CARLA-based Simulation Environment for Testing and Developing Autonomous Vehicles in the Linden Residential Area

Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Pedro Jezel Fernandez Narvaez

Graduate Program in Electrical and Computer Engineering

The Ohio State University

2021

Abstract

The use of autonomous vehicles (AV) for public passenger transport has rapidly grown in the past few year within the mobility industry. They provide a flexible solution that can help reduce traffic congestion, energy consumption, safety, among many others. In 2020, the Smart Columbus Initiative (SCI) deployed two level-four autonomous shuttles to help solve the first mile/last-mile mobility challenge in Linden Residential Area. This thesis focuses on providing a realistic simulation platform for this real-life scenario. The environment use the CARLA (Car Learning to Act) simulator as a backbone and provides co-simulations such as SUMO (Simulation of Urban Mobility) for a more realistic traffic simulation and Autoware for a realistic autonomous driving stack. Multiple traffic scenarios are provided, including NHTSA's (National Highway Traffic Safety Administration) pre-crash scenarios, to test the safety and decision-making of the shuttles [1]. An evaluation and rating scheme is also introduced and illustrated using autonomous driving in the Linden Residential Area soft environment.

Dedication

This thesis is dedicated to my partner, mother, and grandmother.

Acknowledgment

I want to thank everyone who directly or indirectly helped me achieve a higher education degree. Thanks to my mentor, Dr. Levent Guvenc, for providing me with the necessary guidance throughout my entire research.

It was an honor to be part of the Automated Driving Lab (ADL) and be able to work with the people that are part of it. Specially, I want to thank Karina Meneses for her suggestions and input on my work.

Finally, I want to thank my partner, mother, and grandfathers for their support on everything.

Vita

Education

August 2019 - May 2021	M.S. Electrical and Computer Engineering The Ohio State University
August 2013 – December 2018	B.S. Computer Engineering University of Puerto Rico, Mayaguez Campus

Experience

January 2021 - May 2021	GRA-GTA Automated Driving Lab The Ohio State University
May 2020 – August 2020	Software Engineer Intern Qualcomm - Columbus, OH
August 2019 – May 2020	GEM Fellow The Ohio State University
May 2019 – August 2019	Software Engineer Intern Qualcomm - San Diego, CA
June 2018 – August 2018	Software Engineer Intern Qualcomm - San Diego, CA
June 2017 – August 2017	Software Engineer Intern Honeywell Aerospace – Aguadilla, PR

Fields of Study

Major field: Electrical and Computer Engineering.

Table of Contents

List of Tables

List of Figures

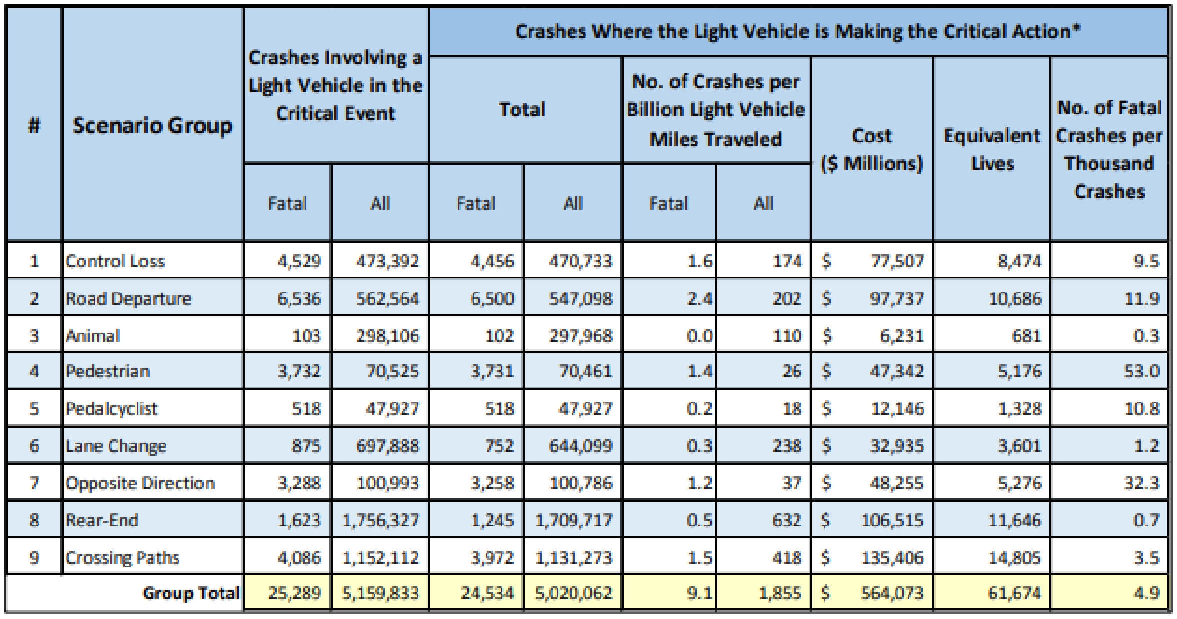
Chapter 1: Introduction

1.1 Introduction to Autonomous Shuttles

Autonomous Vehicles (AV) are starting to revolutionize urban mobility around the globe. Several cities around the United States (US) have already launched pilot programs involving autonomous shuttles. These types of intelligent transportation systems are expected to accommodate different needs and connect residents to different resources. Between 2014 and 2017, more than $80 billion were invested in the AV industry [2]. Companies like May Mobility and EasyMile are some of the startups that are providing cities like Columbus circulates with low-speed self-driving shuttles. These vehicles, usually, travel a fixed route as circulates and are equipped with a sensor stack that allows them to sense the environment, plan, and act accordingly to achieve a specific task. The level of automation will depend on how close they perform the tasks compared to humans. Currently, there are six levels or tiers of autonomous driving capabilities recognized by the Society of Automotive Engineers (SAE) and the NHTSA which range from no autonomation (the human performs all driving tasks) to full automation (the vehicle performs all driving tasks) [3].

1.2 Background

Due to the nature of urban environments, autonomous shuttles are forced to interact with other vehicles, pedestrians, static objects, etc. Thus, there is a big concern when it comes to safety. They must pass a series of safety and operational tests before operating in real life. How can we be sure that the system is safe enough? There are millions of things than can make a system fail. On February 20 of 2020, for example, one of the Linden LEAP (Linden Empowers All People) self-driving shuttles had an incident where a sudden unexpected stop at the very low speed of about 7 mph caused a passenger to fall to the floor and later got medical attention [4]. It was determined that a deviation in the steering of the shuttle was the culprit [4]. Something simple, yet powerful enough to cause harm to the passengers. This incident is just one of millions that happens every year. In fact, a publication from NHTSA accounted for approximately 6,170,000 police reported crashes involving 10,945,000 vehicles with an estimated loss of $120 billion dollars in the United States alone based on 2004 General Estimates Systems (GES) statistics [1]. The report provided 37 pre-crash scenario topologies for light vehicles such as minivans, passenger cars, light pickup trucks, etc. Pre-crash scenarios describe vehicle movements and critical events that occur immediately prior the accidents [1]. The scenarios include Control Loss With Prior Vehicle Action, Vehicles (s) Turning – Same Direction, Lead Vehicle Decelerating, to name a few [1]. In August 2019, a new report updated the 2007 pre-crash scenario in terms of typology and crash characteristics that accounted for new emerging crash avoidance technology [5]. Table 1 depicts Yearly Average Statistics based on 2011-2015 Fatality Analysis Reporting System (FARS) and GES.



#	Scenario Group	Crashes Involving a Light Vehicle in the Critical Event		Crashes Where the Light Vehicle is Making the Critical Action*						
				Total		No. of Crashes per Billion Light Vehicle Miles Traveled		Cost ($ Millions)	Equivalent Lives	No. of Fatal Crashes per Thousand Crashes
		Fatal	All	Fatal	All	Fatal	All			
1	Control Loss	4,529	473,392	4,456	470,733	1.6	174	$ 77,507	8,474	9.5
2	Road Departure	6,536	562,564	6,500	547,098	2.4	202	$ 97,737	10,686	11.9
3	Animal	103	298,106	102	297,968	0.0	110	$ 6,231	681	0.3
4	Pedestrian	3,732	70,525	3,731	70,461	1.4	26	$ 47,342	5,176	53.0
5	Pedalcyclist	518	47,927	518	47,927	0.2	18	$ 12,146	1,328	10.8
6	Lane Change	875	697,888	752	644,099	0.3	238	$ 32,935	3,601	1.2
7	Opposite Direction	3,288	100,993	3,258	100,786	1.2	37	$ 48,255	5,276	32.3
8	Rear-End	1,623	1,756,327	1,245	1,709,717	0.5	632	$ 106,515	11,646	0.7
9	Crossing Paths	4,086	1,152,112	3,972	1,131,273	1.5	418	$ 135,406	14,805	3.5
	Group Total	25,289	5,159,833	24,534	5,020,062	9.1	1,855	$ 564,073	61,674	4.9

Table 1.1: *Yearly Average Statistic – Scenario Groups Based [5]*

Based on the NHTSA National Motor Vehicle Crash Causation Survey (NMVCCS), the final cause of 94% of crashes are due to driver errors [6]. It is often assumed that AVs should decrease this number by taking out the human driver factor [6]. However, only a third, 33.1%, of crashes could likely be preventable by AVs if they are not designed and developed with the sufficient capabilities [6]. Autonomous vehicles are not exempt from the type of pre-crash scenarios presented in the NHTSA report. They will be presented with challenging scenarios and they must be able to overcome them, more so when the task in hand involves passengers.

1.3 Objectives and Scope

Autonomous vehicles are about developing and testing systems that can respond to the millions of scenarios that a vehicle can handle in a safely manner. This is where simulations come in. They enable companies and individuals to cut development time and reduce costs. We can say that simulations will never be the same as road testing but that is outside the scope of this thesis. Simulations enable us to assess different aspects of a system and validate them. This thesis aims to provide a simulation environment that enables researchers, students, and people from the industry to study, test and develop algorithms in a real-life based environment. Providing all kinds of scenarios that autonomous shuttles can face throughout their journey. Scenarios includes turning, obstacle avoiding, pedestrians, roundabouts, and many others. To achieve this we used CARLA, an open-source simulator for autonomous driving research.

1.4 Thesis Outline

The thesis is divided as follows:

- **Chapter 1** introduces autonomous shuttles and the scope of this thesis.
- **Chapter 2** talks about the tools that were used to build the environment and the steps to create it.
- **Chapter 3** discusses everything related to traffic scenarios and how they were incorporated in the work done in this thesis in more depth.
- **Chapter 4** examines the ways in which autonomous agents can be evaluated and benchmarked using the work done in this thesis.
- **Chapter 5** goes through the metrics and scores that can be obtained from the benchmarking tools.
- **Chapter 6** discusses the contributions of this thesis and the next steps.

Chapter 2: Simulation Environment

Simulation has proved to be effective in the self-driving vehicles world. It enables developers to virtually test different scenarios in a cost-effective way. The perks of being able to test algorithms and analyze their performance are infinite. This chapter will cover some of the tools used to create and develop the simulation environment. Other tools will be discussed later.

2.1 CARLA Simulator

CARLA is an open-source simulator specifically created for autonomous driving applications. It supports the development and testing of autonomous driving algorithms in a wide range of areas such as computer vision, path planning, motion, controls, etc. CARLA is based on Unreal Engine 4 which is a state-of-the-art, real-time, open-source game engine with photorealistic graphics [7]. The simulator follows the client-server model. The server manages the simulator itself and the client manages the logic of the actors and the world. The simulator comes with a Python API that allows developers to control different aspects of the simulation such as weather, traffic, Non-Player Characters (NPCs), etc. as shown in Figure 2.1. A screenshot of CARLA simulated environment is shown in Figure 2.2.

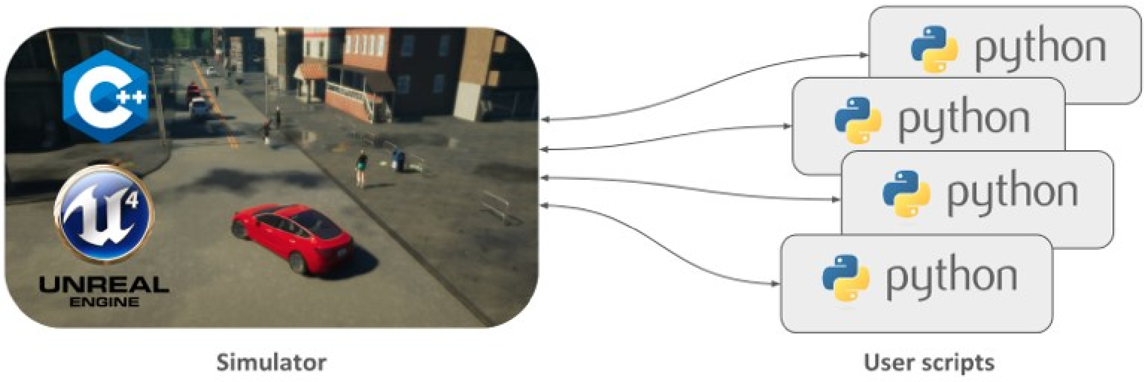
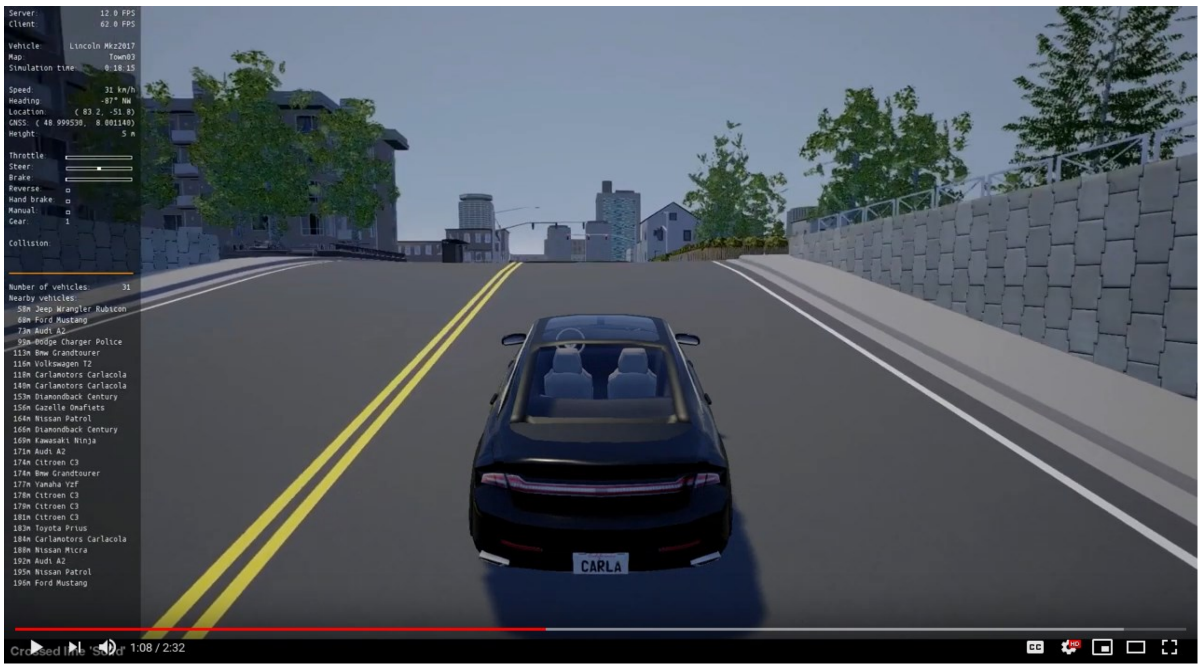


Figure 2.1: *Basic structure of CARLA [8].*



```
Server:              12.0 FPS
Client:              62.0 FPS

Vehicle:       Lincoln Mkz2017
Map:                    Town03
Simulation time:       0:18:15

Speed:                 31 km/h
Heading:                -87° NW
Location:        ( 83.2, -51.8)
GNSS: ( 48.999530,  8.001140)
Height:                    5 m

Throttle:
Steer:
Brake:
Reverse:
Hand brake:
Manual:
Gear:          1

Collision:

Number of vehicles:         31
Nearby vehicles:
  58m Jeep Wrangler Rubicon
  68m Ford Mustang
  73m Audi A2
  99m Dodge Charger Police
 113m Bmw Grandtourer
 116m Volkswagen T2
 118m Carlamotors Carlacola
 140m Carlamotors Carlacola
 153m Diamondback Century
 156m Gazelle Omafiets
 164m Nissan Patrol
 166m Diamondback Century
 169m Kawasaki Ninja
 171m Audi A2
 174m Citroen C3
 174m Bmw Grandtourer
 177m Yamaha Yzf
 178m Citroen C3
 179m Citroen C3
 181m Citroen C3
 183m Toyota Prius
 184m Carlamotors Carlacola
 188m Nissan Micra
 192m Audi A2
 195m Nissan Patrol
 196m Ford Mustang
```

Figure 2.2: *CARLA simulator [9].*

2.2 RoadRunner

RoadRunner is a software that lets you easily customize 3D scenes for simulating and testing autonomous driving systems [8]. It provides tools to create custom roads, junctions, traffic signals and more. To make the environment more realistic, you can insert traffic signs, foliage, and props. The program supports the visualization and importing of LIDAR (Light Detection and Ranging) point clouds, Geographic Information System (GIS) data, etc. Scenes can be exported in a wide range of formats such as FBX, GLTF, USD and be used in other programs such as CARLA, Unreal Engine, LG Silicon Valley Lab (LGSVL) simulator, Unity, etc. In terms of the underlying road network, it can be exported as an OpenDrive file. Figures 2.3 and 2.4 show RoadRunner's GUI (Graphical User Interface), respectively.

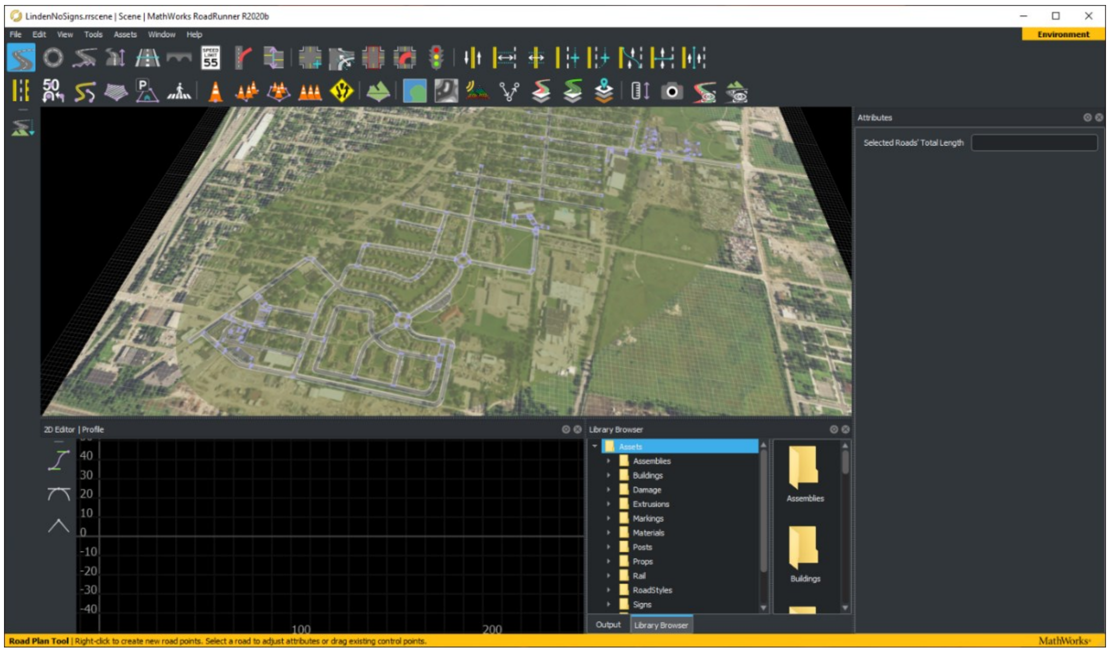

LindenNoSigns.rrscene | Scene | MathWorks RoadRunner R2020b
File Edit View Tools Assets Window Help
Environment
Attributes
Selected Roads' Total Length
2D Editor | Profile
Library Browser
Assets
Assemblies
Buildings
Damage
Extrusions
Markings
Materials
Posts
Props
Rail
RoadStyles
Signs
Output
Library Browser
Road Plan Tool | Right-click to create new road points. Select a road to adjust attributes or drag existing control points.
MathWorks

Figure 2.3: *Linden's map in RoadRunner with GIS enabled.*

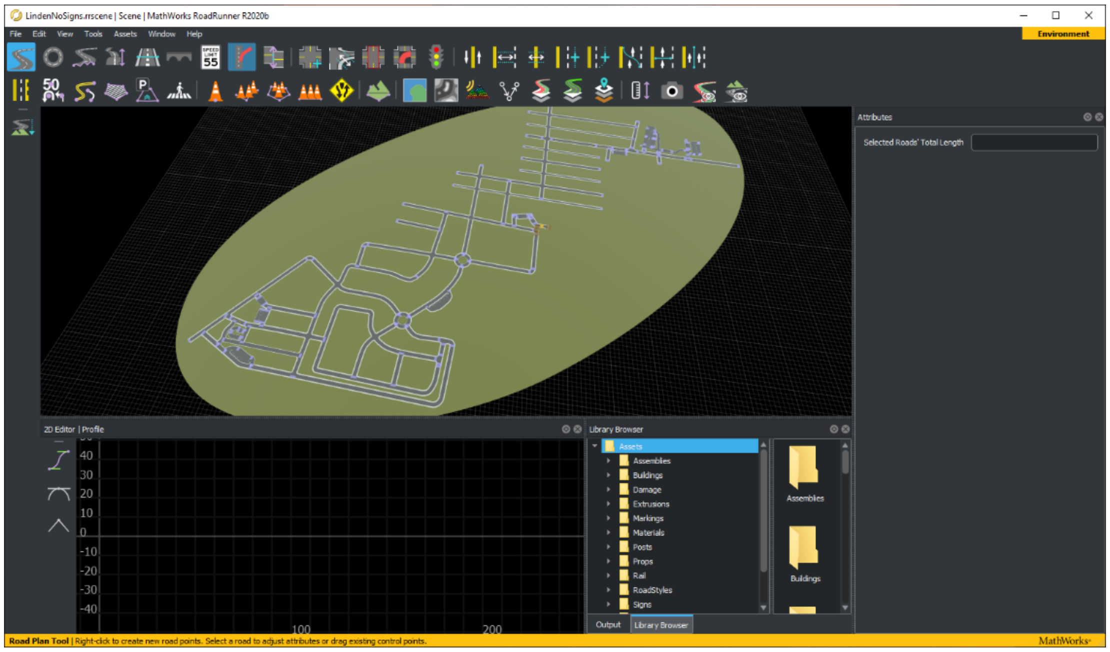

LindenNoSigns.rrscene | Scene | MathWorks RoadRunner R2020b
File Edit View Tools Assets Window Help
Environment
Attributes
Selected Roads' Total Length
2D Editor | Profile
Library Browser
Assets
Assemblies
Buildings
Damage
Extrusions
Markings
Materials
Posts
Props
Rail
RoadStyles
Signs
Output
Library Browser
Road Plan Tool | Right-click to create new road points. Select a road to adjust attributes or drag existing control points.
MathWorks

Figure 2.4: *Linden's map in RoadRunner with GIS disabled.*

2.3 ASSURE Mapping Tool

ASSURE mapping tool is a simple tool for viewing and editing road network maps utilized by autonomous vehicles [9]. It can be used via Docker, open-source software that enables OS-level (Operating System) virtualization in the form of containers, or by direct installation [10]. The built-in editor enables the user to add or modify different map semantics such as lanes, waypoints, road lines, traffic lights, etc. ASSURE allows to convert between different file formats such as OpenPlanner, OpenDrive, Lanelet2, etc. Figure 2.5 depicts the ASSURE mapping tool interface.

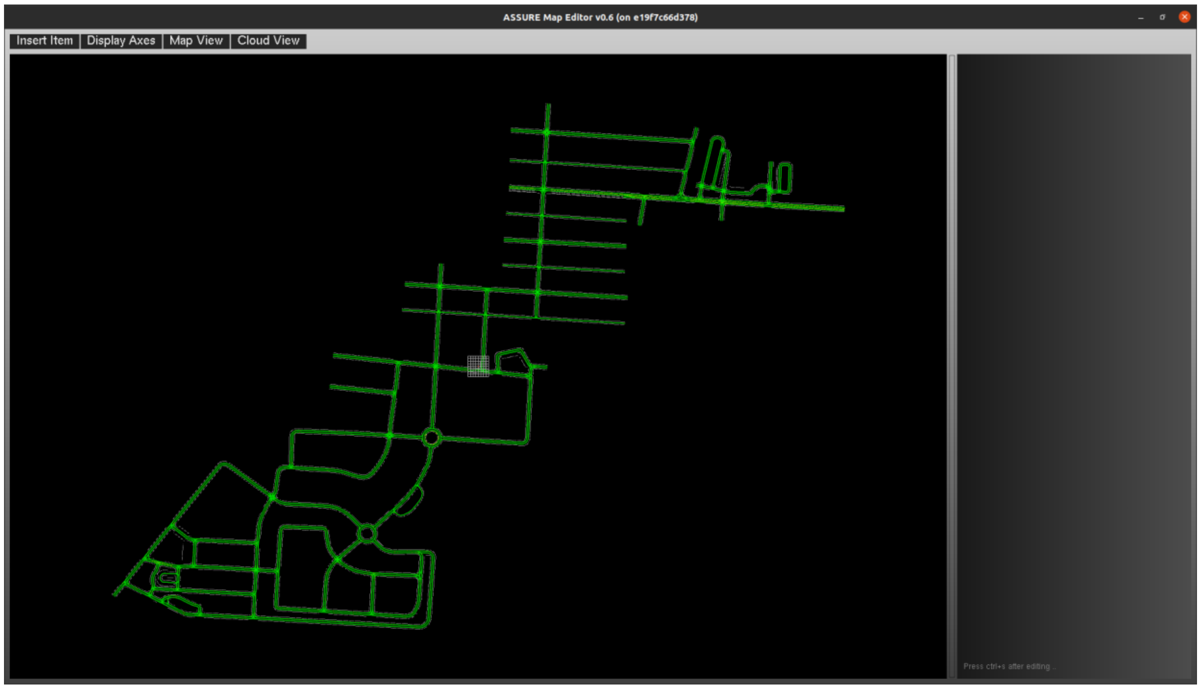

ASSURE Map Editor v0.6 (on e19f7c66d378)
Insert Item
Display Axes
Map View
Cloud View
Press ctrl+s after editing

Figure 2.5: *ASSURE mapping tool interface.*

2.4 Autoware

Autoware, as illustrated in Figure 2.6, is an open-source software for self-driving vehicles [11]. It provides a set of modules that includes sensing, perception, planning, decision, and actuation. Some of the capabilities are object detection, localization and mapping, lane detection, sensor fusion, etc. Autoware is well-suited for urban cities but can be used for highways, freeways, geofenced areas and more. It also provides ROS support. The usage of Autoware will be discussed in sections 2.6.2 and 4.4.2. It is being introduced now because some tools from Autoware were used to obtain certain autonomous maps needed for the simulation.

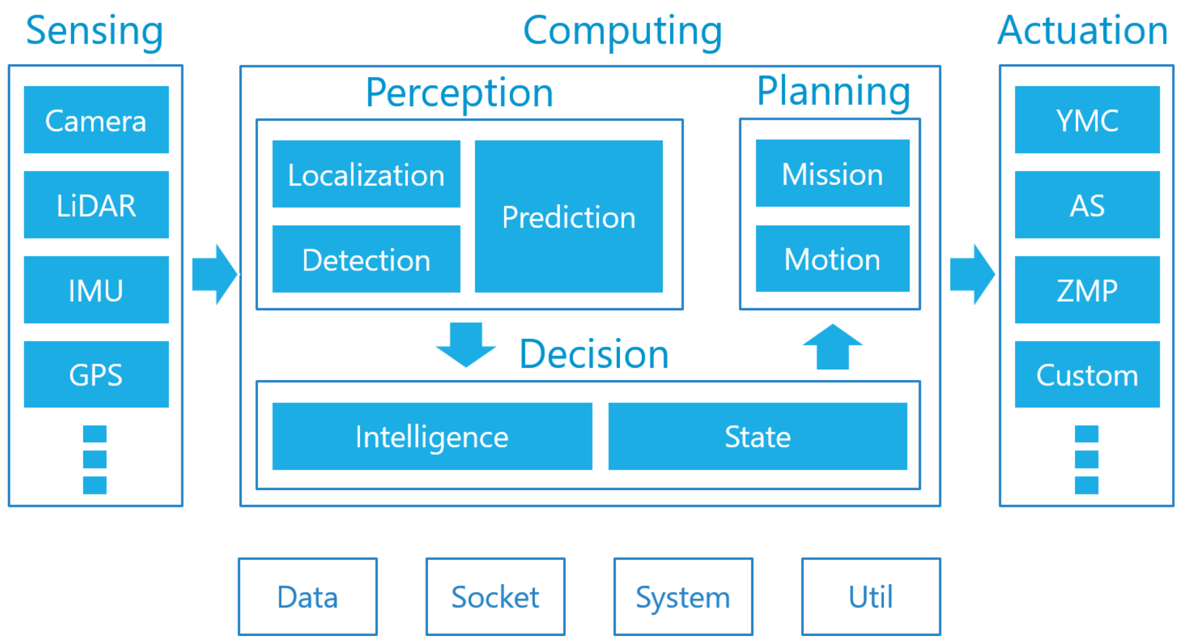
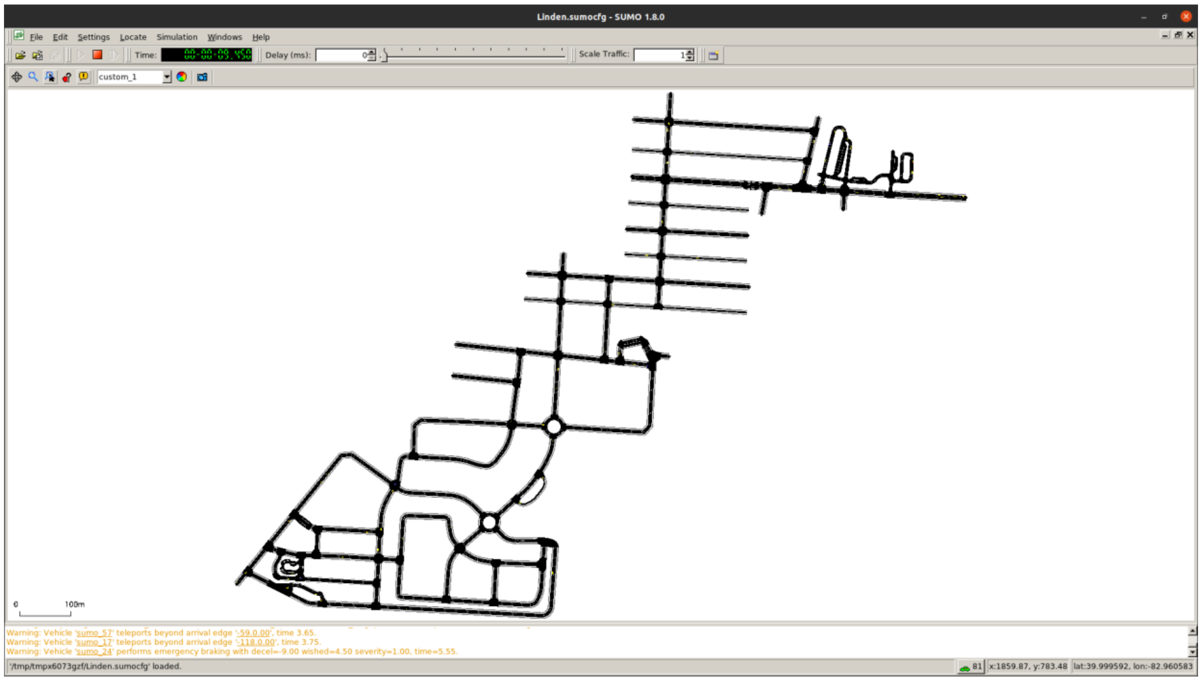


Figure 2.6: *Autoware's architecture [12]*

2.5 SUMO Traffic Simulator

Simulation of Urban Mobility (SUMO) is another open-source microscopic traffic simulator that allows modelling traffic systems [12]. SUMO allows you to integrate automated vehicles and manage the traffic in terms of speed, traffic lights, and other behavior. Also, it provides support for microscopic simulation; where you can control vehicles, pedestrians, and public transport explicitly. The simulator offers various Application Program Interfaces (APIs) to remotely control the simulation. This feature enables CARLA to perform a co-simulation and exploit both simulator's capabilities simultaneously. Figure 2.7 show SUMO's GUI.



```
Warning: Vehicle 'sumo_57' teleports beyond arrival edge '-59.0.00', time 3.65.
Warning: Vehicle 'sumo_17' teleports beyond arrival edge '-118.0.00', time 3.75.
Warning: Vehicle 'sumo_24' performs emergency braking with decel=-9.00 wished=4.50 severity=1.00, time=5.55.
'/tmp/tmpx6073gzf/Linden.sumocfg' loaded.        81  x:1859.87, y:783.48  lat:39.999592, lon:-82.960583
```

Figure 2.7: *SUMO co-simulation interface*

2.6 Simulation Environment Building Pipeline

The process of building the environment was quite tedious due to the of lack of proper documentation. This section of the thesis will cover the process of building the map from scratch to using it for simulation purposes. Additionally, it will explain how each tool was used, hoping to serve as a guide for future map modeling intended to be used on CARLA simulator or any other supported tool.

2.6.1 Building the Environment

The first step of the process was to obtain the (GIS) data of the area of interest, Linden, from the United States Geological Survey (USGS) website [13]. The data collected was elevation, imagery, and point cloud data. After collecting the data it was imported into RoadRunner to build the actual map. The imagery was used as a blueprint to place all the roads, intersections, markings, etc. The roads include the main Linden shuttle route, depicted in Figure 2.8, and surrounding streets for a more realistic simulation. The tools provided by RoadRunner were easy to use and allowed the creation of custom junctions, parking, and sidewalks without difficulty. After tracing all the roads, the map was exported as a Filmbox (*.fbx*) and as OpenDrive (*.xodr*) file and imported into a CARLA build from source. Doing it this way allowed us to use the assets from CARLA and populate the map with buildings, cars, props, etc. Other free assets from the Unreal Engine Marketplace were used to make the environment more realistic in terms of foliage, props, and structures. Figures 2.8 and 2.9 show the main route of Linden autonomous shuttle and an example roundabout of Linden, respectively.

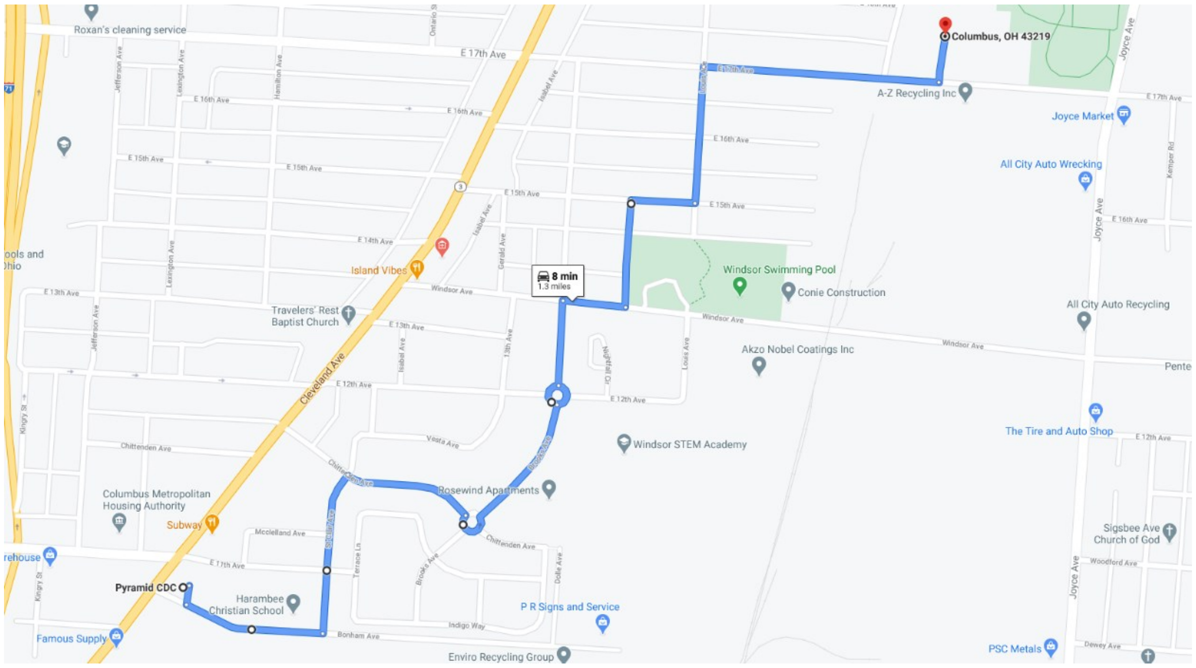


Figure 2.8: *Main Linden autonomous shuttle route [14]*

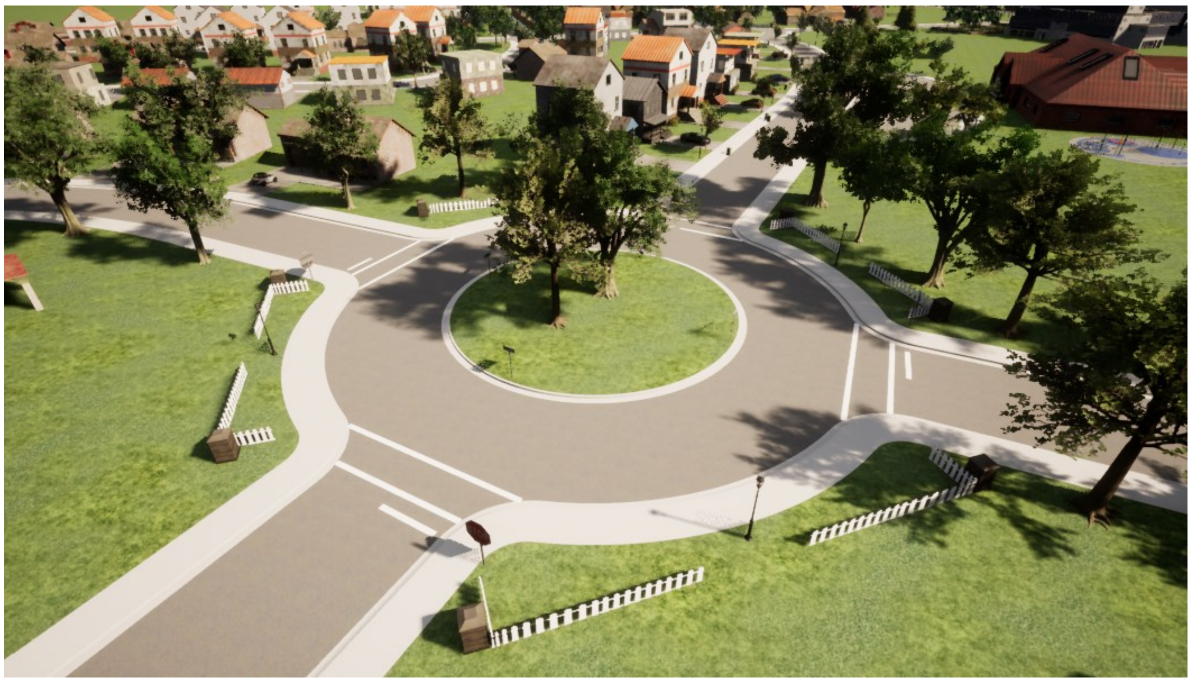


Figure 2.9: *A roundabout from the Linden map.*

2.6.2 Autonomous Driving Map Files

Up to this point, the Linden map can be used in the simulator with no issues whatsoever. However, to be able to use the Autoware agent in our custom environment, we needed two other high-definition maps apart from the OpenDrive file. These are Lanelets (*.osm*) and Aisan vector map (collection of *.csv*). Lanelets describe drivable environments from geometrical and topological perspectives [14]. The maps are represented by *lanelet* elements, interconnected road segments characterized by left and right bounds, which compose the road network with lanes, roads, and intersections [14]. On the other hand, the Aisan vector map is a proprietary mapping format developed by Aisan Technology Company. The version of Autoware that was used for this thesis, version 1.13, utilized the Aisan format internally as a default option. Table 2.1 shows all the necessary files for both the Lanelet and Aisan formats. The process to obtain both map formats was straightforward:

1) Use ASSURE Mapping Tool to convert the OpenDrive file to Lanelet.
2) Use an internal Autoware tool, lanelet2aisan [15], to convert the Lanelet file to Aisan.

Lanelet Files	Aisan Files
linden.osm	area.csv dtlane.csv intersection.csv lane.csv line.csv node.csv point.csv wayarea.csv whiteline.csv

Table 2.1: *Files generated for each file format.*

2.6.3 Point Cloud Map

Linden's point cloud map generation was a simple process. We used a package from CARLA's ROS bridge called *carla_pcl_recorder* that allows you to drive around the map, either manually or with autopilot and record the environment using a soft Light Detection and Ranging (LIDAR) sensor [16]. This package creates a subscriber node that receives messages of type *PCLPointCloud2*, which holds the raw data that is processed in a series of steps. First, it is converted to the *PointCloud<PointT>* template, that is, the base class in Point Cloud Library (PCL) for storing groups of 3D points [17]. Second, the point cloud is transformed by a 3D offset and a quaternion. Finally, the point cloud map is saved to disk as a binary file. This process is repeated for every sensor reading.

After driving around the map while recording the point cloud, we ended up with thousands of single *.pcd* files. We used the *pcl_concatenate_points_pcd* command from *pcl_tools*, which is part of Point Cloud Library, to generate a single point cloud from these *.pcd* files [18]. After that, we used the *pcl_voxel_grid* command to remove duplicate points. This process must be repeated every time a major update is done to the map (e.g. adding new buildings). Figures 2.10 and 2.11 show a close-up screenshot and visualization of the point cloud map, respectively.

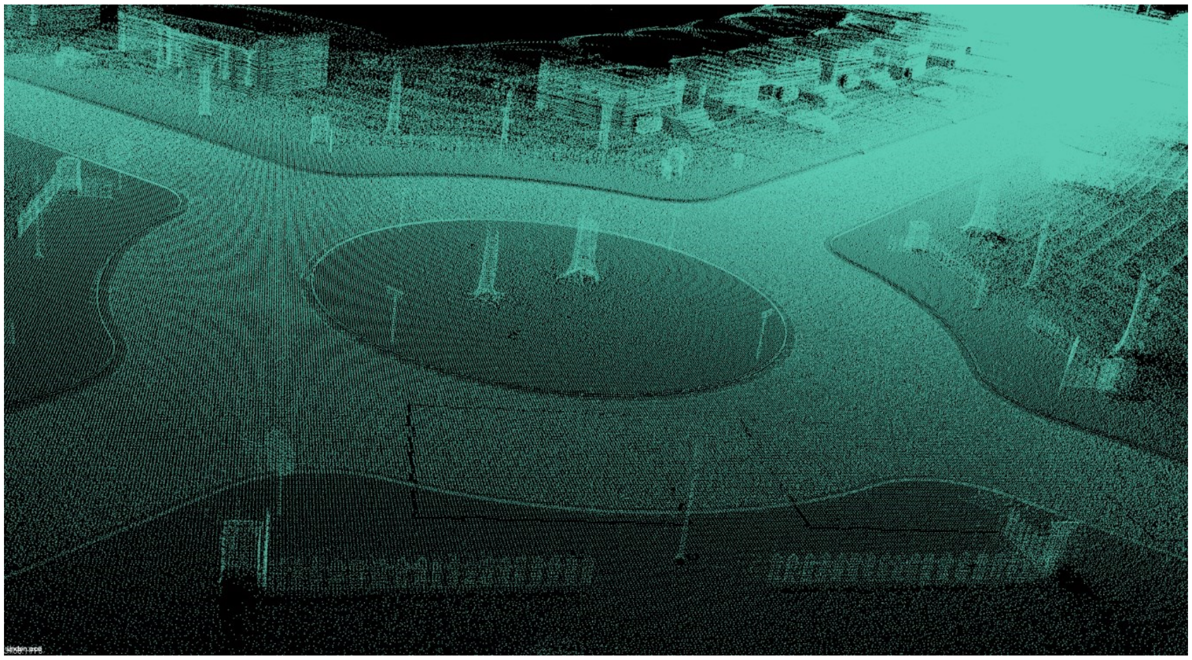


Figure 2.10: *Close-up screenshot of the recorded point cloud map.*

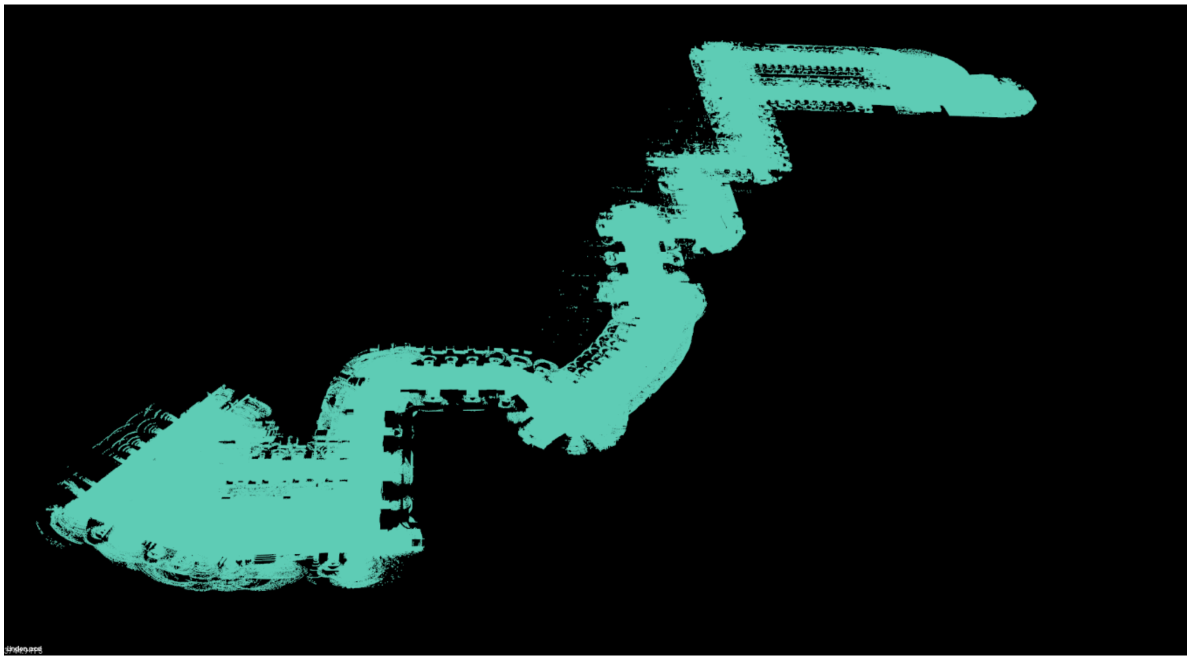


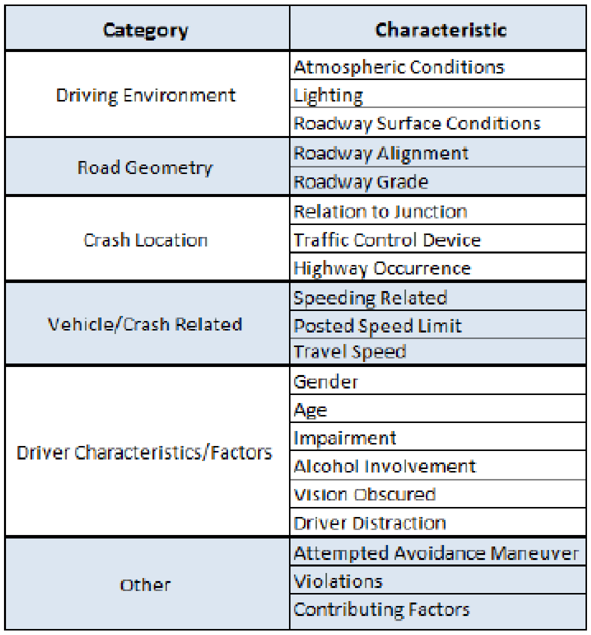
Figure 2.11: *Visualization of the whole point cloud map.*

Chapter 3: Traffic Scenarios

Autonomous shuttles are about human safety. Ethically, morally, and legally, the systems should and must be robust to a certain degree. For this purpose, it is a key factor that simulations provide the tools or means to test complex traffic scenarios. The base simulation platform is provided by CARLA. The realistic road network is provided by RoadRunner. The microscopic simulation is provided by SUMO.

3.1 Pre-Crash Typology

To evaluate agents in the Linden environment, multiple scenarios were defined and other were already defined in CARLA. Most of the traffic scenarios are based on the NHTSA pre-crash topology [1], [5]. This allows us to concentrate the evaluation and testing of agents in high occurring scenarios. The scenarios are grouped into nine (9) pre-crash scenario groups: control loss, road departure, animal, pedestrian, pedal cyclist, lane change, opposite direction, rear-end, and crossing paths [5]. Each scenario has different characteristics that help to quantify the contributing factors of a crash. Table 3.1 depicts these characteristics. This thesis, the traffic scenarios focus on characteristics that can affect an autonomous agent. For example, you will not find a traffic scenario with alcohol involvement, age, gender, etc. contributing factors for obvious reasons.



Category	Characteristic
Driving Environment	Atmospheric Conditions
	Lighting
	Roadway Surface Conditions
Road Geometry	Roadway Alignment
	Roadway Grade
Crash Location	Relation to Junction
	Traffic Control Device
	Highway Occurrence
Vehicle/Crash Related	Speeding Related
	Posted Speed Limit
	Travel Speed
Driver Characteristics/Factors	Gender
	Age
	Impairment
	Alcohol Involvement
	Vision Obscured
	Driver Distraction
Other	Attempted Avoidance Maneuver
	Violations
	Contributing Factors

Table 3.1: *Traffic scenarios characteristics [5]*

3.2 ScenarioRunner

ScenarioRunner is a module from CARLA that allows developers to define and execute custom traffic scenarios within the simulator [19]. The scenarios can be defined either through a Python interface or the OpenScenario standard [19]. ScenarioRunner can be used to replicate specific conditions to evaluate agents on it. Custom metrics can be defined to facilitate the evaluation and validation of agents. The module lets you run single scenarios and route-based scenarios. This is useful if you want to benchmark specific segments of a route or the complete route itself. The following subsections will describe each of the traffic scenarios currently supported by our custom map.

3.2.1 Scenario I – Control Loss

Description: the ego-vehicle is going straight and loses control due to wet road conditions or something in the road. The ego-vehicle must regain control of the vehicle and continue to a target point to finish the scenario execution. In this scenario, noise is added to the steering and throttle data (jittered) to simulate the slippery road or object bump. Figure 3.1 show how the scenario looks like.

Road geometry: rural area.

Weather conditions: wet, rainy, cloudy.

Vehicle speed: variable.

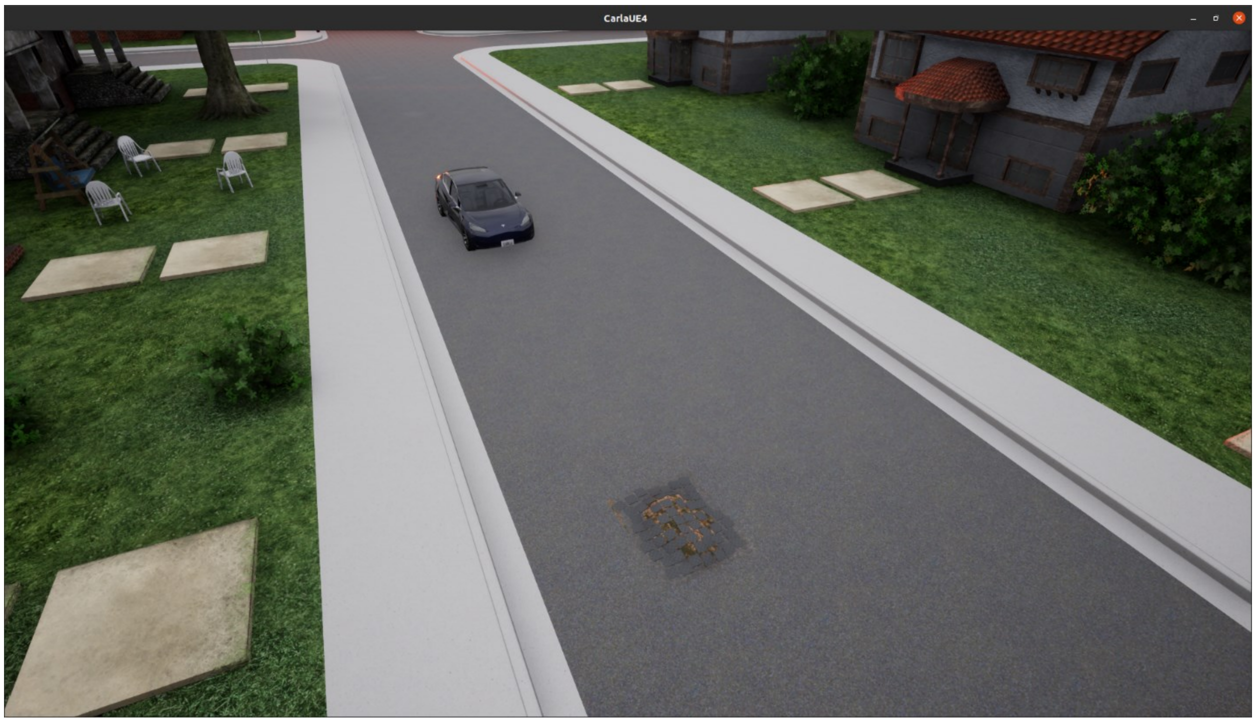


Figure 3.1: *Vehicle loses control due to a bump in the road.*

3.2.2 Scenario II – Follow Leading Vehicle

Description: there are two variations of this scenario: dynamic obstacle in front of the leading vehicle and no obstacle. In both cases, the ego-vehicle must follow the leading vehicle and act accordingly. The leading vehicle will drive until reaching the next intersection. The scenario will end when the ego-vehicle is close to the other actor or via timeout. The leading vehicle can be configured to be a car, motorcycle, or bicycle. Figure 3.2 show the scenario.

Road geometry: approaching intersection.

Weather conditions: daylight, clear conditions.

Vehicle speed: variable.

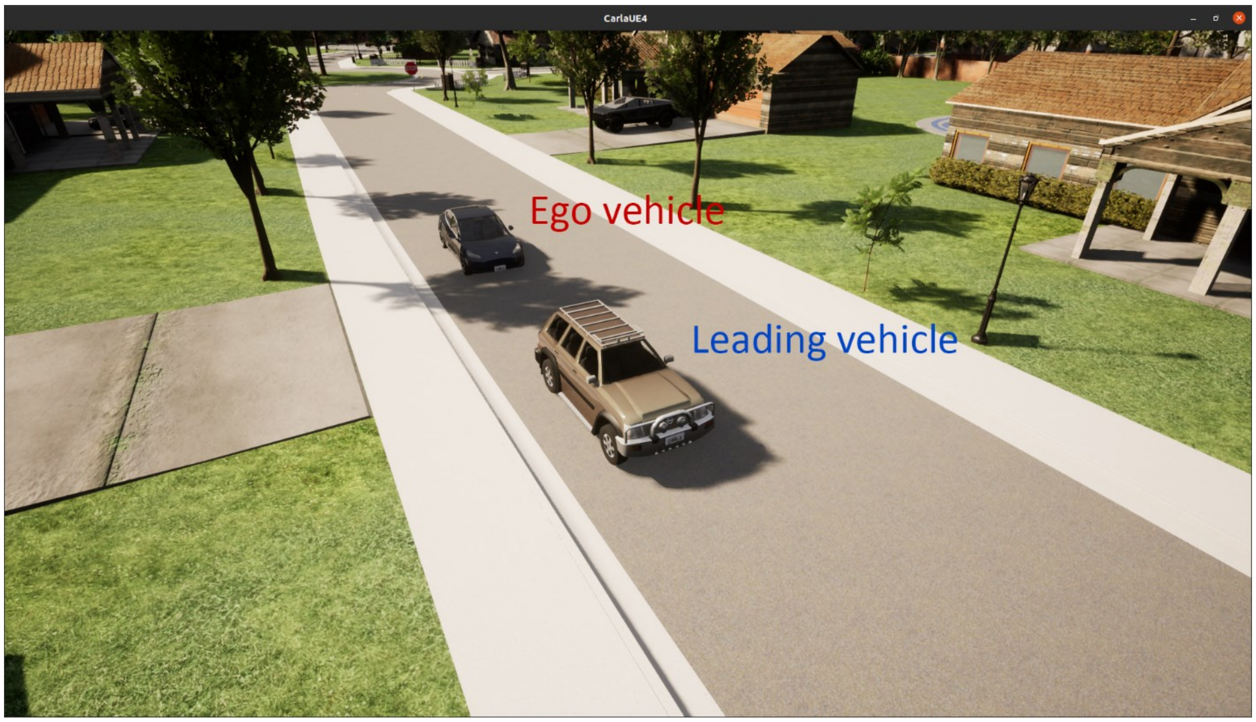


Figure 3.2: *FollowLeadingVehicle traffic scenario.*

3.2.3 Scenario III – Object Avoidance

Description: There are two variations of this scenario: static object and dynamic object. In both cases the ego-vehicle encounters an obstacle on the road and must act accordingly (e.g. avoidance maneuver). The execution of the scenario will finish either when the ego-vehicle reaches a target point or a timeout. Figure 3.3 shows the ego vehicle and a static object.

Road geometry: urban area (approaching intersection), rural area (straight-road).

Weather conditions: variable.

Vehicle speed: variable.

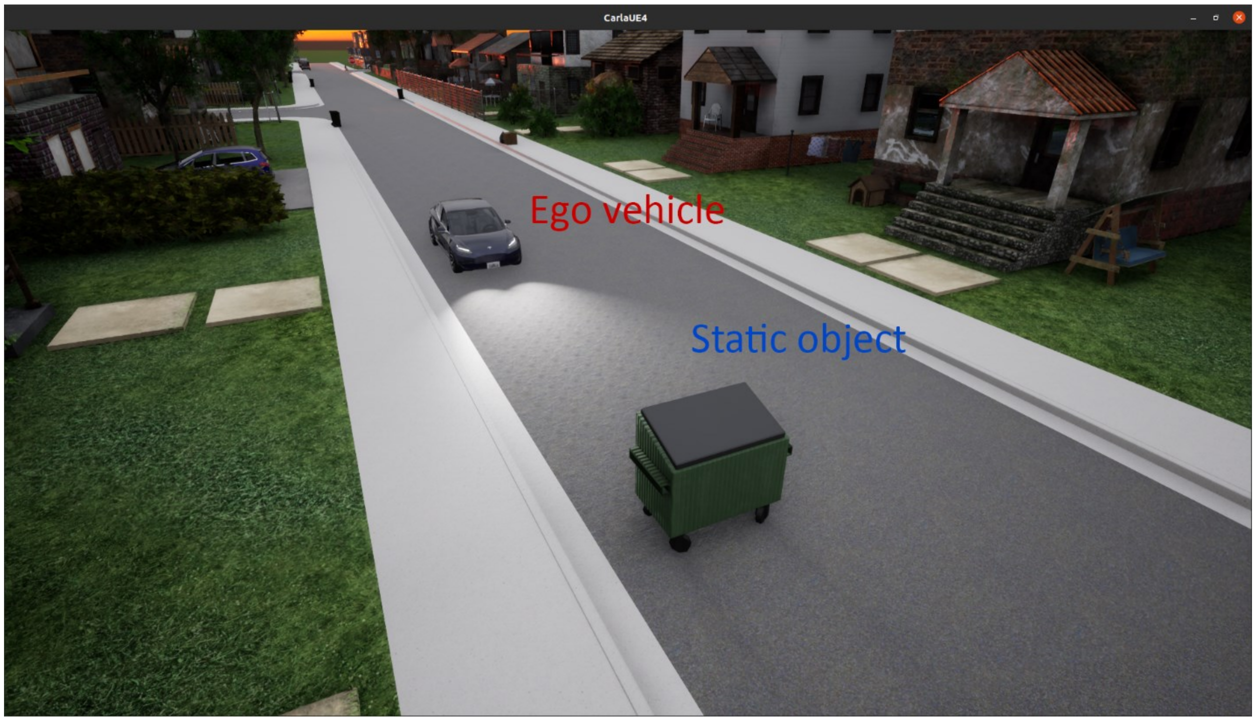


Figure 3.3: *ObstacleAvoidance traffic scenario.*

3.2.4 Scenario IV – Unsignalized Intersection

Description: The ego-vehicle reaches an unsignalized intersection, stops, and must negotiate with other vehicles to cross it. Both actors are spawned when the scenario starts. The ego-vehicle must reach a certain region to trigger the other vehicle to start moving. The other vehicle will start to accelerate and try to meet the ego-vehicle at a certain point. When the ego-vehicle reaches another trigger region, the other vehicle will accelerate and continue its path. Finally, the ego-vehicle must negotiate and pass the intersection. Figure 3.4 show an instance of this scenario.

Road geometry: rural, low-speed road.

Weather conditions: daylight, clear conditions.

Vehicle speed: variable.

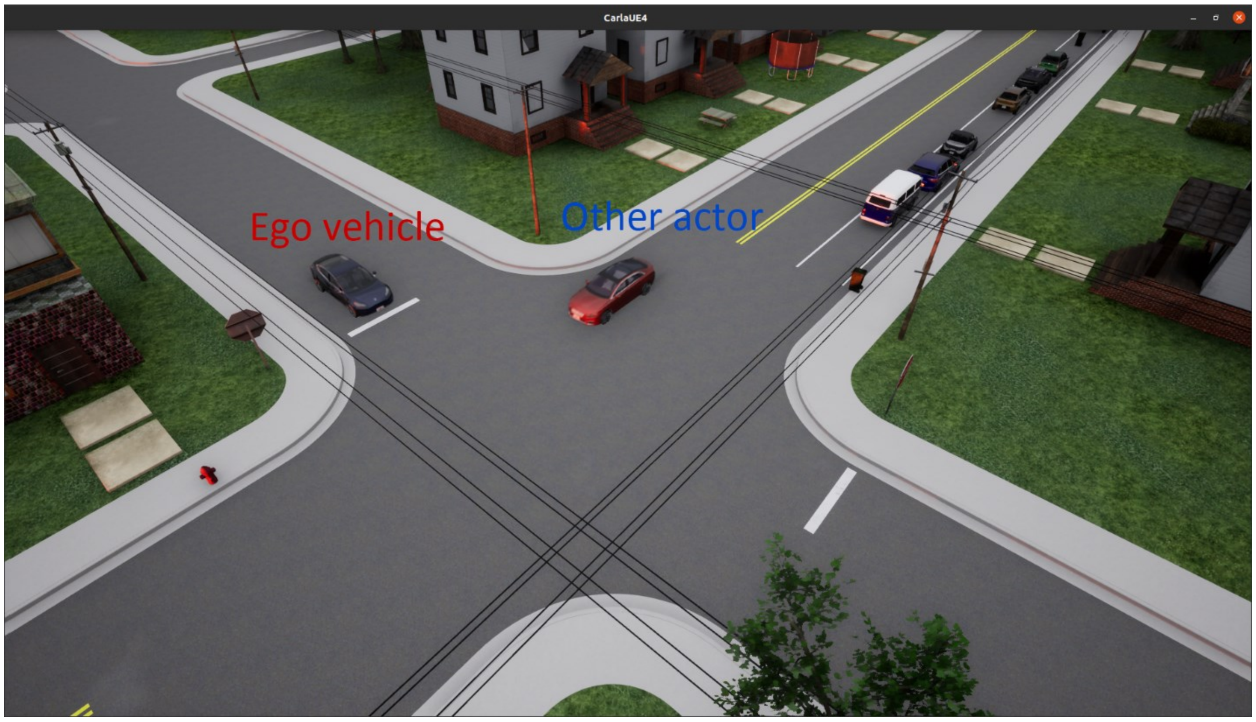


Figure 3.4: *UnsignalizedIntersection traffic scenario.*

3.2.5 Scenario V – Roundabout

Description: The ego-vehicle enters a roundabout and must go out through a specific exit while acting accordingly with other cars on it. The scenario execution is finished when the ego-vehicle reaches a target point or via timeout. Figure 3.5 show an instance of this scenario.

Road geometry: rural, low-speed road.

Weather conditions: variable.

Vehicle speed: around 10 mph or more.

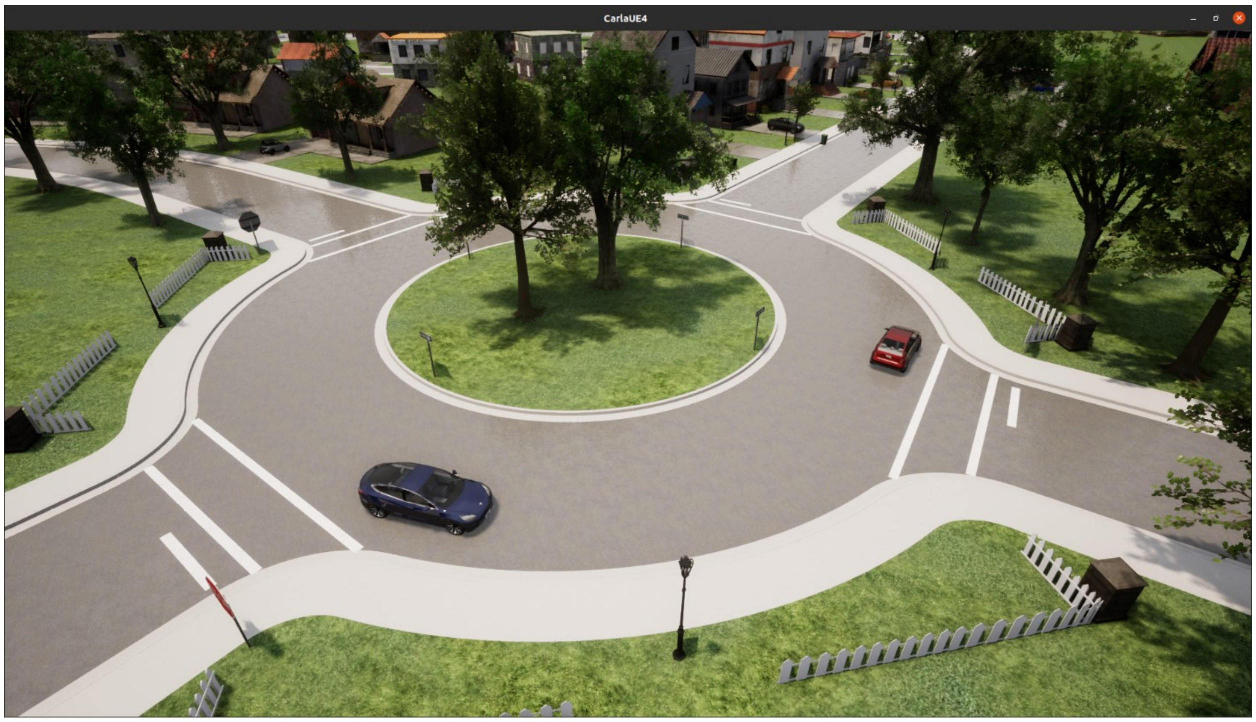


Figure 3.5: *Roundabout traffic scenario.*

Chapter 4: Benchmarks and Use Cases

Benchmarking an autonomous shuttle system allows companies and developers to test and validate agents in a structured way. The information gathered can be used to analyze performance of autonomous agents and find weakness or potential safety issues. This chapter will focus on different ways or tools that can be used to examine the algorithms in the Linden simulation environment.

4.1 Sensors

For vehicles to drive autonomously they must have a way to perceive what is around them. They do it by using a variety of sensors that allows them to "see" the environment. The most widely used autonomous vehicle sensors are camera, radar, and lidar, among others. This type of system enables cars to visualize the surroundings and detect speed and distance of nearby and distant objects [20]. CARLA already provides a variety of sensors that can be attached to a vehicle. This section discusses the most important sensors and how they work. The benchmarks will follow.

4.1.1 RGB Camera Sensor

As the name suggests, this sensor acts as a regular RGB (Red, Green, Blue) camera. Camera attributes included, but not limited to, are field of view, image size, camera sensitivity, aperture, focal distance, etc. Also, different post-processing effects can be applied such as vignette, grain jitter, lens flares, etc. Figure 4.1 and Table 4.1 show the camera output and output attributes of this sensor, respectively.

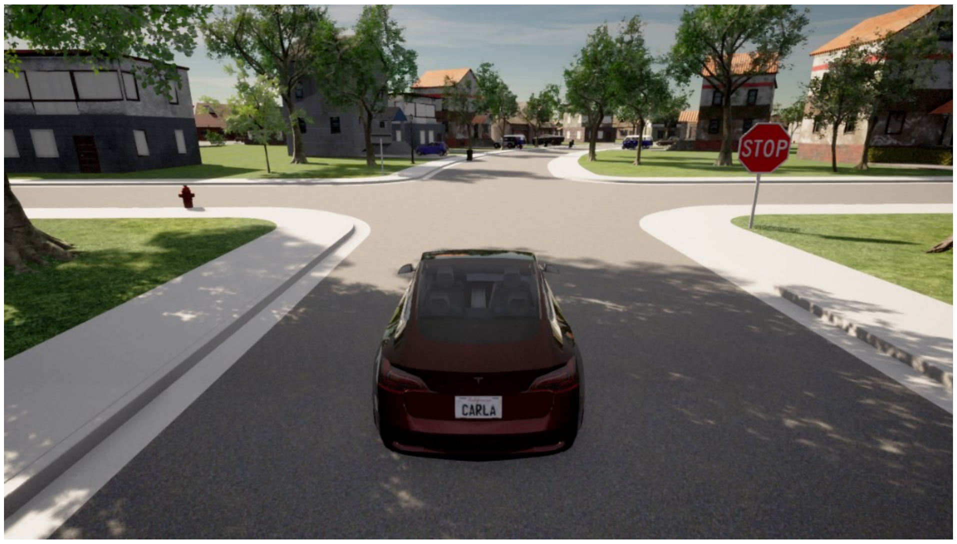


Figure 4.1: *RGB camera image.*

Sensor data attribute	Type	Description
frame	int	Frame number when the measurement took place.
timestamp	double	Simulation time of the measurement in seconds since the beginning of the episode.
transform	carla.Transform	Location and rotation in world coordinates of the sensor at the time of the measurement.
width	int	Image width in pixels.
height	int	Image height in pixels.
fov	float	Horizontal field of view in degrees.
raw_data	bytes	Array of BGRA 32-bit pixels.

Table 4.1: *RGB camera output attributes [15].*

4.1.2 Depth Camera Sensor

The depth camera has the same attributes as the RGB camera. However, this sensor captures the raw data from the scene or environment and codifies the distance of each pixel to create a depth map. CARLA provides color converters that let you convert the distance from RGB format to grayscale, distance codified in [0,1] float values. Figure 4.2 and Table 4.2 show the camera output and output attributes of this sensor, respectively.

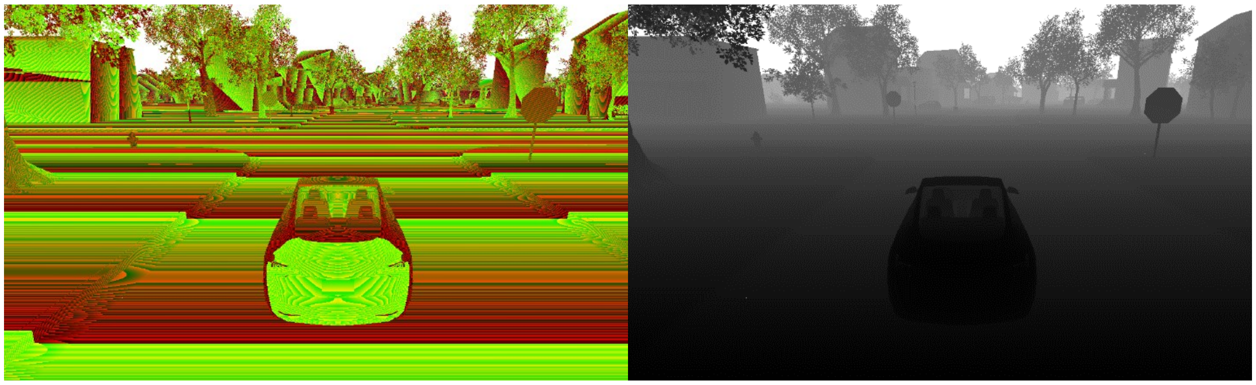


Figure 4.2: *Depth camera before and after conversion.*

Sensor data attribute	Type	Description
frame	int	Frame number when the measurement took place.
timestamp	double	Simulation time of the measurement in seconds since the beginning of the episode.
transform	carla.Transform	Location and rotation in world coordinates of the sensor at the time of the measurement.
width	int	Image width in pixels.
height	int	Image height in pixels.
fov	float	Horizontal field of view in degrees.
raw_data	bytes	Array of BGRA 32-bit pixels.

Table 4.2: *Depth camera output attributes [15].*

4.1.3 Semantic Segmentation Camera Sensor

The semantic segmentation camera sensor classifies every object according to predefined tags (e.g. vegetation, vehicles, etc.). The image is provided with the tag encoded in the red channel, that is, a pixel with a value *x* in the R channel belongs to a type of object with tag *x* . For example, vegetation would be encoded as (107, 142, 35). CARLA provides a simple way to use the CityScapes palette. Figure 4.3 and Table 4.3 show the camera output and output attributes of this sensor, respectively.

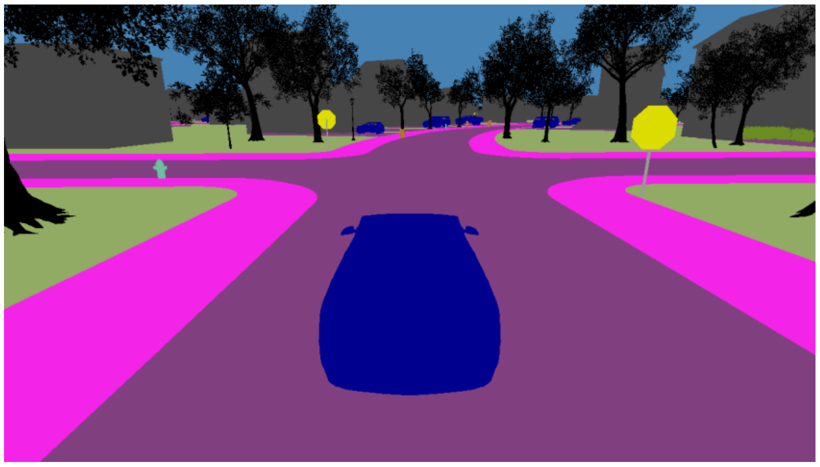


Figure 4.3: *Semantic segmentation camera output image.*

Sensor data attribute	Type	Description
fov	float	Horizontal field of view in degrees.
frame	int	Frame number when the measurement took place.
height	int	Image height in pixels.
raw_data	bytes	Array of BGRA 32-bit pixels.
timestamp	double	Simulation time of the measurement in seconds since the beginning of the episode.
transform	carla.Transform	Location and rotation in world coordinates of the sensor at the time of the measurement.
width	int	Image width in pixels.

Table 4.3: *Semantic segmentation camera output attributes [15].*

4.1.4 GNSS Sensor

The Global Navigation Satellite System (GNSS) sensor provides the current latitude and longitude position by calculating the metric position of the vehicle with the initial geo reference location offered by the OpenDrive map. Basic attributes include noise bias and noise standard deviation of the latitude, longitude, and altitude. Table 4.4 show the output attributes of this sensor.

Sensor data attribute	Type	Description
frame	int	Frame number when the measurement took place.
timestamp	double	Simulation time of the measurement in seconds since the beginning of the episode.
transform	carla.Transform	Location and rotation in world coordinates of the sensor at the time of the measurement.
latitude	double	Latitude of the actor.
longitude	double	Longitude of the actor.
altitude	double	Altitude of the actor.

Table 4.4: *GNSS output attributes [15].*

4.1.5 IMU Sensor

The Inertial Measurement Unit (IMU) sensors provides data from the actor's accelerometer, gyroscope, and compass. Basic attributes are noise bias and noise standard deviation of the accelerometer and gyroscope. Table 4.5 show the output attributes of this sensor.

Sensor data attribute	Type	Description
frame	int	Frame number when the measurement took place.
timestamp	double	Simulation time of the measurement in seconds since the beginning of the episode.
transform	carla.Transform	Location and rotation in world coordinates of the sensor at the time of the measurement.
accelerometer	carla.Vector3D	Measures linear acceleration in m/s^2.
gyroscope	carla.Vector3D	Measures angular velocity in rad/sec.
compass	float	Orientation in radians. North is (0.0, -1.0, 0.0) in UE.

Table 4.5: *IMU output attributes [15].*

4.1.6 LIDAR Sensor

This sensor simulates and actual rotating LIDAR using ray-casting. Virtual light rays are casted from the sensor’s viewpoint to find points of collisions. Lasers are added on each cannel in the vertical Field of View (FOV). The rotation is equivalent to the horizontal angle that the sensor rotated in a frame. Each LIDAR measurement will contain all the points captured during a 1/FPS (Frame Per Seconds) interval. Attributes includes channels, range, points per second, rotation frequency, upper and lower FOV, and more. Table 4.6 show the output attributes of this sensor.

Sensor data attribute	Type	Description
frame	int	Frame number when the measurement took place.
timestamp	double	Simulation time of the measurement in seconds since the beginning of the episode.
transform	carla.Transform	Location and rotation in world coordinates of the sensor at the time of the measurement.
horizontal_angle	float	Angle (radians) in the XY plane of the LIDAR in the current frame.
channels	int	Number of channels (lasers) of the LIDAR.
get_point_count(channel)	int	Number of points per channel captured this frame.
raw_data	bytes	Array of 32-bits floats (XYZI of each point).

Table 4.6: *LIDAR output attributes [15].*

4.1.7 Radar Sensor

This sensor translates a 2D point map of elements from a conic view. The data provided contains the polar coordinates, distance, and velocity of each measure point. Table 4.7 and Table 4.8 show the output attributes of this sensor and the radar detection measurement, respectively.

Sensor data attribute	Type	Description
raw_data	carla.RadarDetection	The list of points detected.

Table 4.7: *Radar output attributes [15].*

RadarDetection attributes	Type	Description
altitude	float	Altitude angle in radians.
azimuth	float	Azimuth angle in radians.
depth	float	Distance in meters.
velocity	float	Velocity towards the sensor.

Table 4.8: *Radar detection attributes [15].*

4.1.8 Collision Sensor

This sensor is not a typical sensor. It registers an event whenever the actor collisions against anything in the world (e.g. cars, pedestrians, trees, etc.). Unlike the previous sensors, this one does not have any configurable attribute. Table 4.9 shows output data of this sensor. This sensor is useful for evaluation of AV driving algorithms.

Sensor data attribute	Type	Description
frame	int	Frame number when the measurement took place.
timestamp	double	Simulation time of the measurement in seconds since the beginning of the episode.
transform	carla.Transform	Location and rotation in world coordinates of the sensor at the time of the measurement.
actor	carla.Actor	Actor that measured the collision (sensor's parent).
other_actor	carla.Actor	Actor against whom the parent collided.
normal_impulse	carla.Vector3D	Normal impulse result of the collision.

Table 4.9: *Collision output attributes [15]*

4.1.9 Lane Invasion Sensor

This sensor, as the collision sensor, is not a typical sensor. It registers an event whenever an actor, in this case, ego vehicle, cross a lane marking. The OpenDrive map definition is used to determine if the actor invaded or crossed to another lane. The sensor will detect that an actor crossed a lane even if it is invisible (e.g. no lane marking). Like the collision sensors, this one does not have any configurable attribute too. Table 4.10 show the output attributes of this sensor. This sensor like the previous one is also useful for evaluating AV driving algorithms.

Sensor data attribute	Type	Description
frame	int	Frame number when the measurement took place.
timestamp	double	Simulation time of the measurement in seconds since the beginning of the episode.
transform	carla.Transform	Location and rotation in world coordinates of the sensor at the time of the measurement.
actor	carla.Actor	Vehicle that invaded another lane (parent actor).
crossed_lane_markings	list(carla.LaneMarking)	List of lane markings that have been crossed.

Table 4.10: *Lane invasion output attributes [15].*

4.2 CARLA Benchmark

Autonomous Driving (AD) agents in CARLA can be evaluated through the ScenarioRunner module that was briefly covered in Chapter 3. This section will discuss that module in-depth and how it can be used or adapted for the simulation environment built in this thesis. Note that this thesis will only focus on the Python interface since it was the method used to define the Linden scenarios.

4.2.1 Scenarios General Architecture

The general structure of a scenario is depicted in Figure 4.4. It is composed of actors, a scenario tree and other stuff needed for the simulation. The actors can range from ego-vehicle(s), pedestrians, other vehicles (e.g. pre-defined traffic, follow a leading car, etc.). The scenario tree is a behavior tree of type py_tress[21]. This type of tree is used for decision making engines for different fields. In the ScenarioRunner case, it is used to define different actions or behaviors, conditions, criteria, triggers, etc.

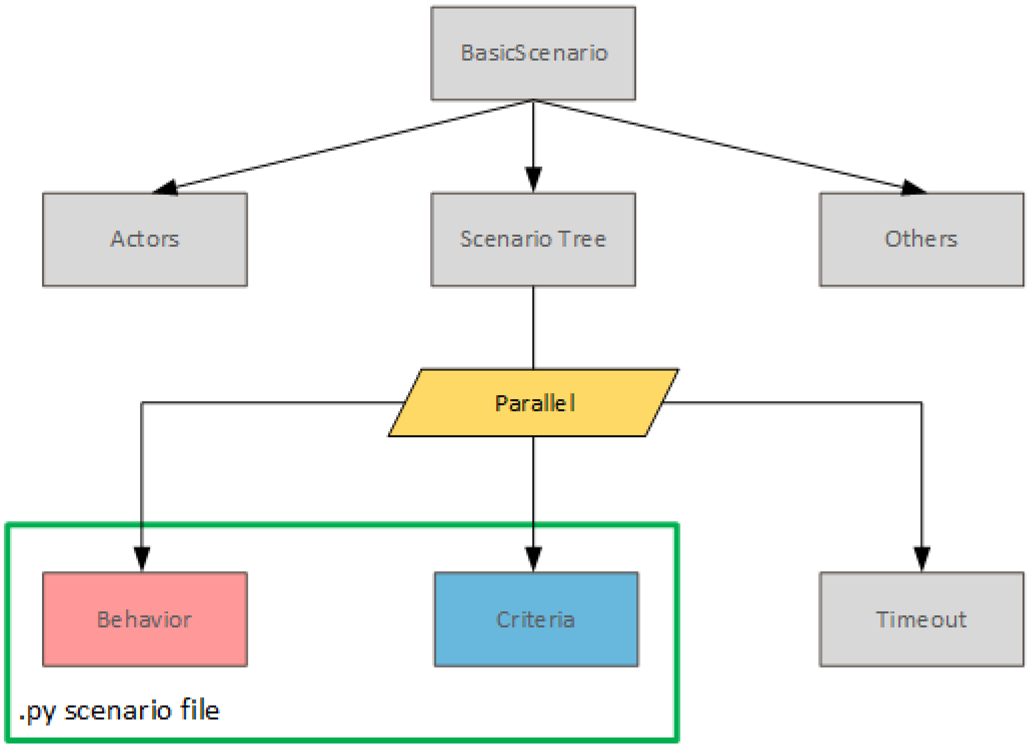
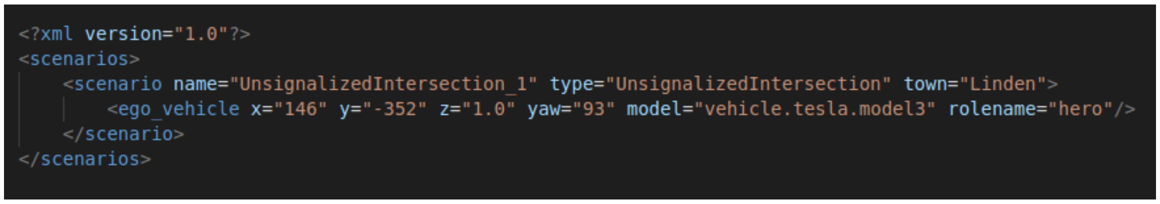


Figure 4.4: *General structure of a basic scenario, adapted from [22].*

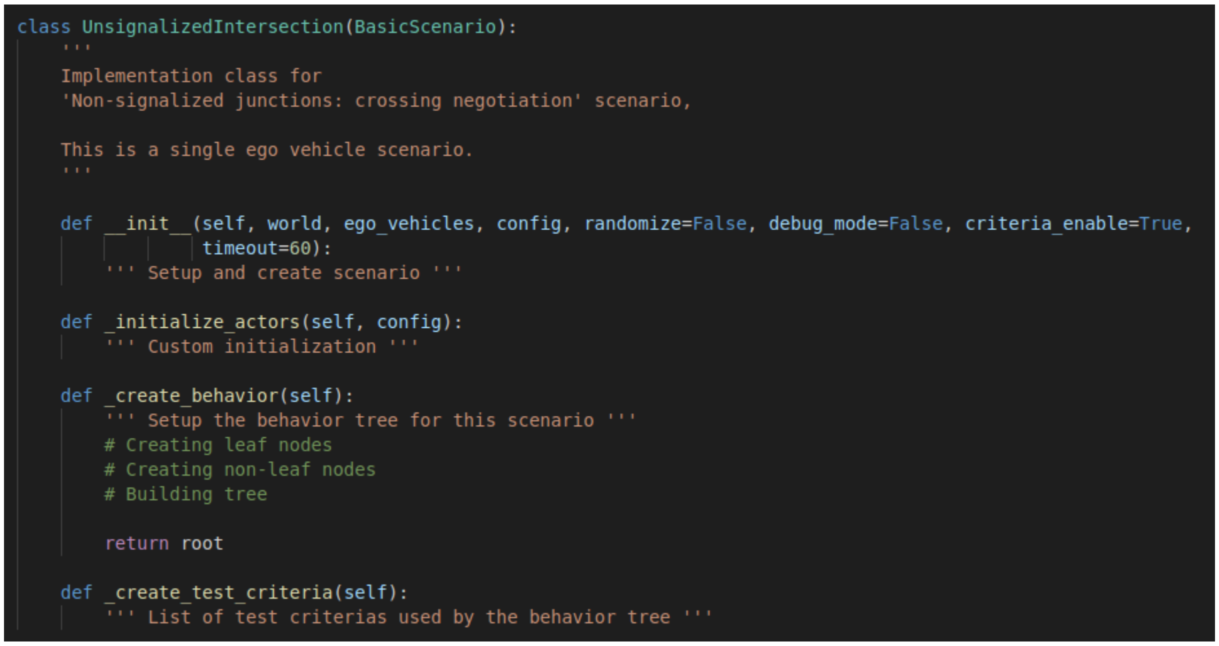
4.2.2 Scenario Definition and Implementation

Scenarios from CARLA's ScenarioRunner module have two main components: a scenario configuration of type .xml and an associated Python class. The .xml defines parameters such as initial ego vehicle location, orientation, type of vehicle, and other actors in the scenario. Actors refer to other vehicles, pedestrians, and objects. The associated Python class implements the behaviors and test criteria of the scenario. Behaviors refer to what happens when, where, and how. For example, two cars approaching the same intersection at the same time. Test criteria refer to pass or fail tests such as collisions, route completion, outside route lanes, actor speed above threshold, etc. Figures 4.5 and 4.6 depict examples of both components, that is, the .xml definition and the associated Python class, respectively.



```xml
<?xml version="1.0"?>
<scenarios>
    <scenario name="UnsignalizedIntersection_1" type="UnsignalizedIntersection" town="Linden">
        <ego_vehicle x="146" y="-352" z="1.0" yaw="93" model="vehicle.tesla.model3" rolename="hero"/>
    </scenario>
</scenarios>
```

Figure 4.5: *Associated .xml of the UnsignalizedIntersection scenario.*



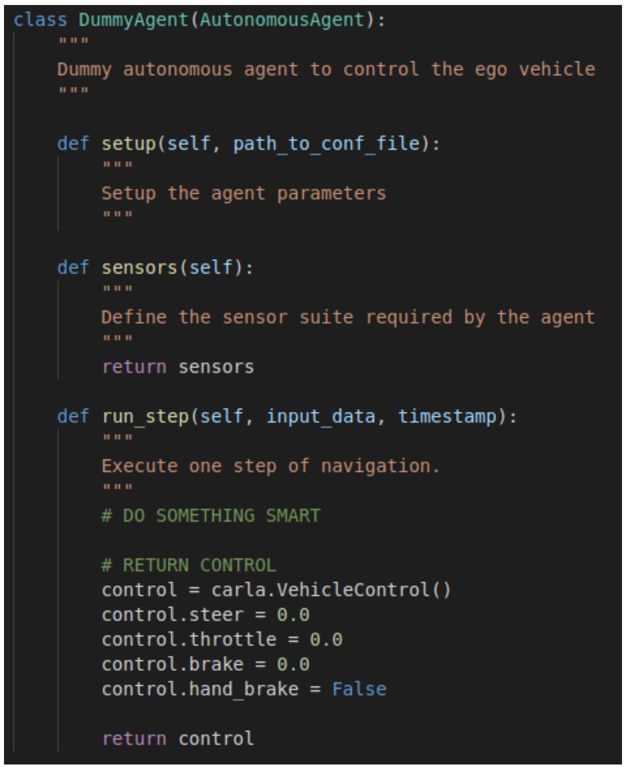
```python
class UnsignalizedIntersection(BasicScenario):
    '''
    Implementation class for
    'Non-signalized junctions: crossing negotiation' scenario,

    This is a single ego vehicle scenario.
    '''

    def __init__(self, world, ego_vehicles, config, randomize=False, debug_mode=False, criteria_enable=True,
                 timeout=60):
        ''' Setup and create scenario '''

    def _initialize_actors(self, config):
        ''' Custom initialization '''

    def _create_behavior(self):
        ''' Setup the behavior tree for this scenario '''
        # Creating leaf nodes
        # Creating non-leaf nodes
        # Building tree

        return root

    def _create_test_criteria(self):
        ''' List of test criterias used by the behavior tree '''
```

Figure 4.6: *Associated Python class of the UnsignalizedIntersection scenario.*

4.2.3 CARLA Agent Definition

The autonomous agents that control the ego vehicle are defined via a Python class. The agent class inherits from the AutonomousAgent base class and must overwrite three function. Figure 4.7 shows the basic structure of an agent. The functions to overwrite are the followings:

- **setup** – all necessary setup
- **sensors** – define the sensors configuration to be used by your agent.
- **run_step** – function that is executed every tick of the simulation for the agent to return a control command based on the provided data.



```python
class DummyAgent(AutonomousAgent):
    """
    Dummy autonomous agent to control the ego vehicle
    """

    def setup(self, path_to_conf_file):
        """
        Setup the agent parameters
        """

    def sensors(self):
        """
        Define the sensor suite required by the agent
        """
        return sensors

    def run_step(self, input_data, timestamp):
        """
        Execute one step of navigation.
        """
        # DO SOMETHING SMART

        # RETURN CONTROL
        control = carla.VehicleControl()
        control.steer = 0.0
        control.throttle = 0.0
        control.brake = 0.0
        control.hand_brake = False

        return control
```

Figure 4.7: *Basic structure of a ScenarioRunner's autonomous agent.*

4.2.4 Metrics Module

Apart from the scenarios test criteria, the traffic scenarios can be evaluated using the Metrics module provided by CARLA [23]. This module relies on the CARLA recorder which allows one to record and reenact any simulation. The saved data includes actor's creation and destruction, traffic lights states, vehicles states, pedestrian states, light states, etc. [24]. Here, vehicle and pedestrian states refer to position, orientation, and velocity. This module allows users to define their own metrics and obtain the results from the simulation files, instead of having to play the simulation repeatedly. Different queries can be made from the recording file such as get all collisions, velocities from a specific actor, applied controls, transforms, etc.

4.3 BASU Benchmark

BASU, *bus* in Japanese, is a custom benchmark tailored towards autonomous shuttles, specifically for the Linden route. It is a lightweight tool that simplifies the setup of simulation and usage of co-simulations like SUMO. This section will discuss the architecture and usage of the BASU benchmark.

4.3.1 BASU Architecture

BASU is composed of three main components: an agent, the managers, and utilities. The agent is the one to be evaluated by the benchmark and the managers take care of the simulation. Utilities are shared resources used internally. The main reason of using managers is to separate different aspect of the simulation. Figure 4.8 show BASU's architecture. The BaseManager is an abstract class that defines the basic structure of a manager. All the other managers inherit from this class as follows:

- **AgentsManager** – manages the agent to be evaluated in term of instantiation and configuration validation.
- **MetricsManager** – manages the output metrics of the simulation. It calculates the scores and output them into the console, text or json file.
- **RecordingManager** – manages the recording of the simulation, that is recording the events for future evaluation, playback, and analysis.
- **RoutesManager** – manages the parsing of the route to be followed, the criteria to be tested (e.g. check for collisions) and watch the status of the simulation (e.g. if vehicle reached the destination, timeouts, etc.).
- **SensorsManager** – manages all the sensors used by the agents. That is, the setup of the sensors, parsing the data from the sensors, and more.
- **TrafficManager** – manages everything related to the traffic. It oversees spawning the Non-controllable Players (NPC) either using CARLA's own TrafficManager module or through SUMO.

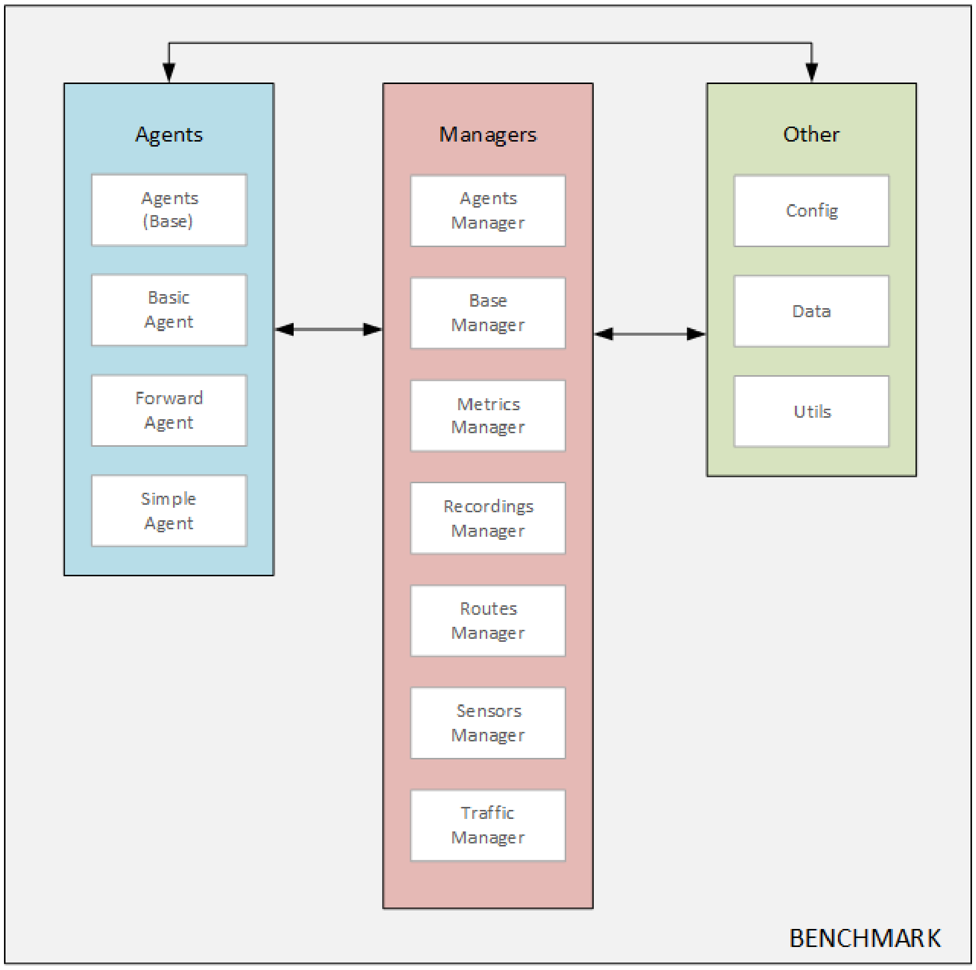
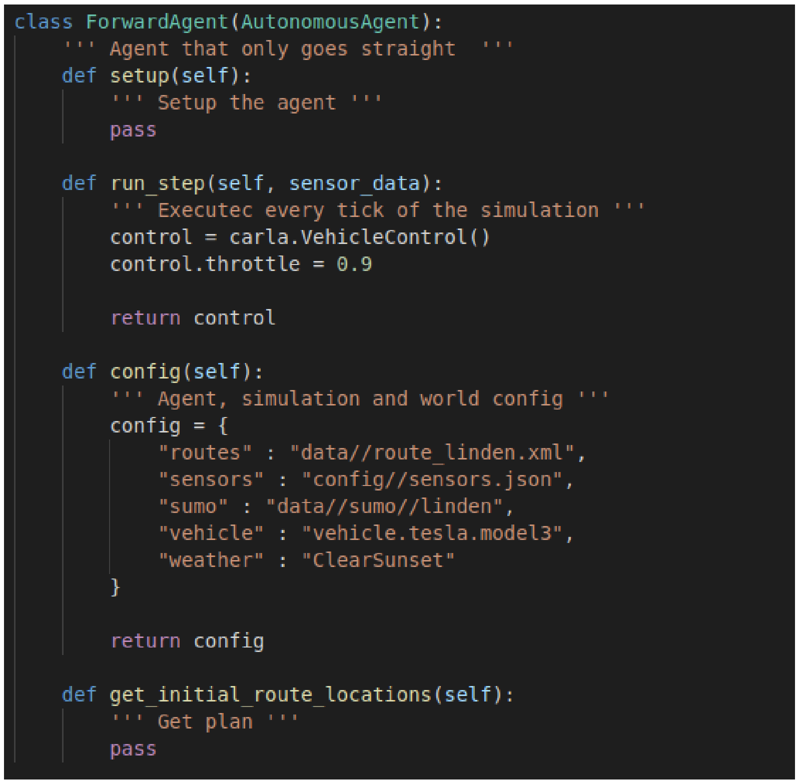


Figure 4.8: *BASU benchmark architecture.*

4.3.2 BASU Agent Definition

BASU's agents follows the same structure of a ScenarioRunner's agent with a few modifications. It was done this way to ease the implementation when switching between benchmarks, that is, CARLA's benchmark and BASU. Figure 4.9 show a sample agent class. There are four functions to be overwritten:

- **setup** – setup all necessary parts for the agent
- **run_step** – executed every step of the simulation.
- **config** – configure the simulation in terms of route to be tested, sensors, etc.
- **get_initial_route_locations** – returns the initial plan as a list of waypoints.



```python
class ForwardAgent(AutonomousAgent):
    ''' Agent that only goes straight  '''
    def setup(self):
        ''' Setup the agent '''
        pass

    def run_step(self, sensor_data):
        ''' Executec every tick of the simulation '''
        control = carla.VehicleControl()
        control.throttle = 0.9

        return control

    def config(self):
        ''' Agent, simulation and world config '''
        config = {
            "routes" : "data//route_linden.xml",
            "sensors" : "config//sensors.json",
            "sumo" : "data//sumo//linden",
            "vehicle" : "vehicle.tesla.model3",
            "weather" : "ClearSunset"
        }

        return config

    def get_initial_route_locations(self):
        ''' Get plan '''
        pass
```

Figure 4.9: *Basic structure of a BASU agent class.*

4.4 Co-simulations

Co-simulations allows the joint simulation between different standalone sub-simulators [25]. They are independent and behave like a black box [25]. It is assumed that each sub-simulator provides some sort of API that lets another program to control them. CARLA has already developed different co-simulations features with PTV-VISSIM, SUMO and Autoware. This section covers SUMO and Autoware co-simulations and how they can be used alongside the Linden environment. Section 4.4.1 introduces the of software that is used by Autoware.

4.4.1 Robot Operating System

The Robot Operating System (ROS) is an open-source framework initially targeted towards robotics development, but not limited to it. As the name stands, it is not an actual operating system (OS) but a collection of tools and libraries. ROS has many layers and functionalities such as hardware abstraction, communication over the network, inter-process communication, visualization and more [26]. AVs can take advantage of the ROS ecosystem by using the many open-source projects such as communications frameworks, algorithms such as perception and navigation, and tools for visualization. On that line, ROS can be used with CARLA through the *carla-ros-bridge* package [27]. This package brings functionalities such as sensors information exchange, AD agents, actors spawning, etc. For this thesis, this package was mainly used to record Linden's point cloud map.

4.4.2 Autoware

Autoware is not a co-simulation in the sense that was talked before because it is not an actual simulator. As discussed in section 2.4, Autoware is an open-source software used for self-driving cars. In this case, the Autoware agent is provided as a ROS package which allows it to be run on CARLA [28]. Figure 4.10 show a basic diagram of how Autoware interacts with CARLA through the *carla-ros-bridge*. To enable the agent to be used in Linden, the following must be done:

1) Modify Autoware's Docker, software platform that simplifies building and deploying software, to be able to inject external files.
2) Generate point cloud map, Lanelet2 (*.osm*) and Aisan (collection of *.csv*) autonomous driving maps, as seen in section 2.6.2 and 2.6.3.
3) Move the files to specific directories inside the container.

Figures 4.11 and 4.12 show Autoware's interface and the Autoware's agent spawned in the map created in this thesis, respectively.

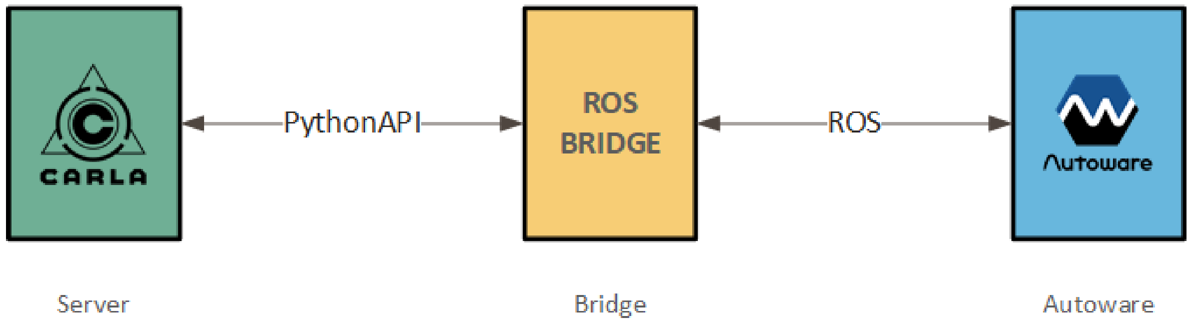
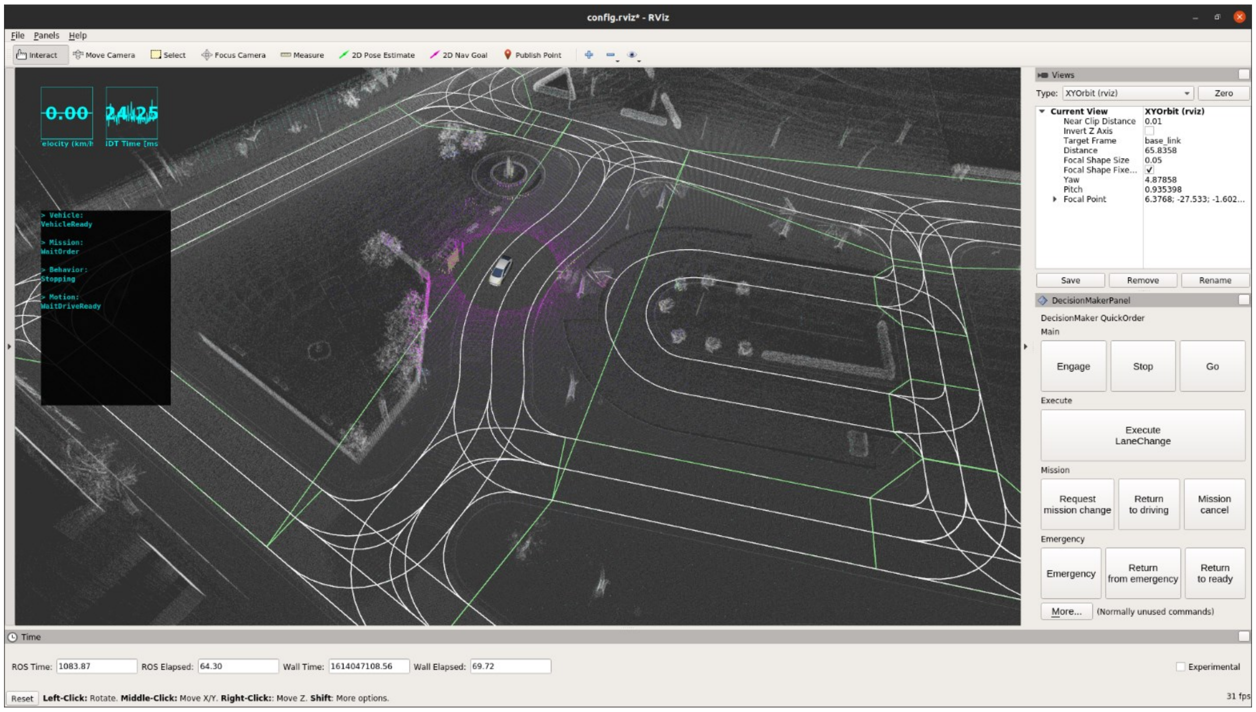


Figure 4.10: *Simplified diagram of carla-autoware bridge.*



```
> Vehicle:
VehicleReady

> Mission:
WaitOrder

> Behavior:
Stopping

> Motion:
WaitDriveReady
```

```
Type: XYOrbit (rviz)                    Zero
▾ Current View           XYOrbit (rviz)
    Near Clip Distance   0.01
    Invert Z Axis        ☐
    Target Frame         base_link
    Distance             65.8358
    Focal Shape Size     0.05
    Focal Shape Fixe...  ✓
    Yaw                  4.87858
    Pitch                0.935398
  ▸ Focal Point          6.3768; -27.533; -1.602...
```

```
ROS Time: 1083.87   ROS Elapsed: 64.30   Wall Time: 1614047108.56   Wall Elapsed: 69.72
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click:: Move Z. Shift: More options.
```

Figure 4.11: *Autoware's agent interface.*

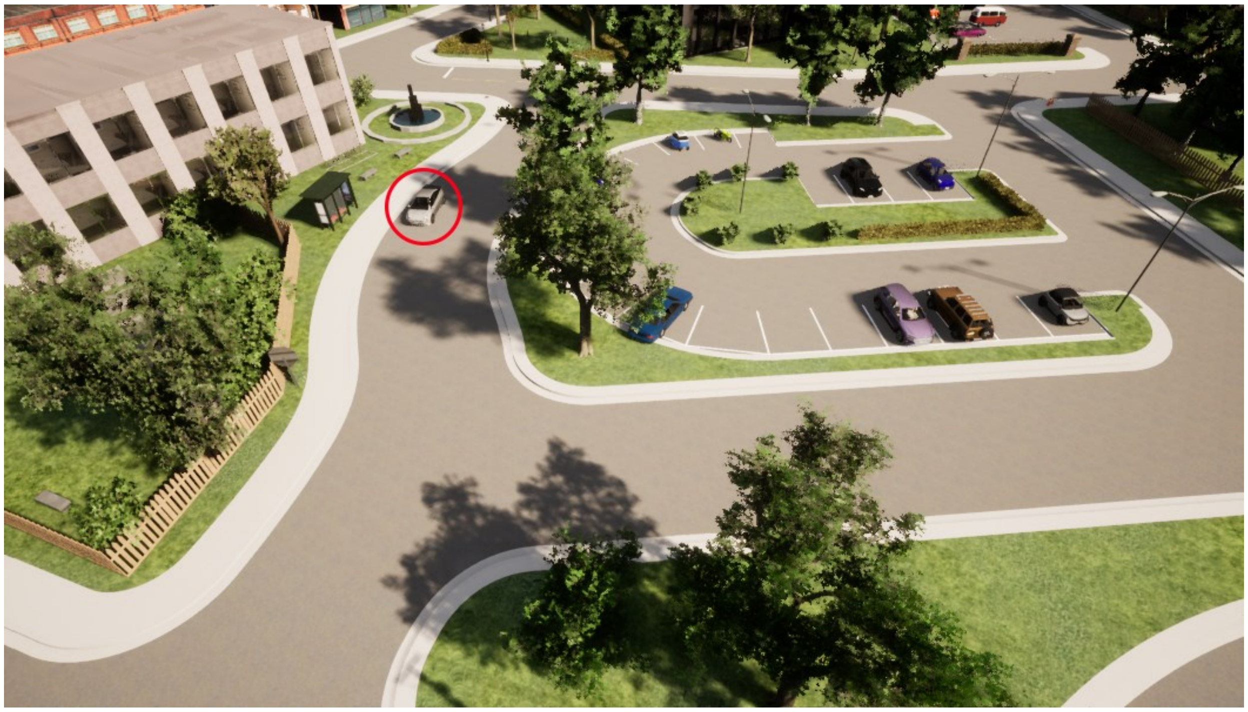
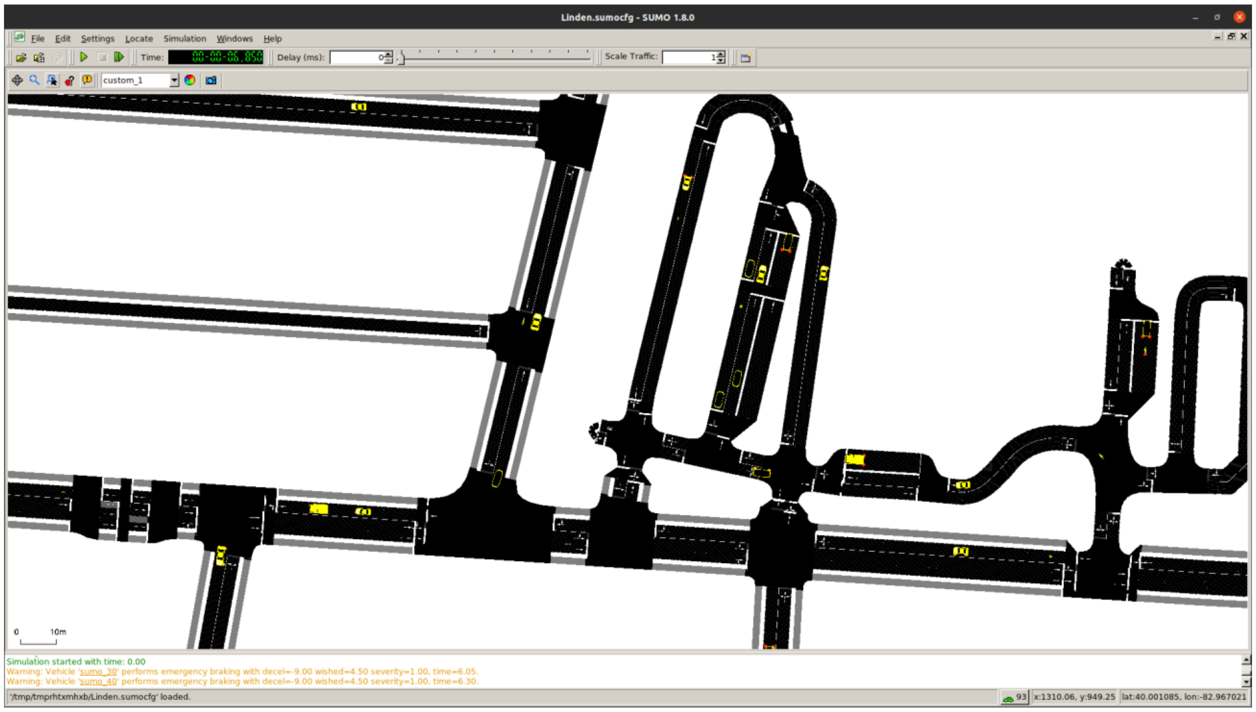


Figure 4.12: *Autoware's agent car in Linden's map.*

4.4.3 SUMO

Unlike Autoware, SUMO is an actual simulator. As discussed in section 2.5, SUMO is an open-source traffic simulator. It enables us to spawn and control NPCs in CARLA directly from SUMO. There are two ways to run the co-simulation: run a synchronization with a previously created simulation or spawn NPCs controlled by SUMO with random paths. The first one requires a configuration file *(.sumocfg*) that defines the simulation's network, routes, time and more. The second one can be run without a previous simulation, but the NPCs will follow random routes. The SUMO co-simulation can also be run through the BASU benchmark via arguments. Figures 4.13 and 4.14 show the SUMO co-simulation and the NPCs spawned in the simulator, respectively.


Linden.sumocfg - SUMO 1.8.0
File Edit Settings Locate Simulation Windows Help
Time:
Delay (ms):
Scale Traffic:
custom_1
0
10m

```
Simulation started with time: 0.00
Warning: Vehicle 'sumo_30' performs emergency braking with decel=-9.00 wished=4.50 severity=1.00, time=6.05.
Warning: Vehicle 'sumo_40' performs emergency braking with decel=-9.00 wished=4.50 severity=1.00, time=6.30.
'/tmp/tmprhtxmhxb/Linden.sumocfg' loaded.
93  x:1310.06, y:949.25  lat:40.001085, lon:-82.967021
```

Figure 4.13: *SUMO co-simulation.*

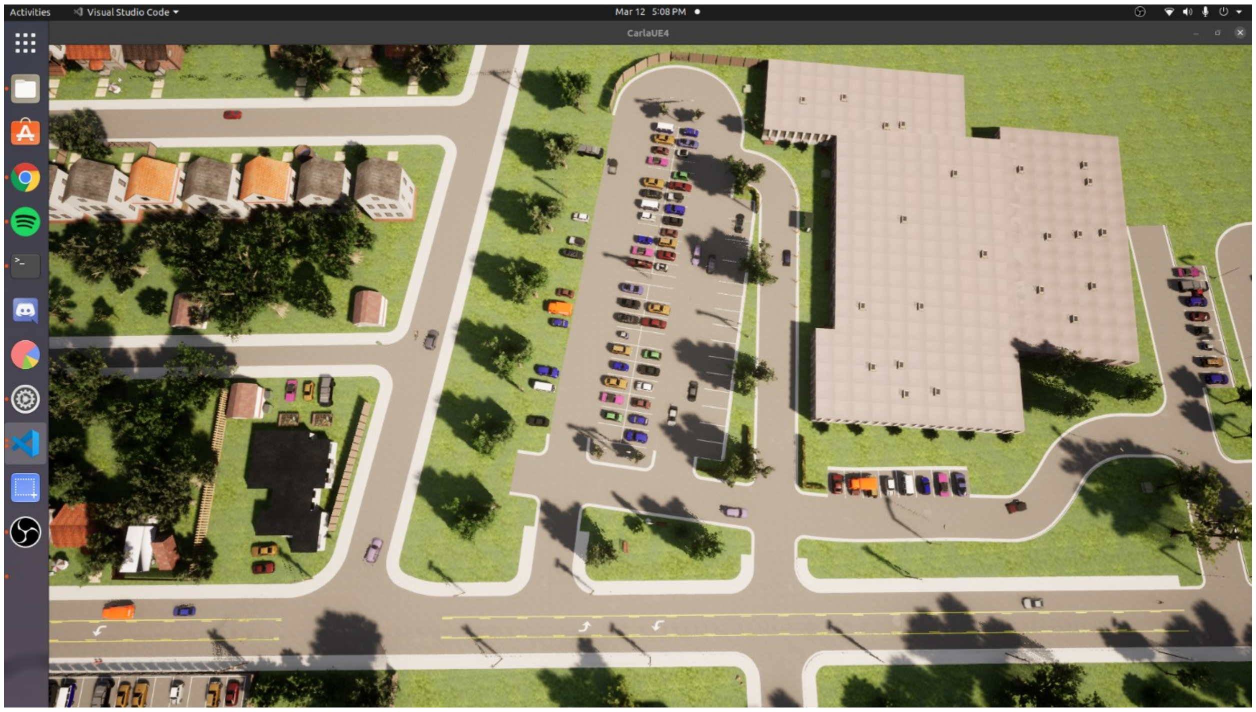


Figure 4.14: *Vehicles from SUMO spawned in the map.*

4.5 Resources Utilization

Running CARLA is a very computationally expensive process, especially when using multiple sensors and spawning a high number of NPCs. It gets even worse when using SUMO and/or Autoware at the same time. Mid to high end CPUs and GPUs are required to get a good number of Frame Per Seconds (FPS) and better performance. The overheads becomes more apparent when using larger maps like Linden. Recommended specs are Intel i7, i9, i10 9th-11th or AMD Ryzen 7, 9, +16 GB RAM memory, NVIDIA RTX 2070/2080 and RTX 30X0, Ubuntu 18.04.

Chapter 5: Evaluation and Metrics

Evaluating the proficiency of autonomous agents is the last step of the development cycle. They help to understand every aspect of the algorithms; to characterize them. This chapter will focus on the evaluations and metrics that can be obtained from the benchmarks and co-simulations in the Linden environment.

5.1 Evaluation

Assessing every aspect of driving is crucial for the safety of people. We want to save lives and prevent vehicle-related crashes. In autonomous driving, generally, we are interested in evaluating the autonomous agent and the impact they have in the transportation system. The first one is more related to the algorithms and performance of the agent. The second one focuses on the impact the agent has at a macro scale such as safety, mobility, environment, etc. In this thesis, we focus mainly on the agent's evaluation. However, the work can be extended to support the macro evaluation either by implementing the required metrics or via the already provided co-simulations.

5.2 Metrics

This section will cover the individual metrics that are used to compute the driving score of an autonomous agent for a simulation.

5.2.1 Infractions

Following the CARLA leaderboard, infractions are divided in two categories: collisions and traffics infractions. Collisions are categorized into three sub-types: with pedestrians, with vehicles, and with static objects. Each of them adds a penalty to the total driving score. Traffic infractions are related to running a red light or stop sign, and speeding. In this thesis, we only focus on stops signs since there are no traffic lights in the environment. The penalty of each infraction is outlined below. Note that they can be adjusted based on the user needs.

- Collisions with pedestrians: 0.50
- Collisions with other vehicles: 0.60
- Collisions with static objects: 0.65
- Running a stop sign: 0.80
- Speeding: 0.80

5.2.2 Lane Invasion

Lane invasion accounts for the times a vehicle crosses a lane marking or invades another lane. Discrepancies may arise when crossing lanes that are not visible in the map but are available in the OpenDrive file, which is the case of Linden [29]. This metric is not considered in the calculation of the driving score since lane invasion can occur for reasons that do not necessarily says anything about the agent itself (e.g. crossing to another lane). Nevertheless, is included in the output report as an individual metric.

5.2.3 Route deviation

Route deviation considers the actual path of the vehicle versus the initial plan. This metric determines the percentage that the autonomous vehicles was outside the planned route. This is done by comparing the arc length of the initial planned path and the actual path the agent took. It is assumed that both paths have the same start and final coordinates. Evasive maneuvers can affect the route deviation score since the car is going “out of the way”. Therefore, this score should be thresholded to not affect or penalize the overall score if that type of event occurs. Figure 5.2 shows multiple examples of the initial plan versus the actual path that agents took.

5.2.4 Disengagements

Disengagement is the action when a human driver takes control over the autonomous vehicle. We can calculate the number of miles driven and the frequency at which the disengagements occurs. For example, if a company traveled 628,839 miles across all its deployed vehicles and the total disengagements number was 21, then the miles per disengagement (mpd) would be 29,9444.69 mpd. To perform a disengagement, the user must manually disengage the autopilot (e.g. agent) using the HUD. This metric can be misleading if not used properly, but that is out of the scope of this thesis.

5.3 Driving Score

Taking the metrics discussed in section 5.2 into account, we can obtain an overall score to understand different aspects of the agents. The main score is an aggregate of all the infractions committed (e.g. running a stop, collisions, etc.) and path following completeness. The infraction penalty is a geometric series with a base score of 1.0 and it is affected by a penalty coefficient for every occurrence. The coefficient will vary depending on the type of infraction. Equation (5.1) shows the infraction penalty formula and is based upon the infraction penalty equation of CARLA's leaderboard challenge [30].

$$infraction\ penalty\ score = \prod_{j}^{infraction\ type} (p_i)^{\#infractions_j} \tag{5.1}$$

The path following score is based on the closeness of the agent's actual path with respect to the initial plan. The first step is to calculate the arc length of each path. The arc length is the distance between two points along a curve. If we divide the whole path or curve into smaller parts, we can approximate the corresponding lengths and add them up each segment. Equation (5.2) and (5.3) show the arc length formula and the path following score formula for the path y=y(x), respectively [31]. In Equation (5.3), L_A is the length of the actual path and L_I is the length of the initial path.

$$L = \int_b^a \sqrt{1 + \left(\frac{dy}{dx}\right)^2}\, dx \tag{5.2}$$

$$path\ following\ score = \min\left(1.0, \frac{L_A}{L_I}\right) \tag{5.3}$$

Finally, to calculate the overall score of the simulation we make use of the weighted average. A weighted average is the average of a data set that gives more importance to a certain number than other. The calculation is equal to the sum of the product of the data x_i times the weight w_i divided by the sum of all the weights. Here, w are the weights and x are the infractions and path following scores. Equations 5.4 and 5.5 shows the formula and expanded formula of the weighted average method [32]. The sum of the weight must be equal to 1.

$$overall\ driving\ score = \bar{x} = \frac{\sum_{i=1}^{n} w_i \cdot x_i}{\sum_{i=1}^{n} w_i} * 100 \tag{5.4}$$

$$\bar{x} = \frac{w_1 x_1 + w_2 x_2 + \cdots + w_n x_n}{w_1 + w_2 + \cdots + w_n} * 100 \tag{5.5}$$

Note that other metrics (e.g. lane invasion) are reported but are not part of the overall score formula. The agents can get a score between 0 (bad) and 1.0 (good). The overall driving score should give developers an idea of how the agent is performing. However, all the metrics should be considered when assessing the safety and efficiency of the agents. For example, as mentioned before, lane invasion does not count toward the total score. Nevertheless, a high number of lane invasions may indicate an underlying problem of the agent.

5.4 BASU and Metrics Module

Apart from the driving score discussed in the previous section, we can make use of the Metrics module described in section 4.2.4 too. The users can enable the recording feature, via command line argument in BASU's benchmark and then use the Metrics module to get additional metrics of interest. For example, we can get the distance of the ego-vehicle to the center of a lane throughout the whole simulation. In case you want to query a metric with respect to another actor, you need the other actor's ID or role name beforehand. Figure 5.1 shows an example plot of the distance to the center lane of the ego-vehicle.

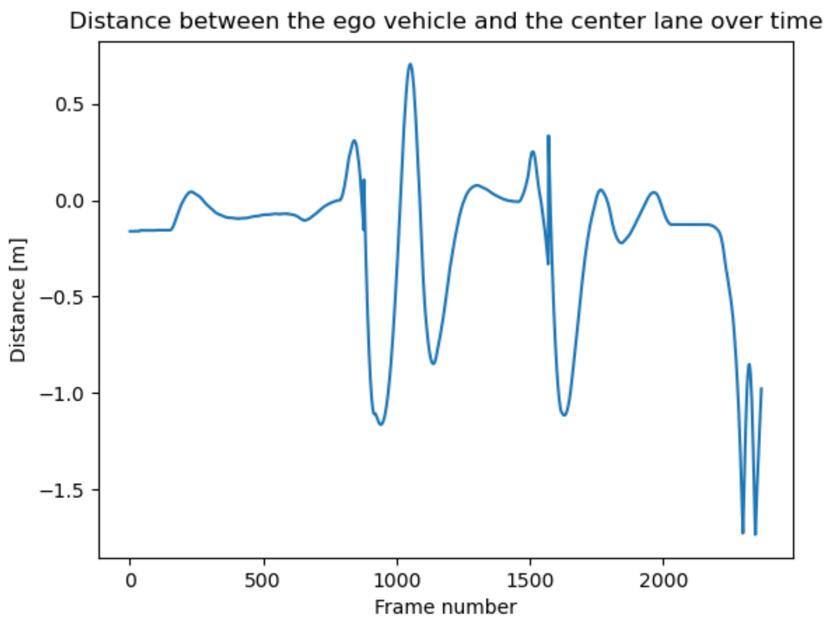


Figure 5.1: *Distance to center lane example*

5.5 Simulation Runs

Simulations were performed to showcase the metrics, driving score, and plots provided by BASU. The task was for agents to drive the predefined route depicted in Figure 5.2. This short route was chosen because it contains roundabouts, stop signs, and turns. Agents were tested in clear sunset weather conditions. Four autonomous agents were tested: basic, behavioral, forward, and human. The basic and behavioral agents are part of CARLA and were adapted to work in BASU. The forward agent was a dummy agent that only drove forward. The last agent was a human that controlled the ego-vehicle using the keyboard.

Before discussing the results, we must mention that running a stop sign was not considered in the driving score for this tests due to a bug in CARLA's agents that causes them to either stop permanently or not detect the stop sign at all. The weights used for these simulator runs were 0.8 and 0.2 for the penalty score and path following score, respectively. The basic agent was able to follow the initial plan successfully with no infractions and a driving score of 0.996. The behavioral agent was initially following the path successfully but, to show how each sub score affects the overall score, at some point we purposely deviated the car from the path and collided it with an object. The driving score for this agent was 0.216 because of that. The forward agent, as expected, did not follow the path quite well and eventually collided with an object, obtaining a driving score of 0.452. The human agent, like the basic agent, drove quite well and obtained a driving score of 0.604. Figure 5.2 shows plots of the agent's actual or followed path versus the initial plan. Table 5.1 shows more detailed metrics and scores.

From Table 5.1, we can observe how each metric affects the overall driving score. For example, the basic and human agents got approximately the same scores, but their driving scores differs by 0.236. If we observe their infraction score, we can see that the human got penalized for the occurrence of multiple speeding infractions. This allows you to see what aspect of the agent needs refining.

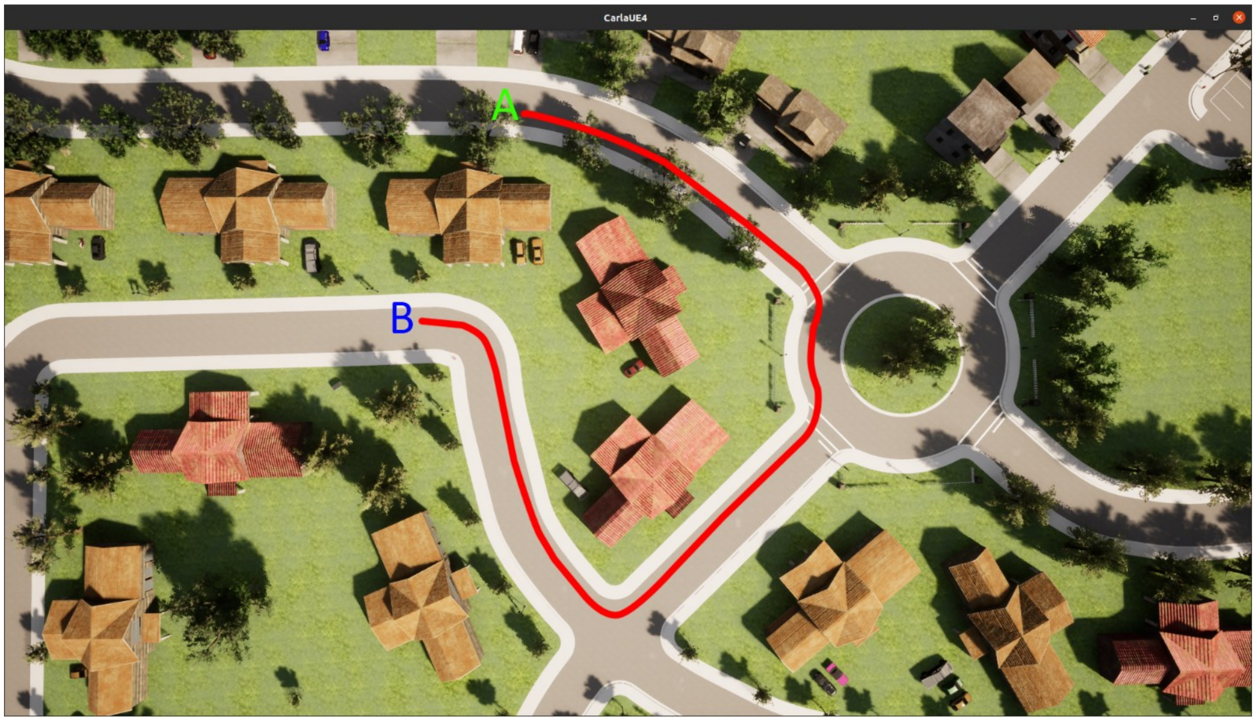


Figure 5.2: *Test route. Agents drove from point A to point B.*

			0 (BAD) - 1.0 (GOOD)		
Agent	Collisions	Speeding	Infraction Score	Path Following Score	Driving Score
Basic	0	0	1	0.98	0.996
Behavioral	1	1	0.13	0.56	0.216
Forward	1	1	0.52	0.18	0.452
Human	0	3	0.51	0.98	0.604

Table 5.1: *Simulations results with w_1 (infraction weight) = 0.8 and w_2 (path following weight = 0.2*

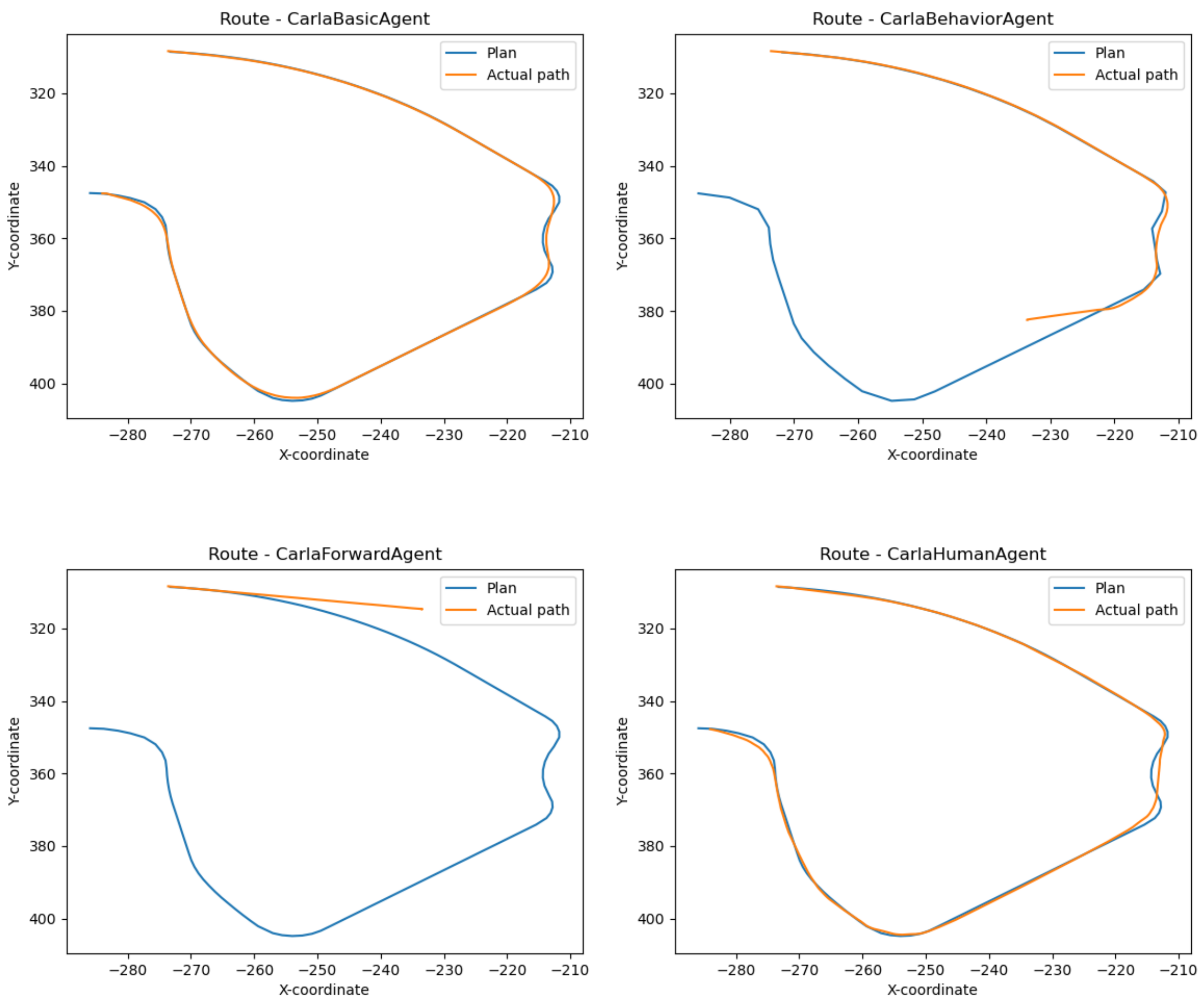


Figure 5.3: *Path following plots*

Chapter 6: Contributions and Future Work

This thesis covered every aspect of the process of building a simulation environment for simulating and testing self-driving vehicles from the tools that were used to build the map to how it can be used for different purposes relating autonomous vehicles. This chapter will talk about the overall contributions of this thesis and the future direction the work done here could go.

6.1 Contributions

First, this thesis created a simulation environment based on a real-life scenario. The map was created from scratch using RoadRunner for the road layout and populated with buildings, props, and vegetation in UE4/CARLA. To make the environment as close as possible to the original, Google Maps 3D and Street View features were used side by side with CARLA.

Second, this thesis provided a modified version of ScenarioRunner from CARLA and BASU benchmarking tool. The former enables the pre-defined traffic scenarios from ScenarioRunner to be used in Linden and define new ones (e.g. roundabout). The latter one is a custom benchmarking framework that allows developers to evaluate AD agent in pre-defined routes and provides different metrics and driving scores.

Finally, this thesis enables support for SUMO co-simulation and Autoware's open-source autonomous driving agent. SUMO can be co-simulated either directly through BASU or by itself. On the other hand, the necessary map files such as point cloud map, Lanelet and Aisan were generated to support Autoware's agent in Linden. Lastly, a detailed documentation of the project was created and is available in the project Github's repository.

6.2 Future Work

This project can be exploited and enhanced in many ways. Modelling new buildings that look more like the actual buildings found in the Linden area is one way to go. The ones placed in the environment are similar but not an exact copy. More complex traffic scenarios can be defined or implemented to increase the testing of the agents. In terms of BASU, more features can be added such as new evaluation metrics, cooperative driving support, and OpenScenario support.

6.3 Final Words

The work done is this thesis aims to provide a realistic environment from which developers and members from the ADL can benefit from. Maybe it will be used for safety assessment? Evaluate perception algorithms? Analyze traffic? Autonomous shuttles seems to be the future of our transportation system and this thesis seeks to be part of it.

Bibliography

[1] W. G. Najm, J. D. Smith, and M. Yanagisawa, "Pre-crash scenario typology for crash avoidance research," no. DOT-VNTSC-NHTSA-06-02, Apr. 2007, [Online]. Available: https://rosap.ntl.bts.gov/view/dot/6281.

[2] C. F. K. and J. Karsten, "Gauging investment in self-driving cars," *Brookings*, Oct. 16, 2017. https://www.brookings.edu/research/gauging-investment-in-self-driving-cars/ (accessed Feb. 28, 2021).

[3] NHTSA, "Automated Vehicles for Safety," *NHTSA*, Sep. 07, 2017. https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety (accessed Jan. 22, 2021).

[4] Smart Columbus, "Putting Safety Management Plan into Practice: Smart Columbus' Response to Linden LEAP Self-driving Shuttle Incident | SmartColumbus," *Smart Columbus*. https://smart.columbus.gov/playbook-asset/connected-electric-autonomous-self-driving-shuttle-incidentvehicles/smart-columbus-response-to-linden-leap- (accessed Jan. 22, 2021).

[5] E. D. Swanson, F. Foderaro, M. Yanagisawa, W. G. Najm, and P. Azeredo, "Statistics of Light-Vehicle Pre-Crash Scenarios Based on 2011–2015 National Crash Data," no. DOT HS 812 745, Aug. 2019, [Online]. Available: https://rosap.ntl.bts.gov/view/dot/41932.

[6] A. S. Mueller, J. B. Cicchino, and D. S. Zuby, "What humanlike errors do autonomous vehicles need to avoid to maximize safety?," *J. Safety Res.*, vol. 75, pp. 310–318, Dec. 2020, doi: 10.1016/j.jsr.2020.10.005.

[7] UE4, "Unreal Engine," *Unreal Engine*. https://www.unrealengine.com/en-US/ (accessed Mar. 02, 2021).

[8] MathWorks, "RoadRunner." https://www.mathworks.com/products/roadrunner.html (accessed Mar. 02, 2021).

[9] H. Darweesh, *hatem-darweesh/assuremappingtools*. 2021.

[10] Docker Team, "Docker." https://www.docker.com/ (accessed Apr. 22, 2021).

[11] Autoware Team, "Autoware-AI/autoware.ai: Open-source software for self-driving vehicles." https://github.com/Autoware-AI/autoware.ai (accessed Mar. 02, 2021).

[12] Eclipse Team, "Eclipse SUMO - Simulation of Urban MObility," *Eclipse SUMO - Simulation of Urban MObility*. https://www.eclipse.org/sumo/ (accessed Mar. 02, 2021).

[13] "USGS, advanced national map explorer interface." https://apps.nationalmap.gov/downloader/#/ (accessed Mar. 25, 2021).

[14] P. Bender, J. Ziegler, and C. Stiller, "Lanelets: Efficient map representation for autonomous driving," in *2014 IEEE Intelligent Vehicles Symposium Proceedings*, Jun. 2014, pp. 420–425, doi: 10.1109/IVS.2014.6856487.

[15] Autoware Team, "lanelet_aisan_converter," *GitLab*. https://gitlab.com/autowarefoundation/autoware.ai/utilities/-/tree/392a0465359d51f63fb877170a330912d71a0537/lanelet_aisan_converter (accessed Mar. 02, 2021).

[16] "ros-bridge/PclRecorder.cpp at master · carla-simulator/ros-bridge." https://github.com/carla-simulator/ros-bridge/blob/master/pcl_recorder/src/PclRecorder.cpp (accessed Mar. 31, 2021).

[17] "Point Cloud Library (PCL): pcl::PointCloud< PointT > Singleton Reference." https://pointclouds.org/documentation/singletonpcl_1_1_point_cloud.html (accessed Mar. 31, 2021).

[18] PCL Team, "Point Cloud Library," *Point Cloud Library*. https://pointcloudlibrary.github.io/ (accessed Apr. 22, 2021).

[19] CARLA Team, "Home - CARLA ScenarioRunner." https://carla-scenariorunner.readthedocs.io/en/latest/ (accessed Mar. 04, 2021).

[20] Katie Burke, "How Does a Self-Driving Car See? | NVIDIA Blog," *The Official NVIDIA Blog*, Apr. 16, 2019. https://blogs.nvidia.com/blog/2019/04/15/how-does-a-self-driving-car-see/ (accessed Mar. 11, 2021).

[21] "Py Trees — py_trees 2.1.4 documentation." https://py-trees.readthedocs.io/en/devel/ (accessed Mar. 23, 2021).

[22] "Introducing ScenarioRunner," *Google Docs*. https://drive.google.com/file/u/1/d/1zgoH_kLOfIw117FJGm2IVZZAIRw9U2Q0/view?usp=sharing&usp=embed_facebook (accessed Mar. 23, 2021).

[23] "Metrics module - CARLA ScenarioRunner." https://carla-scenariorunner.readthedocs.io/en/latest/metrics_module/ (accessed Mar. 25, 2021).

[24] "Recorder - CARLA Simulator." https://carla.readthedocs.io/en/latest/adv_recorder/ (accessed Mar. 25, 2021).

[25] OSP, "Co-simulation," *Open Simulation Platform*. https://opensimulationplatform.com/co-simulation/ (accessed Mar. 05, 2021).

[26] Adnan Ademovic, "An Introduction to Robot OS," *Toptal Engineering Blog*. https://www.toptal.com/robotics/introduction-to-robot-operating-system (accessed Mar. 01, 2021).

[27] CARLA Team, *carla-simulator/ros-bridge*. CARLA, 2021.

[28] CARLA Team, *carla-simulator/carla-autoware*. CARLA, 2021.

[29] CARLA Team, "Sensors reference - CARLA Simulator." https://carla.readthedocs.io/en/latest/ref_sensors/ (accessed Mar. 11, 2021).

[30] "CARLA Autonomous Driving Leaderboard." https://leaderboard.carla.org/ (accessed Mar. 31, 2021).

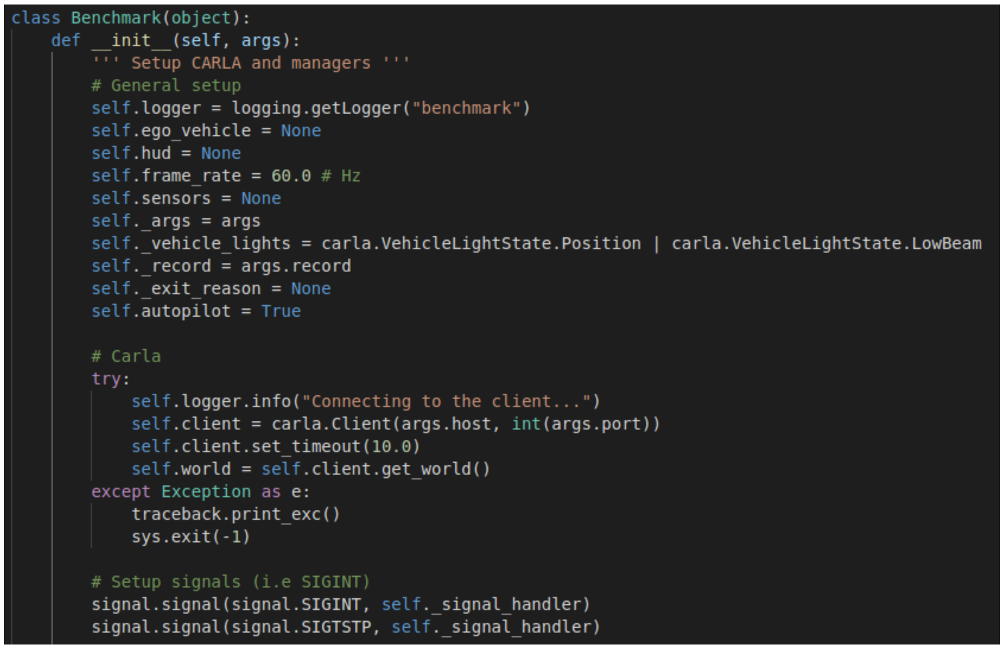
[31] E. W. Weisstein, "Arc Length." https://mathworld.wolfram.com/ArcLength.html (accessed Mar. 31, 2021).

[32] "GNU Scientific Library — GSL 2.6 documentation." https://www.gnu.org/software/gsl/doc/html/ (accessed Apr. 05, 2021).

Appendix A: Source Codes

A.1 BASU

BASU source code is available upon request.



```python
class Benchmark(object):
    def __init__(self, args):
        ''' Setup CARLA and managers '''
        # General setup
        self.logger = logging.getLogger("benchmark")
        self.ego_vehicle = None
        self.hud = None
        self.frame_rate = 60.0 # Hz
        self.sensors = None
        self._args = args
        self._vehicle_lights = carla.VehicleLightState.Position | carla.VehicleLightState.LowBeam
        self._record = args.record
        self._exit_reason = None
        self.autopilot = True

        # Carla
        try:
            self.logger.info("Connecting to the client...")
            self.client = carla.Client(args.host, int(args.port))
            self.client.set_timeout(10.0)
            self.world = self.client.get_world()
        except Exception as e:
            traceback.print_exc()
            sys.exit(-1)

        # Setup signals (i.e SIGINT)
        signal.signal(signal.SIGINT, self._signal_handler)
        signal.signal(signal.SIGTSTP, self._signal_handler)
```

Figure A.1: *Main file snapshot*

A.2 ScenarioRunner

The modified version of ScenarioRunner is available upon request.

Appendix B: Maps

B.1 RoadRunner

RoadRunner map files are available upon request.

B.2 Linden's CARLA Map Package

Linden's UE4/CARLA map is available upon request.

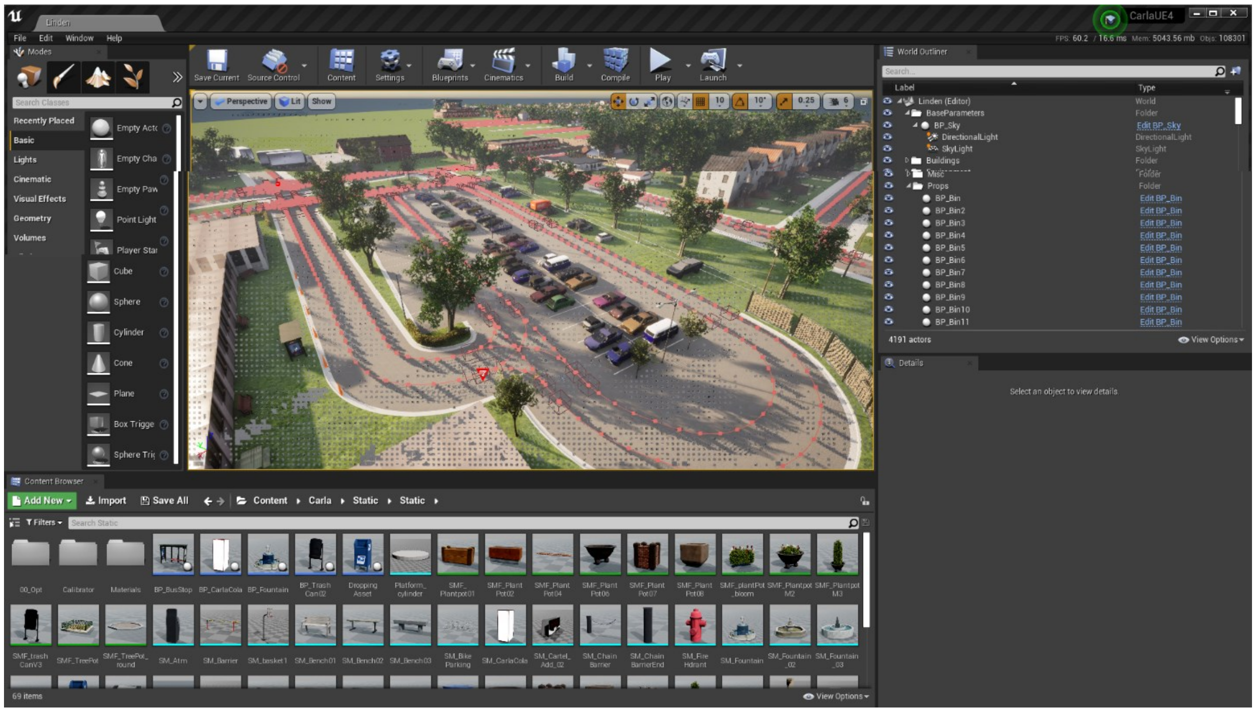
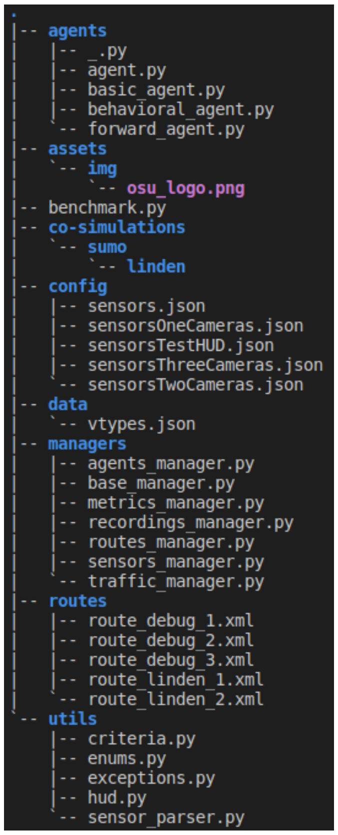


Figure B.1: *UE4/CARLA editor, build from source.*

Appendix C: BASU

C.1 Directory Structure

Figure C.1 shows the file structure of BASU. The *agents* folder contains sample agents that make use of CARLA's autonomous agents. The *co-simulations* folder contains files necessary for the SUMO co-simulation. The *config* folder contains sample sensors configurations. The *data* folder contains specific vehicle types needed for the SUMO co-simulation. The *managers* folders contains all the managers of the benchmark. The *routes* folder contains debugging routes and Linden's shuttle main route. Finally, the *utils* folder contains different scripts used on different part of the benchmark.

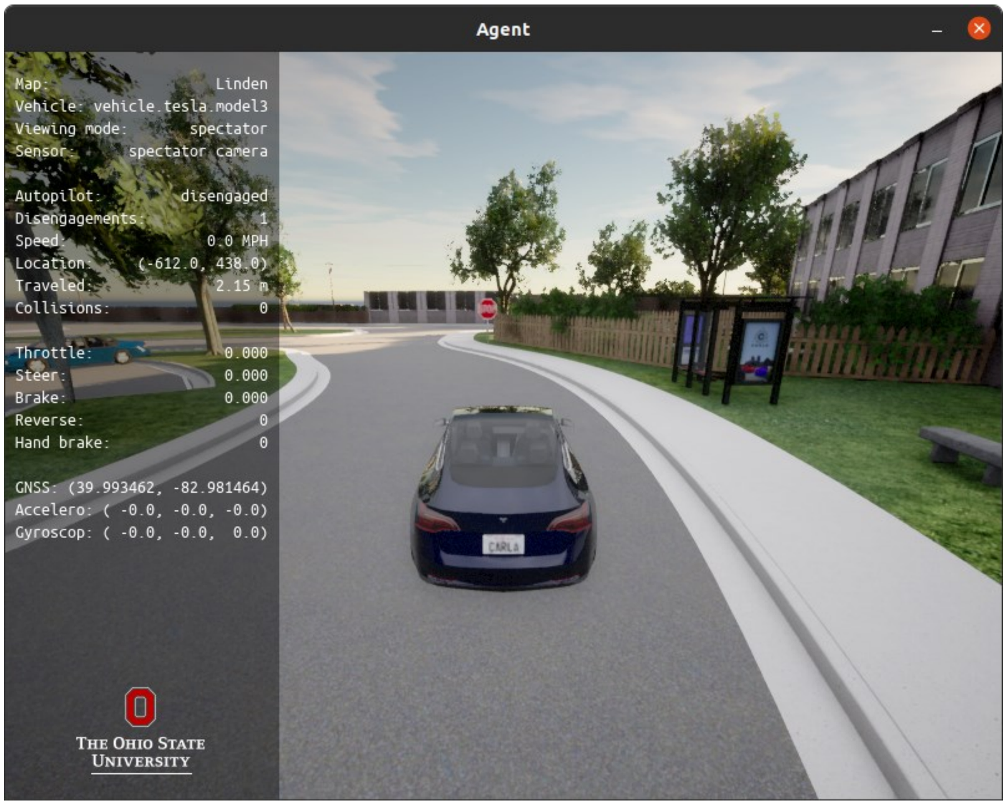


```
.
|-- agents
|   |-- _.py
|   |-- agent.py
|   |-- basic_agent.py
|   |-- behavioral_agent.py
|   `-- forward_agent.py
|-- assets
|   `-- img
|       `-- osu_logo.png
|-- benchmark.py
|-- co-simulations
|   `-- sumo
|       `-- linden
|-- config
|   |-- sensors.json
|   |-- sensorsOneCameras.json
|   |-- sensorsTestHUD.json
|   |-- sensorsThreeCameras.json
|   `-- sensorsTwoCameras.json
|-- data
|   `-- vtypes.json
|-- managers
|   |-- agents_manager.py
|   |-- base_manager.py
|   |-- metrics_manager.py
|   |-- recordings_manager.py
|   |-- routes_manager.py
|   |-- sensors_manager.py
|   `-- traffic_manager.py
|-- routes
|   |-- route_debug_1.xml
|   |-- route_debug_2.xml
|   |-- route_debug_3.xml
|   |-- route_linden_1.xml
|   `-- route_linden_2.xml
`-- utils
    |-- criteria.py
    |-- enums.py
    |-- exceptions.py
    |-- hud.py
    `-- sensor_parser.py
```

Figure C.1: *BASU file structure*

C.2 Heads Up Display (HUD)

Simulations in BASU can be visualizing by enabling the --*hud* flag. The HUD includes information related to the simulation, agent settings, and sensors readings. Figure C.2 depicts how the HUD looks like and the information it provides.



```
Map:                      Linden
Vehicle: vehicle.tesla.model3
Viewing mode:          spectator
Sensor:         spectator camera

Autopilot:            disengaged
Disengagements:                1
Speed:                   0.0 MPH
Location:       (-612.0, 438.0)
Traveled:                 2.15 m
Collisions:                    0

Throttle:                  0.000
Steer:                     0.000
Brake:                     0.000
Reverse:                       0
Hand brake:                    0

GNSS: (39.993462, -82.981464)
Accelero: ( -0.0, -0.0, -0.0)
Gyroscop: ( -0.0, -0.0,  0.0)
```

Figure C.2: *BASU HUD*