Πολυτεχνείο Κρήτης
Σχολή Μηχανικών Παραγωγής και Διοίκησης

Διπλωματική Εργασία

Αυτόνομη Πλοήγηση Ηλεκτροκίνητου Αυτοκινήτου Πόλης

Αλέξανδρος Θεοχάρους

Η εργασία εκπονήθηκε στο πλαίσιο των απαιτήσεων για την απόκτηση Διπλώματος Μηχανικού από τη Σχολή Μηχανικών Παραγωγής και Διοίκησης του Πολυτεχνείου Κρήτης

Ιούλιος 2023

Technical University of Crete
School of Production Engineering and Management

Diploma Thesis

Autonomous Navigation of an Electric Urban Car

Alexandros Theocharous

Submitted in partial fulfillment of the requirements for the Diploma in Production Engineering and Management of the Technical University of Crete

July 2023

Thesis Committee

Eleftherios Doitsidis, Ph.D. *Assistant Professor*	School of Production Engineering and Management, Technical University of Crete
Dimitrios Ispakis, Ph.D. *Assistant Professor*	School of Production Engineering and Management, Technical University of Crete
Savvas Piperidis, Ph.D. *Laboratory Teaching Personnel*	School of Production Engineering and Management, Technical University of Crete

Acknowledgements

Firstly I would like to thank my supervising professor Mr. Eleftherios Doitsidis who passed on to me the love for robotics and also trusted me and motivated me to take on this very challenging thesis.
I would also like to thank my family that always supported me in every way possible throughout this seven year journey of studying in Crete and the people close to me that accompanied me through it. Without all of them this moment would not have been possible.

Finally, I want to dedicate this thesis to my late Grandmother, who I know that would have been the happiest of all today.

Contents

List of Figures

Acronyms and Abbreviations

AI Artificial Intelligence

API Application Programming Interface

BGR Blue Green Red

DBSCAN Density Based Spatial Clustering of Applications with Noise

DDPG Radio Detection And Ranging

DPG Deterministic Policy Gradient

eps epsilon

GNSS Global Navigation Satellite System

GPS Global Positioning System

GPU Graphics Processing Unit

HSV Hue Saturation Value

IMU Inertial measurement unit

LiDAR Light Detection And Ranging

minPts Minimum Points

MLP Multilayer Perception Policy

MSBE Mean Squared Bellman Error

NES Nintendo Entertaintment System

Nm Newton * Meter

OpenCV Library Open Computer Vision Library

Radar Radio Detection And Ranging

RGB Red Green Blue

RGBA Red Green Blue Alpha

RL Reinforcement Learning

rpm rounds per minute

SB3 Stable Baseliness 3

Εκτεταμένη Περίληψη

Τα τελευταία χρόνια η ανάπτυξη της τεχνολογίας τόσο στον τομέα της τεχνητής νοημοσύνης, όσο και στον τομέα της προσομοίωσης προχωράει με γοργούς ρυθμούς. Πλήθος νέων αλγορίθμων βαθιάς ενισχυμένης μάθησης αλλά και παραλλαγών τους έχουν αναπτυχθεί δίνοντάς έτσι την ευκαιρία σε ερευνητές να μπορούν να εκπαιδεύσουν αυτόνομους πράκτορες ώστε να χειρίζονται πολύπλοκα προβλήματα των οποίων η λύση με συμβατικούς αλγορίθμους έμοιαζε αδύνατη. Η βελτίωση των τεχνολογικών μέσων, νέες κάρτες γραφικών και επεξεργαστές, προσφέρουν την υπολογιστική δύναμη για να αναπτυχθούν προσομοιωτές οι οποίοι θα μπορούν να δημιουργήσουν περιβάλλοντα πολύ κοντά στα φυσικά με το επιπλέον πλεονέκτημα της εξάλειψης κάθε τυχαίας παραμέτρου που μπορεί να επηρεάσει τα αποτελέσματα ενός πειράματος και μιας δοκιμής στον φυσικό κόσμο.

Οι προσομοιωτές αυτοί αποτελούν κομβικό εργαλείο για τους ερευνητές οι οποίοι τους χρησιμοποιούν ώστε με ευκολία να δοκιμάσουν τα διάφορα μοντέλα τεχνητής νοημοσύνης και να κρίνουν ποιο από αυτά είναι το βέλτιστο για την επίλυση ενός προβλήματος ή και για την ανάπτυξη καινοτόμων μοντέλων.

Ο συνδυασμός των νέων αλγορίθμων με τους ρεαλιστικούς προσομοιωτές έχει επιφέρει μεγάλη πρόοδο στον τομέα της αυτόνομης οδήγησης καθώς καθιστά πολύ απλή τη διαδικασία της εκπαίδευσης ενός αυτόνομου πράκτορα για ένα αυτοκίνητο καθώς πλέον όλα τα στάδια, από την εκπαίδευση μέχρι τις δοκιμές για τελειοποίηση σε διαφορετικά σενάρια, μπορούν να γίνουν εικονικά χωρίς την ανάγκη πραγματικού αυτοκινήτου. Πλήθος εργασιών μπορούν να βρεθούν στην βιβλιογραφία οι οποίες κάνοντας χρήση του συνδυασμού των παραπάνω τεχνολογιών έχουν αναπτύξει νέες εφαρμογές στο τομέα της αυτόνομης οδήγησης.

Παράλληλα, όλο και περισσότερο οι δυτικές κοινωνίες υοθετούν την προσέγγιση της ηλεκτροκίνησης, με συνέπεια μια συνεχής εξέλιξη του τομέα των ηλεκτρικών οχημάτων. Αυτό οφείλεται στην ολοένα αυξανόμενη τάση για την απομάκρυνση από οχήματα που κινούνται με κινητήρες εσωτερικής καύσης. Τα παραπάνω σε συνδυασμό με την τεχολογική εξέλιξη των αισθητηρίων και τη μείωση του κόστους αγοράς του σχετικού εξοπλισμού, καθιστούν όλο και περισσότερο εφικτή την ανάπτυξη πρωτοτύπων αυτόνομων ηλεκτρικών οχημάτων.

Στα πλαίσια της παραπάνω τάσης, το εργαστήριο Ρομποτικής και Ευφυών Συστημάτων του Πολυτεχνείου Κρήτης προμηθεύτηκε ένα συμβατικό ηλεκτρικό αυτοκίνητο πόλης προκειμένου να το μετατρέψει σε μια πρωτότυπη πλατφόρμα πειραματισμού στον τομέα της αυτόνομης οδήγησης. Το όχημα που επιλέχθηκε είναι το Ecocar της Ecosun, μια πλατφόρμα σχετικά χαμηλού κόστους και με δυνατότητες εύκολης μετατροπής και πειραματισμού των διαφόρων συσκεύων που είναι αναγκαίες για την αυτόνομη οδήγηση.

Συγκεκριμένα, εκτιμάται ότι πάνω στο όχημα θα ενσωματωθούν διαφορετικοί αισθητήρες και συσκευές που θα περιλαμβάνουν μεταξύ άλλων, αισθητήρα GNSS υπεύθυνο για τον υπολογισμό της θέσης του οχήματος, ένα αισθητήρα IMU που θα δίνει τον προσανατολισμό του οχήματος, ένα ή περισσότερους αισθητήρες lidar για τον εντοπισμό εμποδίων, καθως και κάμερες για την οπτική αναγνώριση των χαρακτηριστικών του περιβάλλοντος.

Το πρώτο βήμα της προσπάθειας αυτής, που αποτελεί και βασικό στόχο της παρούσας εργασίας, είναι η κατασκευή ενός λεπτομερούς εικονικού περιβάλλοντος, που θα επιτρέπει τη προσομοίωση της συμπεριφοράς του αυτοκινήτου, τη δοκιμή σε εικονικό περιβάλλον διαφόρων αισθητήρων με όσο δυνατόν μεγαλύτερη ακρίβεια καθώς και την ανάπτυξη στρατηγικών και αλγορίθμων που θα επιτρέπουν την αυτόνομη λειτουργία του προσομοιωμένου οχήματος. Στόχος είναι η συγκεκριμένη προσέγγιση να αποτελέσει το πρώτο βήμα προς την ανάπτυξη ενός ολοκληρωμένου ψηφιακού διδύμου που θα επιτρέπει τη γρήγορη δοκιμή της λειτουργία διαφόρων δοκιμών και αλγορίθμων και στη συνέχεια την δοκιμή τους στο πραγματικό όχημα.

Το περιβάλλον προσομοίωσης που επιλέχθηκε, είναι το Carla το οποίο βασίζεται στην Unreal Engine 4 και αποτελεί μια από τις δημοφιλέστερες και πιο προηγμένες λύσεις, όσον αφορά τα ανοικτού κώδικα (open source) προγράμματα προσομοίωσης για δοκιμές σε θέματα που σχετίζονται με την αυτόνομη οδήγηση. Στο συγκεκριμένο περιβάλλον έχουν πραγματοποιηθεί πλήθος ερευνών σχετικές με την αυτόνομη οδήγηση. Χρησιμοποιεί αρχιτεκτονική server-client η οποία επιτρέπει στον χρήστη καθώς η προσομοίωση τρέχει να εφαρμόζει κώδικα κάνοντας χρήση του python API.

Δημιουργήθηκε εικονικό μοντέλο του οχήματος με χρήση του λογισμικού blender και εισήχθη στο περιβάλλον προσομοίωσης, όπου είναι δυνατόν ο χρήστης να θέσει τις φυσικές ιδιότητες του μοντέλου του. Στο μοντέλο χρησιμοποιήθηκαν όλες οι σχεδιαστικές παράμετροι που είναι διαθέσιμοι από την ιστοσελίδα του κατασκευαστή και έγιναν κάποιες παραδοχές ώστε η συμπεριφορά του να είναι όσο το δυνατόν κοντύτερα στη πραγματική. Το Carla επίσης δίνει την δυνατότητα στον χρήστη να προσομοιώσει τους αισθητήρες που επιθυμεί θέτοντας τις φυσικές τους παραμέτρους. Έτσι στο μοντέλο προστέθηκαν και κάποιοι από τους αισθητήρες που είναι διαθέσιμοι στο εργαστήριο, προκειμένου να μελετηθεί η πιθανή συμπεριφορά του.

Με την ολοκλήρωση της παραπάνω διαδικασίας σχεδιάστηκε το πλαίσιο που θα επιτρέπει την ανάπτυξη ενός αυτόνομου πράκτορα που να είναι σε θέση να οδηγεί το αυτοκίνητο στον τελικό προορισμό ακολουθώντας το βέλτιστο μονοπάτι και δυνητικά να αποφεύγει τυχόν εμπόδια στο δρόμο. Για τον κομμάτι του σχεδιασμού της βέλτιστης διαδρομής, χρησιμοποιήθηκε ο αλγόριθμος A*, που υπολογίζει τη βέλτιστη διαδρομή για δεδομένο αρχικό και τελικό σημείο του προορισμού. Ο υπολογισμός της θέσης και του προσανατολισμού, γίνεται με χρήση εικονικών αισθητήρων GNSS και IMU.

Για την αναγνώριση του περιβάλλοντος χρησιμοποιήθηκε ένας εικονικός αισθητήρας lidar. Πρόκειται για ένα περιστρεφόμενο λέιζερ, το οποίο αναγνωρίζει τα σημεία ενδιαφέροντος γύρω από αυτό, δημιουργώντας ένα νέφος σημείων οπτικοποιώντας το περιβάλλον. Το νέφος αυτό, με χρήση διαφόρων αλγορίθμων, φίλτρων και τεχνικών ομαδοποίησης (clustering), παρέχει μια τελική εικόνα. Σε αυτή περιέχεται κάθε πιθανό εμπόδιο στο οποίο μπορεί να προσκρούσει το όχημα μέσα σε ένα κουτί (bounding box) για το οποίο η τοποθεσία του σε σχέση με το όχημα είναι γνωστή. Παράλληλα γίνεται χρήση μιας κάμερας, που μέσω αλγορίθμων επεξεργασίας εικόνας είναι ικανή

να εντοπίσει και να απομονώσει τις γραμμές των λωρίδων κυκλοφορίας σε πραγματικό χρόνο.

Τα παραπάνω επεξεργασμένα δεδομένα από τους αισθητήρες χρησιμοποιήθηκαν για την εκπαίδευση του αυτόνομου πράκτορα που είναι υπεύθυνος για τη πλοήγηση του οχήματος. Ο αλγόριθμος βασισμένος στη προσέγγιση της βαθιάς ενισχυτικής μάθησης που επιλέχθηκε, είναι ο Deep Deterministic Policy Gradient (DDPG) καθώς βάση της βιβλιογραφίας έχει αναπτυχθεί για προβλήματα όπου οι πιθανές ενέργειες που μπορεί να κάνει ο πράκτορας είναι μη διακριτές. Επίσης λόγο της Actor-Critic αρχιτεκτονικής του επιτρέπει την εφαρμογή του σε περιβάλλοντα τα οποία ο πράκτορας δεν γνωρίζει όλες τις πληροφορίες για αυτά. Αυτά τα δύο προτερήματα καθιστούν τον DDPG ιδανικό για προβλήματα που σχετίζονται με την αυτόνομη οδήγηση.

Δεδομένης της πολυπλοκότητας του συστήματος, η διαδικασία εκπαίδευσης πραγματοποιήθηκε μετατρέποντας το περιβάλλον προσομοίωσης σε περιβάλλον τύπου gym, που επιτρέπει την εύκολη επικοινωνία του προσομοιωτή με τον αλγόριθμο ενισχυτικής μάθησης. Η βιβλιοθήκη που χρησιμοποιήθηκε για την υλοποίηση του αλγορίθμου είναι η Stable Baselines 3 η οποία διευκολύνει την διαδικασία εκπαίδευσης και φόρτωσης μοντέλων, καθώς και την αλλαγή παραμέτρων στον αλγόριθμο.

Η εκπαίδευση του αυτόνομου πράκτορα, προκειμένου να είναι σε θέση να ακολουθεί το βέλτιστο μονοπάτι πραγματοποιήθηκε με την παραπάνω προσέγγιση. Προκειμένου να αναλυθεί η αποτελεσματικότητα, πραγματοποιήθηκε σύγκριση τριών διαφορετικών μοντέλων (όπως αυτά πρόκυπταν κατά τη διαδικασία της εκπαίδευσης). Από αυτά το καλύτερο στη συνέχεια συγκρίθηκε με τον πράκτορα που έχει ήδη εγκατεστημένο το Carla το οποίο χρησιμοποιεί μόνο δεδομένα από την προσομοίωση και όχι από αισθητήρες. Το μοντέλο που προέκυψε έχει παρόμοια λειτουργία όσον αφορά την ακρίβεια με αυτό που είναι διαθέσιμο στον προσομοιωτή, αλλά διατηρεί υψηλότερες ταχύτητες και δίνει στο αυτοκίνητο εντολές για στροφή με πιο ρεαλιστικό τρόπο. Τέλος παρουσιάζονται συμπεράσματα, καθώς και πιθανές προοπτικές εξέλιξης του προτεινόμενου συστήματος.

Abstract

The aim of this thesis, is to design and implement a test-bed which will allow the development of autonomous agents, with the ability to control and navigate urban electric vehicles, in real world scenarios, using a diverse set of on-board sensors. A simulated model of a commercial urban electric vehicle, acquired by the Intelligent Systems and Robotics Laboratory of the Technical University of Crete, has been developed and the possibility of transforming it to an autonomous vehicle has been investigated as a test case. The Carla simulated environment has been used as a tool, for developing realistic simulations and simultaneously asses the functionality of different sensors, that may be integrated to the real vehicle. To demonstrate the applicability of our approach, a gym environment has been created and integrated with the suggested configuration and based on this, we've used the data acquired by the sensors, to train an agent for the vehicle, that allows it to perform autonomous driving tasks at a basic level. Our long term vision is to create a detailed digital twin where the simulated agent will interact with its physical counterpart and will allow the rapid prototyping of algorithms which will add autonomous capabilities to the real system.

Chapter 1

Introduction

1.1 Thesis Motivation

The development of autonomous vehicles is a challenging task for for both the academia and the industry. Different levels of autonomy have been defined by the Society of Automotive Engineers (SAE) which currently categorizes autonomy in vehicles in six Levels [1] ranging from Level 0 (no automation) to Level 5 (fully autonomous operation), meaning that the vehicle may operate without human interaction/intervention. Level 5 autonomy comes closer to reality at each passing day, and the interest around this topic only increases, as more technology becomes available giving more possibilities. Simultaneously, there is also a significant trend for the adoption of electric vehicles, which will make potentially the implementation of autonomous agents less complicated, as their systems allows much easier integration of autonomous technology, than ICE (Internal Combustion Engine) vehicles.

Simulation enables the efficient, safe development and implementation of autonomous agents. The increase in computation power of modern personal computers, increased the level or realism provided to the researchers, allowing them to test new algorithms, sensor configurations and training methods on vehicles, with a high level of certainty that the results will be transferable with minor or no modifications to the real vehicles.

This process is considered an industry standard in autonomous vehicles, where all the new developments are tested and optimized in artificial environments. For this reason, many open source simulators have been created, that are ready to use for teaching and research purposes, allowing the user to test different approaches and make changes necessary with ease. For autonomous driving, a sophisticated solution is Carla [2], which since its original release has been adopted by the academia and the industry.

There are several artificial intelligence (AI) approaches that allow the development of autonomous agents in an easy and efficient manner. DDPG [3] is one of those algorithms, that was developed specifically for continuous action spaces, like the ones an autonomous vehicles have. A lot of libraries have also been developed to easily make changes and implement the algorithm on any environment.

Searching about the new possibilities that the merge of the aforementioned tech-

nologies can bring, one can find a vast literature of papers and projects. Those works range from the testing of the efficiency of different popular reinforcement learning algorithms for performing a certain task,to the development of innovative algorithms and techniques for autonomous driving that can be perfected utilizing the capabilities of the simulator. Aside from the new technologies in the fields of simulation and AI big steps have been made in electric vehicles and sensor technology. Due to those advancements, city electric vehicles along with the hardware that is required for creating an autonomous agent are much more readily available and affordable.

The aim of this thesis, is to develop a test-bed which will allow the development of autonomous agents, with the ability to navigate in real world scenarios, using diverse set of sensors. A simulated model of a commercial electric vehicle, acquired by the Intelligent Systems and Robotics Laboratory of the Technical University of Crete, has been developed and the possibility of transforming it to an autonomous vehicle has been investigated. The Carla simulated environment has been used as a tool, for developing realistic simulations and simultaneously asses the functionality of different sensors, that may be integrated to the real vehicle. A gym environment has been created and integrated with the suggested configuration and based on this we've used the data acquired by the sensors, to train an agent for the vehicle, that will allow it to perform autonomous driving tasks at a basic level. Our long term vision is to create a detailed digital twin where the simulated agent will interact with its physical counterpart and will allow the rapid prototyping of algorithms which will add autonomous capabilities to the real system.

1.2 Thesis Outline

The rest of this thesis is organised as follows.

In Chapter 2, we present all the theoretical background essential, for the development of the thesis, along with a short description of prior work in the field.

Chapter 3 comprises from an overview of the simulator and its main features, along with a short description of the actual vehicle and how it was modeled and integrated to the simulated environment.

Chapter 4, deals with the different devices essential for the autonomous operation and their respective digital modules. We describe in detail the process of gathering and processing the data, in a meaningful manner, so that can be used for path planning and obstacle avoidance.

Chapter 5, provides an an overview of the gym environment used for the training of autonomous agent, along with the actual algorithm used. The applicability of the proposed approach is demonstrated through extensive simulations and comparisson of the related results.

In the final chapter 6 we present some concluding remarks, and discuss possible future extensions to the proposed approach.

Chapter 2

Background work and Literature Review

Autonomous driving is a challenging task which requires the adoption of different algorithmic solutions for performing among others path planning, lane detection, obstacle detection etc. This chapter includes the mathematical background and the steps of those algorithms for a better understanding of how they work and why they are selected for certain tasks. Apart form that, prior work on developing autonomous agents is presented.

2.1 A* search algorithm "A-star"

The A* algorithm as described in [4], is an algorithm that calculates the optimal (shortest) path between an initial and a final point. It is considered "optimal" as it looks for the least cost solution for a given problem and "complete" because it calculates all of the possible solutions.

The algorithm starts from the initial point and in each iteration it decides to which point to go next, based on two parameters. The first one is "g" and it represents the cost to move from the initial point to the one that is currently examined as the potential next point. The second one is "h", which is referred to as heuristics and is the estimate cost to move from the point currently being examined to the final point. The parameter "h" is an estimation and depending on the problem that can be calculated with the following methods:

- If only for directional movement is allowed the Manhattan Distance (2.1) is used.

$$\begin{aligned} h\_Manhattan = &|next\_point\_x - final\_point\_x| \\ &+ |next\_point\_y - final\_point\_y| \end{aligned} \tag{2.1}$$

- If eight directional movement is allowed the Diagonal Distance (2.2) is used.

$$\begin{aligned} d_x &= |next\_point\_x - final\_point\_x| \\ d_y &= |next\_point\_y - final\_point\_y| \\ h\_Diagonal &= D \cdot (d_x + d_y) + (D_2 - 2 \cdot D) \cdot min(d_x, d_y) \end{aligned} \tag{2.2}$$

With D being the length of each node of the grid and D_2 being the diagonal distance between each node of the grid.

- If movement in any direction is allowed the Euclidean Distance (2.3) is used.

$$h\_Euclidean = \sqrt{(next\_point\_x - final\_point\_x)^2 + (next\_point\_y - final\_point\_y)^2} \tag{2.3}$$

At each step the algorithm calculates the "f" (2.4) for each point that can be reached through a route from a point that has been previously discovered. If a smaller "f" is found for a point through a different route than the one found in previous step, the value of "f" is updated. In each step the algorithm chooses the smallest "f" and considers the route that it was achieved through the most optimal path thus far. Finally the algorithm ends when the end node has been reached and the route that lead up to it is the solution for the path finding problem.

$$f = g + h \tag{2.4}$$

2.2 Density-Based Spatial Clustering (DBSCAN)

DBSCAN according to [5], stands for density-based spatial clustering of applications with noise. Given a set of points in some space, it groups together the points that are closely packed together (points with many nearby neighbors), while marking as noise the points that lie in low-density regions (whose nearest neighbors are too far away).

To do this two parameters are considered:

- "eps" which is the maximum distance two points can have in order to be considered neighbors.
- "mimPts" is the minimum number of neighboring points needed to define a cluster

Based on them the points of the given space can fall under three different categories, which are:

- Core Point: The points that have at least "minPts" other points within their radius of length "eps"
- Border Point: The points that are within the "eps" radius of a core point but they do not fulfill the criteria to be a core point themselves.
- Outlier: The points that fulfill neither the Core Point nor the Border Point criteria and are hence considered as noise.

With those concepts established the algorithm in its first step selects a random point from the given space and checks if it is a Core Point. If it is not, it marks it as noise and picks another one, if it is it a cluster formation starts and all of its neighboring points are added to the cluster. In the next steps it checks if those neighboring points are also Core Points and their neighbors should also be added in the cluster

or Border Points, this repeats for every newly added point on the cluster. After no more Core Points can be found the cluster is closed and a new random point is selected to start a different cluster. The algorithm ends when all the points of the given space have been visited. It should be noted that a point that was marked as noise can be visited again from a different Core Point and added to a cluster.

2.3 RGB and HSV Color Spaces

RBG color space stands for Red Blue Green and is considered the default and most commonly used color space in a variety of applications like cameras and image software [6]. It is an additive color model meaning that all color except the three basic ones (red, blue and green) are formed as a combination of different intensities of the three basic one. A visualization of the color space can be seen in figure 2.1 The desired intensity for each basic color is chosen from a scale that has a minimum value of 0 and maximum value of 255

It should be noted that the maximum value in all of three colors returns the color black white and the minimum value returns the color black.

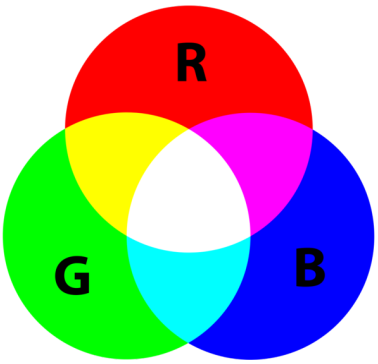


Figure 2.1: The RGB Color Space [6]

An iteration of this color space is the BGR, with the only difference being the order in which the colors are represented. Specifically Blue Green Red than Red Blue Green.

The raw data output of the camera sensor are pixels in the RBG colorspace format, which are easy to be visualized to get a video recording of what the camera sees.

Hue Saturation Value (HSV) color space is an alternative representation of the RBG and BGR color spaces, that aims to mimic more closely the way the human eyes perceives color [7]. This is achieved by having the colors of each hue being arranged as slices of a cone, where its central axis represents the neutral colors, the grayscale. The shape of HSV color space is presented in figure 2.2

Hue can take values from 0 to 360 because it represents degrees, the Value parameter is a percent from 0 to 100 and represents how dark or light a color is while Saturation is also a percent from 0 to 100.

Using this color space, picking a range of similar colors, colors that have a similar Hue but are darker or lighter, is much easier than it would be by using the

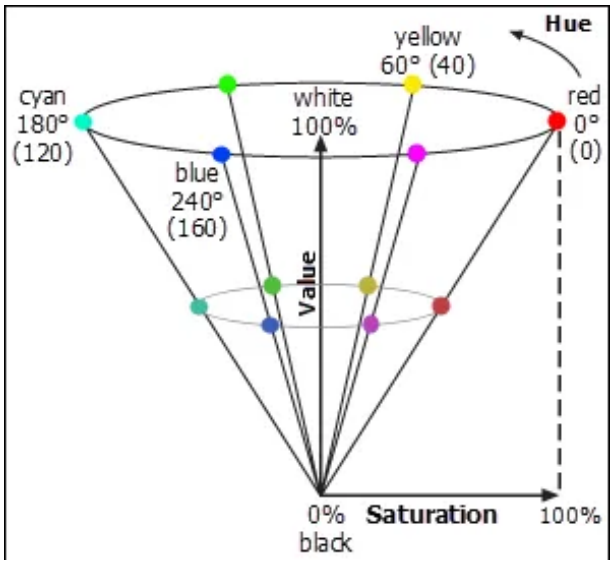


Figure 2.2: The HSV Color Space [8]

RBG one because the hue can stay the same while the Saturation and Value parameters change, thus making it the preferable color space for color segmentation purposes.

This colorspace can be used as a more efficient way to describe a pixels color for image masking.

2.4 Gray Scale Filtering

Gray Scale Filtering is used in images to reduce calculation time when the color of the image is not important for the calculations. In gray scaled images the color of the pixels is encoded with only one parameter which is called Luma and not three like it does in RBG and HSV color spaces [9].

For the transformation of an image from RBG to Gray Scale a function is applied to calculate the luminosity (Luma) of each pixel, which consists of the sums of the Red, Blue and Green values each one multiplied with a different constant.

In OpenCV the exact formula that is used from the transformation of an RGB image to Gray Scale is presented in equation (2.5).

$$Luma = 0.299 \cdot R\_Value + 0.587 \cdot G\_Value + 0.114 \cdot B\_Value \tag{2.5}$$

The same function is also used for the BGR Color Space.

This filter can be used for reducing the image noise during image processing i.e. for lane detection.

2.5 Gaussian Blur

In image processing Gaussian Blurring is the process of blurring an image with the use of a Gaussian function. It is used to reduce noise and detail (thus speeding up further processes) in an image and provides a smooth blurring effect [10].

The method uses a two dimensional Gaussian function Eq. (2.6), to calculate the transformation it will apply on each pixel of the image.

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.6}$$

Where x is the distance from the origin in the horizontal axis, y is the distance from the origin in the vertical axis, and σ is the standard deviation of the Gaussian distribution. It should be mentioned that the origin of those axis is the center (0,0).

It can be used for noise reduction during lane detection pipeline.

2.6 Canny Edge Detection

Canny Edge Detection is a multistage algorithm that is widely used in computer vision systems to extract crucial information about the structure of an image and remove the excess noise, thus speeding up the computational processes and reducing the margin for errors. The resulting image that will only contain the edges of the different shapes of the original image can be then used as a much clearer input for other detection algorithms [11], [12].

The process of edge detection following this technique is described bellow:

1. The first step is to remove the noise of the original image that can result in detection mistakes. This is achieved with the application of a Gaussian filter kernel, that smoothes the original image.

   The equation for the Gaussian filter kernel that is used, with a size of (2k+1) $\times$ (2k+1) is presented in 2.7.

$$H_{ij} = \frac{1}{\pi\sigma^2} exp(-\frac{(i-(k+1))^2 + (j-(k+1))^2}{2\sigma^2}) \\ \forall 1 \leq i, j \leq (2k+1) \tag{2.7}$$

   It should be noted that the larger the size of the Gaussian filter is, the lower the sensitivity of the detector will be to noise. The localization error to detect the edge will increase as the filter increases in size.

2. The intensity gradient of the image must be calculated, where the smoothed image taken from the Gaussian filter is filtered again with a Sobel kernel in both the vertical and the horizontal directions. If A is defined as the input image this process returns two new images (G_x) and (G_y), which contain the

first derivative in the horizontal direction and vertical direction respectively. The computations for the two new images are presented in formula 2.8

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \cdot A \quad \text{and} \quad G_y = \begin{bmatrix} +1 & +2 & +1 \\ +2 & 0 & -2 \\ -1 & -2 & -1 \end{bmatrix} \cdot A \tag{2.8}$$

where $\cdot$ denotes the 2-dimensional signal processing convolution operation.

Based on those two new images the edge gradient and direction for each pixel can be calculated with the formulas in 2.9

$$\begin{aligned} Edge\_Gradient(G) &= \sqrt{G_x^2 + G_y^2} \\ Angle(\theta) &= tan^{-1}(\frac{G_y}{G_x}) \end{aligned} \tag{2.9}$$

The direction of the gradient found is always perpendicular to the edges, for that it is rounded to one of four angles representing vertical, horizontal and two diagonal directions.

3. Next a full scan of the image is performed to remove any unwanted pixel that does not make up an edge. This is achieved by checking if each one of the pixels is a local maximum in its neighborhood in the gradient direction.

   As a visual example, in Figure 2.3 the point A stands on the edge and is on the vertical direction. The points B and C are also in the same direction. So point A is checked with point B and C to see if it forms a local maximum. If it does it passes to the next stage of the algorithm, while if it does not it gets removed, by getting a value of zero. This results in a binary image with thin edges.

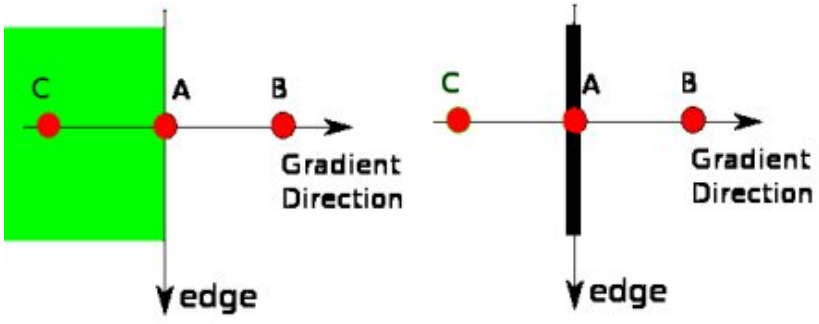


Figure 2.3: Non-maximum Suppression [11]

4. The final stage of the procedure is to decide which edges are actual edges and not an error. This is done by the use of two threshold values, the minVal and the maxVal. If the edge in question has a higher intensity gradient than the value of maxVal it is trully an edge whilst if it has lower than the value

of minVal then it is discarded as an error. The ones that their intensities lie between the two values are categorised based on their connectivity. That means that if they are connected pixels that are confirmed edges, they are also considered as part of the same edge, differently they are also discarded.

This is visualized in Figure 5.23, where the edge A is above the maxVal value, so is confirmed as an edge. Edge C is below the maxVal value but since it is connected to edge A it is also considered as part of the edge. On the other hand edge B, although it is above the minVal value , it is not connected to any confirmed edges, so the algorithm discards it. This example also highlights the importance of the correct selection of the minVal and maxVal values depending on the desired outcome.

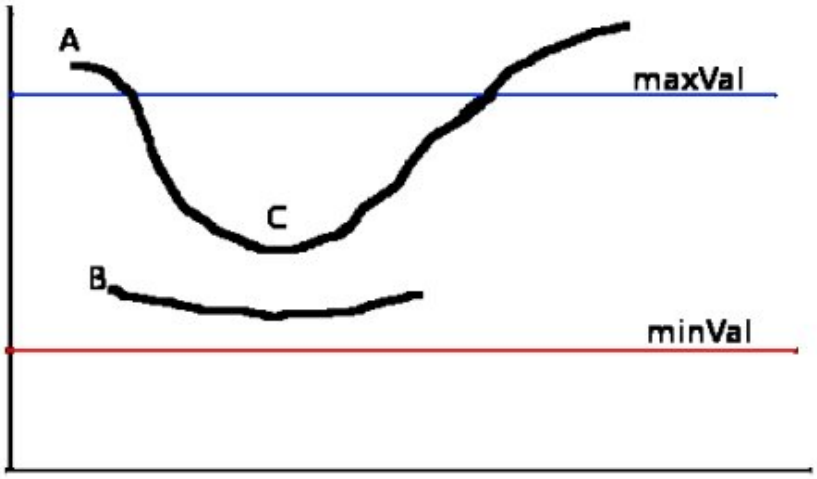


Figure 2.4: Hysteresis Thresholding [11]

Canny Edge detection can be utilized for finding the outlines of the traffic lines during lane detection pipeline.

2.7 Hough Line Transform

The Hough Line Transform [13], is a technique developed to detect and extract simple shapes, with the most common being straight lines from images. Usually it is used, after the implementation of edge detection, because edge detection by itself can result in errors as pixels from the desired curve may be missing or noise pixels may exist. Hough Line Transform solves this problem by using an explicit voting procedure over a set of parameterized image objects to create possible groupings of the edge points that consist a curve.

As it was mentioned the most frequent use of the method is for the detection of straight lines in a given image, which is achieved by following the steps below.

1. While in the Cartesian Coordinate System a line is expressed with the formula $y = m \cdot x + b$, where m is the slope of the line, this system would cause computational problems when coming across vertical lines because their slope

has an infinity value. For this reason the Polar Coordinate System is used, where a line equation is written in the form shown in Eq. 2.10.

$$y = -(\frac{cos\theta}{sin\theta}) \cdot x + (\frac{r}{sin\theta}) \tag{2.10}$$

By rearranging the terms of Eq. 2.10 we get Eq. 2.11

$$r = x \cdot cos\theta + y \cdot sin\theta \tag{2.11}$$

Where r and θ are the distance and the angle from the center of the axis x and y. A line that is described by r and θ can be seen in Figure 2.5.

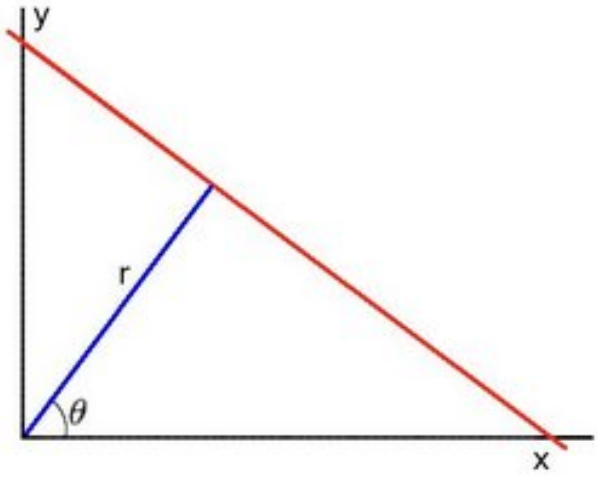


Figure 2.5: r and θ in the Polar Coordinates [13]

2. To generalize this concept for every (x_0, y_0) point in the (x,y) plain, there is a family of lines that pass through it that each one of them can be described by a unique pair of (r_θ, θ)

   So the general equation that describes a family of lines passing through the point (x_0, y_0) in the Polar Coordinate System is the one shown in Eq. 2.12.

$$r_\theta = x_0 \cdot cos\theta + y_0 \cdot sin\theta \tag{2.12}$$

3. If for a given (x_0, y_0) point the r and θ of the lines that pass through are plotted in the (r,θ) plain, the resulting graph will be a sinusoidal curve. A visual example of this can be seen in Figure 2.6 for $x_0 = 8$ and $y_0 = 6$.

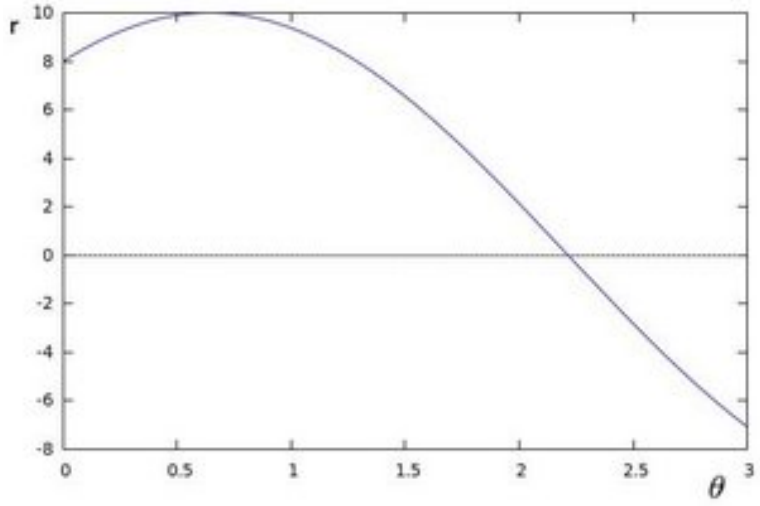


Figure 2.6: Plot of Lines [13]

It should be mentioned that only points that have $r > 0$ and $0 < \theta < 2\pi$ are considered for the plot.

4. Every pixel in the image has a x and a y coordinate, if the r - θ plot is created for each one of them some of the curves will intersect with each other. A reason why the results of the edge detection algorithm are used as input for the Hugh Transform is to avoid doing those calculation for every single pixel of the image but only for the ones that are part of potential lines. If the curves of two different pixels intersect, it means that the pixels can be potentially connected with a straight line. To visualize this, in Figure 2.7 the plot for two more points is added: $x_1 = 4$, $y_1 = 9$ and $x_2 = 12$, $y_2 = 3$.

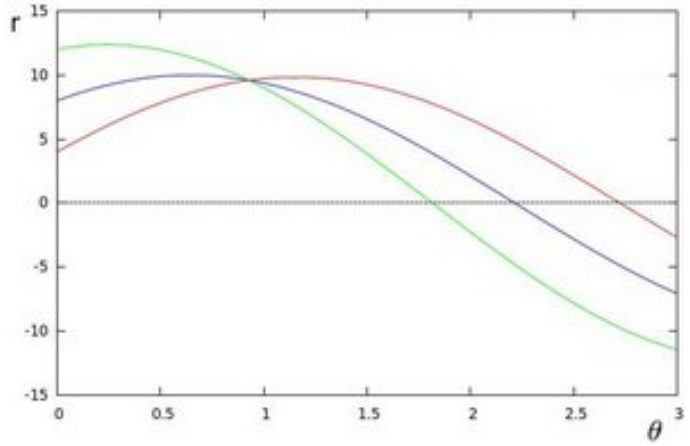


Figure 2.7: Two More Lines Added [13]

The three plots are intersecting in one single point with coordinates $(0.925, 9.6)$ in the (r, θ) plain. Those coordinates are the parameters of the line that passes thought the three different points simultaneously.

5. According to this methodology Hough Line can detect a line based on the number of intersecting curves. The more curves intersecting in one point in the (r,θ) plain means that the line represented by that point can connect

more pixels. The algorithm requires a threshold of the minimum number of intersections that are needed, thus number of points that belong in the same line to detect a line. This depends on the user and the type of problem they wish to apply the algorithm on.

Hough Line Transform can be utilized, for detecting the straight lines that make up the traffic lines from the edges provided by Canny Edge Detection during the line detection pipeline.

2.8 Prior Work

A plethora of works already exist in the literature that deal with testing already existing and developing innovative autonomous agents using simulation. Simulators make the part of testing different training algorithms or different variations of the same one much more consistent as they allow the user to recreate the same scenario over and over, while keeping constant all the different variables that can change and affect the training process in the physical world.

Conducting tests on an environment that can guarantee that the training conditions will remain unchanged provides much more clear outcomes in an efficient manner. This is illustrated in [14], where this particular ability to test efficiency of different deep reinforcement learning algorithms performing the task of following a predetermined route under the same circumstances, in the controlled environment of the simulation can provides clear and cohesive conclusions as to which one performs the best.

Those environments also allow the development of new variations of already existing algorithms by making possible the quick change of the algorithm hyperparameters and the immediate test of those testing in real time on the simulation. With this methodology in [15], a modification of DDPG is developed that trains a car-following model with a real-world human driving experience. This approach is also innovative because it takes advantage of the easy integration of data provided by real human driving data sets. This not only makes the training process faster but also encourages the vehicle to adopt a more human-like style of driving that many agents lack due to the fact that it is difficult to be achieved through only reward functions.

Under the same category of modifying deep learning algorithms to perform better on specific task, in[16], a DDPG variant is developed so the vehicle can receive external control for which path to take. The agent aims to take information from the driver about navigation while providing the smooth and safe driving an autopilot provides. The very nature of the algorithm that requires the cooperation of a human driver and the deep learning algorithm in order to learn the optimal policy would not be very dangerous if not impossible if the option of simulating the whole in a simulator was not possible.

Moreover the advantage that the simulators provide of easy integration of many libraries for data processing and visualisation, provides the opportunity for researcher to create robust agents that cover the whole process of autonomous driving, from data collection and processing from multiple sensors to the training of the algorithm. This is demonstrated in [17] where an interpretable end-to-end urban autonomous

driving agent was developed that utilizes a camera and a Lidar sensor. This agent simultaneously uses, the data collected from the camera and the Lidar to train, while also visualizing them in a way that provides the viewer with insides of how the agent sees its environment and interprets that information.

This ability to visualize the perception of the agent and all of the parts of the data processing in real time, other than simplifying the process of showcasing the work the agent does and its results, also provides a mechanism for the researcher to understand fast and easy if a part of the agent works in an unexpected way and make the necessary changes to modify it.

Chapter 3

Simulated Model

3.1 The Carla Simulator

3.1.1 Carla Overview and Installation

Carla Simulator [2], is an open source simulation platform developed for the Unreal Engine 4 with the purpose of supporting development, training, and validation of autonomous driving systems. Its open source nature provides the user ready to use code, protocols and digital assets such as urban layouts buildings and vehicles that are ready to use in the simulation and are either created by the developers or other users. Because it is build in the Unreal Engine there is a lot of flexibility in the parameters of the sensors, the environment, the different actors, which includes the dynamics of the vehicles and the maps. For the aforementioned reasons it is considered a state-of-the-art simulation platform that is both used in the academia and the industry for developing autonomous driving systems.

According to the documentation [18], [19], Carla can be installed in two different ways. The user can either download a packaged version that runs as an .exe program or build it from source. The main difference between the two versions, is that the packaged version is much more easier to install and has less requirements from the built version. On the other hand it offers less flexibility as advanced customization and development options that require use of the Unreal Engine editor are not possible, since it executes as a separate program without opening the Unreal Engine Editor. The fact that the packaged version does not require the full download and usage of the Unreal Engine 4 is the reason for its reduced memory size and GPU requirements.

Because the vehicle that is used in the simulation does not exist in the base version of Carla, nor its assets can be found in the internet it needs to be simulated into the Unreal Engine Platform in order to be used in Carla Simulation. This can be achieved only in the built version of Carla so this is the version that is used.

Carla can be build and run in either the Windows or the Linux operating system. From those two options the later was selected, because the architecture of the Linux operating system is much more efficient for running Unreal Engine and Carla and provides easier methods to download and implement different python libraries and

run scripts than Windows. Specifically the Ubuntu 18.04 Linux distributor was chosen, because it is mentioned as the optimal one in the Carla documentation. The version of Carla that was selected to be built, was 0.9.13 as it was the latest and most stable version at the time of development.

After checking that the system fulfils the system requirements the building process continues by installing the software requirements. All the steps of the building process can be found as commands ready to be pasted in an Ubuntu terminal in the Linux Build page in Carla's documentation site.

Carla comes with an integrated Python API that can be installed with either Python 2 or 3. The selected version of Python was the 3.6 because it is the optimal choice for the Carla version and libraries that are used.

After Installing the Python API and the dependencies the Unreal Engine must be installed. The version of Unreal Engine that Carla runs on is a modified fork of Unreal Engine 4.26, which contains patches specific to Carla for its optimization.

After having the modified version of the Unreal Engine set up the next step is to build Carla in it. The correct assets need to be downloaded depending on the version of Carla and the Unreal Engine environment variable, must be set for Carla to find the correct installation of Unreal Engine in the files and avoid possible errors.

Finally the Python API client must be compiled, granting control over the simulation. It must be compiled only the first time Carla is build and again only if any updates are performed. After the client is compiled, the user can freely start to run scripts and interacting with the simulation.

Typing the the command "make launch" in the Carla root folder boots up the Unreal Engine and opens Carla as an Unreal Project in the editor. In Figures 3.1 and 3.2, the loading screen of Unreal Engine and the main window of the Editor when it opens are seen respectively.

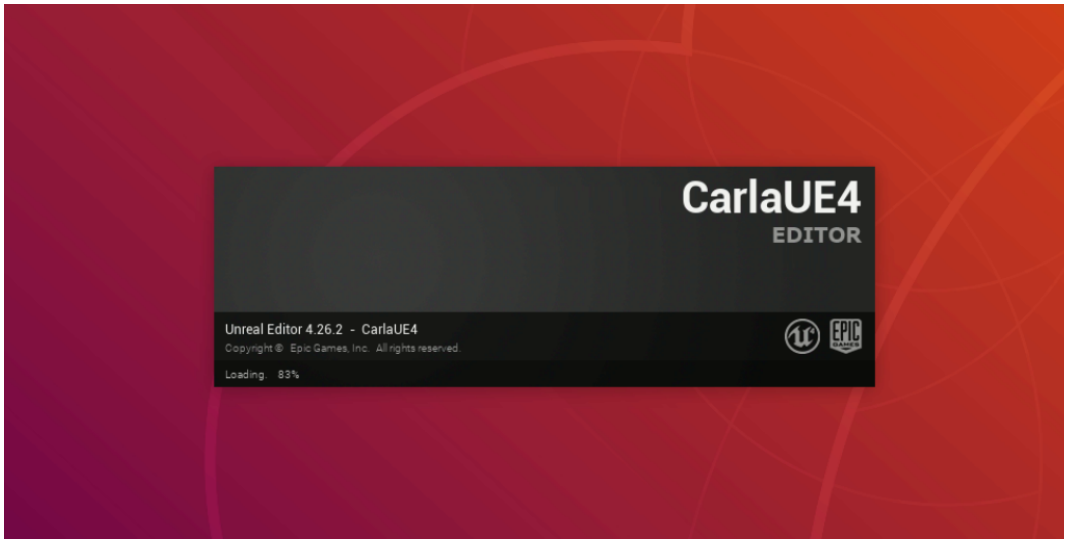


Figure 3.1: Unreal Engine Editor Loading

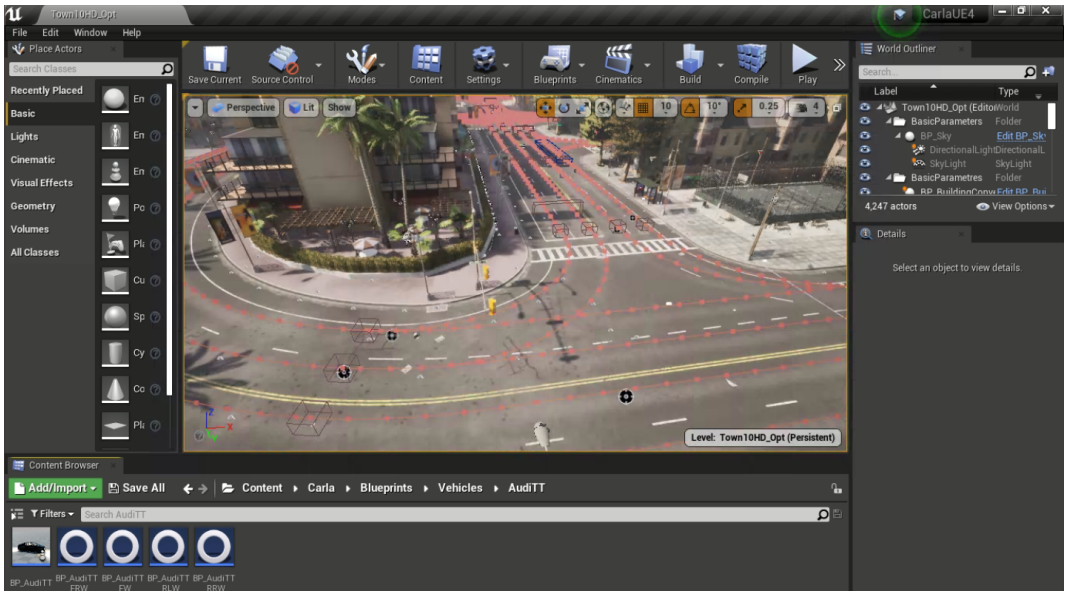


Figure 3.2: An View of the Editors Main Window

The actual simulation starts by pressing the play button that is located on the upper right section o the UE4 Editor as presented in Figure 3.2 and an image of the simulation running is depicted in Figure 3.3.

Figure 3.3: The Simulation Running on the UE4 Editor

3.1.2 Notable Features of the Editor

As it was mentioned, because in the built version Carla opens as an Unreal Project in Unreal Engine, its environment shares the same features as any other project in Unreal Engine.

In the middle, inside the green border as seen in figure 3.4, a camera provides live footage of the simulation running. The user can freely move the camera using the keyboard and the mouse to see any part of the map. This camera is not considered to be a Carla Sensor but rather the spectator of the simulation.

On the right side in a bar, inside the red border as seen in figure 3.4, a list of all the entities that make up the current map exist. The list includes the static meshes, that are elements that cannot affect other elements of the simulation, such as the buildings the trees and other cosmetic elements. The actors, which are elements that can interact with other elements i.e. the vehicles, the walkers and also sensors, traffic signs, traffic lights and the spectator belong in this category. The actors are dynamic objects whose different properties can change as the simulation run and they interact with each other. For example the speed of the vehicles and the walkers and the color of the traffic lights. The actors can be spawned or destroyed as the simulation runs so the list is updated in real time when such changes occur. Finally the list also includes the element of the sky and the different light sources that are applied on the map.

On the bottom of the UE4 window there is a another bar, inside the yellow border as seen in figure 3.4, where the content of the folders of Carla appear. There, the different maps can be found, where the user can change the map of the simulation by clicking on them without the need to run a command in the script. In another folder the static elements used in the maps can be found where the user can potential add more in the simulation, change their properties or potentially import new ones. The dynamic elements, which are called actors, are located in the folder called blueprints. Blue prints are unique layouts that allow the user to smoothly incorporate new actors into the simulation. They are pre-made models with animations and already defined attributes, where some of them are modifiable by the user. Vehicle color, amount of channels in a lidar sensor, a walker's speed, the torque curve of a vehicle are some of the modifiable attributes.

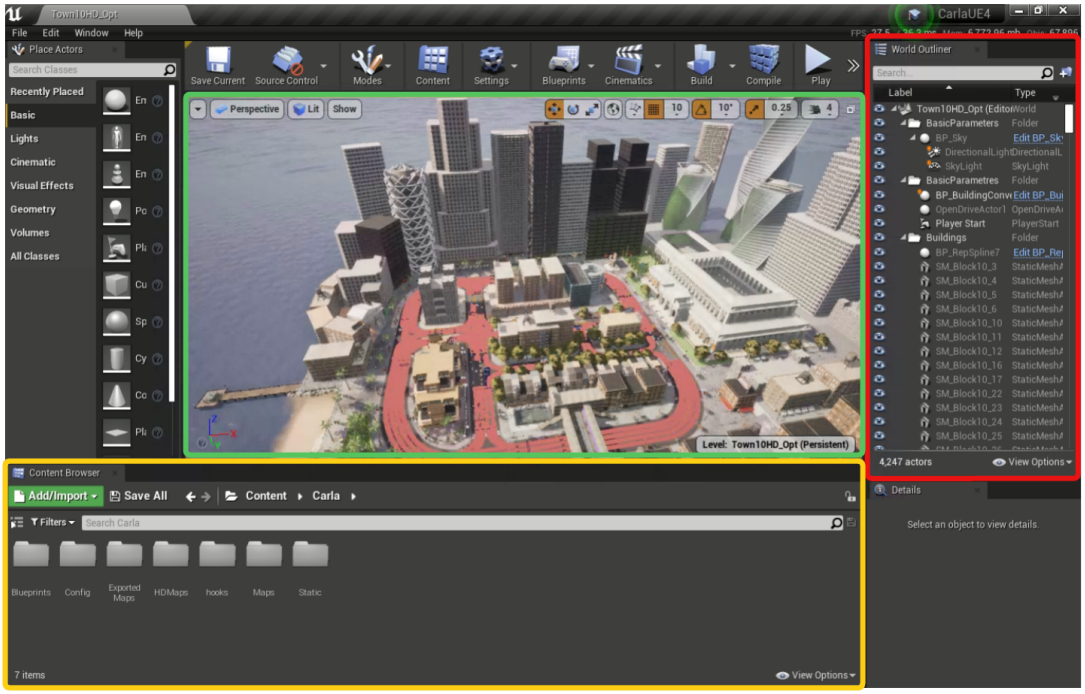


Figure 3.4: Locations of Editors Notable Features

3.1.3 Carla Foundations

Carla uses a server-client architecture to operate [2]. More specifically the server part of the simulator is responsible for the task related to the rendering of the different elements of the simulation, the computation of the physics and the updating of the state of the environment and its actors in real time. On the other hand the client part is constituted by the modules that control the logic of actors on scene and setting world conditions. For the interaction of the two parts the CARLA API is responsible, which can be either in Python or C++.

The server-client communication happens through terminal commands or scripts run through the terminal, many clients can run at the same time and to connect one to the server the following command is used:

```
client = carla.Client('localhost', 2000)
```

Finally Carla can run in two different modes, *synchronous* and *asynchronous mode*. The difference is that in asynchronous mode the CARLA server runs as fast as it can, executing the client request as fast as possible, while in synchronous mode the client tells the server when to update, this information is taken from the script where the user and internal tick timer are defined. For this thesis, asynchronous mode is used as the synchronous mode has problems with specific libraries that are used.

3.2 Modelling the vehicle

3.2.1 Ecocar

The vehicle that was selected to be simulated and have the autonomous agent developed for is the Ecocar [20], which is a two seat electric urban vehicle imported in Greece by Eco Sun. Its interior layout and powertrain design allows the easy implementation of the simulation findings on the real vehicle. The technical specification of the vehicle, specifically the version of the vehicle is the High Speed Line are:

- External dimensions: 2245 × 1290 × 1570mm
- Total weight (with battery): 600kg
- Engine Power: 7,5kW
- Maximum speed: 80km/h
- Front Suspension Type: McPherson knees
- Rear Suspension Type: Semi-rigid shaft bridge type
- Front/Rear Braking System: Disks
- Horsepower: 10hp

In Figures 3.5, 3.6, 3.7 a view of the vehicles front, side and back view with its measurements are presented.

Figure 3.5: Front View of Vehicle [21]

Figure 3.6: Side View of Vehicle [21]

Figure 3.7: Back View of Vehicle [21]

3.2.2 Torque Curve

A very important parameter for the simulation of a vehicle is its Torque Curve. This curve represents the different values of the torque of the vehicles motor (Nm) in different speeds (rpm). The issue that arises here,is that this curve isn't available, so it is estimated by taking into consideration a couple of different parameters that are already known.

The Internal Combustion Engines and Electric Motors have different types of torque curves. In Internal Combustion Engines the torque value ascends slowly as the value of speed increases before reaching the peak point and start descending rapidly. For this reason vehicles with Internal Combustion Engines require a transmission so they can achieve peak torque in different speeds and a clutch, since at zero speed there is no torque, so they can reach a certain torque before moving the vehicle.

In the case of an Electric Motor the vehicle has the advantage that maximum torque is reached as soon as the motor starts spinning. This torque is kept until the motor reaches the corner speed where it starts to decrease. Figure 3.8 illustrates the Torque/Velocity graph that compares the torque curve of an Internal Combustion Engine and an Electric Motor. It can be seen that the Internal Combustion Engine utilizes different gears in order to stay in peak torque and match the curve of the electric motor.

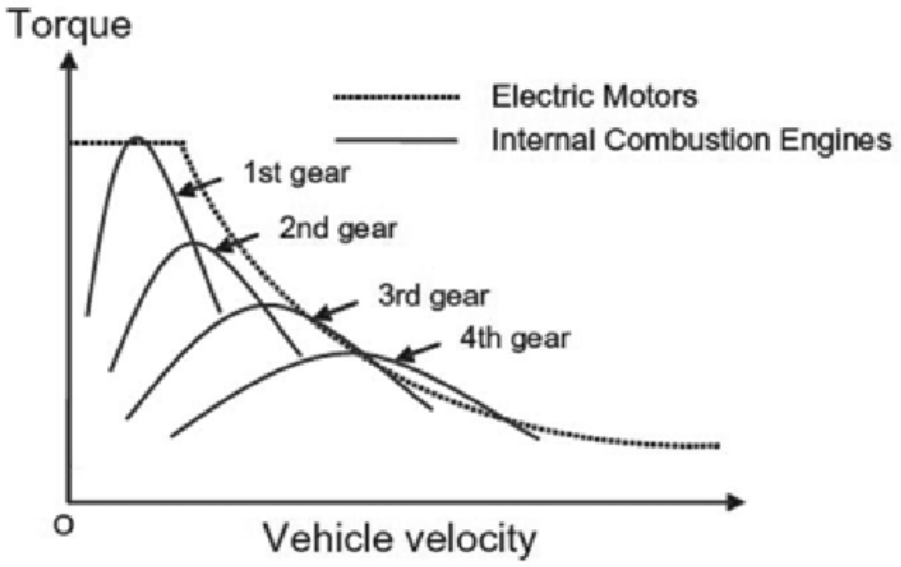


Figure 3.8: Internal Combustion Engine and an Electric Motor Torque Curves [22]

For the case of the Ecocar it is know that the vehicle uses an Electric Motor and has engine power that equals to 10 hp or 7.5kW. To simulate its Torque Curve, it is important to estimate the maximum torque speed. The formula for calculating the torque for a known engine power and speed is pesented in Eq. 3.1.

$$Torque(Nm) = 9.5488 \times Power(kW)/Speed(RPM) \tag{3.1}$$

To calculate the max torque it is considered that its value is reached at the speed of around 1000 rpm. Using this value of speed in equation 3.1 it is 71.21 Nm. The max torque is also estimate to remain unchanged until the speed reaches 2500 rpm after that it slowly declines. The maximum rpm that the vehicle can reach is considered to be 3500rpm. According to those parameters the torque curve for the vehicle is created on the Unreal Editor in the mechanical setup section by giving the aforementioned points as input for the curve. The final curve is presented in Figure 3.9.

Figure 3.9: Final Torque Curve of Ecocar

3.2.3 Simulation of the Vehicle

Carla gives the possibility of adding new vehicles in the simulator, if the built version from source is installed [23] that allows the usage of the Unreal Engine Editor. There are specific steps that a user has to follow, to properly add a vehicle.

Initially, we have to import or create the vehicle's mesh in a 3D modeling software. Because no previous meshes of the specific vehicle exist, a creation of one was necessary. The software that was used is Blender, which is an open source 3D modeling software, used in the creation of 3D graphic. It is widely used in the industry for modeling, rigging, fluid simulations, animation and rendering of assets used in different application such as video games. The main point of the vehicle model is to have dimensions and physics as close as possible to the original vehicle, details on the appearance do not play a role in how it acts inside the simulation so the attention to the appearance was compromised to save time.

The method with which the model was created, was to import three pictures from three different angles of the Ecocar in Blender. The pictures used are the ones already presented in figures 3.5, 3.6 and 3.7. We had to adjust them on the zx, zy, zx from the (opposite direction) plains respectively and the images where scaled to have the exact measurements of the real vehicle. Then an orthogonal parallelepiped mesh was adjusted, so that it fits the outlines of the vehicle set by the pictures. The mesh was further cropped with the free selection tool that Blender provides so that it takes the shape of the Vehicle, as shown in figure 3.10.

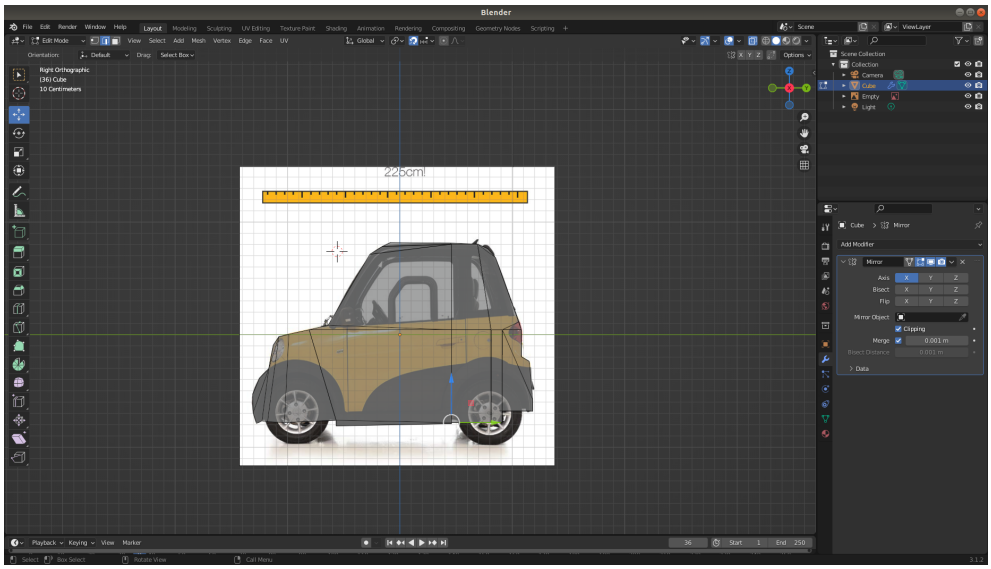


Figure 3.10: Mesh Being Adjusted to Fit the Vehicle Image

This procedure was followed for all of the pictures, which resulted in a satisfying outline mesh of the vehicle, that also has the correct measurements. It should be mentioned that for the front and back side, the mesh was mirrored with the mirroring tool of Blender, so changes that happen in one half also happen in the other, in order to have perfect symmetry. The outline mesh is presented in Figure 3.11.

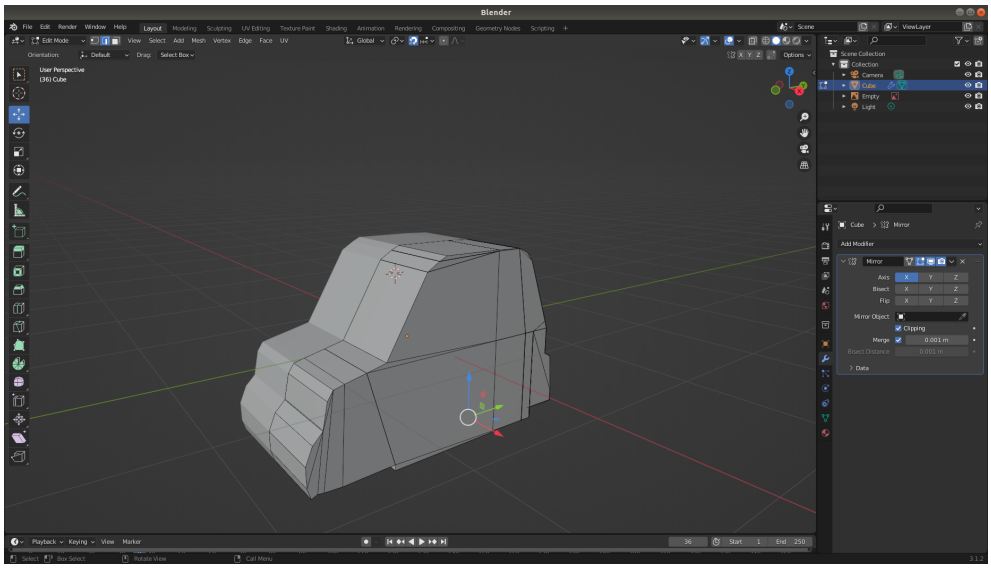


Figure 3.11: Outline Mesh of the Vehicle

After the outline mesh was created more details where added, such as the openings for the wheels, the window mirrors and the dents for the windows, the headlights, the grills and the licence plates. Finally a set of detailed wheels was imported and added to the model and some colors and textures where also added. The final model is presented in Figure 3.12.

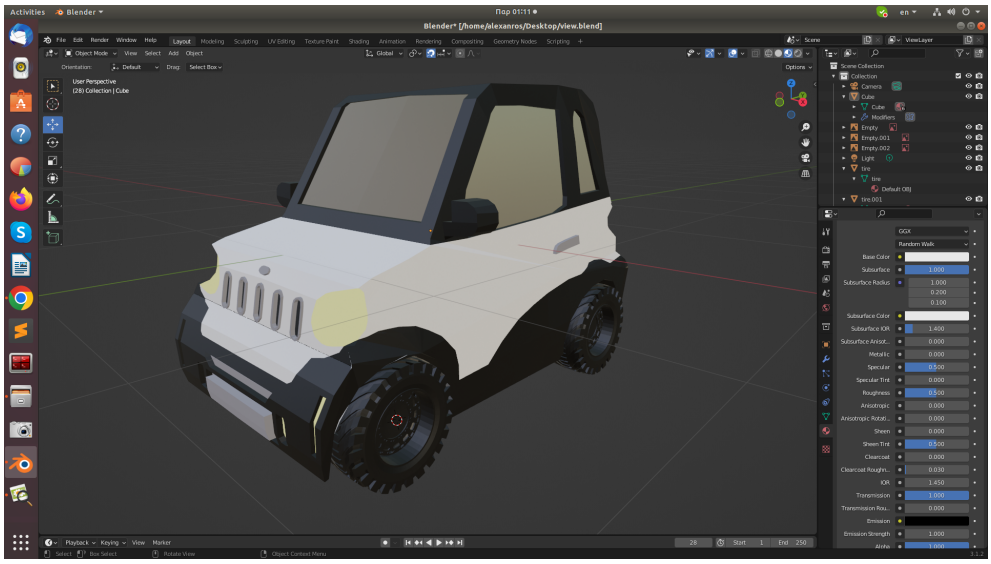


Figure 3.12: Final Model of the Vehicle

After the mesh is created, the base skeleton must be imported and the bones of the skeleton must be binded on the wheels and the main body of the vehicle. This procedure allows the wheels to rotate along the main body, as movement is applied to the vehicle in the simulation, like one object and not five separate ones. In Figures 3.13 and 3.14 the base skeleton can be seen as it is first imported and as it is attached and binded onto the vehicle respectively.

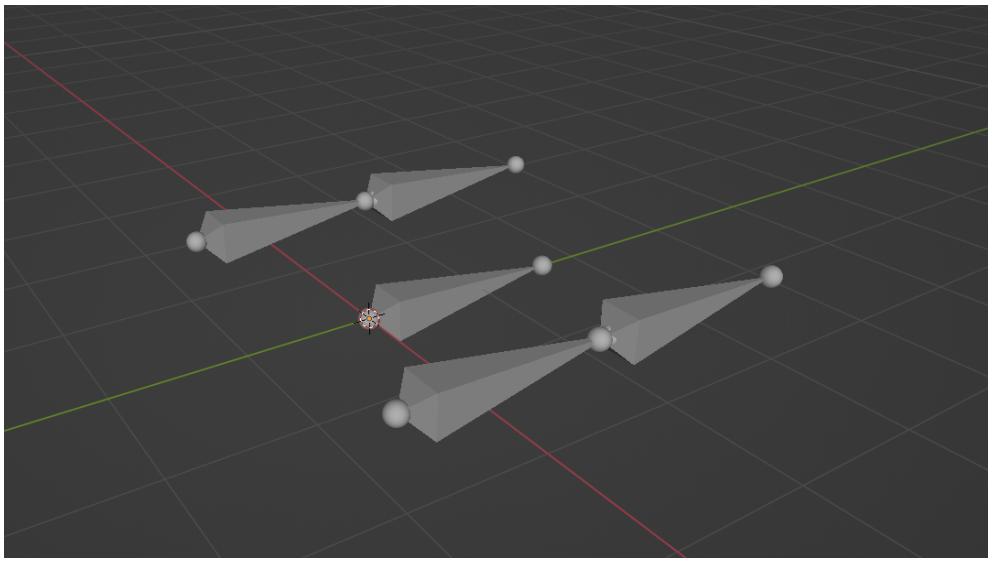


Figure 3.13: The Base Skeleton of the Vehicle

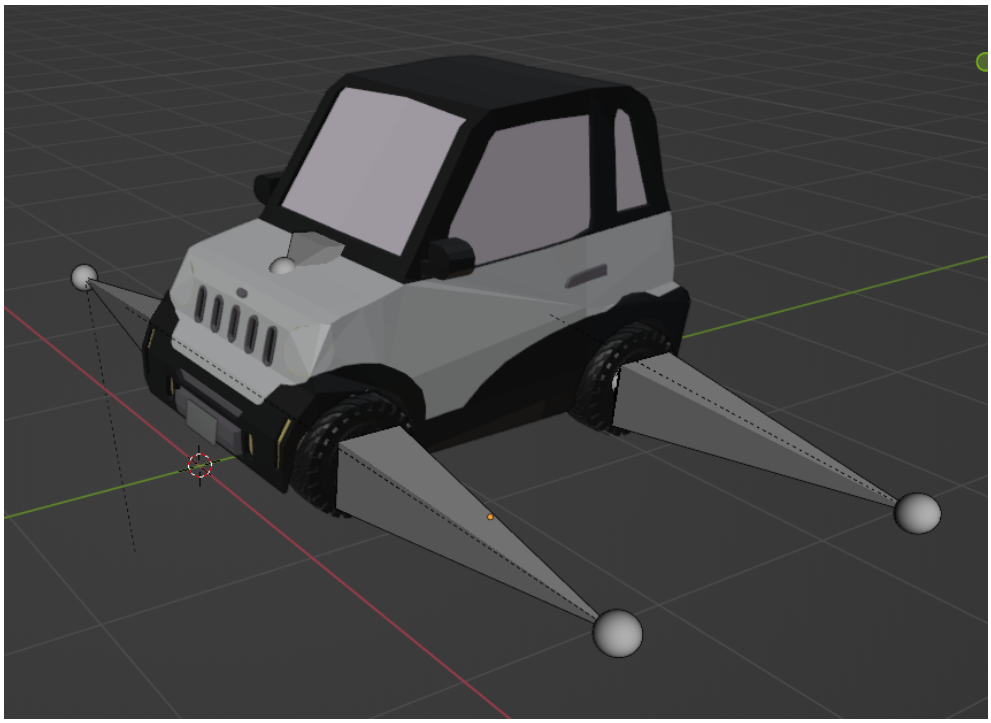


Figure 3.14: The Base Skeleton Attached to the Vehicle

In order for the vehicle to be proper simulated into the Unreal Engine environment two more meshes are needed, the *Physical Asset* and the *Raycast Sensor mesh*. The first mesh serves the purpose of allowing the Unreal Engine to calculate the vehicle's physics. For those calculations there is no need for high detail so the model used is a simpler version of the final mesh that covers the whole vehicle, except for the wheels that have their psychics adjusted separately at a later stage. The Physical Asset mesh should not extend beyond the boundaries of the original model and should have the same position as the original model. Then it should be exported

as a separate .fbx file with the name "SMC_<vehicle_name>.fbx", so it can be recognized by Unreal Engine.

The second mesh, is needed to set up the shape that will be detected by the raycast sensors (RADAR, LiDAR, and Semantic LiDAR). That mesh is more detailed than the Physical Asset mesh, in order to increase the realism of sensor simulation but less detailed than the final model. For this purpose an earlier model of the vehicle was used that lacks a lot of the details of the final model. It should be noted that in order for this mesh to be functional that it should cover all aspects of the vehicle, including the wheels and the side mirrors, where the wheels should be cylinders of no more than 16 loops and it should extend beyond the boundaries of the original model and should have the same position as the original model. Finally it should also be exported as an .fbx file with the name "SM_sc_<vehicle_name>.fbx". The two meshes are presented in Figures 3.15 and 3.16 respectively.

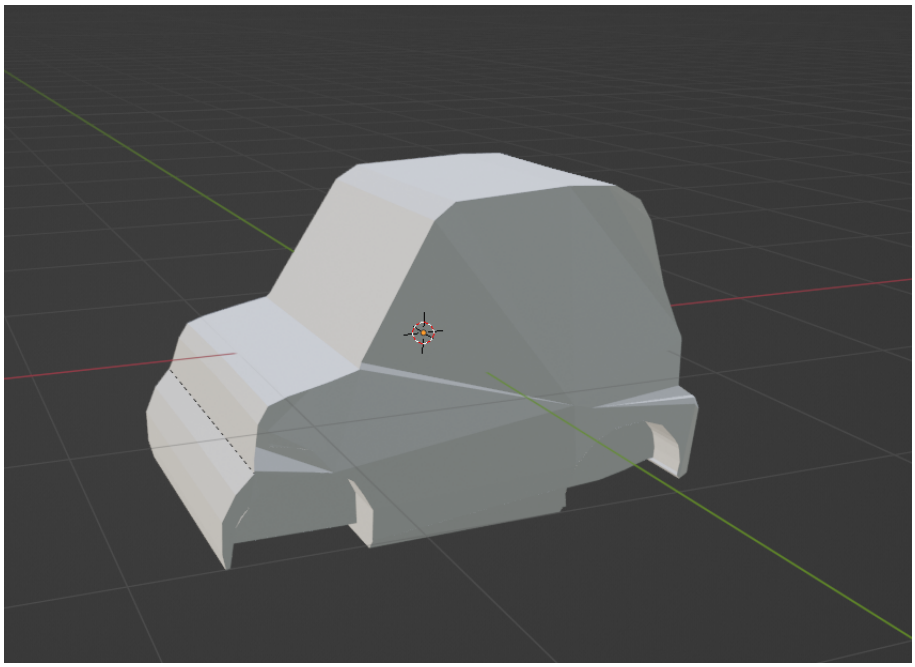


Figure 3.15: The Physical Asset Mesh

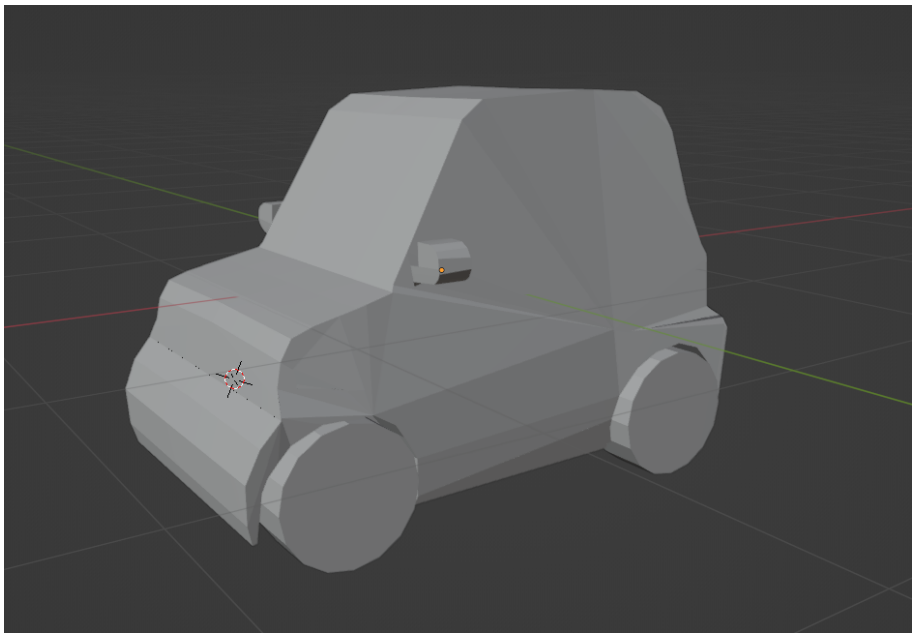


Figure 3.16: The Raycast Sensor Mesh

The next steps concern the importation and configuration of the vehicle into the simulator. First of all a folder named after the vehicles name <vehicle_name> should be created in the path *"Content/Carla/Static/Vehicles/4Wheeled"*.

Inside the newly created folder, the main mesh .fbx file should be imported by right-clicking in the Content Browser and selecting:
Import into the path *"Game/Carla/Static/Vehicles/4Wheeled/<vehicle_name>"*.
A dialogue option will pop up where the Import Content Type should be set to Geometry and Skinning Weights, the Import Method should be set to Import Normals. After this is done the Skeletal Mesh should appear along with two new files, <vehicle_name>_PhysicsAssets and <vehicle_name>_Skeleton. The rest of the .fbx files shoud be uploaded separately from the main vehicle skeleton .fbx file.

To set the physical asset mesh the <vehicle_name>_PhysicsAssets file that was created in the previous step must be opened. There, the Vehicle_Base mesh in the Skeleton Tree panel must be right-clicked and on the window that pops up, the Copy Collision from StaticMesh option is selected. There, the SMC_<vehicle_name> file is selected, which is the physical assets mesh of the vehicle and causes the outline of the physical asset mesh to appear in the viewport. The default capsule shape from the Vehicle_Base must be deleted as the new outline is much more detailed.

Afterwards the four wheels are selected simultaneously and from the Go to the Tools panel the shape is changed from Primitive Type to Sphere. The Physics Type is also changed to Kinematic from the Details panel where the Set Linear Damping option is set to 0, as this will eliminate any extra unwanted friction on the wheels that can hinder their physics. Finally before the Re-generate Bodies button is clicked, the Enable Simulation Generates Hit Event for all meshes is selected as this allows the simulator to understand when a collision occurs between the vehicle and any other asset in the simulation. This procedure will generate four spheres that must be adjusted to the size of the wheel, in order to give the wheels the correct physics.

An image of how the physical assets file of the vehicle should look before closing the window and saving is shown in figure 3.17 where both the physical asset mesh outline and the wheel spheres are visible.

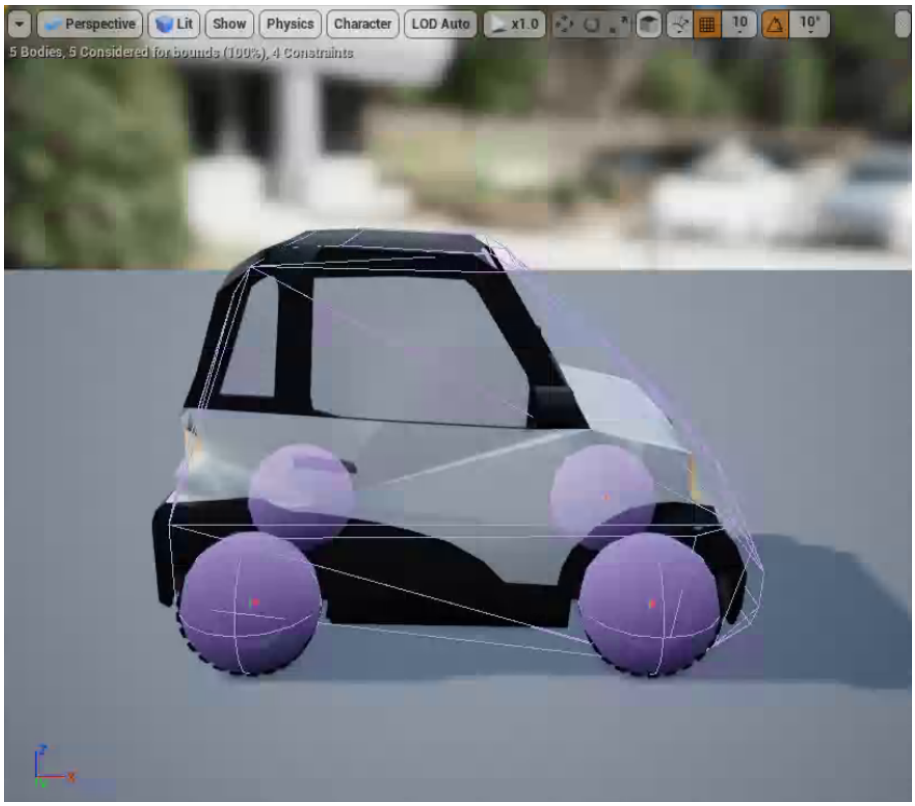


Figure 3.17: The Physics Assets File

The creation and configuration of the animation blueprint is next, which is responsible for the animation of the vehicle as the simulation runs. In the Content Browser, the vehicle folder must be right-clicked and have the Animation $->$ Animation Blueprint option that pops up selected. There in Parent Class, the VehicleAnimInstance option is then selected and finally in Target Skeleton the <vehicle_name>_Skeleton file is selected. After pressing OK the blueprint must be renamed as AnimBP_<vehicle_name>.

For the configuration part an existing one from a native CARLA vehicle is copied as it is practically the same and saves a lot of time. In Content/Carla/Static/Vehicle any CARLA vehicle folder can be chosen and its Animation Blueprint is opened. Then in the My Blueprint panel, the AnimGraph is opened which will show the graph in the viewport. The components Mesh Space Ref Pose, Wheel Handler and Component To Local are all simultaneously selected and copied into the corresponding graph area of the Ecocar's vehicle Animation Blueprint. The Component To Local component must be connected to the Output Pose component by dragging the figure on the end of the first component to the one of the second before having the blueprint compiled, saved and closed. The final Animation Blueprint is presented in Figure 3.18.

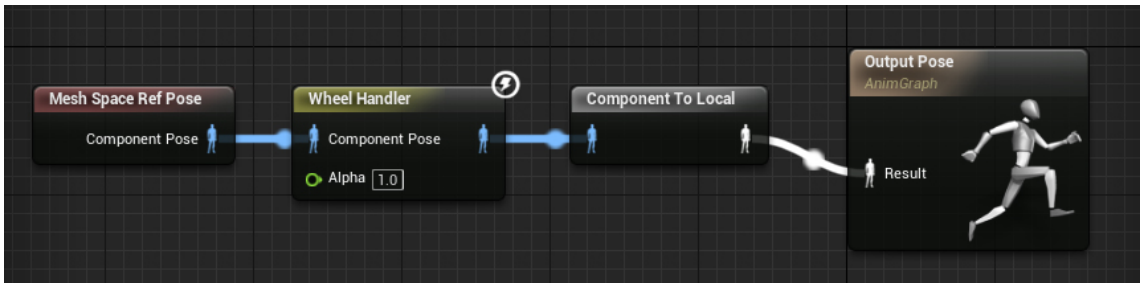


Figure 3.18: The Animation Blueprint

The next step involves the preparation of the vehicle and wheel blueprints so their configuration can follow. In the Content Browser in the path:
"Content/Carla/Blueprints/Vehicles",
a new folder must be created named <vehicle_name>.

Inside that new folder the All Classes section is selected from the pop-up menu Blueprint Class that opens with a right click. There the BaseVehiclePawn is selected and the file is renamed as BP_<vehicle_name>. For the wheels, from the folder of any of the native CARLA vehicles in the path Carla/Blueprints/Vehicles the the four wheel blueprints are copied and pasted into the blueprint folder of the Ecocar and the four individual files are renamed to replace accordingly. The new blueprint folder should be as seen in figure 3.19.

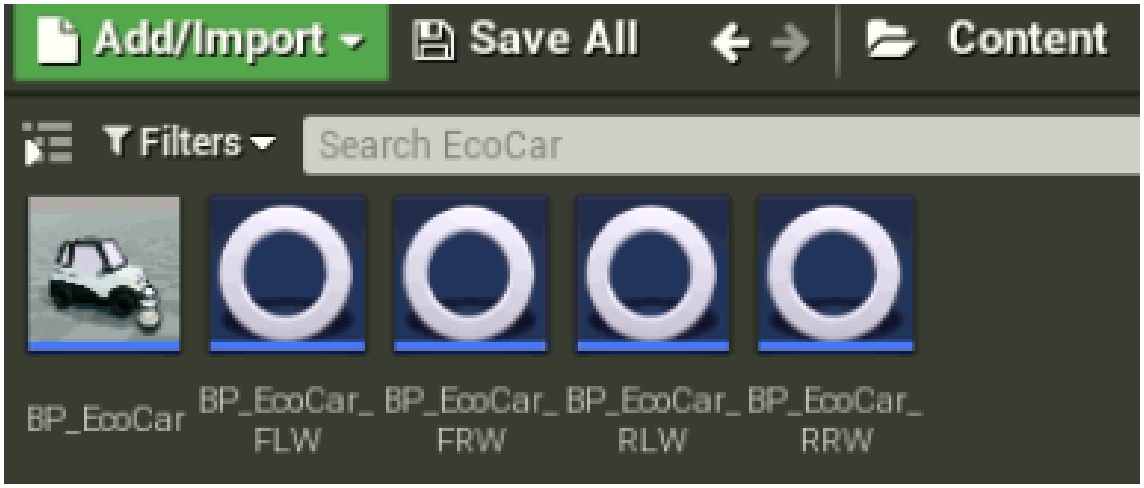


Figure 3.19: The Blueprint Folder

To configure the wheel blueprints the following steps must be followed in all four of them. After opening each of the four blueprints, in the Class Defaults panel, the Collision Mesh is set to Wheel_Shape which prevent the wheels to sink into the ground. Then values for wheel shape radius, width, mass, and damping rate are adjusted to match the specification of the Ecocar. According to the vehicle's manual, it uses wheels with 155/65/R14 specifications. Where 14 symbolizes the Rim Diameter, R the Construction type which is Radial construction, 155 the Width of the tire and 65 the Aspect Ratio which is the ratio of the sidewall height to the cross section width. Next the Tire Config must be set to CommonTireConfig and for the front wheels the Steering Angle is set to 70, also the option Affected by

Handbrake is unchecked. For the rear wheels the Steering Angle is set to 0. and they are affected by Handbrake. Finally before compiling and saving the suspension values are set using the ones of another urban vehicle with a soft suspension as a guide.

For the configuration of the vehicle's blueprint, the one with the vehicle's name that was created is opened and in the Components panel, the VehicleMesh (Inherited) is selected. Then in the Details panel, at the Skeletal Mesh the base skeleton file of the Ecocar is selected. For the Anim Class the AnimBP_<vehicle_name> file that was created in the animation blueprint precess is selected. For the collision in the Components panel, Custom Collision (Inherited) is selected and the file SM_sc_<vehicle_name> raycast sensor mesh is chosen. For the vehicle movement the VehicleMovement (MovementComp) (Inherited) is selected and for each of the wheels on the Wheel Class in the Details Panel the file named BP_<vehicle_name>_<wheel_name> that corresponds to each of the four wheels is selected. The other detail regarding the movement of the vehicle were either set to default or approximated based on the specs of the vehicle that are stated in the website or similar vehicles that are already inside the engine. There are no additional separate meshes for the Ecocar like doors and lights so no further steps to incorporate them need to be implemented.

After all of the blueprints are set the vehicle must be set on the Vehicle Library of the Unreal Engine so it can be accessed by Carla and used in simulations. For this in Content/Carla/Blueprint/Vehicle the VehicleFactory file is opened, where in the Generate Definitions tab the Vehicles option is selected. There the Default Value section is expanded to add a new element to the vehicles array. For that element to be filled the sections of Make and Model of the Ecocar are added and Class value is filled with the BP_<vehicle_name> blueprint file that was configured in the previous step. Finally after the whole process was checked for any potential errors the file was compiled and saved. The Ecocar can now be called in any script by searching for its blueprint in the blueprint library with the command "vehicle_bp = blueprint_library.filter('ecocar')[0]" and spawning it as an actor like any other vehicle. The vehicles and wheels blueprints can be accessed any time through the Unreal Editor and be modified if the user wishes.

Chapter 4

Autonomous Driving Components

For a vehicle to be able to operate autonomously it is key to perceive the environment, plan its course and react in unforeseen events, therefore the vehicle must be equipped with the related modules that provide *Perception*, *Reactive Navigation* and *Path Planning*. Perception is key since it allows the vehicle to detect possible obstacles that might exist in the road, such as pedestrians or other vehicles and also have an understanding of the road and its limits. Path planning is essential since it allows the vehicle to calculate the path that it must follow, according to current position and the desired destination.
In Chapter 4, a description of the related modules developed in the simulated environment to meet the aforementioned goals is presented. The simulated Ecocar, is equipped with a full sensor suite, based on simulated modules of the hardware which is readily-available in the lab and will be installed in the real vehicle. This approach had as a goal, to provide a simulated agent with the maximum level of detail and behavior directly comparable, as far as it concerns its operation, with the real world vehicle.

The locations at which the sensors will be placed on the vehicle are of high importance as it determines the space from which they will derive information from. In figure 4.1 the location of all of the sensors used for the agent is demonstrated for a better overall visual understanding.

Figure 4.1: The Blueprint Folder

The camera sensor is attached on the windshield of the vehicle, capturing the road from the perspective of the driver so it can properly detect the road lanes that the vehicle needs to drive between. The Lidar sensors is attached on the roof of the Ecocar so it can have a holistic view of its surroundings without it being hindered by other parts of the vehicle. The IMU needs to be located at the center of the vehicle in order to properly detect its orientation on the map. Finally the GNSS is placed on the front of the vehicle near the engine. As GNSS gives the vehicle's position it must be placed there to take into account the vehicles lenght in front of the driver to avoid crashing.

4.1 Path Planning

4.1.1 Optimal Path

In the physical world, path planning is performed by a GPS (Global Route System) navigator that retrieves information about the map and the roads, like the geographical coordinates of a place, from a satellite. Likewise in Carla the same information can be retrieved by the simulation itself, because every road in Carla

can be represented by a set of waypoints [24], which are oriented points on the map in the form of "carla.Transform". In this format the information about the coordinates map (x, y, z) and their orientation (pitch, roll, yaw) is given. This approach is used, with the prospect of when the agent is implemented on the physical vehicle this information will be replaced with the one provided by a GPS.

When a map is opened in the UnrealEditor, all the waypoints that consist the roads are visible and they appear as red squares connected with a line to indicate that they belong in the same road. Each one of them is considered as a point of the road with known location and orientation. A screenshot taken from the UnrealEditor where the waypoints of a road in the map "Town10HD_opt" are visible is presented in Figure 4.2

Figure 4.2: Visualization of the waypoints

It is worth noticing that the the exact location and orientation of every actor, including the Ecocar, can be stored and retrieved using the "carla.Transform" method.

To validate the applicability and functionality of the proposed approach a path planning module was integrated and tested, namely the "global_route_planner.py", that comes with the base installation of Carla, was used. This path planner finds the shortest path connecting the origin point and the destination point using the A* search with distance heuristic. An analysis of the algorithm can be found in Chapter (2.1). The optimal path consists of a sequence of waypoints that connect with each other in a way that obliges with the traffic rules and is stored as an array that contains those waypoints in the right order. When the code is executed it passes the location that the vehicle is spawned as the origin point and another random spawn point in the map as the destination point into the "trace_route" function. This function is part of the planning and control agent "global_route_planner.py" that was already mentioned.

In Figure 4.3 an areal view of the car can be seen where the waypoints of the optimal path have been highlighted with red. The final agent will essentially have to be trained to follow those red point to reach the final destination.

Figure 4.3: Visualization of the waypoints

Because one of the goals is to make the agent as realistic as possible, the coordinates of the points in the optimal path are transformed from the relative form Carla provides, to absolute geographical ones that a GPS would provide (latitude, longitude, altitude).

Waypoint coordinates of the optimal path do not change after the initial calculation, so the geographical coordinates are calculated by using the "transform_to_geolocation" function that Carla provides and store the new coordinates in an array like previously.

4.2 Obstacle Avoidance

4.2.1 Vehicle Coordinates

For the agent to perform the task of obstacle avoidance, information about the Cars position must be available to it, for the whole duration of the simulation. The vehicles location is dynamic and in order to calculate the geographical coordinates in real time, avoiding using information that UnrealEngine provides, we use a sensor.

Specifically the "get_location()" function is avoided, that returns an actors location to the user without using any sensor, making the method unrealistic.

The sensor that is used is a GNNS (Global Navigation Satellite System) sensor and is attached to the Ecocar. Carla Simulator provides a ready to use GNSS sensor, that calculates the geographical coordinates of the actor that is attached to, by adding the metric position to an initial geographical reference location defined within the OpenDRIVE map definition. This sensor was used with the default parameters. For the geographical coordinates of the vehicle and the optimal path, only latitude and longitude are of use, because altitude represents changes in the z-axis, which do not occur in the two-dimensional movement on the vehicle. For that reason altitude is not calculated as it would only slow down the agent.

4.2.2 Vehicle Orientation and Speed

Equally important to the vehicles position at any given moment is its orientation and speed. To calculate the orientation an IMU (Inertial measurement Unit) sensor is attached to the vehicle. Similarly to the GNSS (Global Navigation Satellite System) it is provided by Carla Simulator and used with its default parameters. The IMU provides measures that a compass would retrieve for the parent object. The data is collected from the object's current state and the orientation is given in radians.

The information regarding speed, may be acquired by the speedometer and in the real vehicle it can be acquired by the onboard device. Carla can retrieve a vehicles velocity with the "vehicle.get_velocity()" command, which returns the vehicles velocity in the three different axis of the map. In order to calculate the speed of the Ecocar, the velocity in the x and y axis is retrieved and because those velocities are vertical the cumulative velocity, or speed, is calculated using Eq. 4.1.

$$cumulative\_velocity = \sqrt{(x\_velocity)^2 + (y\_velocity)^2} \tag{4.1}$$

4.2.3 Lidar Sensor

In order to train an agent to avoid obstacles, it is essential to efficiently detect them in the environment. This task can be performed in many ways and with the help of different sensors, such as radars, cameras and lidars. For the detection of other vehicles, pedestrians and other stationary objects the agents gets all the required information from a lidar sensor.

A lidar is an acronym of "light detection and ranging" or "laser imaging, detection, and ranging" [25]. It is a method for determining ranges by targeting an object or a surface with a laser and measuring the time for the reflected light to return to the receiver. It is sometimes called 3-D laser scanning, a special combination of 3-D scanning and laser scanning.

A lidar determines the distance of an object or a surface using Eq. 4.2.

$$d = \frac{c \cdot t}{2} \tag{4.2}$$

where c is the speed of light, d is the distance between the detector and the object or surface being detected, and t is the time spent for the laser light to travel to the object or surface being detected, then travel back to the detector (Figure 4.5).

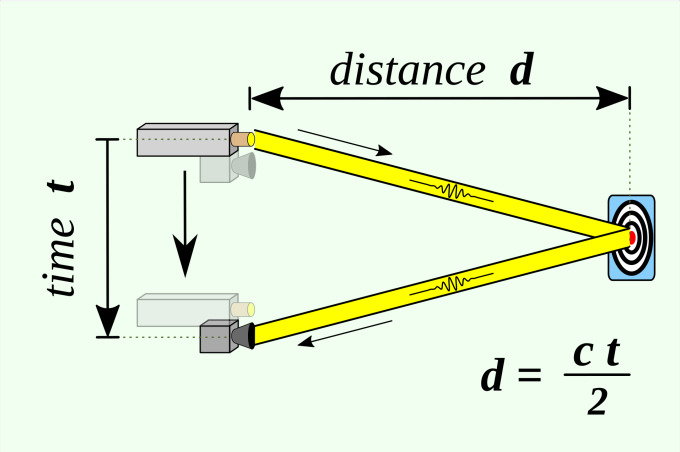


Figure 4.4: Visualization of lidars distance calculation [25]

The specific model that was modeled, is an OS1 Mid-range digital lidar sensor made by OUSTER [26], that is available in the lab, to be integrated in the real vehicle. Similarly to the other sensors a lidar sensor can be attached to the Ecocar as a lidar sensor actor exists in Carla. Its parameters have been adjusted to match with the ones of the OS1. Specifically the max range is set to 200 meters, the vertical field of view to 45 degrees, the channels to 64 and the points per second to 1310720.

Figure 4.5: The OS1-64 Lidar Sensor [26]

The sensor is placed at the top of the Ecorcar and for each step of the simulation it provides a point cloud with its surroundings. A point cloud is a discrete set of data points in space, where Each point position has its set of Cartesian coordinates (X, Y, Z). The point cloud is visualized and further processed with the use of the python library Open3d. The aforementioned library gives many possibilities to the user for easy management, transformation and visualization of the point cloud.

The initial unmodified point cloud that is returned from the sensor can be seen in Figures 4.6 and 4.7. As it can be observed, even though the lidar can give an image of the environment that surrounds the vehicle, it cannot recognize specific objects. Also a lot of noise and cloud points that do belong to obstacles that the Ecocar

cannot collide with, because they are outside the road, exist in the initial point cloud. For the point cloud to be usable by the agent we need to tackle those issues by editing it with the help of the Open3d library, all of the functions used are found in the Open3D documentation page for the point clouds [27].

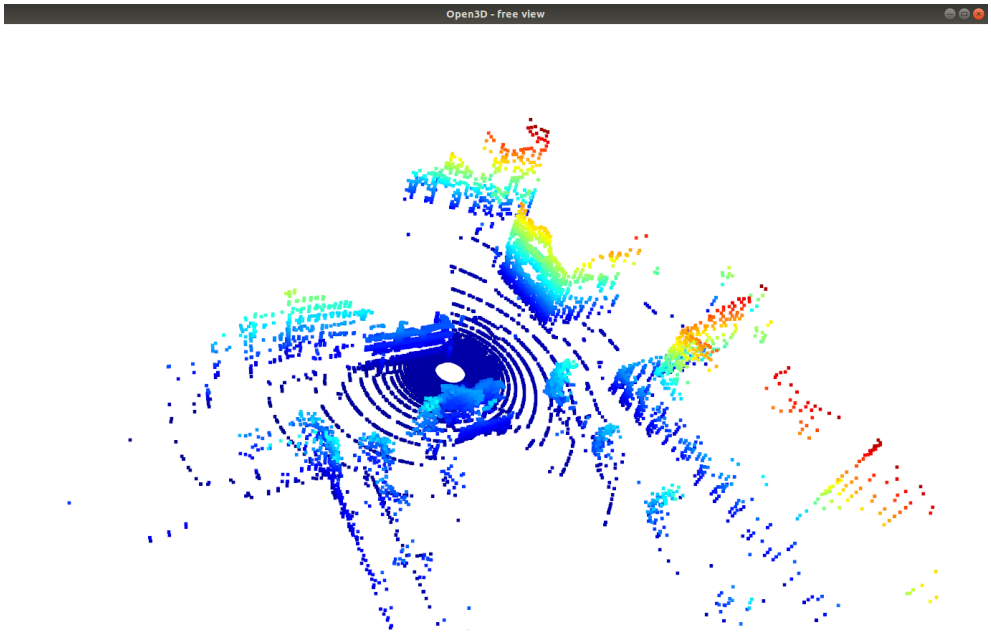


Figure 4.6: The Unmodified Point Cloud Visualized

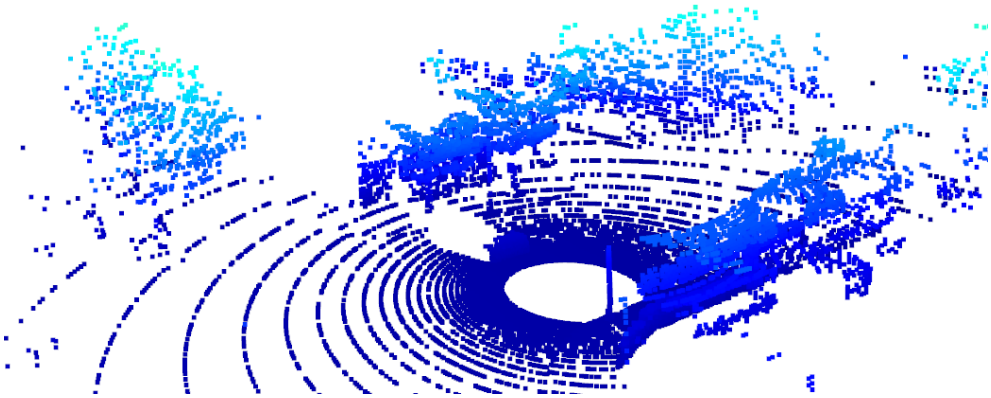


Figure 4.7: A Closer Look of the Unmodified Point Cloud

The Lidar provides points, that are inside a circular area of two hundred meters radius around the lidar. This space was limited so that all points inside it that are not useful for the agent and are essentially noise, impeding the speed of the calculations are removed. Open3D gives the user the option to crop a point cloud by fitting a shape on it whose parameters are given by a json file and cropping anything outside it. The "read_selection_polygon_volume" and "crop_point_cloud" Open3D functions call and use this volume to crop every other point cloud produced by lidar sensor. The volume that was used is a rectangular parallelepiped which extends approximately to 50 metres in front of the vehicle, 10 metres behind, 11

metres to the right and left and one and a half metres above the ground. At the same time, the road itself has been cut off, as we do not want our algorithm to recognize the points that make up the road. The exact parameters of the volume are presented in Figure 4.8.


```
{
        "axis_max" : 10.86202239990234,
        "axis_min" : -50.29427337646484,
        "bounding_polygon" :
        [
                [ 0.0, -11.7735581687269484, -1.404421944415926 ],
                [ 0.0, 11.902886208398076, -1.404421944415926 ],
                [ 0.0, 11.902886208398076, 1.0087438502552768 ],
                [ 0.0, -11.7735581687269484, 1.0087438502552768 ]
        ],
        "class_name" : "SelectionPolygonVolume",
        "orthogonal_axis" : "X",
        "version_major" : 1,
        "version_minor" : 0
}
```


Figure 4.8: Crop Volume Parameters

In Figures 4.9, 4.10 and 4.11, the cropped point cloud is demonstrated, with the road still attached, which is represented by the circles on the ground.

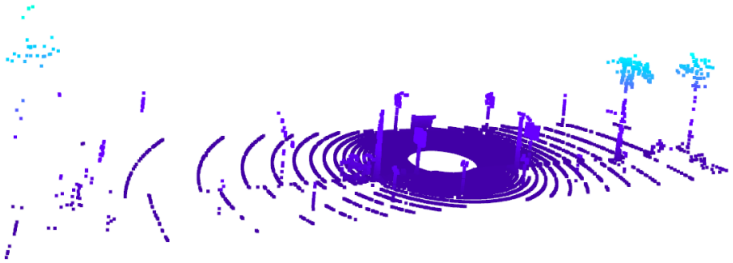


Figure 4.9: Cropped Lidar Volume with Road

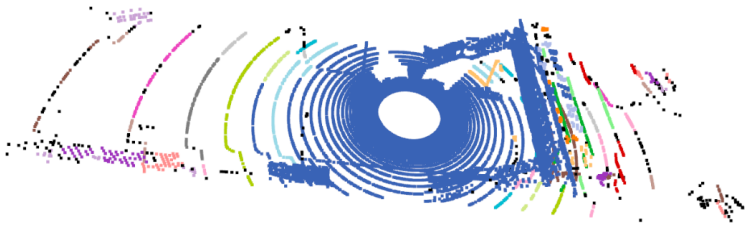


Figure 4.10: Cropped Lidar Volume with Road

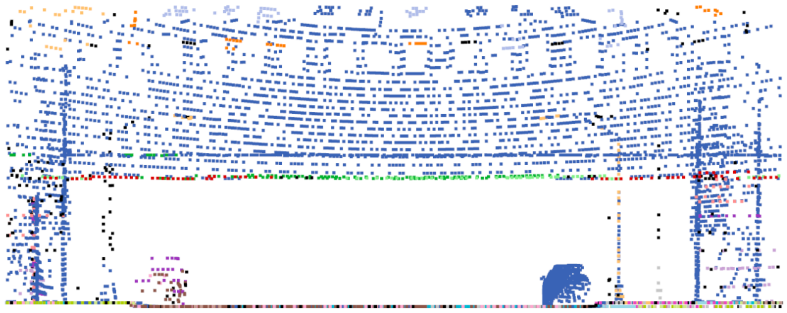


Figure 4.11: Cropped Lidar Volume with Road from Side

After cropping the point cloud, the DBSCAN algorithm was applied to the remaining points for clustering, as it works perfectly for this purpose because it creates a different cluster from the points of each obstacle within the lidars reach. An analysis of the algorithm can be found in Chapter 2.2. DBSCAN can be applied to a point cloud automatically with the cluster_dbscan Oped3D function. This function requires from the user to give an eps radius and the mimPts, for this agent eps was given the value of 2 and the minimum number of points the value of 50.

An aerial view of the point cloud where DBSCAN is applied is presented in Figure 4.12. Each different color represents a different obstacle on the road. The black points are considered noise. It should also be mentioned that this is the uncropped point cloud so it detects things like the road that are not of actual use to the agent, it is presented like that to demonstrate how the DBscan algorithm works.

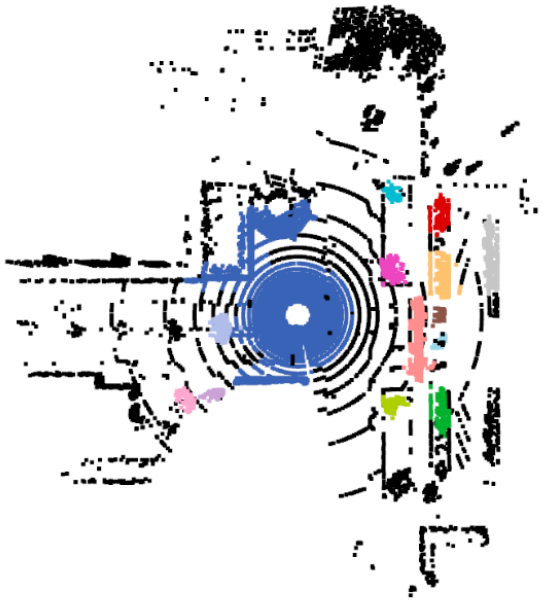


Figure 4.12: DBSCAN Applied on Point Cloud

DBSCAN can be applied on the lidar's point cloud on each step of the simulation, thus providing the agent with knowledge about the obstacles around it in real time.

The output of the algorithm is presented in figure 4.13.

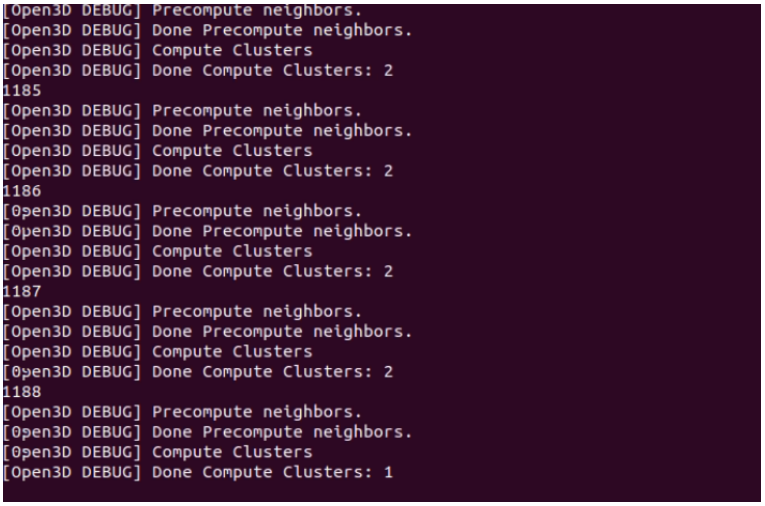


Figure 4.13: DBSCAN Output

Once the clustering process has been carried out, again with the help of Open3d a bounding box in fitted on each Cluster which essentially bounds the points of each different obstacle within a box for easier understanding of its size. In each step of the simulation an array is created where each of its elements is a list of the points contained in one of the clusters created by DBSCAN. The get_axis_aligned_bounding_box Open3D function takes each list of points and creates a rectangular parallelepiped based on the outer points of the cluster. In Figure 4.14, DBscan has calculated two clusters. The max and min points of each of the two bounding boxes created for the clusters are presented, based on which further calculation about the relative distance and orientation of each cluster/obstacle and the Ecocar can be made.

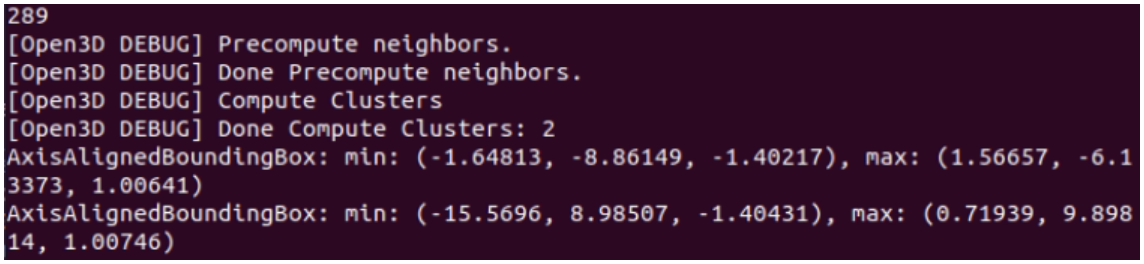


Figure 4.14: Bounding Box Output

Additionally, in Figure 4.15 the camera view alongside the Lidar view after applying the aforementioned algorithms are depicted. The car is the axes in the center of the Carla's Lidar window and the squares are the boxes in which the various Clusters have been placed.

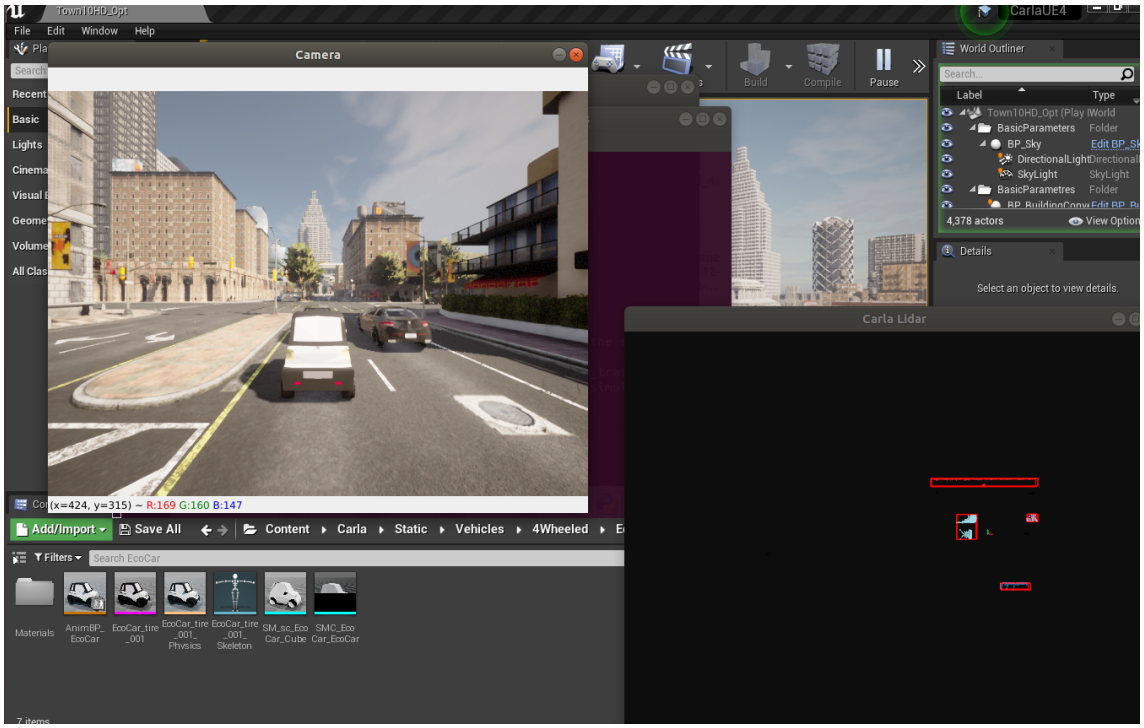


Figure 4.15: Lidar and Camera Output

Looking at the Carla Lidar window closely figure 4.16, in front of the car two cars can be seen, which are identified as an obstacle. To the right of the Ecocar there is another car and a wall, but also another car in the opposite lane of the road. As it can be observed the agent is able to recognise and put into bounding boxes in real time, four different obstacles that the Ecocar car can collide with.

Figure 4.16: Lidar Output Closeup

4.2.4 Lane Detection

Except the obstacles that the Ecocar can collide with, it is important to be able to recognize the lane boundaries. This will allow to implement strategies that will prevent the vehicle from going off-road or on the opposite lane in order to avoid an obstacle. One of the most efficient ways to detect in real time the boundaries of a lane, is through a camera by applying different algorithms to isolate and detect each lane.

Carla provides a ready to use RGB camera as a sensor, that acts as a regular camera capturing images from the scene. The camera was used with its default parameters and the data output given in each sensor tick is an array of BGRA 32-bit pixels. This array of pixels, with the help of the opencv library can be visualized, in order to have a visual image output and also further processed to detect the lanes of the road.

A camera has already been used in the simulation in previous steps. It was attached to the vehicle in order to move along with it to provide a constant video stream of what is happening around the Ecocar as the obstacle detection pipeline was running. An example of the output a camera gives is presented in Figure 4.15.

The first step was to set the resolution for the image output, 640×480 was selected, because the output data are efficient for lane detection and a bigger resolution would slow the calculations by a lot. Then the pixel raw data was reshaped, in order to be visualized by the "imshow" opencv function. The cameras position was changed and was placed on the windscreen of the car, like how cameras are usually placed in autonomous vehicles, in order to realistically capture the front road ahead of the Ecocar, as it is demonstrated in Figure 4.17

Figure 4.17: Camera Location

From the figure 4.17 it is evident that the camera captures much more than just the road. This hinders the agents ability to detect the lanes of the road because object that do not belong in the road are also taken into consideration that can also look like road lines. To avoid this, with the help of opencv a mask is applied on the road, that crops the image leaving only the part of the road that is relevant to the agent, while filling the rest with black color. The front of the car that is visible to the camera is also cropped. The shape of the mask is given as an array, which in this case is a trapezoid with edges, (0, 430), (250, 250), (390, 250), (640, 430).

The edges location are related to the resolution of the image, which as mentioned before is 640×480 pixels and the result after the mask is applied is presented in Figure 4.18

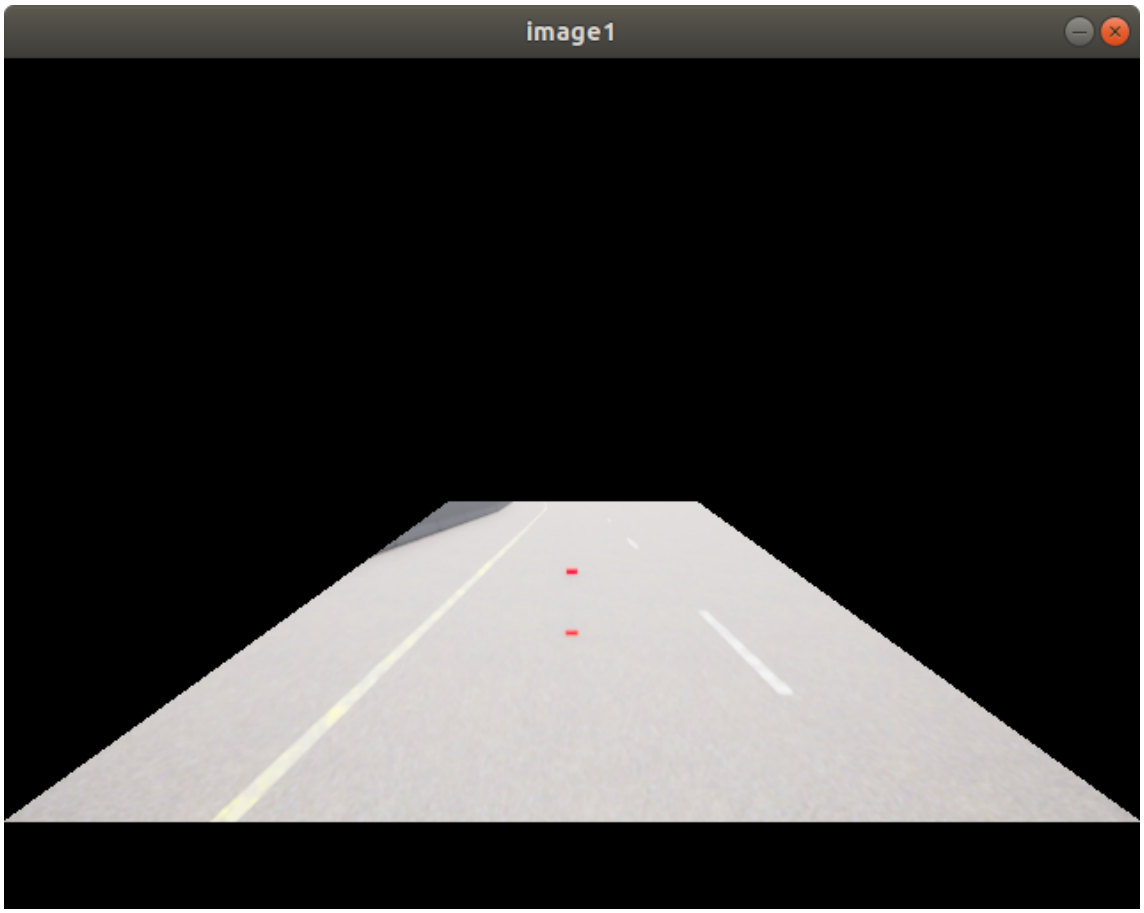


Figure 4.18: The Masked Camera Image

To actually separate the lanes of the road from the rest of the road, the fact that the lanes are either a bright white of bright yellow color that comes in contrast with the grey of the asphalt will be utilized. Opencv has the option of masking an image with a color range. With the right range all of the road can be masked off leaving only the lines from the whole image. While the default color space in OpenCV is BGR, this filter is made using the HSV color space. An explanation of the two color spaces can be found in section (2.3). This is done because in OpenCV the HSV color space is ideal for extracting the range of light colors that are of interest for the lane detection by changing only the Value parameter which scales from black at 0 to white at 255. So by giving the whole range of possible values for the hue and saturation but only

keeping the higher values from the vertical axis (Value parameter) gives exactly the light colors of the lanes. In BGR the three values should increase simultaneously to get the grey scale so creating a range of grey colors is not possible.

To demonstrate this a filter was created where the user can set the minimum and maximum hue, saturation and value range in the HSV color space and see the effect it will have on the image Figure 4.19. In OpenCV for HSV, hue range is [0,179], saturation range is [0,255], and value range is [0,255].

Figure 4.19: The Mask Color Filter

As the minimum in the Value range increases Figure 4.19, the darker grey colors start to fade Figure 4.20. Setting the Value range at minimum:220 and maximum:255 (max possible value) and taking the whole range for Hue [0,179] and Saturation [0,255] always according to opencv documentation gives the result presented in Figure 4.21

Figure 4.20: Darker Colors Start to Face

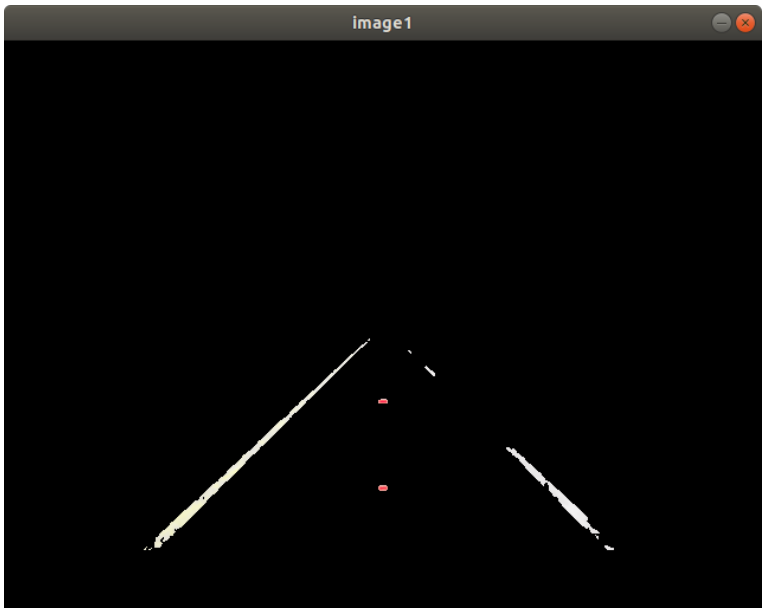


Figure 4.21: Darker Colors Start to Face

It should be noted, that in order for the mask to consistently keep the lanes while cropping the rest of the road, the shading was disabled in the unreal engine map as the shadows were intervening by darkening the color of the lines in different segments of the road, making them pass the color range that was selected, eventually making them undetectable. This is something that should be addressed in the case of a real implementation. The shadings can be easily disabled in the Unreal Engine Editor by selecting in the World Outliner that is in the top right corner of the editor the Directional Light that corresponds to the Sky Blueprint and unselect the Cast Shadows Option.

Afterwards the image is passed through a Grey Scale Filter and is blurred with the Gaussian Blurring method. Explanations for those methods are found in sections 2.4 and 2.5 respectively. Here, it should be noted that even though the previous filter was created in the HSV color space, the original image remains in the BGR format, so its BGR to Gray Scale. This method combination is helping smooth out the lines in the image and make them more distinguishable for the next algorithm. The result of grey scaling and blurring depected in Figure 4.22 on the unmasked image.

Figure 4.22: Grey Scaled and Blurred Image

Canny Edge Detection is a popular edge detection algorithm that OpenCV provides a ready to use function to implement it on any image. An analysis of the algorithm and the steps it follows to detect edges in an image can be found in Section 2.6. The final result of strong edges in the image that the algorithm provides is demonstrated in Figure 4.23 where the algorithm is applied on the cropped and masked image.

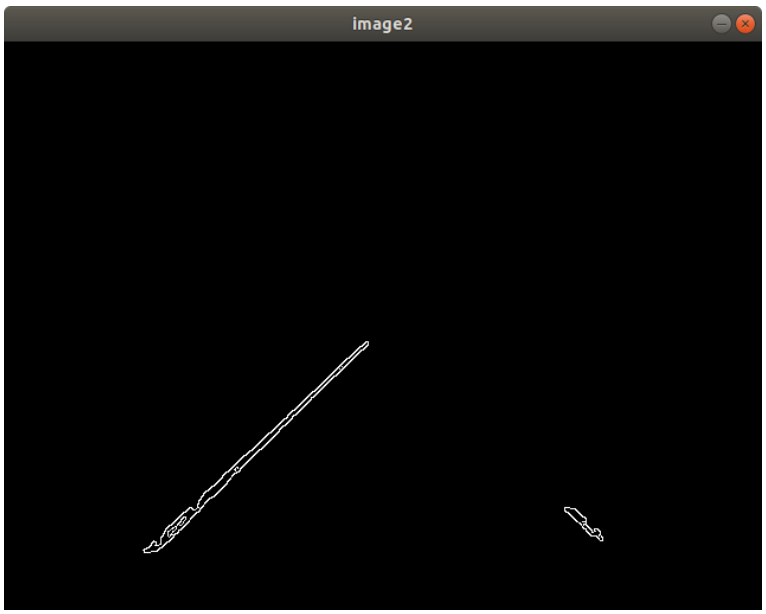


Figure 4.23: Canny Edge Final Result

The edges of the road lines are detected by the Canny Edge algorithm and the output given is the location of pixels that constitute those edges. This output is the input needed for the final algorithm that is used in the lane detection procedure. The final algorithm is the Hough Line Transform, which is a transform used to detect

straight lines. The explanation of the algorithm and the steps it follows to extract straight lines from an image can be found in Section 2.7.

For the development of an autonomous agent, the probabilistic version of the algorithm is chosen, which works with a reduced sampling size that makes the voting procedure shorter, thus the algorithm runs faster at the expense of some accuracy. The accuracy lost does not change the output of the algorithm as the input given is not complex. The threshold that is given is 20 points and the minimum line length is 100 pixels and the maximum line gap is 20 pixels, always compared to the resolution of the image. Those numbers were selected by trial and error for the agent to recognize the continuous lines that mark the end of the road but do not detect the non-continuous ones that would allow a vehicle to change lane legally.

With this method a problem arises as the algorithm can detect lines that are not road lines, like crosswalks and as presented in Figure 4.23, if those points are given as input to the Hough Line algorithm it will calculate two different lines for the left traffic lane. To avoid all those erroneous results the agent checks for the slope of each line detected by the Hough Line algorithm.

The probabilistic version provides the extremes of the detected lines (x_0, y_0, x_1, y_1), so the slope for each one is calculated with the formula:

$$Slope = \frac{y_2 - y_1}{x_2 - x_1} \tag{4.3}$$

If the slope is between -1.7 and -0.2, then it is most probably a traffic line that is detected at the left of the Ecocar and if the slope is between 0.2 and 1.7 then it is most probably a traffic line that is detected at the right of the Ecocar. The rest of the lines found are discarded as they are most probably noise. After the lines left and right of the Ecocar are detected the agent calculates a mean line line for each category and returns its slope and extremes.

In Figure 4.24 the final result of the line detection is demonstrated where the mean of the right and left lines are drawn in real time on the camera image. In Figure 4.25 in addition to the line the output of the algorithm that provides the slope and extremes of the line is provided.

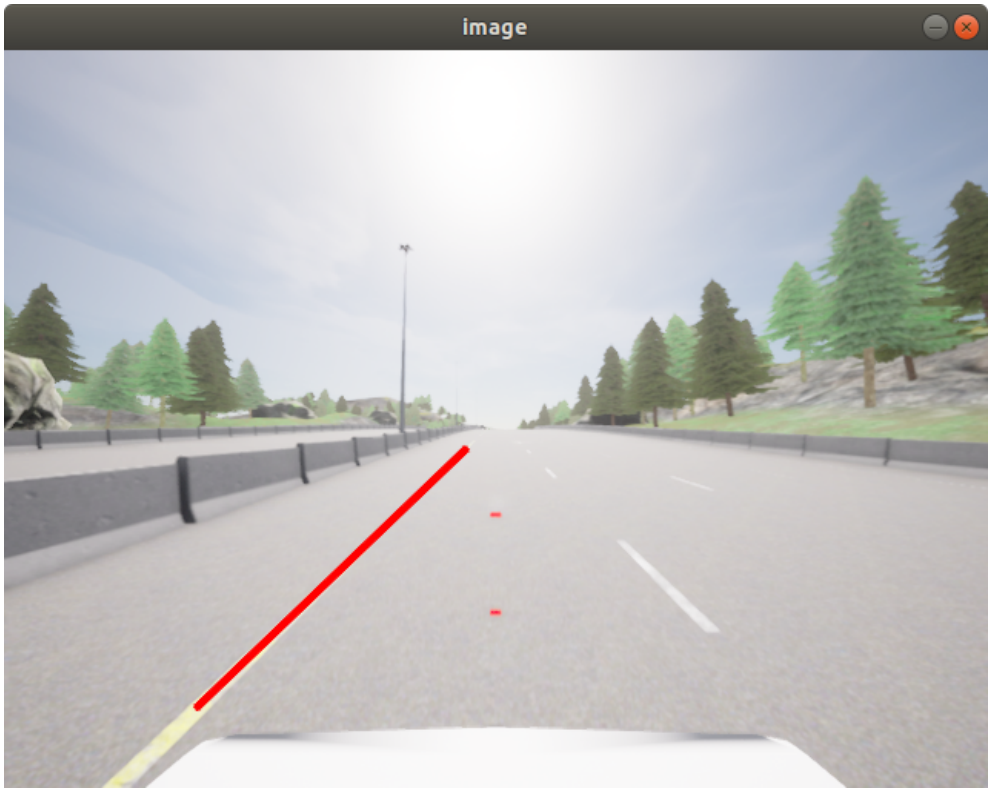


Figure 4.24: Lane Detection Result

Figure 4.25: Lane Detection with Algorithm Output

Chapter 5

The Autonomous Agent

We have already presented the process of developing a realistic simulation of the Ecocar vehicle, by adding a detailed model into the Carla simulating environment and importing a set of potential sensors that can be used so that it can interact with the environment. In order for the vehicle to navigate in an autonomous manner, we developed an autonomous agent, using the the Deep Deterministic Policy Gradient (DDPG) [28] algorithm and implementing it with the usage of Stable Baselines 3 [29] library. For making the training process easier a Gym environment [30] was created. The autonomous agent developed, demonstrated the ability to navigate autonomously in an obstacle free environment, following the optimal path as it is calculated by the A* algorithm and it was compared with Carla's built in agent.

5.1 Agent's Overview

The agent consists of three main parts. The first is about the components inside Carla simulator which includes the model of the vehicle, the sensors gathering the input, the world of the simulation and the path planner. The second part is the python API and includes the algorithms that are applied and the calculations that are made on the data collected by the sensors at each steps of the simulation. While the third uses the outputs of the second part to train the agent using the DDPG algorithm, which when trained properly will return the optimal action for the vehicle to execute. The three parts are connected with each other and work simultaneously for the agent to perform the autonomous driving tasks in real time.

Figure 5.1, depicts the flow of information and the interdependence of the aforementioned components. The blue background highlights the components inside the Carla simulator, the red the python API including the algorithms that are applied and the related calculations and finally the green components are related to the training procedure.

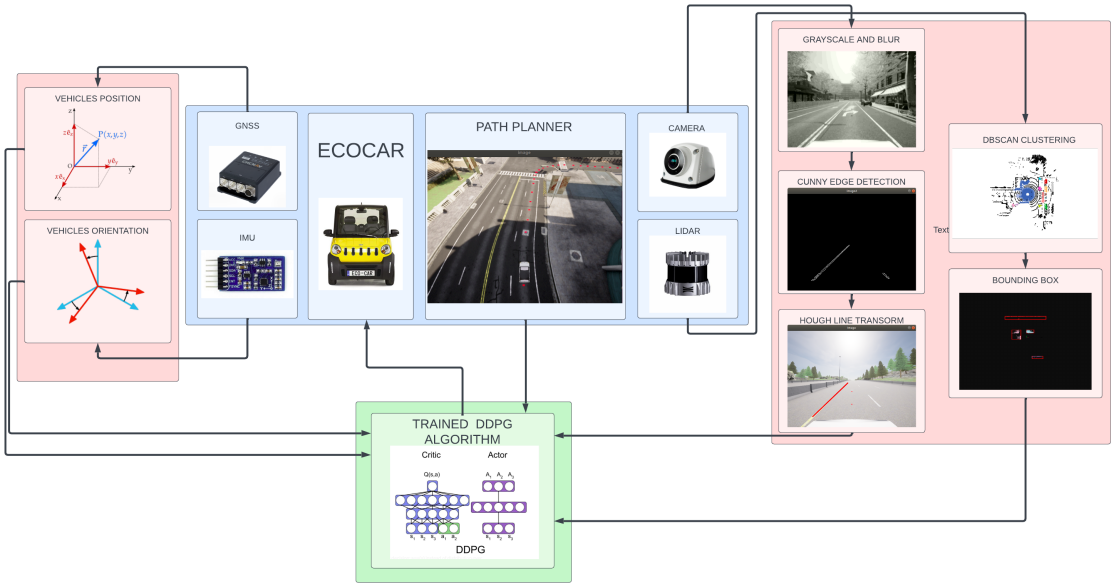


Figure 5.1: Flowchart of the Agent

The arrows from the blue to the red part symbolize the raw data the python API receives from the simulation, the arrows from the red to the green part the input for the DDPG algorithm and finally the arrow from the green part back to the blue the optimal action of the vehicle for the given moment of the simulation.

5.2 The DDPG Algorithm

When selecting the ideal algorithm for training an agent, a really important issue is to consider the exact problem at hand. This mainly depends on the type of action that the agent takes. For example many agents have been developed that play classic arcade video games like “Super Mario Bros” for the Nintendo Entertainment System (NES) [31]. The controller of the NES (Figure 5.2) is constituted of an arrow button that move the character and two buttons for jump and attack. This configuration allows the agent to take only discrete actions that would match the inputs of the controller.

In problems that regard robots, autonomous vehicles included, where the movement needs to be fluid the agent needs to take continues actions that vary from a specific range. For example the agent that is developed and will be described here, can take two actions, one for steering and one for throttling the vehicle. Both of those actions can take values from -1 to 1, where in steering -1 means to turn the steering wheel fully right and 1 turn it fully left and for throttling -1 mean to press the break fully and 1 to press the throttle fully, while 0 is to take no action in both cases. For solving those types of problems the Deep Deterministic Policy Gradient (DDPG) algorithm was developed and is considered a very solid option when it comes to applying reinforcement learning on robotics.

Figure 5.2: An NES Controller [32]

5.2.1 Structure of the Algorithm

DDPG is based on the Actor-Critic method, this essentially means that two neural networks are used. The first one, the actor, proposes an action given a state in the simulation, based on the inputs, while the critic is trained to predict if the action proposed by the actor is good or bad based on the current state of the simulation. The algorithm is also model free, which means that it doesn't have explicit knowledge of the model that the environment follows, so it can predict the next moves based on its actions but rather explores the world and slowly learns how it functions. A good example of a problem where model based algorithms can be used, is chess as the agent knows all the possible next moves of the opponent based on its action and can plan further ahead. In addition DDPG is an off-policy algorithm which allows it to use old data. This helps the agent to explore all the possible paths to find the optimal solution and not focus on maximizing just one like the on-policy models do.

DDPG has its critic network learning the Q-function while its actor network simultaneously learns the policy. This is achieved by using the off-policy data and the Bellman equation to learn the Q-function and uses the Q-function to learn the policy. The Q-function is a function that given a state and an action estimates how good or bad is this action, depending on the state of the simulation, while the policy determines which action the agent should take.

The implementation of the algorithm that is used by the Stable Baselines3 Library is the one found in [3], using the approach described on [33] which establishes the theory for deterministic policy gradients algorithm DPG and [28] that adapts this theory for the deep RL setting thus creating the DDPG algorithm. This implementation is used in this thesis for the creation and training of the RL model as it is going to be described in detail.

The Bellman equation describes the optimal action-value function for the agent, $Q^*(s,a)$ as depicted in Eq. 5.1.

$$Q^*(s,a) = \underset{s' \sim P}{\mathrm{E}} \left[r(s,a) + \gamma \max_{a'} Q^*(s',a') \right] \tag{5.1}$$

$s' \sim P$ indicates that the next state, s' and is sampled by the environment from a distribution $P(\cdot|s,a)$. Also r(s,a) is the reward function of the agent.

For DDPG algorithm the equation 5.1 is the starting point from which the $Q^*(s, a)$ is starting to be approximated. That equation will be changed by adding replay buffers and target networks in order to be able to approximate the Q values more accurately for the environment.

For the next steps a neural network, $Q_\phi(s, a)$ (the target network), with parameters ϕ is used and a set $\mathcal{D}$ of transitions (s, a, r, s', d), (the replay buffer) has been collected where d is an indicator to see if the s' is the terminal one. Using those parameters a mean-squared Bellman error (MSBE) function is set (Eq. 5.2), which provides the information regarding how closely Q_ϕ comes to satisfying the Bellman equation.

It is worth noticing that the target networks and the replay buffer are techniques employed by all Q-learning algorithms for function approximators in order to minimize the MSBE loss function (Eq. 5.2) efficiently.

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathrm{E}} \left[\left(Q_\phi(s, a) - \left(r + \gamma(1 - d) \max_{a'} Q_\phi(s', a') \right) \right)^2 \right] \tag{5.2}$$

The $(1 - d)$ part of the function indicates whether the s' is a terminal state for the agent thus preventing the agent from getting additional rewards after the current state. The set of $\mathcal{D}$ previous experiences that functions as the replay buffer must contain a wide range of different experiences to provide stability to the agent. Usage of the only recent data will cause them to overfit to that and the training sequence will break, while using too much experience data can cause the training process to slow down significantly.

Because the target the Q-function aims to take the same values, as depends on the same parameters that are being training, is causing instability. To solve this a second network is implemented, called the target network that uses parameters which comes close to ϕ, but with a time delay that is described by updated the target network once per main network update by polyak averaging as shown in Eq. 5.3.

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi \tag{5.3}$$

ρ is a hyperparameter called polyak that can take values from 0 to 1 included.

A second target network for the policy of the algorithm is also implemented in order to solve the challenge that comes with computing maximum over actions in the target in continuous action spaces. By utilizing this network DDPG computes an action which approximately maximizes $Q_{\phi_{\text{targ}}}$. This target policy is also computed by polyak averaging the policy parameters over the course of training and is symbolized with $\mu_{\theta_{\text{targ}}}$.

Combining all of the above parts, the Q-learning part of the DDPG alogirthm is presented in Eq. 5.4.

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathrm{E}} \left[\left(Q_\phi(s,a) - \left(r + \gamma(1-d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s')) \right) \right)^2 \right], \quad (5.4)$$

For the part of the policy learning, a deterministic policy $\mu_\theta(s)$ is learned, that will give the action that will maximize the $Q_\phi(s,a)$ function of the critic, thus making him approve the action and let the agent implement it on the simulation. Because the action space of our problem is continuous, and the Q-function is assumed to be differentiable with respect to the action, gradient ascent is performed to solve Eq. 5.5.

$$\max_\theta \underset{s\sim\mathcal{D}}{\mathrm{E}} \left[Q_\phi(s, \mu_\theta(s)) \right]. \quad (5.5)$$

A visualisation of the architecture of the DDPG is provided in figure 5.3. It is visualized how the different networks interact within the actor and the critic and then how the actor and the critic interact with each other, the experience pool (replay buffer) and the environment of the simulation.

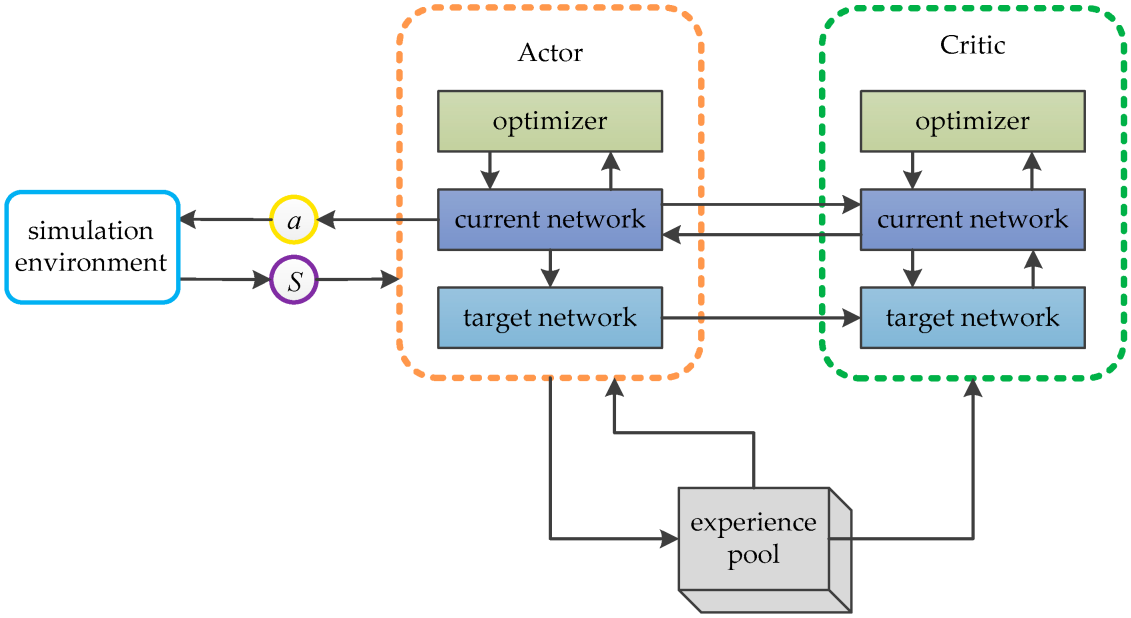


Figure 5.3: A Visual Depiction of DDPG [34]

5.3 Gym Environment

Gym is an open source python library that makes the interaction of reinforcement learning agents with different environments a simple process. Based on the paper about the fundamentals of the Gym library [30] and the Gym documentation page about custom environments [35], a Gym environment is a python class that contains functions that the agent can call to gather all the information it needs for the process of training and solving the initial problem. In the context of this thesis, the Gym

environment includes the whole Carla Simulation, the world, the vehicle, the sensors and the calculations they make. Those calculations constitute the observation, that along with the reward function, they are used by the agent as input so it will be able to train and then provide the actions needed by the vehicle to perform the given task. This process is continuous as the agent constantly takes information, the observation, from the environment and returns the action back to it, so it can be applied on the simulation. The overall procedure is decomposed in two steps: (i) in the training step, the agent uses the observation, sensor data and reward function, to train and after it starts returning actions that can be considered acceptable for solving our problem it passes to the next step, (ii) where the agent does not train any further and only uses the observation as a way to calculate the optimal action the vehicle has to take to achieve the goal.

Gym provides a plethora of ready to use environments, such as Atari games and classic control problems, such as the cart pole problem, where the user can easily apply different reinforcement learning algorithms with the use of other python libraries such as keras or pytorch to train agent. This library also allows the user to create their own environments which allows them to make them into a package that can be handled much easier than applying the different algorithms on the main script and also allows other users to use the environment as it is in the form of a python class.

In the context of this thesis, such an environment has been created for the Ecocar and it was named "CarlaEnv". The process that needs to be followed to transform the script used for our Carla client to communicate with the Carla Server, spawn the vehicle and the sensors and extract information from them into a Gym environment is as follows.

1. Initially the class of the environment must be created, that will contain all the functions that the agent will use to train and then guide the vehicle. The first function is the initialisation function __init__ where the client is connect to the server. The sizes of the action and observation spaces are set and the parameters of the different actors that will be used in the simulation are adjusted.

2. At the second part, the function reset is created, that acts like the initial state of the simulation and it is called by the agent once at the beginning of every episode. There the vehicle and the sensors are spawned and the optimal path that the vehicle needs to follow to reach the destination is computed. There the first observation is also created and returned to the agent.

3. Finally the function of the step is created. This function in addition to the observation also contains the action variable and the reward function. It serves as the step of the simulation and every time it is called it applies the action suggested by the agent based on the input of the previous step to the simulator, while it provides the agent with information about the current step, via the observation variable and the reward function.

5.4 Stable Baselines 3

As it is stated by Raffin, A. *et al.* [29] and on the documentation of Stable Baselines3 (SB3) [36], SB3 is a set of reliable implementations of reinforcement learning algorithms in PyTorch. This library is chosen for the implementation of DDPG ,as it was designed to harmonically interact with Gym environments to train and then apply the trained agents on them. SB3 contain DDPG as a function that can be called to be applied on a Gym environment so there is no need for the algorithm to be coded from scratch.

Three different types of scripts where created using this library.

- The first one is a method to check if a Gym environment is working properly. For this the only thing that is needed, is for the environment to be imported and then the function "check_env()" is used. This function runs the environment quickly without it interacting with an agent, for the user to observe if it is functioning properly.
- The second is the script used to train the agent, where after the environment is imported a folder is created to store the models of the agent, which can be set to be done manually after a set amount of steps. Also another folder is created that will contain the training logs which are graphs created in real time with the use of Tensorboard. Finally before the training loop starts the maximum timesteps are set and algorithm faction is called, this is particularly useful because the user can change RL algorithms without the need to change the environment.
- The final script concerns the loading and the usage of the already trained agent in the environment. The environment is once again imported and the exact model of the trained agent as it was saved in the folder created by the previous script is selected. The number of episodes can also be set by the user to test the agent multiple times.

5.5 Implementation

As it was mentioned in section 5.3 a Gym environment was created that contained all the script needed for the interaction of the Carla Server with the Client, which includes the creation of the vehicle, the camera sensor and the calculation off the optimal path. In addition to this the observation variable was created that contains the distance of the vehicle from the next point of the optimal path in meters, its angule error from it and the speed of the vehicle for every tick of the simulation. Finally the reward function was created that estimates how good or bad the current situation that the vehicle is.

5.5.1 Observation

The observation variable contains three different measurements. The ideal type of observation space for the observations of this problem according to Gym documentation is a possibly unbounded box in $\mathbb{R}^n$, which in this case are three. Basically it

represents the Cartesian product of n closed intervals. Each of them can have the form of $[a, b], (-\infty, b], [a, \infty) or (-\infty, \infty)$.

The first variable is the distance of the vehicle from the next point of the optimal path. Since the coordinates are in geolocation where the distance is calculated in degrees the range from where this variable can take values is [0, 0.0002]. It should be mentioned that 1 degree change in either the longitude and the latitude coordinate can be approximated as a distance of 111km or 111000 meters. So the maximum value of the distance is $0.0002 \cdot 111000 = 22.2$m, which is acceptable since if the vehicles distance from the next point reaches that number at any point of the simulation it means that the agent has failed.

The points of the optimal path are inside a python list which means that the vehicle can take as target the one that is on the first spot and when it reaches it move on to the next one slowly traversing through the whole list reaching the final point which is also the end goal.

For each point its longitude and latitude value are already calculated and the same values are know or the vehicle for each moment through the GNSS sensor. That allows to calculate the difference in latitude and longitude between the vehicle and the point at each moment by simply subtracting one value from another. Those two differences that give the relative distance in the x and y axis are then used in Eq. 5.6 to calculate the square distance of the vehicle to the point.

$$Square\_Dist = \sqrt{(latitude\_difference)^2 + (longitude\_difference)^2} \tag{5.6}$$

Finally because it is not possible for the square distance to have a value of exactly 0 while the simulation is running, nor would this small value allow the agent to adjust the trajectory of vehicle in time to reach the next point the threshold for setting the next point as the target of the vehicle is 0.00005 degrees or 5.55 meters.

The second variable is the angle error between the vehicle and the target point. This is important because with this value the agent understands at what angle the target point is compared to the front the vehicle. This along with the distance allows the agent to understand the exact relative position of the target point to the vehicle. When the angle error takes a value of zero it means that if the vehicle moves into a straight line it will reach the target point. This variable can take values from -180 to 180 degrees, thus covering a full circle.

In Figure 5.4 an illustration is provided where the desired angle is shown.

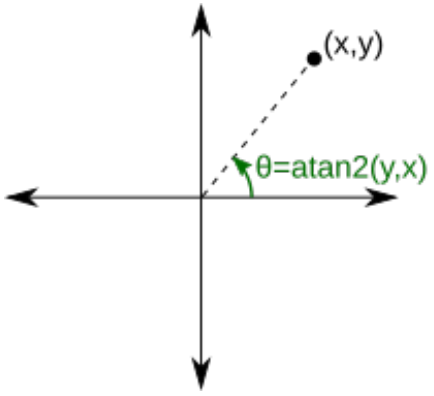


Figure 5.4: The θ Angle [37]

Considering that the vehicle lays on the center of the axis and is also aligned with the y axis, while the target point is at the (x,y) coordinate the angle θ is the angle error. In case the vehicle has a different orientation, the axis in the carla simulator are set and do not change depending on the orientation of the main vehicle. This θ angle needs to have the vehicles compass value subtracted from it. The compass value is essentially the vehicles orientation on the x,y plain and is calculated for each tick of the simulation with help of the IMU sensor. It is given in radians and not degrees for a better visualisation they conversion is made where $degrees = radians \cdot 180/\pi$. Because x and y are the relative distances of the target point to the Ecocar the already calculated longitude and latitude distances are used. The atan2 function is the 2-argument arctangent that given the distance in the x and y plain returns the angle θ. The atan2 function is part of the math python library and is used just by calling it in the script. The output of the function gives the angle in radians too so their are also transformed into degrees.

For the next step depending on the output of atan2 function some transformations are performed. If the output is negative, which is possible depending on the placement of the target point and the vehicle on the map, the value of 360 is added in order to get a similar angle of positive value. Then the value of the vehicles orientation is subtracted to find the actual angle error that takes into consideration the vehicles orientation. Depending on the value of the difference, if its more than 180 degrees the value 360 is subtracted or if it less -180 degrees the value of 360 is added. Those transformations end up providing a positive angle error if the point is to the right of the vehicle or a negative one if the point is to the left. This allows the agent to understand the orientation of the target point and that it needs to bring the value as close to 0 in both occasions.

The final variable of the observation is the speed of the Ecocar. Ideally the agent must keep the vehicle in a comfortable speed that is not too slow nor surpass the mechanical limitations of the vehicle, which are the 80km/h. For this this variable takes values from 0 to 80. All of the vehicles have a speedometer by default so no extra sensor was used but rather the get_velocity function that Carla provides which returns the velocity of any actor in the desired direction. Utilizing this, the velocity in both the x and y direction are calculated for each step of the simulation and since the vectors of those speeds are vertical to each other the overall speed can be calculated with Eq. 5.7.

$$Overall\_Speed = \sqrt{(Speed\_in\_x\_axis)^2 + (Speed\_in\_y\_axis)^2} \tag{5.7}$$

Finally the velocity that is calculated is in meter per second and not kilometers per hour which is the common way of expressing a vehicles velocity. It is know that if a vehicle has a velocity of 1m/s that equals to 3.6km/h so the final result is multiplied by the constant of 3.6 for the final velocity.

5.5.2 Action Space

The action space contains two variables and has the same box form as the observation space. Its purpose, is to contain the actions that the agent passes to the vehicle in each step of the DDPG algorithm.

The first variable of the action space is the acceleration of the vehicle. This essentially tells the vehicle to either throttle of brake. This is achieved by allowing this variable to take values from -1 to 1, where -1 means full brake and 1 full throttle, while 0 is no action regarding the vehicles acceleration at all. This allows to combine two actions in one variable of the action space since throttling and braking can happen at the same time. The Gym environment depending if the value it receives from the agent is positive or negative sets the other value at 0 and with the vehicle control and apply control functions that the Python API provides those values are passed to the vehicle for each step of the algorithm.

The second variable is the one for steering the vehicle and like the one for the acceleration it can take values from -1 to 1. Where -1 means to steer the driving wheel fully to the left and 1 steer it fully to the right while 0 means to leave it in the normal position. The value is once again passed to the Ecocar via the Python API functions vehicle control and apply control, for every step of the algorithm.

5.5.3 Reward Function

The final step before the training process can begin, is the creation of the reward function. This function checks the values of the observation variables from the current state and returns a number that indicates how good or bad the action proposed by the agent on the previous step actually is. This can help the agent understand the training goal and what type of actions is supposed to propose according to the observation values.

This function is created according to what the user that creates the agent believes is the ideal behavior of the agent, formulated as a maximization problem for the given function. It can contain penalties for inappropriate behavior or rewards for desired behaviors. According to this, the agent learns to repeat the actions that brought higher rewards in previous steps for similar observation values.

In this proof of concept the reward function contains six different parts.

- The *r_speed* parameter regards the speed of the vehicle. Because the tests take place in a city and the speed limit in Greece inside cities is 50 km/h, if the vehicles speed exceeds that value, the *r_speed* takes the value of -2, which

works as a penalty in the reward function. This way the vehicle will learn to drive in a comfortable and safe speed.

- The *r_acc* parameter is complementary to the *r_speed* one as it ensures that the vehicle is going to accelerate and maintain that acceleration during the simulation. As it was stated the acceleration of the vehicle can take values from -1 to 1. If the value of *acc* positive the *r_acc* works as a reward and if its negative it works as a penalty as it can be seen in equation 5.11. This encourages the vehicle to maintain a high speed throughout the simulation and break only whenever it is necessary.
- The *r_steer* parameter works in a similar fashion with the *r_acc* as it ensures that the vehicle will steer only when necessary and not "zig-zag" in the lane. The function that determines the reward or penalty applied to the agent depending of how much the wheel is steered at each moment of the simulation is presented in equation 5.12.
- The *r_ang* parameter is about the angle error the vehicle has with the target point. If this error is a positive number it is transformed to a negative and if it is a negative is kept as it is. Then they are used as exponential of e as seen in equation 5.9, so they provide a reward to the agent that increases as this type of error is decreasing.
- Same applies to the *r_dist* parameter for point distance, which is multiplied by 111000 again to transform the distance from geolocation degrees to meters. The distance value is transformed to a negative number and used as the exponential of e to provide a reward as the distance of to the target point decreases.
- The final parameter *r_k* contains the reward which checks if a target point has been reached. If the car reaches a point, it is subtracted from the list of total points of the optimal path, that means that if the size of that list gets smaller from one iteration of the algorithm to the other. The function takes that difference, which will always be 1, as it is not physically possible for the vehicle to reach two target points within the same simulation step and applies it as the reward for reaching a point.

Because the different parameters have different importance to the reward function they are multiplied by different constants (weights) in the reward function to have their values adjusted.

- *r_speed* is multiplied by 1 so the maximum penalty it can provide is -2.
- *r_acc* is also multiplied by 1, providing a maximum reward or penalty equal to the value of e.
- *r_steer* is multiplied by 2 because a bigger weight was needed or the vehicle to stop "zig-zaging", so the maximum reward or penalty is equal to 2.
- *r_ang* is multiplied by 2 as the agent needed a bigger reward in order not to neglect minimizing the angular error. So the maximum reward is equal to double the value of e.
- *r_dist* is multiplied by 1, so the maximum reward is equal to the value of e.

- r_k is multiplied by 5 as this would be a good bonus reward for the agent as it reaches one of the target points, since it is its main objective.

The reward function is presented in Eq. 5.8 to 5.14.

$$r\_speed = \begin{cases} -2 & \text{if } vehicle\_speed > 10 \\ 0 & \text{if else} \end{cases} \tag{5.8}$$

$$r\_angle\_error = \begin{cases} e^{-angle\_error} & \text{if } angle\_error > 0 \\ e^{angle\_error} & \text{if } angle\_error < 0 \\ 0 & \text{if } angle\_error = 0 \end{cases} \tag{5.9}$$

$$r\_point\_distance = \begin{cases} e^{-distance\_from\_point*111000} & \text{if } distance\_from\_point > 0 \\ 0 & \text{if else} \end{cases} \tag{5.10}$$

$$r\_acceleration = \begin{cases} e^{acceleration\_value} & \text{if } acceleration > 0 \\ -e^{acceleration\_value} & \text{if } acceleration < 0 \end{cases} \tag{5.11}$$

$$r\_steering = 1 - 2 \cdot |steering\_value| \tag{5.12}$$

$$r\_reward = \begin{cases} number\_of\_points\_reached & \text{if a point is reached} \\ 0 & \text{if no points are reached} \end{cases} \tag{5.13}$$

$$r\_overall = 1 \cdot r\_speed + 2 \cdot r\_angle\_error + 1 \cdot r\_point\_distance + 1 \cdot r\_acceleration + 2 \cdot r\_steering + 5 \cdot r\_reward \tag{5.14}$$

The reward function also serves the purpose to terminate the current training episode if the reward passes a negative threshold. This helps the agent reset when it has taken too many wrong actions and try from a fresh episode rather than trying to fix the erroneous situation which could delay the training process by a considerable amount. I also helps the unstuck the the vehicle in case it has collided and does not know how to turn. For this test this threshold was set to the value of -4.

5.6 Training of the Agent

Before the agent can start training with the DDPG algorithm the architecture of the neural networks of the algorithm and its hyperparameters must be set. The architecture and hyperparameters which were used are presented in Figure 5.5. As there is no input is in the form of an image there is no need for a features extractor. The actor shares the same architecture with the target actor and the critic shares the same architecture with the target critic. The actor takes the 3 observation variables and returns the action, the critic receives the action proposed by the actor and the observation and returns a value that scores how good that particular action is for the given moment.

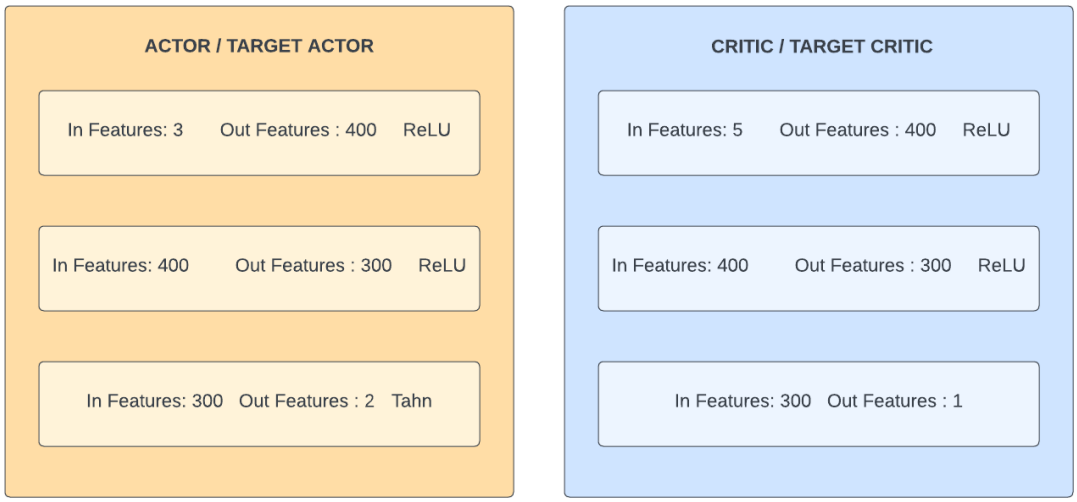
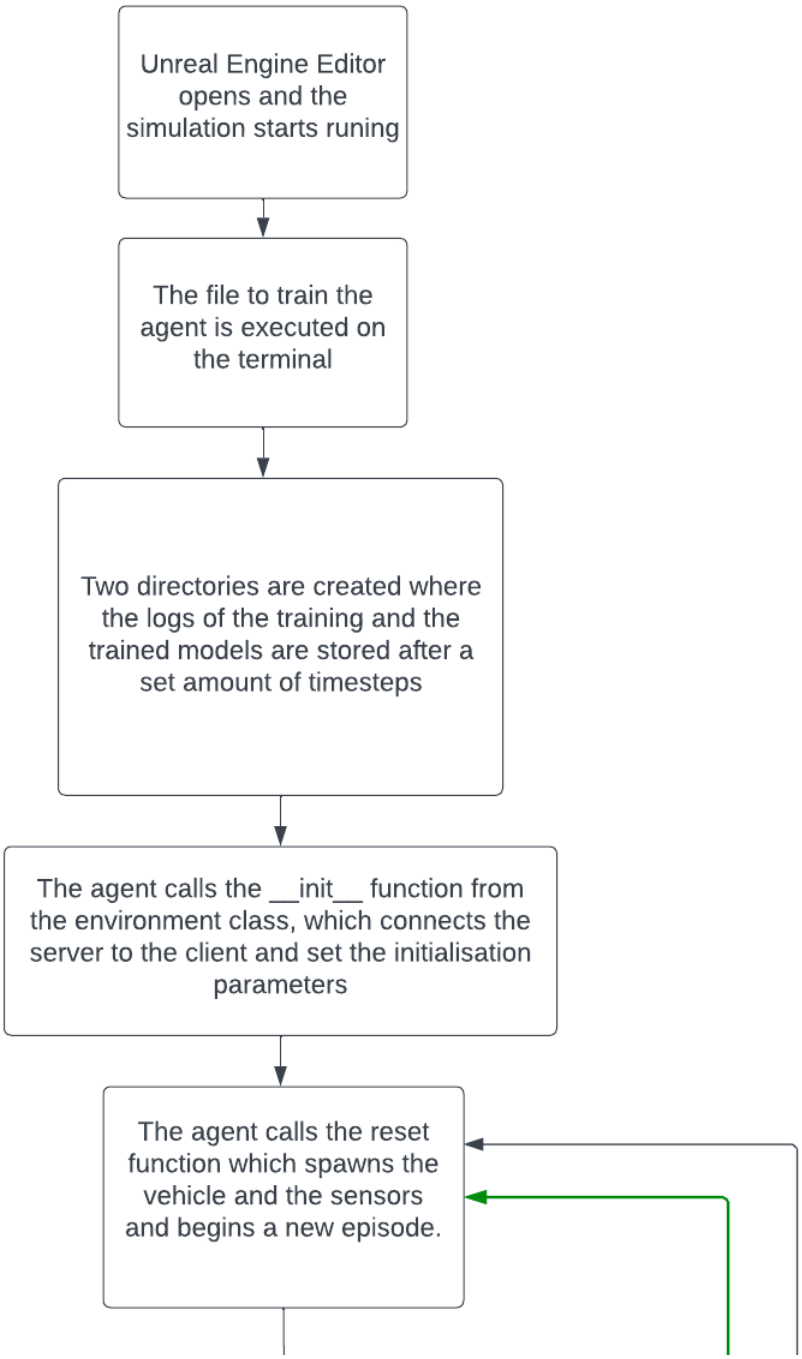


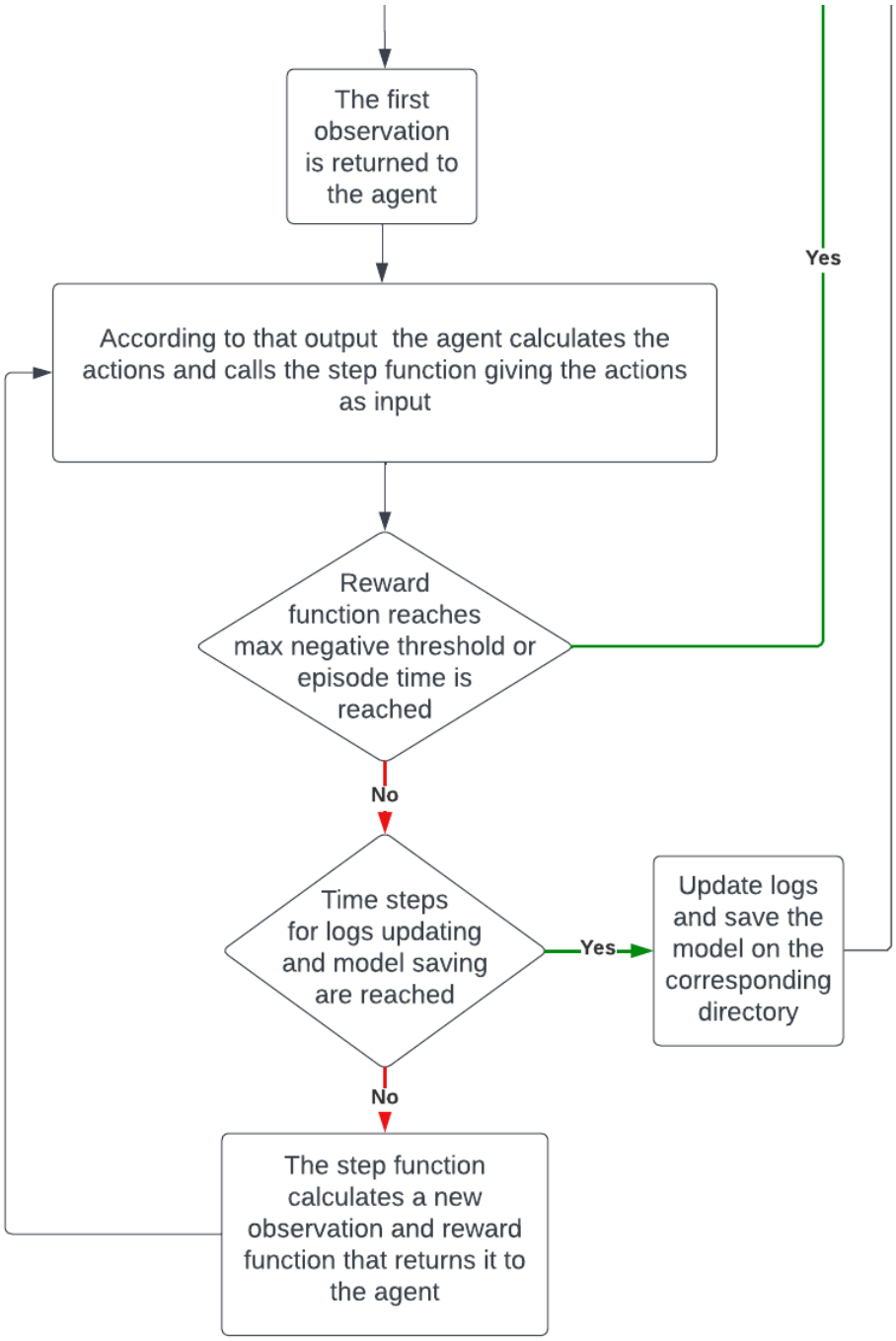
Figure 5.5: The Architecture of the neural networks of DDPG

The hyperparameters used in the algorithm are listed below:

- Policy = Multilayer Perception Policy (MLP policy)
- Optimizer = Adam, which is a method for stochastic optimization [38].
- Learning rate = 0.001, the learning rate for the adam optimizer that is used in all networks.
- Buffer size = 1000000, the size of the riplay buffer.
- Learning starts = 100, how many steps of the model to collect transitions for before learning starts.
- Batch size = 100, Minibatch size for each gradient update.
- tau = 0.005, the soft update coefficient ("Polyak update", between 0 and 1).
- gamma = 0.99, the discount factor.
- Train Frequency = (1, 'episode') , the model is updated after the end of each episode.
- Gradient Steps = -1, How many gradient steps to do after each rollout where -1 means to do as many gradient steps as steps done in the environment during the rollout.

Below the flowchart of the training process is presented.


Unreal Engine Editor opens and the simulation starts runing
The file to train the agent is executed on the terminal
Two directories are created where the logs of the training and the trained models are stored after a set amount of timesteps
The agent calls the __init__ function from the environment class, which connects the server to the client and set the initialisation parameters
The agent calls the reset function which spawns the vehicle and the sensors and begins a new episode.



It should be noted that there is no condition that stops the training process. It continues until the user is satisfied with the training results that are logged in the tensorboard after a set number of steps and stops the training process manually. In this case the steps to save the model and log the training data was set to 10000 algorithm iterations. After around 95000 simulation steps passed the training process was terminated and the tensorboard returned the statistics shown in Figures 5.6, 5.7, 5.8, 5.9 respectively.

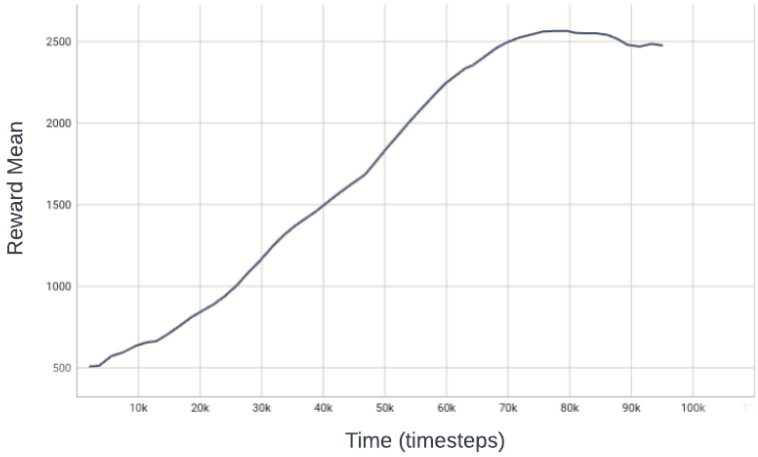


Figure 5.6: Mean Episode Reward

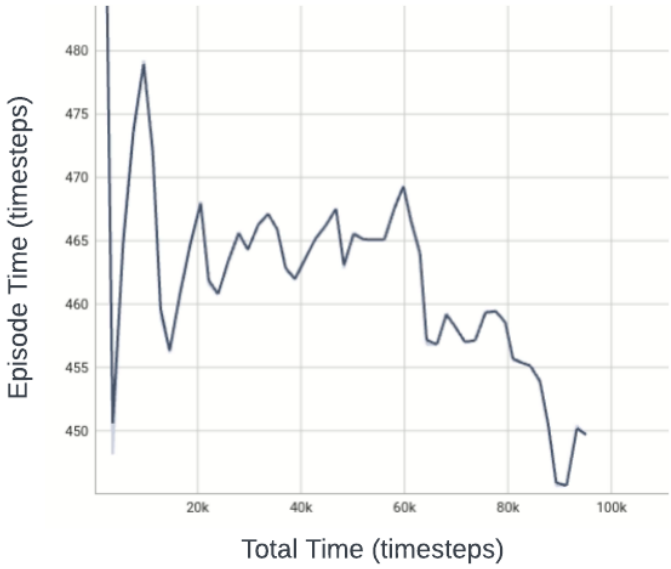


Figure 5.7: Mean Time per Episode

Where in Figure 5.6 it can be observed that mean reward increased up to the point of 80000 steps and then started to decrease. While in Figure. 5.7 the mean time per episode being below 500 means that the vehicle manages to reach its destination, thus complete the task before the max time of the episode runs out. This value decreasing as the timesteps pass also means that the vehicle reaches its destination faster.

In Figure. 5.8 the actor loss function can be seen to decrease, which indicates that the Q-Learning function is decreasing as expecting from an agent that manages to learn. On the other hand in Figure. 5.9 the loss of the critic is steadily increase which is something unexpected for an agent that learns. This can be an indication

of the actor exploring new parameters that are unknown to the policy or that it learns with a rate much higher than the actors.

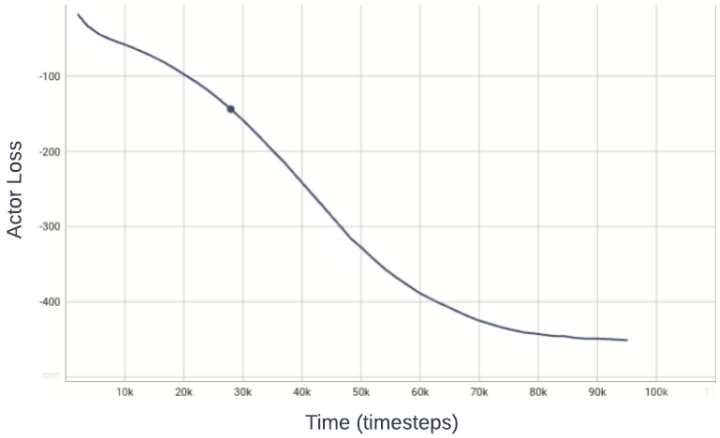


Figure 5.8: Actor Loss

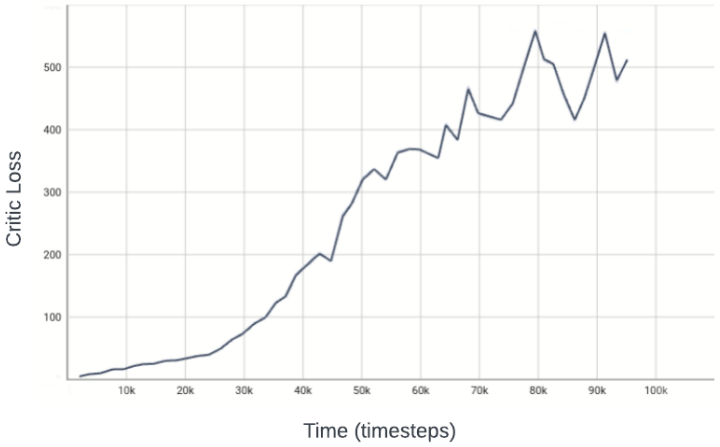


Figure 5.9: Critic Loss

5.7 Testing the Agent

After the training process is complete the different models that were saved at each 10000 time steps can be loaded and used for evaluation. A script that specifies the path of the trained models and loads it on the simulation in the same way the training script did is created, with the only difference being that that model is not being trained any further.

Three different models where tested on a specific path for two different maps available in carla simulator, namely "Town10HD_Opt" and "Town05" and the steering, speed and location of the vehicles where tracked. In Figures 5.10 and 5.11 the optimal path, as derived from the implementation of A* for specific starting and ending points, that the trained agent has to follow is presented.

Figure 5.10: Optimal Path in Map Town05

Figure 5.11: Optimal Path in Map Town10

During the training process it was observed through the camera that is attached on the vehicle that the most stable model is the one at 50000 steps. Because the reward function continued to increase as mentioned previously until 90000 steps, the model of 50000 steps was tasted against it and also against the model of 70000 as it is the mean of the two.

The differences on steering between the models for the two different maps, is depicted in Figures 5.12 to 5.15.

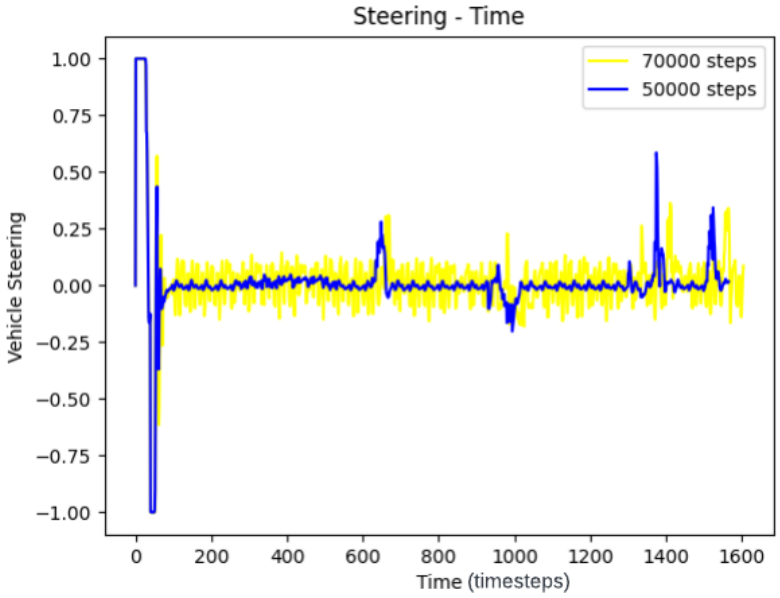


Figure 5.12: Steering *vs.* Time for the case of Map Town5

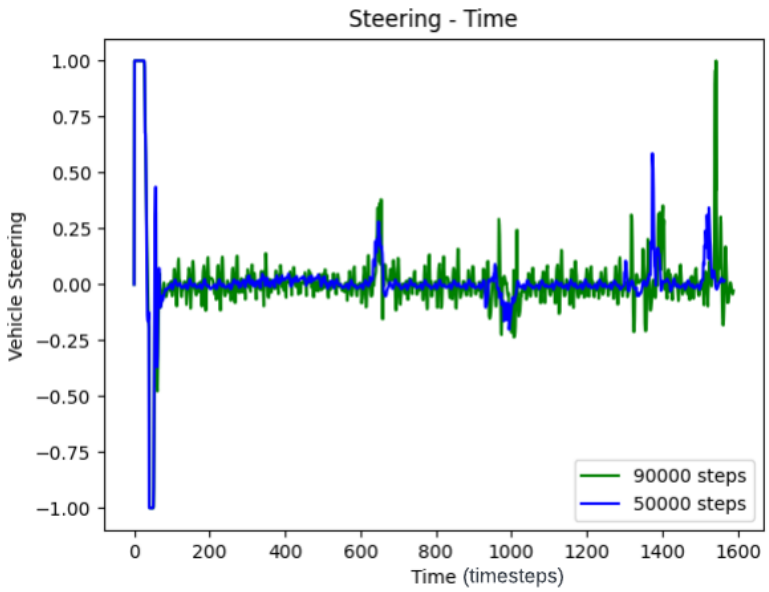


Figure 5.13: Steering *vs.* Time for the case of Map Town5

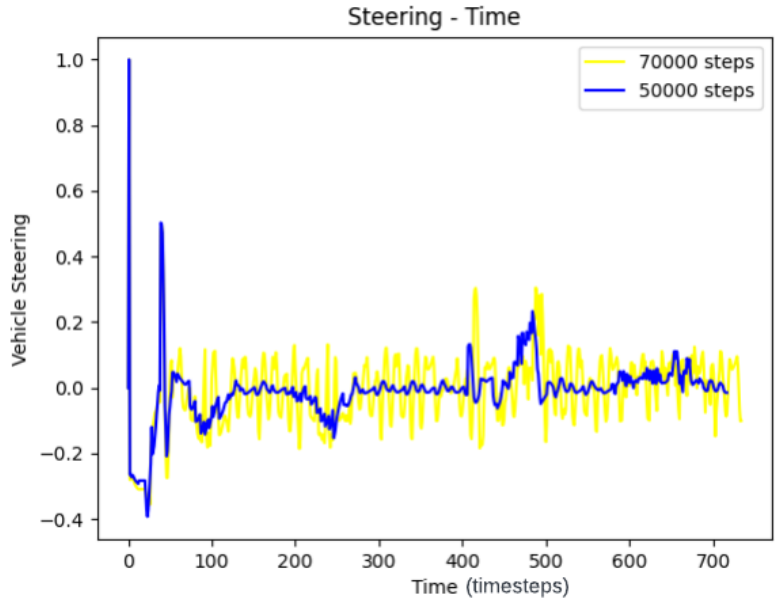


Figure 5.14: Steering *vs.* Time for the case of Map Town10

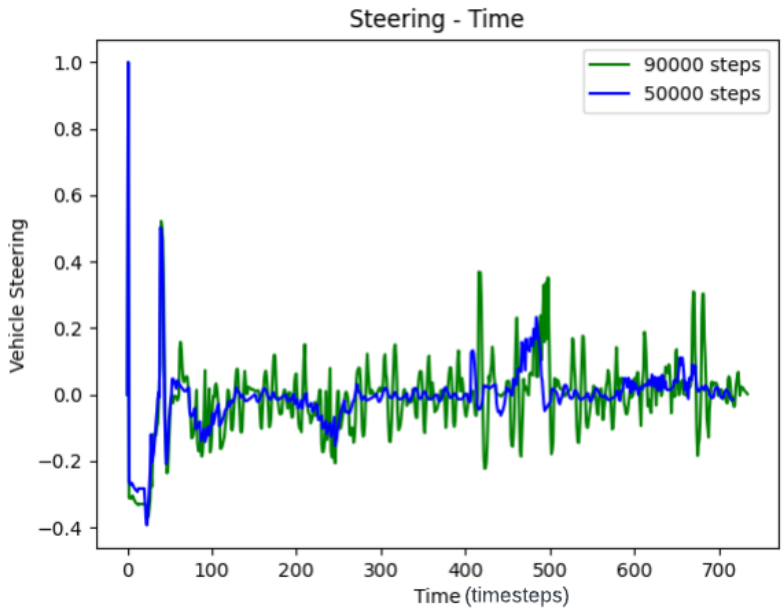


Figure 5.15: Steering *vs.* Time for the case of Map Town10

Those diagrams make it evident that the most stable model, in terms of driving comfort, is indeed the one found in 50000 time steps. This model compared to the others turns smoothly and slowly while maintaining a straight path when turning is not necessary. The fluctuations at the beginning on the diagram are caused by the agent, trying to align the vehicle with the starting point of the optimal path.

While testing the accuracy of the models in Figures 5.16 and 5.17, it is observed that they all manage to follow the optimal path as they all overlap with the optimal

path. The excess steering on the models at 70000 and 90000 do not cause the vehicle to change lanes but only to perform an uncomfortable change of orientation.

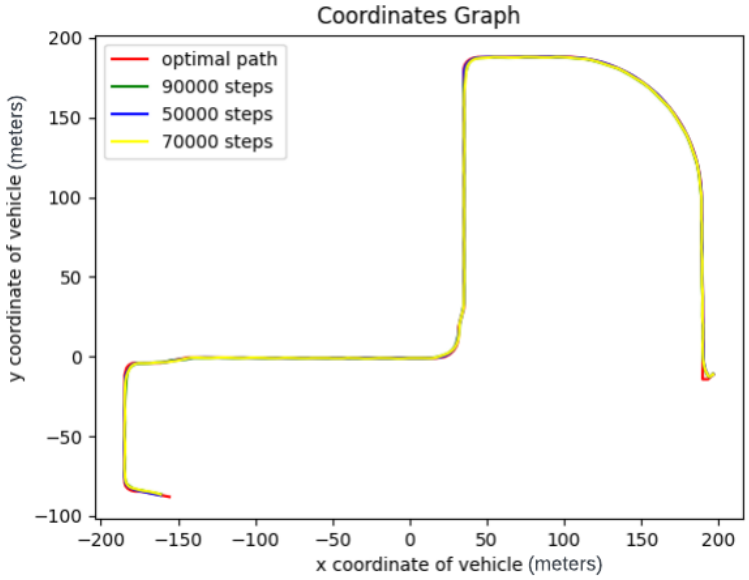


Figure 5.16: Coordinates Map Town5

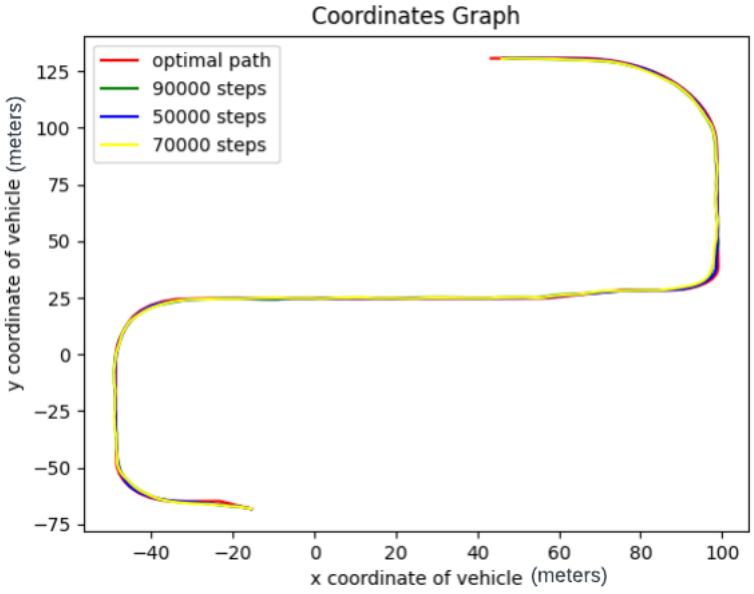


Figure 5.17: Coordinates Map Town10

Finally comparing the velocities of the three models in Figures 5.18 and 5.19, it is observed that they maintain the same speed throughout the the whole duration of the path following that fluctuates between 15 and 20 km/h.

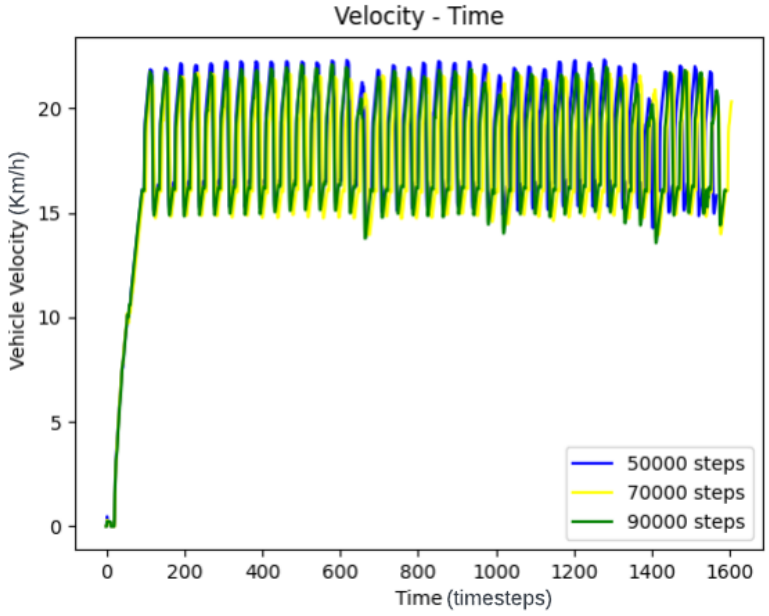


Figure 5.18: Velocity *vs.* Time for the case of Map Town5

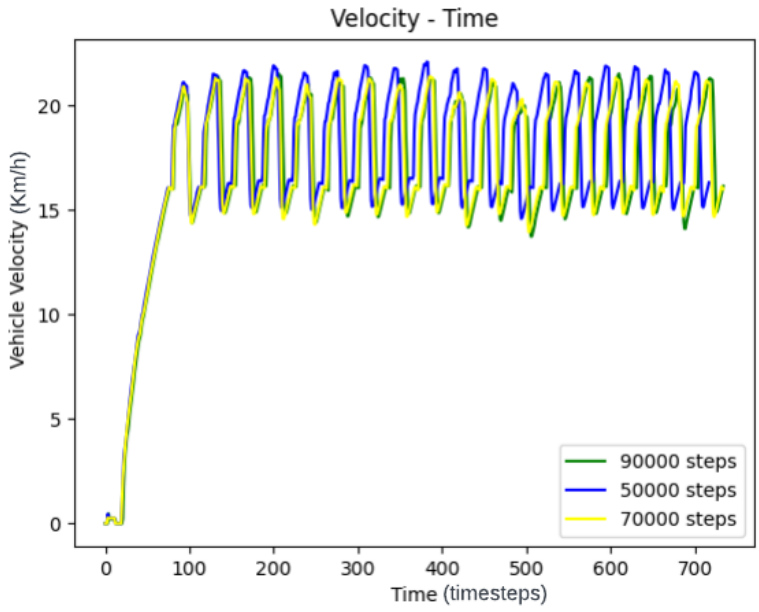


Figure 5.19: Velocity *vs.* Time for the case of Map Town10

The agent trained for 50000 steps has the same speed and accuracy with the other, but better driving performance, as far as it concerns the driver's comfort since it is steering a lot less.

Moreover in Figures 5.20 and 5.21 the trajectory of the Ecocar that was created in real time (green line) is depicted on top of the optimal path (red line) for both cases during an episode of the testing of the agent with the model of 50000 steps.

Figure 5.20: The Vehicle's Trajectory for the case of Map Town5

Figure 5.21: The Vehicle's Trajectory for the case of Map Town10

In addition in Figures 5.22 and 5.23 5.24 and 5.25 snapshots of the same trajectories are presented from an up-close angle.

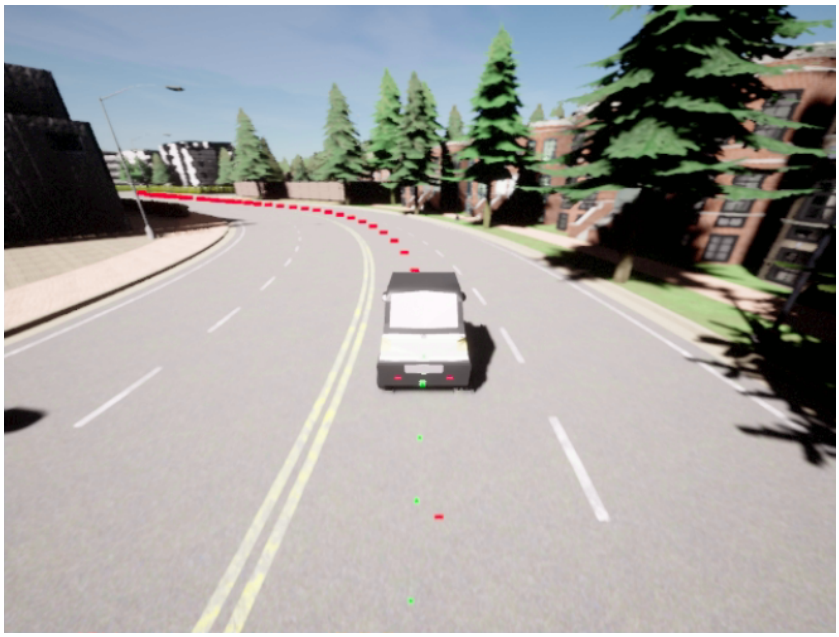


Figure 5.22: An Up-Close Vehicle's Trajectory (Map Town5, Snapshot 1)

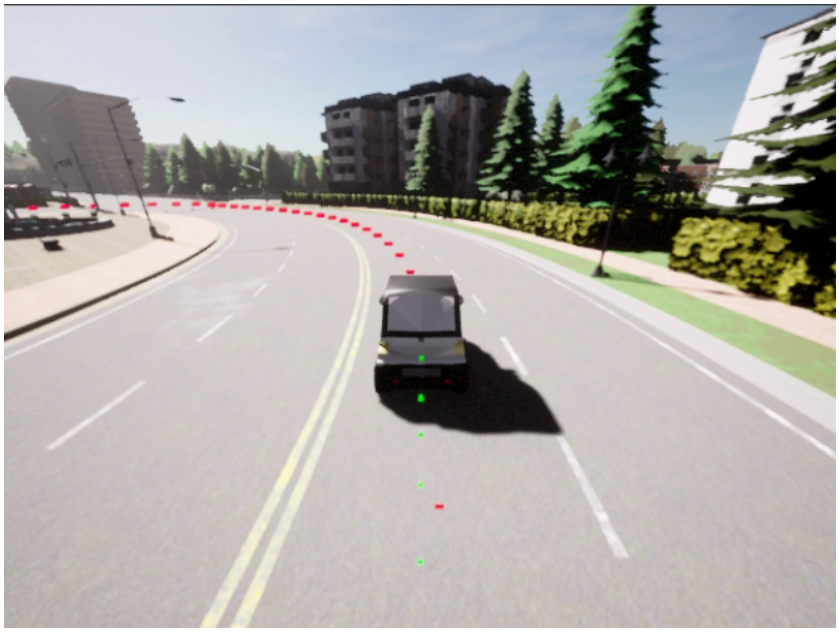


Figure 5.23: An Up-Close Vehicle's Trajectory(Map Town5, Snapshot 2)

Figure 5.24: An Up-Close Vehicle's Trajectory (Map Town10, Snapshot 1)

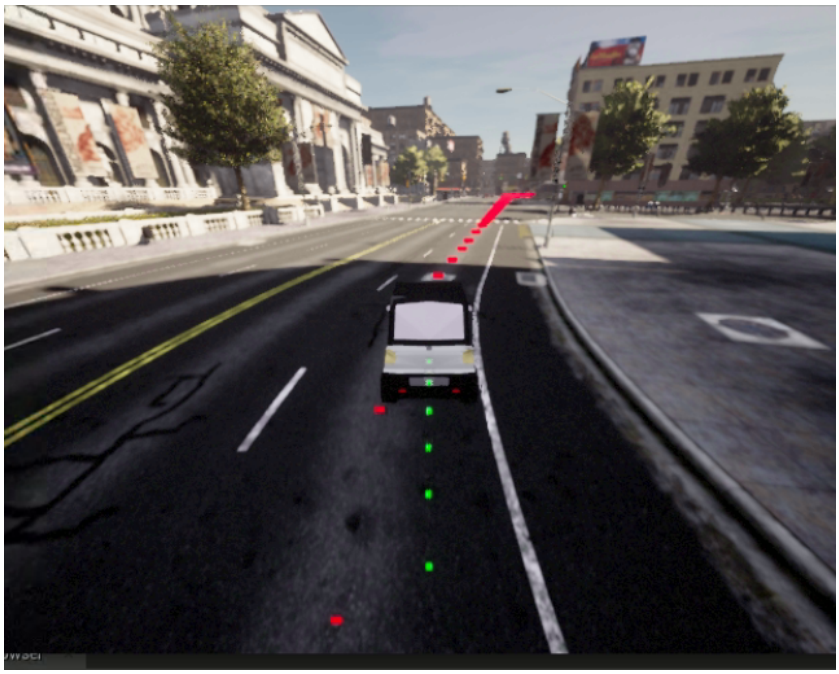


Figure 5.25: An Up-Close Vehicle's Trajectory (Map Town10, Snapshot 2)

As a final test the agent using the best model was compared with Carla's pre-installed agent. This agent is hard coded, taking information *directly from the simulation and not sensors* and can guide the vehicle to specific points on the map. With this in mind the pre-installed was tasked to follow the target points that constitute the optimal path in both maps in the same way the DDPG agent did.

Initially the agent was tested in the task of guiding the vehicle to the end of the optimal path, by comparing the coordinates in Figures 5.26 and 5.27. Where it is observed that both agents have the exact same performance.

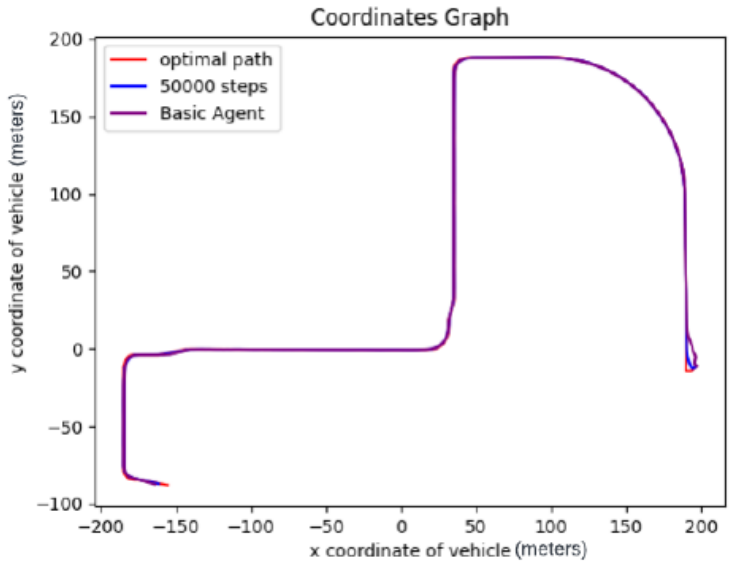


Figure 5.26: Coordinates Map Town5

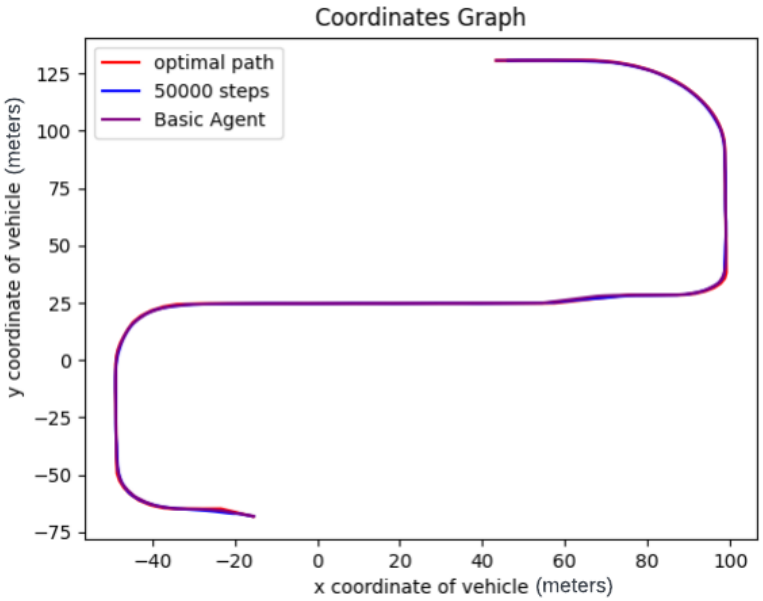


Figure 5.27: Coordinates Map Town1

The difference in the way the two agents make the vehicle steer is interesting as it is presented in Figures 5.28 and 5.29. While on straight lines the Carla agent gives steering values of 0 when it necessary to turn it steers in a much more abrupt manner than the DDPG agent. It gives a high steer value and then immediately a low one to keep the vehicle under control, which is less realistic and comfortable than what the trained agent does which although is more noisy it turns in a much smoother and way with continuous low steering values as it was penalised for high ones.

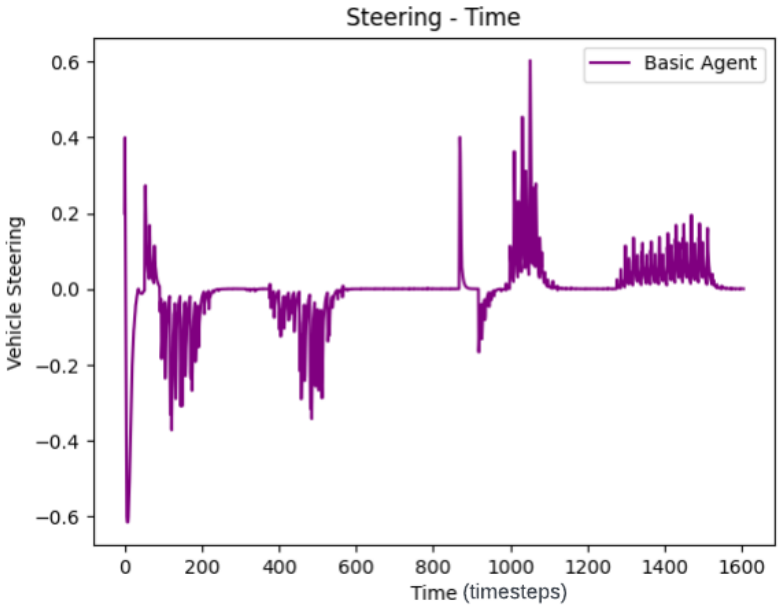


Figure 5.28: Carla Agent Steering

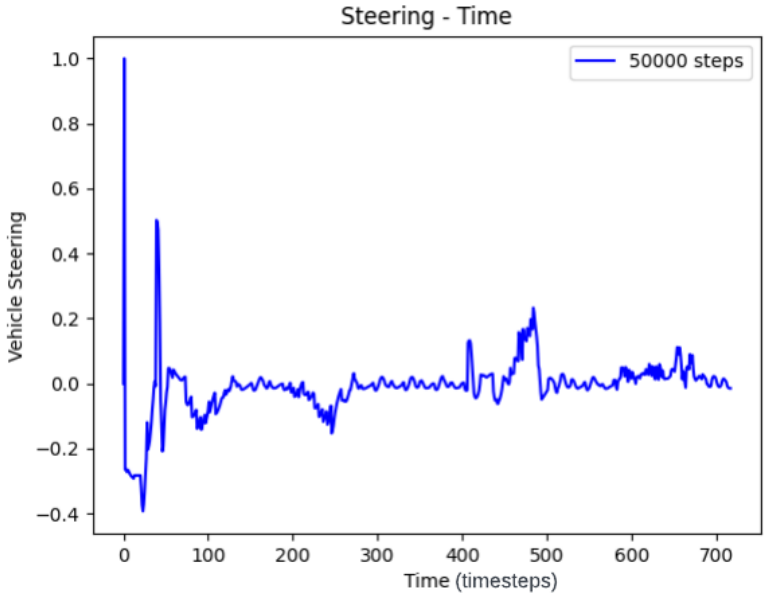


Figure 5.29: Velocity *vs.* Time for the case of Map Town5

The last comparison was between the velocities of the two agents, which are presented in Figures 5.30 and 5.31. While the Carla agent speeds up and maintains a steady speed throughout the distance it is on average three to four times slower than the one the DDPG agent maintains. The DDPG agent fluctuates between 15 and 20 km/h which is not a big difference that could cause nausea to the passenger, while the Carla agent maintains a steady speed of 4,5 km/h.

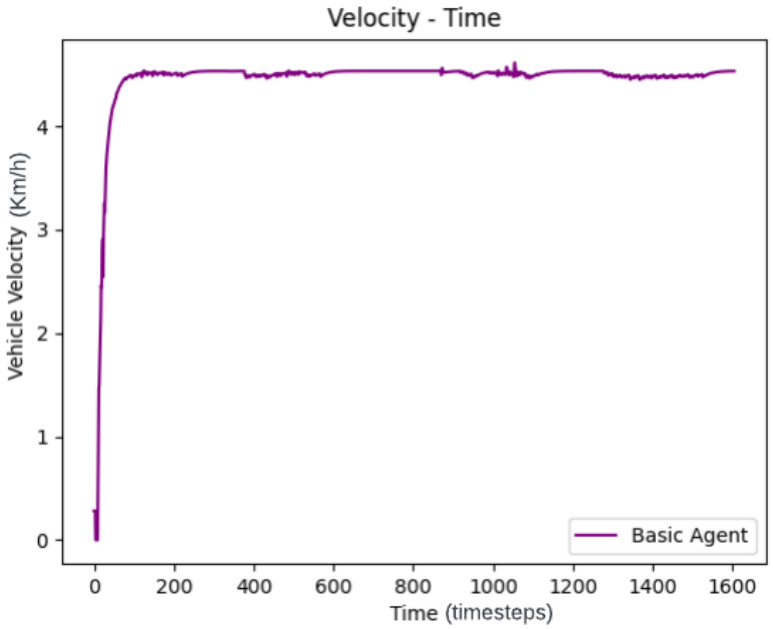


Figure 5.30: 50000 Steps Model Velocity

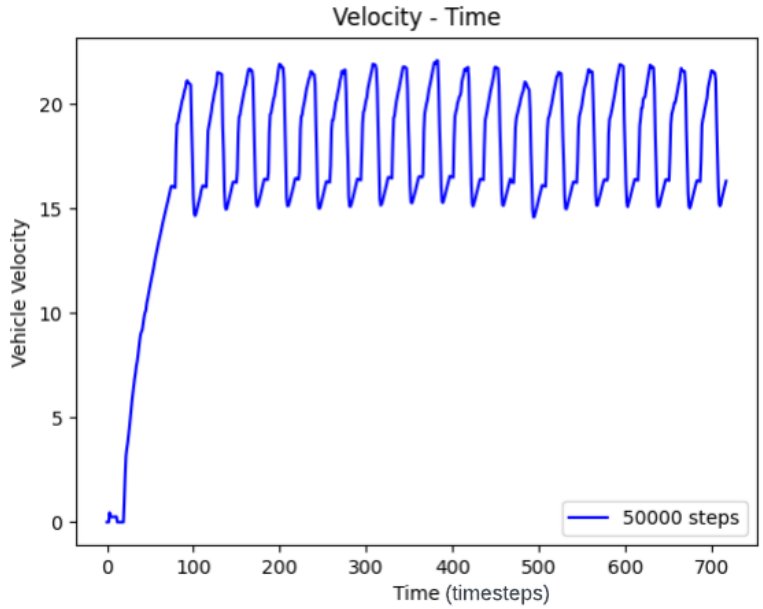


Figure 5.31: 50000 Steps Model Velocity

To summarize the trained agent seems not only to perform the task of path following successfully but also to adapt to the environment condition and arguably surpass the performance of the hard-coded Carla agent that has the advantage of taking data directly from the simulation and to not rely on complex sensor data processing to decide on its next move.

Chapter 6

Conclusions and Future Work

Upon the completion of this thesis, a simulated model of the Ecocar vehicle available to the Intelligent Systems and Robotics laboratory of the Technical University of Crete, and the potential sensors that can be integrated to it, towards developing an autonomous vehicle have been smoothly integrated into the Carla simulation environment and a gym environment has been created so the python API can smoothly communicate with Carla. The python API receives and the data from different sensors like the GNSS and IMU sensors that provide the position and orientation of the vehicle, while simultaneously receives a potential target point at each step of the simulation and processes them, along with the data regarding the vehicles speed. That processed data is then used as input to the DDPG algorithm which trains the agent to perform the task of path following. The final trained agent as it previously presented is is compared with the hard coded Carla agent on the same task. Those result are rather positive as they indicate that the concept of creating a digital twin of a real vehicle with real sensors, that utilizes only realistic information to train and perform tasks is actually possible. It should be noted that the hard-coded agent runs of a different script that the trained agent so the time variable which is based on the time steps is inconsistent between the two agents because they update on a different pace. This time variable corresponds to the relative simulation time and not the actual one that passes and is used as a way to visualize the speed and steering of the vehicle throughout the episode. Therefore, it should not be considered as a way to measure the actual time it took the agent to finish the task, for this the mean speed during the episode can be used, which is the case of the trained agent is higher.

Future steps and research directions may include the adoption of the sensors already implemented, so that the vehicle will be able to perform real time collision avoidance in dynamic environments. We should also consider creating an even more realistic model of the actual car, taking into account parameters which weren't available at the time of this thesis development. Our long term goal is to connect the simulated work with a real working prototype so that we can have a fully functional digital twin, which will allow the rapid prototyping and testing of autonomous navigation strategies for the real electric vehicle. We strongly believe that small scale autonomous electric vehicles will play a significant role in urban mobility in the years to come, therefore making our approach, good starting point towards developing

a dedicated test-bed for the development of autonomous agents, able to guide the vehicles safely in an urban environment

Bibliography

1. Society of Automotive Engineers. *Levels of Driving Automation (May 3, 2021)*
2. Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A. & Koltun, V. *CARLA: An Open Urban Driving Simulator* 2017. arXiv: 1711.03938 [cs.LG].
3. OpenAI. *Deep Deterministic Policy Gradient - Spinning Up documentation* 2018. https://spinningup.openai.com/en/latest/algorithms/ddpg.html.
4. Hart, P. E., Nilsson, N. J. & Raphael, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics* **4,** 100–107 (1968).
5. Ester, M., Kriegel, H.-P., Sander, J. & Xu, X. *A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise* in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining* (AAAI Press, Portland, Oregon, 1996), 226–231.
6. Wikipedia contributors. *RGB color model — Wikipedia, The Free Encyclopedia* [Online; accessed 14-July-2023]. 2023. https://en.wikipedia.org/w/index.php?title=RGB_color_model&oldid=1164653111.
7. Wikipedia contributors. *HSL and HSV — Wikipedia, The Free Encyclopedia* [Online; accessed 15-July-2023]. 2023. https://en.wikipedia.org/w/index.php?title=HSL_and_HSV&oldid=1161414004.
8. Online, A. *Color Model Conversion function-ArcGIS Online — Documentation* https://doc.arcgis.com/en/arcgis-online/analyze/color-model-conversion-function.htm.
9. Wikipedia contributors. *Grayscale — Wikipedia, The Free Encyclopedia* [Online; accessed 15-July-2023]. 2023. https://en.wikipedia.org/w/index.php?title=Grayscale&oldid=1162389246.
10. Wikipedia contributors. *Gaussian blur — Wikipedia, The Free Encyclopedia* [Online; accessed 15-July-2023]. 2023. https://en.wikipedia.org/w/index.php?title=Gaussian_blur&oldid=1154879725.
11. OpenCV. *OpenCV: Canny Edge Detection* https://docs.opencv.org/3.4/da/d22/tutorial_py_canny.html.
12. Wikipedia contributors. *Canny edge detector — Wikipedia, The Free Encyclopedia* [Online; accessed 15-July-2023]. 2023. https://en.wikipedia.org/w/index.php?title=Canny_edge_detector&oldid=1165475521.
13. OpenCV. *OpenCV: Hough Line Transform* https://docs.opencv.org/3.4/d9/db0/tutorial_hough_lines.html.
14. Pérez-Gil, Ó. *et al.* Deep reinforcement learning based control for autonomous vehicles in Carla. *Multimedia Tools and Applications* **81,** 3553–3576 (2022).

15. Li, D. & Okhrin, O. Modified DDPG car-following model with a real-world human driving experience with CARLA simulator. *Transportation Research Part C: Emerging Technologies* **147,** 103987. ISSN: 0968-090X (2023).
16. Youssef, F. & Houda, B. *Deep Reinforcement Learning with External Control: Self-Driving Car Application* in *Proceedings of the 4th International Conference on Smart City Applications* (Association for Computing Machinery, Casablanca, Morocco, 2019). ISBN: 9781450362894.
17. Jianyu Chen and Shengbo Eben Li and Masayoshi Tomizuka. Interpretable End-to-end Urban Autonomous Driving with Latent Deep Reinforcement Learning. *CoRR* **abs/2001.08726.** arXiv: 2001.08726 (2020).
18. Team, C. *Quick start package installation - CARLA Simulator* https://carla.readthedocs.io/en/latest/start_quickstart/.
19. Team, C. *Linux build - CARLA Simulator* https://carla.readthedocs.io/en/latest/build_linux/.
20. EcoCar & EcoCar. Technical Specifications - ecocar. *ecocar.* https://ecocar.city/technical-specifications/?lang=en (Apr. 2018).
21. EcoCar & EcoCar. New Ecocar City - ecocar. *ecocar.* https://ecocar.city/new-ecocar-city/?lang=en (July 2023).
22. Zhang, R., Li, K., Yu, F., He, Z. & Yu, Z. Novel electronic braking system design for EVS based on constrained nonlinear hierarchical control. *International Journal of Automotive Technology* **18,** 707–718 (Aug. 2017).
23. Team, C. *Add a new vehicle - CARLA Simulator* https://carla.readthedocs.io/en/latest/tuto_A_add_vehicle/.
24. Team, C. *Maps - CARLA Simulator* https://carla.readthedocs.io/en/latest/core_map/#waypoints.
25. Wikipedia contributors. *Lidar — Wikipedia, The Free Encyclopedia* [Online; accessed 15-July-2023]. 2023.
26. Ouster. *High-resolution OS1 lidar sensor: robotics, trucking, mapping* https://ouster.com/products/scanning-lidar/os1-sensor/.
27. Open3D. *Point cloud — Open3D 0.17.0 documentation* http://www.open3d.org/docs/release/tutorial/geometry/pointcloud.html#.
28. Lillicrap, T. P. *et al. Continuous control with deep reinforcement learning* 2019. arXiv: 1509.02971 [cs.LG].
29. Raffin, A. *et al.* Stable-Baselines3: Reliable Reinforcement Learning Implementations. *Journal of Machine Learning Research* **22,** 1–8 (2021).
30. Brockman, G. *et al.* OpenAI Gym. *CoRR* **abs/1606.01540.** arXiv: 1606.01540 (2016).
31. LeBlanc, D. & Lee, G. General Deep Reinforcement Learning in NES Games. *Proceedings of the Canadian Conference on Artificial Intelligence* (June 2021).
32. Commons, W. *File:NES-Controller-Flat.jpg — Wikimedia Commons, the free media repository* [Online; accessed 15-July-2023]. 2020. https://commons.wikimedia.org/w/index.php?title=File:NES-Controller-Flat.jpg&oldid=495694314.
33. Silver, D. *et al.* Deterministic Policy Gradient Algorithms. *31st International Conference on Machine Learning, ICML 2014* **1** (June 2014).
34. Gong, H., Wang, P., Ni, C. & Cheng, N. Efficient Path Planning for Mobile Robot Based on Deep Deterministic Policy Gradient. *Sensors* **22.** ISSN: 1424-8220. https://www.mdpi.com/1424-8220/22/9/3579 (2022).

35. OpenAI. *Make your own custom environment - Gym Documentation* 2022. https://www.gymlibrary.dev/content/environment_creation/.
36. Team, S. B. *Stable Baselines Documentation Wiki* 2022. https://stable-baselines3.readthedocs.io/en/master/.
37. Wikipedia contributors. *Atan2 — Wikipedia, The Free Encyclopedia* [Online; accessed 14-July-2023]. 2023. https://en.wikipedia.org/w/index.php?title=Atan2&oldid=1156958741.
38. Kingma, D. & Ba, J. Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations* (Dec. 2014).