Bachelor Thesis

Martin Gochevski

Simulation of an autonomous vehicle in real, complex, urban driving scenes in Hamburg including its infrastructure and a local dynamic map

Martin Gochevski

Simulation of an autonomous vehicle in real, complex, urban driving scenes in Hamburg including its infrastructure and a local dynamic map

Bachelor Thesis eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Information Engineering
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betruender Prüfer: Prof. Dr. Rasmus Rettig
Zweitgutachter: Prof. Dr. Marc Hensel

Abgabedatum: 20.12.2022

Zusammenfassung

Martin Gochevski

Thema der Arbeit

Simulation of an autonomous vehicle in real, complex, urban driving scenes in Hamburg including its infrastructure and a local dynamic map

Stichworte

CARLA, HiL, Simulation, LDM, Testing, Urban Mobility

Kurzzusammenfassung

In diesem Dokument wird erklärt, wie man eine CARLA-Simulationsumgebung einrichtet, um einen realen Ort zu repräsentieren, in diesem Fall die Teststrecke für das automatisierte und vernetzte Fahren (TAVF), um eine einfachere standortbezogene Forschung zu ermöglichen, die unter ressourcenknappen Bedingungen durchgeführt werden kann. Die Simulationsumgebung umfasst ein Fahrzeugmodell, das dem Tesla Model S des Urban Mobility Lab nachempfunden ist und zusätzliche Messmöglichkeiten bietet. Darüber hinaus wird in diesem Beitrag demonstriert, wie eine solche Simulation genutzt werden kann, indem eine Kommunikation mit der Urban Dynamic Map des Cloud-Servers von European Digital Dynamic Mapping hergestellt wird, um - wenn auch derzeit noch begrenzt - Verkehrsobjekte anzuzeigen. Auch die Reproduzierbarkeit wird berücksichtigt, indem die in CARLA eingebaute Umgebungsaufzeichnung genutzt wird, mit der Möglichkeit, zusätzliche Zustände aufzuzeichnen, d.h. dynamische Objekte, die aus der Cloud abgerufen werden, und das Fahrzeug selbst.

Martin Gochevski

Title of the paper

Simulation of an autonomous vehicle in real, complex, urban driving scenes in Hamburg including its infrastructure and a local dynamic map

Keywords

CARLA, HiL, Simulation, LDM, Testing, Urban Mobility

Abstract

Within this document it is explained how to setup a CARLA simulation environment to represent a real-world location, in this case the Teststrecke für das automatisierte und vernetzte Fahren (TAVF), for the purpose of enabling easier location-based research which can be performed under resource scarce conditions. The simulation environment includes a vehicle model which mirrors the Urban Mobility Lab’s Tesla Model S with additional measurement capabilities. Additionally, this paper demonstrates how such a simulation can be used by establishing communication with the Urban Dynamic Map of the European Digital Dynamic Mapping’s cloud server to display, albeit currently limited, traffic objects. Reproducibility is considered as well by using CARLA’s built-in environment recording with the ability to record additional states i.e., dynamic objects retrieved from the cloud and the vehicle itself.

Acknowledgment

I would like to take this opportunity to thank my parents for the years of support of any kind throughout my studies as I could not have managed without them. Furthermore, I am extremely grateful to Prof. Dr. Rettig mentoring me and introducing me to the Urban Mobility Lab and their research efforts, it has been a great journey so far and I hope that it stays that way. Finally, I must acknowledge Prof. Dr. Hensel for his ability to inspire young students to do more and do better. One of the major highlights during my first semester at the HAW was his lecture.

Table of Contents

Acknowledgment 1

1 Introduction 4
1.1 Motivation 4
1.2 Objective 4
1.3 Structure 5

2 Terms and Concepts 6
2.1 TAVF 6
2.2 Hardware-In-The-Loop 6
2.3 CARLA 8
2.4 Local Dynamic Map 8
2.5 EDDY 9
2.6 Robot Operating System 9

3 State of the art 11

4 Requirements Analysis 13
4.1 System Expectations 13
4.2 EDDY-in-The-Loop 13

5 Hardware and Software 14
5.1 Hardware 14
5.2 Software 14
5.2.1 Requirements 14
5.2.2 Used 15
5.2.3 Circumvention 15

6 Implementation 16
6.1 System Concept 16
6.1.1 Communication with UDM 16
6.1.2 Simulated Test Vehicle 17
6.2 Simulation Setup 18
6.2.1 TAVF in CARLA 19
6.2.2 Test Vehicle 22
6.2.3 Path Planning and Route Following 24
6.3 ROS Communication 25

7 Functionality Test 26
7.1 Underlying Components 26
7.1.1 Vehicle controller test 26
7.1.2 Displaying objects received from UDM 27

8 Testing UDM with the Digital Twin 29

9 Conclusion 30

10 Future Work 31

11 Appendix 32

11.1 System Setup Manual 32
11.1.1 .... CARLA Installation32
11.1.2 .... Distributed Client34
11.2 Source Code 41
11.2.1 .... manager.py41
12.1.1 .... pid.py49
12.1.2 .... sensors.py60
12.1.3 .... simulation_setup.py62
12.1.4 .... conversion.py66
12.1.5 .... eddy_digital_twin.launch67

List of Figures 68

List of Tables70

List of Listings 71

Bibliography 72

1 Introduction

1.1 Motivation

Most automotive researchers and manufacturers are focused on the vision of future mobility. While it’s apparent that fully autonomous vehicles are the goal, defining how this will be achieved is a seemingly Sisyphean task that becomes more difficult with the introduction of every proprietary Autonomous Driving (AD) system [1]. This has been recognized and is being worked on by the European Digital Dynamic Mapping (EDDY) [2] project, its goal being the creation of a Local Dynamic Map (LDM) [3] implementation called the Urban Dynamic Map (UDM) for the city of Hamburg and establishing a framework for UDM implementation for other European cities. Such an implementation of an LDM will provide enhanced collective perception capabilities to automated vehicles which in turn should result in improved localization and protection of Vulnerable Road Users (VRUs) [4].

The UDM as conceptualized by EDDY will allow the vehicles of tomorrow to communicate in a standardized manner within urban environments with a focus on safer and more efficient mobility for all participants in traffic. AD vehicles will specifically benefit from openly available information on traffic conditions and obstructions and by integrating the collective perception offered by the UDM VRUs will be better protected with early recognition of potentially dangerous scenarios.

Testing the systems developed for the improvement of the UDM itself is crucial. However, running such tests using a physical vehicle is costly in terms of man-hours and energy and their adaptability is limited [5]. Complex systems for which physical tests can be potentially destructive, externally or to the system, have been tested in simulations for the past century [6]. Testing the UDM with an AD capable vehicle is currently bureaucratically prohibitive since authorization is required to drive a vehicle using its AD system. The process to acquire such authorization is time consuming and the authorization itself might be limited by the authorities based on road and traffic conditions [7]. Whereas a simulation can be run at any point in time for any length of time and requires no active monitoring, meaning new features of the UDM can be tested as soon as they are developed.

1.2 Objective

The fundamental goal of the thesis is to establish a digital twin [8] of the Teststrecke für Automatisiertes und Vernetztes Fahren (TAVF) [9] within a simulation environment, including a vehicle capable of automated driving, communication with the UDM, visual representation of UDM data and session recording. Furthermore, the aim is to use the digital twin to test the UDM’s performance to the extent possible as of writing this paper and allow for anyone reading to create tests suitable for future features of the UDM.

Testing the UDM as part of this study shall be rudimentary, as the UDM is in its early stages of development and the reliable data available is basic, mainly consisting of the location of traffic signs along the TAVF. Therefore, the criteria for the test will be reliability of the connection to the UDM and a comparison of the latency of a standalone request and the latency of a request as part of the digital twin.

1.3 Structure

In section 2 the terms and concepts necessary to understand the rest of the paper are introduced and explained. Section 3 presents a brief overview of the progress of Hardware-in-the-Loop (HiL) testing, how widely CARLA is used with and without HiL in research studies and some noted publications relevant to the thesis and EDDY. Section 4 describes what the system should do, how it should perform and what is its purpose. In section 5 a systemic overview of hardware and software requirements and how they were fulfilled is given. Section 6 shows the implementation of the system i.e., how the system expectations from 4.1 were accomplished. Sections 7 and 8 present test results, where the former revolves around the functionality of the system's components and it as a whole and the latter shows the use of the system coupled with the UDM. At the end section 9 discusses the system as a whole and how it can impact future mobility and section 10 proposes possibilities for improvement and additions to the system.

2 Terms and Concepts

2.1 TAVF

The TAVF in Hamburg is a track which serves the purpose of testing future mobility concepts. The track can be seen in Figure 1. It consists of more than 70 communication systems designed to exchange data with vehicles using it. The data is sent via ITS-G5 [10], a communication standard based on WLAN 802.11p. Currently provided are Signal Phase and Timing (SMaT) messages of the traffic lights, Map Data Messages (MAP) messages of the road topology, virtual signs as In-Vehicle Information (IVI) and sensor data for VRU protection as Collective Perception Messages (CPM) [11]. Researchers and manufacturers alike take advantage of the TAVF to develop and shape their concepts of what urban mobility will look like [12].

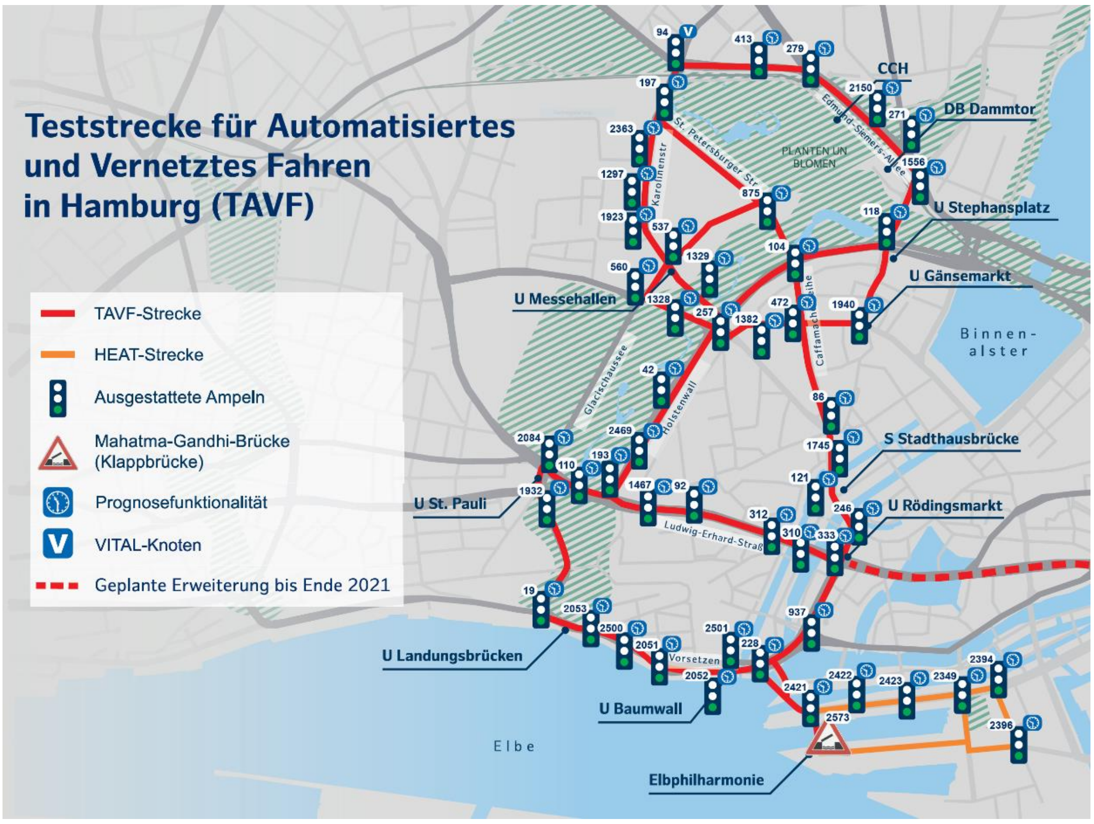


Figure 1: TAVF [13]

2.2 Hardware-In-The-Loop

Hardware-In-The-Loop (HiL) testing is a design approach based on using simulated process signals as input to a real controller [14]. A visual representation of a simulation substituting the physical process is shown in Figure 2. Simulating the process allows for the design of the application to increase in complexity while keeping testing costs in terms of time, money, and energy lower than building and installing the components for an experimental test. In the case of all simulations the trade-off for cost savings is accuracy [15]. A simulation does not account for all factors that influence the physical system as its purpose is to approximate the behavior of the process with a model accurately enough to ensure that the rest of the system operates as expected. Figure 3 presents the trade-off between costs and precision of testing methods, where accuracy reduction refers to the ability of the testing method to provide precise outputs.

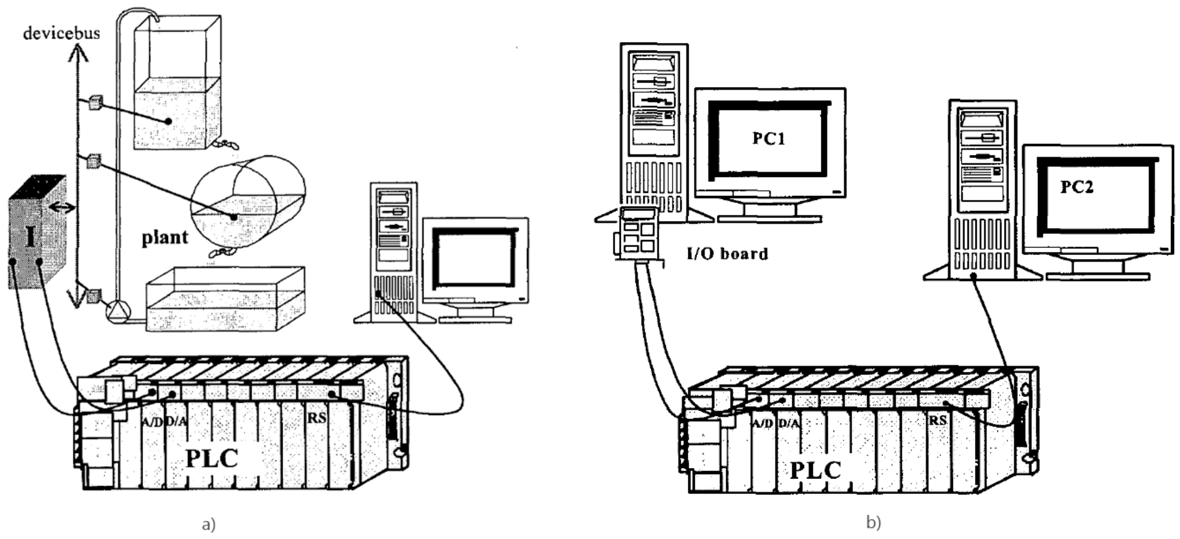


Figure 2: a) Classical control experimental setup; b) Hardware-in-the-loop method, adapted from [14]

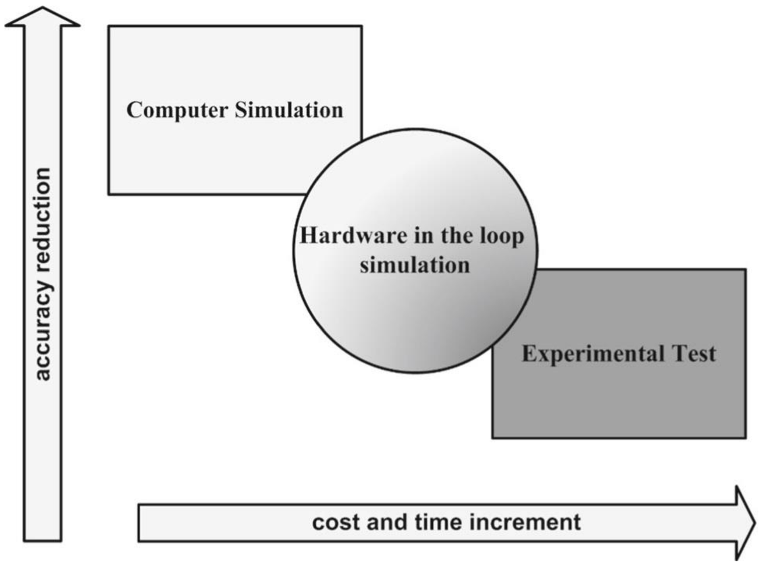


Figure 3: Precision, time, and cost trade off in computer simulation, HiL simulation and actual situation [15]

2.3 CARLA

CAR Learning to Act (CARLA) is an open-source research simulator [16] with continued regular maintenance and updates contributed by dozens more [17]. CARLA is implemented in C++ and provides a range of features and tools for simulating and testing autonomous driving systems, including:

- A realistic 3D urban environment with a variety of roads, intersections, buildings, and other objects
- A physics engine for simulating vehicle dynamics and sensor behavior
- A range of sensors, including cameras, lidar, radar, and GPS
- Tools for controlling and interacting with the environment, such as the ability to spawn and control vehicles, manipulate traffic conditions, and set up scenarios
- APIs for connecting to external autonomous driving algorithms and systems

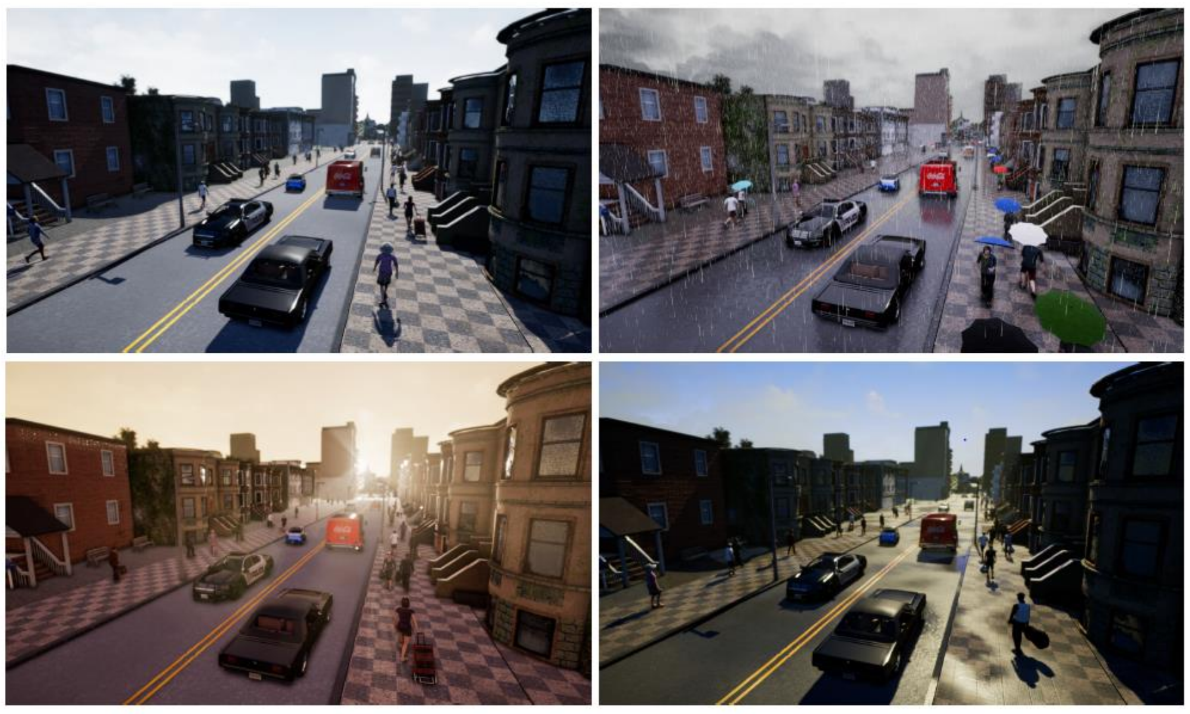


Figure 4: CARLA simulating an urban environment in different weather conditions [16]

2.4 Local Dynamic Map

A Local Dynamic Map (LDM), standardized by ETSI [3], combines static and dynamic information relevant for Cooperative Intelligent Transport Systems (C-ITS) as four layers, as seen in Figure 3.

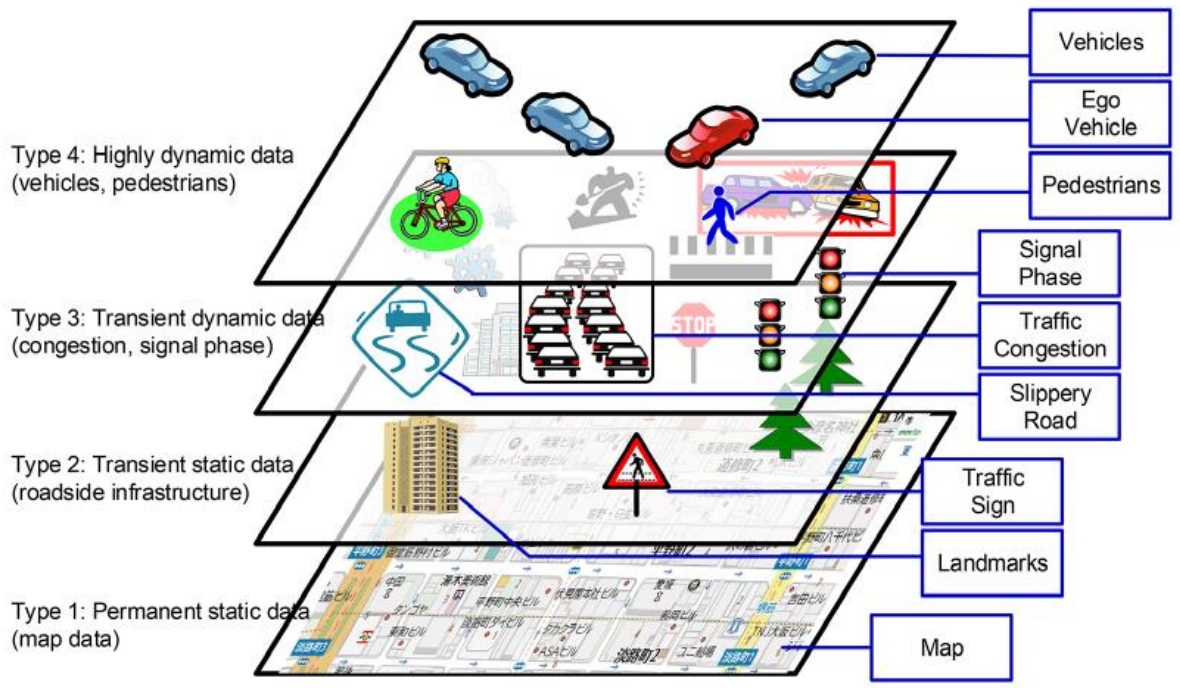

Type 4: Highly dynamic data (vehicles, pedestrians)
Type 3: Transient dynamic data (congestion, signal phase)
Type 2: Transient static data (roadside infrastructure)
Type 1: Permanent static data (map data)
Vehicles
Ego Vehicle
Pedestrians
Signal Phase
Traffic Congestion
Slippery Road
Traffic Sign
Landmarks
Map

Figure 5: LDM and its four layers [18]

2.5 EDDY

The European Digital Dynamic Mapping (EDDY) project is an on-going, publicly funded, collaborative effort from the German Center for Air and Space Travel [19] (DLR – Deutsches Zentrum für Luft- und Raumfahrt e. V), Urban Mobility Lab [20] (UML) of the University of Applied Sciences Hamburg (HAW – Hochschule für Angewandte Wissenschaften Hamburg), Institute for Climate Protection, Energy and Mobility [21] (IKEM - Institut für Klimaschutz, Energie und Mobilität), OECON Products & Services [22], consider it [23] and Ubilabs [24]. Its goal is to conceptualize and develop an Urban Dynamic Map, a cloud database implementation of an LDM which is open to contribution and use of data for ITS. This implementation is based in the city of Hamburg with the vision of creating a framework to be used by cities across Germany and Europe.

2.6 Robot Operating System

Robot Operating System (ROS) [25] is a flexible framework for developing robotic applications. It provides a set of tools and libraries for building and deploying robot software, as well as a communication infrastructure for connecting different parts of a robot system.

ROS is based on a publish-subscribe model, in which nodes (individual components of a robot system) publish data to and subscribe to data from a shared message-passing system. This allows nodes to communicate with each other and exchange information about their environment, sensors, and actions. An overview of a system called turtlesim, a simulation most commonly used as an introduction to ROS, using the publish-subscribe model to control a robot is given in Figure 6.

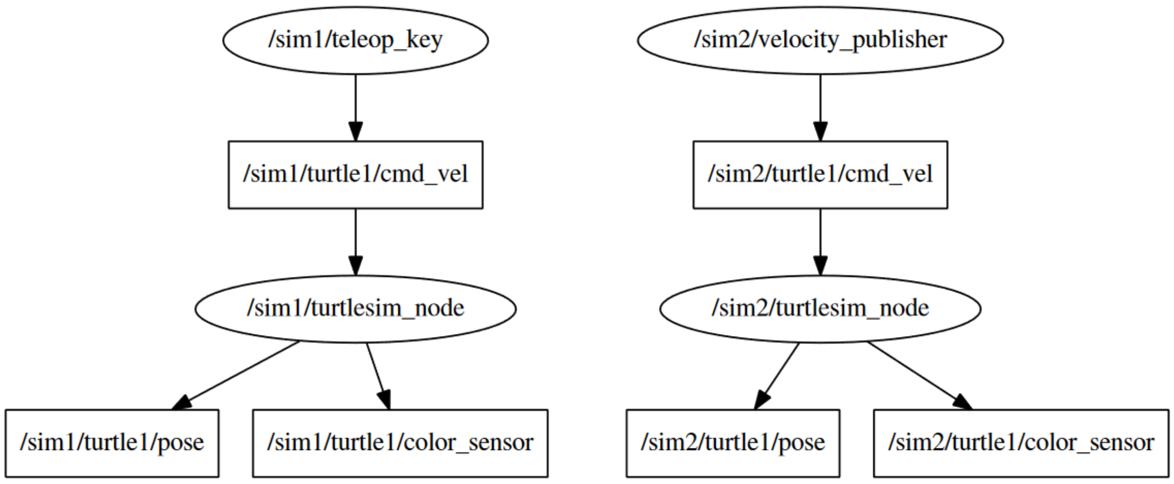


Figure 6: Nodes and topics (in ellipses and rectangles, respectively) [27]

ROS provides a wide range of libraries and tools for tasks such as robot localization, motion planning, and perception. It also has a large community of users and developers, which makes it easy to find support and resources for building and deploying robotic applications.

In summary, ROS is a powerful and flexible platform for building and deploying robotic applications, with a rich set of libraries and tools and a strong community of users and developers.

3 State of the art

The first system which can be considered as having applied HiL even by today's standards was an analog flight simulation, created in 1910 by the "Sanders Teacher". Since then, HiL has been adapted to the emerging technologies of the time. The Defense and Aerospace industry had been taking advantage of HiL since the 1950's, however major improvements to digital computers in the 1970's and 1980's led to the use of HiL in the automotive industry. The introduction of commercial HiL systems in the 1990's resulted in widespread applications relying on HiL [6]. The aerospace and automotive industries significantly rely on HiL as part of their validation process for critical systems due to the inherent risk posed by possibly destructive tests and cost-savings as opposed to building the full physical system. An often-overlooked benefit in recent years is the ability to perform distributed tests i.e., remotely, as the design and manufacturing team can be located at such a distance from the testing team where it is economically unfeasible to physically test the system [26].

With AD being "the next big thing" in the automotive industry for the past decade, HiL has followed suit, as can be seen in Figure 7. While the number of publications seems quite low, the trend indicates growth of the academic interest in researching HiL for AD. This is no surprise however, as most systems using HiL are developed with the efforts and for the purposes of private companies, as indicated by dSPACE's, a leading company in HiL and one of the first to offer commercial vertical HiL solutions [27], list of cooperating partners [28] consisting of only private companies for which there is no incentive to publish their work.

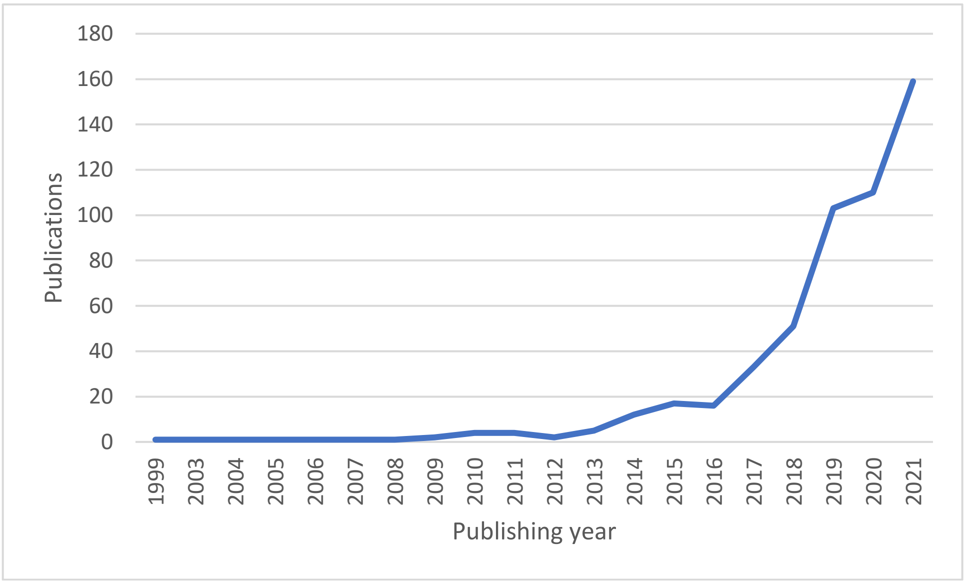


Figure 7: Number of publications on use of HiL for AD per year in Scopus database

Research publications in which CARLA is used are scarcer and even more so when it comes to publications involving application of HiL within CARLA, as can be seen in Figure 8.

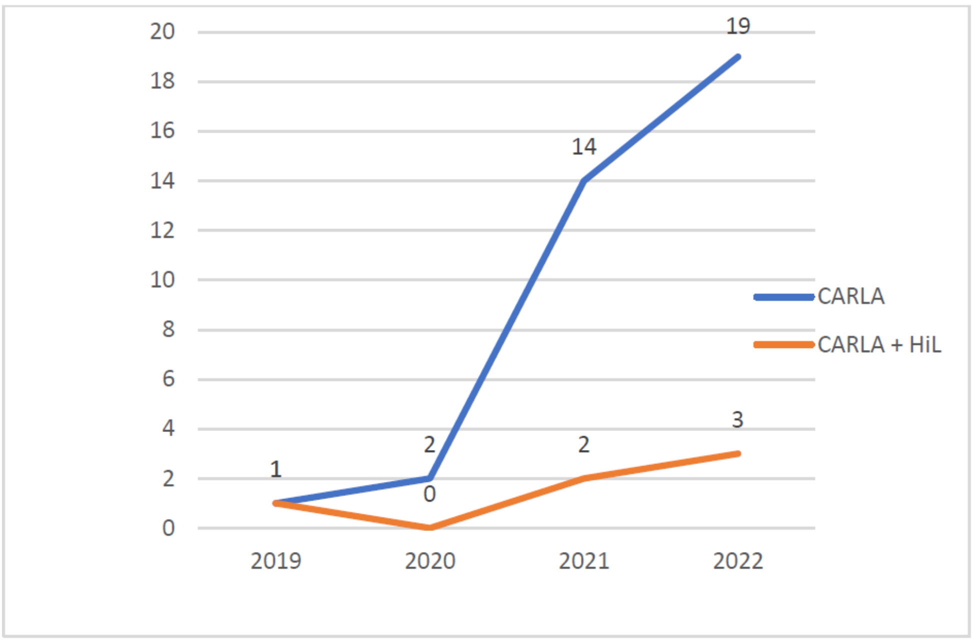


Figure 8: Number of publications per year in Scopus database, CARLA vs. CARLA and HiL

Two of the publications using CARLA are of special interest to this study. One of them proposes a framework for testing communication between vehicles and other systems using the ITS-G5 communication standard [29], whereas the other developed a framework which focuses on preparing sensor information data to be used for collective perception [30]. As the EDDY project involves both concepts, these publications provide an insight to how the digital twin can incorporate them using their respective frameworks. Testing the performance of ITS-G5 communication between multiple test vehicles and the UDM simultaneously can be more easily implemented based on the framework. The standardization of object messages for the UDM requires a significant effort from the EDDY project partners, thus the ability to measure their effectiveness with the digital twin is of great importance.

4 Requirements Analysis

4.1 System Expectations

The digital twin shall consist of a fully setup CARLA simulation containing a digital recreation of the TAVF, a simulated test vehicle and have established communication with the UDM. The CARLA simulation server will run on a computer capable of maintaining a framerate which will provide viable accuracy of the data. As the data received from the UDM depends on the vehicle's location, significant lag in reporting the location to the UDM, caused by a low framerate, will stagger the representation of objects from the UDM, in practical terms the simulation might not reflect the actual state surrounding the vehicle. The simulated vehicle shall be capable of reenacting test drives based on real recorded data and random path following when no such data is used. Communication with the UDM shall be conducted by a ROS node which will report the GNSS position of the test vehicle, gathered by a simulated GNSS sensor, and the traffic objects returned from the UDM based on that position shall be visualized. Additionally, the system will visualize data from the simulated sensors.

To fulfill the expectations, the following procedure is established:

- Choosing hardware components for a computer which are compatible and powerful enough to run a CARLA server at a high framerate
- Installing Ubuntu 22.04 on the computer
- Downloading and building CARLA and Unreal Engine 4.26 from source
- Creating a CARLA compatible replica of the TAVF
- Integrating a simulated vehicle into the simulation
- Establishing a controller capable of guided and randomized test drives
- Communication using ROS's Publisher/Subscriber architecture
- Recording and storing simulation data

4.2 EDDY-in-The-Loop

In the interest of presenting the viability of HiL testing using CARLA, the communication with the UDM shall be tested. As the simulated test vehicle executes test drives it shall communicate its position to the UDM via a ROS node and measure the response time needed for the UDM to respond. As the planned features of the UDM are not yet implemented this is the most viable test as of the development of the system. EDDY-in-The-Loop is to be expanded with the addition of features to the UDM. The addition of tests shall be a simple procedure due to the digital twin's adaptability.

5 Hardware and Software

5.1 Hardware

The minimum and recommended hardware requirements for CARLA presented in Table 1 are based on the minimum and recommended hardware for Unreal Engine 4.26 [31] [32], aside from the storage which takes both Unreal Engine and CARLA into account [33] [1]. Access to the used hardware was provided by the Urban Mobility Lab [21] for the writing of this thesis.

	Minimum	Recommended	Used
CPU [threads]	6	8	32
RAM [GB]	16	32	128
VRAM [GB]	6	8	24
Storage [GB]	Min. 130	Min. 130	2000

Table 1: Hardware Requirements for Unreal Engine 4.26 + CARLA

The choice of components may seem excessive when compared to the recommended hardware, however these requirements are more suited for game development where the visuals make use of most of the graphics processing unit (GPU). CARLA on the other hand approximates real life physics in its simulations which alongside the visuals on a large enough scale, such as that of the TAVF, will throttle most commercial grade GPUs. Therefore, a factory overclocked Nvidia RTX 3090 with 24GB of VRAM was chosen. As for the processor, a CARLA client script is executed exclusively on the central processing unit so in order to ensure best case scenario parallelization the 16 core, 32 thread AMD Ryzen 9 5950X with a base clock of 3.4GHz and a maximal boost clock of 4.9GHz was the obvious choice. The only dilemma arised upon considering the dynamic RAM, where a trade-off between capacity and speed had to be made, however capacity prevailed at the end as it is more versatile and the computer, being funded by the UML, shall be used for further projects.

5.2 Software

5.2.1 Requirements

The latest version of CARLA (0.9.13 as of writing this paper) is officially supported for Ubuntu 18.04 [34], however it is available for Ubuntu 20.04 as well.

[1] The amount of required storage is not prohibitively high, regardless its speed might be a limiting factor to performance. Both minimum and recommended system builds for Unreal Engine 4.26 specifically recommend an SSD for storage.

5.2.2 Used

Software	Purpose
CARLA version 0.9.13	Creating a CARLA simulation
Unreal Engine 4.26	
SUMO's netconvert	Conversion from OpenStreetMap file to OpenDRIVE file
RoadRunner	Conversion from OpenDRIVE file to CARLA compatible objects
VirtualBox	Set up of Virtual Machine for distributed CARLA client
ROS	Communication with UDM

The author used Ubuntu 22.04 for the implementation of the digital twin, despite the apparent lack of support for it by CARLA. The reason for this is the GPU used in the system, an NVIDIA GeForce RTX 3090, was unrecognizable to the drivers available on Ubuntu 20.04. This resulted in CARLA running purely on the CPU which led to an abysmal framerate, under 5 FPS at any given time. To take advantage of the full potential of the system the circumvention described next was used. Furthermore, a Virtual Machine (VM) with Ubuntu 20.04 LTS, created using VirtualBox, was employed for executing the CARLA client scripts.

5.2.3 Circumvention

Since Ubuntu 22.04 was deemed necessary to continue with the efforts for this study, the author attempted building CARLA from source on it. This attempt was unsuccessful as the compiler central to building CARLA and Unreal Engine 4.26 [34], clang version 8 or 10, is not provided for Ubuntu 22.04 [35] [36]. Upon coming across this issue, Ubuntu 20.04 was reinstalled on the system. This allowed for CARLA to be built. The author took advantage of Ubuntu's ability to upgrade the operating system and kernel while keeping all data intact [37]. Such an upgrade is only possible on an Ubuntu Long Term Supported (LTS) system. CARLA works without any issues under Ubuntu 22.04 and the NVIDIA RTX 3090 card is fully recognized and utilized with the latest drivers. One must take note that when applying this circumvention the make commands to use/adapt CARLA from the terminal [38] can not be used, as they depend on the clang-10 compiler, this is easily remediated by running the shell scripts called by the make commands. To launch CARLA, we use BuildCarlaUE4.sh from the directory it is in as shown in Listing 1.

```
./BuildCarlaUE4.sh --launch
```

Listing 1: Terminal command to launch CARLA

6 Implementation

6.1 System Concept

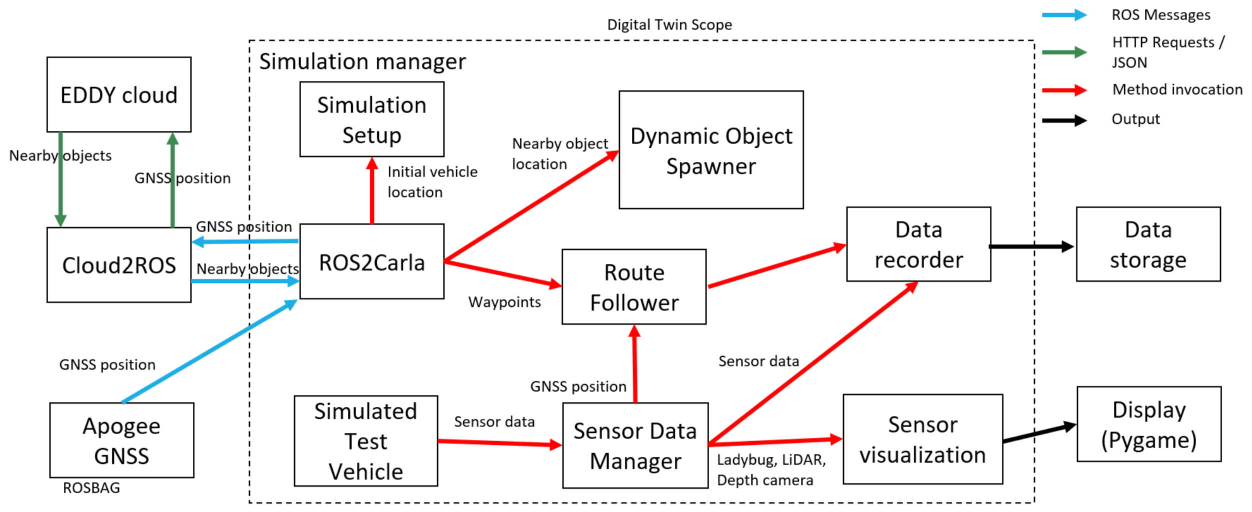


Figure 9: Diagram displaying the Digital Twin functionalities and communications

The diagram in Figure 9 shows the system architecture of the Digital Twin, accompanied by external nodes necessary for testing the UDM on the left and keeping records and visualization on the right. The implementation of the diagram modules belonging to the scope of the digital twin will be discussed in the following sections based on their joint functionality in terms of the digital twin, whereas the rest of this section gives an overview of the modules externally provided for the improvement of the digital twin.

6.1.1 Communication with UDM

The communication with the UDM is established via the Cloud2ROS ROS node [39] which sends an HTTP request to the server. Based on the GNSS position, reported by ROS2Carla, Cloud2ROS establishes a square bounding box with a customizable side length and packs it in an HTTP request that is then sent to the UDM. The UDM returns all objects registered for the area represented by the bounding box i.e., in the vicinity of the vehicle. These objects are then stored in an array and published to its respective ROS node.

To make use of ROS, it must be installed on the system and a ROS workspace must be configured. The guides [40], [41] for both procedures were followed. Additionally, a launch file was created in order to have the ability to pass parameters via the command line. The launch file is shown in Listing 2, it effectively initializes the ROS parameters [42] to be used within the simulation management script and then runs the script.

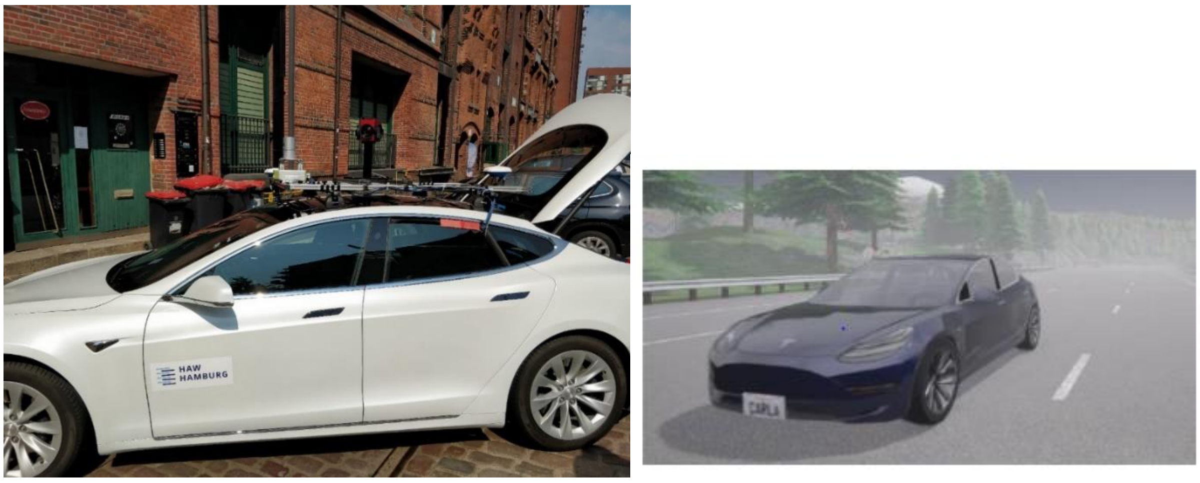
```
<!-- -->

<launch>

  <!-- launch a complete carla-ros-environment -->

    <!-- Simulation manager -->

    <param name="display" type="bool"/>

    <param name="rosbag" type="str"/>

    <param name="loop_rosbag" type="bool"/>

    <param name="timer" type="float"/>

  <node pkg="eddy_digital_twin" type="manager.py"
name="eddy_digital_twin_manager">

</launch>
```

Listing 2: Launch file used for managing the simulation

6.1.2 Simulated Test Vehicle

The simulated test vehicle used for this thesis is modelled after the Urban Mobility Lab's Tesla Model S [43] with advanced sensing and communications capabilities, both the physical and simulated vehicle can be seen in Figure 10. The model used in CARLA reflects all capabilities of the UML's test vehicle.

In addition to the LiDAR and cameras, a GNSS sensor [44] was coupled to the simulated vehicle. The sensor is configured to report the Geodetic location of the test vehicle every 10 frames to ensure the UDM is not overloaded with requests, nor the ROS communication pipeline gets bottlenecked.



a) b)

Figure 10: a) Urban Mobility Lab’s Upgraded Tesla Model S [20];

b) Simulated Tesla Model S [43]

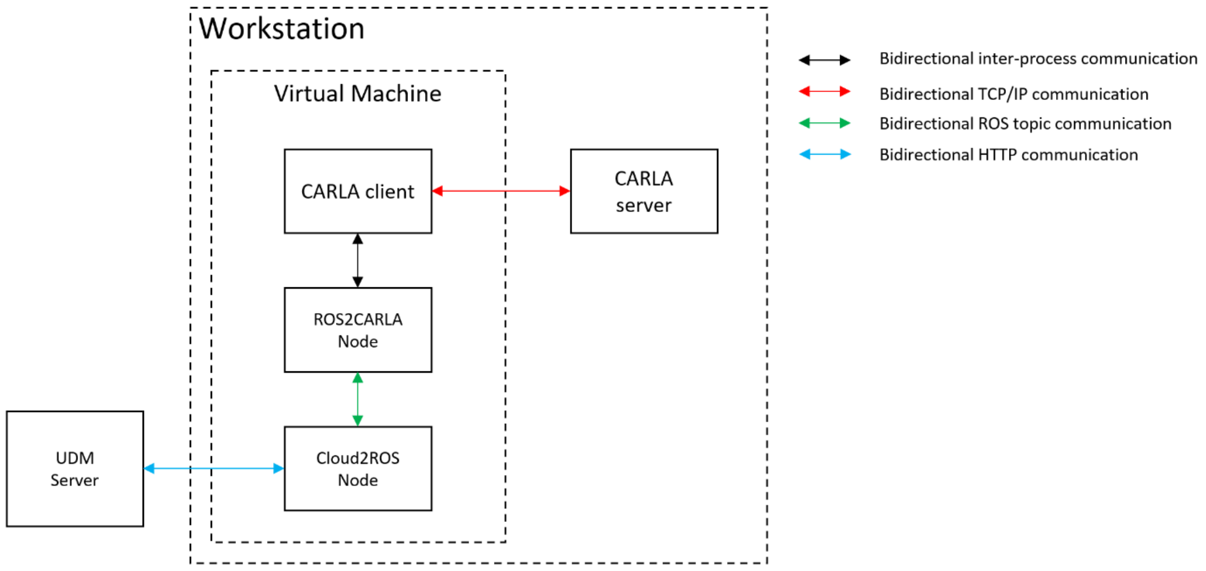
6.2 Simulation Setup

CARLA version 0.9.13 was built from source as other installation methods do not provide access to the Unreal Engine editor which limits the user to interacting with the CARLA server only via the Python API. The additional capabilities allowed by the editor are essential when integrating a custom map in CARLA, as it provides the ability to check the map for accuracy and correctness easily.

The CARLA client alongside the Cloud2ROS node were ran in a VM to take advantage of CARLA’s server/client architecture. While this was not strictly necessary for this study, it was deemed beneficial for future use of the digital twin when direct access to the system running the server is not immediately available. Ubuntu 20.04 LTS was chosen for the VM’s operating system primarily due to ROS noetic, the ROS version of choice for the UML’s research activities, is not supported on Ubuntu 22.04 LTS, additionally the graphics card is not utilized by CARLA client scripts. The virtualization software of choice for this thesis was VirtualBox due to the author’s previous familiarity with it and its ease of use on Ubuntu.

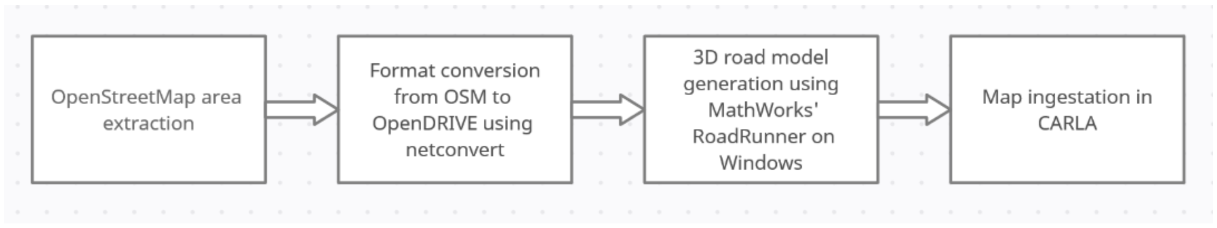
The full, step by step, procedure to set up a system mirroring the one used for this thesis is provided in the Appendix, section 11.1.

An overview of the communication flow of the digital twin is shown in Figure 11.


Workstation
Virtual Machine
CARLA client
CARLA server
ROS2CARLA Node
Cloud2ROS Node
UDM Server
Bidirectional inter-process communication
Bidirectional TCP/IP communication
Bidirectional ROS topic communication
Bidirectional HTTP communication
Figure 11: Communication flow of the digital twin

6.2.1 TAVF in CARLA

At the start of writing this paper the TAVF was not available as a CARLA compatible map, thus the procedure to create one such map, shown in Figure 12 and described next, was established.


OpenStreetMap area extraction
Format conversion from OSM to OpenDRIVE using netconvert
3D road model generation using MathWorks' RoadRunner on Windows
Map ingestation in CARLA
Figure 12: Custom map generation and ingestion in CARLA
sources left to right: [45], [46], [47], [48]

Foremost a base map must be acquired. OpenStreetMap (OSM) hosts open-source map data and allows extraction of a specific area on their website [45]. After navigating the map view to display the area which is to be extracted, it is recommended to use OSM’s feature for manual selection of a specific area.

A rectangle representing the bounds of the area to be extracted is displayed overlayed on the map view. Figure 13 shows the selected map area of interest within the rectangle’s highlighted inner area, for the purposes of this paper this area is focused on the TAVF.

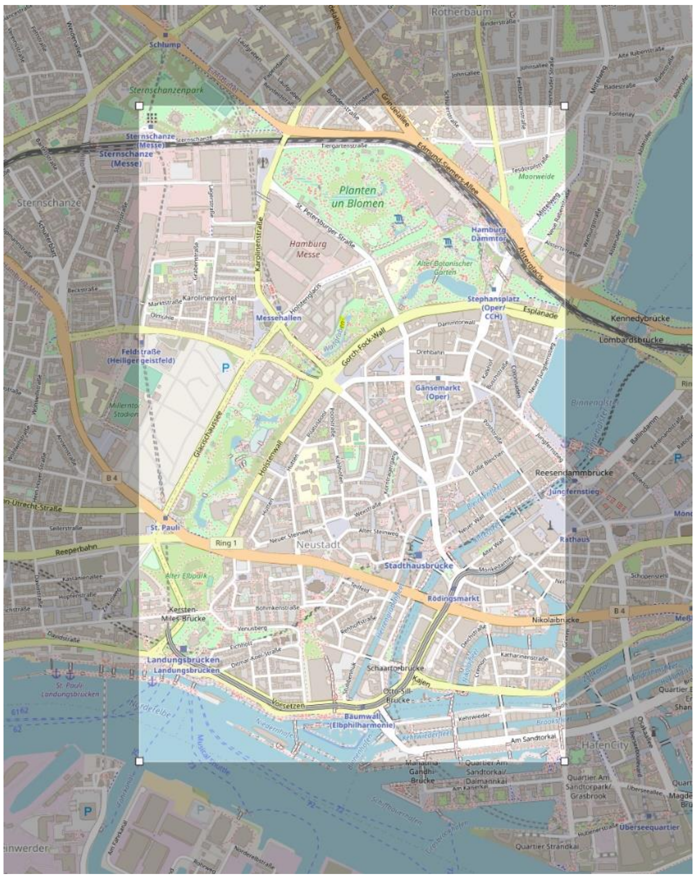


Figure 13: OSM map view showing the highlighted TAVF to be extracted

In case the selected area is too large to request from OSM directly, they recommend BBBike's [49] extraction tool which allows the user to define a shape with more than four sides, as well as JOSM [50], a Java based OSM editor which can handle larger amounts of data at once.

The conversion from OSM to ASAM OpenDRIVE is done using Simulation of Urban Mobility's (SUMO) netconvert tool [48]. First SUMO must be installed, which can be done by using the command from Listing 2 in the terminal.

```
sudo apt install sumo
```

Listing 3: Terminal command to install SUMO

After navigating to the folder where the OSM map is stored, the command from Listing 3 can be run to convert it to an ASAM OpenDRIVE road network description.

```
netconvert –osm-files <OSM filename>.osm -o <OpenDRIVE filename>.xodr
```

Listing 4: Terminal command to convert from OSM to OpenDRIVE using netconvert

The next step requires MathWorks' RoadRunner program, an interactive 3D editor for automated driving system simulation. Using Blender [51], a 3D modelling program, with the blender-osm [52] addon is an open-source alternative however the author had little success using it to properly extract the roads. Additionally, there is the possibility to ingest an OSM or OpenDRIVE map directly in CARLA [48], however this method is rudimentary and depends on the robustness of the map file.

The installation of RoadRunner, version R2022b, was executed according to the guide provided by MathWorks [53]. While the program runs well on Ubuntu, some OpenDRIVE files could not be imported in it, thus using a Windows system to generate the 3D road models with RoadRunner is recommended. The attempts to import OpenDRIVE files in RoadRunner on Ubuntu provided no warnings or errors upon failure, thus it is not specifically clear as to why they happen. The same files can be imported on Windows flawlessly so it is presumed that RoadRunner does not function quite as well on Ubuntu.

Following MathWorks' guide [54] the OpenDRIVE file previously created was imported in RoadRunner, the successful output of this is shown in Figure 14. The Filmbox file and necessary accompanying files were exported from RoadRunner and ingested in CARLA by following the sections relevant for an export/import using Filmbox in MathWorks' guide [47] as well.

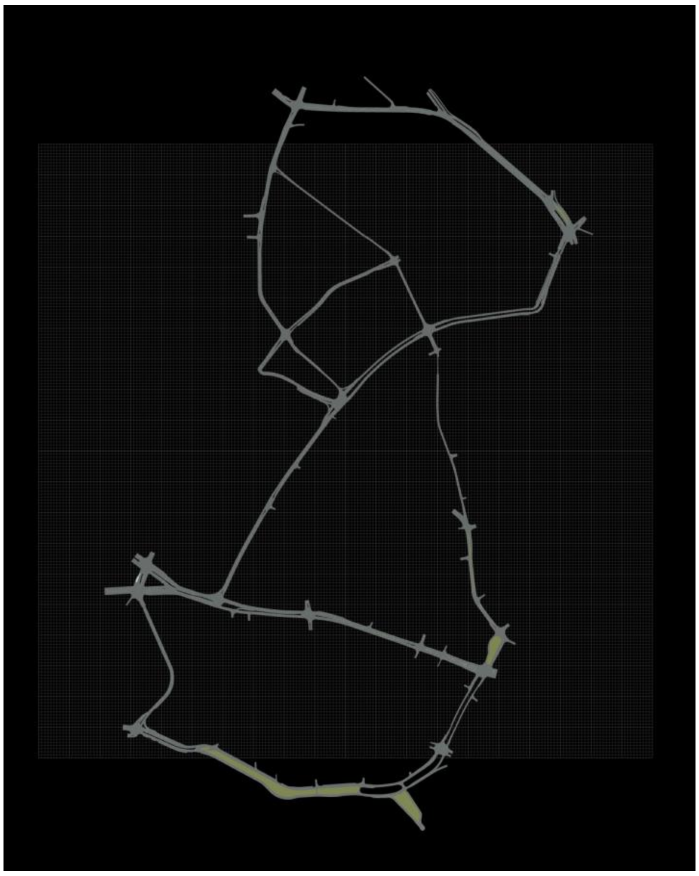


Figure 14: TAVF OpenDRIVE road network supplied by LGV imported in RoadRunner

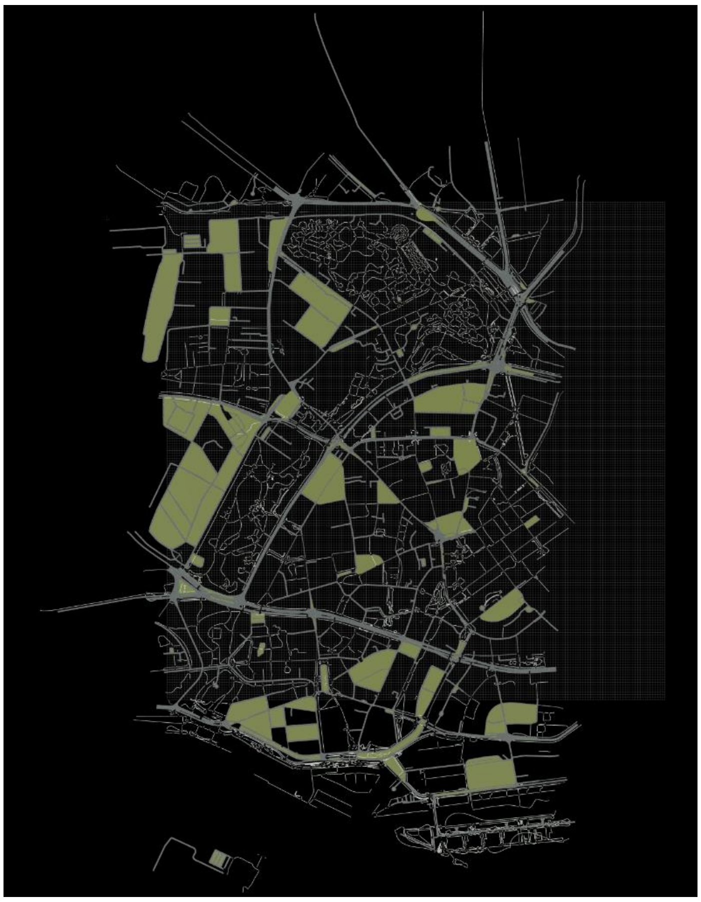


Figure 15: TAVF OpenDRIVE generated from OSM data

6.2.1.1 Provided OpenDRIVE data

The Landesbetrieb Geoinformation und Vermessung (LGV) [55] i.e., State Office for Geoinformation and Surveying, Hamburg, provided a more accurate and handcrafted OpenDRIVE network representing the TAVF as part of the EDDY project. Such a network will be in most cases of higher quality than a semi-automatically generated one resulting in its use for the purposes of this thesis. This difference in quality and workability can be seen between Figure 14 and Figure 15, where the former which shows the LGV provided map contains only the roads that are part of the TAVF, whereas the latter includes every side street and alley within the designated area. The additional information extracted from OSM primarily impacts the performance of the CARLA server negatively as well as being irrelevant to the study.

6.2.2 Test Vehicle

As previously stated, the test vehicle used in the digital twin was provided [43], however some adaptations to the management of sensor data were made to ensure better performance and visual clarity. Figure 11 displays the architecture employed. The major distinction from the provided sensor management system is the synchronized gathering and output of sensor data, as well as the organization of the display, the difference of which can be seen in Figure 17 and Figure 18. The new management system display output is more human comprehensive by outlaying the 360-degree camera panoramically.

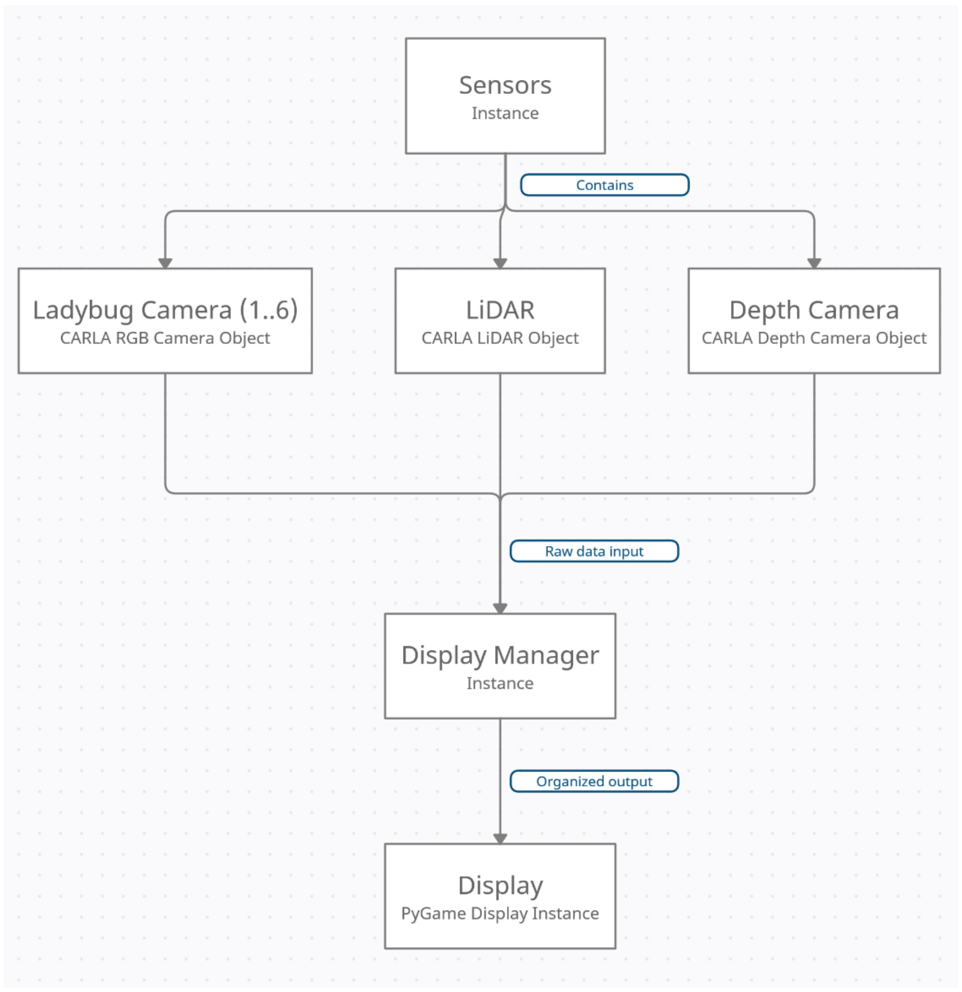
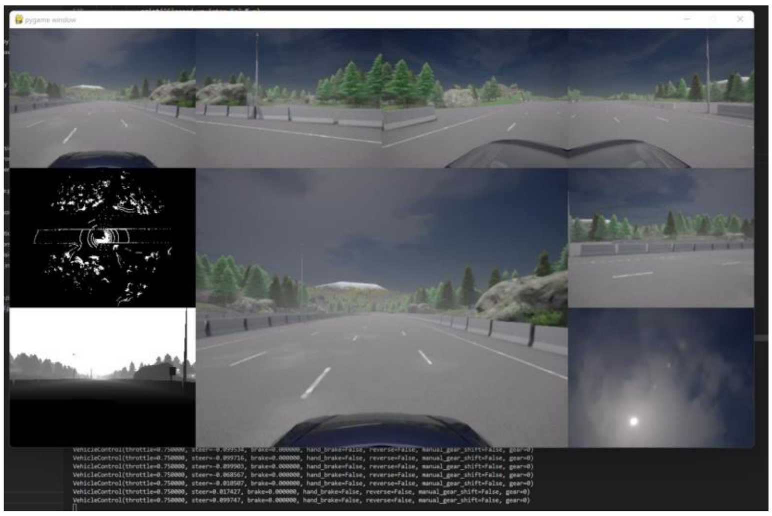


Figure 16: Sensor Management and Display Diagram



```
VehicleControl(throttle=0.750000, steer=-0.099534, brake=0.000000, hand_brake=False, reverse=False, manual_gear_shift=False, gear=0)
VehicleControl(throttle=0.750000, steer=-0.099716, brake=0.000000, hand_brake=False, reverse=False, manual_gear_shift=False, gear=0)
VehicleControl(throttle=0.750000, steer=-0.099903, brake=0.000000, hand_brake=False, reverse=False, manual_gear_shift=False, gear=0)
VehicleControl(throttle=0.750000, steer=-0.068567, brake=0.000000, hand_brake=False, reverse=False, manual_gear_shift=False, gear=0)
VehicleControl(throttle=0.750000, steer=-0.010507, brake=0.000000, hand_brake=False, reverse=False, manual_gear_shift=False, gear=0)
VehicleControl(throttle=0.750000, steer=0.017427, brake=0.000000, hand_brake=False, reverse=False, manual_gear_shift=False, gear=0)
VehicleControl(throttle=0.750000, steer=0.099747, brake=0.000000, hand_brake=False, reverse=False, manual_gear_shift=False, gear=0)
```

Figure 17: Display organization of provided sensor management system, [45]

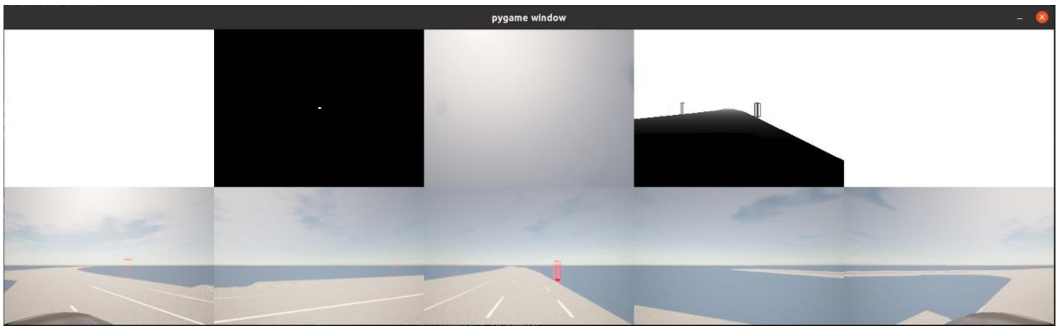


Figure 18: Display organization of used sensor management system, top row: blank, LiDAR visualization, Top Camera, Depth Camera; bottom row: Back-Left Camera, Front-Left Camera, Front Camera, Front-Right Camera, Back-Right Camera

6.2.3 Path Planning and Route Following

To simulate a test drive within CARLA, a path planning and following algorithm had to be created and implemented, this algorithm is shown in Listing 5, basically if a route is given in the form of Geodetic coordinates the vehicle will follow it, otherwise the system chooses a random point on the map to start and chooses the next point as the nearest to the previous of the same road network. Important to note is that a CARLA waypoint requires the location provided to be converted from Geodetic to Scene East-North-Up (ENU) coordinates i.e., a conversion from GNSS to local Unreal Engine 4 World coordinates. The formula for this conversion is provided in Equation 1. The reference latitude and longitude, Lat_{ref} (reference latitude) and Lon_{ref} (reference longitude), represent the center of the map as Geodetic coordinates and R_{earth} is the Earth's equatorial radius in meters.

```
if rosbag path given:
      extract GNSS positions and store in array: positions
      convert positions[0] to World coordinates and store as spawn point
      convert positions[1] to World coordinates
      find closest waypoint to position[1] and store as next waypoint

      while waypoint index < len(positions):
            if car distance to next waypoint < 2.5 (m):
                  increase waypoint index
                  get next waypoint
            compute control with PID
            apply control to vehicle
      else:
            reset
else:
      choose random waypoint and store as spawn point
      find closest waypoint to spawn point
      if closest waypoint exists:
            set as next
      else:
            reset

      while next waypoint exists:
            if car distance to next waypoint < 2.5 (m):
                  get next waypoint
            compute control with PID
            apply control to vehicle
```

Listing 5: Algorithm for path planning and following

$$x = \cos\left(Lat_{ref} * \frac{\pi}{180}\right) * \pi + \frac{R_{earth}}{180} * lon - \cos\left(Lat_{ref} * \frac{\pi}{180}\right) * \pi * \frac{R_{earth}}{180} * Lon_{ref}$$

$$y = cos\left(Lat_{ref} * \frac{\pi}{180}\right) * \pi + \frac{R_{earth}}{180} * \log\left(\tan(90 * lat) * \frac{\pi}{360}\right) - \left(Lat_{ref} * \frac{\pi}{180}\right) * R_{earth} * \log\left(\tan\left(90 + Lat_{ref}\right) * \frac{\pi}{360}\right)$$

Equation 1: Conversion from given latitude and longitude to Unreal Engine 4 World Coordinates [56]

Aside from the coordinate system conversion a proportional–integral–derivative (PID) controller [56] must be tuned to ensure proper route following. While it is possible to give specific instructions on tuning the PID controller, it is not practical as it is highly system dependent i.e., any difference whether apparent or not between the hardware or software used to run the simulation will drastically impact the behavior of the PID controller. A much more effective way of considering the coefficient adaptations is anecdotally, where the effects of the tuning can be thought of as such:

- Proportional coefficient corresponds to the immediate response of the system to changes in its environment
- Derivative coefficient adapts the immediate response based on the instantaneous error of the system's output
- Integral coefficient corrects the system over time

6.3 ROS Communication

As mentioned in Communication with UDM, a ROS node is necessary for transmitting the GNSS position to the UDM and to receive traffic data from it. The CARLA client script creates a "sim_manager" node which is used to publish the simulated vehicle's GNSS position to the "gps_pos" topic. Before publishing the vehicle's longitude and latitude are packed into a message of type SbgGpsPos [57]. The "cloud_linker" node, created by the Cloud2ROS script, is subscribed to the "gps_pos" topic and upon seeing a message it sends an HTTP request to the UDM containing the position and the radius for which traffic objects are to be returned. Any objects that are returned are then packed into an array of a custom message type where the object's name, type, object class and position are stored. This message array is published to the "eddy_objects_msg" topic to which the "sim_manager" node is subscribed. The objects are then displayed within CARLA using the debugging feature.

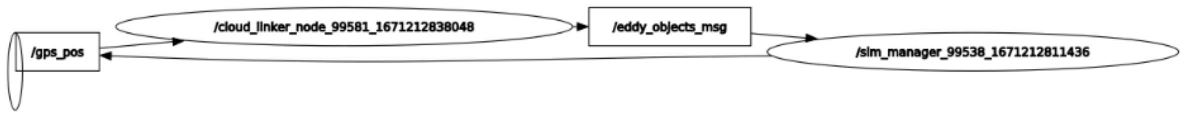

/cloud_linker_node_99581_1671212838048
/eddy_objects_msg
/gps_pos
/sim_manager_99538_1671212811436

Figure 19: RQT Graph displaying the ROS nodes of communication between Digital Twin and UDM

7 Functionality Test

As the nature of the test results is partly visual, recordings are cited to display them. The following subsections shall describe the conditions under which the tests were performed, any issues with the test results and whether they are deemed acceptable.

7.1 Underlying Components

It is vital to ensure that the components of the digital twin are functional individually, before attempting to use the system to test another system.

7.1.1 Vehicle controller test

7.1.1.1 Provided route

The ability of the simulated test vehicle controller to follow a given route was tested by inputting recorded GNSS positions from a physical test drive, performed as part of the activities of the UML. The waypoints for the provided route were visualized using CARLA's debugging module which draws bounding boxes in the simulation. [58]

The conversion from GNSS positions to CARLA waypoints is imperfect, as the assignment of a position to a waypoint depends on the accuracy of the position recording and the underlying road network in CARLA. The inaccuracies that result from faults in the recording and road network have been summarized in Figure 20. The mean error between the recorded positions and simulated waypoints is 12.03318 meters. Despite such a significant error, the route following of the controller approximates the real route well enough to be feasible for use in testing the UDM.

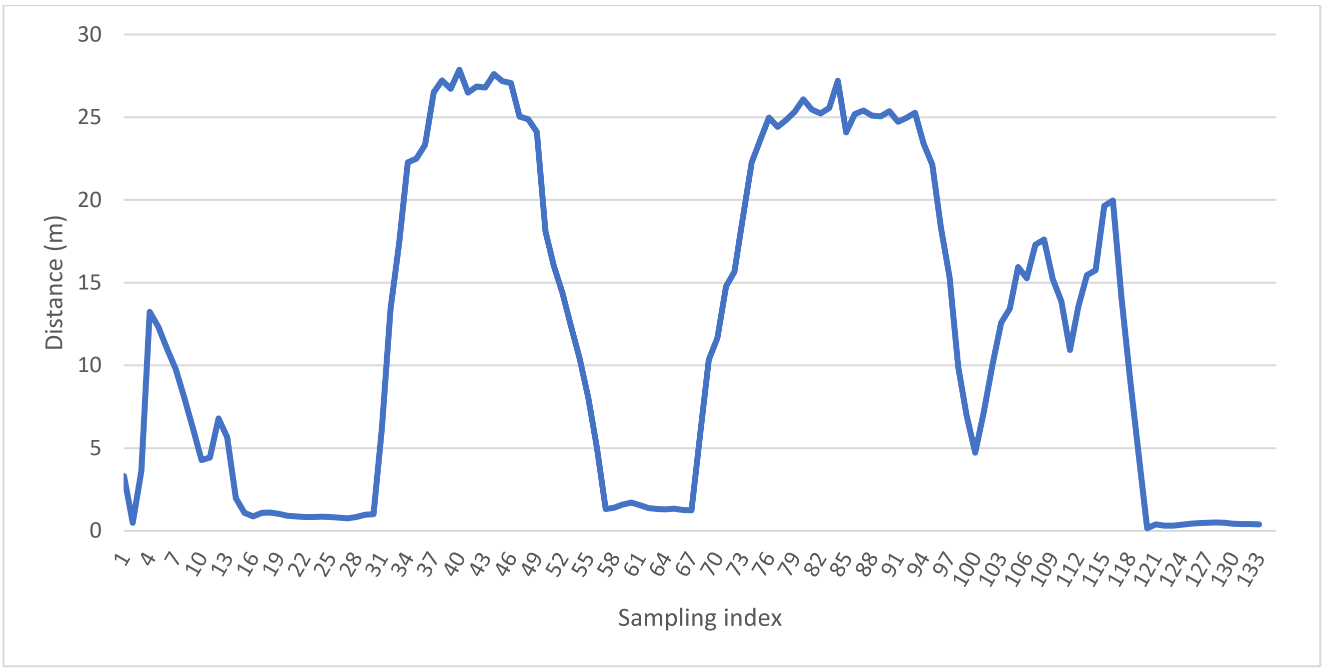


Figure 20: Distance between recorded positions and simulated waypoints

7.1.1.2 Randomized route

Simulating the test vehicle driving along randomized routes can be beneficial when collecting data across a longer period, as repeating a provided route will produce approximately the same results each time [59]. The time to reset was the chosen metric to check whether the randomized route following is viable for testing the UDM. After letting the simulation run for 30 minutes the mean time to reset the simulated vehicle was 15.71548 seconds, with a minimum of 7 nanoseconds i.e., instantaneous reset after choosing a spawn point with no following waypoint, and a maximum time to reset of 72.143 seconds. The time to reset will be improved with the improvement of the road network representing the TAVF.

7.1.2 Displaying objects received from UDM

Upon receiving the list of traffic objects from the UDM, the digital twin displays them without any issues [60]. The visual representation of the traffic objects can be seen in Figure 21.

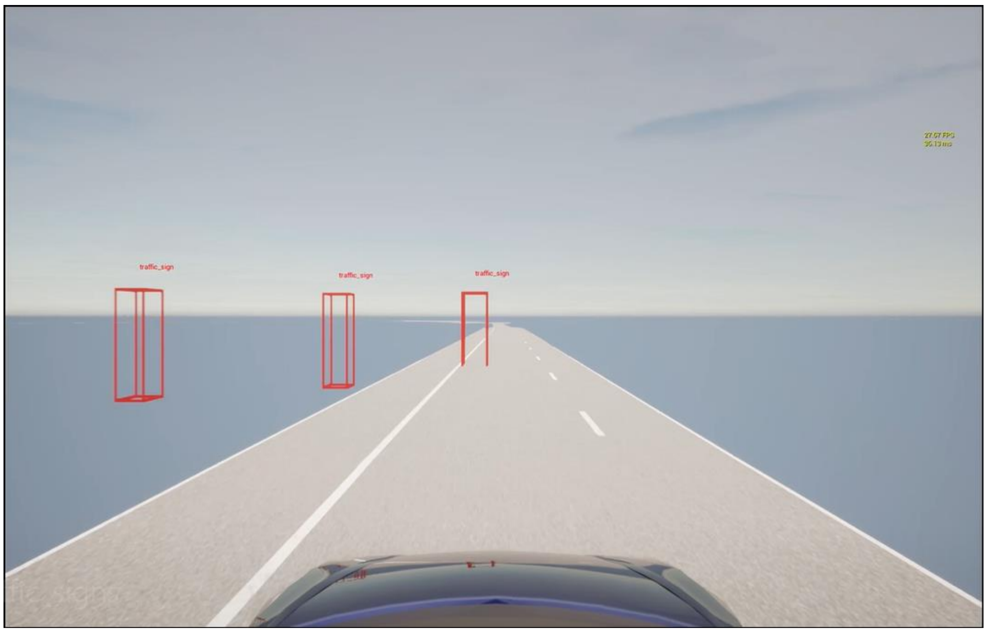


Figure 21: Traffic Signs returned from the UDM displayed in the simulation

8 Testing UDM with the Digital Twin

Testing the UDM was done by gathering the latencies of the response to a sent GNSS location. The latency was gathered from within the Cloud2ROS and the client script to observe the effects of the digital twin by processing the information. As can be seen in Figure 22, each request takes longer when sent from the digital twin. This is expected, as the communication pipeline has been extended with an additional ROS node along with the processing inside the client script.

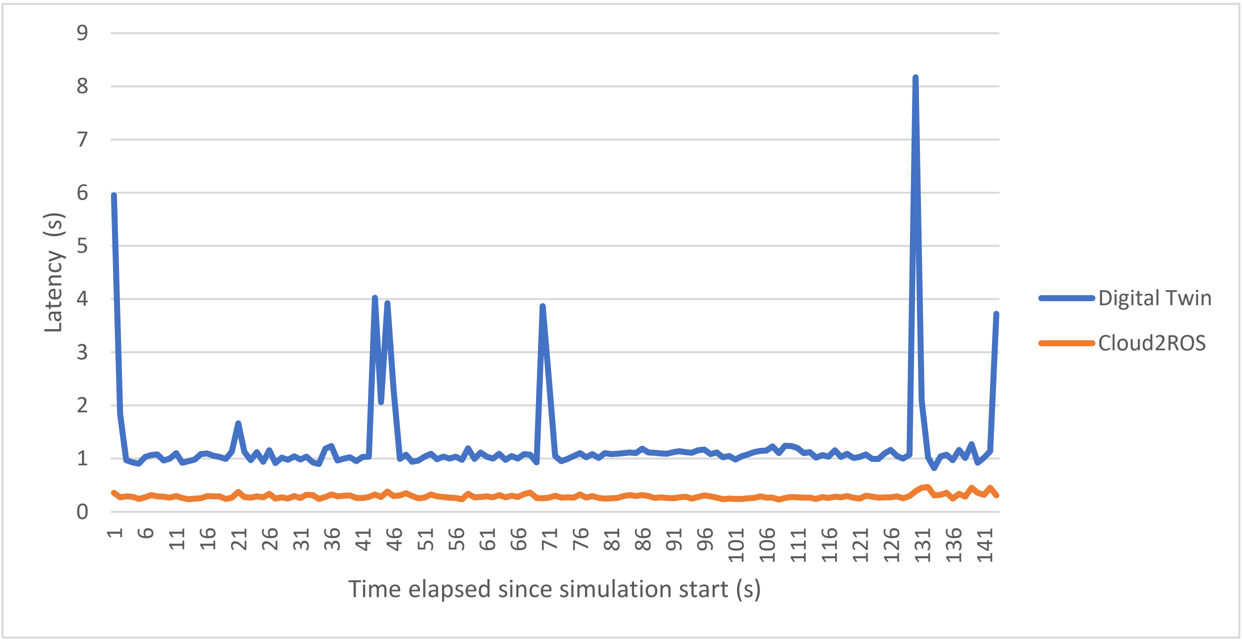


Figure 22: Latencies of Cloud2ROS script (orange) and Digital Twin client script (blue) for communication with UDM

The data has been summarized in Table 2. An overall factor of 4.368 was found in the time it takes to send to and receive data from the UDM between the Digital Twin and the Cloud2ROS script.

	Cloud2ROS latency (s)	Digital Twin latency (s)
Minimum	0.23088	0.82161
Maximum	0.465356	8.17102
Mean	0.288734	1.261217

Table 2: Latency statistics for Cloud2ROS and Digital Twin scripts

9 Conclusion

A framework for creating a digital twin which consists of a CARLA simulation and communication with the UDM was established. The CARLA simulation successfully simulates the TAVF and its road network, and a test vehicle with sensing capabilities, modelled after the physical sensors, capable of autonomous and guided driving. Furthermore, the simulation visualizes the sensor data and traffic objects provided by the UDM at an acceptable framerate. Unfortunately, CARLA's functionality for the addition of custom assets to the simulation is not working on version 0.9.13. Such issues are expected of open-source projects which are still in development, thus a workaround taking advantage of CARLA's debugging abilities was employed.

For the purposes of the digital twin a highly capable computer was assembled and Ubuntu 22.04 LTS was set up as its operating system. Additionally, the computer runs a VM with Ubuntu 20.04 LTS as its operating system to use ROS for the communication with the UDM.

To retrieve the traffic objects from the UDM with ROS a ROS workspace was created which was used to manage the message types and run the CARLA client script. The script initializes its own ROS node via which the messages are transmitted and received using ROS's publisher/subscriber communication model. This was chosen to ensure compatibility with the rest of the efforts of the UML as well as the EDDY project.

The digital twin performs as per its expected tasks i.e., the test vehicle drives along the TAVF while gathering data from the UDM and using that data to test the UDM's functionalities. As the UDM is in its early stages of data collection the test presented in this study is rudimentary, yet adequate, with the purpose of presenting the system's capabilities in future testing. EDDY-in-the-Loop has been established and can be adapted with minimal effort.

10 Future Work

As the digital twin was built as part of EDDY, a project that will be continuously developed throughout the next 2 years, naturally the system will have to be adjusted for the changes and additions from the EDDY project. Changes that will predictably impact the digital twin are the addition of data to the UDM for which new tests should be conceptualized and implemented. The additions most significant to the digital twin will be the VRU recognition and improved localization based on landmarks. With the implementation of VRUs and landmarks within the digital twin, the capabilities of the simulated vehicle to perceive VRUs and report them to the UDM, react to situations involving VRUs announced by the UDM, and calculate its location based on surrounding landmarks can be tested. Likewise, a test to inspect the viability of communicating objects to and from the UDM simultaneously can be set up with a multi-client environment where several test vehicles are simulated. Once the object message formats of the UDM have been proposed, their ability to effectively convey information between the UDM and vehicles can be empirically tested. Furthermore, the simulated map of the TAVF shall change to reflect the actual TAVF.

Considering more than additional functionalities and data of the UDM on which the system relies, the opportunity to further increase the performance of the simulation server and with that the digital twin is open. This could be fulfilled by implementing multi-threading to the client script where the route following, sensor data collection, and UDM communication are each managed by a separate thread to ensure concurrent executions of these processes. However, the introduction of multi-threading comes with the possibility of limiting the system's adaptability as multi-threaded scripts require quite a bit more care when working with. The advantages and disadvantages of this approach should be taken into consideration before the script is reworked.

A significant improvement to the visualization of traffic objects can be made by taking advantage of CARLA's custom object addition as soon as an updated version of CARLA fixes this feature. Simulating and displaying the traffic objects as would be seen in real life could be utilized for additional testing of the UDM.

With the aim of more closely approximating the real behavior of vehicle localization noise can be introduced to the simulated GNSS sensor as part of its configuration. The noise could be gathered from the measurements of the physical GNSS sensor.

11 Appendix

11.1 System Setup Manual

11.1.1 CARLA Installation

Guide URL	Purpose
https://ubuntu.com/tutorials/install-ubuntu-desktop#1-overview	Ubuntu installation
https://carla.readthedocs.io/en/0.9.13/build_linux/ - software-requirements	
https://www.unrealengine.com/en-US/ue-on-github	CARLA and Unreal Engine installation
https://carla.readthedocs.io/en/0.9.13/build_linux/ - unreal-engine	
https://carla.readthedocs.io/en/0.9.13/build_linux/ - part-two-build-carla	

Table 3: Guides to aid with CARLA installation

Install Ubuntu 20.04 LTS on the computer.

Install the software necessary for building CARLA and Unreal Engine.

Link a GitHub account to an Unreal Engine (Epic Games) account.

Clone Unreal Engine 4.26's source code adapted for CARLA from GitHub and build Unreal Engine.

Clone CARLA's source code from GitHub, setup the environment and build CARLA.

Back up Unreal Engine and CARLA directories to another drive.

Ensure system is up to date by entering command shown in Listing 6 into a terminal and following the terminal's instructions until completion of the update.

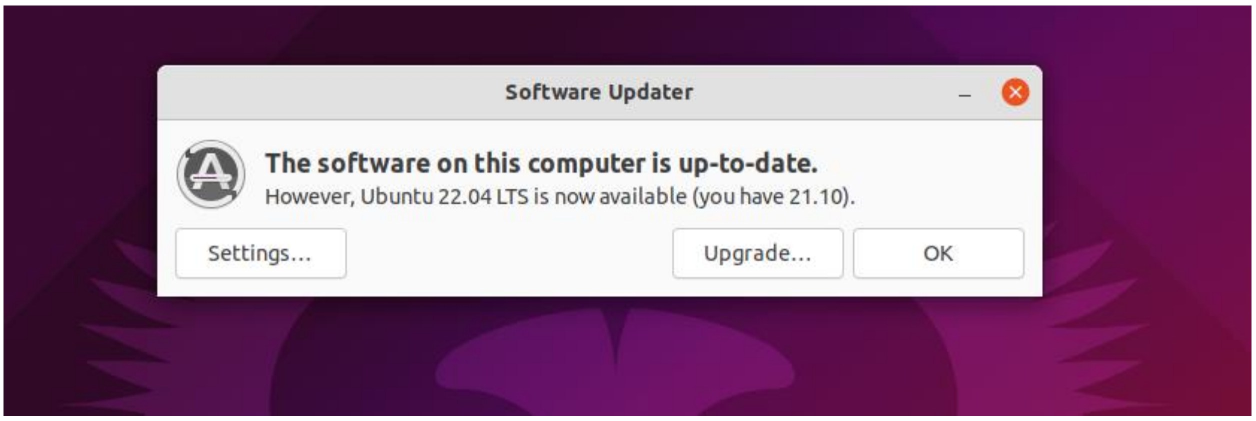
```
sudo apt update && sudo apt upgrade
```

Listing 6: Command to update Ubuntu system

Open the Software Updater from the Ubuntu application menu

If additional updates are offered by the Software Updater, install them

A prompt to upgrade Ubuntu to the latest LTS version (22.04 as of writing this paper) will be presented as seen in Figure 23.


Software Updater

The software on this computer is up-to-date.
However, Ubuntu 22.04 LTS is now available (you have 21.10).

Figure 23: Prompt to upgrade Ubuntu to latest LTS version (22.04)

Follow the instructions by the systems until Ubuntu 22.04 LTS is installed and running

Create a convenience shell script to launch CARLA from the Desktop as shown in Listing 7.

```
#!/bin/bash
echo “Launching CARLA Server”
export carla_dir=<CARLA installation directory>/carla/Util/BuildTools
cd $carla_dir
source BuildCarlaUE4.sh --launch
```

Listing 7: Shell script that launches CARLA from script’s directory

Within a terminal navigate to the Desktop and make the convenience script executable as in Listing 8.

```
sudo chmod +x <name of convenience shell script>.sh
```

Listing 8: Command to make convenience shell script executable

Run convenience shell script and verify the Unreal Engine editor is opened with the CARLA project loaded as seen in Figure 15.

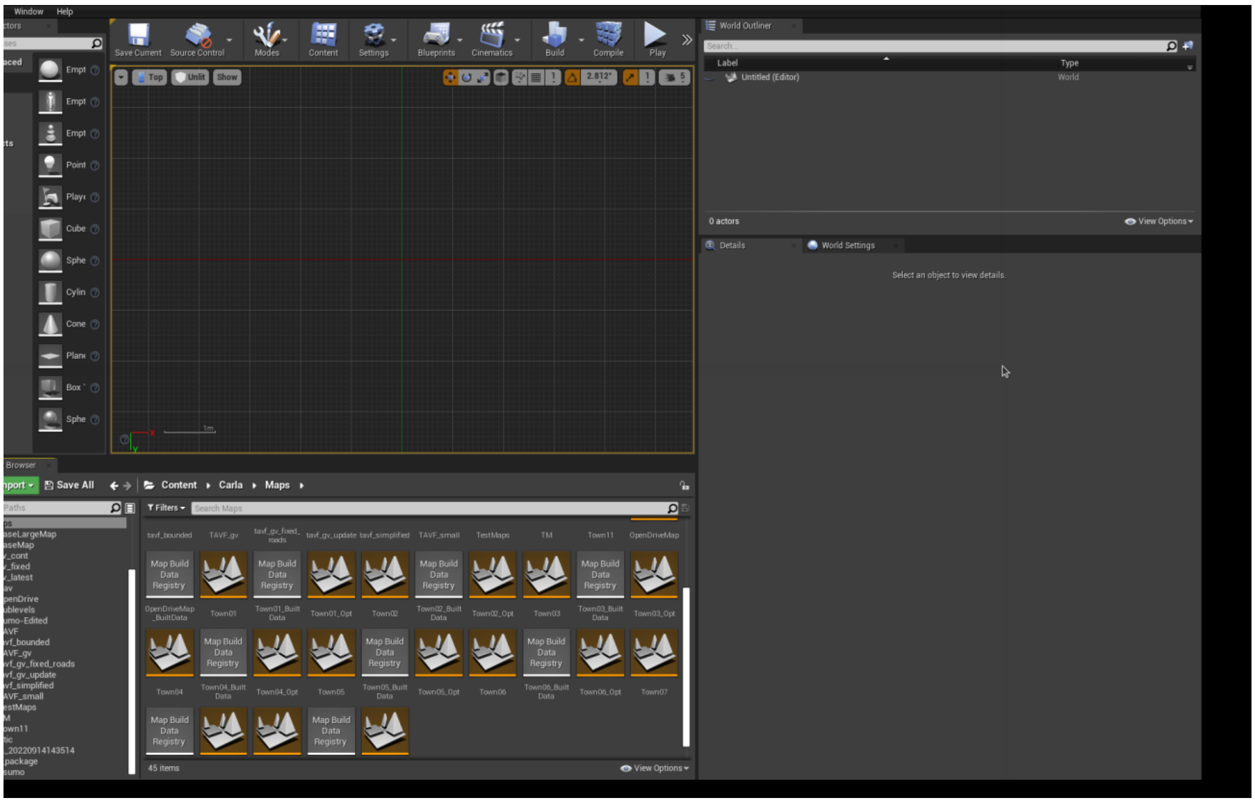


Figure 24: Unreal Engine Editor with CARLA project loaded

11.1.2 Distributed Client

Guide URL	Purpose
https://www.virtualbox.org/wiki/Linux_Downloads	VirtualBox download
https://help.ubuntu.com/community/VirtualBox/SharedFolders	Creation of shared folder with Virtual Machine

Table 4: Collected guides for the purposes of setting up a Virtual Machine

Download the Ubuntu 22.04 VirtualBox package.

Within a terminal navigate to the Downloads folder, then execute the command as shown in Listing 9.

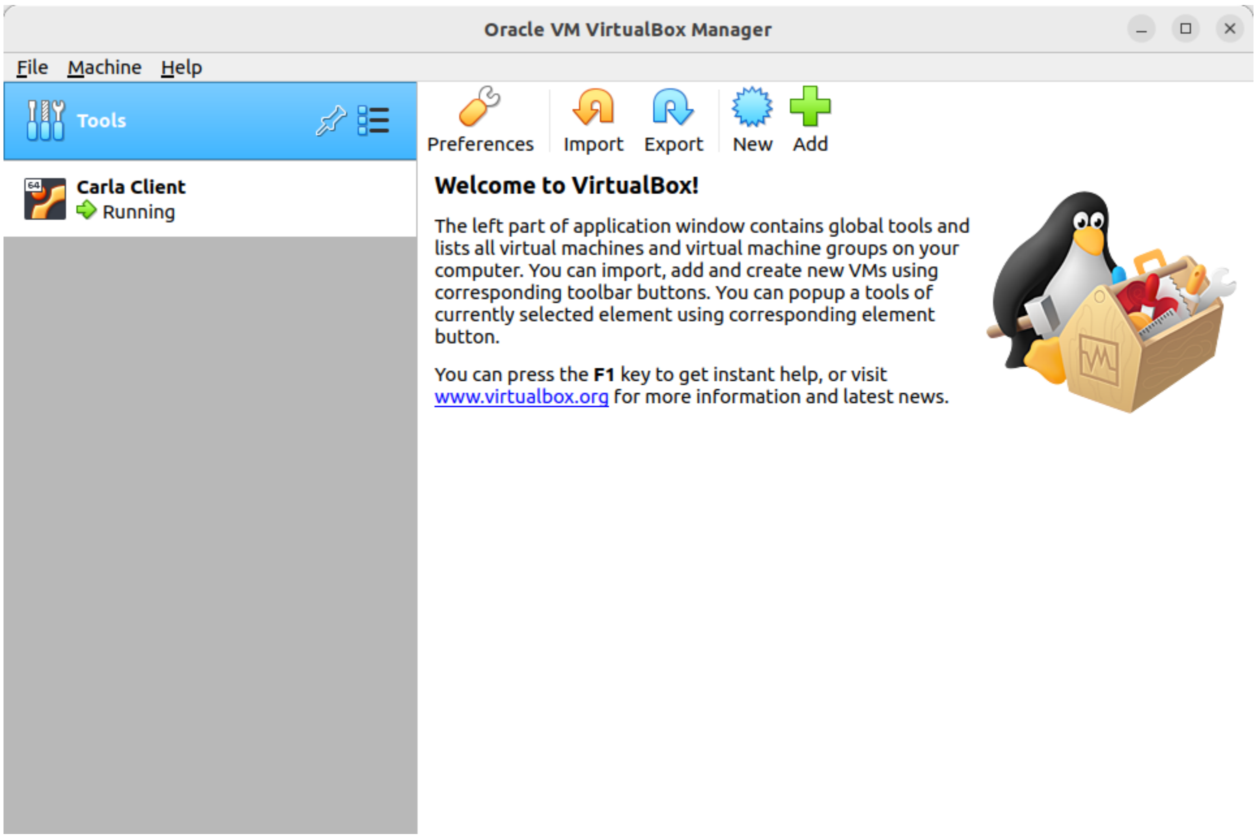
```
sudo apt install ./virtualbox-<package version>.deb
```

Listing 9: Terminal command to install Debian package

Follow instructions as shown in terminal output

Run VirtualBox from the Application Menu

Add a new Virtual Machine from VirtualBox’s start menu by clicking on New in the toolbar shown in Figure 25.



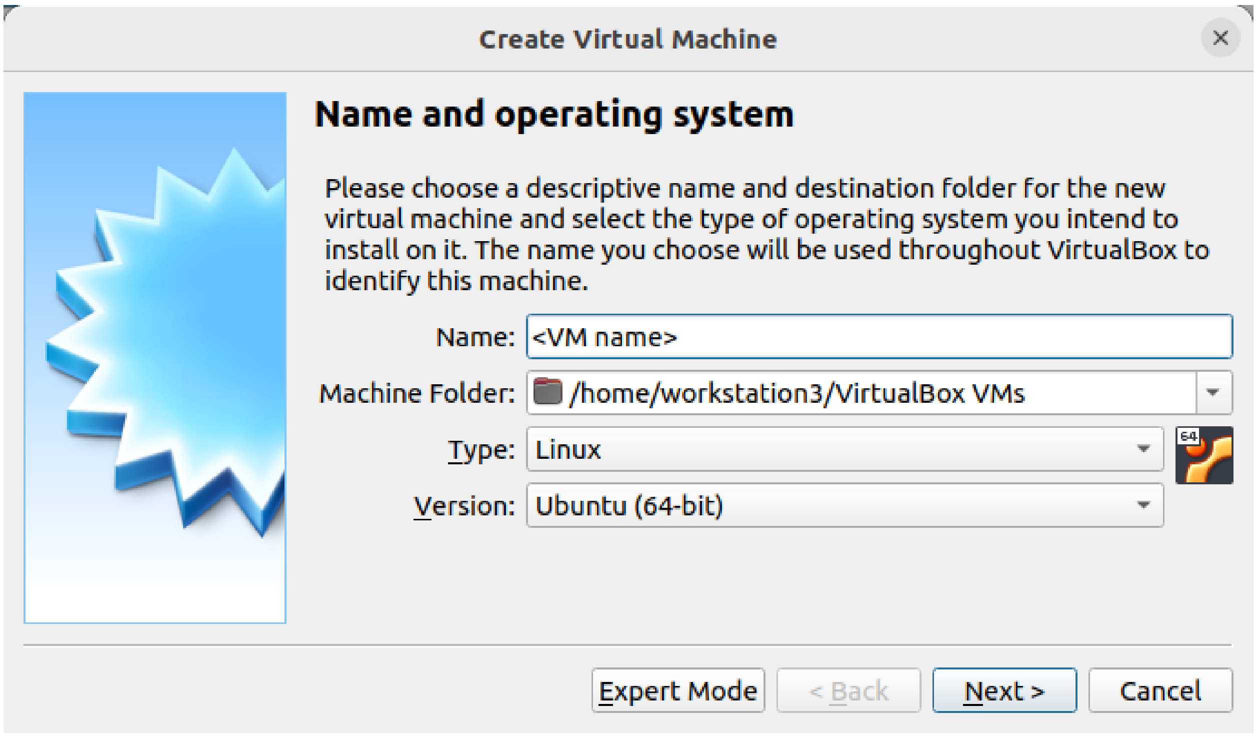
Welcome to VirtualBox!

The left part of application window contains global tools and lists all virtual machines and virtual machine groups on your computer. You can import, add and create new VMs using corresponding toolbar buttons. You can popup a tools of currently selected element using corresponding element button.

You can press the **F1** key to get instant help, or visit www.virtualbox.org for more information and latest news.

Figure 25: VirtualBox Start Menu

In the prompt that follows use the settings as shown in Figure 26, with <VM name> changed to the readers preference


Create Virtual Machine

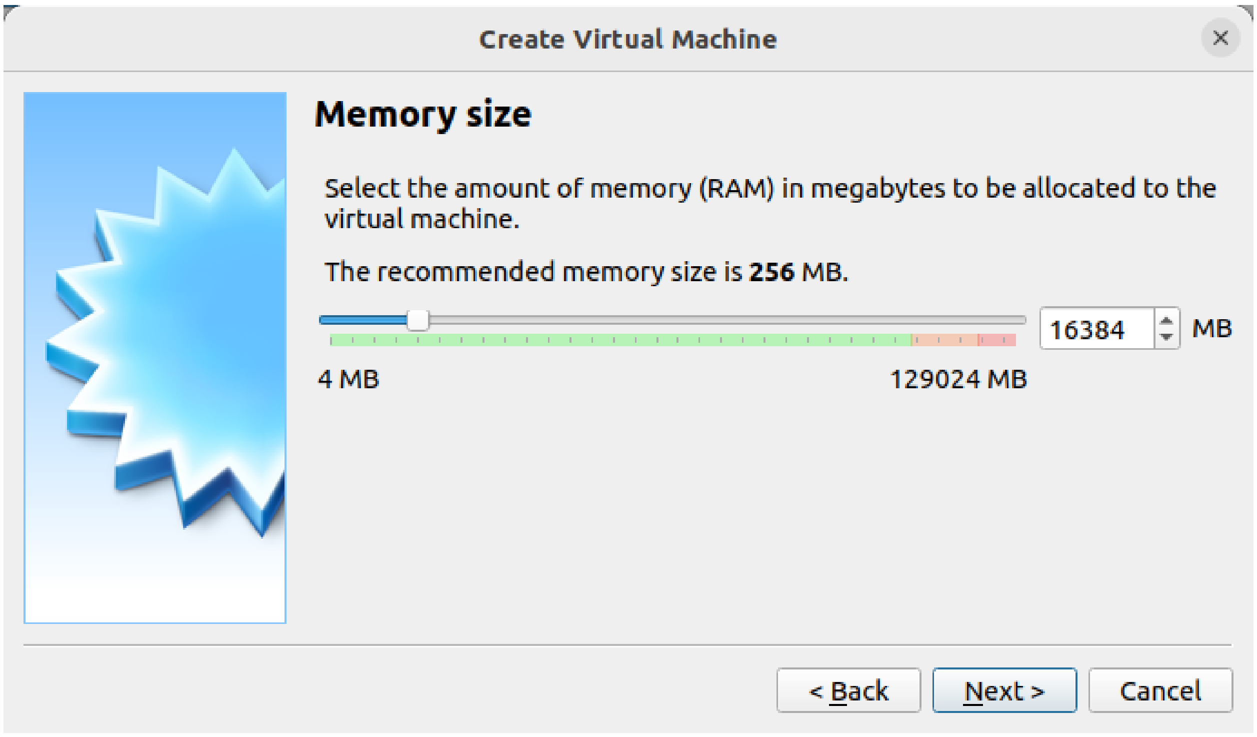
Name and operating system

Please choose a descriptive name and destination folder for the new virtual machine and select the type of operating system you intend to install on it. The name you choose will be used throughout VirtualBox to identify this machine.

Name: <VM name>
Machine Folder: /home/workstation3/VirtualBox VMs
Type: Linux
Version: Ubuntu (64-bit)

Figure 26: Virtual Machine Initial Settings

Choose RAM size according to what is available in the system, author's recommendation is minimum 2GB allocated, prompt to choose RAM size is shown in Figure 27


Create Virtual Machine

Memory size

Select the amount of memory (RAM) in megabytes to be allocated to the virtual machine.

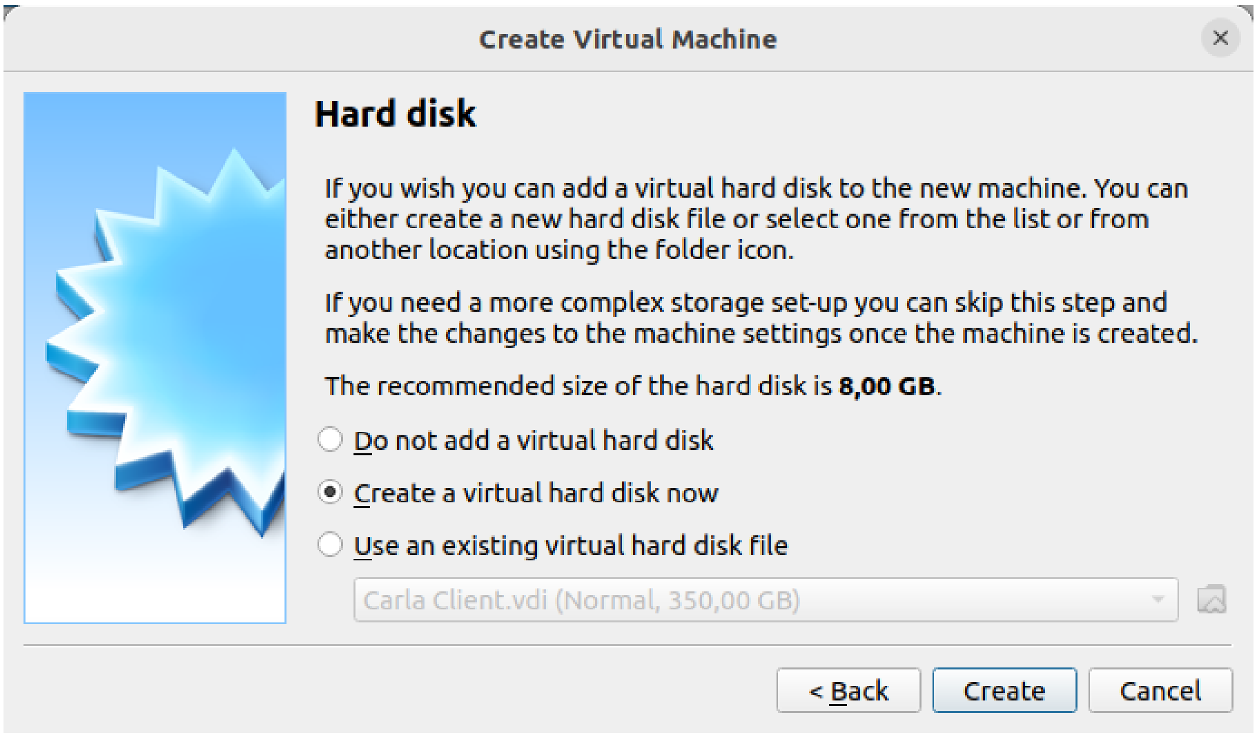
The recommended memory size is **256** MB.

16384 MB

4 MB 129024 MB

Figure 27: Virtual Machine RAM Size Setting

Set up a virtual hard disk to store data as shown in Figures 28, 29, 30 and 31, the amount of hard disk storage can be as small as 20GB


Create Virtual Machine

Hard disk

If you wish you can add a virtual hard disk to the new machine. You can either create a new hard disk file or select one from the list or from another location using the folder icon.

If you need a more complex storage set-up you can skip this step and make the changes to the machine settings once the machine is created.

The recommended size of the hard disk is **8,00 GB**.

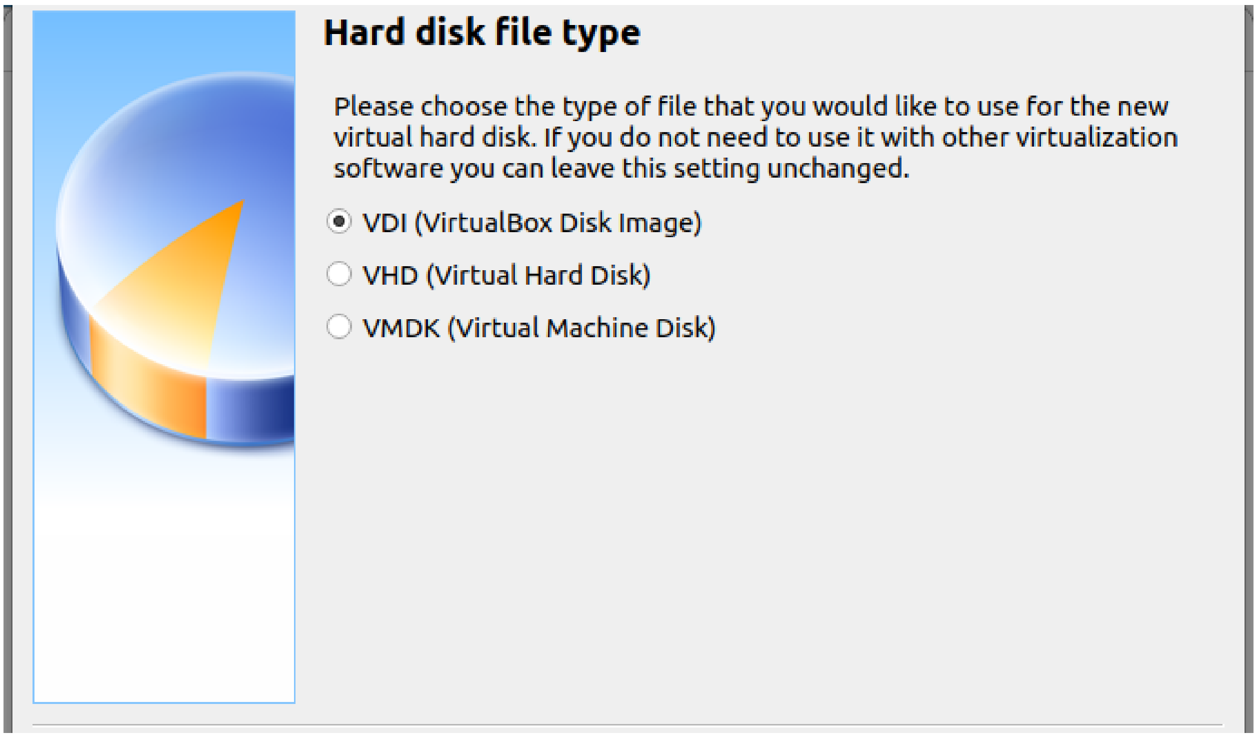
○ Do not add a virtual hard disk

● Create a virtual hard disk now

○ Use an existing virtual hard disk file

Carla Client.vdi (Normal, 350,00 GB)

Figure 28: Virtual Hard Disk Creation


Hard disk file type

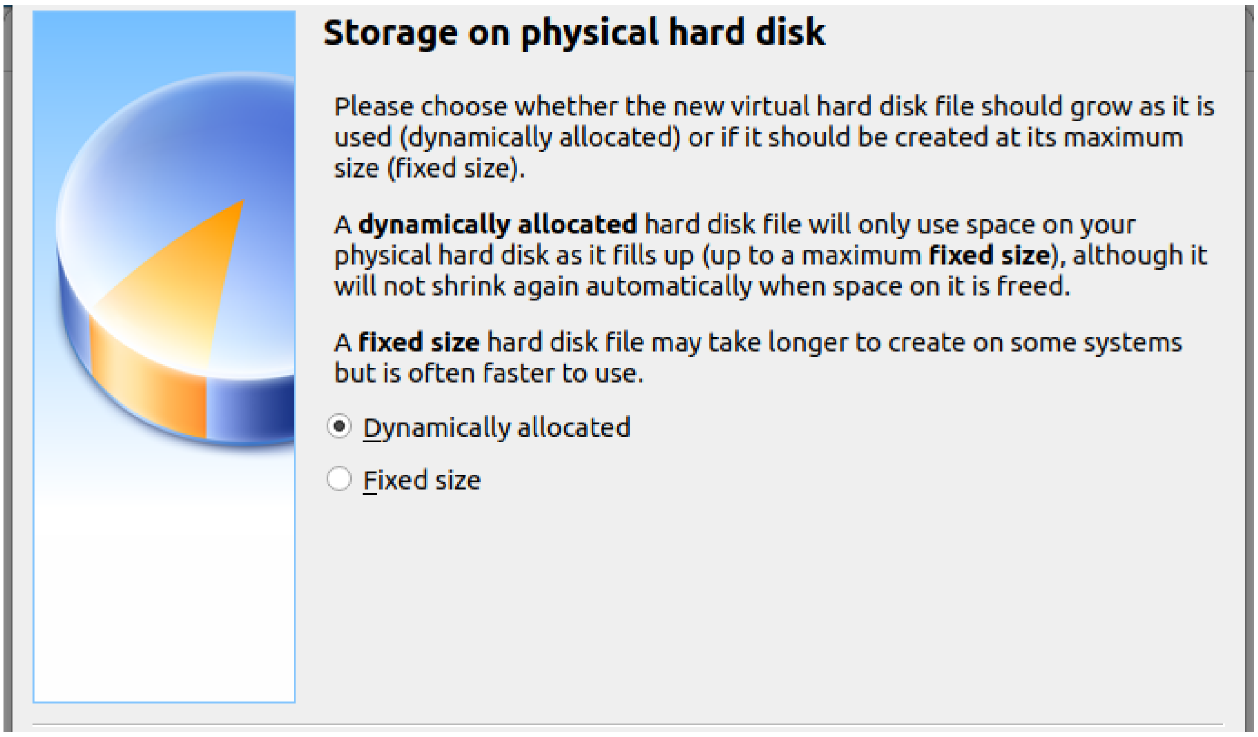
Please choose the type of file that you would like to use for the new virtual hard disk. If you do not need to use it with other virtualization software you can leave this setting unchanged.

● VDI (VirtualBox Disk Image)

○ VHD (Virtual Hard Disk)

○ VMDK (Virtual Machine Disk)

Figure 29: Hard Disk File Type Setting



Storage on physical hard disk

Please choose whether the new virtual hard disk file should grow as it is used (dynamically allocated) or if it should be created at its maximum size (fixed size).

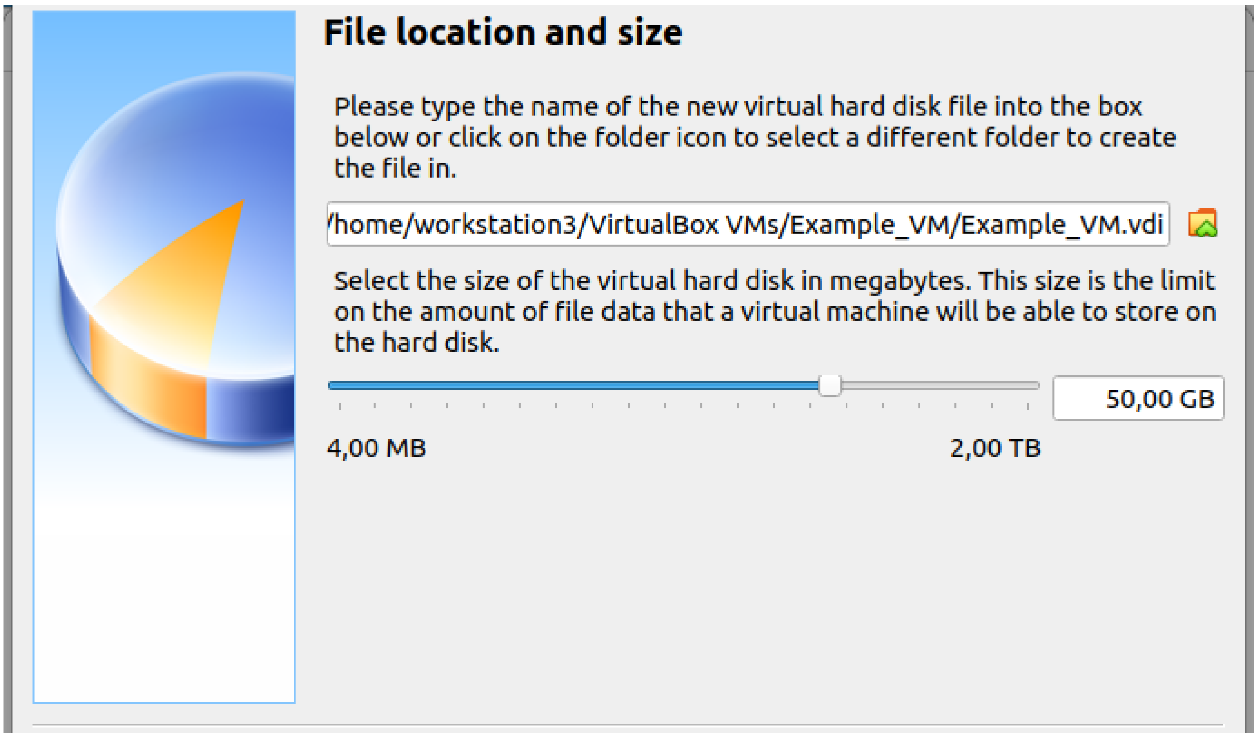
A **dynamically allocated** hard disk file will only use space on your physical hard disk as it fills up (up to a maximum **fixed size**), although it will not shrink again automatically when space on it is freed.

A **fixed size** hard disk file may take longer to create on some systems but is often faster to use.

[✓ Dynamically allocated]

Fixed size

Figure 30: Hard Disk Allocation Setting



File location and size

Please type the name of the new virtual hard disk file into the box below or click on the folder icon to select a different folder to create the file in.

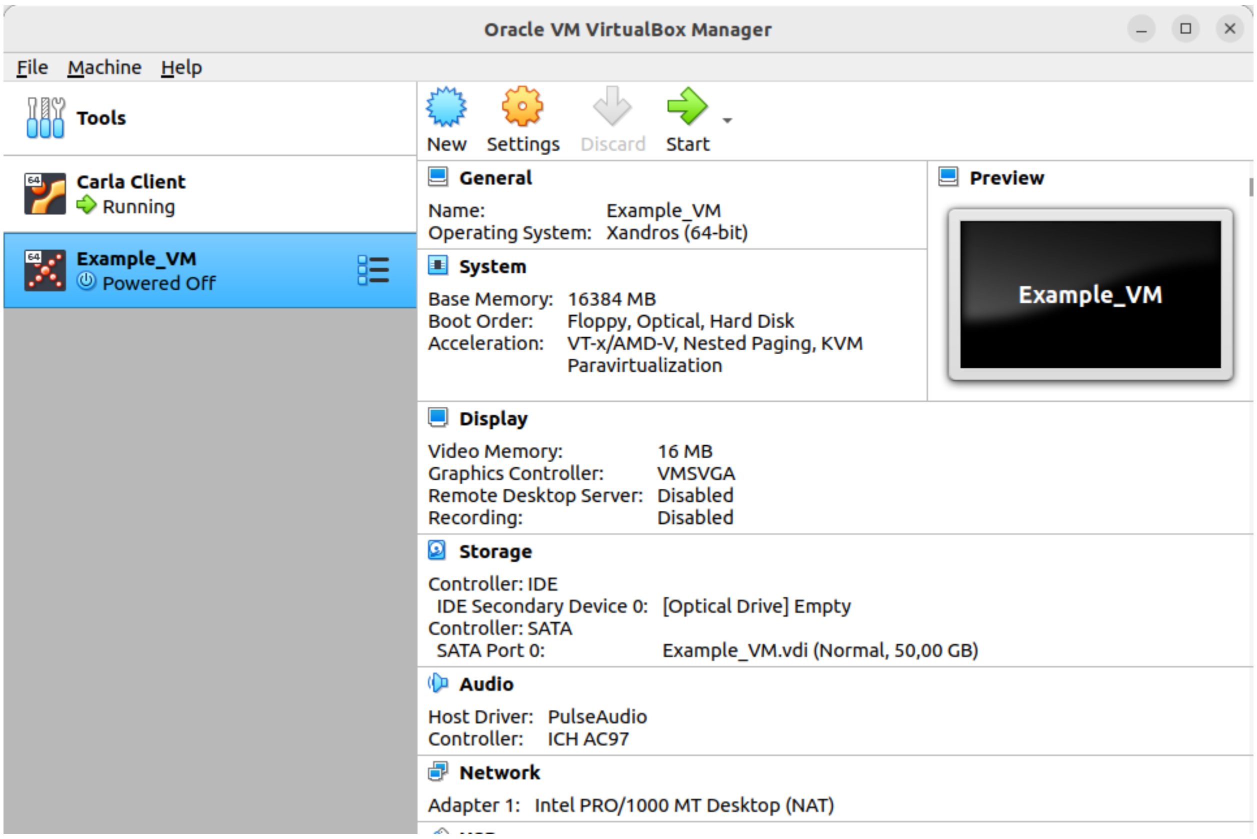
/home/workstation3/VirtualBox VMs/Example_VM/Example_VM.vdi

Select the size of the virtual hard disk in megabytes. This size is the limit on the amount of file data that a virtual machine will be able to store on the hard disk.

50,00 GB

Figure 31: Hard Disk Location and Size Settings

Once all settings have been applied, the new VM will be displayed in the start menu as in Figure 32.



Oracle VM VirtualBox Manager

File Machine Help

Tools

Carla Client
Running

Example_VM
Powered Off

New Settings Discard Start

General

Name: Example_VM
Operating System: Xandros (64-bit)

System

Base Memory: 16384 MB
Boot Order: Floppy, Optical, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, KVM Paravirtualization

Preview

Example_VM

Display

Video Memory: 16 MB
Graphics Controller: VMSVGA
Remote Desktop Server: Disabled
Recording: Disabled

Storage

Controller: IDE
IDE Secondary Device 0: [Optical Drive] Empty
Controller: SATA
SATA Port 0: Example_VM.vdi (Normal, 50,00 GB)

Audio

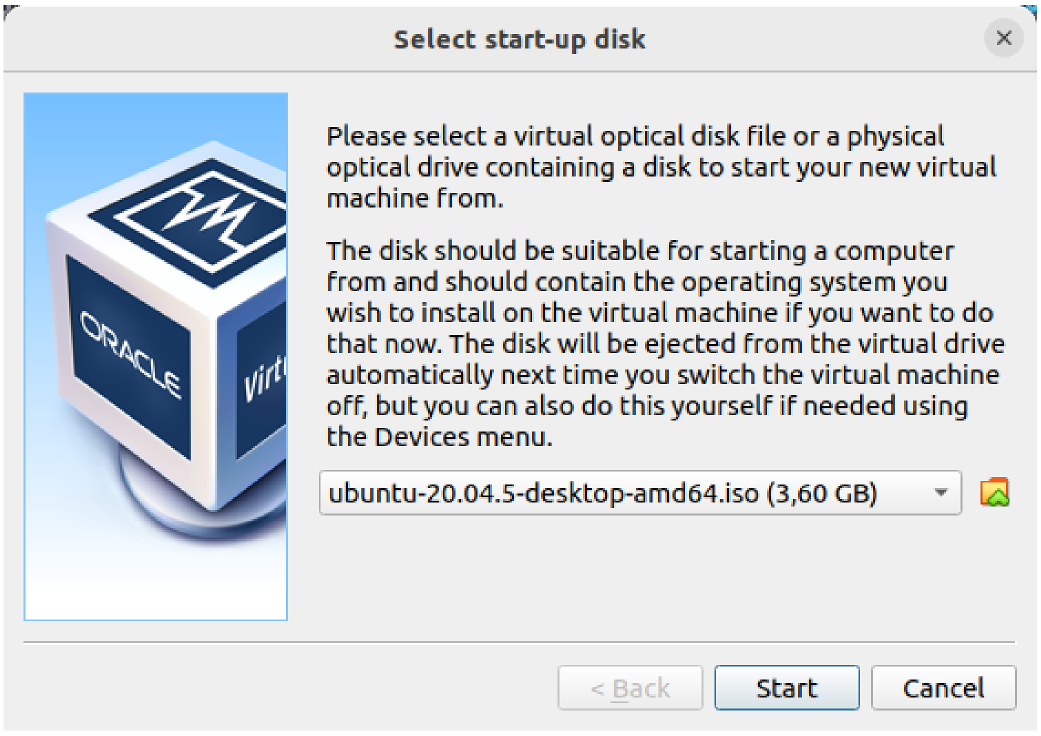
Host Driver: PulseAudio
Controller: ICH AC97

Network

Adapter 1: Intel PRO/1000 MT Desktop (NAT)

Figure 32: New Virtual Machine displayed in VirtualBox Start Menu

By clicking Start the prompt displayed in Figure 33 will be presented where the disk image downloaded in the first step must be chosen


Select start-up disk

Please select a virtual optical disk file or a physical optical drive containing a disk to start your new virtual machine from.

The disk should be suitable for starting a computer from and should contain the operating system you wish to install on the virtual machine if you want to do that now. The disk will be ejected from the virtual drive automatically next time you switch the virtual machine off, but you can also do this yourself if needed using the Devices menu.

ubuntu-20.04.5-desktop-amd64.iso (3,60 GB)

Figure 33: Virtual Machine start-up disk settings

Final step is to follow the installation instructions given by the operating system

Once the VM is running CARLA's Python API needs to be set up as follows:

Copy the PythonAPI folder located under <Carla Installation Directory>/carla/ to the VM via method of choice, the author created a virtual share folder for this purpose.

Install the CARLA Python package by running the command from Listing 10 in the terminal

```
pip3 install carla
```

Listing 10: Terminal command to install CARLA Python package

Navigate to the PythonAPI folder in the terminal and install the packages required for running client scripts with the command from Listing 11.

```
pip3 install -r requirements.txt
```

Listing 11: Terminal command to install Python packages necessary for running client scripts

Verify successful setup by executing one of the example scripts found in <PythonAPI path>/examples/, as shown in Listing 12.

```
python3 manual_control.py
```

Listing 12: Terminal command to run example client script

11.2 Source Code

11.2.1 manager.py

```python
#!/usr/bin/env python
import random
import time
import carla
import rospy
from simulation_setup import Simulation, DisplayManager
from sensors import Sensors
from pid import VehiclePIDController
import pygame
from pygame.locals import K_ESCAPE
from conversion import convert_world_to_sim
from eddy_cloud.msg import sbg_gps_pos_old, eddy_cloud_array
import rosbag
import datetime
import csv

LATENCY_LOG_NAME = f"/home/umlclient/catkin_ws/src/eddy_digital_twin/log/" \
           f"latency_{datetime.datetime.now().strftime('%Y-%m-%d-%H:%M:%S.csv')}"

RESET_LOG_NAME = f"/home/umlclient/catkin_ws/src/eddy_digital_twin/log/" \
           f"reset_{datetime.datetime.now().strftime('%Y-%m-%d-%H:%M:%S.csv')}"


def bag_handler(bag_name):
    try:
        bag = rosbag.Bag(bag_name)
    except ValueError:
        return None
    positions = []
    for topic, msg, t in bag.read_messages(topics='/gps_pos'):
        positions.append(msg.position)

    bag.close()

    return positions


class WaypointManager:
    def __init__(self, wmap, rosbag_path):
        self.rosbag_exists = rosbag_path is not None
        self.reset = False
        self.timer = None

        if self.rosbag_exists:
            self.target_positions = bag_handler(rosbag_path)
            self.waypoint_index = 4200

        self.look_ahead_wp = None
        self.map = wmap
        self.waypoints = self.map.generate_waypoints(1.0)
        self.start_location = self.generate_start_location()
        self.next()

    def generate_start_location(self):
        if self.rosbag_exists:
            latitude = self.target_positions[self.waypoint_index].x
            longitude = self.target_positions[self.waypoint_index].y
            x, y, z = convert_world_to_sim(latitude, longitude)

            location = carla.Location(x / 100, y / 100, 0)
            return location
        else:
            try:
                location = random.choice(self.waypoints)
                self.look_ahead_wp = location.next(50.0)[0]
                # print(self.look_ahead_wp)
                location = location.transform.location

                return location
            except IndexError:
                self.reset = True

    def look_ahead(self):

        if self.rosbag_exists:
            look_ahead_lat = self.target_positions[self.waypoint_index].x
            look_ahead_lon = self.target_positions[self.waypoint_index].y
            look_ahead_x, look_ahead_y, z = convert_world_to_sim(look_ahead_lat, look_ahead_lon)

            look_ahead_loc = carla.Location(look_ahead_x / 100, look_ahead_y / 100, 0)
            possible_waypoints_la = [wp for wp in self.waypoints if wp.transform.location.distance(look_ahead_loc)
< 2.5]
            next_waypoint = min(possible_waypoints_la, key=lambda wp:
wp.transform.location.distance(look_ahead_loc))
        else:
            try:
                next_waypoint_possible = self.look_ahead_wp.next(10.0)
                next_waypoint = min(next_waypoint_possible,
                                    key=lambda wp:
wp.transform.location.distance(self.look_ahead_wp.transform.location)
                                    )
            except (ValueError, AttributeError):
                self.reset = True
                return

        self.look_ahead_wp = next_waypoint

    def next(self):
        if self.rosbag_exists:
            if self.waypoint_index >= len(self.waypoints):
                self.reset = True
            # print(self.waypoint_index)
            self.waypoint_index += 10
        self.look_ahead()


class Manager:
    def __init__(self, host: str, port: int, display: bool, rosbag_path: str, loop_rosbag: bool):
        self.display = display
        # create sim
        self.sim = Simulation(host, port)
        # create waypoints
        self.rosbag_path = rosbag_path
        self.loop_rosbag = loop_rosbag
        self.waypoint_manager = WaypointManager(self.sim.map, self.rosbag_path)
        spawn_location = self.waypoint_manager.start_location
        # create vehicle
        self.tesla = self.create_tesla(spawn_location)
        self.pid = VehiclePIDController(self.tesla, {"K_P": 1.5, "K_D": 0.75, "K_I": 0.25, "dt": 0,
"use_real_time": True},
                                                                  {"K_P": 5.0, "K_D": 4.0, "K_I": 0.5, "dt": 0, "use_real_time":
True})
        # create sensors
        self.sensors = Sensors(self.sim.world, self.tesla)
        if self.display:
            self.display_manager = DisplayManager(self.sensors.display_sensors)
        self.spectator = self.sim.world.get_spectator()
        self.publisher = rospy.Publisher('/gps_pos', sbg_gps_pos_old)
        rospy.Subscriber("/eddy_objects_msg", eddy_cloud_array, self.draw_objects)
        self.timer = 0
        self.reset_timer = 0
        self.time_to_reset = []
        self.response_times = []
        self.received = True

    def create_tesla(self, spawn_location):
        if spawn_location is not None:
            wp = self.sim.map.get_waypoint(spawn_location)
            spawn_transform = wp.transform
            spawn_transform.location.z = 0.5
        else:
            spawn_transform = random.choice(self.sim.map.get_spawn_points())
        vehicle_bp = self.sim.world.get_blueprint_library().filter('model3')[0]
        vehicle_bp.set_attribute('role_name', 'ego_vehicle')

        return self.sim.world.spawn_actor(vehicle_bp, spawn_transform)

    def vehicle_handler(self):
        car_loc = self.tesla.get_location()
        sensor_tf = self.sensors.gnss.get_transform()
        spec_tf = carla.Transform(sensor_tf.location + carla.Location(z=2), sensor_tf.rotation)

        self.spectator.set_transform(spec_tf)

        current_wp = self.waypoint_manager.look_ahead_wp
        # check if car close
        if car_loc.distance(current_wp.transform.location) < 5:
            self.waypoint_manager.next()

        control = self.pid.run_step(30, current_wp)
        self.tesla.apply_control(control)

    def publish_position(self):
        location = self.sensors.location

        if location is None:
            return

        msg = sbg_gps_pos_old()
        msg.position.x = location[0]
        msg.position.y = location[1]
        msg.position.z = location[2]

        # start response timer
        if self.received:
            self.timer = time.time()
        self.publisher.publish(msg)

    def draw_objects(self, msg):
        # end response timer
        end = time.time()

        # log time
        self.response_times.append([str(end), str(end - self.timer)])

        for traffic_sign in msg.objects:
            world_position = traffic_sign.position.position
            position = convert_world_to_sim(world_position.x, world_position.y)
            sign_location = carla.Location(position[0] / 100, position[1] / 100, 1)

            sign_bb = carla.BoundingBox(sign_location, carla.Vector3D(0.5, 0.5, 2))

            self.sim.world.debug.draw_box(
                sign_bb,
                carla.Rotation(0, 0, 0), 0.1, carla.Color(255, 0, 0),
                5)

            self.sim.world.debug.draw_string(
                sign_location + carla.Location(z=3),
                traffic_sign.object_class,
                False,
                carla.Color(255, 0, 0),
                5)
        self.received = True

    def log_response_times(self):
        with open(LATENCY_LOG_NAME, 'w') as log:
            wr = csv.writer(log, delimiter=';')
            wr.writerows(self.response_times)

    def log_reset_times(self):
        with open(RESET_LOG_NAME, 'w') as log:
            wr = csv.writer(log, delimiter=';')
            wr.writerows(self.time_to_reset)

    def destroy(self):
        self.sensors.destroy()
        self.tesla.destroy()
        self.sim.quit()
        self.log_response_times()
        self.log_reset_times()

    def update(self):
        self.sim.world.tick()
        if self.reset_timer == 0:
            self.reset_timer = time.time()
        if self.waypoint_manager.reset or self.tesla.get_location().z < -0.25 or \
                abs(self.tesla.get_transform().rotation.roll) > 5:
            reset_time = time.time() - self.reset_timer
            self.time_to_reset.append((reset_time, 0))
            self.reset_timer = 0
            self.reset_vehicle()
            return
        if self.display:
            for event in pygame.event.get():
                if event.type == pygame.QUIT or (event.type == pygame.KEYDOWN and event.key == K_ESCAPE):
                    raise SystemExit
            self.display_manager.clock.tick()
            self.display_manager.render()
        self.publish_position()
        self.received = False
        self.vehicle_handler()

    def reset_vehicle(self):
        print("RESET")
        self.sensors.destroy()
        self.tesla.destroy()

        time.sleep(3)
        rosbag_path = None
        if self.waypoint_manager.rosbag_exists and self.loop_rosbag:
            rosbag_path = self.rosbag_path

        self.waypoint_manager = WaypointManager(self.sim.map, rosbag_path)

        spawn_location = self.waypoint_manager.start_location

        self.tesla = self.create_tesla(spawn_location)

        self.pid = VehiclePIDController(self.tesla,
                                        {"K_P": 1.5, "K_D": 0.75, "K_I": 0.25, "dt": 0, "use_real_time": True},
                                        {"K_P": 5.0, "K_D": 4.0, "K_I": 0.5, "dt": 0, "use_real_time": True})

        # create sensors
        self.sensors = Sensors(self.sim.world, self.tesla)
        if self.display:
            self.display_manager = DisplayManager(self.sensors.display_sensors)


def main():
    rospy.init_node('sim_manager', anonymous=True)
    display = rospy.get_param("~display", True)
    rosbag_path = rospy.get_param("~rosbag", None)
    loop_rosbag = rospy.get_param("~loop_rosbag", False)
    timer = rospy.get_param("~timer", -1)
    timer_enabled = False if timer == -1 else True
    timer = timer * 60

    manager = None
    try:
        manager = Manager("192.168.10.109", 2000, display, rosbag_path, loop_rosbag)
        if timer_enabled:
            start_time = time.time()
        # Game loop
        while True:
            manager.update()
```

```
                if timer_enabled:
                    if time.time() - start_time >= timer:
                        break
                if rospy.is_shutdown():
                    raise KeyboardInterrupt()
        except KeyboardInterrupt:
            print("Executing console shutdown.")
        finally:
            print("Removing actors.")
            manager.destroy()
            rospy.signal_shutdown("Simulation done")


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

12.1.1 pid.py

```
"""This module implements a longitudinal and lateral controller."""

import math
import time
from collections import deque

import carla
import numpy as np


class Vector3D(object):
    """Represents a 3D vector and provides useful helper functions.
    Args:
        x: The value of the first axis.
        y: The value of the second axis.
        z: The value of the third axis.
    Attributes:
        x: The value of the first axis.
        y: The value of the second axis.
        z: The value of the third axis.
    """
    def __init__(self, x: float = 0, y: float = 0, z: float = 0):
        self.x, self.y, self.z = float(x), float(y), float(z)

    @classmethod
    def from_simulator_vector(cls, vector):
        """Creates a pylot Vector3D from a simulator 3D vector.
        Args:
            vector: An instance of a simulator 3D vector.
        Returns:
            :py:class:`.Vector3D`: A pylot 3D vector.
        """
        from carla import Vector3D
        if not isinstance(vector, Vector3D):
            raise ValueError('The vector must be a Vector3D')
        return cls(vector.x, vector.y, vector.z)

    def as_numpy_array(self):
        """Retrieves the 3D vector as a numpy array."""
        return np.array([self.x, self.y, self.z])

    def as_numpy_array_2D(self):
        """Drops the 3rd dimension."""
        return np.array([self.x, self.y])

    def as_simulator_vector(self):
        """Retrieves the 3D vector as an instance of simulator 3D vector.
        Returns:
            An instance of the simulator class representing the 3D vector.
        """
        from carla import Vector3D
        return Vector3D(self.x, self.y, self.z)

    def l1_distance(self, other):
```

```
    """Calculates the L1 distance between the point and another point.
    Args:
        other (:py:class:`~.Vector3D`): The other vector used to
            calculate the L1 distance to.
    Returns:
        :obj:`float`: The L1 distance between the two points.
    """
    return abs(self.x - other.x) + abs(self.y - other.y) + abs(self.z -
                                                           other.z)

def l2_distance(self, other) -> float:
    """Calculates the L2 distance between the point and another point.
    Args:
        other (:py:class:`~.Vector3D`): The other vector used to
            calculate the L2 distance to.
    Returns:
        :obj:`float`: The L2 distance between the two points.
    """
    vec = np.array([self.x - other.x, self.y - other.y, self.z - other.z])
    return np.linalg.norm(vec)

def magnitude(self):
    """Returns the magnitude of the 3D vector."""
    return np.linalg.norm(self.as_numpy_array())

def to_camera_view(self, extrinsic_matrix, intrinsic_matrix):
    """Converts the given 3D vector to the view of the camera using
    the extrinsic and the intrinsic matrix.
    Args:
        extrinsic_matrix: The extrinsic matrix of the camera.
        intrinsic_matrix: The intrinsic matrix of the camera.
    Returns:
        :py:class:`.Vector3D`: An instance with the coordinates converted
        to the camera view.
    """
    position_vector = np.array([[self.x], [self.y], [self.z], [1.0]])
```

```
        # Transform the points to the camera in 3D.
        transformed_3D_pos = np.dot(np.linalg.inv(extrinsic_matrix),
                                    position_vector)

        # Transform the points to 2D.
        position_2D = np.dot(intrinsic_matrix, transformed_3D_pos[:3])

        # Normalize the 2D points.
        location_2D = type(self)(float(position_2D[0] / position_2D[2]),
                                 float(position_2D[1] / position_2D[2]),
                                 float(position_2D[2]))
        return location_2D

    def rotate(self, angle: float):
        """Rotate the vector by a given angle.
        Args:
            angle (float): The angle to rotate the Vector by (in degrees).
        Returns:
            :py:class:`.Vector3D`: An instance with the coordinates of the
            rotated vector.
        """
        x_ = math.cos(math.radians(angle)) * self.x - math.sin(
            math.radians(angle)) * self.y
        y_ = math.sin(math.radians(angle)) * self.x - math.cos(
            math.radians(angle)) * self.y
        return type(self)(x_, y_, self.z)

    def __add__(self, other):
        """Adds the two vectors together and returns the result."""
        return type(self)(x=self.x + other.x,
                          y=self.y + other.y,
                          z=self.z + other.z)

    def __sub__(self, other):
        """Subtracts the other vector from self and returns the result."""
        return type(self)(x=self.x - other.x,
                          y=self.y - other.y,
                          z=self.z - other.z)

    def __repr__(self):
        return self.__str__()

    def __str__(self):
        return 'Vector3D(x={}, y={}, z={})'.format(self.x, self.y, self.z)


class Location(Vector3D):
    """Stores a 3D location, and provides useful helper methods.
    Args:
        x: The value of the x-axis.
        y: The value of the y-axis.
        z: The value of the z-axis.
    Attributes:
        x: The value of the x-axis.
        y: The value of the y-axis.
        z: The value of the z-axis.
    """
    def __init__(self, x: float = 0, y: float = 0, z: float = 0):
        super(Location, self).__init__(x, y, z)

    @classmethod
    def from_simulator_location(cls, location):
        """Creates a pylot Location from a simulator location.
        Args:
            location: An instance of a simulator location.
        Returns:
            :py:class:`.Location`: A pylot location.
        """
        from carla import Location, Vector3D
        if not (isinstance(location, Location)
                or isinstance(location, Vector3D)):
            raise ValueError('The location must be a Location or Vector3D')
        return cls(location.x, location.y, location.z)
```

```python
@classmethod
def from_gps(cls, latitude: float, longitude: float, altitude: float):
    """Creates Location from GPS (latitude, longitude, altitude).
    This is the inverse of the _location_to_gps method found in
    https://github.com/carla-simulator/scenario_runner/blob/master/srunner/tools/route_manipulation.py
    """
    EARTH_RADIUS_EQUA = 6378137.0
    # The following reference values are applicable for towns 1 through 7,
    # and are taken from the corresponding OpenDrive map files.
    # LAT_REF = 49.0
    # LON_REF = 8.0
    # TODO: Do not hardcode. Get the references from the open drive file.
    LAT_REF = 0.0
    LON_REF = 0.0

    scale = math.cos(LAT_REF * math.pi / 180.0)
    basex = scale * math.pi * EARTH_RADIUS_EQUA / 180.0 * LON_REF
    basey = scale * EARTH_RADIUS_EQUA * math.log(
        math.tan((90.0 + LAT_REF) * math.pi / 360.0))

    x = scale * math.pi * EARTH_RADIUS_EQUA / 180.0 * longitude - basex
    y = scale * EARTH_RADIUS_EQUA * math.log(
        math.tan((90.0 + latitude) * math.pi / 360.0)) - basey

    # This wasn't in the original method, but seems to be necessary.
    y *= -1

    return cls(x, y, altitude)

def distance(self, other) -> float:
    """Calculates the Euclidean distance between the given point and the
    other point.
    Args:
        other (:py:class:`~.Location`): The other location used to
            calculate the Euclidean distance to.
    Returns:
        :obj:`float`: The Euclidean distance between the two points.
        """
        return (self - other).magnitude()


    def as_simulator_location(self):
        """Retrieves the location as a simulator location instance.
        Returns:
            An instance of the simulator class representing the location.
        """
        from carla import Location
        return Location(self.x, self.y, self.z)

    def __repr__(self):
        return self.__str__()

    def __str__(self):
        return 'Location(x={}, y={}, z={})'.format(self.x, self.y, self.z)


class PIDLongitudinalController(object):
    """Implements longitudinal control using a PID.
    Args:
       K_P (:obj:`float`): Proportional term.
       K_D (:obj:`float`): Differential term.
       K_I (:obj:`float`): Integral term.
       dt (:obj:`float`): time differential in seconds.
    """
    def __init__(self,
                 K_P: float,
                 K_D: float,
                 K_I: float,
                 dt: float = 0.03,
                 use_real_time: bool = False):
        self._k_p = K_P
        self._k_d = K_D
        self._k_i = K_I
        self._dt = dt
        self._use_real_time = use_real_time
        self._last_time = time.time()
        self._error_buffer = deque(maxlen=10)

    def run_step(self, target_speed: float, current_speed: float):
        """Computes the throttle/brake based on the PID equations.
        Args:
            target_speed (:obj:`float`): Target speed in m/s.
            current_speed (:obj:`float`): Current speed in m/s.
        Returns:
            Throttle and brake values.
        """
        # Transform to km/h
        error = (target_speed - current_speed) * 3.6
        self._error_buffer.append(error)

        if self._use_real_time:
            time_now = time.time()
            dt = time_now - self._last_time
            self._last_time = time_now
        else:
            dt = self._dt
        if len(self._error_buffer) >= 2:
            _de = (self._error_buffer[-1] - self._error_buffer[-2]) / dt
            _ie = sum(self._error_buffer) * dt
        else:
            _de = 0.0
            _ie = 0.0

        return np.clip(
            (self._k_p * error) + (self._k_d * _de) + (self._k_i * _ie), -1.0,
            1.0)


class PIDLateralController(object):
    """Implements lateral control using a PID.
    Args:
        K_P (:obj:`float`): Proportional term.
        K_D (:obj:`float`): Differential term.
        K_I (:obj:`float`): Integral term.
        dt (:obj:`float`): time differential in seconds.
    """
    def __init__(self,
                 K_P: float = 1.0,
                 K_D: float = 0.0,
                 K_I: float = 0.0,
                 dt: float = 0.03,
                 use_real_time: bool = False):
        self._k_p = K_P
        self._k_d = K_D
        self._k_i = K_I
        self._dt = dt
        self._use_real_time = use_real_time
        self._last_time = time.time()
        self._e_buffer = deque(maxlen=10)

    def run_step(self, waypoint, vehicle_transform):
        v_begin = Location(vehicle_transform.location.x, vehicle_transform.location.y, vehicle_transform.location.z)
        v_end = v_begin + Location(
            x=math.cos(math.radians(vehicle_transform.rotation.yaw)),
            y=math.sin(math.radians(vehicle_transform.rotation.yaw)))

        v_vec = np.array([v_end.x - v_begin.x, v_end.y - v_begin.y, 0.0])
        w_vec = np.array([
            waypoint.location.x - v_begin.x, waypoint.location.y - v_begin.y,
            0.0
        ])
        _dot = math.acos(
            np.clip(
                np.dot(w_vec, v_vec) /
                (np.linalg.norm(w_vec) * np.linalg.norm(v_vec)), -1.0, 1.0))

        _cross = np.cross(v_vec, w_vec)
        if _cross[2] < 0:
            _dot *= -1.0

        if self._use_real_time:
            time_now = time.time()
            dt = time_now - self._last_time
            self._last_time = time_now
        else:
            dt = self._dt

        self._e_buffer.append(_dot)
        if len(self._e_buffer) >= 2:
            _de = (self._e_buffer[-1] - self._e_buffer[-2]) / dt
            _ie = sum(self._e_buffer) * dt
        else:
            _de = 0.0
            _ie = 0.0

        return np.clip(
            (self._k_p * _dot) + (self._k_d * _de) + (self._k_i * _ie), -1.0,
            1.0)


class VehiclePIDController:
    """
    VehiclePIDController is the combination of two PID controllers
    (lateral and longitudinal) to perform the
    low level control a vehicle from client side
    """

    def __init__(self, vehicle, args_lateral, args_longitudinal, offset=0, max_throttle=0.75, max_brake=0.3,
                 max_steering=0.8):
        """
        Constructor method.

        :param vehicle: actor to apply to local planner logic onto
        :param args_lateral: dictionary of arguments to set the lateral PID controller
        using the following semantics:
            K_P -- Proportional term
            K_D -- Differential term
            K_I -- Integral term
        :param args_longitudinal: dictionary of arguments to set the longitudinal
        PID controller using the following semantics:
            K_P -- Proportional term
            K_D -- Differential term
            K_I -- Integral term
        :param offset: If different than zero, the vehicle will drive displaced from the center line.
        Positive values imply a right offset while negative ones mean a left one. Numbers high enough
        to cause the vehicle to drive through other lanes might break the controller.
        """

        self.max_brake = max_brake
        self.max_throt = max_throttle
        self.max_steer = max_steering

        self._vehicle = vehicle
        #self._world = self._vehicle.get_world()
        self.past_steering = self._vehicle.get_control().steer
        self._lon_controller = PIDLongitudinalController(**args_longitudinal)
        self._lat_controller = PIDLateralController(**args_lateral)

    def run_step(self, target_speed, waypoint):
        """
        Execute one step of control invoking both lateral and longitudinal
        PID controllers to reach a target waypoint
        at a given target_speed.

            :param target_speed: desired vehicle speed
            :param waypoint: target location encoded as a waypoint
            :return: distance (in meters) to the waypoint
        """

        speed = self._vehicle.get_velocity()
        current_speed = 3.6 * math.sqrt(speed.x ** 2 + speed.y ** 2 + speed.z ** 2)
```

```
        acceleration = self._lon_controller.run_step(target_speed, current_speed)
        current_steering = self._lat_controller.run_step(waypoint.transform, self._vehicle.get_transform())
        control = carla.VehicleControl()
        if acceleration >= 0.0:
            control.throttle = min(acceleration, self.max_throt)
            control.brake = 0.0
        else:
            control.throttle = 0.0
            control.brake = min(abs(acceleration), self.max_brake)

        # Steering regulation: changes cannot happen abruptly, can't steer too much.

        if current_steering > self.past_steering + 0.1:
            current_steering = self.past_steering + 0.1
        elif current_steering < self.past_steering - 0.1:
            current_steering = self.past_steering - 0.1

        if current_steering >= 0:
            steering = min(self.max_steer, current_steering)
        else:
            steering = max(-self.max_steer, current_steering)

        control.steer = steering
        control.hand_brake = False
        control.manual_gear_shift = False
        self.past_steering = steering
        return control
```

12.1.2 sensors.py

```
import carla


class Sensors:
```

```python
def __init__(self, world, tesla):
    self.world = world
    self.tesla = tesla
    self.display_sensors = self.create_display_sensors()
    self.gnss = self.create_gnss()
    self.location = None
    self.gnss.listen(self.__gnss_data)

def create_display_sensors(self):
    # camera setup
    yaws = [0, 216, 288, 0, 72, 144]  # top, back left, front left, front, front right, back right
    pitches = [90, 0, 0, 0, 0, 0]
    cameras = []
    camera_bp = self.world.get_blueprint_library().find('sensor.camera.rgb')
    camera_bp.set_attribute('image_size_x', '320')
    camera_bp.set_attribute('image_size_y', '240')
    camera_bp.set_attribute('focal_distance', '0.48')

    for yaw, pitch in zip(yaws, pitches):
        position = carla.Transform(carla.Location(z=2), carla.Rotation(yaw=yaw, pitch=pitch))
        cameras.append(self.world.spawn_actor(camera_bp, position, attach_to=self.tesla), )

    # setup lidar
    lidar_bp = self.world.get_blueprint_library().find('sensor.lidar.ray_cast')
    lidar_bp.set_attribute('rotation_frequency', '20.0')
    lidar_bp.set_attribute('points_per_second', '1390000')
    lidar_bp.set_attribute('range', '100')
    lidar_bp.set_attribute('horizontal_fov', '360')
    lidar_bp.set_attribute('channels', '32')

    position = carla.Transform(carla.Location(z=1.55, y=0.5), carla.Rotation())
    lidar = self.world.spawn_actor(lidar_bp, position, attach_to=self.tesla)

    # setup depth cam
    depth_cam_bp = self.world.get_blueprint_library().find('sensor.camera.depth')
    depth_cam_bp.set_attribute('image_size_x', '320')
    depth_cam_bp.set_attribute('image_size_y', '240')
```

```
        depth_cam_bp.set_attribute('fov', '110')

        position = carla.Transform(carla.Location(z=1.50, y=0.5))
        depth_cam = self.world.spawn_actor(depth_cam_bp, position, attach_to=self.tesla)

        return cameras, lidar, depth_cam

    def create_gnss(self):
        gnss_bp = self.world.get_blueprint_library().find('sensor.other.gnss')
        position = carla.Transform(carla.Location())
        gnss = self.world.spawn_actor(gnss_bp, position, attach_to=self.tesla)

        return gnss

    def __gnss_data(self, data):
        if data.frame % 10 == 0:
            self.location = (data.latitude, data.longitude, data.altitude)

    def destroy(self):
        for sensor in self.display_sensors:
            if type(sensor) == list:
                for camera in sensor:
                    camera.destroy()
            else:
                sensor.destroy()

        self.gnss.destroy()
```

12.1.3 simulation_setup.py

```
import carla
import numpy as np
import pygame
```

```
class Simulation:
    def __init__(self, host: str, port: int):
        self.client = carla.Client(host, port)
        self.client.set_timeout(5.0)
        self.world = self.client.get_world()

        self.old_settings = settings = self.world.get_settings()

        traffic_manager = self.client.get_trafficmanager(10000)
        traffic_manager.set_synchronous_mode(True)
        settings.synchronous_mode = True
        settings.fixed_delta_seconds = 0.05
        self.world.apply_settings(settings)

        self.map = self.world.get_map()
        print("done with map")

    def quit(self):
        self.client.reload_world()


class DisplayManager:
    def __init__(self, sensors):
        self.surface = None
        self.window_x = 1600
        self.window_y = 450
        self.sensors = sensors
        self.display = pygame.display.set_mode((self.window_x, self.window_y), pygame.HWSURFACE | pygame.DOUBLEBUF)
        self.display.fill((255, 255, 255))
        self.__init_listeners()

        pygame.init()
        pygame.font.init()
        self.clock = pygame.time.Clock()
        pygame.display.flip()
```

```python
class LadybugHandler:
    def __init__(self, lb_display, lb_camera, index):
        self.display = lb_display
        self.camera = lb_camera
        self.offset = self.__get_offset(index)
        self.surface = None
        self.init_listener()

    def init_listener(self):
        self.camera.listen(self.create_surface)

    def create_surface(self, image):
        image.convert(carla.ColorConverter.Raw)
        array = np.frombuffer(image.raw_data, dtype=np.dtype("uint8"))
        array = np.reshape(array, (image.height, image.width, 4))
        array = array[:, :, :3]
        array = array[:, :, ::-1]

        surface = pygame.surfarray.make_surface(array.swapaxes(0, 1))

        if surface is not None:
            rect = self.display.blit(surface, self.offset)
            pygame.display.update(rect)

    def __get_offset(self, index):
        offset_width = [320, 240]
        offset_factors = [[2, 0], [0, 1], [1, 1], [2, 1], [3, 1], [4, 1]]

        offset_factor = offset_factors[index]
        return offset_factor[0] * offset_width[0], offset_factor[1] * offset_width[1]

def __init_listeners(self):
    cameras, lidar, depth_cam = self.sensors

    for ladybug_cam in cameras:
        index = cameras.index(ladybug_cam)
```

```
            self.LadybugHandler(self.display, ladybug_cam, index)

        depth_cam.listen(self.depth_cam_handler)
        lidar.listen(self.lidar_handler)

    def lidar_handler(self, data):
        disp_size = [320, 240]
        lidar_range = 2.0 * float("100")

        points = np.frombuffer(data.raw_data, dtype=np.dtype('f4'))
        points = np.reshape(points, (int(points.shape[0] / 4), 4))

        lidar_data = np.array(points[:, :2])
        lidar_data *= min(disp_size) / lidar_range
        lidar_data += (0.5 * disp_size[0], 0.5 * disp_size[1])
        lidar_data = np.fabs(lidar_data)
        lidar_data = lidar_data.astype(np.int32)
        lidar_data = np.reshape(lidar_data, (-1, 2))
        lidar_img_size = (disp_size[0], disp_size[1], 3)
        lidar_img = np.zeros(lidar_img_size, dtype=np.uint8)

        lidar_img[tuple(lidar_data.T)] = (255, 255, 255)

        surface = pygame.surfarray.make_surface(lidar_img)

        if surface is not None:
            rect = self.display.blit(surface, (320, 0))
            pygame.display.update(rect)

    def depth_cam_handler(self, image):
        image.convert(carla.ColorConverter.Raw)
        image.convert(carla.ColorConverter.LogarithmicDepth)
        array = np.frombuffer(image.raw_data, dtype=np.dtype("uint8"))
        array = np.reshape(array, (image.height, image.width, 4))
        array = array[:, :, :3]

        def grayscale_image(image_rgba):
            # turning RGBA-Surface to Grayscale-Surface
            width, height = image_rgba.get_size()
            for x in range(width):
                for y in range(height):
                    red, green, blue, aplha = image_rgba.get_at((x, y))
                    L = (red + green + blue) // 3
                    gs_color = (L, L, L)
                    image_rgba.set_at((x, y), gs_color)
            return image_rgba

        surface = pygame.surfarray.make_surface(array.swapaxes(0, 1))
        surface_gray = grayscale_image(surface)

        if surface_gray is not None:
            rect = self.display.blit(surface_gray, (3 * 320, 0))
            pygame.display.update(rect)

    def render(self):
        pygame.display.update()
```

12.1.4 conversion.py

```
import math


def convert_world_to_sim(lat, lon):
    EARTH_RADIUS_EQUA = 637813700.
    LAT_REF, LON_REF = 53.55417195412835, 9.979838256781566

    # LAT_REF, LON_REF =0.01162190670750506, 4.521712683426776
    latitude, longitude = lat, lon

    scale = math.cos(LAT_REF * math.pi / 180.0)
    basex = scale * math.pi * EARTH_RADIUS_EQUA / 180.0 * LON_REF
    basey = scale * EARTH_RADIUS_EQUA * math.log(
        math.tan((90.0 + LAT_REF) * math.pi / 360.0))

    x = scale * math.pi * EARTH_RADIUS_EQUA / 180.0 * longitude - basex
    y = scale * EARTH_RADIUS_EQUA * math.log(
        math.tan((90.0 + latitude) * math.pi / 360.0)) - basey

    # This wasn't in the original carla method, but seems to be necessary.
    y *= -1

    return x, y, 0
```

12.1.5 eddy_digital_twin.launch

```
<!-- -->
<launch>
  <!-- launch a complete carla-ros-environment with an ad agent that steers the ego-vehicle -->
    <!-- Simulation manager -->
    <param name="display" type="bool"/>
    <param name="rosbag" type="str"/>
    <param name="loop_rosbag" type="bool"/>
    <param name="timer" type="float"/>


  <node pkg="eddy_digital_twin" type="manager_new.py" name="eddy_digital_twin_manager">
  </node>

  <!--<include file="$(find carla_spawn_objects)/launch/carla_spawn_objects.launch">
    <arg name="objects_definition_file" value='$(find carla_spawn_objects)/config/objects.json'/>
    <arg name="spawn_sensors_only" value="True" />
  </include>-->

</launch>
```

List of Figures

Figure 1: TAVF [13] 6

Figure 2: a) Classical control experimental setup; b) Hardware-in-the-loop method, adapted from [14] 7

Figure 3: Precision, time, and cost trade off in computer simulation, HiL simulation and actual situation [15] 7

Figure 4: CARLA simulating an urban environment in different weather conditions [16] 8

Figure 5: LDM and its four layers [18]... 9

Figure 6: Nodes and topics (in ellipses and rectangles, respectively) [27] 10

Figure 7: Number of publications on use of HiL for AD per year in Scopus database 11

Figure 8: Number of publications per year in Scopus database, CARLA vs. CARLA and HiL 12

Figure 9: Diagram displaying the Digital Twin functionalities and communications....... 16

Figure 10: a) Urban Mobility Lab's Upgraded Tesla Model S [20]; 18

Figure 11: Communication flow of the digital twin... 19

Figure 12: Custom map generation and ingestion in CARLA sources left to right: [45], [46], [47], [48] 19

Figure 13: OSM map view showing the highlighted TAVF to be extracted 20

Figure 14: TAVF OpenDRIVE road network supplied by LGV imported in RoadRunner .. 21

Figure 15: TAVF OpenDRIVE generated from OSM data 22

Figure 16: Sensor Management and Display Diagram 23

Figure 17: Display organization of provided sensor management system, [45].... 23

Figure 18: Display organization of used sensor management system, top row: blank, LiDAR visualization, Top Camera, Depth Camera; bottom row: Back-Left Camera, Front-Left Camera, Front Camera, Front-Right Camera, Back-Right Camera24

Figure 19: RQT Graph displaying the ROS nodes of communication between Digital Twin and UDM 25

Figure 20: Distance between recorded positions and simulated waypoints.......... 27

Figure 21: Traffic Signs returned from the UDM displayed in the simulation 28

Figure 22: Latencies of Cloud2ROS script (orange) and Digital Twin client script (blue) for communication with UDM.. 29

Figure 23: Prompt to upgrade Ubuntu to latest LTS version (22.04)..................... 33

Figure 24: Unreal Engine Editor with CARLA project loaded................................ 34

Figure 25: VirtualBox Start Menu.. 35

Figure 26: Virtual Machine Initial Settings.. 36

Figure 27: Virtual Machine RAM Size Setting.. 36

Figure 28: Virtual Hard Disk Creation 37

Figure 29: Hard Disk File Type Setting 37

Figure 30: Hard Disk Allocation Setting 38

Figure 31: Hard Disk Location and Size Settings 38

Figure 32: New Virtual Machine displayed in VirtualBox Start Menu.................... 39

Figure 33: Virtual Machine start-up disk settings.. 40

List of Tables

Table 1: Hardware Requirements for Unreal Engine 4.26 + CARLA 14

Table 2: Latency statistics for Cloud2ROS and Digital Twin scripts 29

Table 3: Guides to aid with CARLA installation 32

Table 4: Collected guides for the purposes of setting up a Virtual Machine 34

List of Listings

Listing 1: Terminal command to launch CARLA 15

Listing 2: Launch file used for managing the simulation 17

Listing 3: Terminal command to install SUMO 20

Listing 4: Terminal command to convert from OSM to OpenDRIVE using netconvert 20

Listing 5: Algorithm for path planning and following... 24

Listing 6: Command to update Ubuntu system 32

Listing 7: Shell script that launches CARLA from script's directory....................... 33

Listing 8: Command to make convenience shell script executable....................... 33

Listing 9: Terminal command to install Debian package....................................... 34

Listing 10: Terminal command to install CARLA Python package 40

Listing 11: Terminal command to install Python packages necessary for running client scripts40

Listing 12: Terminal command to run example client script 40

Bibliography

[1] CB Insights, „Autonomous Driverless Vehicles Corporations List,“ 16 December 2020. [Online]. Available: https://www.cbinsights.com/research/autonomous-driverless-vehicles-corporations-list/. [Zugriff am 20 September 2022].

[2] Bundesministerium für Digitales und Verkehr, „European Digital Dynamic Mapping - "EDDY",“ 2 November 2021. [Online]. Available: https://bmdv.bund.de/SharedDocs/DE/Artikel/DG/mfund-projekte/eddy.html. [Zugriff am 20 September 2022].

[3] European Telecommunications Standards Institute, *Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Local Dynamic Map (LDM),* European Telecommunications Standards Institute, 2014.

[4] H.-J. Günther, „Collective Perception in Vehicular Ad-hoc Networks,“ 2018.

[5] C. Brogle, C. Zhang, K. Li Lim und T. Bräunl, „Hardware-in-the-Loop Autonomous Driving Simulation Without Real-Time Constraints,“ *IEEE Trans. Intell. Veh,* Bd. 4, Nr. 3, pp. 375-384, 2019.

[6] N. Brayanov und A. Stoynova, „Review of hardware-in-the-loop – a hundred years progress in the pseudo-real testing,“ Bd. 54, Nr. 3-4, pp. 70-84, 2019.

[7] N. Vignard, A. Bolovinou, A. Amditis, G. Wallraf, O. Ur-Rehman, M. Kremer, A. Milingal Ziyad, A. T. Sheik, U.-I. Atmaca, M. Dianati, V. Mayr, P. Borodani, J.-F. Grönvall, A. Van Vliet, A. Rahman, Y. Page und T. Gasser, „Legal Requirements for AD Piloting and Cyber Security Analysis,“ 2019.

[8] M. Grieves und J. Vickers, „Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems,“ in *Transdisciplinary Perspectives on Complex Systems: New Findings and Approaches*, Springer International Publishing Switzerland, 2017, pp. 85-113.

[9] Geschäftsstelle der Teststrecke für automatisiertes und vernetztes Fahren Hamburg c/o ITS mobility e. V., „Homepage,“ [Online]. Available: https://tavf.hamburg/.

[10] ETSI, „Intelligent Transport Systems; Access layer specification for Intelligent Transport Systems operating in the 5 GHz frequency band,“ ETSI, 2019.

[11] Geschäftsstelle der Teststrecke für automatisiertes und vernetztes Fahren Hamburg c/o ITS mobility e. V., „TAVF Fact Sheet,“ Hamburg, 2022.

[12] Geschäftsstelle der Teststrecke für automatisiertes und vernetztes Fahren Hamburg c/o ITS mobility e. V., „Referenzen,“ [Online]. Available: https://tavf.hamburg/referenzen. [Zugriff am 20 September 2022].

[13] Geschäftsstelle der Teststrecke für automatisiertes und vernetztes Fahren Hamburg c/o ITS mobility e. V., „Test track map,“ Hamburg.

[14] W. Grega, „Hardware-in-the-loop simulation and its application in control education,“ *FIE'99 Frontiers in Education. 29th Annual Frontiers in Education Conference. Designing the Future of*

Science and Engineering Education. Conference Proceedings (IEEE Cat. No.99CH37011}, Bd. 2, pp. 12B6/7-12B612 vol.2, 1999.

[15] P. Sarhadi und S. Yousefpour, „State of the art: hardware in the loop modeling and simulation with its applications in design, development and implementation of system and control software," *International Journal of Dynamics and Control,* Bd. 3, pp. 470-479, 2015.

[16] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez und V. Koltun, „CARLA: An Open Urban Driving Simulator," *Proceedings of the 1st Annual Conference on Robot Learning,* Bd. 78, pp. 1-16, 2017.

[17] CARLA, „About page," [Online]. Available: https://carla.org/about/.

[18] H. Shimada, A. Yamaguchi, H. Takada und K. Sato, „Implementation and Evaluation of Local Dynamic Map in Safety Driving Systems," *Journal of Transportation Technologies,* Bd. 5, Nr. 2, January 2015.

[19] DLR, „Homepage," Deutsches Zentrum für Luft- und Raumfahrt. [Online].

[20] Urban Mobility Lab, „Urban Mobility Lab Homepage," [Online]. Available: http://urbanmobility.gnet.haw-hamburg.de/. [Zugriff am 21 September 2022].

[21] Institut für Klimaschutz, Energie und Mobilität e.V:, „Homepage," [Online]. Available: https://www.ikem.de/.

[22] OECON Products & Services, „Homepage," [Online]. Available: https://www.oecon-line.de/en/startseite-2/.

[23] consider it GmbH, „Homepage," [Online]. Available: https://consider-it.de/.

[24] Ubilabs GmbH, „Homepage," [Online]. Available: https://ubilabs.com/en.

[25] „ROS Homepage," [Online]. Available: https://www.ros.org/.

[26] U. Lauff, C. Störmer und D. Vijayaraghavan, „Entwicklung und Test verteilter Funktionen; Simulation und Virtualisierung von Fahrzeugsystemen," *Automoblil Elektronik,* 2017.

[27] dSPACE, „Company History," [Online]. Available: https://www.dspace.com/en/pub/home/company/companyhistory.cfm.

[28] dSPACE, „Cooperating Partners," [Online]. Available: https://www.dspace.com/en/pub/home/company/cooperations/cooperating_partners.cfm#.

[29] C. Anagnostopoulos, C. Koulamas, A. Lalos und C. Stylios, „Open-Source Integrated Simulation Framework for Cooperative Autonomous Vehicles," in *11th Mediterranean Conference on Embedded Computing (MECO)*, 2022.

[30] M. Shan, K. Narula, S. Worrall, Y. F. Wong, J. S. Berrio Perez, P. Gray und E. Nebot, „A Novel Probabilistic V2X Data Fusion Framework for Cooperative Perception," in *2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC)*, 2022.

[31] Epic Games Inc., „Hardware and Software Specifications," Epic Games Inc., [Online]. Available: https://docs.unrealengine.com/4.26/en-US/Basics/RecommendedSpecifications/. [Zugriff am 20 September 2022].

[32] „Recommended Hardware," Catalyst Softworks, 7 April 2020. [Online]. Available: https://unrealcommunity.wiki/recommended-hardware-x1p9qyg0. [Zugriff am 20 September 2022].

[33] CARLA, „Linux build - System Requirements," [Online]. Available: https://carla.readthedocs.io/en/0.9.13/build_linux/#system-requirements. [Zugriff am 20 September 2022].

[34] CARLA, „Linux Build - Software Requirements," [Online]. Available: https://carla.readthedocs.io/en/latest/build_linux/#software-requirements. [Zugriff am 21 September 2022].

[35] Canonical Ltd., „Package Search Results: clang-8," Canonical Ltd., 2022. [Online]. Available: https://packages.ubuntu.com/search?keywords=clang-8. [Zugriff am 21 September 2022].

[36] Canonical Ltd., „Package Search Results: clang-10," Canonical Ltd., 2022. [Online]. Available: https://packages.ubuntu.com/search?keywords=clang-10. [Zugriff am 21 September 2022].

[37] Canonical Ltd., „Upgrade Ubuntu desktop," Canonical Ltd., [Online]. Available: https://ubuntu.com/tutorials/upgrading-ubuntu-desktop#1-before-you-start. [Zugriff am 21 September 2022].

[38] CARLA, „Linux Build - Other Make Commands," [Online]. Available: https://carla.readthedocs.io/en/latest/build_linux/#other-make-commands. [Zugriff am 21 September 2022].

[39] M. Weltz, „Master Thesis," to be published.

[40] „ROS Installation on Ubuntu," [Online]. Available: http://wiki.ros.org/noetic/Installation/Ubuntu.

[41] „Create a workspace," [Online]. Available: http://wiki.ros.org/catkin/Tutorials/create_a_workspace.

[42] „ROS Parameter Server," [Online]. Available: http://wiki.ros.org/Parameter%20Server.

[43] A. C. Rüdenauer, „Entwicklung der Simulation eines autonomen Fahrzeugs mit zugehöriger Sensorik auf einer Beispielstrecke in CARLA," HAW Hamburg, Hamburg, 2022.

[44] CARLA, „Sensors References - GNSS Sensor," [Online]. Available: https://carla.readthedocs.io/en/0.9.13/ref_sensors/#gnss-sensor.

[45] OpenStreetMap Foundation, „OSM Export," OpenStreetMap Foundation, [Online]. Available: https://www.openstreetmap.org/export. [Zugriff am 22 September 2022].

[46] „netconvert," [Online]. Available: https://sumo.dlr.de/docs/netconvert.html.

[47] The MathWorks, Inc., „Export to CARLA," [Online]. Available: https://de.mathworks.com/help/roadrunner/ug/export-to-carla.html. [Zugriff am 23 September 2022].

[48] CARLA, „Generate maps with OpenStreetMap - Ingest into CARLA," [Online]. Available: https://carla.readthedocs.io/en/0.9.13/tuto_G_openstreetmap/#ingest-into-carla. [Zugriff am 22 September 2022].

[49] W. Schneider, „OSM Extract," 2022. [Online]. Available: https://extract.bbbike.org/. [Zugriff am 22 September 2022].

[50] „JOSM," [Online]. Available: https://josm.openstreetmap.de/. [Zugriff am 22 September 2022].

[51] Blender Foundation, „Blender Homepage," Blender Foundation, [Online]. Available: https://www.blender.org/. [Zugriff am 22 September 2022].

[52] Prochitecture, „Blender OSM Add-On," 12 January 2017. [Online]. Available: https://github.com/vvoovv/blender-osm. [Zugriff am 22 September 2022].

[53] The MathWorks, Inc., „Install and Activate RoadRunner," The MathWorks, Inc., [Online]. Available: https://de.mathworks.com/help/roadrunner/ug/install-and-activate-roadrunner.html. [Zugriff am 23 September 2022].

[54] The MathWorks, Inc., „Importing ASAM OpenDRIVE Files," [Online]. Available: https://de.mathworks.com/help/roadrunner/ug/importing-opendrive-files.html. [Zugriff am 23 September 2022].

[55] „Landesbetrieb Geoinformation und Vermessung - Hamburg," [Online]. Available: https://www.hamburg.de/bsw/landesbetrieb-geoinformation-und-vermessung/.

[56] I. Gog, S. Kalra, P. Schafhalter, M. A. Wright, J. E. Gonzales und I. Stoica, „Pylot: A modular platform for exploring latency-accuracy tradeoffs in autonomous vehicles," *2021 IEEE International Conference on Robotics and Automation (ICRA),* pp. 8806-8813, 2021.

[57] „SBG messages," [Online]. Available: http://wiki.ros.org/sbg_driver#SBG_custom_messages.

[58] „CARLA Simulation: Simulated Test Vehicle Controller With Provided Route," 2022.

[59] „CARLA Simulation: Simulated Test Vehicle Controller - Randomized Route," 2022.

[60] „CARLA Simulation: Displaying Traffic Signs Along Randomized Test Drive," 2022.

[61] ETSI, „Intelligent Transport systems; Vehicular Communications; Basic Set of Applications; Local Dynamic Map," ETSI, 2014.

[62] J. M. O'Kane, „Creating launch files," in *A Gentle Introduction to ROS*, Jason Matthew O'Kane, 2018, pp. 86-91.

[63] Bundesministerium für Digitales und Verkehr, „EDDY Project," [Online]. Available: https://bmdv.bund.de/SharedDocs/DE/Artikel/DG/mfund-projekte/eddy.html.

Declaration

I declare that this Bachelor Thesis has been completed by myself independently without outside help and only the defined sources and study aids were used.

Hamburg, ______________ ________________________

Martin Gochevski