems in order to obtain a way to correctly detect obstacles and interpret horizontal signs.

Nevertheless, the images that can be obtained through the cameras mounted on cars are subject to perspective distortions, i.e. the phenomenon that occurs when objects or figures at different distances from the camera appear with distorted dimensions or shapes. This emerged in the early testing phases of the project: the lines of the roadway, which are parallel, actually appear convergent or generally distorted, making it difficult if not impossible to correctly analyze their position.

To address this problem, in every Lane Detection project, it is necessary to apply an image manipulation technique, that is, the perspective transformation that makes it possible to obtain a view from above, or as it is more commonly called, a "bird's - eye view". This technique, completely artificial and based on mathematical - geometric principles, allows to correct perspective distortions, providing a vision free from such problems, more uniform, as if it allowed to flatten the image, returning a realistic view of the lines of the roadway, which return to being parallel and therefore allow a much more precise analysis of the position.

The perspective transformation technique to obtain a top-down image starts by identifying the key points of the starting image and their subsequent mapping onto a new position in the transformed image space. In practice, therefore, four points are chosen, which in the case of the project in question represent a trapezium, which represent the vertices corresponding to the visible road area: consequently, these points are then mapped onto a new image that represents the "artificial" top-down view that we want to obtain, where the vertices of the previous trapezium become the vertices of a rectangle that indicates the shape of the roadway seen without distortions.

This allows us to obtain excellent results in the context of autonomous driving as it allows us to solve problems such as in the case of curves or slopes, where the lines appear to converge.

The perspective transformation in order to obtain a view from above is therefore a first of all mathematical - geometric operation capable of taking a portion of a frame and simulating the point of view from above. This operation requires the use of different formulas and the knowledge of different relationships in particular between the coordinates of the points in the starting image and the coordinates of the "arrival" points, i.e. the image that you want to obtain.

The bird's - eye perspective view is represented by a homographic matrix **H**, i.e. a relationship between points of two spaces such that each point of a space corresponds to one and only one point of the second space, of dimensions 3 x 3. This matrix **H** represents the mapping between the coordinates of the points of the starting image and the "transformed" image. The relationship between these coordinates $\boldsymbol{x} = (x, y)$ in the original image and the corresponding coordinates $\boldsymbol{x'} = (x', y')$ in the artificially created one respects the following rule [7]:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \mathbf{H} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

where the matrix **H** is:

$$\mathbf{H} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix}$$

The new coordinates (x', y') can be expressed by the following equations obtained:

$$x' = \frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + h_{33}}$$

$$y' = \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + h_{33}}$$

In essence, the **H** matrix is obtained by computing error minimization techniques, using numerical optimization algorithms such as singular value decomposition (SVD) in order to ensure accuracy in the transformation.

This perspective transformation can be easily visualized through the OpenCV library: the official documentation [8] also shows some code fragments and the related results that can be obtained, figure 3.6, clearly showing the result of the operation. Below the specific calculation for the numbers and consequently the

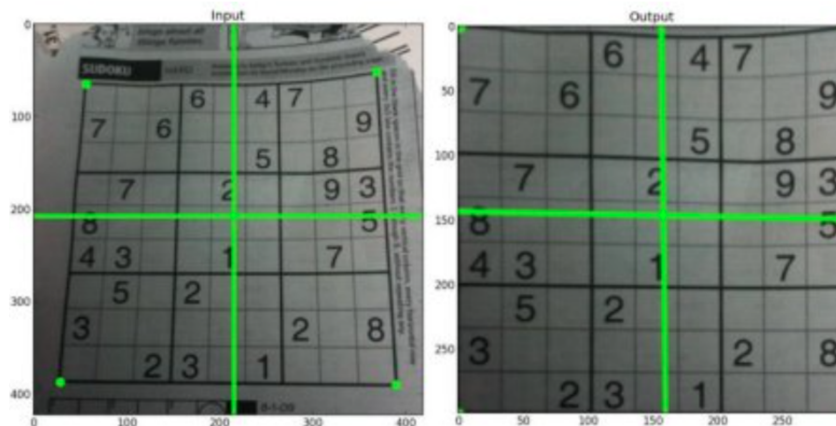


Fig. 3.6 Original image (left, input) and transformed image (right, output) of the perspective transformation

references relating to the project will be carried out, where in particular the values of the starting trapezium are:

$$(x_1, y_1) = (851, 590)$$
$$(x_2, y_2) = (1105, 590)$$
$$(x_3, y_3) = (300, 944)$$
$$(x_4, y_4) = (1690, 944)$$

While the rectangle in the transformed image will have the following values:

$$(x'_1, y'_1) = (50, -100)$$
$$(x'_2, y'_2) = (1870, -100)$$
$$(x'_3, y'_3) = (50, 1080)$$
$$(x'_4, y'_4) = (1870, 1080)$$

In order to obtain the perspective transformation matrix **H**, the following equations obtained from the homography are used:

$$x'_i = \frac{h_{11}x_i + h_{12}y_i + h_{13}}{h_{31}x_i + h_{32}y_i + h_{33}}$$

$$y'_i = \frac{h_{21}x_i + h_{22}y_i + h_{23}}{h_{31}x_i + h_{32}y_i + h_{33}}$$

For any pair of points corresponding to the original and transformed image, it is possible to write two equations: consequently, for the four points we can obtain eight equations that can be solved for the coefficients $\mathbf{h_{ij}}$ of the matrix **H**, thus writing the resulting equations as:

$$\begin{pmatrix} x'_1 & y'_1 & 1 & 0 & 0 & 0 & -x'_1x_1 & -x'_1y_1 & -x'_1 \\ 0 & 0 & 0 & x'_1 & y'_1 & 1 & -y'_1x_1 & -y'_1y_1 & -y'_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x'_4 & y'_4 & 1 & 0 & 0 & 0 & -x'_4x_4 & -x'_4y_4 & -x'_4 \\ 0 & 0 & 0 & x'_4 & y'_4 & 1 & -y'_4x_4 & -y'_4y_4 & -y'_4 \end{pmatrix} \begin{pmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

By solving the linear system we obtain the matrix **H** with which it is possible to transform the entire starting image obtaining the desired top view.

The Python submodule ‘eye_perspective.py‘ is therefore developed in such a way as to apply a perspective transformation of the starting image, thus obtaining a view from above to correct the distortions due to the perspective characteristic of frontal shots such as that of the project in question.

This operation is essential to obtain a high accuracy in the recognition of the roadway and horizontal signs.

The code in particular is made up of different functions and classes, each with a very specific task: the main class is called ‘PerspectiveTransformParameters‘ and is responsible for managing the numerical parameters useful for the transformation, such as the source points (relative to the starting image) and the destination points

(relative to the output image). Furthermore, this class is responsible for saving the points and the information relating to the radius of the circles drawn on the source points, useful for debugging functions or visual purposes in general.

Furthermore, following several approaches attempted during the development phase, it was necessary to implement support functions, such as ‘validate_points‘ and ‘validate_and_adjust_detection_points‘, in order to ensure that the transformation was applied to points that lie on white pixels, that is, that with a high probability correspond to the lines of the garage, or that on the contrary are corrected in such a way as to satisfy certain application conditions.

The main function is then ‘apply_eye_perspective_transform‘, that is, the function that manages the perspective transformation: after having validated the points, or if necessary moved them and saved the new positions, the function calculates the transformation matrix that during the theoretical part was called **H** , and consequently uses it to return the output image.

The submodule is presented in its entirety in the Appendix B.

Below is a more detailed explanation of the individual functions that make up the submodule:

- `PerspectiveTransformParameters`: is a class that is part of the "data class" category, functional to group the numerical parameters useful for the geometric transformation. It includes in particular the coordinates of the source and destination points, that is, the vertices of the trapezium of the original image and the vertices of the rectangle of the transformed image; it also includes the settings for drawing the circles of the source points, the adjustment limits for the positioning of the points based on the detection of white pixels;
- `is_white_pixel(frame, point)`: this function is responsible for checking whether a given point, i.e., the single vertex of the trapezium, corresponds to a white pixel: this operation is essential for detecting the roadway with precision and then applying the transformation;
- `find_nearest_white_pixel(frame, point, max_distance)`: placed after the `is_white_pixel` function when it returns "false", in order to find the nearest white pixel along the x-axis, but within a specific range of pixels. The source points are then corrected, where necessary;

- `validate_point(...)`: this function first checks and then adjusts the single point where necessary, moving for the correction so that it does not exceed the threshold specified in the data class: it also checks that the eventual moved point does not violate the geometric constraints defined, as happens for the internal trapezium;

- `validate_and_adjust_detection_points(frame, params)`: this function applies the validation and adjustment feature to all four points, i.e. the upper and lower vertices;

- `apply_eye_perspective_transform(frame, params)`: it is the main function of the submodule, because first of all, using the other functions mentioned above, it validates and adjusts the source points, draws circles on these points to help the visualization and finally applies the geometric transformation. The final result is an image that represents the view from above, in an artificial way but that respects the geometric rules in order to perform a correct search for the positions of the lines.

After the development and testing phase of the ‘eye_perspective_view‘ submodule, it was important to perform a code quality check phase, in order to further understand whether or not this part of the project respects the practices in terms of readability, maintainability and overall performance. Similarly to what was done for the image thresholding module, Radon was used to measure different code metrics, in order to provide important information on the code quality. These values therefore denote a

Metric	Value
Lines of Code (LOC)	321
Logical Lines of Code (LLOC)	122
Source Lines of Code (SLOC)	156
Total Comments	24
Single-line Comments	21
Multi-line Comments	80
Blank Lines	64
C % L (Comments on LOC)	7%
C % S (Comments on SLOC)	15%
C + M % L (Comments + Multi-line on LOC)	32%

Table 3.3 Code metrics for the `eye_perspective.py` file

moderately sized structure, with a fair amount of lines dedicated to the system logic, while maintaining a good part dedicated to declarations or in general to structures that do not contribute directly to the operational logic of the program.

Finally, the presence of 80 multi-line comments suggests that certain parts of the code could be complex and that consequently it was necessary to introduce more in-depth explanations. The 32% of LOCs that contain detailed comments indicates a good attention to making the code understandable and maintainable, as well as functional.

After having implemented and carefully tested the module, it was possible to obtain a precise perspective transformation of the individual frames acquired in real time during the execution of the application. The final result is therefore a view from above, capable of correcting perspective distortions and making the analysis of the roadway lines much more accurate. This transformation has proven useful for improving the precision of the system, allowing a realistic and correct perception of the road lane. Below, figure 3.7, is a visual example in the CARLA simulation environment, where it is possible to notice how the roadway is faithfully represented, with parallel and no longer converging lines.

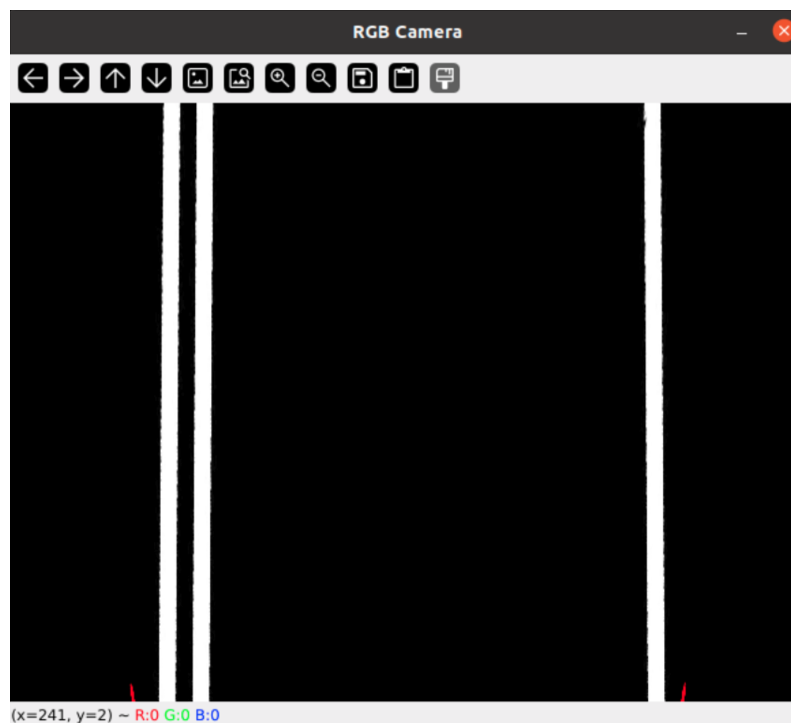


Fig. 3.7 Result of the view from above

Chapter 4

Analysis of Implementation Strategies for Lane Detection

4.1 Introduction to the attempts made

Lane Detection within the autonomous driving panorama represents an aspect of fundamental importance in modern driver assistance systems (ADAS). The primary objective is to be able to precisely recognize the lines that divide the car's lane from the other lanes in the opposite direction or from the end of the same road. This aspect is therefore very important also with regard to the consequent automatic steering maneuvers.

The right precision in lane recognition becomes more difficult in critical road conditions, such as in the presence of curves, intersections or in cases of poorly defined road signs. For these reasons, it is necessary to develop a robust algorithm capable of improving vehicle safety, minimizing the risk of error.

In addition to the previous and already mentioned difficulties, other critical situations may be poor visibility, the presence of obstacles on the road and the natural or otherwise degradation of the lane lines, which may hinder or even prevent correct road recognition.

After implementing the previous modules related first to the thresholding applied to real images and then to the perspective transformation, the next step was completely dedicated to the development of the real algorithm capable of recognizing

the position of the lines, as the other two modules were used as support. This implementation was done directly inside the client, inside a very specific callback function.

The development process encountered several problems, solved step by step in order to obtain the best performance, but going by trial and error, as the choice was to apply a slightly different approach compared to the pre-existing one in the documentation. The main limits that were encountered were due to the need and the desire to perform an algorithm capable of satisfying the performance requests especially in 'chaotic' conditions such as those of the city, and not only on the highway where the conditions are simpler.

The preliminary attempts, however, allowed us to collect important data and above all to develop experience and critical sense towards the problems encountered, guiding the project towards better developments and optimizations, thus leading to the solution then definitively adopted.

4.2 First Attempt: Lane Detection Using Horizontal Segmentation

The focus of this first attempt was to obtain a method of lane recognition in such a way as to identify the white pixels corresponding to the lane demarcation lines, all starting from the image previously processed with the submodules mentioned above.

The method is mainly based on the use of horizontal scan lines, traced at non-regular intervals, but rather increasing and increasingly dense as you go towards the top of the image, with the aim of intersecting the lane lines.

The method, in the first phase, involves outlining a series of horizontal lines in such a way as to compress the entire image along its height, giving particular importance to the upper part of the same because it tends to be more critical.

Once traced, the algorithm has the aim of scanning each line from one side of the image to the other, marking with a red circle each time it is found.

The choice behind the use of horizontal lines is dictated by the mixed simplicity and effectiveness of this approach, aimed at reducing the complexity of the research to a one-dimensional case only.

Once all the white pixels have been collected and identified, the next step involves saving the position of these points and consequently creating a polygon capable of enclosing all these points and consequently identifying the lane as precisely as possible.

Once the method described above was implemented, it was tested first in a static situation, i.e. by giving an image as input and not a series of frames from the simulation.

The resulting image, at first, shows a good ability to identify the points along the lanes, making it possible to build the polygon.

The result shown in the figure 4.1 shows a good starting point for the detection of the lines, offering a first indication of the potential effectiveness of the method. The following image illustrates the first results obtained with this approach, highlighting in red the points where white pixels were recognized along the green lines:

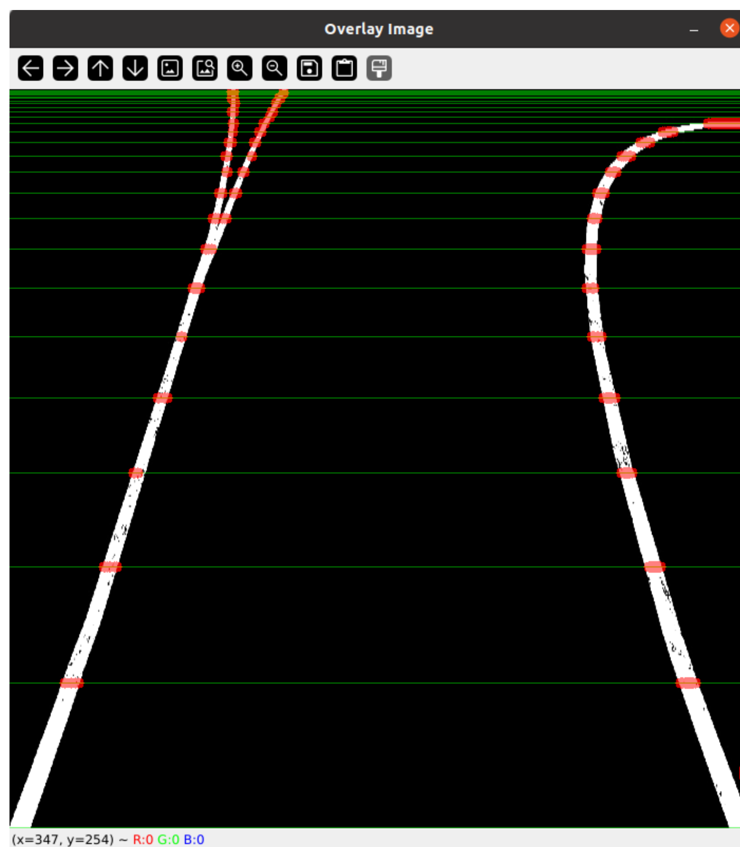


Fig. 4.1 Detection result by scanning horizontal lines

Once the positions of the white pixels along the horizontal lines have been found and marked, the next step involves first building curves capable of interpolating the different points found and consequently finding a way to build a continuous and geometrically correct polygon in order to identify the roadway, in such a way as to provide a realistic and usable representation. The first attempt involved the use of a refined interpolation technique based on the Root Mean Square Error (RMSE). Interpolation is the calculation by which, according to a certain law, values of a function are determined within an interval in which only some of them are known [9]. The RMSE method is defined as a statistical measure used in order to quantify the difference between the observed values and those instead expected. In particular, it is defined by the following formula:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

where:

- n number of total observations;
- y_i observed values;
- $\hat{y}_i$ values predicted by the fitted curve.

In particular, the RMSE method is used to provide a measure of how accurate the overall system of the interpolated model is, thus indicating how much the values are distant from the real values: in this project it was used to minimize the error, so that the interpolated curve best fits the points found.

Nevertheless, the use of the RMSE presented some problems: this method tends to detect the error in a accumulated manner, but hides smaller and more local errors. Unfortunately, when this behavior occurs in more complex situations, in which perhaps a sudden change of direction must be managed, or a difficult situation of learning the lanes, it becomes difficult to obtain good performance and therefore obtain precise curves that are able to respect the road lanes.

In our project, therefore, the RMSE did not appear to be a method accurate enough to be used. After trying to apply the RMSE interpolation method described above to build the curves, the results shown in the following images were obtained:

in particular, they show the interpolation of the points found starting from the image presented in the figure 4.1, and the construction of the internal polygon.

In the first image, figure 4.2, the interpolation curves are shown, without the internal polygon, so as to be able to identify the problems right from the starting point: in fact, when the points recognized on the same horizontal become multiple, the model begins to give problems, trying to make an average, but with poor performance results.

In the second image, figure 4.3, the result is shown with the internal polygon highlighted: this geometric figure was created by joining the ends of the interpolation curves of the right and left lane, and only then filled inside. However, the quality of the result was definitely below expectations.

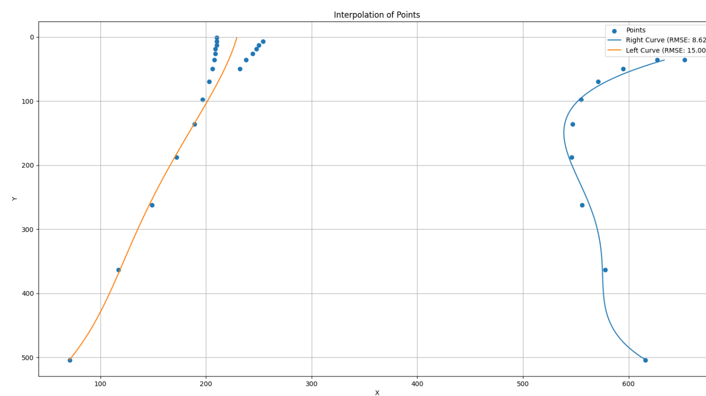


Fig. 4.2 Interpolation with the curve

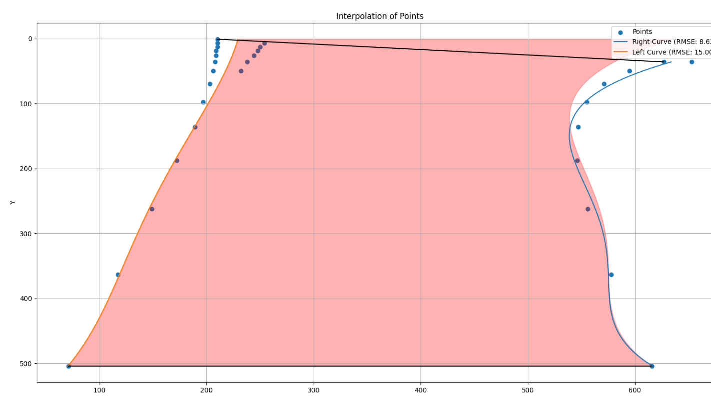


Fig. 4.3 Interpolation with the polygon

4.3 Second attempt: Lane Detection using the continuity of the white pixels

After having analyzed and thought about a possible solution to the problems given by the RMSE method, we tried to explore a different approach, thinking concretely about the proposed situation.

This new method explores the possibility of searching, similarly to before, starting from the center and going to the right and left the first series of thirteen consecutive white pixels, followed and preceded by black pixels: the choice of thirteen pixels was made following a visual analysis of the lines of the roadway.

The principle on which this method is based is therefore to use the presence of continuous white pixels, in order to identify in a more robust way only the lines of the lane and exploiting the factors that do not change during the simulation, that is, that the camera set on the machine almost always points to the center of the lane.

Once a consecutive series of white pixels is found, the median point of the thirteen points found is identified with a red dot.

Below, in the figure 4.4, a graphic result of what is obtained is shown: Although the new method related to the search for thirteen pixels may seem like a good upgrade compared to the previous method, this procedure presents the possibility of encountering errors in the detection of the lane, as can be seen in the upper part of the figure 4.4.

A first important aspect is that the number thirteen chosen is not always ideal: in some parts of the detection, the lines of the roadway may not be continuous or "full", thus hindering this search. The algorithm therefore, not correctly finding a series of thirteen white pixels, risks selecting incorrect points.

This aspect in fact occurs in the upper part of the figure 4.4, where a segment outside the lane is incorrectly detected, taking a part of the guard rail. Since the resolution of the image is lower in the upper part of the image, it is a problem that can occur several times.

These aspects presented therefore underline how the choice of a fixed number of consecutive pixels may not be the best choice.

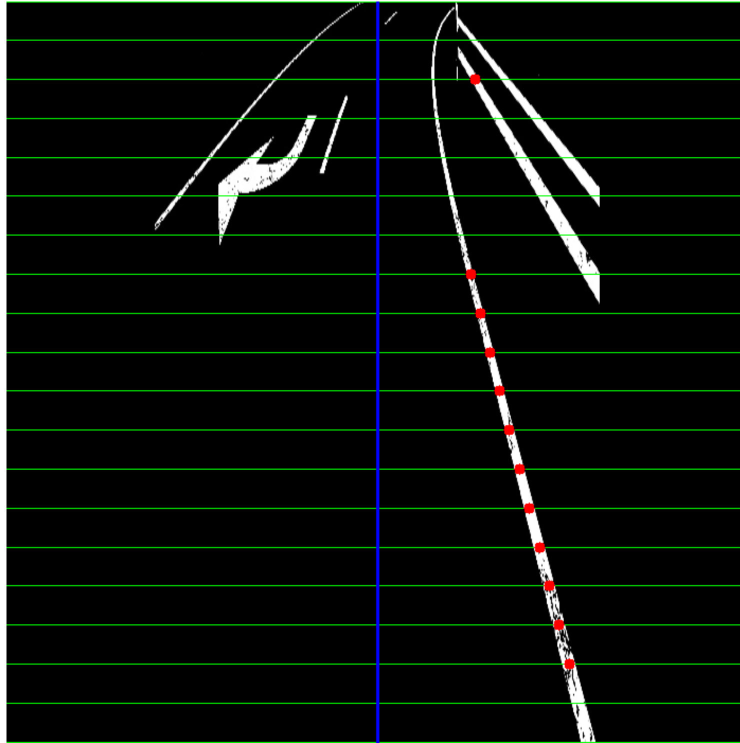


Fig. 4.4 Interpolation with the new method

Having therefore explored two different possibilities and having thoroughly analyzed the strengths and weaknesses of both solutions, an alternative method was sought that could on the one hand address the critical issues that arose during the development of the two previous versions. In particular, during this phase of the project, an attempt was made to address with particular dedication the problems relating to the most complex situations, such as those on curves or in situations where horizontal signs are present.

4.4 Ultimate Choice for Lane Detection

After trying several attempts and different approaches to obtain the best results for Lane Detection, it was decided not to continue with the development of a separate module, as was done for the previously explained modules: the logic for lane recognition was inserted within a function, in order to manage both the acquisition of images and their processing, which come from the camera mounted on board the simulated vehicle.

This choice was made for several reasons, both practical and technical, which then proved to be successful for the purposes of the project.

The main reasons are:

- `Code efficiency:` the complexity of the code has been significantly reduced: when complicated functions such as obtaining images and processing them are performed separately, the complexity of the algorithm itself is added to the complexity of the communication between the different modules.

  The interfaces to maintain therefore increase significantly, making the system more prone to bugs or errors of all kinds, such as those of synchronization or data sharing.

  The choice of opting for a single workflow within the `‘camera_callback‘` function simplifies the logic, reduces the risk of bugs and the code itself becomes more readable and consequential.

- `Resource optimization:` since the perception algorithm for the Lane detection is a process within a large world such as that of autonomous driving, the demand for rapid and continuous processing of visual data is very large.

  The reactivity of the program is a factor of fundamental importance, and the more you can make it lean, the greater the performance obtained. One way to increase the optimization of the program is precisely to manage these aspects within the single function.

  Access to specific hardware, memory management and other aspects of a total modular approach could further weigh down the program.

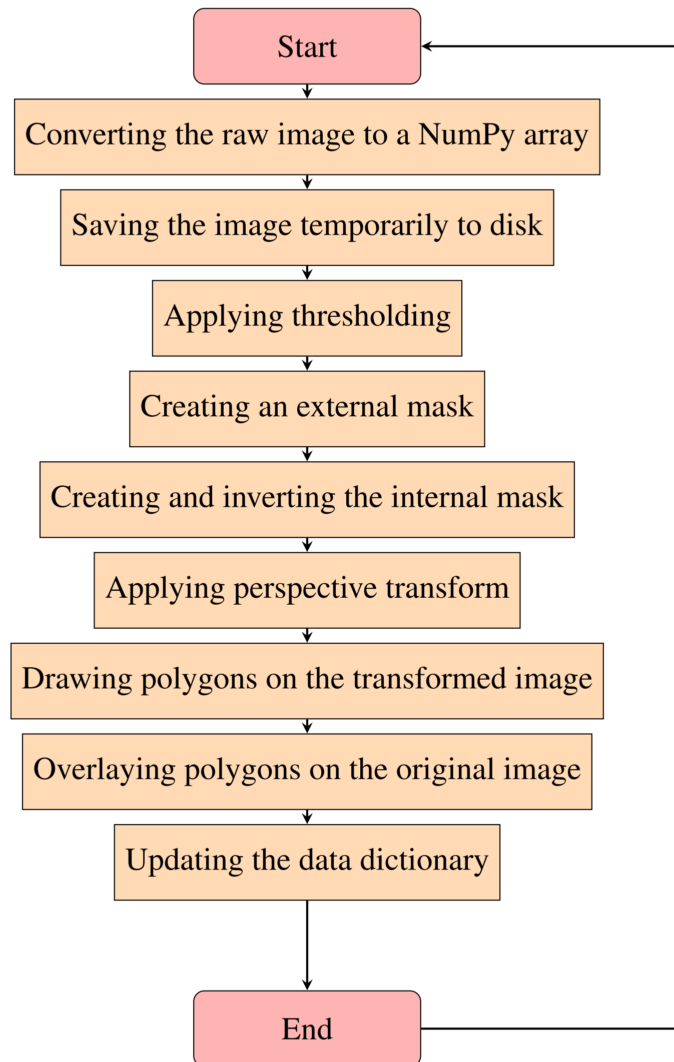
  The result is therefore a substantial reduction in processing times and a consequent freeing up of resources that can be allocated to other applications.

- `General benefits:` putting the lane detection logic directly into the `camera_callback` function inside the client not only improves the efficiency and accuracy of the program, but also facilitates a greater internal structure of the code, making it easier to control the development of the entire processing process.

4.4.1 Implementing of the camera_callback() function

The final ‘camera_callback‘ function, reported inside the Appendix C , outlines the brain of the chosen approach: in particular it is crucial to manage the obtaining of the images that come from the camera, first by applying the thresholding module, then obtaining the top perspective of the same through the ‘eye_perspective‘ module, and then applying two different masks to correctly confine the areas of interest.

The function code is made up of several parts, which are essential to obtain an excellent result at the end of it, both in terms of performance and visual result. Below is a flow chart representing the main steps of the ‘camera_callback‘ function:



The `‘camera_callback‘` function is responsible for the heart of Lane Detection. The following is a detailed explanation of the main actions performed by this function:

- `Primitive image conversion`: the first step carried out by the function is basic but fundamental, as it must convert the image taken by the camera into a NumPy array: this step is essential as the NumPy library provides several tools for image processing, quickly and effectively. The method used is `‘reshape‘`, capable of ordering the raw data effectively for subsequent manipulations;
- `Temporary saving of images`: in order to process the images in a simple and effective way, it was necessary to save them on disk: in order not to burden the memory too much and above all because they are not useful for the project, the same images are then deleted once used. This process is useful and prior to the thresholding process;
- `Application of the threshold module`: the next step involves using the thresholding module, explained in the chapter 3.4.2: by separating the pixels of interest for the project, the image is converted into a binary version, obtaining with the black pixels the background and what is not interesting in the frame, and in white everything that needs to be worked on;
- `Definition and creation of masks`: two masks are created, an external one that limits the area of interest, reducing the noise so as to focus only on the areas of interest, and an internal mask instead that serves to not consider the horizontal road lane signs or to avoid disturbances during medium-radius curves. This step serves to focus only on the areas of interest;
- `Application of the top view module`: the ‘eye_perspective‘ module, explained in the chapter 3.4.3, is used in order to obtain the top view;
- `Polygon representation`: the function ‘draw_polygon‘ draws the polygons, taking the source points of the perspective transformation as reference. This step is used to obtain a graphical view of what is happening during the analysis;
- `Transparent polygon overlay`: This part of the code is responsible for the final result, as it creates a transparent polygon to overlay the original,

colored image. The OpenCV library function ‘cv2.fillPoly‘ is used to draw the filled polygon, and then via the same library function ‘cv2.addWeighted‘, the union with the starting image is performed, thus creating a transparent effect of the recognized lane;

- `Image data update:` As a final step, the processed images are saved inside a data dictionary, a particular Python structure, useful for the real-time visualization needed later.

In order to be more clear about the creation and use of the mask during the running of the `‘camera_callback‘` function, we divide this operation into several steps, each of which is responsible for a specific action.

1. Defining Mask Functionality

Masks used through the OpenCV library are used in the treatment of images to separate or highlight certain areas of interest within a frame. In a binary mask in particular, the belonging pixels are set to a certain value, usually white, while all the other pixels are set to another value, usually 0 to represent black.

The isolation of objects, the removal of backgrounds or as in the case of the project to delimit the reasons of interest (ROI) for subsequent treatment. Within the project aimed at Lane Detection two masks are used:

- `External mask:` this tool was used to delimit the area of interest of the image, excluding everything that does not include the lane, therefore cars, pedestrians or anything outside the road that could make it difficult to obtain good results.

  All pixels outside the mask are therefore set to a value equal to 0, that is, they are black pixels that are not considered. Through this strategy, the noise of the image is reduced and we can focus only on the roadway.

- `Internal mask:` this tool is used in an opposite or complementary way to the previous one; the values inside the mask are inverted, that is, the road lane is further isolated, making the pixels inside the mask black, so as not to consider any horizontal road signs present.

  The combination with the external mask means that the result is the activation of the pixels present between one mask and the other, with the exclusion of the others.

2. Application within the code of masks:

The main function used during these operations is taken from the OpenCV library and is ‘cv2.bitwise_and‘:

- `Applying the outer mask:` bitwise AND operation between the thresholded image and the created mask; this step is able to filter the image, so that only the pixels inside the mask are kept;
- `Applying the inverted inner mask:` after having also created the inner mask, inverted so as to operate with the opposite behavior, a bitwise AND is used again in order to obtain the final image as required: the pixels inside the inner mask are no longer considered;
- `Advantages in choosing this approach:` the reduction of image noise through the use of these masks represents one of the main focuses, as it greatly facilitates the application of the algorithm. In particular, the internal mask becomes very important as it is able to avoid the choice of points inside the lane, which can confuse the algorithm.

3. Choice of coordinates for masks:

The evaluation of the coordinates for the masks is the focal point of the project since the success of the lane recognition depends on it. The ‘eye_perspective‘ module in particular also has the purpose of managing these aspects through some functions already explained in the dedicated paragraph, which will now be slightly revisited to explain how they were used.

In a first moment the source and destination points are defined within the transformation parameters class: these points were chosen following several practical attempts.

The ‘validate_and_adjust_detection_points‘ function verifies and adjusts the points thanks to the ‘is_white_pixel‘ and ‘find_nearest_white_pixel‘ functions, which make sure that the chosen points lie on white pixels, i.e. with good probability on lane lines. If they are not on a white pixel, they are moved looking for one.

The usefulness of this approach lies in its flexibility: the coordinates of the points are dynamic, that is, they adapt to the conditions of the application in order to be as

precise as possible. This flexibility makes the algorithm further scalable on several different work planes.

The inclusion of some constraints such as ‘`top_left_internal_trapezium`‘ and ‘`top_right_internal_trapezium`‘ give the possibility to further elevate the control over these points, ensuring that they are positioned in the correct place.

4. Code snippet for creating masks:

For greater clarity, the code fragment intended for creating the masks is inserted below:

```
    # Create external mask
        external_mask = create_mask(
            thresholded_image.shape[:2], thresholding_settings.
                extern_mask_points
        )
        masked_image = cv2.bitwise_and(
            thresholded_image, thresholded_image, mask=
                external_mask
        )

        # Create internal mask
        internal_mask = create_mask(
            masked_image.shape[:2], thresholding_settings.
                intern_mask_points
        )
        inverted_internal_mask = cv2.bitwise_not(internal_mask)

        masked_image = cv2.bitwise_and(
            masked_image, masked_image, mask=
                inverted_internal_mask
        )

        # Apply the eye perspective
        eye_perspective_image, source_points =
            apply_eye_perspective_transform(
            masked_image
        )
```

Listing 4.1 mask code

5. Results and conclusions:

In conclusion, this part of the code within the analyzed function is of fundamental importance as it ensures that the control and analysis areas of the lane are only those of interest: without these two masks, the algorithm would risk analyzing and consequently considering external elements as lanes, as was noted during the use of the previous methods that were then discarded.

Below, in figure 4.5, are some screenshots of the real-time simulation of the application, in which the aspects described above can be visually noted.

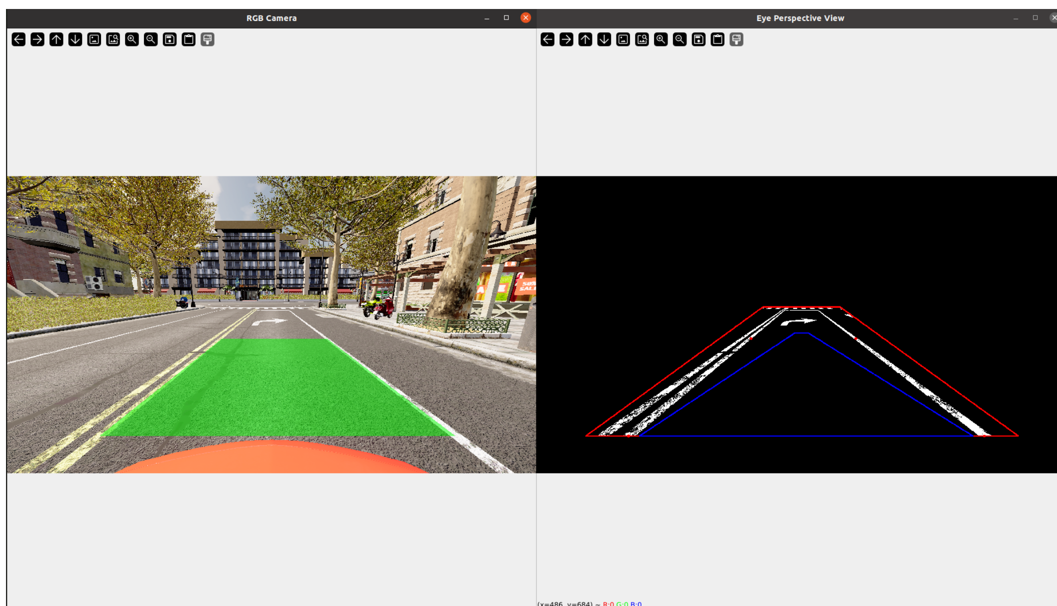


Fig. 4.5 Results of the ‘`camera_callback`‘ function: the final image on the left, the functional polygons on the right

The figure 4.5 shows the results obtained during the running of the application, in which a frame was captured by the camera mounted on the vehicle.

On the left, the original RGB image taken from the front camera is represented, to which no transformation or processing was applied. The only thing that was done was to add in green the area of the polygon, which represents the lane detected during the execution of the application. This colored region is therefore the result of the application of the two different masks and the other functions described above.

On the right of the figure 4.5, instead, the image functional to achieving the required result is visible: in this frame, in fact, both the application of the thresholding

are clearly visible, as every detail not necessary for the Lane Detection is put in black, while the rest is in white, and in addition the masks are visible: the external mask is represented in red, while the internal mask is visible in blue.

In the figure on the right, there are also dynamic red points, which are the real responsible for the Lane Detection as they are free to move along the x-axis when, during the running of the application, they are not on white pixels, and therefore presumably on a road lane.

In summary, the image on the right visually demonstrates how the accurate use of masks, after an accurate choice of the points with which to form them, allows to achieve a clear and precise vision of the roadway.

This final representation highlights how the method chosen in the end is able to obtain good results in terms of performance.

Chapter 5

Client Code Implementation and Description

5.1 Introduction to ‘`client.py`‘

The ‘`client.py`‘ code represents the core around which the entire perception algorithm for Lane Detection revolves within the simulation system, as it is the interface responsible for connecting the user with the virtual world provided and managed by CARLA.

The code in particular governs the creation of the world, the background vehicles and the vehicle protagonist of the project, controlling it from different points of view, including the configuration of the camera mounted on board and the consequent processing of the images obtained, in order to obtain a realistic simulation of autonomous driving.

First of all, the client takes care of initializing the world where the simulation will be performed, thus allowing the relationship between the different actors of the world, such as vehicles, pedestrians and cameras/sensors.

The method in this phase is modular, in order to allow precise management of each factor of the simulation, giving the possibility of managing and combining each specification of each actor present during the running of the application.

Once the simulation's peripheral aspects have been managed, the client is responsible for the proper functioning of the RGB camera mounted in the center of

the vehicle taken into consideration, in order to obtain images of the surrounding environment on which it is possible to work. The image processing is done in real time with a series of consequential operations, such as thresholding, perspective transformation and the consequent calculations related to the algorithm itself, all fundamental elements in order to obtain good performance for the project.

Another important aspect of client management concerns the surrounding environment, as even the extra simulation vehicles are managed within the same code.

The vehicle protagonist of the experiment is managed via a PID controller (acronym for Proportional Integral Derivative), capable of autonomously driving the car by simulating its behavior in autonomous driving.

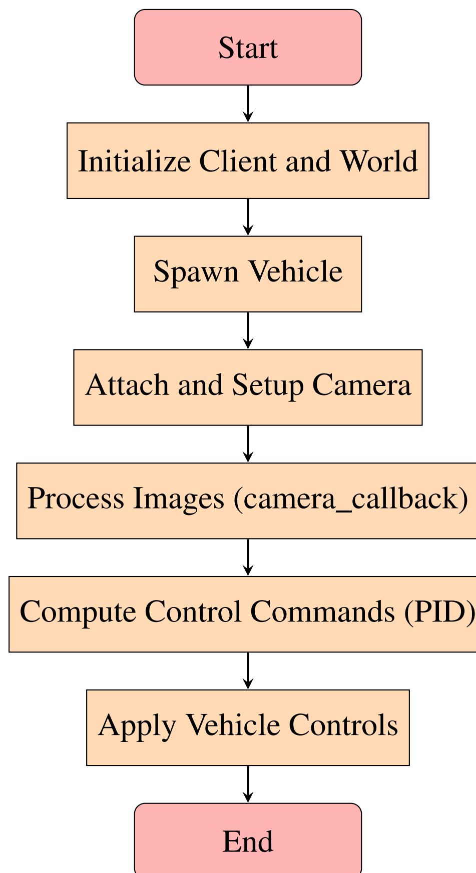
In conclusion, the ‘`client.py`‘ code provides the interaction interface with CARLA, processes the images, controls the vehicle and generally simulates different complex scenarios such as traffic: all this is done in order to obtain a representation of reality as truthful as possible.

5.2 General structure of the code

The client code ‘`client.py`‘ has been structured in a modular way in order to obtain a more linear form of the application and follows a logical structure aimed at the good maintenance of the project itself.

The code is divided into functions and classes responsible for the success of the project, in such a way as to divide the responsibilities on different fronts, making the code readable and easy to manage in case of debugging needs.

The complete code is presented in the Appendix D, while a flow chart useful for understanding its behavior at a visual level is shown below.



1. Importing Modules and Libraries

In a first part, as is done for every code in every programming language, the first lines of code are dedicated to importing standard and external Python modules, as well as functional libraries for the purpose.

In this case, the modules ‘sys‘, ‘random‘, ‘time‘ and ‘tempfile‘ were important and were used for the management of the system in general, to optimize temporary files.

The important external modules are instead ‘carla‘, the main protagonist of the project as it allows the relationship with the simulator of the same name; then the module ‘cv2‘ (OpenCV) is imported, important for working on the images obtained from the camera.

Finally, specific functions are imported from two modules built externally for the success of the project: in particular, they are ‘apply_threshold_and_save_image‘

coming from the module ‘frame_threshold‘ and ‘apply_eye_perspective_transform‘ coming from the module ‘eye_perspective‘.

2. Determining Classes for the Project

Secondly, the two main classes of the project are defined, one aimed at defining the values related to the RGB camera setting (‘CameraSettings‘), while the other class is responsible for defining the settings related to the thresholding of the images (‘ThresholdingSettings‘).

These classes have been introduced to facilitate the management of the values inside them, so that if they need to be changed or monitored they are easily identifiable: furthermore, the use of the ‘@dataclass‘ decorator makes everything even simpler, removing any doubts from the reader regarding some numbers that otherwise might seem random.

3. List of main functions

The most important functions of the perception project for Lane Detection are as follows:

- `‘initialize_client()‘`: responsible for starting the relationship between the client and the CARLA simulation server, initializing the world and retrieving the blueprint of the actors in play through the CARLA API during the running of the application;
- `‘spawn_vehicle()‘`: controls the creation and precise positioning of a vehicle within the newly created virtual world; checks are applied to ensure that the car is successfully created in the desired position;
- `‘spawn_camera‘`: creation and positioning of the RGB camera on the newly created vehicle; the specifics are detailed inside the ‘CameraSettings‘ class we talked about above;
- `‘camera_callback‘`: function for managing and manipulating images coming from the RGB camera where the final result of the project is processed: this is discussed in more detail in the chapter 4.4.1;
- `‘setup_camera()‘`: definition of the camera and the callback function for image acquisition, also creating the two windows for viewing the real-time simulation video.

4. Main Client Loop

The main loop of the ‘main()‘ function manages all the different aspects of the simulation and is therefore responsible for the aggregation of the different modules that are part of the project. It therefore manages, in addition to the continuous execution of the client itself, the vehicle through a PID controller and also the updating of the images obtained from the camera.

It also manages the cases of exceptions or code malfunctions in order to avoid unjustified interruptions of the application.

5.3 Accurate description of the behavior of the ‘main()‘ function

1. Initialization and configuration of key elements

First, two variables are initialized: ‘actor_list‘, which is an empty list that will be filled with the simulation protagonists, such as vehicles and the camera, and ‘ego_cam‘, which will be considered for the RGB camera attached to the vehicle. Using the list of actors makes it easier to handle them when they will have to be destroyed at the end of the simulation.

The simulation seed is then stopped, using the string ‘random.seed(500)‘, in order to set the generation of random numbers. This part is of fundamental importance as regards the need to have repeatability in the simulation for testing and debugging phases.

The ‘initialize_client()‘ function that we talked about before is then called, which is responsible for the connection with the CARLA server: in particular, it reports three elements: the client for communication with the server, the simulated world and the blueprint library because it contains the models of the actors present during the running of the application.

After the client setup and consequently the world setup, the vehicle responsible for the simulation is inserted through the ‘spawn_vehicle‘ function, which will be placed inside the list we talked about above.

The RGB camera is created and set thanks to the ‘spawn_camera()‘ function and then configured thanks to the ‘setup_camera()‘ function: as you can imagine this part is fundamental for the success of the project and the simulation. With the association with the other fundamental function, ‘camera_callback‘, the real-time viewing windows are created. Here is this code snippet:

```
actor_list = []
ego_cam = None

random.seed(500)

# Initialize the client, world, and blueprint library
client, world, blueprint_library = initialize_client()

# Spawn the vehicle
vehicle = spawn_vehicle(world, actor_list)

# Define transform
transform = vehicle.get_transform()

# Spawn attached RGB camera
ego_cam, cam_bp = spawn_camera(world, vehicle)

# Set up the camera
camera_data = {}
setup_camera(ego_cam, cam_bp, camera_data)
```

Listing 5.1 mask code

2. Defining PID Controller Gains

This code snippet specifies the parameters for the PID controller, the controller chosen to manage autonomous driving, in order to regulate the speed and direction of the vehicle accurately.

The parameters for lateral control are as follows:

- K_P (Proportional): gain set to ‘1.0‘, i.e. a correction applied that is only proportional: this means that if, during the simulation, the machine deviates slightly from the chosen trajectory, this controller applies a proportional correction to manage this error;

- K_I (integral): integral gain set to ‘0.0‘, i.e. this effect is not used because it is not necessary and above all to avoid unwanted oscillations;
- K_D (derivative): similarly to the previous gain, here too it is set to ‘0.0‘, i.e. here too it is not used, because it is not necessary to dampen the response.

As for the longitudinal gains, i.e. acceleration and deceleration, they are set in a similar way to the previous ones, with the K_P equal to ‘1.0‘, while the other two are set to ‘0.0‘: these values mean that the controller is pure proportional, without integrative and derivative effects.

The controller is therefore defined as fast and reactive.

Once the gains are defined, for the purpose of software development, an instance of the PID controller is created with the class ‘VehiclePIDController‘. The result obtained is therefore a predefined trajectory with the speed also set a priori and maintained.

Below is shown this part of the ‘main()‘ code:

```
1      # Define the proportional (K_P), integral (K_I),
2      # and derivative (K_D) gains for lateral control
3      args_lateral = {"K_P": 1.0, "K_I": 0.0, "K_D": 0.0}
4
5      # Define the proportional (K_P), integral (K_I),
6      # and derivative (K_D) gains for longitudinal control
7      args_longitudinal = {"K_P": 1.0, "K_I": 0.0, "K_D": 0.0}
8
9      # Create a PID controller instance for the vehicle
10     # using the lateral and longitudinal control gains
11     pid_controller = VehiclePIDController(vehicle, args_lateral
           , args_longitudinal)
```

Listing 5.2 mask code

3. Main loop and vehicle control

A so-called infinite loop, ‘while True‘, is started, which continues the application running until it is manually stopped. Inside this loop the entire control of the vehicle during the simulation is managed. First inside the loop the function ‘get_target_waypoint()‘ is launched to set the next reference point for the PID controller and to set the safe and practicable navigation speed. Through the function ‘run_step()‘ the PID controller establishes the proportional gains to keep the vehicle along the correct trajectory and with the right speed, always maintaining the safety of the machine.

Once the control command is calculated, it is applied through the function ‘apply_control()‘, which is then able to adjust the trajectory and other parameters if necessary. During the main execution, the images inside the two windows are updated through the function, shown thanks to the function ‘cv2.imshow()‘: this step is fundamental as it allows the step by step vision of the success of the algorithm. To end the application, a logic was then inserted that closes all the display windows when the ‘q‘ key is pressed.

Finally, the exception management is the last aspect covered, in order to control possible errors or any exceptions. Ultimately, all the actors that were created during the simulation are destroyed in order to be able to possibly run another one later. The code relating to this part just described is shown below.

```
    try:

        while True:

            # Get the next waypoint and target speed for the
               vehicle using the defined function
            waypoint, target_speed = get_target_waypoint(
               vehicle, world)

            # Compute the control command for the vehicle based
                on
            # the target speed and waypoint using the PID
               controller
            control = pid_controller.run_step(target_speed,
               waypoint)

            # Apply the computed control command to the vehicle
            vehicle.apply_control(control)

```

```
            cv2.imshow("RGB Camera", camera_data["image"])
            cv2.imshow("Eye Perspective View", camera_data["eye_perspective_image"])

            if cv2.waitKey(1) == ord("q"):  # Close windows when press ‘q‘
                break

        cv2.destroyAllWindows()

        # Add a few more vehicles to the simulation
        transform.location += carla.Location(x=40, y=-3.2)
        transform.rotation.yaw = -180.0
        for _ in range(0, 10):
            transform.location.x += 8.0

            bp = random.choice(blueprint_library.filter("vehicle"))

            # Use try_spawn_actor. If the spot is occupied by another object,
            # the function will return None.
            npc = world.try_spawn_actor(bp, transform)
            if npc is not None:
                actor_list.append(npc)
                npc.set_autopilot(True)
                logging.info("created %s", npc.type_id)

        time.sleep(5)

    except (carla.ServerError, carla.ClientError, carla.RPCError) as e:
        logging.error("Carla error occurred: %s", e)
    except UnexpectedError as e:
        logging.error("An unexpected error occurred: %s", e)

    finally:
        logging.info("destroying actors")
        if ego_cam is not None:
            ego_cam.destroy()
        client.apply_batch([carla.command.DestroyActor(x) for x in actor_list])
        logging.info("done.")
```

Listing 5.3 mask code

4. Final analysis of the client code goodness

After the development and testing phase of the ‘client.py‘ code, it was important to perform a code quality check phase, in order to further understand whether or not this part of the project respects the practices in terms of readability, maintainability and overall performance. Similarly to what was done for the image thresholding module and for the eye_perspective module, Radon was used to measure different code metrics, in order to provide important information on the code quality.

Metric	Value
Lines of Code (LOC)	455
Logical Lines of Code (LLOC)	197
Source Lines of Code (SLOC)	220
Total Comments	70
Single-line Comments	64
Multi-line Comments	65
Blank Lines	106
C % L (Comments on LOC)	15%
C % S (Comments on SLOC)	32%
C + M % L (Comments + Multi-line on LOC)	30%

Table 5.1 Code metrics for the `client.py` file

The table 5.1 shown in describes an analysis of the client code’s quality by providing specific metrics: by examining these values it is possible to evaluate the code in the most objective way possible.

In particular, the ‘client.py‘ code includes a total of 455 lines of code (LOC), a number that actually intends to contain the actual executable code, comments and also empty lines. A relatively high value like this represents a fair amount of complexity in the file, due to the concatenation of different functions.

The lines of logical code (LLOC) are 197 and are responsible for carrying out logical operations in the program: comments and empty lines are therefore excluded.

The lines of source code (SLOC) are 220 and include, in addition to the actual lines of code, also the cases in which multiple factors are declared on the same line.

Since the LLOC value is slightly higher than the one just mentioned, it is possible to notice how in some cases in the code an approach of declaring multiple factors on the same line has been used, in order to make the code more synthetic in some cases.

The comments in the code are 70, a number consistent with the project that guarantees a good understanding of the logical development: the greater the number of comments, the greater the possibility that the code can be understood by external developers and that they can eventually get their hands on it.

The empty lines within the code, useful for being able to correctly divide the different parts, are 106: this is a number that indicates a good organization that once again guarantees a concrete possibility of being read and understood without problems.

The program documentation covers 32% of the program, a significant number that ensures the right importance to this part of the project development.

In conclusion, the ‘client.py‘ file presents a good balance between the necessary complexity of the code and a fair level of documentation, with a clear structure that simplifies the understanding of the logical steps and the possible maintenance of the same.

These values collected above represent the achievement of the pre-established project objectives, to be added to those of developing a functioning algorithm that is suitable for the project requests.

Chapter 6

Evaluation of results and development prospects

6.1 Results achieved

The aim of this thesis project was to develop a road lane detection algorithm using a simulation tool, the CARLA simulated world, and a software architecture that would allow the processing of images in real time. For this purpose, it was necessary to introduce several modules, both external and standard, so that they all contributed to the creation of an accurate and precise recognition system, both in a simulated context and in the future in a real context.

This last part of the document deals with showing the results obtained through the experiments carried out by running the client code presented before. The simulations in particular were carried out in an environment that was structured to be as truthful and similar to reality as possible, so that it could be replicated or reused.

The development path was not entirely linear, as several paths were taken that, for the purpose of creating the project, served to understand the strengths but also those that had to be the aspects to be interpreted and possibly changed, as was described for the specific Lane Detection part.

Below, some images of different conditions during the running application will be shown, in order to underline the system's capabilities in tracing the detected polygon of the road lane. The different aspects of the perception algorithm will

then be discussed, both thanks to the images that will be reported, and thanks to the metrics presented at the end of each explanation of the different codes presented.

Finally, this final part of the paper is aimed at providing and proposing food for thought for possible future improvements, with techniques that could be implemented and even greater precautions for possible future iterations of the product.

In the first analysis, in figure 6.1, a simulation scene is presented in a highway environment: this context was initially chosen because it is easier to manage than others, such as the city, to evaluate an initial effectiveness of the detection system.

The highway as such has been taken as a starting point for each Lane Detection system even in the past, where driving assistance was in its infancy and consequently we started from simpler contexts. In fact, this context does not include important horizontal signs, has well-defined lanes, does not have large obstacles to avoid in recognition and above all does not include curves with very evident angles.

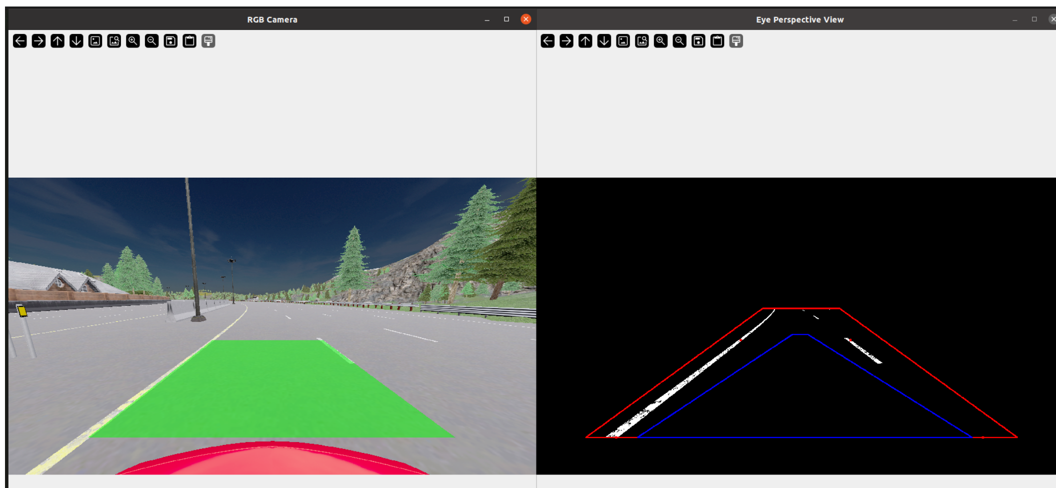


Fig. 6.1 Highway view, first attempt

This first scenario made it possible to notice the first problems and then solve them quickly and easily, such as the position of the mask creation points, rather than the reference points for the creation of the images from above.

From a purely technical point of view, you can already see how the view from above is applied correctly, avoiding the annoying perspective problems.

The lanes are also immediately clearly represented, with the full polygon colored in green clearly visible and faithfully representing the lane of the road: this aspect therefore makes you think about how, despite the environment of the first develop-

ment being simpler than the one that will actually be taken into consideration for the final project, i.e. a city, the models used, starting from the masks, are functional.

However, these results are a starting point, as the goal is to deal with more realistic driving conditions in reference to more everyday situations, and therefore urban life situations.

After having supported with excellent performances the first situations in a simplified environment such as the highway, the focus was shifted to a more realistic but complex environment, namely the urban environment.

The image below, figure 6.2, shows a simulation made inside the standard CARLA simulated city, characterized in this frame by a straight road with the presence of important buildings, trees and vehicles parked on the side of the road.

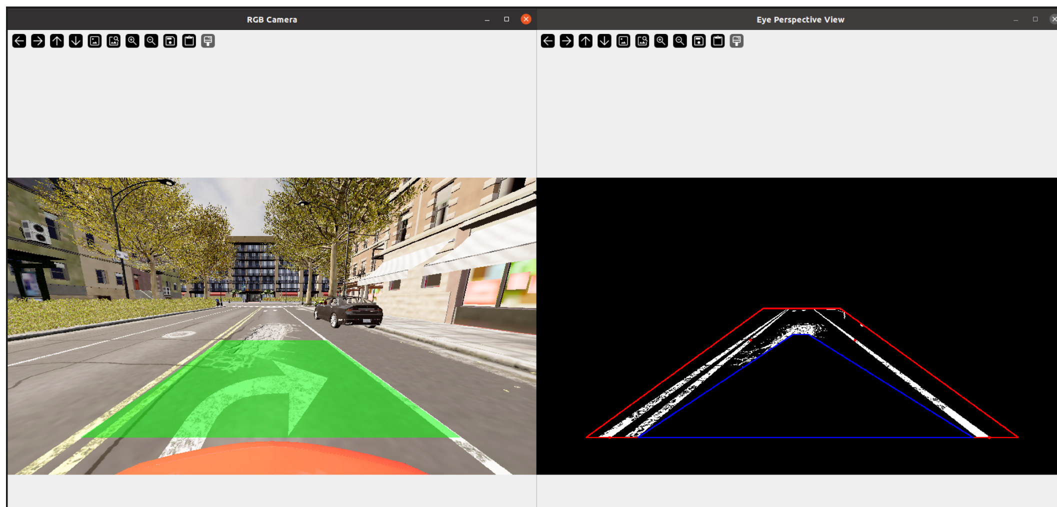


Fig. 6.2 Linear urban context frame

This situation, although still linear, introduces some elements that could complicate the success of the algorithm, such as the presence of sidewalks on the right and left side of the road or horizontal signs.

Despite the increase in difficulty, the proposed Lane Detection system continued to have excellent results in terms of performance and reliability, managing to accurately recognize the road lane. The perspective view, as can be seen on the right, continues to work correctly, providing great support to the realization of the project.

The representation of the green polygon on the left of the image remains of excellent workmanship, with a clear identification of the lane.

In conclusion, despite the increase in potential elements that could disturb the success of the perception algorithm, the basis for addressing the difficulties of the project is good and has made further improvements possible. The robustness of the code has proven to be still able to withstand situations of greater complexity.

The last image brought into analysis, figure 6.3, represents a further increase in the difficulty in recognizing the road lanes in the chosen simulation environment, as the situation of an urban curve is analyzed with the consequent complications that concern it. Differently from the previously analyzed contexts, in which the roads were mostly straight, in this situation the program must be able to manage a context in which there may be several complications, starting from the fact that the camera is fixed and therefore does not always point in the direction of the curve taken.

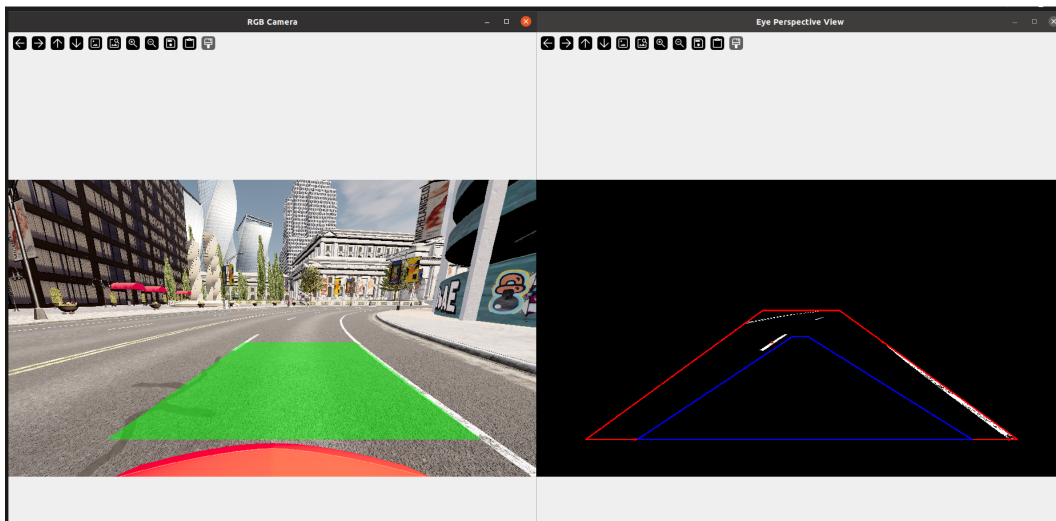


Fig. 6.3 Situation on a curve in an urban context

Despite the increase in difficulties represented, the improved and developed system of the Lane Detection perception algorithm continues to have excellent results in terms of performance, highlighting once again the refinement of the project and therefore of the developed code. The confinement masks continue to play a crucial role in the analysis and recognition of the position of the lanes, once again affirming that the choice to use them was correct, even in more difficult situations.

The colored filled polygon continues to correctly represent the lane in a clear and precise way, reinforcing with another good result the chosen approach, even in urban contexts and with curves of non-negligible angle.

This result, more than the previous ones, underlines that the project is robust and can be an excellent starting point for future applications, also because it represents a valid alternative to the projects present online, only able to manage situations on the highway or similar.

6.2 Conclusions

The current thesis project has had from the beginning the focus of developing and implementing a perception algorithm for Lane Detection in a simulated environment through the use of the CARLA simulator in an urban environment, in order to propose a valid alternative to some technologies present online.

This work is part of the large context of autonomous driving, currently one of the greatest challenges for the automotive sector and in general for the human race as it could completely revolutionize even everyday life.

Through a well-organized work with iterative development phases together with the Luxoft company, a robust recognition system has been created, capable of recognizing road lanes with excellent accuracy: the results at the level of pure software development have been evaluated through specific metrics demonstrating how the algorithm is able to meet expectations.

In this last part of the treatise, some evaluations of the system performance will be presented, some considerations regarding the problems faced during the project and finally possible future improvements to be made will be discussed.

1. Goals achieved

The success of the project is due in particular to the CARLA simulator, the open-source platform that made it possible to design the thesis in its entirety, creating dynamic environments that were able to adapt to the needs of the moment.

From the very first moment of development, the goal was to propose a valid alternative to the existing Lane Detection proposals, trying to develop a lean but effective algorithm, capable of recognizing lanes even in difficult environments.

The techniques used, namely the submodules related to thresholding and perspective transformation, are only the final aspects of the project: it was necessary to go through several failed attempts but which made us understand new aspects of

the topic. In fact, the final solution does not only involve image segmentation but through various controls of the single frame analyzed allows us to obtain a clear improvement with respect to the overall performance.

One of the greatest achievements of this project is undoubtedly the possibility of running the application within a simulated urban context, an important goal that had been set at the beginning, where driving conditions are much more complex than in a simple road environment.

The dynamic masks introduced within the main client then made it possible to increase the accuracy of the final program.

In addition, another aspect of fundamental importance was the modularity of the code, capable of integrating different aspects in an easy and intuitive way, thus easily recognizing the problems that are normally encountered and solving bugs without affecting the integrity of the code. The choice of using a pipeline for image processing made it possible to obtain an effective, fast and lean system.

In conclusion, the project was able to achieve all the results set at the beginning, both from the point of view of performance and from the point of view of the goodness of the code in terms of IT metrics.

2. Performance Evaluation

The screening of the program performance and the project as a whole focused in particular on two aspects: the precision in recognizing the lanes, the core of the theses, and the accuracy of the code, through a critical analysis of the metrics using the Radon tool. The values obtained demonstrated how, despite the exponential increase in the complexity of the program, it remains structured in the correct way, with a good ease of understanding even for an external user and above all well documented for any future projects: this is an important factor as it was the hope from the very first moment of development, to be able to obtain a scalable product for other activities. The separation between the different sections of the code and a low cyclomatic complexity ensure an excellent logical flow.

The choice to dedicate a submodule to the two auxiliary functions, the one dedicated to thresholding and the one dedicated to perspective transformation, proved to be a winning choice as they allowed to keep things separate and consequently allowed to work in a logical and simple way on the development: the integration

of the same was then simple with an important gain in terms of scalability of the project.

The computational performances were then extensively documented and commented, so as not to leave anything unfinished or inadequately explained: the management of resources was therefore able to benefit greatly, obtaining a fluid program that can easily respond to the needs of an urban context.

Another aspect of fundamental importance and reason for great reasoning was how to manage situations in which the lane lines are totally absent, as in the case of large road intersections or more simply when the signs are totally absent for other reasons: the choice fell on a probabilistic approach, that is, to keep in memory the last polygon drawn in which the road lines were present, using the latest information saved to project a virtual trajectory, based solely on previous data, obtaining excellent results despite the scarcity of data in those situations. This choice therefore gave the possibility of obtaining a trajectory calculated a priori, keeping the analysis fluid and without interruptions, as can be seen in the figure 6.4.

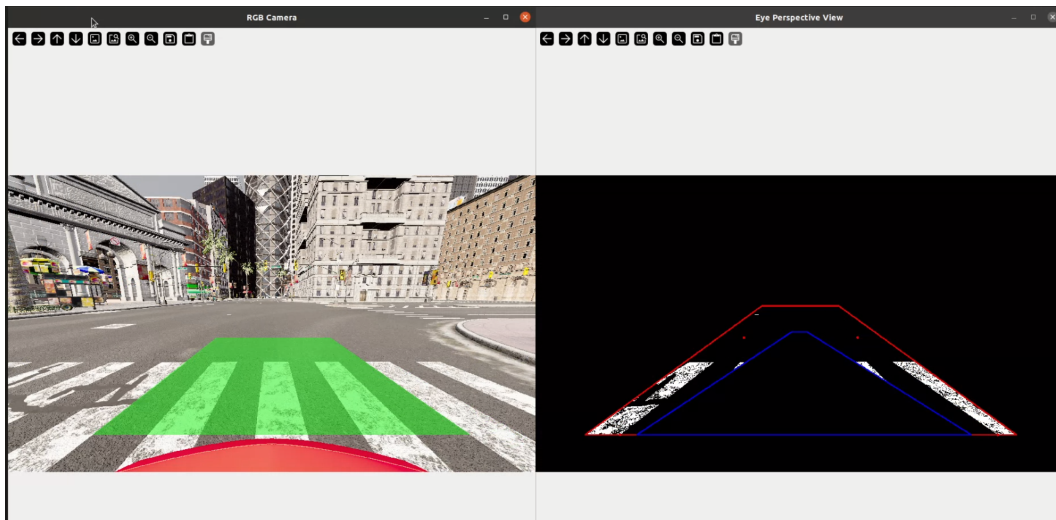


Fig. 6.4 Situation on a curve in an urban context with no data for Lane Detection

In essence, the code has been designed in such a way as to achieve an ideal balance between complexity and performance, allowing to maximize the relationship between the resources used and the response times of the application. The choice to save the individual polygons in memory, although it weighs a little on the resources requested from the system, has allowed to obtain a good fluidity of use, ensuring operational continuity even in complex situations.

3. Limits and obstacles encountered

During the development phases of the project, several problems were encountered that presented the continuous need to question the work done in order to obtain the best result, sometimes even different from what was intended. However, this should not be misleading: within the great topic of software development, it is more important to be able to have a critical thought regarding the relationship between the results obtained and those that were intended, in such a way as to be able to possibly compensate for shortcomings or malfunctions.

One of the aspects that was given greater weight was certainly that of being able to perform with an algorithm in complex conditions such as those of the urban context: the presence of obstacles, horizontal or vertical signs, more rigid curves than the highway or the noise of the image, made the development of the project more complicated: however, it is thanks to these difficulties that the program was able to reach a level of accuracy that was gradually increasing throughout the development phase, allowing in its latest version to compensate for problems that were initially decidedly hindering. It was found that the first approaches used, although they worked perfectly during the entire straight path, were not able to do adequately well during the curvature phases.

Another aspect that gave some difficulty concerns the position or more generally the use of masks for the regions of interest of the algorithm and therefore for the filtering of the images. At first, static masks were used, where the position was obtained experimentally but without great success, since in several cases the algorithm was not able to distinguish the roadway from other disturbing lines or white pixels. This approach in fact led to several false positives, a number decidedly higher than the threshold that had been set. To get to the bottom of this problem, the only solution that was possible to introduce was to use dynamic masks that were able to adapt to the conditions in real - time, certainly requiring a greater effort in terms of performance from the machine, but guaranteeing a high precision in the recognition of the lines. The concatenation of the internal and external masks has therefore given the possibility of isolating the regions of interest, where in particular the internal one has allowed us to eliminate portions of images that are not useful for our purposes.

Finally, a last aspect that has been a source of debate during the development phases was the one related to what to show when some lines are not present in

the image in order to be able to perform the Detection of the same: the decision taken, which then turned out to be interesting and productive, was to keep the last recognized polygon saved and to project the one in the case of absence of data. This made it possible to maintain continuity during the process.

The aspect parallel to all this that has always been taken into consideration has certainly been the one related to the management of the machine's resources, which could not be required to make an excessive effort in terms of memory and energy. The focus was to find in any case a compromise between performance and expenditure of energy.

4. Impact, Applications and Future Developments of the Algorithm

The project that has been carried out on Perception algorithm for Lane Detection in a simulated urban environment has shown significant interesting ideas that can be worked on in the future, showing a new frontier for autonomous driving, setting the goal ever further, while ensuring high performance and ever greater safety.

Although this development has been carried out within the CARLA simulated system, this should not lead one to think of a system in which real-life complexities have been neglected: in fact, the primary objective has always been to provide a world as true as possible, in such a way as to make everything scalable for possible future integrated applications. In fact, the algorithm could be adapted with other parallel platforms of ADAS systems. The peculiarity of the program is in fact the ability to be able to manage the running of the application even in complicated contexts, in which other proposals present online would not be able to do the same. In fact, the variable conditions of the city do not present a particular problem.

A possible future application, which could undoubtedly further increase the program's performance, concerns the integration of a greater number of sensors, perhaps positioned in different positions with respect to the single camera adopted in this project, in such a way as to be able to provide different points of view of the surrounding environment and therefore be able to refine the recognition of the lanes. The complexity related to this approach could be linked to the method through which to merge the different images, not encountering errors of perception between the different shots obtained. Undoubtedly, some of these sensors should be LIDAR or radar, in such a way as to overcome the difficulties that could be encountered in the event of fog or difficult weather conditions.

The addition of these new data acquisition tools would not only allow to refine the perception technique and therefore make the process even more precise, but would give new ideas to implement other technologies; in particular, the entire project could include functions to recognize dynamic obstacles or vertical signs, such as signs or other types of indications.

The real turning point for this project, however, could undoubtedly be the introduction of machine learning systems, such as convolutional neural networks (CNN), which are very advanced data processing systems in order to learn from the machine's past. The power of this application would reside above all in the exponential learning capacity capable of improving performance very quickly and being able to understand more and more different conditions each time [10]. A recent study has shown how the use of machine learning systems such as CNN has given enormous improvements in the accuracy of the results, especially where atmospheric conditions or heavy traffic did not always allow an accurate view of the road [11].

The combined use of machine learning together with other sensors such as LIDAR or radar would allow the algorithm to be strengthened, obtaining a multimodal approach, fusing the data received at a visual level with the calculated and processed results of artificial intelligence to obtain even greater safety.

The use of supervised learning algorithms could therefore give the possibility of integrating the current project with other ADAS systems, thus moving in the direction of obtaining a dynamic system where lane recognition is only one of the fundamental pieces regarding autonomous driving.

Another possible aspect of great importance could be that linked to the recognition of dynamic obstacles, such as pedestrians, bicycles or why not other vehicles moving on the roadway [12].

In conclusion, the developed project represents a starting point from which to launch various annexed studies aimed at autonomous driving, with enormous growth capacity thanks to the fusion with other technologies. Such future prospects would not only allow increasing the precision of the program, but would also allow us to take paths that have not yet been explored regarding this great topic of autonomous driving.

5. General Conclusion

The work project developed and explained within this represents an important point within the vast world of perception systems aimed at autonomous driving, where in particular the aspect relating to the recognition of road lanes was analyzed, seeking the best balance between performance and energy demand from the machine.

The CARLA platform, the simulation environment used to make everything even more realistic without having to spend on real-time simulation, gave the possibility of testing and simulating the algorithm with high precision and for the time necessary to be able to ensure the success of the project. The complexity was managed in the best possible way, gradually increasing its critical issues, starting from a motorway environment, simpler and excellent to start with, up to the urban environment in which the last difficulties encountered were perfected, such as the most challenging curves or the presence of other cars.

The main objective of this work was to develop an algorithm capable of recognizing road lines with high precision, despite the disturbances that can be encountered such as horizontal and vertical signs that sometimes create difficulties, or with more critical atmospheric conditions: these objectives were all successfully achieved, as demonstrated by the various testing phases carried out.

From a technical and development point of view, this thesis gave particular importance both to the software part, fundamental for the success of the project, but also to the hardware part, thus focusing on the feasibility and the relationship between performance and energy and computational requirements. This result is the core of mechatronic engineering as an overall vision is strictly necessary in order to obtain a concrete and interesting result for future applications, but which are reliable and realistic in the field of engineering.

The program was optimized through a pipeline for image processing, through the OpenCV library, fundamental for the success of the project and above all to maximize response times and management of the camera used. The response speeds were in fact immediately a theme of fundamental importance as they were strictly necessary to optimize the system. Another very important aspect that required a fair amount of time to be perfected concerns the need for integration between the different modules used, starting from the one relating to thresholding and perspective transformation, up to the actual management of Lane Detection.

Although the entire project was tested only in simulated environments, the focus during the entire development phase was to have the possibility in the future to integrate this system in a real context, bringing the level of accuracy to an even higher step. In fact, the algorithm, with the improvements and the addition of the other techniques discussed in the previous subchapter, could represent a new frontier for lane recognition.

Therefore, looking back at the objectives set, they have been definitely achieved and in some specific cases exceeded, demonstrating how the work that has been done can have significant practical repercussions. The robustness of the project and the solidity of the results obtained are certainly excellent ideas for the future.

If we then take into consideration the impact that this study can have on mechatronic engineering, we must think that at the basis of this entire project there is undoubtedly a particular attention to the integration of different but necessarily connected worlds such as mechanics, electronics and obviously IT. The control of these three areas in fact represents the challenge carried forward by every mechatronic project.

The future prospects for this enormous world still partially unexplored are in fact extremely rosy, as technology is making huge steps in this direction, perfectly understanding the risks but above all the benefits that these efforts could bring, in a "futuristic" world to which it is right to tend.

In conclusion, it is right to underline how this project is placed in a state of the art where research is not yet saturated, indeed, it is waiting to be explored to bring news and give a demonstration how mechatronic engineering can give new and important ideas for the intelligent mobility of the future.

References

[1] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, and J. Highsmith. Manifesto for agile software development, 2001.

[2] Microsoft Docs. *API Definition*, 2023. Available online: https://docs.microsoft.com/.

[3] CARLA Simulator. *CARLA Python API Documentation*, 2024. Available online: https://carla.readthedocs.io/en/latest/python_api/.

[4] CARLA Developers. *CARLA Documentation: Core Map*, 2024. Available online: https://carla.readthedocs.io/en/0.9.15/core_map/.

[5] OpenCV Documentation. *Image Thresholding*, 2024. Available online: https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html.

[6] Radon Documentation. *Radon Documentation*, 2024. Available online: https://radon.readthedocs.io/en/latest/.

[7] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2004.

[8] OpenCV. *OpenCV Documentation - Geometric Transformations of Images*, 2024. Available online: https://docs.opencv.org/.

[9] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Science & Business Media, 2009.

[10] X. Zhou, L. Wang, X. Tang, and Z. Liu. Lane detection algorithm based on deep learning for autonomous driving. *IEEE Transactions on Intelligent Vehicles*, 2020.

[11] X. Chen, H. Ma, J. Wan, B. Li, and T. Xia. Multi-view 3d object detection network for autonomous driving. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[12] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

Appendix A

Threshold_code

```
"""
Threshold module

This module provides functions for thresholding images.
"""

import logging
import os
import cv2

# Configuring logging
logging.basicConfig(level=logging.DEBUG)

# pylint: disable=no-member


def apply_threshold_and_save_image(
    img_path="photo/road.jpg", output_path=None,
        is_preview_enable=False
):
    """
    Apply_threshold_and_save_image function.

    Args:
        img_path (str): the path to the input image.
        output_path (str, optional): the path to save the
            output image.
```

```
            If not provided, the same directory as the input is used.

        Returns:
            None
        """

        # Input parameter validation
        if isinstance(img_path, str):
            dirname = os.path.dirname(__file__)
            img_path = os.path.join(dirname, img_path)
        if not os.path.isfile(img_path):
            logging.error("Invalid input image path.")
            return None

        # Output parameter validation
        if output_path is None:
            input_dir = os.path.dirname(img_path)
            filename, extension = os.path.splitext(os.path.basename(img_path))
            output_filename = filename + "_thresholded" + extension
            output_path = os.path.join(input_dir, output_filename)
        elif not isinstance(output_path, str):
            logging.warning("Invalid output image path.")
            return None

        original_frame = cv2.imread(img_path)

        frame_grayscale_applied = cv2.cvtColor(original_frame, cv2.COLOR_BGR2GRAY)

        # cv2.imshow("Gray", frame_grayscale_applied)

        # Simple thresholding
        _, frame_threshold_applied = cv2.threshold(
            src=frame_grayscale_applied, thresh=205, maxval=255, type=cv2.THRESH_BINARY
        )

        if is_preview_enable:
            cv2.imshow("Road", original_frame)
            cv2.imshow("Gray", frame_grayscale_applied)
```

```
64          cv2.imshow("Simple thresholding",
                frame_threshold_applied)
65          cv2.waitKey(0)
66          cv2.destroyAllWindows()
67
68      # Output image saving
69      cv2.imwrite(output_path, frame_threshold_applied)
70      logging.info("Output image saved successfully at %s.",
            output_path)
71      return output_path
72
73
74  def main():
75      """
76      Main function.
77
78      This function is responsible for executing the
            apply_threshold_and_save_image function.
79      """
80
81      apply_threshold_and_save_image()
82
83
84  if __name__ == "__main__":
85      main()
```

Listing A.1 Thresholding Code

Appendix B

Perspective_Transform_code

```
    """
eye_perspective.py

This module provides functionality to apply a perspective transform to an image,
simulating an "eye perspective" effect. The main components include:

1. `PerspectiveTransformParameters`:
   A data class that holds parameters for the perspective transformation, such as
   source and destination points, circle drawing settings, and constraints for
   adjusting points based on white pixels in the image.

2. `is_white_pixel(frame, point)`:
   Checks if a given point in the frame is a white pixel.

3. `find_nearest_white_pixel(frame, point, max_distance)`:
   Finds the nearest white pixel to the given point along the x-axis within a specified
   maximum distance.

4. `validate_point(frame, point, max_distance, max_move_distance, internal_trapezium_point=None, check_right=None)`:
   Validates and adjusts a single point based on specified conditions, ensuring it is a
```

```
21     white pixel within an allowable move distance and optionally
           within trapezium constraints.
22
23  5. `validate_and_adjust_detection_points(frame, params)`:
24     Validates and adjusts the top left, top right, bottom left,
          and bottom right points
25     before applying the perspective transform to ensure they are
           white pixels and within
26     allowable distances.
27
28  6. `apply_eye_perspective_transform(frame, params)`:
29     Applies the perspective transformation to the input frame
          using the provided
30     parameters. It first validates and adjusts the points, then
          draws circles at the
31     source points and performs the transformation.
32
33  7. `main()`:
34     The main function that reads an input image, applies the
          perspective transform, and
35     displays and saves the original and transformed frames.
36  """
37
38  import logging
39  from dataclasses import dataclass
40  import cv2
41  import numpy as np
42
43  # pylint: disable=no-member
44
45  # This check is disabled because we are using an external
        library OpenCv
46  # that Pylint cannot correctly analyze. We are confident that
        the members we are accessing do exist.
47
48  # pylint: disable=I1101
49
50  # Configuring logging
51  logging.basicConfig(level=logging.DEBUG)
52
53
54  @dataclass
55  class PerspectiveTransformParameters:
```

```
56      """
57      Data class for perspective transform parameters.
58      """
59  
60      output_width: int = 1920
61      output_height: int = 1080
62      top_left: tuple = (851, 590)
63      bottom_left: tuple = (300, 944)
64      top_right: tuple = (1105, 590)
65      bottom_right: tuple = (1690, 944)
66      destination_top_left: tuple = (50, -100)
67      destination_bottom_left: tuple = (50, 1080)
68      destination_top_right: tuple = (1870, -100)
69      destination_bottom_right: tuple = (1870, 1080)
70      circle_radius: int = 5
71      circle_color: tuple = (0, 0, 255)
72      circle_thickness: int = -1
73      max_distance: int = 150  # Maximum limit for searching for
           white pixels
74      max_move_distance: int = (
75          150  # Maximum distance to move points from the initial
                position
76      )
77      top_left_internal_trapezium: tuple = (938, 570)
78      top_right_internal_trapezium: tuple = (988, 570)
79  
80  
81  def is_white_pixel(frame, point):
82      """
83      Check if the given point in the frame is a white pixel.
84  
85      Args:
86          frame (numpy.ndarray): Input frame to be processed.
87          point (tuple): The point to check.
88  
89      Returns:
90          bool: True if the point is a white pixel, False
              otherwise.
91      """
92      x, y = point
93      # Check if the point is within the frame boundaries
94      if x < 0 or x >= frame.shape[1] or y < 0 or y >= frame.
           shape[0]:
```

```
95            return False
96        # Check if the pixel is white
97        return np.array_equal(frame[y, x], [255, 255, 255])
98
99
100   def find_nearest_white_pixel(frame, point, max_distance):
101       """
102       Find the nearest white pixel to the given point along the x
              -axis within the max distance.
103
104       Args:
105           frame (numpy.ndarray): Input frame to be processed.
106           point (tuple): The point to check.
107           max_distance (int): Maximum distance to search.
108
109       Returns:
110           tuple: The new point with the nearest white pixel.
111       """
112       x, y = point
113       for offset in range(1, max_distance + 1):
114           # Check pixel to the right
115           if is_white_pixel(frame, (x + offset, y)):
116               return (x + offset, y)
117           # Check pixel to the left
118           if is_white_pixel(frame, (x - offset, y)):
119               return (x - offset, y)
120       return point
121
122
123   def validate_point(
124       frame,
125       point,
126       max_distance,
127       max_move_distance,
128       internal_trapezium_point=None,
129       check_right=None,
130   ):
131       """
132       Validate and adjust a single point based on the conditions.
133
134       Args:
135           frame (numpy.ndarray): Input frame to be processed.
136           point (tuple): The point to validate and adjust.
```

```
137         max_distance (int): Maximum distance to search for a
                white pixel.
138         max_move_distance (int): Maximum allowed movement
                distance.
139         internal_trapezium_point (tuple): Internal trapezium
                boundary point for validation.
140         check_right (bool): If True, check if the new point is
                not to the right of the internal trapezium point.
141                              If False, check if the new point is
                                    not to the left of the internal
                                    trapezium point.
142 
143     Returns:
144         tuple: The adjusted point or the original point if no
                valid adjustment found.
145     """
146 
147     def valid_move(new_point, point, max_move_distance):
148         return (
149             np.linalg.norm(np.array(new_point) - np.array(point
                    )) <= max_move_distance
150         )
151 
152     def within_trapezium_constraints(new_point,
            internal_trapezium_point, check_right):
153         if internal_trapezium_point is None or check_right is
                None:
154             return True
155         return (check_right and new_point[0] <=
                internal_trapezium_point[0]) or (
156             not check_right and new_point[0] >=
                    internal_trapezium_point[0]
157         )
158 
159     if is_white_pixel(frame, point):
160         return point
161 
162     logging.info(f"Adjusting point from {point}")
163     new_point = find_nearest_white_pixel(frame, point,
            max_distance)
164 
165     if new_point == point:
166         logging.info(
```

```
167                 f"No white pixel found within {max_distance} pixels
                        for the point. Keeping original position."
168             )
169             return point
170
171         if not valid_move(new_point, point, max_move_distance):
172             logging.info(
173                 f"New point {new_point} exceeds max move distance.
                        Keeping original position."
174             )
175             return point
176
177         if not within_trapezium_constraints(
178             new_point, internal_trapezium_point, check_right
179         ):
180             logging.info(
181                 f"New point {new_point} is invalid based on
                        internal trapezium constraints. Keeping original
                         position."
182             )
183             return point
184
185         logging.info(f"Adjusted point to {new_point}")
186         return new_point
187
188
189     def validate_and_adjust_detection_points(frame, params):
190         """
191         Validate and adjust the top left, top right, bottom left,
                and bottom right points before applying the perspective
                transform.
192
193         Args:
194             frame (numpy.ndarray): Input frame to be processed.
195             params (PerspectiveTransformParameters): Parameters for
                    perspective transformation.
196
197         Returns:
198             PerspectiveTransformParameters: Updated parameters with
                    validated and adjusted points.
199         """
200         max_distance = params.max_distance
201         max_move_distance = params.max_move_distance
```

```
202
203     # Validate and adjust top points with internal trapezium
            constraints
204     params.top_left = validate_point(
205         frame,
206         params.top_left,
207         max_distance,
208         max_move_distance,
209         params.top_left_internal_trapezium,
210         check_right=True,
211     )
212     params.top_right = validate_point(
213         frame,
214         params.top_right,
215         max_distance,
216         max_move_distance,
217         params.top_right_internal_trapezium,
218         check_right=False,
219     )
220
221     # Validate and adjust bottom points without internal
            trapezium constraints
222     params.bottom_left = validate_point(
223         frame, params.bottom_left, max_distance,
                max_move_distance
224     )
225     params.bottom_right = validate_point(
226         frame, params.bottom_right, max_distance,
                max_move_distance
227     )
228
229     return params
230
231
232 def apply_eye_perspective_transform(frame, params=
        PerspectiveTransformParameters()):
233     """
234     Apply eye perspective transformation function.
235
236     Args:
237         frame (numpy.ndarray): Input frame to be processed.
238         params (PerspectiveTransformParameters): Parameters for
                perspective transformation.
```

```
239
240     Returns:
241         tuple: A tuple containing the transformed frame and the
                transformation matrix.
242     """
243
244     # Validate and adjust points before applying the
          transformation
245     params = validate_and_adjust_detection_points(frame, params
          )
246
247     # List of source points
248     source_points = [
249         params.top_left,
250         params.bottom_left,
251         params.top_right,
252         params.bottom_right,
253     ]
254
255     # Draw circles on the frame at the source points
256     for point in source_points:
257         cv2.circle(
258             frame,
259             point,
260             params.circle_radius,
261             params.circle_color,
262             params.circle_thickness,
263         )
264
265     # Apply geometrical transformation
266     source_points_array = np.float32(source_points)
267
268     # Define new destination points for a higher view
269     destination_points = np.float32(
270         [
271             params.destination_top_left,
272             params.destination_bottom_left,
273             params.destination_top_right,
274             params.destination_bottom_right,
275         ]
276     )
277
```

```
278     matrix = cv2.getPerspectiveTransform(source_points_array,
            destination_points)
279     transformed_frame = cv2.warpPerspective(
280         frame, matrix, (params.output_width, params.
                output_height)
281     )
282
283     return frame, source_points_array
284
285
286 def main():
287     """
288     Main function.
289     """
290
291     # Read the input image
292     input_image_path = "input.jpg"
293     frame = cv2.imread(input_image_path)
294
295     if frame is None:
296         logging.error("Error loading image")
297         return
298
299     # Apply the perspective transform
300     params = PerspectiveTransformParameters()
301     original_frame, transformed_frame =
            apply_eye_perspective_transform(frame, params)
302
303     if transformed_frame is not None:
304         # Display the original and transformed frames
305         cv2.imshow("Original Frame", original_frame)
306         cv2.imshow("Transformed Frame", transformed_frame)
307
308         # Save the transformed frame
309         output_image_path = "transformed.jpg"  # Change this to
                your desired output path
310         cv2.imwrite(output_image_path, transformed_frame)
311         logging.info(f"Transformed image saved to {
                output_image_path}")
312
313         # Wait for a key press and close the windows
314         cv2.waitKey(0)
315         cv2.destroyAllWindows()
```

```
316     else:
317         logging.error("Transformed frame is None. Skipping
            display and save.")
318
319
320 if __name__ == "__main__":
321     main()
```

Listing B.1 Eye Perspective View code

Appendix C

Camera_callback

```python
def camera_callback(image, data_dict, thresholding_settings=
    ThresholdingSettings()):
    """
    Custom callback for processing camera images.
    """

    # Convert the raw image to a numpy array
    image_array = np.array(image.raw_data)
    image_np = image_array.reshape((image.height, image.width,
        4))

    # Save the image to a temporary file
    with tempfile.NamedTemporaryFile(suffix=".jpg", delete=
        False) as temp_file:
        temp_image_path = temp_file.name
        image.save_to_disk(temp_image_path)

        # Apply thresholding to the image
        thresholded_image_path = apply_threshold_and_save_image
            (temp_image_path)

        # Load the thresholded image back
        thresholded_image = cv2.imread(thresholded_image_path)

        # Create external mask
        external_mask = create_mask(
            thresholded_image.shape[:2], thresholding_settings.
                extern_mask_points
```

```
24         )
25         masked_image = cv2.bitwise_and(
26             thresholded_image, thresholded_image, mask=
                   external_mask
27         )
28 
29         # Create internal mask
30         internal_mask = create_mask(
31             masked_image.shape[:2], thresholding_settings.
                   intern_mask_points
32         )
33         inverted_internal_mask = cv2.bitwise_not(internal_mask)
34 
35         masked_image = cv2.bitwise_and(
36             masked_image, masked_image, mask=
                   inverted_internal_mask
37         )
38 
39         # Apply the eye perspective
40         eye_perspective_image, source_points =
               apply_eye_perspective_transform(
41             masked_image
42         )
43 
44         draw_polygon(
45             eye_perspective_image,
46             thresholding_settings.extern_mask_points,
47             (0, 0, 255),
48             thresholding_settings.line_width,
49         )
50         draw_polygon(
51             eye_perspective_image,
52             thresholding_settings.intern_mask_points,
53             (255, 0, 0),
54             thresholding_settings.line_width,
55         )
56 
57         # Draw the polygon using the source_points
58         source_points = np.array(
59             source_points, dtype=np.int32
60         )  # Ensure source_points are in correct format
61 
```

```
62         # Ensure the points are in the correct order: top_left,
               bottom_left, bottom_right, top_right
63         polygon_points = np.array(
64             [source_points[0], source_points[1], source_points
                   [3], source_points[2]],
65             dtype=np.int32,
66         )
67 
68         # Create an overlay image for the transparent polygon
69         overlay = image_np.copy()
70 
71         # Draw the filled polygon on the overlay
72         cv2.fillPoly(overlay, [polygon_points], (0, 255, 0))
73 
74         # Blend the overlay with the original image using
               addWeighted
75         cv2.addWeighted(
76             overlay,
77             thresholding_settings.alpha,
78             image_np,
79             1 - thresholding_settings.alpha,
80             0,
81             image_np,
82         )
83 
84         # Update the data dictionary with the result image
85         data_dict["image"] = image_np
86         data_dict["eye_perspective_image"] =
               eye_perspective_image
```

Listing C.1 Thresholding Code

Appendix D

client.py

```
"""
Carla Simulation with Thresholded Camera:

This script connects to a Carla simulator instance, spawns a
    vehicle,
and attaches an RGB camera to it.
Images captured by the camera are showed in a separate window
    after applying a thresholding filter.
The script also adds additional vehicles to the simulation.
The thresholded camera feed is displayed in a window.

Requirements:
- Carla simulator installed and running
- Frame thresholding module ('frame_threshold.frame_threshold')
     for image processing
"""

import sys
import random
import time
import tempfile
from dataclasses import dataclass
import logging

import carla

import cv2
import numpy as np
```

```
26
27 from frame_threshold.frame_threshold import
       apply_threshold_and_save_image
28 from eye_perspective_module.eye_perspective import
       apply_eye_perspective_transform
29
30 from lux_ad_carla.PythonAPI.examples.controller import
       VehiclePIDController
31
32 # pylint: disable=no-member
33
34 # This check is disabled because we are using an external
       library OpenCv
35 # that Pylint cannot correctly analyze. We are confident that
       the members we are accessing do exist.
36
37 # pylint: disable=I1101
38
39 # Configure logging
40 logging.basicConfig(
41     level=logging.DEBUG,
42     format="%(asctime)s - %(name)s - %(levelname)s - %(message)
           s",
43     handlers=[logging.FileHandler("carla_simulation.log"),
           logging.StreamHandler()],
44 )
45
46
47 @dataclass
48 class CameraSettings:
49     """
50     A class to manage settings for an RGB camera in a Carla
           simulation.
51     """
52
53     image_size_x: int = 1920
54     image_size_y: int = 1080
55     field_of_view: float = 120
56     sensor_tick: float = 0.0333  # 30 frames per second
57     location: carla.Location = carla.Location(1.8, 0, 1.3)
58     rotation: carla.Rotation = carla.Rotation(-10, 0, 0)
59
60
```

```
61  @dataclass
62  class ThresholdingSettings:
63      """
64      A class to manage settings for thresholding in the camera
            callback.
65      """
66  
67      extern_mask_points: list = ((180, 944), (823, 475), (1103,
            475), (1750, 944))
68      intern_mask_points: list = (
69          (368, 944),
70          (931, 570),
71          (988, 570),
72          (1585, 944),
73      )
74      alpha: float = 0.5  # Transparency factor for blending
75      line_width: int = 3
76  
77  
78  def initialize_client():
79      """
80      Initializes the client and returns the client and the world
            .
81      If the connection fails, return None.
82      """
83      try:
84          # First of all, we need to create the client that will
                send the requests
85          # to the simulator. Here we assume the simulator is
                accepting
86          # requests at localhost port 2000
87          client = carla.Client("localhost", 2000)
88          client.set_timeout(15.0)
89  
90          # Once we have a client we can retrieve the world that
                is currently running
91          world = client.get_world()
92  
93          # The world contains the list of blueprints that we can
                 use for adding new
94          # actors into the simulation
95          blueprint_library = world.get_blueprint_library()
96  
```

```
97          logging.info("Client and world initialized successfully")
98          return client, world, blueprint_library
99      except carla.ServerError as e:
100         logging.error("Server error occurred during client initialization: %s", e)
101         sys.exit(1)
102     except carla.ClientConnectionError as e:
103         logging.error("Error connecting to the Carla server: %s", e)
104         sys.exit(2)
105
106
107 def spawn_vehicle(world, actor_list, default_color="255, 0, 0"):
108     """
109     Spawns a vehicle in the Carla simulation world.
110
111     Args:
112         world (carla.World): The Carla simulation world.
113         actor_list (list): A list to store the spawned actor.
114         default_color (str): The default color for the spawned vehicle in RGB format.
115
116     Returns:
117         carla.Actor or None: The spawned vehicle actor if successful, None otherwise.
118     """
119     # Get the blueprint library from the world
120     blueprint_library = world.get_blueprint_library()
121
122     # Find the blueprint for the vehicle
123     bp = blueprint_library.find("vehicle.mercedes.coupe_2020")
124
125     # Set the default color attribute for the vehicle blueprint if available
126     if bp.has_attribute("color"):
127         bp.set_attribute("color", default_color)
128
129     # Initialize vehicle as None
130     vehicle = None
131
132     # Attempt to spawn the vehicle multiple times
```

```
133     for _ in range(10):
134         # Select a spawn point from the map
135         transform = random.choice(world.get_map().
                get_spawn_points())
136 
137         # Try to spawn the vehicle at the selected spawn point
138         vehicle = world.try_spawn_actor(bp, transform)
139 
140         # If successful, add the vehicle to the actor list and
                print a success message
141         if vehicle is not None:
142             actor_list.append(vehicle)
143             logging.info("Created %s", vehicle.type_id)
144             break
145         # If unsuccessful, print a retry message
146         logging.debug("Retrying to spawn vehicle...")
147 
148     # If the vehicle is still None after multiple attempts,
            print a failure message
149     if vehicle is None:
150         logging.error("Failed to spawn vehicle after multiple
                attempts.")
151         sys.exit(3)
152 
153     return vehicle
154 
155 
156 def spawn_camera(world, vehicle, camera_settings=CameraSettings
        ()):
157     """
158     Spawns an RGB camera attached to the given vehicle.
159     Returns the spawned camera actor.
160     """
161     cam_bp = world.get_blueprint_library().find("sensor.camera.
            rgb")
162     cam_bp.set_attribute("image_size_x", str(camera_settings.
            image_size_x))
163     cam_bp.set_attribute("image_size_y", str(camera_settings.
            image_size_y))
164     cam_bp.set_attribute("fov", str(camera_settings.
            field_of_view))
165     cam_bp.set_attribute("sensor_tick", str(0.0333))  # 30
            frames per second
```

```
166     cam_transform = carla.Transform(camera_settings.location,
            camera_settings.rotation)
167     ego_cam = world.spawn_actor(
168         cam_bp,
169         cam_transform,
170         attach_to=vehicle,
171         attachment_type=carla.AttachmentType.Rigid,
172     )
173 
174     return ego_cam, cam_bp
175 
176 
177 def create_mask(image_shape, points):
178     """
179     Create a binary mask with a filled polygon.
180 
181     Args:
182         image_shape (tuple): Shape of the image (height, width)
                .
183         points (list of tuples): Polygon vertices as (x, y)
                coordinates.
184 
185     Returns:
186         numpy.ndarray: Binary mask with the polygon filled.
187     """
188 
189     mask = np.zeros(image_shape, dtype=np.uint8)
190     points_array = np.array(points, dtype=np.int32)
191     cv2.fillPoly(mask, [points_array], 255)
192     return mask
193 
194 
195 def draw_polygon(image, points, color, line_width):
196     """
197     Draw a polygon on an image by connecting the given points.
198 
199     Args:
200         image (numpy.ndarray): The image to draw on.
201         points (list of tuples): Polygon vertices as (x, y)
                coordinates.
202         color (tuple): Color of the polygon in BGR format.
203         line_width (int): Thickness of the polygon edges.
204     """
```

```
205     num_points = len(points)
206     for i in range(num_points):
207         start_point = points[i]
208         end_point = points[
209             (i + 1) % num_points
210         ]  # Ensures that the last point connects to the first
211         cv2.line(image, start_point, end_point, color,
            line_width)
212 
213 
214 def camera_callback(image, data_dict, thresholding_settings=
        ThresholdingSettings()):
215     """
216     Custom callback for processing camera images.
217     """
218 
219     # Convert the raw image to a numpy array
220     image_array = np.array(image.raw_data)
221     image_np = image_array.reshape((image.height, image.width,
            4))
222 
223     # Save the image to a temporary file
224     with tempfile.NamedTemporaryFile(suffix=".jpg", delete=
            False) as temp_file:
225         temp_image_path = temp_file.name
226         image.save_to_disk(temp_image_path)
227 
228         # Apply thresholding to the image
229         thresholded_image_path = apply_threshold_and_save_image
            (temp_image_path)
230 
231         # Load the thresholded image back
232         thresholded_image = cv2.imread(thresholded_image_path)
233 
234         # Create external mask
235         external_mask = create_mask(
236             thresholded_image.shape[:2], thresholding_settings.
                extern_mask_points
237         )
238         masked_image = cv2.bitwise_and(
239             thresholded_image, thresholded_image, mask=
                external_mask
240         )
```

```
241
242         # Create internal mask
243         internal_mask = create_mask(
244             masked_image.shape[:2], thresholding_settings.
                    intern_mask_points
245         )
246         inverted_internal_mask = cv2.bitwise_not(internal_mask)
247
248         masked_image = cv2.bitwise_and(
249             masked_image, masked_image, mask=
                    inverted_internal_mask
250         )
251
252         # Apply the eye perspective
253         eye_perspective_image, source_points =
                apply_eye_perspective_transform(
254             masked_image
255         )
256
257         draw_polygon(
258             eye_perspective_image,
259             thresholding_settings.extern_mask_points,
260             (0, 0, 255),
261             thresholding_settings.line_width,
262         )
263         draw_polygon(
264             eye_perspective_image,
265             thresholding_settings.intern_mask_points,
266             (255, 0, 0),
267             thresholding_settings.line_width,
268         )
269
270         # Draw the polygon using the source_points
271         source_points = np.array(
272             source_points, dtype=np.int32
273         )  # Ensure source_points are in correct format
274
275         # Ensure the points are in the correct order: top_left,
                bottom_left, bottom_right, top_right
276         polygon_points = np.array(
277             [source_points[0], source_points[1], source_points
                    [3], source_points[2]],
278             dtype=np.int32,
```

```
279            )
280
281            # Create an overlay image for the transparent polygon
282            overlay = image_np.copy()
283
284            # Draw the filled polygon on the overlay
285            cv2.fillPoly(overlay, [polygon_points], (0, 255, 0))
286
287            # Blend the overlay with the original image using
                   addWeighted
288            cv2.addWeighted(
289                overlay,
290                thresholding_settings.alpha,
291                image_np,
292                1 - thresholding_settings.alpha,
293                0,
294                image_np,
295            )
296
297            # Update the data dictionary with the result image
298            data_dict["image"] = image_np
299            data_dict["eye_perspective_image"] =
                   eye_perspective_image
300
301
302    def setup_camera(ego_cam, cam_bp, camera_data):
303        """
304        Set up the RGB camera with the given parameters.
305        """
306        # Set the camera recording data:
307        image_w = cam_bp.get_attribute("image_size_x").as_int()
308        image_h = cam_bp.get_attribute("image_size_y").as_int()
309
310        camera_data["image"] = np.zeros((image_h, image_w, 4),
               dtype=np.uint8)
311        camera_data["eye_perspective_image"] = np.zeros(
312            (image_h, image_w, 4), dtype=np.uint8
313        )
314
315        # Define the camera callback
316        ego_cam.listen(lambda image: camera_callback(image,
               camera_data))
317
```

```
318     # Create named windows and display the camera feed
319     cv2.namedWindow("RGB Camera", cv2.WINDOW_NORMAL)
320     cv2.namedWindow("Eye Perspective View", cv2.WINDOW_NORMAL)
321 
322     # Resize windows to occupy half the screen width
323     screen_width = int(cv2.getWindowImageRect("RGB Camera")[2])
324     screen_height = int(cv2.getWindowImageRect("RGB Camera")
            [3])
325     cv2.resizeWindow("RGB Camera", screen_width // 2,
            screen_height)
326     cv2.resizeWindow("Eye Perspective View", screen_width // 2,
             screen_height)
327 
328     # Move the window of the RGB Camera to the left side of the
             screen
329     cv2.moveWindow("RGB Camera", 0, 0)
330 
331     # Move the window of the Eye Perspective View to the right
            side of the screen
332     cv2.moveWindow("Eye Perspective View", screen_width // 2,
            0)
333 
334     cv2.waitKey(1)
335 
336 
337 def get_target_waypoint(vehicle, world):
338     """
339     Function to get the target waypoint for the vehicle.
340     """
341 
342     # Get the map from the world
343     carla_map = world.get_map()
344 
345     # Get the current location of the vehicle
346     location = vehicle.get_location()
347 
348     # Get the waypoint closest the current location
349     waypoint = carla_map.get_waypoint(location)
350 
351     # Get the next waypoint at a distance of 2.0 meters ahead
352     next_waypoint = waypoint.next(2.0)[0]
353 
354     # Set the target speed for the vehicle
```

```
355     target_speed = 20.0
356 
357     # Return the next waypoint and the target speed
358     return next_waypoint, target_speed
359 
360 
361 def main():
362     """
363     This main function connects to a Carla simulator instance,
364     spawns a vehicle with an attached RGB camera, and sets the
        vehicle to drive in autopilot mode.
365     Images captured by the camera undergo thresholding and are
        displayed in a window.
366     Additional vehicles are also spawned in the simulation.
367     """
368     actor_list = []
369     ego_cam = None
370 
371     random.seed(500)
372 
373     # Initialize the client, world, and blueprint library
374     client, world, blueprint_library = initialize_client()
375 
376     # Spawn the vehicle
377     vehicle = spawn_vehicle(world, actor_list)
378 
379     # Define transform
380     transform = vehicle.get_transform()
381 
382     # Spawn attached RGB camera
383     ego_cam, cam_bp = spawn_camera(world, vehicle)
384 
385     # Set up the camera
386     camera_data = {}
387     setup_camera(ego_cam, cam_bp, camera_data)
388 
389     # Define the proportional (K_P), integral (K_I),
390     # and derivative (K_D) gains for lateral control
391     args_lateral = {"K_P": 1.0, "K_I": 0.0, "K_D": 0.0}
392 
393     # Define the proportional (K_P), integral (K_I),
394     # and derivative (K_D) gains for longitudinal control
395     args_longitudinal = {"K_P": 1.0, "K_I": 0.0, "K_D": 0.0}
```

```
396
397     # Create a PID controller instance for the vehicle
398     # using the lateral and longitudinal control gains
399     pid_controller = VehiclePIDController(vehicle, args_lateral
            , args_longitudinal)
400
401     try:
402
403         while True:
404
405             # Get the next waypoint and target speed for the
                    vehicle using the defined function
406             waypoint, target_speed = get_target_waypoint(
                    vehicle, world)
407
408             # Compute the control command for the vehicle based
                     on
409             # the target speed and waypoint using the PID
                    controller
410             control = pid_controller.run_step(target_speed,
                    waypoint)
411
412             # Apply the computed control command to the vehicle
413             vehicle.apply_control(control)
414
415             cv2.imshow("RGB Camera", camera_data["image"])
416             cv2.imshow("Eye Perspective View", camera_data["
                    eye_perspective_image"])
417
418             if cv2.waitKey(1) == ord("q"):  # Close windows
                    when press ‘q‘
419                  break
420
421         cv2.destroyAllWindows()
422
423         # Add a few more vehicles to the simulation
424         transform.location += carla.Location(x=40, y=-3.2)
425         transform.rotation.yaw = -180.0
426         for _ in range(0, 10):
427             transform.location.x += 8.0
428
429             bp = random.choice(blueprint_library.filter("
                    vehicle"))
```

```
                # Use try_spawn_actor. If the spot is occupied by
                    another object,
                # the function will return None.
                npc = world.try_spawn_actor(bp, transform)
                if npc is not None:
                    actor_list.append(npc)
                    npc.set_autopilot(True)
                    logging.info("created %s", npc.type_id)

            time.sleep(5)

    except (carla.ServerError, carla.ClientError, carla.
        RPCError) as e:
        logging.error("Carla error occurred: %s", e)
    except UnexpectedError as e:
        logging.error("An unexpected error occurred: %s", e)

    finally:
        logging.info("destroying actors")
        if ego_cam is not None:
            ego_cam.destroy()
        client.apply_batch([carla.command.DestroyActor(x) for x
            in actor_list])
        logging.info("done.")


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

Listing D.1 Client code